

Giancarlo Zaccone, Md. Rezaul Karim,
Ahmed Menshawy

Deep Learning with TensorFlow

Take your machine learning knowledge to the next level
with the power of TensorFlow 1.x



Packt

Title Page

Deep Learning with TensorFlow

Take your machine learning knowledge to the next level with the power of
TensorFlow 1.x

Giancarlo Zaccone
Md. Rezaul Karim
Ahmed Menshawy



BIRMINGHAM - MUMBAI

Copyright

Deep Learning with TensorFlow

Copyright © 2017 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 authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be 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.

First published: April 2017

Production reference: 1200417

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78646-978-6

www.packtpub.com

Credits

Authors Giancarlo Zaccone Md. Rezaul Karim Ahmed Menshawy	Copy Editor Safis Editing
Reviewers Swapnil Ashok Jadhav Chetan Khatri	Project Coordinator Shweta H Birwatkar
Commissioning Editor Veena Pagare	Proofreader Safis Editing

Acquisition Editor Vinay Agrekar	Indexer Aishwarya Gangawane
Content Development Editor Amrita Norohna	Graphics Tania Dutta
Technical Editor Deepti Tuscano	Production Coordinator Nilesh Mohite

About the Authors

Giancarlo Zaccione has more than ten years of experience in managing research projects both in scientific and industrial areas. He worked as researcher at the C.N.R, the National Research Council, where he was involved in projects relating to parallel computing and scientific visualization.

Currently, he is a system and software engineer at a consulting company developing and maintaining software systems for space and defense applications.

He is author of the following Packt volumes: *Python Parallel Programming Cookbook* and *Getting Started with TensorFlow*.

You can follow him at <https://it.linkedin.com/in/giancarlozaccione>.

Md. Rezaul Karim has more than 8 years of experience in the area of research and development with a solid knowledge of algorithms and data structures, focusing C/C++, Java, Scala, R, and Python and big data technologies such as Spark, Kafka, DC/OS, Docker, Mesos, Hadoop, and MapReduce. His research interests include machine learning, deep learning, Semantic Web, big data, and bioinformatics. He is the author of the book titled *Large-Scale Machine Learning with Spark*, Packt Publishing.

He is a Software Engineer and Researcher currently working at the Insight Center for Data Analytics, Ireland. He is also a Ph.D. candidate at the National University of Ireland, Galway. He also holds a BS and an MS degree in Computer Engineering. Before joining the Insight Centre for Data Analytics, he had been working as a Lead Software Engineer with Samsung Electronics, where he worked with the distributed Samsung R&D centers across the world, including Korea, India, Vietnam, Turkey, and Bangladesh. Before that, he worked as a Research Assistant in the Database Lab at Kyung Hee University, Korea. He also worked as an R&D Engineer with BMTech21 Worldwide, Korea. Even before that, he worked as a Software Engineer with i2SoftTechnology, Dhaka, Bangladesh.

I would like to thank my parents (Mr. Razzaque and Mrs. Monoara) for their continuous encouragement and motivation throughout my life. I would also like to thank my wife (Saroar) and my kid (Shadman) for their never-ending support, which keeps me going. I would like to give special thanks to Ahmed Menshawy and Giancarlo Zaccione for authoring this book. Without their contributions, the writing would have been impossible. Overall, I would like to dedicate this book to my elder brother Md. Mamta Uddin (Manager, International Business, Biopharma Ltd., Bangladesh) for his endless contributions to my life.

Further, I would like to thank the acquisition, content development and technical editors of Packt Publishing (and others who were involved in this book title) for their sincere cooperation and coordination. Additionally, without the work of numerous researchers and deep learning practitioners who shared their expertise in publications, lectures, and source code, this book might

not exist at all! Finally, I appreciate the efforts of the TensorFlow community and all those who have contributed to APIs, whose work ultimately brought the deep learning to the masses.

Ahmed Menshawy is a Research Engineer at the Trinity College Dublin, Ireland. He has more than 5 years of working experience in the area of Machine Learning and Natural Language Processing (NLP). He holds an MSc in Advanced Computer Science. He started his Career as a Teaching Assistant at the Department of Computer Science, Helwan University, Cairo, Egypt. He taught several advanced ML and NLP courses such as Machine Learning, Image Processing, Linear Algebra, Probability and Statistics, Data structures, Essential Mathematics for Computer Science. Next, he joined as a research scientist at the Industrial research and development lab at IST Networks, based in Egypt. He was involved in implementing the state-of-the-art system for Arabic Text to Speech. Consequently, he was the main machine learning specialist in that company. Later on, he joined the Insight Centre for Data Analytics, the National University of Ireland at Galway as a Research Assistant working on building a Predictive Analytics Platform. Finally, he joined ADAPT Centre, Trinity College Dublin as a Research Engineer. His main role in ADAPT is to build prototypes and applications using ML and NLP techniques based on the research that is done within ADAPT.

I would like to thank my parents, my Wife Sara and daughter Asma for their support and patience during the book. Also I would like to sincerely thank Md. Rezaul Karim and Giancarlo Zaccone for authoring this book.

Further, I would like to thank the acquisition, content development and technical editors of Packt Publishing (and others who were involved in this book title) for their sincere cooperation and coordination. Additionally, without the work of numerous researchers and deep learning practitioners who shared their expertise in publications, lectures, and source code, this book might not exist at all! Finally, I appreciate the efforts of the TensorFlow community and all those who have contributed to APIs, whose work ultimately brought the machine learning to the masses.

About the Reviewers

Swapnil Ashok Jadhav is a Machine Learning and NLP enthusiast. He enjoys learning new Machine Learning and Deep Learning technologies and solving interesting data science problems and has around 3 years of working experience in these fields.

He is currently working at Haptik Infotech Pvt. Ltd. as a Machine Learning Scientist. Swapnil holds Masters degree in Information Security from NIT Warangal and Bachelors degree from VJTI Mumbai.

You can follow him at <https://www.linkedin.com/in/swapnil-jadhav-9448872a>.

Chetan Khatri is a data science researcher with having total of five years of experience in research and development. He works as a lead technology at Accionlabs India. Prior to that he worked with Nazara Games, where he lead data science practice as a principal big data engineer for Gaming and Telecom Business. He has worked with a leading data companies and a Big 4 companies, where he managed the data science practice platform and one of the Big 4 company's resources team.

He completed his master's degree in computer science and minor data science at KSKV Kachchh.

University, and was awarded as “Gold Medalist” by the Governor of Gujarat for his “University 1st Rank” achievements.

He contributes to society in various ways, including giving talks to sophomore students at universities and giving talks on the various fields of data science, machine learning, AI, IoT in academia and at various conferences. He has excellent correlative knowledge of both academic research and industry best practices. Hence, He always come forward to remove gap between Industry and Academia where he has good number of achievements. He was core co-author of various courses such as data science, IoT, machine learning/AI, distributed databases at PG/UG cariculla at university of Kachchh. Hence, university of Kachchh become first government university in Gujarat to introduce Python as a first programming language in Cariculla and India’s first government university to introduce data science, AI, IoT courses in Cariculla entire success story presented by Chetan at Pycon India 2016 conference. He is one of the founding members of PyKutch—A Python Community.

Currently, he is working on intelligent IoT devices with deep learning , reinforcement learning and distributed computing with various modern architectures. He is committer at Apache HBase and Spark HBase connector.

I would like to thank Prof. Devji Chhang, Head of the Computer Science, University of Kachchh, for routing me to the correct path and for his valuable guidance in the field of data science research.

I would also like to thanks Prof. Shweta Gorania for being the first to introduce genetic algorithm and neural networks.

Last but not least, I would like to thank my beloved family for their support.

www.PacktPub.com

For support files and downloads related to your book, please visit www.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.



<https://www.packtpub.com/mapt>

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Customer Feedback

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at <https://www.amazon.com/dp/1786469786>.

If you'd like to join our team of regular reviewers, you can e-mail us at customerreviews@packtpub.com. We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products!

Table of Contents

Preface

- What this book covers
- What you need for this book
- Who this book is for
- Conventions
- Reader feedback
- Customer support
- Downloading the example code
- Downloading the color images of this book
- Errata
- Piracy
- Questions

1. Getting Started with Deep Learning

- Introducing machine learning
 - Supervised learning
 - Unsupervised learning
 - Reinforcement learning
- What is deep learning?
 - How the human brain works
 - Deep learning history
 - Problems addressed
- Neural networks
 - The biological neuron
 - An artificial neuron
- How does an artificial neural network learn?
 - The backpropagation algorithm
 - Weights optimization
 - Stochastic gradient descent
- Neural network architectures
 - Multilayer perceptron
 - DNNs architectures
 - Convolutional Neural Networks
 - Restricted Boltzmann Machines
- Autoencoders
- Recurrent Neural Networks
- Deep learning framework comparisons
- Summary

2. First Look at TensorFlow

- General overview
 - What's new with TensorFlow 1.x?
 - How does it change the way people use it?
 - Installing and getting started with TensorFlow
 - Installing TensorFlow on Linux

Which TensorFlow to install on your platform?

Requirements for running TensorFlow with GPU from NVIDIA

Step 1: Install NVIDIA CUDA

Step 2: Installing NVIDIA cuDNN v5.1+

Step 3: GPU card with CUDA compute capability 3.0+

Step 4: Installing the libcupti-dev library

Step 5: Installing Python (or Python3)

Step 6: Installing and upgrading PIP (or PIP3)

Step 7: Installing TensorFlow

How to install TensorFlow

Installing TensorFlow with native pip

Installing with virtualenv

Installing TensorFlow on Windows

Installation from source

Install on Windows

Test your TensorFlow installation

Computational graphs

Why a computational graph?

Neural networks as computational graphs

The programming model

Data model

Rank

Shape

Data types

Variables

Fetches

Feeds

TensorBoard

How does TensorBoard work?

Implementing a single input neuron

Source code for the single input neuron

Migrating to TensorFlow 1.x

How to upgrade using the script

Limitations

Upgrading code manually

Variables

Summary functions

Simplified mathematical variants

Miscellaneous changes

Summary

3. Using TensorFlow on a Feed-Forward Neural Network

Introducing feed-forward neural networks

Feed-forward and backpropagation

Weights and biases

Transfer functions

Classification of handwritten digits

Exploring the MNIST dataset

Softmax classifier

 Visualization

How to save and restore a TensorFlow model

 Saving a model

 Restoring a model

 Softmax source code

 Softmax loader source code

Implementing a five-layer neural network

 Visualization

 Five-layer neural network source code

ReLU classifier

 Visualization

 Source code for the ReLU classifier

Dropout optimization

 Visualization

 Source code for dropout optimization

Summary

4. TensorFlow on a Convolutional Neural Network

Introducing CNNs

CNN architecture

 A model for CNNs - LeNet

Building your first CNN

 Source code for a handwritten classifier

Emotion recognition with CNNs

 Source code for emotion classifier

 Testing the model on your own image

 Source code

Summary

5. Optimizing TensorFlow Autoencoders

Introducing autoencoders

Implementing an autoencoder

 Source code for the autoencoder

Improving autoencoder robustness

Building a denoising autoencoder

 Source code for the denoising autoencoder

Convolutional autoencoders

 Encoder

 Decoder

 Source code for convolutional autoencoder

Summary

6. Recurrent Neural Networks

RNNs basic concepts

RNNs at work

Unfolding an RNN

The vanishing gradient problem

LSTM networks

An image classifier with RNNs

Source code for RNN image classifier

Bidirectional RNNs

Source code for the bidirectional RNN

Text prediction

Dataset

Perplexity

PTB model

Running the example

Summary

7. GPU Computing

GPGPU computing

GPGPU history

The CUDA architecture

GPU programming model

TensorFlow GPU set up

Update TensorFlow

TensorFlow GPU management

Programming example

Source code for GPU computation

GPU memory management

Assigning a single GPU on a multi-GPU system

Source code for GPU with soft placement

Using multiple GPUs

Source code for multiple GPUs management

Summary

8. Advanced TensorFlow Programming

Introducing Keras

Installation

Building deep learning models

Sentiment classification of movie reviews

Source code for the Keras movie classifier

Adding a convolutional layer

Source code for movie classifier with convolutional layer

Pretty Tensor

Chaining layers

Normal mode

Sequential mode

Branch and join

Digit classifier

Source code for digit classifier

TFLearn

TFLearn installation

Titanic survival predictor

Source code for titanic classifier

Summary

9. Advanced Multimedia Programming with TensorFlow

Introduction to multimedia analysis

Deep learning for Scalable Object Detection

Bottlenecks

Using the retrained model

Accelerated Linear Algebra

Key strengths of TensorFlow

Just-in-time compilation via XLA

JIT compilation

Existence and advantages of XLA

Under the hood working of XLA

Still experimental

Supported platforms

More experimental material

TensorFlow and Keras

What is Keras?

Effects of having Keras on board

Video question answering system

Not runnable code!

Deep learning on Android

TensorFlow demo examples

Getting started with Android

Architecture requirements

Prebuilt APK

Running the demo

Building with Android studio

Going deeper - Building with Bazel

Summary

10. Reinforcement Learning

Basic concepts of Reinforcement Learning

Q-learning algorithm

Introducing the OpenAI Gym framework

FrozenLake-v0 implementation problem

Source code for the FrozenLake-v0 problem

Q-learning with TensorFlow

Source code for the Q-learning neural network

Summary

Preface

Machine learning is concerned with algorithms that transform raw data into information into actionable intelligence. This fact makes machine learning well suited to the predictive analytics of big data. Without machine learning, therefore, it would be nearly impossible to keep up with these massive streams of information altogether. On the other hand, the deep learning is a branch of machine learning algorithms based on learning multiple levels of representation. Just in the last few years have been developed powerful deep learning algorithms to recognize images, natural language processing and perform a myriad of other complex tasks. A deep learning algorithm is nothing more than the implementation of a complex neural network so that it can learn through the analysis of large amounts of data. This book introduces the core concepts of deep learning using the latest version of TensorFlow. This is Google's open-source framework for mathematical, machine learning and deep learning capabilities released in 2011. After that, TensorFlow has achieved wide adoption from academia and research to industry and following that recently the most stable version 1.0 has been released with a unified API. TensorFlow provides the flexibility needed to implement and research cutting-edge architectures while allowing users to focus on the structure of their models as opposed to mathematical details. Readers will learn deep learning programming techniques with the hands-on model building, data collection and transformation and even more!

Enjoy reading!

What this book covers

[Chapter 1](#), *Getting Started with TensorFlow*, covers some basic concepts that will be found in all the subsequent chapters. We'll introduce machine learning and deep learning architectures. Finally, we'll introduce deep learning architectures, the so-called Deep Neural Networks: these are distinguished from the more commonplace single-hidden-layer neural networks by their depth; that is, the number of node layers through which data passes in a multistep process of pattern recognition. We will provide a comparative analysis of deep learning architectures with a chart summarizing all the neural networks from where most of the deep learning algorithm evolved.

[Chapter 2](#), *First Look at TensorFlow*, will cover the main features and capabilities of TensorFlow 1.x: getting started with computation graph, data model, programming model and TensorBoard. In the last part of the chapter, we'll see TensorFlow in action by implementing a Single Input Neuron. Finally, it will show how to upgrade from TensorFlow 0.x to TensorFlow 1.x.

[Chapter 3](#), *Using TensorFlow on a Feed-Forward Neural Network*, provides a detailed introduction of feed-forward neural networks. The chapter will be also very practical, implementing a lot of application examples using this fundamental architecture.

[Chapter 4](#), *TensorFlow on a Convolutional Neural Network*, introduces the CNNs networks that are the basic blocks of a deep learning-based image classifier. We'll develop two examples of CNN networks; the first is the classic MNIST digit classification problem, while the purpose for the second is to train a network on a series of facial images to classify their emotional stretch.

[Chapter 5](#), *Optimizing TensorFlow Autoencoders*, presents autoencoders networks that are designed and trained for transforming an input pattern so that, in the presence of a degraded or incomplete version of an input pattern, it is possible to obtain the original pattern. In the chapter, we'll see autoencoders in action with some application examples.

[Chapter 6](#), *Recurrent Neural Networks*, explains this fundamental architecture designed to handle data that comes in different lengths, that is very popular for various natural language processing tasks. Text processing and image classification problems will be implemented in the course if this chapter.

[Chapter 7](#), *GPU Computing*, shows the TensorFlow facilities for *GPU* computing. In this chapter, we'll explore some techniques to handle GPU using TensorFlow.

[Chapter 8](#), *Advanced TensorFlow Programming*, gives an overview of the following TensorFlow-based libraries: Keras, Pretty Tensor, and TFLearn. For each library, we'll describe the main features with an application example.

[Chapter 9](#), *Advanced Multimedia Programming with TensorFlow*, covers some advanced and emerging aspects of multimedia programming using TensorFlow. Deep neural networks for scalable object detection and deep learning on Android using TensorFlow with an example with the code will be

discussed. The Accelerated Linear Algebra (XLA) and Keras will be discussed with examples to make the discussion more concrete.

[Chapter 10](#), *Reinforcement Learning*, covers the basic concepts of RL. We will experience the Q-learning algorithm that is one of the most popular reinforcement learning algorithms. Furthermore, we'll introduce the OpenAI gym framework that is a TensorFlow compatible, toolkit for developing and comparing reinforcement learning algorithms.

What you need for this book

All the examples have been implemented using Python version 2.7 (and 3.5) on an Ubuntu Linux 64 bit including the TensorFlow library version 1.0.1. However, all the source codes that are shown in the book are Python 2.7 compatible. Further, source codes for Python 3.5 compatible can be downloaded from the Packt repository. Source codes for Python 3.5+ compatible can be downloaded from the Packt repository.

You will also need the following Python modules (preferably the latest version):

- Pip
- Bazel
- Matplotlib
- NumPy
- Pandas
- mnist_data

For chapters 8, 9 and 10, you will need the following frameworks too:

- Keras
- XLA
- Pretty Tensor
- TFLearn
- OpenAI gym

Most importantly, GPU-enabled version of TensorFlow has several requirements such as 64-bit Linux, Python 2.7 (or 3.3+ for Python 3), NVIDIA CUDA® 7.5 (CUDA 8.0 required for Pascal GPUs) and NVIDIA cuDNN v4.0 (minimum) or v5.1 (recommended). More specifically, the current implementation of TensorFlow supports GPU computing with NVIDIA toolkits, drivers and software only.

Who this book is for

This book is dedicated to developers, data analysts, or deep learning enthusiasts who do not have much background with complex numerical computations but want to know what deep learning is. The book majorly appeals to beginners who are looking for a quick guide to gain some hands-on experience with deep learning. A rudimentary level of programming in one language is assumed as is a basic familiarity with computer science techniques and technologies including basic awareness of computer hardware and algorithms. Some competence in mathematics is needed to the level of elementary linear algebra and calculus.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "To save a model, we use the `Saver()` class."

A block of code is set as follows:

```
saver = tf.train.Saver()  
save_path = saver.save(sess, "softmax_mnist")  
print("Model saved to %s" % save_path)
```

Any command-line input or output is written as follows:

```
$ sudo apt-get install python-pip python-dev
```

Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Click on the GRAPH tab you'll be able to see the *computation graph* with the *auxiliary nodes*."

 *Warnings or important notes appear in a box like this.*

 *Tips and tricks appear like this.*

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book-what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

Feedback from our readers is always welcome. Let us know what you think about this book-what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail feedback@packtpub.com, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the SUPPORT tab at the top.
3. Click on Code Downloads & Errata.
4. Enter the name of the book in the Search box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on Code Download.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Deep-Learning-with-TensorFlow>. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from https://www.packtpub.com/sites/default/files/downloads/DeepLearningwithTensorFlow_ColorImages.pdf.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the code-we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the Errata Submission Form link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the book in the search field. The required information will appear under the Errata section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

Getting Started with Deep Learning

In this chapter, we will discuss about some basic concepts of deep learning and their related architectures that will be found in all the subsequent chapters of this book. We'll start with a brief definition of machine learning, whose techniques allow the analysis of large amounts of data to automatically extract information and to make predictions about subsequent new data. Then we'll move onto deep learning, which is a branch of machine learning based on a set of algorithms that attempt to model high-level abstractions in data.

Finally, we'll introduce deep learning architectures, the so-called **Deep Neural Networks (DNNs)**--these are distinguished from the more commonplace single hidden layer neural networks by their depth; that is, the number of node layers through which data passes in a multistep process of pattern recognition. we will provide a chart summarizing all the neural networks from where most of the deep learning algorithm evolved.

In the final part of the chapter, we'll briefly examine and compare some deep learning frameworks across various features, such as the native language of the framework, multi-GPU support, and aspects of usability.

This chapter covers the following topics:

- Introducing machine learning
- What is deep learning?
- Neural networks
- How does an artificial neural network learn?
- Neural network architectures
- DNNs architectures
- Deep learning framework comparison

Introducing machine learning

Machine learning is a computer science research area that deals with methods to identify and implement systems and algorithms by which a computer can learn, based on the examples given in the input. The challenge of machine learning is to allow a computer to learn how to automatically recognize complex patterns and make decisions that are as smart as possible. The entire learning process requires a dataset as follows:

- **Training set:** This is the knowledge base used to train the machine learning algorithm. During this phase, the parameters of the machine learning model (hyperparameters) can be tuned according to the performance obtained.
- **Testing set:** This is used only for evaluating the performance of the model on unseen data.

Learning theory uses mathematical tools that are derived from probability theory and information theory. This allows you to assess the optimality of some methods over others.

There are basically three learning paradigms that will be briefly discussed:

- Supervised learning
- Unsupervised learning
- Learning with reinforcement

Let's take a look at them.

Supervised learning

Supervised learning is the automatic learning task simpler and better known. It is based on a number of preclassified examples, in which, namely, is known a prior the category to which each of the inputs used as examples should belong. In this case, the crucial issue is the problem of generalization. After the analysis of a sample (often small) of examples, the system should produce a model that should work well for all possible inputs.

The set consists of labeled data, that is, objects and their associated classes. This set of labeled examples, therefore, constitutes the training set.

Most of the supervised learning algorithms share one characteristic: the training is performed by the minimization of a particular loss or cost function, representing the output error provided by the system with respect to the desired possible output, because the training set provides us with what must be the desired output.

The system then changes its internal editable parameters, the weights, to minimize this error function. The goodness of the model is evaluated, providing a second set of labeled examples (the test set), evaluating the percentage of correctly classified examples and the percentage of misclassified examples.

The supervised learning context includes the classifiers, but also the learning of functions that predict numeric values. This task is the **regression**. In a regression problem, the training set is a pair formed by an object and the associated numeric value. There are several supervised learning algorithms that have been developed for classification and regression. These can be grouped into the formula used to represent the classifier or the learned predictor, among all, decision trees, decision rules, neural networks and Bayesian networks.

Unsupervised learning

In unsupervised learning, a set of inputs is supplied to the system during the training phase which, however, contrary to the case supervised learning, is not labeled with the related belonging class. This type of learning is important because in the human brain it is probably far more common than supervised learning.

The only objects in the domain of learning models, in this case, are the observed data inputs, which often is assumed to be independent samples of an unknown underlying probability distribution.

Unsupervised learning algorithms are used particularly used in clustering problems, in which given a collection of objects, we want to be able to understand and show their relationships. A standard approach is to define a similarity measure between two objects, and then look for any cluster of objects that are more similar to each other, compared to the objects in the other clusters.

Reinforcement learning

Reinforcement learning is an artificial intelligence approach that emphasizes the learning of the system through its interactions with the environment. With reinforcement learning, the system adapts its parameters based on feedback received from the environment, which then provides feedback on the decisions made. For example, a system that models a chess player who uses the result of the preceding steps to improve their performance is a system that learns with reinforcement. Current research on learning with reinforcement is highly interdisciplinary, and includes researchers specializing in genetic algorithms, neural networks, psychology, and control engineering.

The following figure summarizes the three types of learning, with the related problems to address:

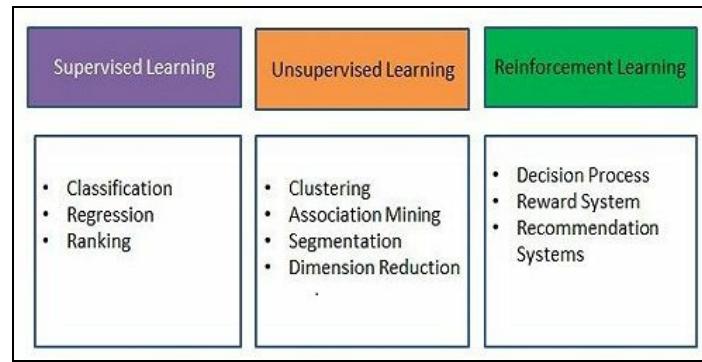


Figure 1: Types of learning and related problems

What is deep learning?

Deep learning is a machine learning research area that is based on a particular type of learning mechanism. It is characterized by the effort to create a learning model at several levels, in which the most profound levels take as input the outputs of previous levels, transforming them and always abstracting more. This insight on the levels of learning is inspired by the way the brain processes information and learns, responding to external stimuli.

Each learning level corresponds, hypothetically, to one of the different areas which make up the cerebral cortex.

How the human brain works

The visual cortex, which is intended to solve image recognition problems, shows a sequence of sectors placed in a hierarchy. Each of these areas receives an input representation, by means of flow signals that connect it to other sectors.

Each level of this hierarchy represents a different level of abstraction, with the most abstract features defined in terms of those of the lower level. At a time when the brain receives an input image, the processing goes through various phases, for example, detection of the edges or the perception of forms (from those primitive to those gradually more and more complex).

As the brain learns by trial and activates new neurons by learning from the experience, even in deep learning architectures, the extraction stages or layers are changed based on the information received at the input.

The scheme, on the next page shows what has been said in the case of an *image classification system*, each block gradually extracts the features of the input image, going on to process data already preprocessed from the previous blocks, extracting features of the image that are increasingly abstract, and thus building the hierarchical representation of data that comes with on deep learning based system.

More precisely, it builds the layers as follows along with the figure representation:

- **Layer 1:** The system starts identifying the dark and light pixels
- **Layer 2:** The system identifies edges and shapes
- **Layer 3:** The system learns more complex shapes and objects
- **Layer 4:** The system learns which objects define a human face

Here is the visual representation of the process:

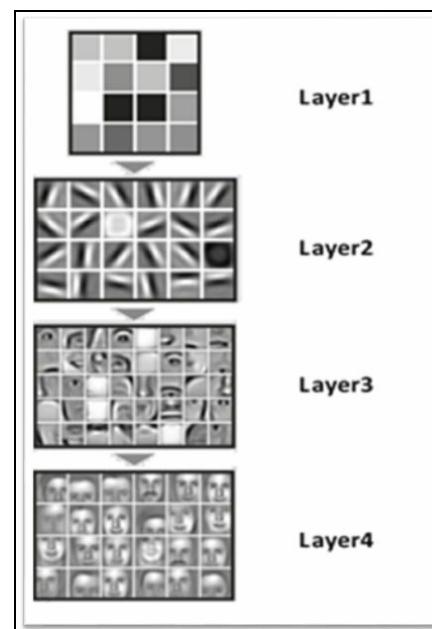
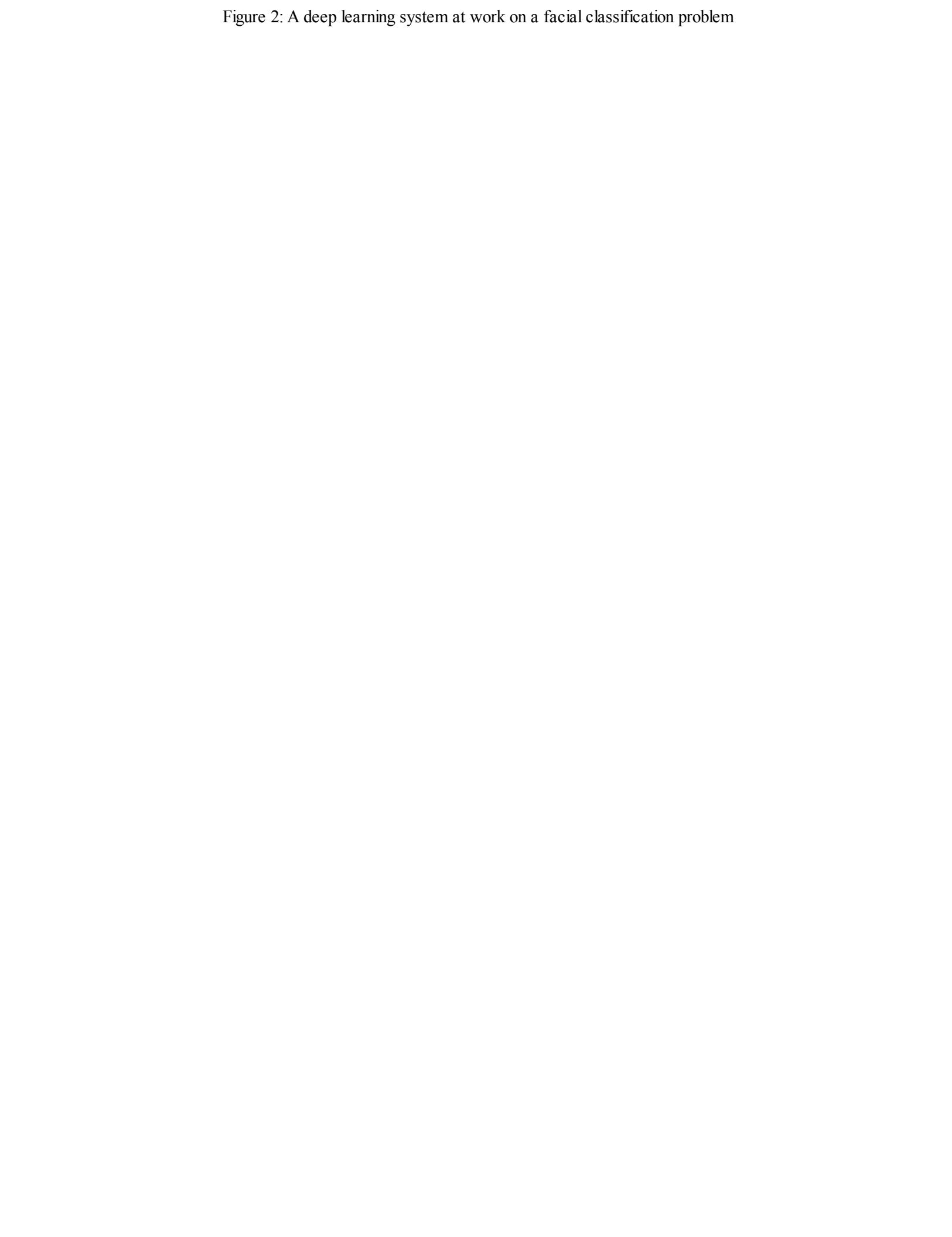


Figure 2: A deep learning system at work on a facial classification problem



Deep learning history

The development of deep learning consequently occurred parallel to the study of artificial intelligence, and especially neural networks. After beginning in the 50 s, it is mainly in the 80s that this area grew, thanks to Geoff Hinton and machine learning specialists who collaborated with him. In those years, computer technology was not sufficiently advanced to allow a real improvement in this direction, so we had to wait until the present day to see, thanks to the availability of data and the computing power, even more significant developments.

Problems addressed

As for the areas of application, deep learning is employed in the development of *speech recognition* systems, in the *search patterns*, and especially, in the *image recognition*, thanks to its learning characteristics for levels, which enable it to focus, step by step, on the various areas of an image to be processed and classified.

Neural networks

Artificial Neural Networks (ANNs) are one of the main tools that take advantage of the concept of deep learning. They are an abstract representation of our nervous system, which contains a collection of neurons that communicate with each other through connections called **axons**. The first artificial neuron model was proposed in 1943 by McCulloch and Pitts in terms of a computational model of nervous activity. This model was followed by another, proposed by John von Neumann, Marvin Minsky, Frank Rosenblatt (the so-called perceptron), and many others.

The biological neuron

As you can see in the following figure, a biological neuron is composed of the following:

- A cell *body* or *soma*
- One or more *dendrites*, whose responsibility is to receive signals from other neurons
- An *axon*, which in turn conveys the signals generated by the same neuron to the other connected neurons

This is what a biological neuron model looks like:

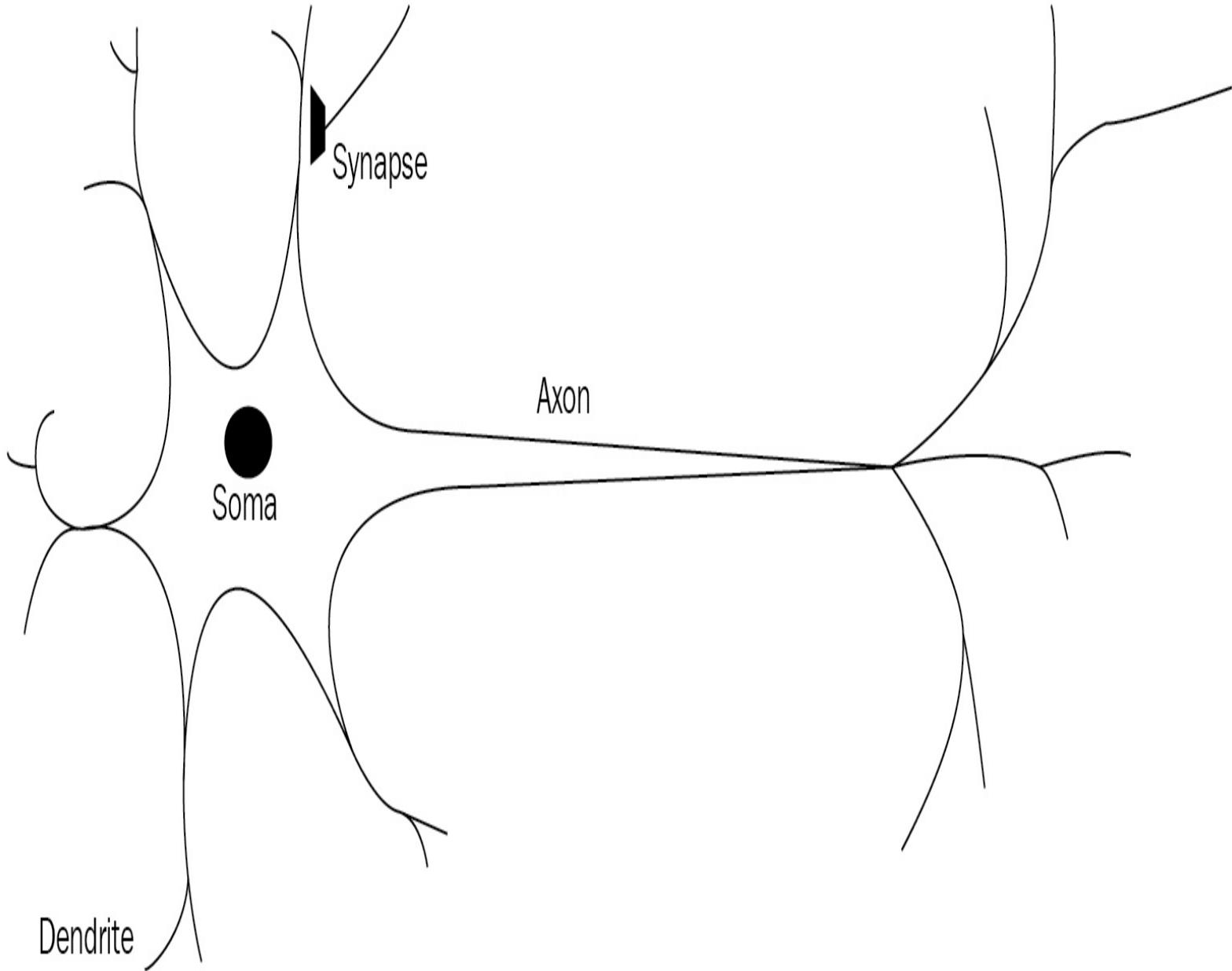


Figure 3: Biological neuron model

The neuron activity is in the alternation of sending the signal (active state) and rest/reception of

signals from other neurons (inactive state).

The transition from one phase to another is caused by the external stimuli represented by signals that are picked up by the dendrites. Each signal has an excitatory or inhibitory effect, conceptually represented by a weight associated with the stimulus. The neuron in an idle state accumulates all the signals received until they have reached a certain activation threshold.

An artificial neuron

Similar to the biological one, the artificial neuron consists of the following:

- One or more incoming connections, with the task of collecting numerical signals from other neurons; each connection is assigned a weight that will be used to consider each signal sent
- One or more output connections that carry the signal for the other neurons
- An activation function determines the numerical value of the output signal, on the basis of the signals received from the input connections with other neurons, and suitably collected from the weights associated with each picked-up signal and the activation threshold of the neuron itself

The following figure represents the artificial neuron:

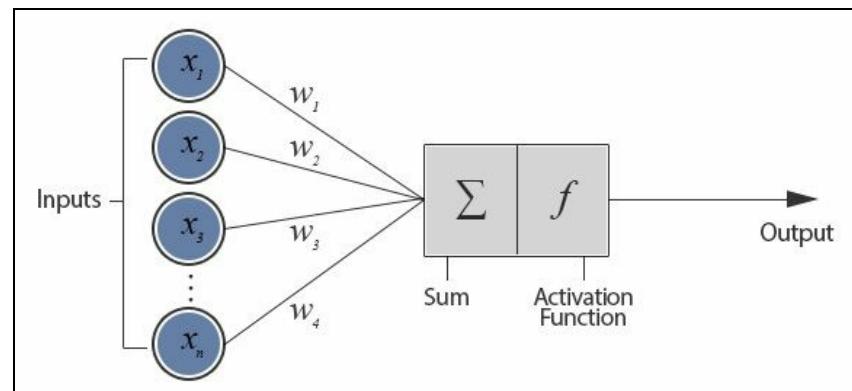


Figure 4: Artificial neuron model

The output, that is, the signal whereby the neuron transmits its activity outside, is calculated by applying the activation function, also called the transfer function, to the weighted sum of the inputs. These functions have a dynamic between -1 and 1, or between 0 and 1.

There is a set of activation functions that differs in complexity and output:

- **Step function:** This fixes the threshold value x (for example, $x=10$). The function will return 0 or 1 if the mathematical sum of the inputs is at, above, or below the threshold value.
- **Linear combination:** Instead of managing a threshold value, the weighted sum of the input values is subtracted from a default value; we will have a binary outcome, but it will be expressed by a positive ($+b$) or negative ($-b$) output of the subtraction.
- **Sigmoid:** This produces a sigmoid curve, a curve having an S trend. Often, the sigmoid function refers to a special case of the logistic function.

From the simplest forms, used in the prototyping of the first artificial neurons, we then move on to more complex ones that allow greater characterization of the functioning of the neuron. The following are just a few:

- Hyperbolic tangent function
- Radial basis function

- Conic section function
- Softmax function

It should be recalled that the network, and then the weights in the activation functions, will then be trained. As the selection of the activation function is an important task in the implementation of the network architecture, studies indicate marginal differences in terms of output quality if the training phase is carried out properly.

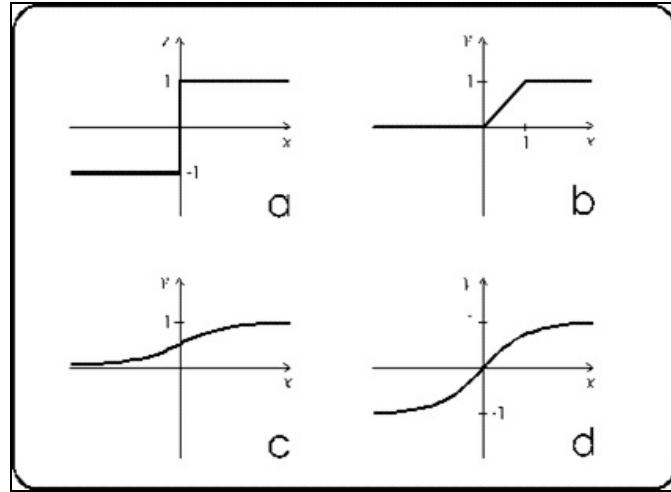


Figure 5: Most used transfer functions

In the preceding figure, the functions are labeled as follows:

- **a:** Step function
- **b:** Linear function
- **c:** Computed sigmoid function with values between 0 and 1
- **d:** Sigmoid function with computed values between -1 and 1

How does an artificial neural network learn?

The learning process of a neural network is configured as an iterative process of optimization of the weights, and is therefore of the supervised type. The weights are modified based on the network performance on a set of examples belonging to the training set, where the category they belong to is known. The aim is to minimize a loss function, which indicates the degree to which the behavior of the network deviates from the desired one. The performance of the network is then verified on a test set consisting of objects (for example, images in a image classification problem) other than those of the training set.

The backpropagation algorithm

A supervised learning algorithm used is the backpropagation algorithm.

The basic steps of the training procedure are as follows:

1. Initialize the net with random weights.
2. For all training cases:
 - **Forward pass:** Calculates the error committed by the net, the difference between the desired output and the actual output.
 - **Backward pass:** For all layers, starting with the output layer, back to the input layer.
3. Show the network layer output with correct input (error function).
4. Adapt weights in the current layer to minimize the error function. This is the backpropagation's optimization step. The training process ends when the error on the validation set begins to increase, because this could mark the beginning of a phase of over-fitting of the network, that is, the phase in which the network tends to interpolate the training data at the expense of generalization ability.

Weights optimization

The availability of efficient algorithms to weights optimization, therefore, constitutes an essential tool for the construction of neural networks. The problem can be solved with an iterative numerical technique called **gradient descent (GD)**.

This technique works according to the following algorithm:

1. Some initial values for the parameters of the model are chosen randomly.
2. Compute the gradient G of the error function with respect to each parameter of the model.
3. Change the model's parameters so that they move in the direction of decreasing the error, that is, in the direction of $-G$.
4. Repeat steps 2 and 3 until the value of G approaches zero.

i In mathematics, the gradient of a scalar field is a real-valued function of several variables, then defined in a region of a space in two, three, or more dimensions. The gradient of a function is defined as the vector that has Cartesian components for the partial derivatives of the function. The gradient represents the direction of maximum increment of a function of n variables: $f(x_1, x_2, \dots, x_n)$. The gradient is then a vector quantity that indicates a physical quantity as a function of its various different parameters.

The gradient G of the error function E provides the direction in which the error function with the current values has the steeper slope, so to decrease E , we have to make some small steps in the opposite direction, $-G$ (see the following figures).

By repeating this operation several times in an iterative manner, we move in the direction in which the gradient G of the function E is minimal (see the following figure):

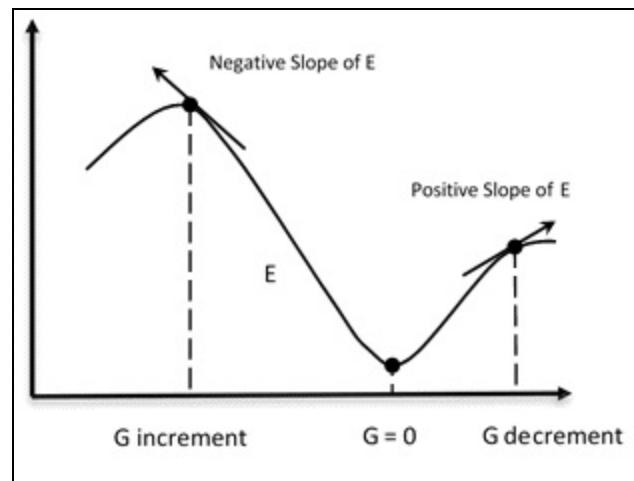


Figure 6: Gradient descent procedure

As you can see, we move in the direction in which the gradient G of the function E is minimal.

Stochastic gradient descent

In GD optimization, we compute the cost gradient based on the complete training set; hence, we sometimes also call it batch GD. In the case of very large datasets, using GD can be quite costly, since we are only taking a single step for one pass over the training set. Thus, the larger the training set, the slower our algorithm updates the weights and the longer it may take until it converges to the global cost minimum.

An alternative approach and the fastest of gradient descent, and for this reason, used in DNNs, is the **Stochastic Gradient Descent (SGD)**.

In SGD, we use only one training sample from the training set to do the update for a parameter in a particular iteration. Here, the term stochastic comes from the fact that the gradient based on a single training sample is a stochastic approximation of the true cost gradient.

Due to its stochastic nature, the path toward the global cost minimum is not direct, as in GD, but may zigzag if we are visualizing the cost surface in a 2D space (see the following figure, *(b) Stochastic Gradient Descent - SGD*).

We can make a comparison between these optimization procedures, showing the next figure, the gradient descent (see the following figure, *(a) Gradient Descent - GD*) assures that each update in the weights is done in the right direction--the one that minimizes the cost function. With the growth of datasets' size, and more complex computations in each step, SGD came to be preferred in these cases. Here, updates to the weights are done as each sample is processed and, as such, subsequent calculations already use improved weights. Nonetheless, this very reason leads to it incurring some misdirection in minimizing the error function:

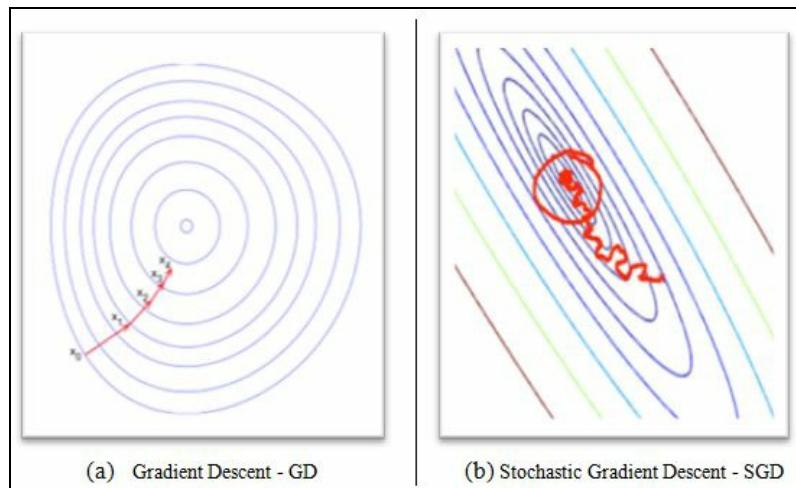


Figure 7: GD versus SDG

Neural network architectures

The way to connect the nodes, the number of layers present, that is, the levels of nodes between input and output, and the number of neurons per layer, defines the architecture of a neural network. There are various types of architecture in neural networks, but this book will focus mainly on two large architectural families.

Multilayer perceptron

In multilayer networks, one can identify the artificial neurons of layers such that:

- Each neuron is connected with all those of the next layer
- There are no connections between neurons belonging to the same layer
- There are no connections between neurons belonging to non-adjacent layers
- The number of layers and of neurons per layer depends on the problem to be solved

The input and output layers define inputs and outputs; there are hidden layers, whose complexity realizes different behaviors of the network. Finally, the connections between neurons are represented by as many matrices are the pairs of adjacent layers. Each array contains the weights of the connections between the pairs of nodes of two adjacent layers. The feed-forward networks are networks with no loops within the layers.

Following is the graphical representation of multilayer perceptron architecture:

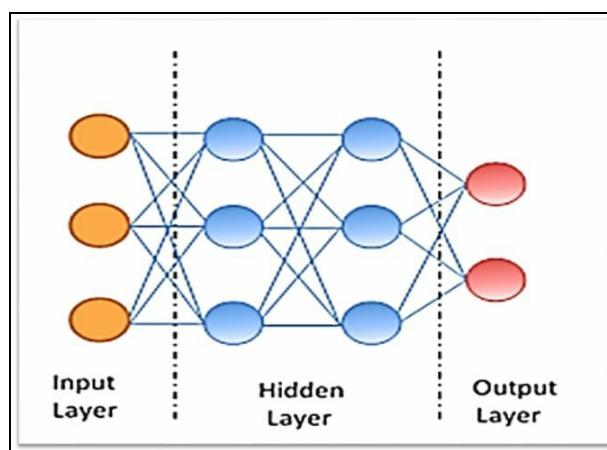


Figure 8: A multilayer perceptron architecture

DNNs architectures

Deep Neural Networks (DNNs) are artificial neural networks strongly oriented to deep learning. Where normal procedures of analysis are inapplicable due to the complexity of the data to be processed, such networks are an excellent modeling tool. DNNs are neural networks, very similar to those we have discussed, but they must implement a more complex model (a great number of neurons, hidden layers, and connections), although they follow the learning principles that apply to all machine learning problems (that is, supervised learning).

As they are built, the DNNs work in parallel, so they are able to treat a lot of data. They are a sophisticated statistical system, equipped with a good immunity to errors.

Unlike algorithmic systems where you can examine the output generation step by step, in neural networks, you can also have very reliable results, but sometimes without the ability to understand the reasons for those results. There are no theorems to generate optimal neural networks--the likelihood of getting a good network is all in the hands of its creator, who must be familiar with statistical concepts, and particular attention must be given to the choice of predictor variables.



For brief cheat sheet on the different neural network architecture and their related publications, refer to the website of Asimov Institute at <http://www.asimovinstitute.org/neural-network-zoo/>.

Finally, we observe that, in order to be productive, the DNNs require training that properly tunes the weights. Training can take a long time if the data to be examined and the variables involved are high, as is often the case when you want optimal results. This section introduces the deep learning architectures that we will cover during the course of this book.

Convolutional Neural Networks

Convolutional Neural Networks (CNNs) has been designed specifically for image recognition. Each image used in learning is divided into compact topological portions, each of which will be processed by filters to search for particular patterns. Formally, each image is represented as a three-dimensional matrix of pixels (width, height, and color), and every sub-portion is put on convolution with the filter set. In other words, scrolling each filter along the image computes the inner product of the same filter and input. This procedure produces a set of feature maps (activation maps) for the various filters. By superimposing the various feature maps of the same portion of the image, we get an output volume. This type of layer is called a **convolutional layer**.

The following figure shows a typical CNN architecture:

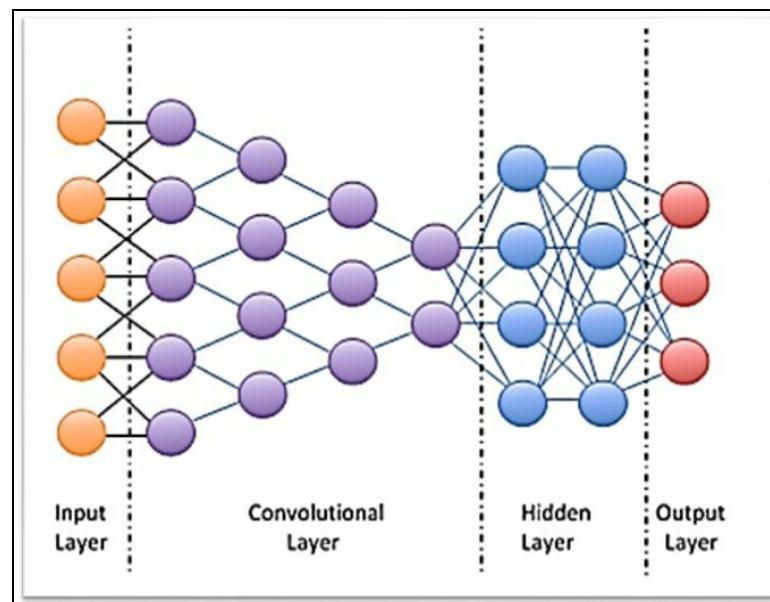


Figure 9: Convolutional neural network architecture

Restricted Boltzmann Machines

A **Restricted Boltzmann Machine (RBM)** consists of a visible and a hidden layer of nodes, but without visible-visible connections and hidden-hidden by the term restricted. These restrictions allow more efficient network training (training that can be supervised or unsupervised).

This type of neural network can represent with few size of the network a large number of features of the inputs; in fact, the n hidden nodes can represent up to $2n$ features. The network can be trained to respond to a single question (yes/no), up until (again, in binary terms) a total of $2n$ questions.

The architecture of the RBM is as follows, with neurons arranged according to a symmetrical bipartite graph:

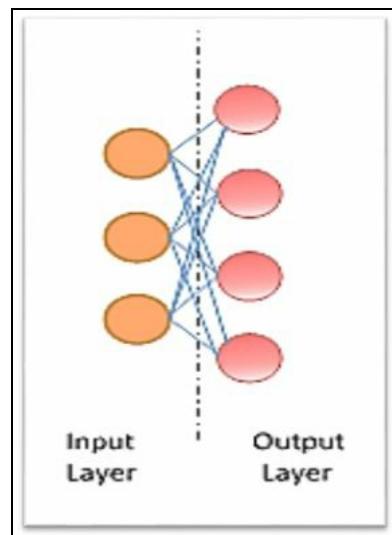


Figure 10: Restricted Boltzmann Machine architecture

Autoencoders

Stacked autoencoders are DNNs that are typically used for data compression. Their particular hourglass structure clearly shows the first part of the process, where the input data is compressed, up to the so-called bottleneck, from which the decompression starts.

The output is then an approximation of the input. These networks are not supervised in the pretraining (compression) phase, and the fine-tuning (decompression) phase is supervised:

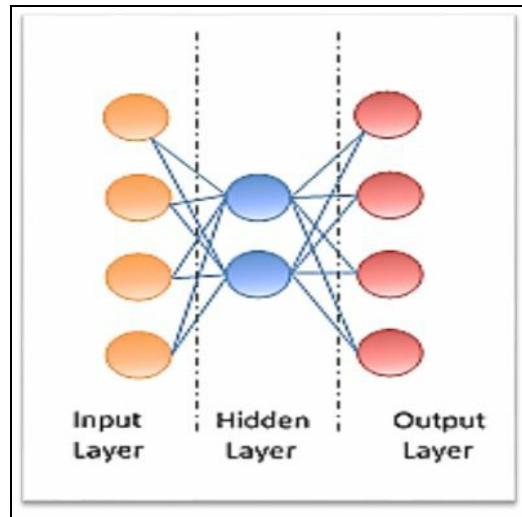


Figure 11: Stack autoencoder architecture

Recurrent Neural Networks

The fundamental feature of a **Recurrent Neural Network (RNN)** is that the network contains at least one feedback connection, so the activations can flow around in a loop. It enables the networks to do temporal processing and learn sequences, for example, perform sequence recognition/reproduction or temporal association/prediction. RNN architectures can have many different forms. One common type consists of a standard **multilayer perceptron (MLP)** plus added loops. These can exploit the powerful non-linear mapping capabilities of the MLP, and also have some form of memory. Others have more uniform structures, potentially with every neuron connected to all the others, and may also have stochastic activation functions. For simple architectures and deterministic activation functions, learning can be achieved using similar gradient descent procedures to those leading to the backpropagation algorithm for feed-forward networks.

The following figure shows a few of the most important types and features of RNNs:

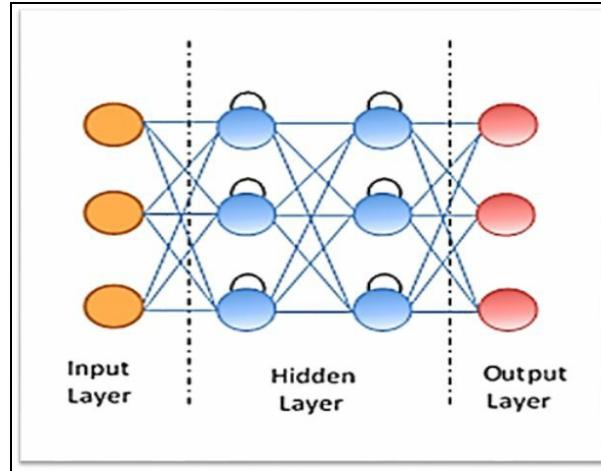


Figure 12: Recurrent Neural Network architecture

Deep learning framework comparisons

In this section, we will look at some of the most popular frameworks regarding deep learning. In short, almost all libraries provide the possibility of using the graphics processor to speed up the learning process, and are released under an open license and are the result of implementation by university research groups. Before starting the comparison, refer figure 13 which is one of the most complete charts of neural networks till date. If you see the URL and related papers, you will find that the idea of neural networks is pretty old and the software frameworks that we are going to compare below also adapts similar architecture during their framework development:

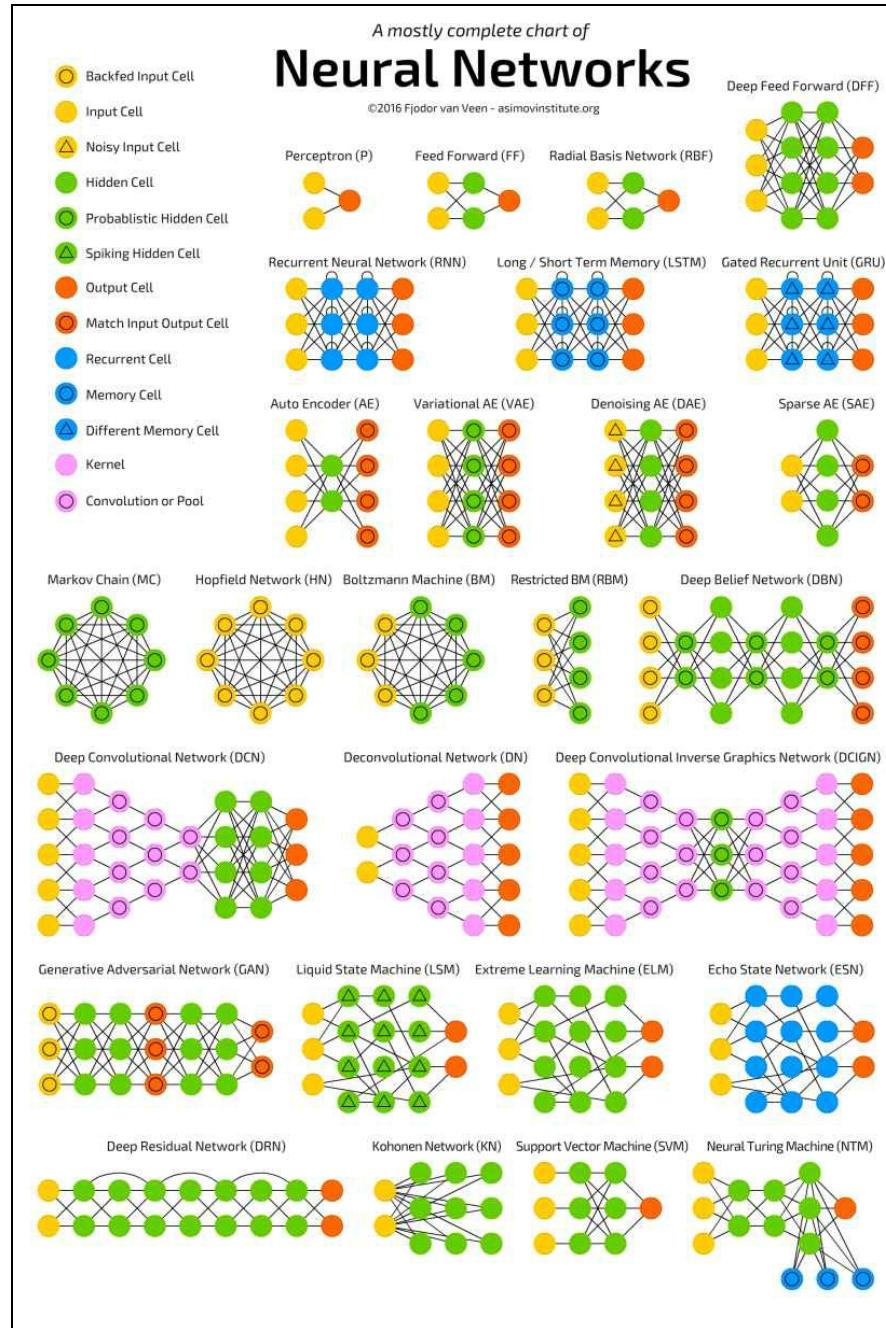


Figure 13: complete charts of neural networks (Source: <http://www.asimovinstitute.org/neural-network-zoo/>)

- **Theano**: This is probably the most widespread library. Written in Python, one of the languages most used in the field of machine learning (also in TensorFlow), allows the calculation also GPU, coming to have 24x the performance of the CPU. It allows you to define, optimize, and evaluate complex mathematical expressions such as multidimensional arrays.
- **Caffe**: This was developed primarily by **Berkeley Vision and Learning Center (BVLC)**, and is a framework designed to stand out in terms of expression, speed, and modularity. Its unique architecture encourages application and innovation by allowing easier transition from calculation of CPU to GPU. The large community of users allowed for considerable development in recent times. It is written in Python, but the installation process can be long due to the numerous support libraries to compile.
- **Torch**: This is a vast ecosystem for machine learning which offers a large number of algorithms and functions, including those for deep learning and processing of various types of multimedia, such as audio and video data, with particular attention to parallel computing. It provides an excellent interface for the C language and has a large community of users. Torch is a library that extends the scripting language Lua, and is intended to provide a flexible environment for designing and training machine learning systems. Torch is a self-contained and highly portable framework on each platform (for example, Windows, Mac, Linux, and Android) and scripts manage to run on these platforms without modification. The Torch package provides many useful features for different applications.

Finally, the following table provides a summary of each framework's salient features, including TensorFlow, which will be described (of course!!) in the coming chapters:

	TensorFlow	Torch	Caffe	Theano
Programming language used	Python and C++	Lua	C++	Python
GPU card support	Yes	Yes	Yes	By default, no
Pros	<ul style="list-style-type: none"> • Computational graph abstraction, like Theano • Faster compile times than Theano • TensorBoard for visualization • Data and model 	<ul style="list-style-type: none"> • Easy to set up • Helpful error messages • Large amount of sample code and tutorials 	<ul style="list-style-type: none"> • Good for feedforward networks and image processing • Good for fine tuning existing networks • Python interface is pretty useful 	<ul style="list-style-type: none"> • Expressive Python syntax • Higher-level spin-off frameworks • Large amount of sample code

	parallelism			and tutorials
Cons	<ul style="list-style-type: none"> • Slower than other frameworks • Not many pre-trained models • Computational graph is pure Python, therefore slow • No commercial support 	<ul style="list-style-type: none"> • Can be somewhat difficult to set up in CentOS 	<ul style="list-style-type: none"> • Error messages are cryptic • Need to write C++ / CUDA for new GPU layers • Not good for recurrent networks • No commercial support 	<ul style="list-style-type: none"> • Error messages can be unhelpful • Large models can have long compile times

Deep learning framework comparison

Summary

In this chapter, we introduced some of the fundamental themes of deep learning. It consists of a set of methods that allow a machine learning system to obtain a hierarchical representation of data, on multiple levels. This is achieved by combining simple units, each of which transforms the representation at its own level, starting from the input level, in a representation at a higher level, slightly more abstract.

In recent years, these techniques have provided results never seen before in many applications, such as image recognition and speech recognition. One of the main reasons for the spread of these techniques has been the development of GPU architectures, which considerably reduced the training time of DNNs. There are different DNN architectures, each of which has been developed for a specific problem. We'll talk more about those architectures in later chapters, showing examples of applications created with the TensorFlow framework.

The chapter ended with a brief overview of the implemented deep learning frameworks.

In the next chapter, we begin our journey into deep learning, introducing the TensorFlow software library. We will describe its main features and look at how to install it and set up a first working session.

First Look at TensorFlow

TensorFlow is mathematical software and an open-source software library for Machine Intelligence, developed in 2011, by Google Brain Team. The initial target of TensorFlow was to conduct research in *machine learning* and in *deep neural networks*. However, the system is general enough to be applicable in a wide variety of other domains as well.

The name is derived from the *data model* which is represented by **tensors** and from the **data flow graph** that stands for the TensorFlow's *execution model*. In 2015, Google has open-sourced the TensorFlow and all of its reference implementation and made all the source code available on GitHub under the Apache 2.0 license. After that, TensorFlow has achieved wide adaption, form academia and research to industry and following that recently the most stable version 1.0 has been released with a unified API.

Keeping in mind your needs and based on all the latest and exciting features of TensorFlow 1.x, this chapter will give a description of the main TensorFlow's capabilities. The following topics will be discussed in this chapter:

- General overview
- Installing and getting started with TensorFlow
- Computation graph
- The programming model
- Data model
- TensorBoard
- Implementing a single input neuron
- Migrating to TensorFlow 1.x

General overview

TensorFlow is an open source software library for numerical computation using data flow graphs that enables machine learning practitioners to do more data-intensive computing. It provides some robust implementations of widely used deep learning algorithms. Nodes in the flow graph represent mathematical operations. On the other hand, the edges represent multidimensional tensors that ensure communication between edges and nodes. TensorFlow offers you a very flexible architecture that enables you to deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device with a single API.

What's new with TensorFlow 1.x?

The APIs in TensorFlow 1.0 have changed in ways that are not all backward-compatible. That is, TensorFlow programs that worked on TensorFlow 0.x won't necessarily work on TensorFlow 1.x. These API changes have been made to ensure an internally-consistent API. In other words, Google does not have any plans to make TensorFlow backwards-breaking changes throughout the 1.x lifecycle.

In the latest TensorFlow 1.x version, Python APIs resemble NumPy more closely. This has made the current version more stable for array-based computation. Two experimental APIs for Java and GO have been introduced too. This is very good news for the Java and GO programmer.

A new tool called **TensorFlow Debugger (tfdbg)** has been introduced. This is a command-line interface and API for debugging live TensorFlow programs. A new Android demos (<https://github.com/tensorflow/tensorflow/tree/r1.0/tensorflow/examples/android>) for object detection and localization and camera-based image stylization have been made available.

Now the installation of TensorFlow can be done through an Anaconda and Docker image of TensorFlow. Finally and most importantly, a new domain-specific compiler for TensorFlow graphs targeting CPU and GPU computing has been introduced. This is called **Accelerated Linear Algebra (XLA)**.

How does it change the way people use it?

The main features offered by the latest release of TensorFlow are as follows:

- **Faster computing:** The major versioning upgrade to TensorFlow 1.0 has made its capability incredibly faster including a 7.3x speedup on 8 GPUs for inception v3 and 58x speedup for distributed Inception (v3 training on 64 GPUs).
- **Flexibility:** TensorFlow is not just a deep learning or machine learning software library but also great a library full with powerful mathematical functions with which you can solve most different problems. The execution model that uses the data flow graph allows you to build very complex models from simple sub-models. TensorFlow 1.0 introduces high-level APIs for TensorFlow, with `tf.layers`, `tf.metrics`, `tf.losses` and `tf.keras` modules. These have made TensorFlow very suitable for high-level neural network computing
- **Portability:** TensorFlow runs on Windows, Linux, and Mac machines and on mobile computing platforms (that is, Android).
- **Easy debugging:** TensorFlow provides the TensorBoard tool for the analysis of the developed models.
- **Unified API:** TensorFlow offers you a very flexible architecture that enables you to deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device with a single API.
- **Transparent use of GPU computing:** Automating management and optimization of the same memory and the data used. You can now use your machine for large-scale and data-intensive GPU computing with NVIDIA, cuDNN, and CUDA tool kits.
- **Easy Use:** TensorFlow is for everyone; it's for students, researchers, and deep learning practitioners and also for readers of this book.
- **Production ready at scale:** Recently it has been evolved as the neural network for machine translation, at production scale. TensorFlow 1.0 promises Python API stability making it easier to choose new features without worrying too much about breaking your existing code.
- **Extensibility:** TensorFlow is relatively new technology and it's still under active development. However, it is extensible because it was released with the source code available on GitHub ([http://github.com/tensorflow/tensorflow](https://github.com/tensorflow/tensorflow)). And if you don't see the low-level data operator you need, you can write it in C++ and add it to the framework.
- **Supported:** There is a large community of developers and users working together to improve TensorFlow both by providing feedback and by actively contributing to the source code.
- **Wide adaption:** Numerous tech giants are using TensorFlow to increase their business intelligence. For example, ARM, Google, Intel, eBay, Qualcomm, SAM, DropBox, DeepMind, Airbnb, Twitter and so on.

Installing and getting started with TensorFlow

You can install and use TensorFlow on a number of platforms such as Linux, Mac OSX, and Windows. You can also build and install TensorFlow from the latest GitHub source of TensorFlow. Also if you have a Windows machine you can install TensorFlow only if you have a virtual machine. The TensorFlow Python API supports Python 2.7 and Python 3.3+ so you need to install Python to start the TensorFlow installation. You must install Cuda Toolkit 7.5 and cuDNN v5.1+. In this section, we will show how to install and getting started with TensorFlow. More details on installing TensorFlow on Linux will be shown. A short overview of Windows will be provided as well.



Note that, for this and the rest of the chapters, we will provide all the source codes with Python 2.7 computable. However, you will find all of them with Python 3.3+ compatible on the Packt repository.



Installing on Mac OS is more or less similar to Linux. Please refer to URL at https://www.tensorflow.org/install/install_mac for more details.

Installing TensorFlow on Linux

In this section, we will show how to install TensorFlow on Ubuntu 14.04 or higher. The instructions presented here also might be applicable for other Linux distros.

Which TensorFlow to install on your platform?

However, before proceeding with the formal steps, we need to determine which TensorFlow to install on your platform. TensorFlow has been developed such that you can run data-intensive tensor application on GPU as well as CPU. Thus, you should choose one of the following types of TensorFlow to install on your platform:

- **TensorFlow with CPU support only:** If there is no GPU such as NVIDIA installed on your machine, you must install and start computing using this version. This is very easy and you can do it in just 5 to 10 minutes.
- **TensorFlow with GPU support:** As you might know, a deep learning application requires typically very high intensive computing resources. Thus TensorFlow is no exception but can typically speed up data computation and analytics significantly on a GPU rather than on a CPU. Therefore, if there's NVIDIA GPU hardware on your machine, you should ultimately install and use this version.

From our experience, even if you have NVIDIA GPU hardware integrated on your machine, it would be worth installing and trying the CPU only version first and if you don't experience good performance you should switch to GPU support then.

Requirements for running TensorFlow with GPU from NVIDIA

The GPU-enabled version of TensorFlow has several requirements such as 64-bit Linux, Python 2.7 (or 3.3+ for Python 3), NVIDIA CUDA 7.5 (CUDA 8.0 required for Pascal GPUs) and NVIDIA, cuDNN v4.0 (minimum) or v5.1 (recommended). More specifically, the current development of TensorFlow supports only GPU computing using NVIDIA toolkits and software. Now the following software must be installed on your machine.

Step 1: Install NVIDIA CUDA

To use TensorFlow with NVIDIA GPUs, CUDA Toolkit 8.0 and associated NVIDIA drivers with CUDA Toolkit 8+ need to be installed.



*For more details, refer to this NVIDIA's documentation
<https://developer.nvidia.com/cuda-downloads>.*

Now download and install the required package from <https://developer.nvidia.com/cuda-downloads> as shown in the following screenshot:

The screenshot shows the 'Select Target Platform' section of the NVIDIA CUDA Download page. It includes fields for Operating System (Windows, Linux, Mac OSX), Architecture (x86_64, ppc64le), Distribution (Fedora, OpenSUSE, RHEL, CentOS, SLES, Ubuntu), Version (16.04, 14.04), and Installer Type (runfile [local], deb [local], deb [network], cluster [local]). A note at the top says: 'Click on the green buttons that describe your target platform. Only supported platforms will be shown.'

Available CUDA packages based on various platforms

Also, ensure that you have added the Cuda installation path to the `LD_LIBRARY_PATH` environment variable.

Step 2: Installing NVIDIA cuDNN v5.1+

Once the CUDA Toolkit is installed, download the cuDNN v5.1 Library from <https://developer.nvidia.com/cudnn> for Linux as shown in the following screenshot:

cuDNN Download

NVIDIA cuDNN is a GPU-accelerated library of primitives for deep neural networks.

I Agree To the Terms of the cuDNN Software License Agreement

Please check your framework documentation to determine the recommended version of cuDNN. If you are using cuDNN with a Pascal (GTX 1080, GTX 1070), version 5 or later is required.

[Download cuDNN v5.1 \[Jan 20, 2017\], for CUDA 8.0](#)

[cuDNN User Guide](#)
[cuDNN Install Guide](#)
[cuDNN v5.1 Library for Linux](#)
[cuDNN v5.1 Library for Power8](#)
[cuDNN v5.1 Library for Windows 7](#)
[cuDNN v5.1 Library for Windows 10](#)
[cuDNN v5.1 Library for OSX](#)
[cuDNN v5.1 Release Notes](#)
[cuDNN v5.1 Runtime Library for Ubuntu14.04 \(Deb\)](#)
[cuDNN v5.1 Developer Library for Ubuntu14.04 \(Deb\)](#)
[cuDNN v5.1 Code Samples and User Guide \(Deb\)](#)
[cuDNN v5.1 Runtime Library for Ubuntu16.04 Power8 \(Deb\)](#)
[cuDNN v5.1 Developer Library for Ubuntu16.04 Power8 \(Deb\)](#)
[cuDNN v5.1 Code Samples and User Guide Power8 \(Deb\)](#)

[Download cuDNN v5.1 \[Jan 20, 2017\], for CUDA 7.5](#)

Available cuDNN v5.1 Library depending on platform

Once downloaded, uncompressing the files and copy them into the CUDA Toolkit directory (assumed here to be in `/usr/local/cuda/`):

```
$ sudo tar -xvf cudnn-8.0-linux-x64-v5.1.tgz -C /usr/local
```



Note that, to install the cuDNN v5.1 library, you must need to register for the Accelerated Computing Developer Program at <https://developer.nvidia.com/accelerated-computing-developer>.

Now when you have installed cuDNN v5.1 library, ensure that you create the `CUDA_HOME` environment variable.

Step 3: GPU card with CUDA compute capability 3.0+

Make sure that your machine comes with the GPU card with CUDA compute capability 3.0+, to use the preceding library and tools in Steps 1 and 2.

Step 4: Installing the libcupti-dev library

Lastly, you need have `libcupti-dev` library installed on your machine. This is the NVIDIA CUDA provides advanced profiling support. To install this library, issue the following command:

```
| $ sudo apt-get install libcupti-dev
```


Step 5: Installing Python (or Python3)

For those who are new to Python or TensorFlow, we recommend you install TensorFlow using `pip`. Python 2+ and 3.3+ are automatically installed on Ubuntu. Make check to sure that `pip` or `pip3` is installed using the following command:

```
| $ python -V  
| Expected output:  
| Python 2.7.6  
  
| $ which python  
| Expected output:  
| /usr/bin/python
```

For Python 3.3+ use the following:

```
| $ python3 -V  
| Expected output:  
| Python 3.4.3
```

If you want a very specific version:

```
| $ sudo apt-cache show python3  
$ sudo apt-get install python3=3.5.1*
```


Step 6: Installing and upgrading PIP (or PIP3)

The `pip` or `pip3` package manager usually comes with Ubuntu. Check that `pip` or `pip3` is installed using the following command:

```
| $ pip -v  
| Expected output:  
|   pip 9.0.1 from /usr/local/lib/python2.7/dist-packages/pip-9.0.1-py2.7.egg (python 2.7)
```

For Python 3.3+ use the following:

```
| $ pip3 -v
```

The expected output is as follows:

```
| pip 1.5.4 from /usr/lib/python3/dist-packages (python 3.4)
```

It is to be noted that `pip` version 8.1+ or `pip3` version 1.5+ are strongly recommended for better results and smooth computation. If version 8.1+ for `pip` and 1.5+ for `pip3` are not installed, either install or upgrade to the latest `pip` version:

```
| $ sudo apt-get install python-pip python-dev
```

For Python 3.3+, use the following command:

```
| $ sudo apt-get install python3-pip python-dev
```


Step 7: Installing TensorFlow

Refer to the following section, for more step-by-step guidelines on how to install the latest version of TensorFlow for the CPU only and GPU supports with NVIDIA cuDNN and CUDA computing capability.

How to install TensorFlow

You can install TensorFlow on your machine in a number of ways; such as using virtualenv, pip, Docker, and Anaconda. However, using Docker and Anaconda are a bit advanced and this is why we have decided to use pip and virtualenv instead.



Interested readers can try using Docker and Anaconda from this URL at <https://www.tensorflow.org/install/>.

Installing TensorFlow with native pip

If Steps 1 to 6 have been completed, install TensorFlow by invoking one of the following commands, for Python 2.7 and of course with only CPU support:

```
| $ pip install tensorflow
```

For Python 3.x and of course with only CPU support:

```
| $ pip3 install tensorflow
```

For Python 2.7 and of course with GPU support:

```
| $ pip install tensorflow-gpu
```

For Python 3.x and of course with GPU support:

```
| $ pip3 install tensorflow-gpu
```

If Step 3 failed, install the latest version of TensorFlow by issuing this command manually:

```
| $ sudo pip install --upgrade TF PYTHON URL
```

For Python 3.x, use the following command:

```
| $ sudo pip3 install --upgrade TF PYTHON URL
```

For both cases, `TF PYTHON URL` signifies the URL of the TensorFlow Python package presented at this URL: https://www.tensorflow.org/install/install_linux#the_url_of_the_tensorflow_python_package

For example, to install the latest version with CPU only support (currently v1.0.1), use the following command:

```
| $ sudo pip3 install --upgrade https://storage.googleapis.com/tensorflow/linux/cpu/tensorflow-1.0.1-cp34-cp34m-linux_x86_64.whl
```


Installing with virtualenv

We assume that you have already Python 2+ (or 3+) and pip (or pip3) are installed on your system. If so, the following are the steps to install TensorFlow:

1. Create a virtualenv environment as follows:

```
| $ virtualenv --system-site-packages targetDirectory
```

The `targetDirectory` signifies the root of the `virtualenv` tree. By default it is `~/tensorflow` (however, you may choose any directory).

2. Activate the `virtualenv` environment as follows:

```
| $ source ~/tensorflow/bin/activate # bash, sh, ksh, or zsh  
| $ source ~/tensorflow/bin/activate.csh # csh or tcsh
```

If the command succeeds in Step 2, then you should see following on your terminal:

```
| (tensorflow)$
```

3. Install TensorFlow:

Use one of the following commands to install TensorFlow in the active virtualenv environment. For Python 2.7 with CPU only support, use the following command:

```
| (tensorflow)$ pip install --upgrade tensorflow
```

4. For Python 3.x with CPU only supports, use the following command:

```
| (tensorflow)$ pip3 install --upgrade tensorflow
```

6. For Python 2.7 with GPU support, use the following command:

```
| (tensorflow)$ pip install --upgrade tensorflow-gpu
```

7. For Python 3.x with GPU supports, use the following command:

```
| (tensorflow)$ pip3 install --upgrade tensorflow-gpu
```

If the preceding command succeeds, skip Step 5. If the preceding command fails, perform Step 5.

If Step 3 failed, try to install TensorFlow in the active virtualenv environment by issuing a command in the following format. For Python 2.7 (select appropriate URL with CPU or GPU support).

```
| (tensorflow)$ pip install --upgrade TF_PYTHON_URL
```

For Python 3.x (select appropriate URL with CPU or GPU support).

```
| (tensorflow)$ pip3 install --upgrade TF PYTHON URL
```

For Python 2.7 with CPU/GPU support, select the appropriate value of `TF PYTHON URL`. For both cases, `TF PYTHON URL` signifies the URL of the TensorFlow Python package presented at this URL: https://www.tensorflow.org/install/install_linux#the_url_of_the_tensorflow_python_package.

For example, to install the latest version with CPU only support (currently v1.0.1), use the following command:

```
| (tensorflow)$ pip3 install --upgrade  
| https://storage.googleapis.com/tensorflow/linux/cpu/tensorflow-1.0.1-cp34-cp34m-linux_x86_64.whl
```

To **validate the installation** in Step 3, you must activate the virtual environment. If the `virtualenv` environment is not currently active, issue one of the following commands:

```
| $ source ~/tensorflow/bin/activate      # bash, sh, ksh, or zsh  
| $ source ~/tensorflow/bin/activate.csh  # csh or tcsh
```

To **uninstall TensorFlow**, simply remove the tree you created. For example:

```
| $ rm -r targetDirectory
```


Installing TensorFlow on Windows

If you can't get a Linux-based system, you must install Ubuntu on a virtual machine; just use a free application called **VirtualBox**, which lets you create a virtual PC on Windows and install Ubuntu in it. TensorFlow only supports version 3.5.x of Python on Windows. Note that Python 3.5.x comes with the `pip3` package manager, which is the program you'll use to install TensorFlow. To install TensorFlow, start a command prompt and then issue the appropriate `pip3` install command in that terminal. To install the CPU-only version of TensorFlow, enter the following command:

```
| C:> pip3 install --upgrade tensorflow
```

To install the GPU version of TensorFlow, enter the following command:

```
| C:> pip3 install --upgrade tensorflow-gpu
```


Installation from source

The Pip installation can cause problems using TensorBoard.



See <https://github.com/tensorflow/tensorflow/issues/530> for more references.

For this reason, I suggest you build TensorFlow directly from the source.

The steps are as follows:

1. Clone the entire TensorFlow repository:

```
| $git clone --recurse-submodules  
| https://github.com/tensorflow/tensorflow
```

2. Install Bazel, which is a tool that automates software builds and tests.

Now, to build TensorFlow from source, the Bazel build system must be installed on your machine. If not, issue the following command:

```
$ sudo apt-get install software-properties-common swig  
$ sudo add-apt-repository ppa:webupd8team/java  
$ sudo apt-get update  
$ sudo apt-get install oracle-java8-installer  
$ echo "deb http://storage.googleapis.com/bazel-apt stable jdk1.8" | sudo tee /etc/apt/sources.list.d/bazel  
$ curl https://storage.googleapis.com/bazel-apt/doc/apt-key.pub.gpg | sudo apt-key add -  
$ sudo apt-get update  
$ sudo apt-get install bazel
```

Following the instructions and guidelines on how to install Bazel on your platform at: <http://bazel.io/docs/install.html>.

1. Run the Bazel installer:

```
| $ chmod +x bazel-version-installer-os.sh
```

2. Then run the following command:

```
| $ ./bazel-version-installer-os.sh --user
```

3. Install Python dependencies:

```
| $ sudo apt-get install python-numpy swig python-dev
```

4. Configure the installation (GPU or CPU) by the following command:

```
| $ ./configure
```

5. Create your TensorFlow package using `bazel`:

```
| $ bazel build -c opt //tensorflow/tools/pip_package:  
| $ build_pip_package
```

6. To build with GPU support use the following command:

```
| $ bazel build -c opt --config=cuda  
| //tensorflow/tools/pip_package:build_pip_package
```

7. Finally, install TensorFlow.

The following are the code listed as per Python version:

For Python 2.7:

```
| $ sudo pip install --upgrade /tmp/tensorflow_pkg/tensorflow-1.0.11-* .whl
```

For Python 3.4:

```
| $ sudo pip3 install --upgrade /tmp/tensorflow_pkg/tensorflow-1.0.1-* .whl
```

The name of the `.whl` file will depend on your platform (aka OS).

Install on Windows

If you can't get a Linux-based system, you must install Ubuntu on a virtual machine. Use a free application called VirtualBox, which lets you create a virtual PC on Windows and install Ubuntu on the latter `cudnn-8.0-linux-x64-v5.1.tgz` file.

Test your TensorFlow installation

Open a Python terminal and enter the following lines of code:

```
| >>> import tensorflow as tf  
| >>> hello = tf.constant("hello TensorFlow!")  
| >>> sess=tf.Session()
```

To verify your installation just type:

```
| >>> print sess.run(hello)
```

If the installation is okay, you'll see the following output:

```
| Hello TensorFlow!
```


Computational graphs

When performing an operation, for example training a neural network, or the sum of two integers, TensorFlow internally represent its computation using a *data flow graph* (or *computational graph*).

This is a directed graph consisting of the following:

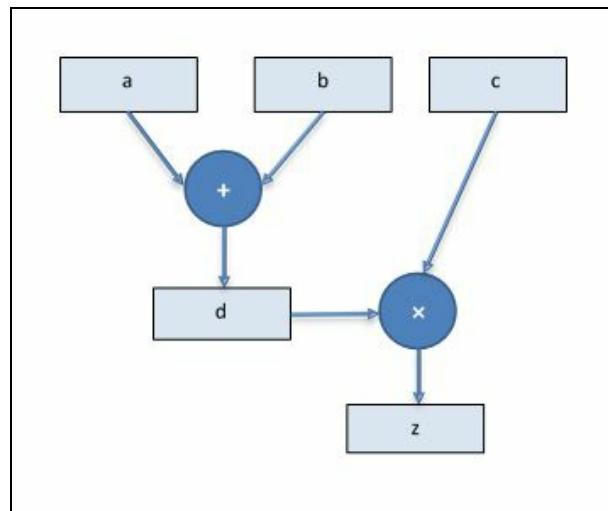
- A set of nodes, each one representing an operation
- A set of directed arcs, each one representing the data on which the operations are performed

TensorFlow has two types of edge:

- **Normal:** They are only carriers of data structures, between the nodes. The output of one operation (from one node) becomes the input for another operation. The edge connecting two nodes carry the values.
- **Special:** This edge doesn't carry values. It represents a *control dependency* between two nodes A and B. It means that the node B will be executed only if the operation in A will be ended before the relationship between operations on the data.

The TensorFlow implementation defines *control dependencies* to enforce orderings between otherwise independent operations as a way of controlling peak memory usage.

A computational graph is basically like a *flow chart*; the following is the computational graph for a simple computation: $z = d \times c = (a + b) \times c$.



A very simple execution graph

In the preceding figure, the circles in the graph indicate the operations, while rectangles indicate the data computational graph

Why a computational graph?

Another key idea in TensorFlow is the *deferred execution*, during the building phase of the computational graph, you can compose very complex expressions (we say it is *highly compositional*), when you want to evaluate them through the running session phase, TensorFlow schedules the running in the most efficient manner (for example, parallel execution of independent parts of the code using the GPU).

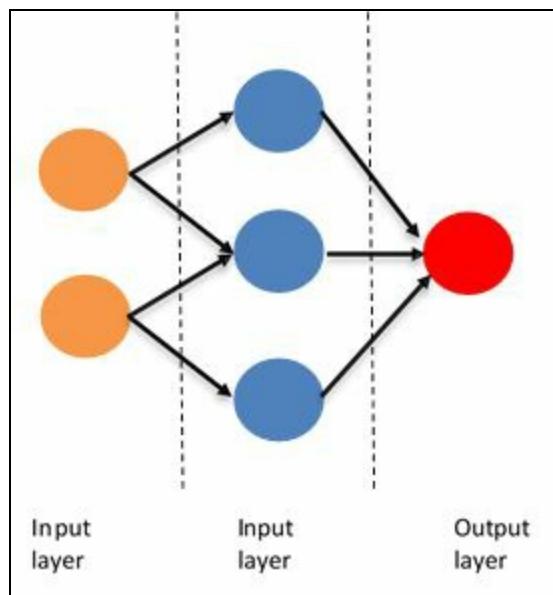
In this way, a graph helps to distribute the computational load if one must deal with complex models containing a large number of nodes and layers.

Finally, a neural network can be compared to a *composite function* where each network layer can be represented as a function.

This consideration leads us to the next section, where the role of the computational graph in implementing a neural network is explained.

Neural networks as computational graphs

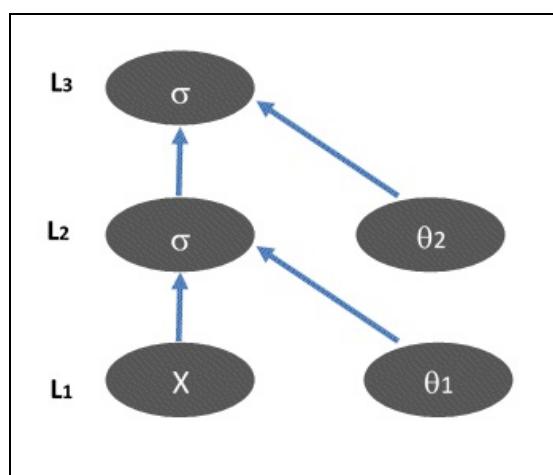
Consider the *feedforward neural network* represented in the preceding figure, for example, it could represent the model for a XOR gate:



Neural net architecture for the XOR problem

The following architectural representation is *not very useful* if you want to implement the network, because it does not consider the *weight* and *bias* vectors, which play a key role, in training phase, during the backpropagation algorithm. For this reason, it is convenient to *represent* the network as a computational graph, that is, in terms of data *flowing* through the network.

Just by way of explanation, let's focus on the following figure:



Computational graph implementation of the XOR neural network

Here L3 denotes the *output layer*, L2 the *hidden layer*, and L1 the *input layer*. Similarly, θ_2 denotes the weight vector between layer 2 and layer 3; θ_1 denotes the weight vector between layer 1 and layer 2. The σ notation just refers to the sigmoid operation that takes place within those nodes (however, the outputs of those nodes will be referred to using the L notation, that is, L1, L2, and L3).

In this representation, each node does not represent a *single neuron*, but rather *an operation*, while the arrows *are not connections* but represent the *flow of information* along the network.

This computational graph shows us, step by step, which functions occur and the inputs those functions operated on.

The L2 layer function operates on two inputs: the outputs L1 layer (a vector) and the weight vector θ_1 , while the L2 and L3 function operates on θ_2 , and is our final output.

These considerations lead us to another peculiar aspect of TensorFlow, the *programming model*.

The programming model

A TensorFlow program is generally divided into three phases:

- *Construction* of the computational graph
- *Running* a session, which is performed for the operations defined in the graph
- Resulting data *collection* and *analysis*

These main steps define the *programming model* in TensorFlow.

Consider the following example, in which we want to multiply two numbers.

```
import tensorflow as tf
with tf.Session() as session:
    x = tf.placeholder(tf.float32, [1], name="x")
    y = tf.placeholder(tf.float32, [1], name="y")
    z = tf.constant(2.0)
    y = x * z
x_in = [100]
y_output = session.run(y, {x:x_in})
print(y_output)
```

Surely TensorFlow is not necessary to multiply two numbers; also the number of lines of the code for this simple operation is so many. However, the example wants to clarify *how to structure any code*, from the simplest as in this instance, to the most complex.

Furthermore, the example also contains some basic instructions will found in all the other examples given in the course of this book.

Note that, for this and the remaining chapters, we will provide all the source code with Python 2.7 computable. However, you will find all of them with Python 3.3+ compatible on the Packt repository.

The first line tells us that the TensorFlow library is imported as `tf`:

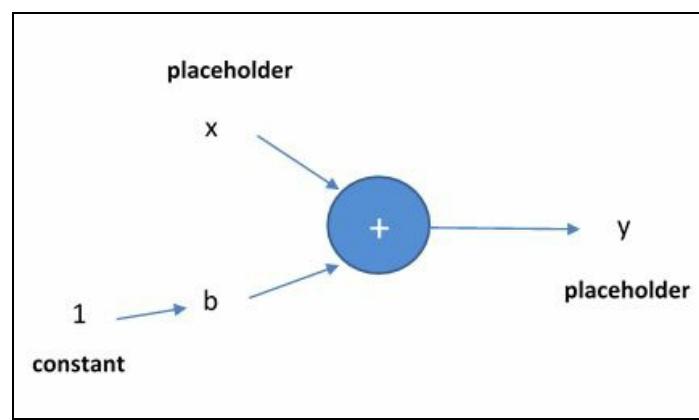
```
| import tensorflow as tf
```

The TensorFlow operator will then be expressed by `tf` and the dot `.` and by the name of the operator to use. In the next line, we construct the object `session`, by means of the instruction `tf.Session()`:

```
| with tf.Session() as session:
```

This object *contains* the computation graph, which as we said earlier, are the calculations to be carried out.

For our example, the computational graph will be the following:



A computational graph that represents multiplying the constant by 1.0 to an input array x

The following two lines define variables x and y , using the notion of a placeholder. Through a placeholder you may define both an input (such as the variable x of our example) and an output variable (such as the variable y):

```
| x = tf.placeholder(tf.float32, [1], name='x')
| y = tf.placeholder(tf.float32, [1], name='y')
```

A placeholder is, therefore, an interface between the elements of the graph and the computational data of the problem; it allows us to create our operations and build our computation graph, without needing the data, but only a reference to it.

To define a *data* or *tensor* via the `placeholder` function three arguments are required; the first is the *data type*, the second argument is the *shape* of a placeholder, which is a single dimensional Tensor (https://www.tensorflow.org/versions/r0.8/api_docs/python/framework.html#Tensor) with one entry, and the third argument is a *name*, very useful for debugging and code analysis purposes.

The following statement defines a *constant* (=1.0) to which we give the name b :

```
| b = tf.constant(1.0)
```

So, we may introduce the model that we want to compute; it has as arguments the placeholder and the constant previously defined:

```
| y = x * b
```

This statement, inside the *session*, builds the data structures of the x product with b , and the subsequent assignment of the result of the operation to the placeholder y .

In the next lines, we define the computational model. Our model has one input x , so we create a list x_in that will be associated with the evaluated placeholder x :

```
| x_in = [2]
```

Variable x_in contains the value 2 which will be during the computation, the value that will be passed to placeholder x .

Finally, we execute the graph, through the `session.run` statement:

```
| y_final = session.run([y], {x: x_in})
```

The first argument is a list of graph elements to evaluate, in our case we are considering only the `[y]`, while the second argument `{x: x_in}` indicates to what values we are evaluating our argument.

`session.run` returns `y_output` as the output value for each graph element that you pass in as the first argument, and the value will correspond to the evaluated value for that element.

In the final instruction, we print out the result:

```
|     9. print(y_output)
```

We recall that the defined graph elements are processed only when `session.run()` is executed.

It can manipulate very large and complex values of `x`, so we create a list `x_in` that will be associated with the placeholder `x`.

Data model

The *data model* in TensorFlow is represented by **tensors**. Without using complex mathematical definitions, we can say that a tensor (in TensorFlow) identifies a *multidimensional numerical array*.

This data structure is characterized by three parameters--*Rank, Shape, and Type*.

Rank

Each tensor is described by a unit of dimensionality called **rank**. It identifies the number of dimensions of the tensor, for this reason, a rank is a known-as order or n-dimensions of a tensor. A rank zero tensor is a scalar, a rank one tensor ID a vector, while a rank two tensor is a matrix.

The following code defines a TensorFlow *scalar*, a *vector*, a *matrix* and a *cube_matrix*, in the next example we show how the rank works:

```
import tensorflow as tf

scalar = tf.constant(100)
vector = tf.constant([1,2,3,4,5])
matrix = tf.constant([[1,2,3],[4,5,6]])
cube_matrix = tf.constant([[ [1], [2], [3] ], [[4], [5], [6]], [[7], [8], [9]] ] )

print(scalar.get_shape())
print(vector.get_shape())
print(matrix.get_shape())
print(cube_matrix.get_shape())
```

The results are printed here:

```
>>>
()
(5,)
(2, 3)
(3, 3, 1)
>>>
```


Shape

The *shape* of a tensor is the number of rows and columns it has. Now we see how to relate the shape to a rank of a tensor:

```
>>scalar1.get_shape()
TensorShape([])

>>vector1.get_shape()
TensorShape([Dimension(5)])

>>matrix1.get_shape()
TensorShape([Dimension(2), Dimension(3)])

>>cube1.get_shape()
TensorShape([Dimension(3), Dimension(3), Dimension(1)])
```


Data types

In addition to rank and shape, tensors have a *data type*. The following is a list of the data types:

Data type	Python type	Description
DT_FLOAT	tf.float32	32-bit floating point.
DT_DOUBLE	tf.float64	64-bit floating point.
DT_INT8	tf.int8	8-bit signed integer.
DT_INT16	tf.int16	16-bit signed integer.
DT_INT32	tf.int32	32-bit signed integer.
DT_INT64	tf.int64	64-bit signed integer.
DT_UINT8	tf.uint8	8-bit unsigned integer.
DT_STRING	tf.string	Variable length byte arrays. Each element of a tensor is a byte array.
DT_BOOL	tf.bool	Boolean.
DT_COMPLEX64	tf.complex64	Complex number made of two 32 bits floating points, real and imaginary parts.
DT_COMPLEX128	tf.complex128	Complex number made up of two 64 bits floating points, real and imaginary parts.
DT_QINT8	tf.qint8	8-bit signed integer used in quantized Ops.
DT_QINT32	tf.qint32	32-bit signed integer used in quantized ops.
DT_QUINT8	tf.quint8	8-bit unsigned integer used in quantized ops.

We believe the preceding table is self-explanatory hence we did not provide details on the data types. Note that the TensorFlow APIs are implemented to manage data *to* and *from* NumPy arrays.

To build a tensor with a constant value, pass a NumPy array to the `tf.constant()` operator, and the result will be a TensorFlow tensor with that value:

```
import tensorflow as tf
import numpy as np

tensor_1d = np.array([1,2,3,4,5,6,7,8,9,10])
tensor_1d = tf.constant(tensor_1d)
with tf.Session() as sess:
    print (tensor_1d.get_shape())
    print sess.run(tensor_1d)
```

Running the example, we obtain the following output:

```
>>>
(10,)
[ 1  2  3  4  5  6  7  8  9 10]
```

To build a tensor, with variable values, use a `NumPy` array and pass it to the `tf.Variable` constructor, the result will be a TensorFlow variable tensor with that initial value:

```
import tensorflow as tf
import numpy as np

tensor_2d = np.array([(1,2,3),(4,5,6),(7,8,9)])
tensor_2d = tf.Variable(tensor_2d)
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    print (tensor_2d.get_shape())
    print sess.run(tensor_2d)
```

The result is as follows:

```
>>>
(3, 3)
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

For ease of use in interactive Python environments, we can use the `InteractiveSession` class, (https://www.tensorflow.org/versions/r0.10/api_docs/python/client/#InteractiveSession) and then use that session for all `Tensor.eval()` and `Operation.run()` calls:

```
import tensorflow as tf
import numpy as np

interactive_session = tf.InteractiveSession()
tensor = np.array([1,2,3,4,5])
tensor = tf.constant(tensor)
print(tensor.eval())
interactive_session.close()
```

The result is as follows:

```
>>>
[1 2 3 4 5]
```

This can be easier in an interactive setting, such as the shell or an IPython Notebook, when it's tedious to pass around a `Session` object everywhere.

Another way to define a tensor uses the TensorFlow statement `tf.convert_to_tensor`:

```
import tensorflow as tf
import numpy as np

tensor_3d = np.array([[[ 0,  1,  2],[ 3,  4,  5],[ 6,  7,  8]],
                     [[ 9, 10, 11],[12, 13, 14],[15, 16, 17]],
                     [[18, 19, 20],[21, 22, 23],[24, 25, 26]]])

tensor_3d = tf.convert_to_tensor(tensor_3d, dtype=tf.float64)
with tf.Session() as sess:
    print (tensor_3d.get_shape())
    print sess.run(tensor_3d)
```

The result is as follows:

```
>>>
(3, 3, 3)
[[[ 0.   1.   2.]
 [ 3.   4.   5.]
 [ 6.   7.   8.]]]

[[ 9.  10.  11.]
 [12.  13.  14.]
 [15.  16.  17.]]]

[[ 18.  19.  20.]
 [21.  22.  23.]
 [24.  25.  26.]]]
```


Variables

Variables are TensorFlow objects that hold and update parameters. A variable must be initialized; also you can save and restore it to analyze your code.

Variables are created by the `tf.Variable()` statement.

In the following example, we want to count the numbers from 1 to 10:

```
| import tensorflow as tf
```

We create a variable that will be initialized to the scalar value 0:

```
| value = tf.Variable(0, name="value")
```

The `assign()` and `add()` operators are just nodes of the computation graph so they do not execute the assignment until the session is run:

```
one = tf.constant(1)
new_value = tf.add(value, one)
update_value= tf.assign(value, new_value)

initialize_var = tf.global_variables_initializer()
```

We can instantiate the computation graph:

```
with tf.Session() as sess:
    sess.run(initialize_var)
    print(sess.run(value))
    for _ in range(10):
        sess.run(update_value)
        print(sess.run(value))
```

Let's recall that a tensor object is a symbolic handle to the result of an operation, but does not actually hold the values of the operation's output:

```
>>>
0
1
2
3
4
5
6
7
8
9
10
>>>
```

 *You typically represent the parameters of a statistical model as a set of variables. For example, you would store the weights for a neural network as a tensor in a variable. During the training phase, you update this tensor by running a training graph repeatedly.*

Fetches

To fetch the outputs of operations, execute the graph by calling the `run()` function on the `Session` object and pass in the tensors to be retrieved. Except for fetching the single tensor node, you can also fetch *multiple tensors*. In the following example, the `sum_` and `multiply_` tensors are fetched together, using the `run()` call:

```
import tensorflow as tf

constant_A = tf.constant([100.0])
constant_B = tf.constant([300.0])
constant_C = tf.constant([3.0])

sum_ = tf.add(constant_A, constant_B)
mul_ = tf.multiply(constant_A, constant_C)

with tf.Session() as sess:
    result = sess.run([sum_, mul_])
    print(result)
```

The output is as follows:

```
>>>
[array([ 400.], dtype=float32), array([ 300.], dtype=float32)]
```

All the ops needed to produce the values of the requested tensors are run once (not once per requested tensor).

Feeds

A *feed mechanism* is provided to patch a tensor into a graph node. It temporarily replaces the output of an operation with a tensor value. The feed is only used for the run call to which it is passed using the `feed_dict` parameter. The most common use case involves designating specific operations to be *feed* operations by using `tf.placeholder()` to create them.

The following example shows how to feed data to build a random 2×3 matrix:

```
import tensorflow as tf
import numpy as np

a = 3
b = 2
x = tf.placeholder(tf.float32, shape=(a,b))
y = tf.add(x,x)
data = np.random.rand(a,b)
sess = tf.Session()
print sess.run(y,feed_dict={x:data})
```

The output is as follows:

```
>>>
[[ 1.78602004  1.64606333]
 [ 1.03966308  0.99269408]
 [ 0.98822606  1.50157797]]
>>>
```


TensorBoard

When training a neural network, it may be useful to keep track of network parameters, typically the inputs and outputs from the nodes, so you can see whether your model is learning such verifying after each training step if the function error is minimized or not. Of course, writing code to display the behavior of the network during the learning phase, it can be not easy.



Installing TensorBoard is pretty straight forward. Just issue the following command on Terminal (On Ubuntu for Python 2.7+):

```
$ sudo pip install tensorboard
```

Fortunately, TensorFlow provides **TensorBoard** which is a framework designed for analysis and debugging of neural network models. TensorBoard uses the so-called *summaries* to view the parameters of the model; once a TensorFlow code is executed, we can call TensorBoard to view summaries in a **graphical user interface (GUI)**.

Furthermore, TensorBoard can be used to display and study the TensorFlow's computational graph, that can be very complex for a Deep Neural Network model.

How does TensorBoard work?

As explained, TensorFlow uses the computation graph to execute an application, where each node represents an operation and the arcs are the data between operations. The main idea in TensorBoard is to associate the so-called *summary* with nodes (*operations*) of the graph.

Running the code, the summary operations will *serialize the data* of the node that is associated with it, and output the data, into a file that can be read by TensorBoard.

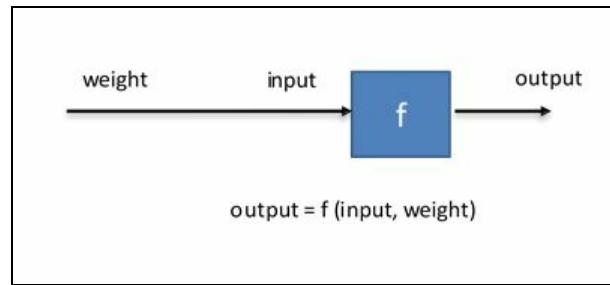
Then we can run TensorBoard and visualize the *summarized operations*. The *workflow* when using TensorBoard is:

- *Build* your computational graph/code
- *Attach* summary ops to the nodes you are interested in examining
- Start *running* your graph as you normally would
- Additionally, *run* the summary ops
- When the code is done running, *run* TensorBoard
- *Visualize* the summary outputs

In the last section, we'll combine all the ideas previously explained to build a *single input neuron* model and to analyze it with TensorBoard.

Implementing a single input neuron

In this example, we will take a closer look at TensorFlow's and TensorBoard's main concepts and try to do some basic operations to get you started. The model we want to implement simulates a *single neuron*. For reference, see the following diagram:



A schema representing the single input neuron

The output will be the input product for the weight.

The preceding schema consists of the following:

- An *input value*, which stimulates the neuron.
- A single *weight*, which is multiplied by the input, provides the output of the neuron. The weight value will vary in the course of the training procedure.
- The *output* is given by the product $\text{input} \times \text{weight}$. The neuron will learn when the given output will be issued next to the expected value.
- These elements are enough to define the *model*. The input value represents a stimulus to the neuron. It is a constant which is defined with the TensorFlow operator, that is, `tf.constant`.

We define the `input_value`, which is `0.5` a floating point value:

```
| input_value = tf.constant(0.5, name="input_value")
```

The *weight* is the input that we pass to the individual neuron; it will vary during the training phase of the network, so it will be identified by a TensorFlow variable:

```
| weight = tf.Variable(1.0, name="weight")
```

The starting weight is `1.0` floating point.

The *expected value* is the value that we expect to get as output after the learning phase of the network. It will be a TensorFlow constant:

```
| expected_output = tf.constant(0.0, name="expected_output")
```

The *expected value* that we impose is equal to `0.0`.

The *model* or the output we want to compute is the following product is `weight × input`.

```
| model = tf.mul(input_value, weight,"model")
| mul_ = tf.multiply(constant_A,constant_C)
```

Finally, we defined the neuron's input and output, but a fundamental aspect is still missing, we must tell to the neuron, *how it must learn*, as we saw in [Chapter 1, Getting Started with Deep Learning](#), the learning phase requires the definition of two key elements.

The first defines a *metric*; it is a measure of how the value obtained differs from the expected value. It is the so-called *loss function*. Formally, it is the square of the difference between the current output by the model and the expected value:

```
| loss_function = (model - expected_value)**2
```

The *loss function* is not enough; we must find a way to *optimize* it or to minimize its value during the training phase of the neuron. TensorFlow has several optimization functions, in this example, we use *gradient descent* (discussed in [Chapter 1, Getting Started with Deep Learning](#)) that is the most common optimization function. The TensorFlow function for gradient optimization is

```
tf.train.GradientDescentOptimizer.
```

```
| optim = tf.train.GradientDescentOptimizer(learning_rate=0.025)
```

The argument of this function is the `learning_rate` variable, that we have imposed equal to `0.025`.

We have defined all we need to build our model for a single neuron input.

However, the example in question also wants to show how to use in TensorBoard practice. The definition of *summaries* consists of a preliminary setup phase, which defines which parameters we want to display.

```
| for value in [input,weight,expected_output,output]:
|     tf.scalar_summary(value.op.name,value)
```

Each value to display is passed to `tf.scalar_summary` function. It provides the following two arguments:

- `value.op.name`: This is a tag for the summary
- `value`: A real numeric tensor. This is a value for the summary

The `tf.scalar_summary` variable outputs a `summary` protocol buffer with scalar values. Then we merge all summaries collected in the computation graph:

```
| summaries = tf.merge_all_summaries()
```

To run the graph, we have to create the `Session`:

```
| sess = tf.Session()
```

Now we create our `SummaryWriter` that writes the summary protocol buffers to the `filelog_simple_stats` event:

```
| summary_writer = tf.summary.FileWriter('log_simple_stats',sess.graph)
```

The `SummaryWriter` class provides a mechanism to create an event file in each directory and add summaries and events to it. The class updates the file contents asynchronously. This allows a training program to call methods to add data to the file directly from the training loop, without slowing down training. Finally, we can run the model:

```
| sess.run(tf.global_variables_initializer())
```

We perform `100` simulation steps, for each of which we monitor the parameters defined in the `summary_writer` parameter:

```
| for i in range(100):
|     Summary_writer.add_summary(sess.run(summaries),i)
|     Sess.run(train_step)
```

After running the code, we can see the log file created with TensorBoard. Running the TensorBoard is very simple; open a terminal and digit the following:

```
| $tensorboard --logdir= log_simple_stats
```

In the same directory where the `log_simple_stats` file is created, if all is going well a message like this should appear:

```
| starting tensorboard on port 6006
```

It means that to use TensorBoard you must open a web browser to the following address, `localhost:6006`.

The TensorBoard's opening page should display something like the following figure:

TensorBoard

EVENTS

IMAGES

GRAPH

HISTOGRAMS

Regex filter



- Split on underscores
- Data download links

Horizontal Axis

STEP

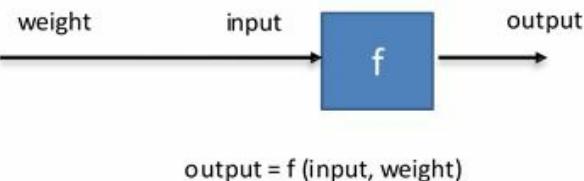
RELATIVE

WALL

Runs



TOGGLE ALL RUNS

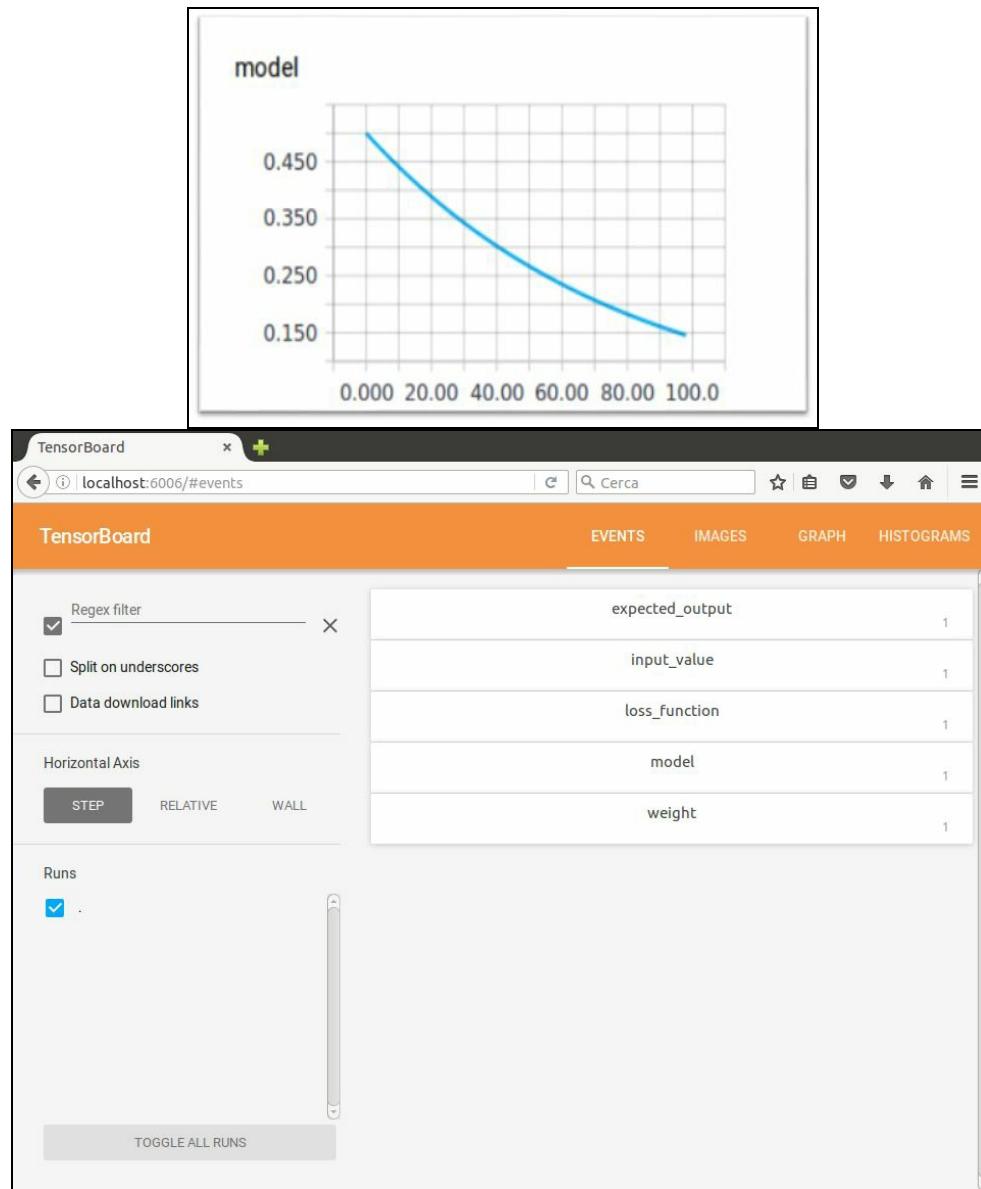


TensorBoard opening page

You can visualize the events data as well. For example, the following data objects can be visualized as follows:

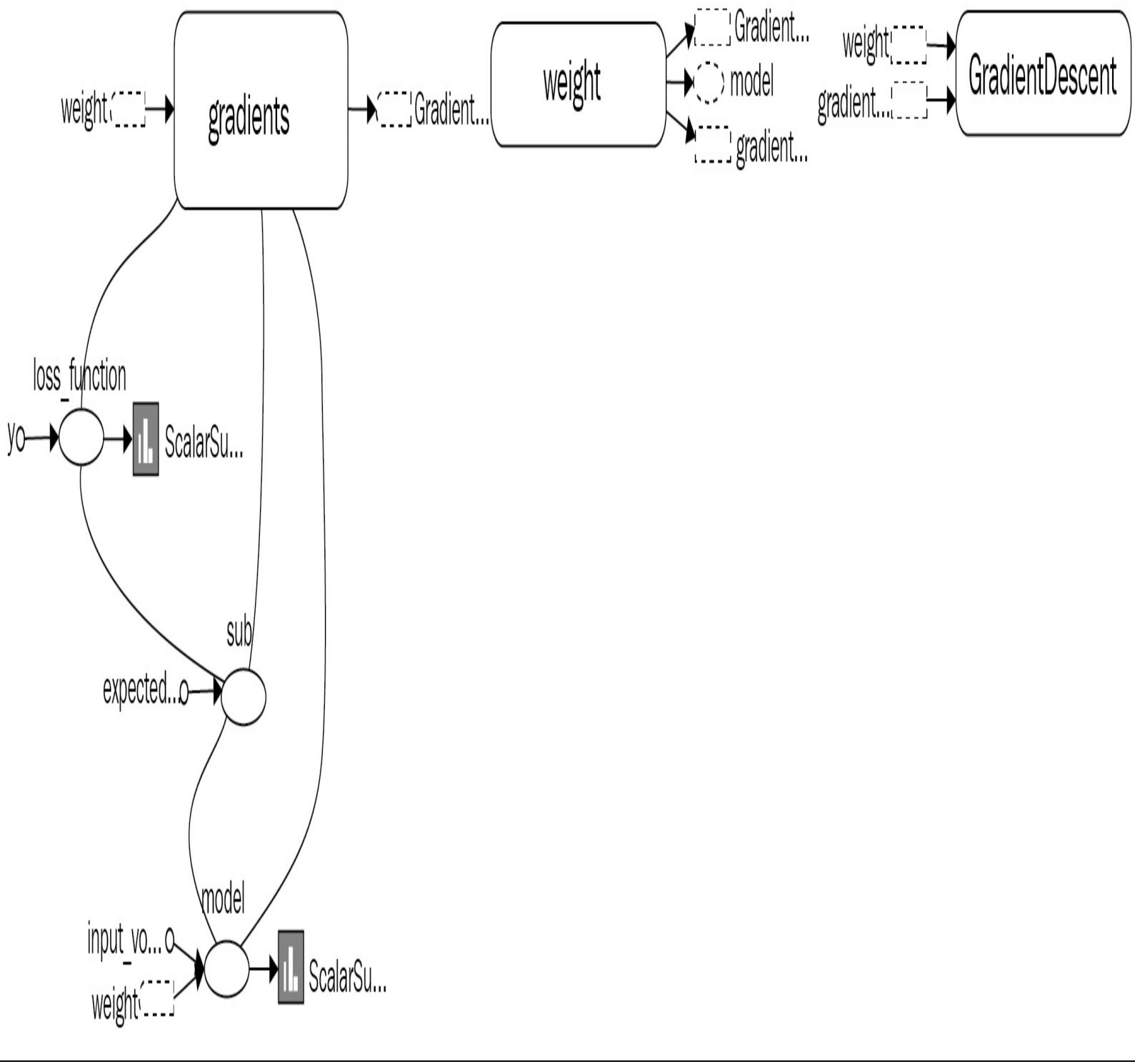
- expected_output
- input_value
- loss_function
- model
- weight

The following figure shows the neuron's output against the training iteration number.



TensorBoard visualization of the model output

Click on the GRAPH tab you'll be able to see the *computation graph* with the *auxiliary nodes*:



The computation graph for the single input neuron

You can see the `model` and `loss_function` data in the Events tab. If you want to display several runs with different values, for example changing the `expected_value` or the `input_value` in your model, you need to run the script several times and save the log files into separate folders, then you can then start TensorBoard reading from several folders.

Source code for the single input neuron

We reported for the entire source code for the example previously described is as follows:

```
import tensorflow as tf

input_value = tf.constant(0.5, name="input_value")
weight = tf.Variable(1.0, name="weight")
expected_output = tf.constant(0.0, name="expected_output")
model = tf.multiply(input_value, weight, "model")
loss_function = tf.pow(expected_output - model, 2, name="loss_func")
optimizer = tf.train.GradientDescentOptimizer(0.025).minimize(loss_function)

for value in [input_value, weight, expected_output, model, loss_function]:
    tf.summary.scalar(value.op.name, value)
summaries = tf.summary.merge_all()
sess = tf.Session()
summary_writer = tf.summary.FileWriter('log_simple_stats', sess.graph)
sess.run(tf.global_variables_initializer())
for i in range(100):
    summary_writer.add_summary(sess.run(summaries), i)
    sess.run(optimizer)
```


Migrating to TensorFlow 1.x

The latest release of TensorFlow 1.0 has left in ways such that all the computing and using reusing the previous codes that all of them are not backward compatible. This means an application developed on TensorFlow 0.x won't necessarily work on TensorFlow 1.x directly. Now to upgrade 0.x codes to 1.x compatible, there are two ways using the upgrade script or manually.

How to upgrade using the script

The `tf_upgrade.py` script helps you to upgrade your 0.x source codes to 1.x (or higher) compatible. This script can be downloaded from GitHub at <https://github.com/tensorflow/tensorflow/tree/master/tensorflow/tools/compatibility>. To convert a single 0.x TensorFlow source file to 1.x (or higher) issue a command in following format:

For Python 2.7 type the following command:

```
| $ python tf_upgrade.py --infile InputFile --outfile OutputFile
```

For Python 3.3+n type the following command:

```
| $ python3 tf_upgrade.py --infile InputFile --outfile OutputFile
```

For example, suppose you have a script named `five_layers_relu.py` which is 0.x compatible written on Python 2.7 with the following code:

```
import mnist_data
import tensorflow as tf
import math
logs_path = 'log_simple_stats_5_layers_relu_softmax'
batch_size = 100
learning_rate = 0.5
training_epochs = 10
mnist = mnist_data.read_data_sets("data")
X = tf.placeholder(tf.float32, [None, 28, 28, 1])
Y_ = tf.placeholder(tf.float32, [None, 10])
lr = tf.placeholder(tf.float32)
L = 200
M = 100
N = 60
O = 30
W1 = tf.Variable(tf.truncated_normal([784, L], stddev=0.1))
B1 = tf.Variable(tf.ones([L])/10)
W2 = tf.Variable(tf.truncated_normal([L, M], stddev=0.1))
B2 = tf.Variable(tf.ones([M])/10)
W3 = tf.Variable(tf.truncated_normal([M, N], stddev=0.1))
B3 = tf.Variable(tf.ones([N])/10)
W4 = tf.Variable(tf.truncated_normal([N, O], stddev=0.1))
B4 = tf.Variable(tf.ones([O])/10)
W5 = tf.Variable(tf.truncated_normal([O, 10], stddev=0.1))
B5 = tf.Variable(tf.zeros([10]))

XX = tf.reshape(X, [-1, 784])
Y1 = tf.nn.relu(tf.matmul(XX, W1) + B1)
Y2 = tf.nn.relu(tf.matmul(Y1, W2) + B2)
Y3 = tf.nn.relu(tf.matmul(Y2, W3) + B3)
Y4 = tf.nn.relu(tf.matmul(Y3, W4) + B4)
Ylogits = tf.matmul(Y4, W5) + B5
Y = tf.nn.softmax(Ylogits)
cross_entropy = tf.nn.softmax_cross_entropy_with_logits(Ylogits, Y_)
cross_entropy = tf.reduce_mean(cross_entropy)*100

correct_prediction = tf.equal(tf.argmax(Y, 1), tf.argmax(Y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
train_step = tf.train.AdamOptimizer(lr).minimize(cross_entropy)
tf.scalar_summary("cost", cross_entropy)
tf.scalar_summary("accuracy", accuracy)
summary_op = tf.merge_all_summaries()
init = tf.initialize_all_variables()
sess = tf.Session()
```

```

sess.run(init)
with tf.Session() as sess:
    sess.run(tf.initialize_all_variables())
    writer = tf.train.SummaryWriter(logs_path,
                                    graph=tf.get_default_graph())
for epoch in range(training_epochs):
    batch_count = int(mnist.train.num_examples/batch_size)
    for i in range(batch_count):
        batch_x, batch_y = mnist.train.next_batch(batch_size)
        max_learning_rate = 0.003
        min_learning_rate = 0.0001
        decay_speed = 2000
        learning_rate = min_learning_rate+
                        (max_learning_rate - min_learning_rate)
                        * math.exp(-i/decay_speed)
        _, summary = sess.run([train_step, summary_op],
                             {X: batch_x, Y: batch_y, lr: learning_rate})
        writer.add_summary(summary, epoch)
    #if epoch % 2 == 0:
    #    print "Epoch: ", epoch
print "Accuracy: ", accuracy.eval(feed_dict={X: mnist.test.images, Y: mnist.test.labels})
print "done"

```

Now the following command converts a 0.x TensorFlow program:

```
| five_layers_relu.py to a 1.x TensorFlow program named five_layers_relu_1.py:
$ python tf_upgrade.py --infile five_layers_relu.py --outfile five_layers_relu_1.py
```

If there's no compile error, in the file, the `tf_upgrade.py` script will generate a file named `report.txt` where the script is located and of course the upgraded script (that is, `five_layers_relu_1.py`) will be saved in the current working directory. This text file, shows changes have been performed and makes additional suggestions about changes you might need to make manually. Now let's see the contents of the `report.txt` files:

```
| $ cat report.txt
```

```
asif@ubuntu:~$ cat report.txt
-----
Processing file 'five_layers_relu.py'
outputting to 'five_layers_relu_1.py'
-----
'five_layers_relu.py' Line 64
-----
Renamed function 'tf.initialize_all_variables' to 'tf.global_variables_initializer'
  Old:     sess.run(tf.initialize_all_variables())
  New:     sess.run(tf.global_variables_initializer())

'five_layers_relu.py' Line 65
-----
Renamed function 'tf.train.SummaryWriter' to 'tf.summary.FileWriter'
  Old:     writer = tf.train.SummaryWriter(logs_path, \
  New:     writer = tf.summary.FileWriter(logs_path, \

'five_layers_relu.py' Line 45
-----
Added keyword 'logits' to reordered function 'tf.nn.softmax_cross_entropy_with_logits'
Added keyword 'labels' to reordered function 'tf.nn.softmax_cross_entropy_with_logits'
  Old: cross_entropy = tf.nn.softmax_cross_entropy_with_logits(Ylogits, Y_)
  New: cross_entropy = tf.nn.softmax_cross_entropy_with_logits(logits=Ylogits, labels=Y_)

'five_layers_relu.py' Line 55
-----
Renamed function 'tf.scalar_summary' to 'tf.summary.scalar'
  Old: tf.scalar_summary("cost", cross_entropy)
  New: tf.summary.scalar("cost", cross_entropy)

'five_layers_relu.py' Line 56
-----
Renamed function 'tf.scalar_summary' to 'tf.summary.scalar'
  Old: tf.scalar_summary("accuracy", accuracy)
  New: tf.summary.scalar("accuracy", accuracy)

'five_layers_relu.py' Line 59
-----
Renamed function 'tf.merge_all_summaries' to 'tf.summary.merge_all'
  Old: summary_op = tf.merge_all_summaries()
  New: summary_op = tf.summary.merge_all()

'five_layers_relu.py' Line 59
-----
Renamed function 'tf.initialize_all_variables' to 'tf.global_variables_initializer'
  Old: init = tf.initialize_all_variables()
  New: init = tf.global_variables_initializer()
```

Changes made by the `tf_upgrade.py` script written on the `report.txt`

Thus, based on the preceding suggestion, if you see the contents of the resulting `five_layers_relu_1.py` file, you should observe the changes as follows (highlighted in red):

```

cross_entropy = tf.nn.softmax_cross_entropy_with_logits(logits=Ylogits, labels=Y_)
cross_entropy = tf.reduce_mean(cross_entropy)*100

correct_prediction = tf.equal(tf.argmax(Y, 1), tf.argmax(Y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

train_step = tf.train.AdamOptimizer(lr).minimize(cross_entropy)

tf.summary.scalar("cost", cross_entropy)
tf.summary.scalar("accuracy", accuracy)
summary_op = tf.summary.merge_all()

init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    writer = tf.summary.FileWriter(logs_path,
                                    graph=tf.get_default_graph())
    for epoch in range(training_epochs):
        batch_count = int(mnist.train.num_examples/batch_size)
        for i in range(batch_count):
            batch_x, batch_y = mnist.train.next_batch(batch_size)
            max_learning_rate = 0.003
            min_learning_rate = 0.0001
            decay_speed = 2000
            learning_rate = min_learning_rate+\n                (max_learning_rate - min_learning_rate)\\
                    * math.exp(-i/decay_speed)
            _, summary = sess.run([train_step, summary_op], \\
                {X: batch_x, Y_: batch_y,\n                 lr: learning_rate})
            writer.add_summary(summary,\n                epoch * batch_count + i)
        #if epoch % 2 == 0:
        #    print "Epoch: ", epoch
        print "Accuracy: ", accuracy.eval(\n            feed_dict={X: mnist.test.images, Y_: mnist.test.labels})
    print "done"

```

Changes made on the upgrade file

For Python 3.3+, you will have to change the print statements in the `sample.py` file and run the upgrade script.

Now to upgrade a whole directory containing source codes of 0.x compatible to 1.x, issue a command in the following format:

```
| $ python tf_upgrade.py --intree InputDir --outtree OutputDir
```


Limitations

The `tf_upgrade.py` script has some limitations. For example, you must manually fix any instances of `tf.reverse()`. This is not that hard since the `tf_upgrade.py` script will warn you about the `tf.reverse()` in the `report.txt` file. For the reordered arguments, `tf_upgrade.py` tries to minimally reformat your code. However, the actual argument order cannot be changed automatically. For another example, constructions such as `tf.get_variable_scope().reuse_variables()` will likely not work. Therefore, we recommend deleting those lines and replacing them with the following:

```
tf.variable_scope(tf.get_variable_scope(), reuse=True).
```


Upgrading code manually

As described in the preceding section, there are many limitations on the `tf_upgrade.py` script. Therefore, instead of running the script, you can also manually upgrade your codes. In this section, we will provide a comprehensive list of all backward-incompatible changes made in TensorFlow 1.x.

For the following cases (and if you don't use the upgrade script), you will have to make manual updates in your code. These suggestions were adapted from the TensorFlow website <https://www.tensorflow.org/install/migration>.

Variables

In the latest TensorFlow release, the variable functions have been made more consistent and less confusing. The following are some upgrade suggestions that you need to apply manually:

- `tf.VARIABLES` should be renamed to `tf.GLOBAL_VARIABLES`
- `tf.all_variables` should be renamed to `tf.global_variables`
- `tf.initialize_all_variables` should be renamed to `tf.global_variables_initializer`
- `tf.initialize_local_variables` should be renamed to `tf.local_variables_initializer`
- `tf.initialize_variables` should be renamed to `tf.variables_initializer`

Summary functions

In the latest TensorFlow release, summary functions have been consolidated under the `tf.summary` namespace. The following are some upgrade suggestions that you need to apply manually:

```
tf.audio_summary should be renamed to tf.summary.audio
tf.contrib.deprecated.histogram_summary should be renamed to tf.summary.histogram
tf.contrib.deprecated.scalar_summary should be renamed to tf.summary.scalar
tf.histogram_summary should be renamed to tf.summary.histogram
tf.image_summary should be renamed to tf.summary.image
tf.merge_all_summaries should be renamed to tf.summary.merge_all
tf.merge_summary should be renamed to tf.summary.merge
tf.scalar_summary should be renamed to tf.summary.scalar
tf.train.SummaryWriter should be renamed to tf.summary.FileWriter
```


Simplified mathematical variants

In the latest TensorFlow release, batched versions of math operations have been removed. Now the functionality is contained in the non-batched versions. The following are some upgrades suggestions that you need to apply manually:

```
tf.batch_band_part should be renamed to tf.band_part
tf.batch_cholesky should be renamed to tf.cholesky
tf.batch_cholesky_solve should be renamed to tf.cholesky_solve
tf.batch_fft should be renamed to tf.fft
tf.batch_fft3d should be renamed to tf.fft3d
tf.batch_ifft should be renamed to tf.ifft
tf.batch_ifft2d should be renamed to tf.ifft2d
tf.batch_ifft3d should be renamed to tf.ifft3d
tf.batch_matmul should be renamed to tf.matmul
tf.batch_matrix_determinant should be renamed to tf.matrix_determinant
tf.batch_matrix_diag should be renamed to tf.matrix_diag
tf.batch_matrix_inverse should be renamed to tf.matrix_inverse
tf.batch_matrix_solve should be renamed to tf.matrix_solve
tf.batch_matrix_solve_ls should be renamed to tf.matrix_solve_ls
tf.batch_matrix_transpose should be renamed to tf.matrix_transpose
tf.batch_matrix_triangular_solve should be renamed to tf.matrix_triangular_solve
tf.batch_self_adjoint_eig should be renamed to tf.self_adjoint_eig
tf.batch_self_adjoint_eigvals should be renamed to tf.self_adjoint_eigvals
tf.batch_set_diag should be renamed to tf.set_diag
tf.batch_svd should be renamed to tf.svd
tf.complex_abs should be renamed to tf.abs
```


Miscellaneous changes

In the latest TensorFlow release, several other changes have been made. The following are some upgrades suggestions that you need to apply manually:

```
tf.image.per_image_whitening should be renamed to tf.image.per_image_standardization  
tf.nn.sigmoid_cross_entropy_with_logits arguments have been reordered to tf.nn.sigmoid_cross_entropy_with_logit  
tf.nn.softmax_cross_entropy_with_logits arguments have been reordered to tf.nn.softmax_cross_entropy_with_logit  
tf.nn.sparse_softmax_cross_entropy_with_logits arguments have been reordered to tf.nn.sparse_softmax_cross_entropy_with_logits  
tf.ones_initializer should be changed to a function call i.e. tf.ones_initializer()  
tf.pack should be renamed to tf.stack  
tf.round The semantics of tf.round now match Banker's rounding.  
tf.unpack should be renamed to tf.unstack  
tf.zeros_initializer should be changed to a function call i.e. tf.zeros_initializer()
```

This is more or less the end of the source code migration process to TensorFlow 1.x.

However, these are not all the suggested changes in the latest version of TensorFlow. Thus, for more information about the upgrade procedure, please refer to this URL: <https://www.tensorflow.org/install/migration>.

Summary

TensorFlow is designed to make distributed machine and deep learning easy for everyone, but using it does require understanding some general principles and algorithms. Furthermore, the latest release of TensorFlow comes with lots of exciting features. Thus we also tried to cover them so that you can use them with ease. We have shown how to install TensorFlow on different platforms including Linux, Windows, and Mac OS. Before, covering this in even greater depth, we showed some example of how to upgrade the source code from the previous version of TensorFlow to the latest version 1.x.

In summary, here is a brief recap of the key concepts of TensorFlow explained in this chapter:

- **Graph:** A TensorFlow computation is represented as dataflow graph. Each graph is built as a set of operations (http://www.tensorflow.org/api_docs/python/framework.html#Operation) objects.
- **Operation:** Each Operation object represents a graph node, which is a unit of computation (addition, multiplication, or something more complex), performed on a tensor flow. It takes tensors as input and produces a tensor as output.
- **Tensor:** Tensors can be represented as the edges of a data flow graph. They do not represent or hold any value produced by an application of an operation but instead define the type of this value and the means by which this value should be calculated during the session.
- **Session:** This is an entity that represents an environment for running calculations on the data flow graph.

In a later section of the chapter, we introduced Tensorboard, which is a powerful tool for analyzing and debugging neural network models. The chapter ended with an example that shows how to implement a simple neuron model and how to analyze its learning phase with TensorBoard.

Finally, we showed the source code migration process from TensorFlow v0.x to v1.x.

In the next chapter, we will introduce *feed-forward neural networks*. After examining the theoretical features, we'll show how to train and evaluate this type of neural network on image classification problems. Some practical examples using feed-forward neural networks will be shown to facilitate understanding.

Using TensorFlow on a Feed-Forward Neural Network

Neural network's architectures can be very different; these configurations are often organized on different layers, the first of which receives the input signals and the last returns the output signals. Usually these networks are identified as feed-forward neural networks.

Feed-forward neural networks, which we intend to illustrate briefly, are well suited to be used for the approximation of functions and for the interpolation.

The following topics are covered in this chapter:

- Introducing feed-forward neural network
- Classification of handwritten digits
- Exploring the MNIST dataset
- Softmax classifier
- How to save and restore a TensorFlow model
- Implementing a five-layer neural network
- ReLU classifier
- Dropout optimization

Introducing feed-forward neural networks

A **feed-forward neural network (ffnn)** consists of a large number of neurons, organized in layers: one input layer, one or more hidden layers, and one output layer. Each neuron is connected to all the neurons of the previous layer; the connections are not all the same, because they have a different weight. The weights of these connections encode the knowledge of the network.

Data enters at the inputs and passes through the network, layer by layer, until it arrives at the outputs; during this operation there is no feedback between layers.

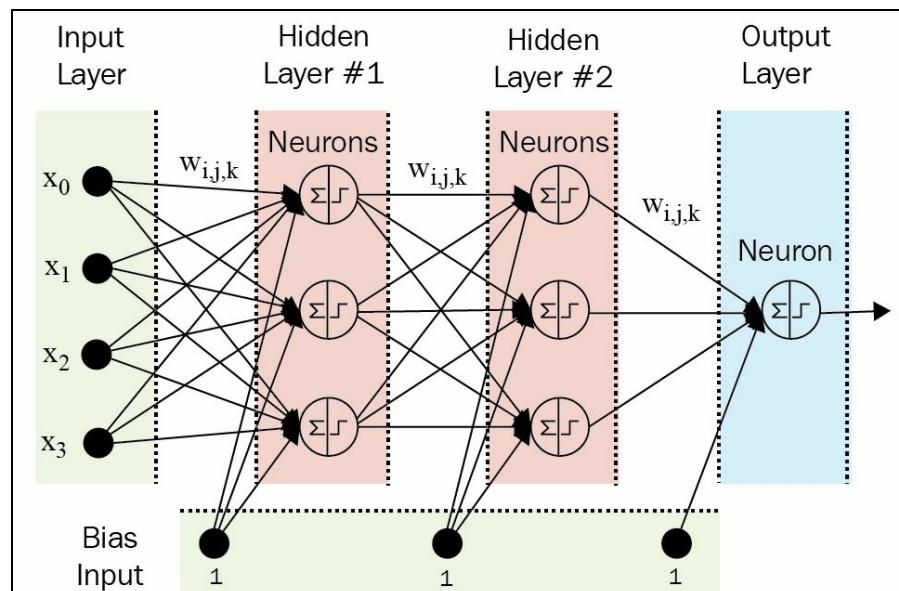
Therefore, these networks are called feed-forward neural networks.

An ffnn with enough neurons in the hidden layer is able to approximate with arbitrary precision:

- Any continuous function, with one hidden layer
- Any function, even discontinuous, with two hidden layers

However, it is not possible to determine a priori, with adequate precision, the required number of hidden layers, or even the number of neurons that must be contained inside it to compute a non-linear function. Still, despite some rules of thumb, it relies on the experience and on some heuristics to determine the structure of the network.

If the neural network architecture is constituted by a low number of hidden layers or neurons, the network is not able to approximate with adequate precision the unknown function, because this is too complex or because the backpropagation algorithm falls within a local minimum. If the network is composed of a high number of hidden layers, we have an over-fitting problem, namely a worsening of the network's generalization ability.



Feedforward neural network with two hidden layers and an input bias

Feed-forward and backpropagation

The backpropagation algorithm aims to minimize the error between the current and desired output. Since the network feed-forward, the activation flow always proceeds forward from the input units to the output units. When compared with the output from the one expected, the gradient of the cost function is backpropagated through the modification of weights.

This method is recursive and can be applied to any number of hidden layers.

The backpropagation algorithm processes the information in such a way that the network decreases the global error during the learning iterations; however, this does not guarantee that the global minimum is reached. The presence of the hidden units and the non-linearity of the output function means that the behavior of the error is very complex and has many local minimal. The backpropagation algorithm can therefore stop at a local minimum, providing a suboptimal solution. Normally the error always decreases on the training set (that improves the ability to represent the input-output relationship between the data supplied) because the network is learning, while on the testing set (which measures the predictive capabilities), from a certain value then, it can grow due to the over-fitting problem: the resulting network (or model) will have a high classification accuracy for the training samples and a low classification accuracy for unknown samples.

Weights and biases

Besides considering the state of the neuron and the way it is linked to others, we should consider the synaptic weight, which is precisely, the influence of that connection within the network. Each weight has a numerical value indicated by W_{ij} , which is the synaptic weight connecting the neuron i to neuron j .

Depending on the point where a neuron is located, it will always have one or more links, which correspond to relative synaptic weights.

The weights and output function determine the behavior of an individual neuron and the network in general.

They should be correctly changed during the training phase, to ensure the correct behavior of the model.

For each unit, i is defined an input vector $x_i = (x_1, x_2, \dots, x_n)$ and a weight vector $w_i = (w_{i1}, w_{i2}, \dots, w_{in})$, and the neuron performs a weighted sum of the inputs:

$$\text{net}_i = \sum_j w_{ij} x_j \dots \text{(a)}$$

Among the weights, there is one special, called *bias*. It is not tied to any other unit of the network and it is considered to have input equal to 1. This expedient allows for establishing a kind of *reference point* or threshold for neurons, and formally, the bias performs a translation along the abscissa axis to the output function. The previous formula will become as follows:

$$\text{net}_i = \sum_j w_{ij} x_j + b_i \dots \text{(b)}$$

Transfer functions

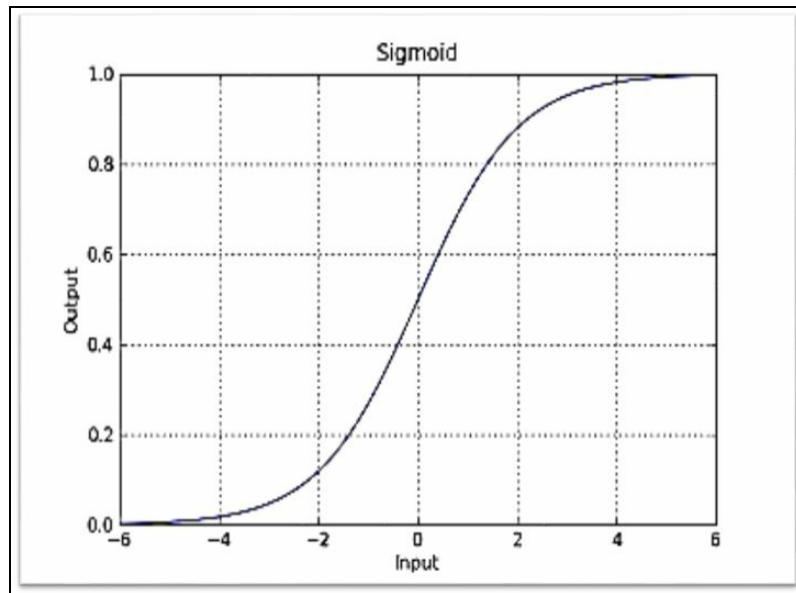
Each neuron receives as input signal the weighted sum of the synaptic weights and the activation values of the neurons connected to it. To allow the neuron to calculate its activation value, that is, what the neuron retransmits, the weighted sum must be passed as the argument of the transfer function. The transfer function allows the receiving neuron to transmit the received signal modifying it.

One of the most used functions for this purpose is the so-called sigmoid function:

$$out_i = \frac{1}{1 + e^{-net_i}}$$

The domain of this function includes all real numbers and the co-domain is $(0, 1)$. This means that any value will obtain, as output from a neuron per the calculation of its activation state, it will always be between 0 and 1.

The sigmoid function, as represented in the following diagram, provides an interpretation of the saturation rate of a neuron, from not being active ($= 0$), to its complete saturation, which occurs at a predetermined maximum value ($=1$):



Sigmoid function

When new data has to be analyzed, it is loaded by the input layer, which through (a) or (b) generates an output. This result, together with the output from neurons of the same layer, will form a new input to the neurons on the next layers. The process will be iterated until the last layer. In general, in the last level of an ffn the softmax function is adopted. It comes in handy whenever we want to interpret the output of a network as a posteriori probability.

The softmax function is denoted as follows:

$$out_i = \frac{e^{net_i}}{\sum_{j=1}^N e^{net_j}}$$

Here, N represents the total number of outputs from net.

Also, the following important properties are *valid* for the softmax function:

$$0 \leq out_i \leq 1 \text{ con } \sum_i out_i = 1$$

Classification of handwritten digits

Automatic recognition of handwritten digits is an important problem, which can be found in many practical applications. In this chapter, we will implement some feed-forward networks to address this problem.

To train and test the implemented models we use the MNIST database of handwritten digits.

The MNIST dataset is made of a training set of 60,000 examples, plus a test set of 10000 examples. An example of the data, as it is stored in the files of the examples, is shown in the following figure:

0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9

Example of data extracted from the MNIST database

The source images were originally in black and white, but later, to normalize them to the size of 20×20 pixels, intermediate brightness levels have been introduced, due to the effect of the anti-aliasing filter for resizing. Subsequently, the images were focused in the center of mass of the pixels, in an area of 28×28 pixels, in order to improve the learning process.

The entire database is stored in four files:

Training database:

```
| train-images-idx3-ubyte.gz: training set images (9912422 bytes)
| train-labels-idx1-ubyte.gz: training set labels (28881 bytes)
```

Testing database:

```
| t10k-images-idx3-ubyte.gz: test set images (1648877 bytes)
| t10k-labels-idx1-ubyte.gz: test set labels (4542 bytes)
```

Each database consists of two files; the first contains the images, while the second contains the respective labels.

Exploring the MNIST dataset

Let's show a short example of how to access the MNIST data and how to display a selected image.

We import the following libraries:

The `numpy` library because we have to do some image manipulation:

```
| >>import numpy as np
```

The `pypplot` function in `matplotlib` for drawing the images:

```
| >>import matplotlib.pyplot as plt
```

Finally, the `mnist_data` library, which you can download from the code repository of this book. It is a Google script that allows us to download the MNIST database and to build the dataset:

```
| >>import mnist_data
```

Then we load the dataset using the `read_data_sets` method:

```
| >>_input = mnist_data.read_data_sets("data")
```

The `data` is the name of the directory where the images will be uploaded.

The shape of the images and labels:

```
>>_input.train.images.shape  
(60000, 28, 28, 1)  
  
>>_input.train.labels.shape  
(60000, 10)  
  
>>_input.test.images.shape  
(10000, 28, 28, 1)  
  
>>_input.test.labels.shape  
(10000, 10)
```

Using the Python library, `matplotlib`, we want to visualize a single digit:

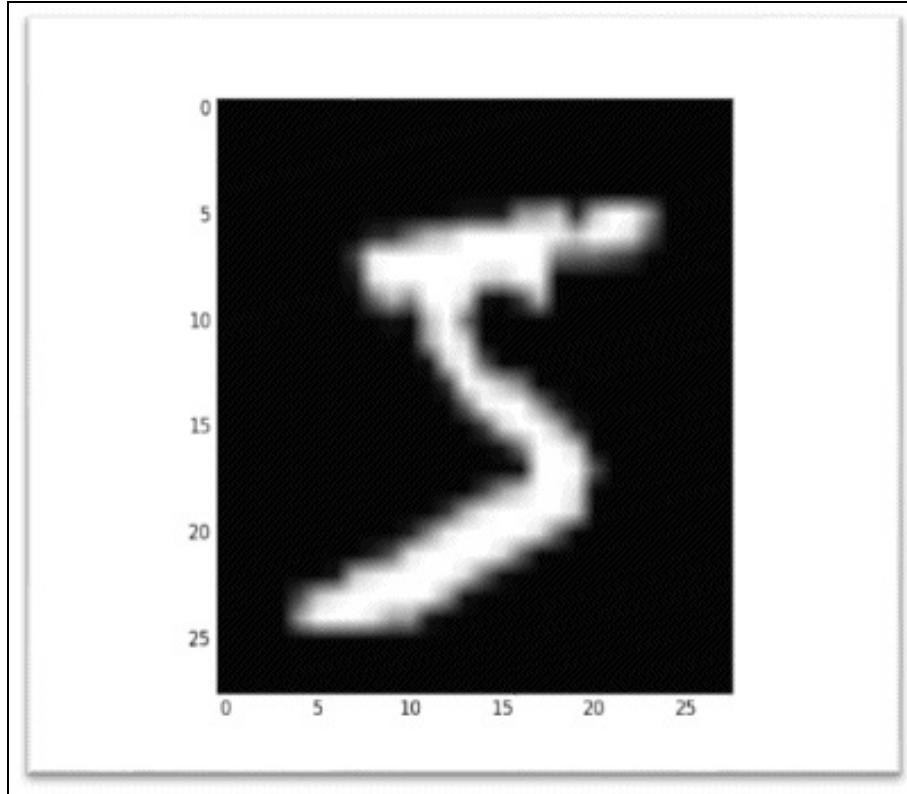
```
>>image_0 = _input.train.images[0]  
>>image_0 = np.resize(image_0, (28,28))  
  
>>label_0 = _input.train.labels[0]  
label set = [ 0.  0.  0.  0.  0.  1.  0.  0.  0.  0.]
```

The number `1` is the sixth position of the array. It means that the figure for our image is the digit `5`.

Finally, we verify that the digit is really `5`.

We used the imported `plt` function, to draw the `image_0` tensor:

```
| >>plt.imshow(image_0, cmap='Greys_r')  
| >>plt.show()
```

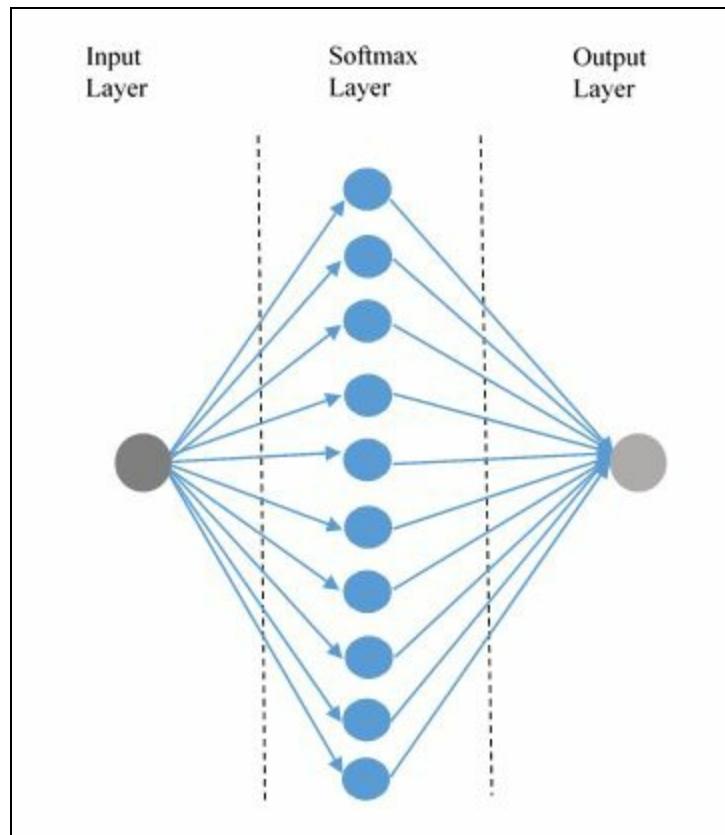


The extracted image from the MNIST dataset

Softmax classifier

In the previous section, we showed how to access and manipulate the MNIST dataset. In this section, we will see how to address the classification problem of handwritten digits via the TensorFlow library.

We'll apply the concepts taught to build more models of neural networks in order to assess and compare the results of the different approaches followed. The first feed-forward network architecture that will be implemented is represented in the following figure:



The softmax neural network architecture

The hidden layer (or softmax layer) of the network consists of 10 neurons, with a softmax transfer function. Remember that it is defined so that its activation is a set of positive values with total sum equal to 1; this means that the j^{th} value of the output is the probability that j is the class that corresponds with the network input.

Let's see how to implement our neural network model.

The first thing to do is import the necessary libraries and prepare the data for our model:

```
import tensorflow as tf
import mnist_data

logs_path = 'log_simple_stats_softmax'
batch_size = 100
learning_rate = 0.5
training_epochs = 10
```

```
| mnist = mnist_data.read_data_sets("data")
```

Now let's move on to defining the network model.

The input network consists of a set of images extracted from the MNIST datasets; each image has a size of 28x28 pixels:

```
| x = tf.placeholder(tf.float32, [None, 28, 28, 1], name="input")
```

The problem is assigning a probability value for each of the possible classes of membership, that is, the digits from 0 to 9. The corresponding output describes a probability distribution from which we can provide a prediction about the value tested. The output network will be contained in the following placeholder, a tensor of 10 elements:

```
| y_ = tf.placeholder(tf.float32, [None, 10])
```

The weights take into account the size of the hidden layer (10 neurons) and the input size. The value of the weights must vary at each iteration of the calculation, for which they are defined by the following variable:

```
| w = tf.Variable(tf.zeros([784, 10]))
```

The matrix weight is $W[784, 10]$ where $784 = 28 \times 28$.

Let's flatten the images into a single line of pixels; the number -1 in the shape definition means the only possible dimension that will preserve the number of elements:

```
| xx = tf.reshape(x, [-1, 784])
```

In a similar way, we define the biases of the network, the effect of which is to control the translatory motion of the trigger relative to the origin of the input signals. Formally, the bias has a role, not different from the weights, which act as an intensity regulator of the emitted/received signals.

The bias tensor is therefore a variable tensor:

```
| b = tf.Variable(tf.zeros([10]))
```

Also the size (=10) is equal to the total number of neurons of the hidden layer.

The *input*, *weight*, and *bias* tensors are sized to define the `evidence` parameter that quantifies if a certain image belongs to a particular class:

```
| evidence = tf.matmul(xx, w) + b
```

The neural network has only one hidden layer, composed of 10 neurons. Using the feed-forward network definition we know that all neurons at the same level must have the same activation function.

In this model, the activation function is the softmax function, it converts the `evidence` into *probabilities*

of belonging to each of the 10 *possible* classes:

```
| Y = tf.nn.softmax(evidence, name="output")
```

The Y output matrix will be formed of 100 rows and 10 columns.

To train our model and to determine if we have a good one, we must define a *metric*. Indeed, the next goal is to get values for the W and b tensors that minimize the value of a metric and indicate how bad the implemented model is.

Different metrics calculate the degree of error between the desired output and the computed outputs obtained from the training data. The most common error measure is the mean squared error; however, there is some research out there that suggests using other metrics to a neural network like this.

In this example, we use the so-called `cross_entropy` error function. It is defined as follows:

```
| cross_entropy = -tf.reduce_mean(Y_ * tf.log(Y)) * 1000.0
```

We minimize the error function using the gradient descent algorithm:

```
| train_step = tf.train.GradientDescentOptimizer(0.005).  
|           \minimize(cross_entropy)
```

Here, we have set the learning rate equal to 0.005.

We will have a correct prediction if the Y network output and the desired output $Y_{\text{are equal}}$:

```
| correct_prediction = tf.equal(tf.argmax(Y, 1),  
|                               tf.argmax(Y_, 1))
```

The `correct_prediction` variable allows defining the accuracy of the implemented model:

```
| accuracy = tf.reduce_mean(tf.cast(correct_prediction,  
|                           tf.float32))
```

We define the summaries that we want to analyze with TensorBoard:

```
| tf.summary.scalar("cost", cross_entropy)  
| tf.summary.scalar("accuracy", accuracy)  
| summary_op = tf.summary.merge_all()
```

Finally, implemented model is, we must build a *session* that will take place in the training and testing step:

```
| with tf.Session() as sess:  
|   sess.run(tf.global_variables_initializer())  
|   writer = tf.summary.FileWriter(logs_path, \  
|                                graph=tf.get_default_graph())
```

The network's training procedure is iterative. It slightly modifies for each learning cycle (or *epoch*) the synaptic weights by using a selected subset (or *batch set*):

```
| for epoch in range(training_epochs):
```

It slightly modifies for each learning cycle or epoch the synaptic weights by using a selected set:

```
| batch_count = int(mnist.train.num_examples/batch_size)
```

The selected sets are respectively `batch_x` e `batch_y`:

```
| for i in range(batch_count):
|     batch_x, batch_y = mnist.train.next_batch(batch_size)
```

They will be used by the `feed_dict` statement to feed the network during the training procedure.

At each cycle:

- The weights are modified to minimize the error function.
- The results are added to the summaries using the following `writer.add_summary` statement:

```
| _, summary = sess.run([train_step, summary_op], \
|                     feed_dict={X: batch_x, \
|                               Y_: batch_y})
| writer.add_summary(summary, \
|                     epoch * batch_count + i)
| print "Epoch: ", epoch
```

Finally, we can test the model and evaluate its `accuracy`:

```
| print "Accuracy: ", accuracy.eval\
|       (feed_dict={X: mnist.test.images, \
|                  Y_: mnist.test.labels})
| print "done"
```

After testing the network, we can, while remaining within the session, run the network model on a single image. For example, we can randomly choose via the `randint` function, a picture from the `mnist.test` database:

```
| num = randint(0, mnist.test.images.shape[0])
| img = mnist.test.images[num]
```

So, we can use the implemented classifier on the selected image:

```
| classification = sess.run(tf.argmax(Y, 1), feed_dict={X: [img]})
```

The arguments of the `sess.run` function are respectively the *output* and the *input* of the network. The `tf.argmax(Y, 1)` function returns the maximum index value for the `Y` tensor, which is the image we are looking for, the next argument, `feed_dict={X: [img]}`, allows us to feed the network with the selected image.

Finally, we display the results, that is, the predicted label and the effective label:

```
| print 'Neural Network predicted', classification[0]
| print 'Real label is:', np.argmax(mnist.test.labels[num])
```

The execution is shown in the following snippet. As you can see, after loading the MNIST data, the training epochs are displayed until the ninth:

```
>>>
Loading data/train-images-idx3-ubyte.mnist
Loading data/train-labels-idx1-ubyte.mnist
Loading data/t10k-images-idx3-ubyte.mnist
Loading data/t10k-labels-idx1-ubyte.mnist
Epoch:  0
Epoch:  1
Epoch:  2
Epoch:  3
Epoch:  4
Epoch:  5
Epoch:  6
Epoch:  7
Epoch:  8
Epoch:  9
```

Then we display the model's accuracy:

```
Accuracy:  0.9246
done

The predicted and real label:

Neural Network predicted 6
Real label is: 6
>>>
```

After running the model, we can analyze the execution phases of the executions by using TensorBoard.

Visualization

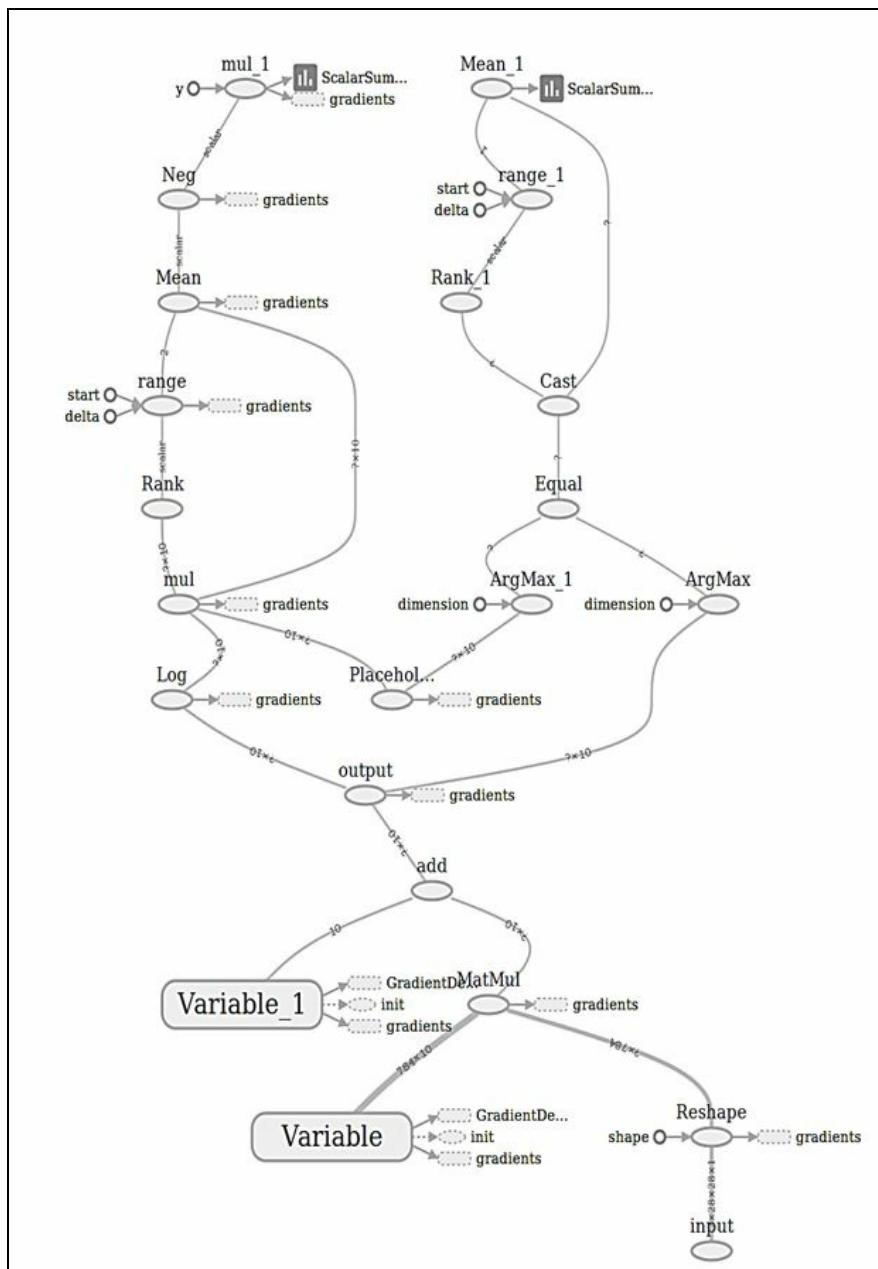
To run TensorBoard, open the terminal in the folder where the code was executed. Then digit the following command:

```
| $> tensorboard --logdir= 'log_simple_stats_softmax'
```

Once TensorBoard is running, navigate your web browser to `localhost:6006` to view the TensorBoard starting page.

When looking at TensorBoard, you will see the *navigation tabs* in the top right-corner. Each tab represents a set of serialized data that can be visualized.

The following figure shows the computational graph for the implemented classifier:



Graph representation for the softmax classifier

How to save and restore a TensorFlow model

Let's suppose we want to use the results of this trained model repeatedly, but without re-training the model each time.

Saving a model

To save a model, we use the `saver()` class. It saves the graph structure using checkpoints: these are binary files in a proprietary format, which map variable names to tensor values. The following code saves the model into our current working directory as two files:

- `softmax_mnist.ckpt`, which contains the weights
- `softmax_mnist.ckpt.meta`, which contains the graph definition

The following code must be inserted at the end of the previous model:

```
| saver = tf.train.Saver()  
| save_path = saver.save(sess, "softmax_mnist")  
| print("Model saved to %s" % save_path)
```


Restoring a model

In a second file, we create the following script to restore the deployed network.

Let's start by loading the required libraries:

```
| import matplotlib.pyplot as plt  
| import tensorflow as tf  
| import input_data  
| import numpy as np  
| import mnist_data
```

And then by adding the MNIST dataset using the following:

```
| mnist = mnist_data.read_data_sets('data', one_hot=True)
```

Implement an *interactive* session:

```
| sess = tf.InteractiveSession()
```

The following line imports the meta graph saved, which contains all the information we need on the topology of the model and its variables:

```
| new_saver = tf.train.import_meta_graph('softmax_mnist.ckpt.meta')
```

Then import the checkpoint file, which contains the weights we developed during training:

```
| new_saver.restore(sess, 'softmax_mnist.ckpt')
```

To run the loaded model, we need the computation graph, which we call through the following function:

```
| tf.get_default_graph().
```

The following function will return the default graph being used in the current thread:

```
| tf.get_default_graph().as_graph_def()
```

We then define the `x` and `y_conv` variables and associate them with the nodes that we need to handle in order to feed input and retrieve output:

```
| x = sess.graph.get_tensor_by_name("input:0")  
| y_conv = sess.graph.get_tensor_by_name("output:0")
```

To test the restored model, we take a single image from the MNIST database:

```
| image_b = mnist.test.images[100]
```

Then we run the restored model, on the selected input:

```
| result = sess.run(y_conv, feed_dict={x:image_b})
```

The `result` variable is the output tensor of 10 items, each of which represents the probability for each digit to be classified. So we print our result and the index with the highest probability, which is just the classified digit:

```
| print(result)
| print(sess.run(tf.argmax(result, 1)))
```

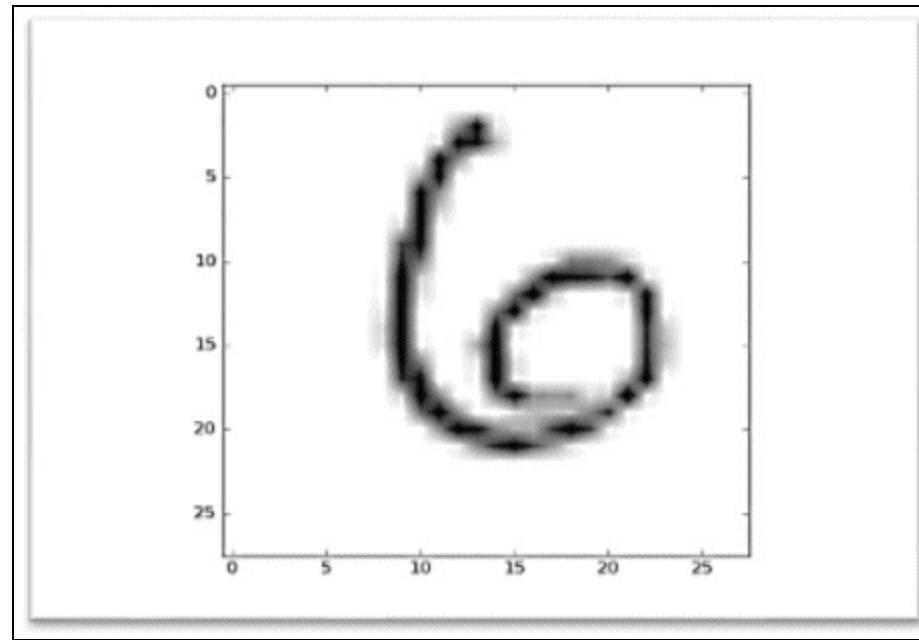
We show the classified image using the `plt` function imported from `matplotlib`:

```
| plt.imshow(image_b.reshape([28, 28]), cmap='Greys')
| plt.show()
```

Running the network, we should have the following output:

```
>>>
Loading data/train-images-idx3-ubyte.mnist
Loading data/train-labels-idx1-ubyte.mnist
Loading data/t10k-images-idx3-ubyte.mnist
Loading data/t10k-labels-idx1-ubyte.mnist
[[ 5.37428750e-05   6.65060536e-04   1.42298099e-02   3.05720314e-04
  2.49665667e-04   6.00658204e-05   9.83844459e-01   4.97680194e-05
  4.59994393e-04   8.17739274e-05]]
[6]
```

The higher array item is $9.83844459e-01$ ($= 90\%$), which corresponds to the digit 6, represented as follows:



Classified image

However, the confirmation of the goodness of the simulation is verified by drawing the classified image.

Softmax source code

Here we report the complete source code for the softmax classifier:

```
import tensorflow as tf
import mnist_data
import matplotlib.pyplot as plt
from random import randint
import numpy as np

logs_path = 'log_mnist_softmax'
batch_size = 100
learning_rate = 0.5
training_epochs = 10

mnist = mnist_data.read_data_sets("data")

X = tf.placeholder(tf.float32, [None, 28, 28, 1], name="input")
Y_ = tf.placeholder(tf.float32, [None, 10])
W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))
XX = tf.reshape(X, [-1, 784])

Y = tf.nn.softmax(tf.matmul(XX, W) + b, name="output")
cross_entropy = -tf.reduce_mean(Y_ * tf.log(Y)) * 1000.0
correct_prediction = tf.equal(tf.argmax(Y, 1), \
                             tf.argmax(Y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, \
                                   tf.float32))
train_step = tf.train.GradientDescentOptimizer\
            (0.005).minimize(cross_entropy)

tf.summary.scalar("cost", cross_entropy)
tf.summary.scalar("accuracy", accuracy)
summary_op = tf.summary.merge_all()
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())

writer = tf.summary.FileWriter(logs_path, graph=tf.get_default_graph())
for epoch in range(training_epochs):
    batch_count = int(mnist.train.num_examples/batch_size)
    for i in range(batch_count):
        batch_x, batch_y = mnist.train.next_batch(batch_size)
        sess.run(train_step, feed_dict={X: batch_x, \
                                       Y_: batch_y})

        print "Epoch: ", epoch

    print "Accuracy: ", accuracy.eval\
        (feed_dict={X: mnist.test.images, \
                   Y_: mnist.test.labels})
    print "done"

    num = randint(0, mnist.test.images.shape[0])
    img = mnist.test.images[num]

    classification = sess.run(tf.argmax(Y, 1), \
                               feed_dict={X: [img]})
    print 'Neural Network predicted', classification[0]
    print 'Real label is:', np.argmax(mnist.test.labels[num])

    saver = tf.train.Saver()
    save_path = saver.save(sess, "saved_mnist_cnn.ckpt")
    print("Model saved to %s" % save_path)
```


Softmax loader source code

To load the network and to test on a single image, you need to execute the following lines of code.
Here is the full source code:

```
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
import mnist_data

mnist = mnist_data.read_data_sets('data', one_hot=True)
sess = tf.InteractiveSession()
new_saver = tf.train.import_meta_graph('saved_mnist_cnn.ckpt.meta')
new_saver.restore(sess, 'saved_mnist_cnn.ckpt')
tf.get_default_graph().as_graph_def()
x = sess.graph.get_tensor_by_name("input:0")
y_conv = sess.graph.get_tensor_by_name("output:0")
image_b = mnist.test.images[100]
result = sess.run(y_conv, feed_dict={x:image_b})
print(result)
print(sess.run(tf.argmax(result, 1)))

plt.imshow(image_b.reshape([28, 28]), cmap='Greys')
plt.show()
```


Implementing a five-layer neural network

The following implementation increases the network complexity by adding four layers before the softmax layer. To determine the appropriate size of the network, that is, the number of hidden layers and the number of neurons per layer, generally we rely on general empirical criteria, the personal experience, or appropriate tests.

The following table summarizes the implemented network architecture, it shows the number of neurons per layer and the respective activation functions:

Layer	Number of neurons	Activation function
First	$L = 200$	sigmoid
Second	$M = 100$	sigmoid
Third	$N = 60$	sigmoid
Fourth	$O = 30$	sigmoid
Fifth	10	softmax

The transfer function for the first four layers is the **sigmoid** function; the last layer of the transfer function is always the **softmax** since the output of the network must express a probability for the input digit. In general, the number and the size of the intermediate layers greatly affect the network performance:

- In a positive way, because on these layers is based the ability of the net to generalize and to detect peculiar characteristics of the input
- In a negative way because if the network is redundant it unnecessarily weighs down the learning phase

We will now start to implement the network, importing the following libraries:

```
import mnist_data
import tensorflow as tf
import math
```

Then we will set the following configuration parameters:

```
logs_path = 'log_simple_stats_5_layers_relu_softmax'
batch_size = 100
learning_rate = 0.5
training_epochs = 10
```

We will then download images and labels and prepare the dataset:

```
| mnist = mnist_data.read_data_sets("data")
```

Starting with the input layer, we'll now see how to build the network's architecture.

The input layer is now a tensor of the shape $[1 \times 784]$, which represents the image to classify:

```
| x = tf.placeholder(tf.float32, [None, 28, 28, 1])
| xx = tf.reshape(x, [-1, 784])
```

The first layer receives the pixels of the input image to be classified combined with the w_1 weight connections and added to the respective values of the b_1 biases tensor:

```
| W1 = tf.Variable(tf.truncated_normal([784, L], stddev=0.1))
| B1 = tf.Variable(tf.zeros([L]))
```

The first layer sends its output to the second layer, through the sigmoid activation function:

```
| y1 = tf.nn.sigmoid(tf.matmul(xx, W1) + B1)
```

The second layer receives the y_1 output from the first layer and combines it with the w_2 weight connections and adds it to the respective values of the b_2 biases tensor:

```
| W2 = tf.Variable(tf.truncated_normal([L, M], stddev=0.1))
| B2 = tf.Variable(tf.zeros([M]))
```

The second layer sends its output to the third layer, through the sigmoid activation function:

```
| y2 = tf.nn.sigmoid(tf.matmul(y1, W2) + B2)
```

The third layer receives the y_2 output from the second layer and combines it with the w_3 weight connections and adds it to the respective values of the b_3 biases tensor:

```
| W3 = tf.Variable(tf.truncated_normal([M, N], stddev=0.1))
| B3 = tf.Variable(tf.zeros([N]))
```

The third layer sends its output to the fourth layer, through the sigmoid activation function:

```
| y3 = tf.nn.sigmoid(tf.matmul(y2, W3) + B3)
```

The fourth layer receives the y_3 output from the third layer and combines it with the w_4 weight connections and adds it to the respective values of the b_4 biases tensor:

```
| W4 = tf.Variable(tf.truncated_normal([N, O], stddev=0.1))
| B4 = tf.Variable(tf.zeros([O]))
```

It sends its output to the fifth layer, through the sigmoid activation function:

```
| y4 = tf.nn.sigmoid(tf.matmul(y3, W4) + B4)
```

The fifth layer will receive in input the $O = 30$ stimuli coming from the fourth layer that will be converted in the respective classes of probability for each number, through the softmax activation function:

```
| W5 = tf.Variable(tf.truncated_normal([O, 10], stddev=0.1))
| B5 = tf.Variable(tf.zeros([10]))
```

```
| Ylogits = tf.matmul(Y4, W5) + B5  
| Y = tf.nn.softmax(Ylogits)
```

Here, our loss function is the `cross-entropy` between the target and the softmax activation function applied to the model's prediction:

```
| cross_entropy = tf.nn.softmax_cross_entropy_with_logits(logits=Ylogits, labels=Y_)  
| cross_entropy = tf.reduce_mean(cross_entropy)
```

The `tf.train.AdamOptimizer` uses the *Kingma and Ba's Adam* algorithm (<https://arxiv.org/pdf/1412.6980v8.pdf>) to control the learning rate. `AdamOptimizer` offers several advantages over the simple `tf.train.GradientDescentOptimizer`; in fact, it uses a larger effective step size, and the algorithm will converge to this step size without fine tuning:

```
| learning_rate = 0.003  
| train_step = tf.train.AdamOptimizer(learning_rate).minimize(cross_entropy)
```

Also, we define the `correct_prediction` and the model's `accuracy`:

```
| correct_prediction = tf.equal(tf.argmax(Y, 1), tf.argmax(Y_, 1))  
| accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

The source code for the definition of the summaries and the running of the session is almost identical to the previous. We can pass directly to evaluate the implemented model. Running the model, we have the following output. The final test set accuracy after running this code should be approximately 97%:

```
>>>  
Loading data/train-images-idx3-ubyte.mnist  
Loading data/train-labels-idx1-ubyte.mnist  
Loading data/t10k-images-idx3-ubyte.mnist  
Loading data/t10k-labels-idx1-ubyte.mnist  
Epoch: 0  
Epoch: 1  
Epoch: 2  
Epoch: 3  
Epoch: 4  
Epoch: 5  
Epoch: 6  
Epoch: 7  
Epoch: 8  
Epoch: 9  
Accuracy: 0.9744  
done  
>>>
```

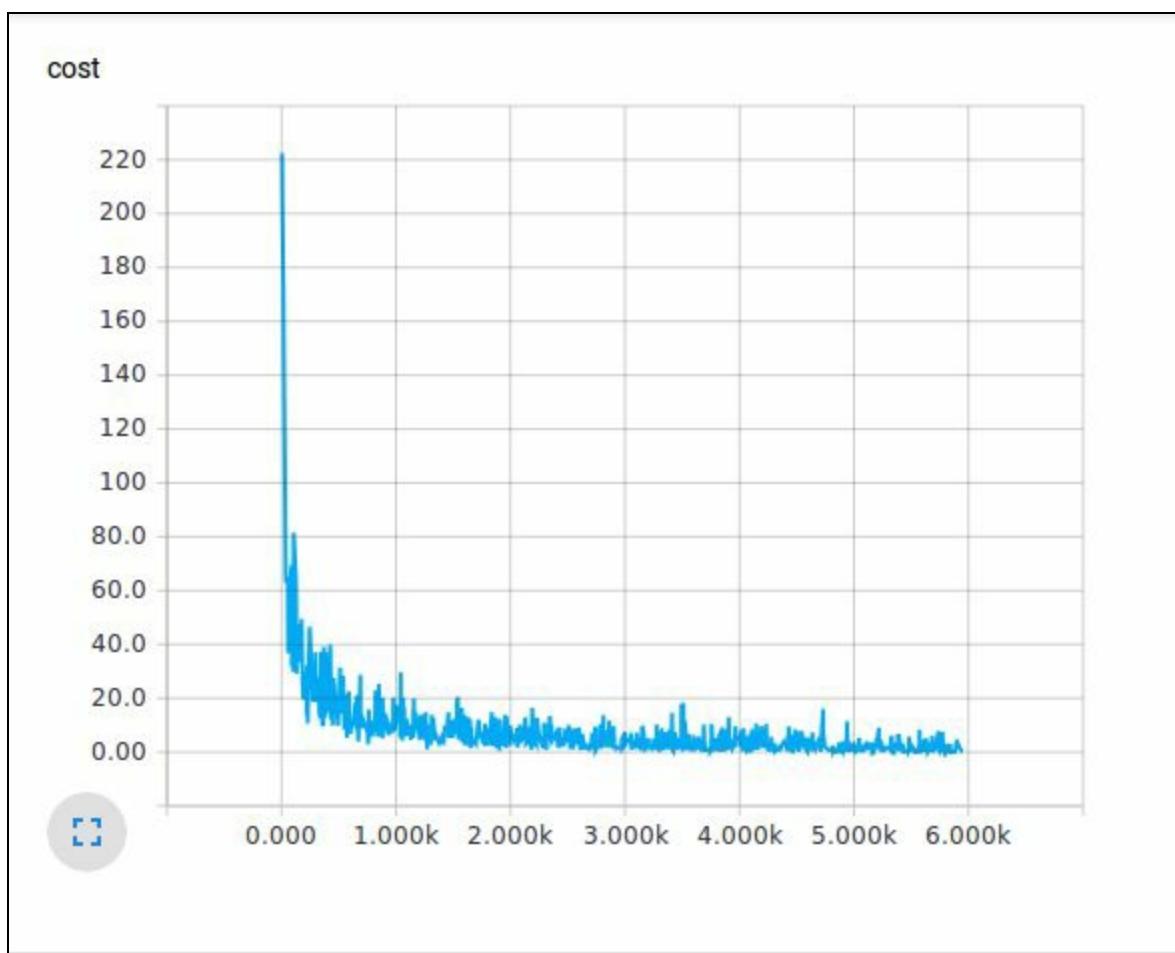

Visualization

We can now move onto TensorBoard, by simply opening the terminal in the running folder, followed by this command:

```
| $> Tensorboard --logdir = 'log_simple_stats_5_layers_relu_softmax'
```

Then open your browser at `localhost`.

In the following figure, we show the trend of the `cost` function as a function of the number of examples over the training set:



Cost function over train set

The `cost` function decreases with increasing iterations. This pattern is correct; it is exactly shown in the preceding figure. If this doesn't happen, it means that something went wrong: in the best case, it could simply be due to the fact that some parameters have not been set properly; at worst, it could be a problem in the constructed dataset, for example, too little information, poor quality images, or more. In this case, we must directly fix the dataset.

Five-layer neural network source code

To complete this section, we will now report the entire source code:

```
import mnist_data
import tensorflow as tf
import math

logs_path = 'log_simple_stats_5_layers_relu_softmax'
batch_size = 100
learning_rate = 0.5
training_epochs = 10

mnist = mnist_data.read_data_sets("data")

the images in the mini-batch
X = tf.placeholder(tf.float32, [None, 28, 28, 1])
Y_ = tf.placeholder(tf.float32, [None, 10])
lr = tf.placeholder(tf.float32)

L = 200
M = 100
N = 60
O = 30

W1 = tf.Variable(tf.truncated_normal([784, L], stddev=0.1))
B1 = tf.Variable(tf.ones([L])/10)
W2 = tf.Variable(tf.truncated_normal([L, M], stddev=0.1))
B2 = tf.Variable(tf.ones([M])/10)
W3 = tf.Variable(tf.truncated_normal([M, N], stddev=0.1))
B3 = tf.Variable(tf.ones([N])/10)
W4 = tf.Variable(tf.truncated_normal([N, O], stddev=0.1))
B4 = tf.Variable(tf.ones([O])/10)
W5 = tf.Variable(tf.truncated_normal([O, 10], stddev=0.1))
B5 = tf.Variable(tf.zeros([10]))

XX = tf.reshape(X, [-1, 784])
Y1 = tf.nn.relu(tf.matmul(XX, W1) + B1)
Y2 = tf.nn.relu(tf.matmul(Y1, W2) + B2)
Y3 = tf.nn.relu(tf.matmul(Y2, W3) + B3)
Y4 = tf.nn.relu(tf.matmul(Y3, W4) + B4)
Ylogits = tf.matmul(Y4, W5) + B5
Y = tf.nn.softmax(Ylogits)

cross_entropy = tf.nn.softmax_cross_entropy_with_logits(logits=Ylogits, labels=Y_)
cross_entropy = tf.reduce_mean(cross_entropy)

correct_prediction = tf.equal(tf.argmax(Y, 1), tf.argmax(Y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
train_step = tf.train.AdamOptimizer(lr).minimize(cross_entropy)

tf.summary.scalar("cost", cross_entropy)
tf.summary.scalar("accuracy", accuracy)
summary_op = tf.summary.merge_all()

init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)

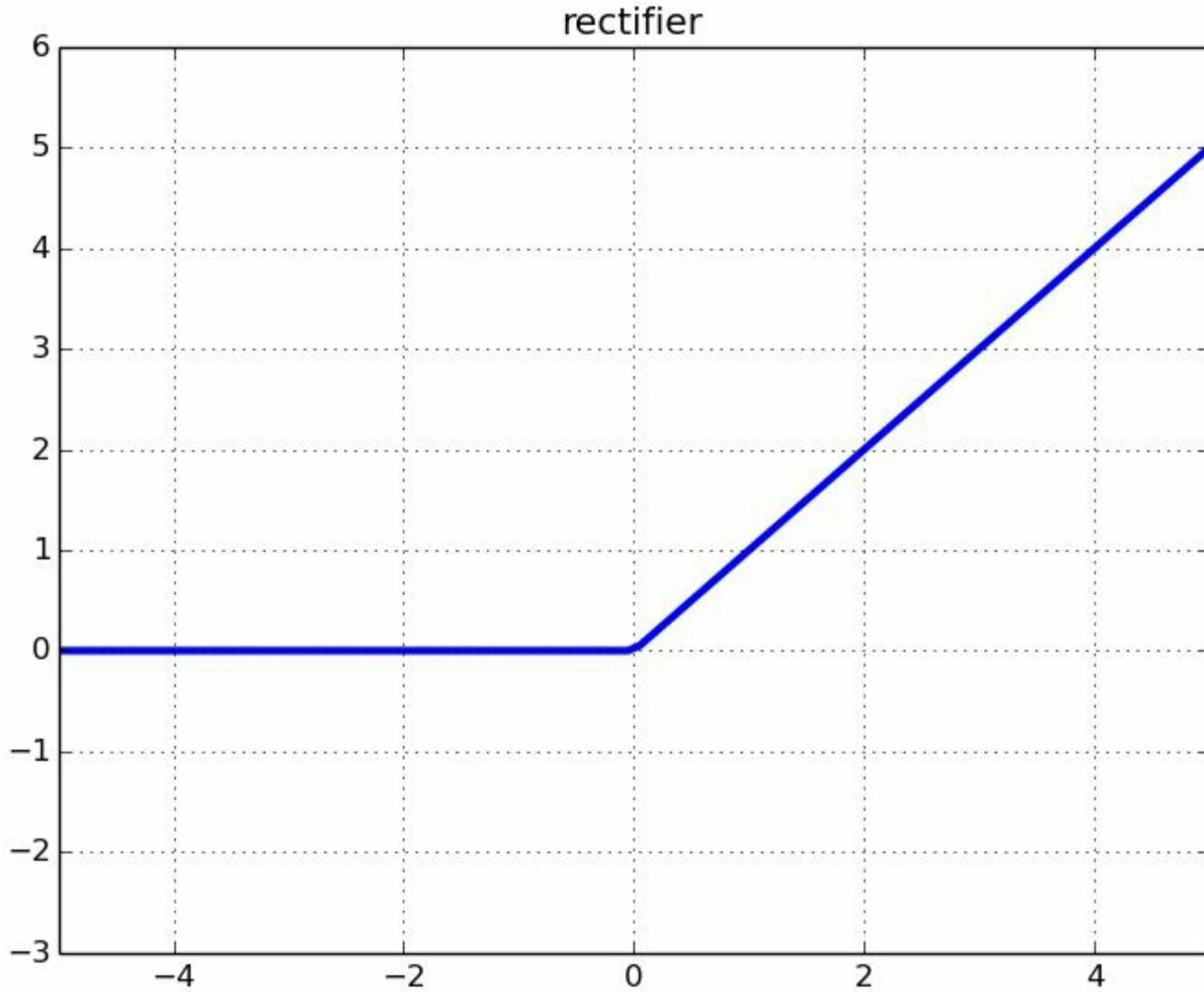
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    writer = tf.summary.FileWriter(logs_path,
                                    graph=tf.get_default_graph())

    for epoch in range(training_epochs):
        batch_count = int(mnist.train.num_examples/batch_size)
        for i in range(batch_count):
            batch_x, batch_y = mnist.train.next_batch(batch_size)
            _, summary = sess.run([train_step, summary_op], \
                feed_dict={X: batch_x, \
                           Y_: batch_y})
            writer.add_summary(summary, \
                epoch * batch_count + i)
```

```
#if epoch % 2 == 0:  
print "Epoch: ", epoch  
  
print "Accuracy: ", accuracy.eval\  
      (feed_dict={X: mnist.test.images,  
                  Y_: mnist.test.labels})  
print "done"
```


ReLU classifier

The last architectural change improved the accuracy of our model, but we can do even better by changing the sigmoid activation function with the Rectified Linear Unit, shown as follows:



ReLU function

A **Rectified Linear Unit (ReLU)** unit computes the function $f(x) = \max(0, x)$, ReLU is *computationally* fast because it does not require any exponential computation, such as those required in sigmoid or tanh activations, furthermore it was found to greatly accelerate the convergence of stochastic gradient descent compared to the sigmoid/tanh functions.

To use the ReLU function, we simply change, in the previously implemented model, the following definitions of the first four layers, in the previously implemented model.

First layer output:

```
| Y1 = tf.nn.relu(tf.matmul(XX, W1) + B1)
```

Second layer output:

```
| Y2 = tf.nn.relu(tf.matmul(Y1, W2) + B2)
```

Third layer output:

```
| Y3 = tf.nn.relu(tf.matmul(Y2, W3) + B3)
```

Fourth layer output:

```
| Y4 = tf.nn.relu(tf.matmul(Y3, W4) + B4)
```

Of course `tf.nn.relu` is TensorFlow's implementation of ReLU.

The accuracy of the model is almost 98%, as you could see running the network:

```
>>>
Loading data/train-images-idx3-ubyte.mnist
Loading data/train-labels-idx1-ubyte.mnist
Loading data/t10k-images-idx3-ubyte.mnist
Loading data/t10k-labels-idx1-ubyte.mnist
Epoch:  0
Epoch:  1
Epoch:  2
Epoch:  3
Epoch:  4
Epoch:  5
Epoch:  6
Epoch:  7
Epoch:  8
Epoch:  9
Accuracy:  0.9789
done
>>>
```

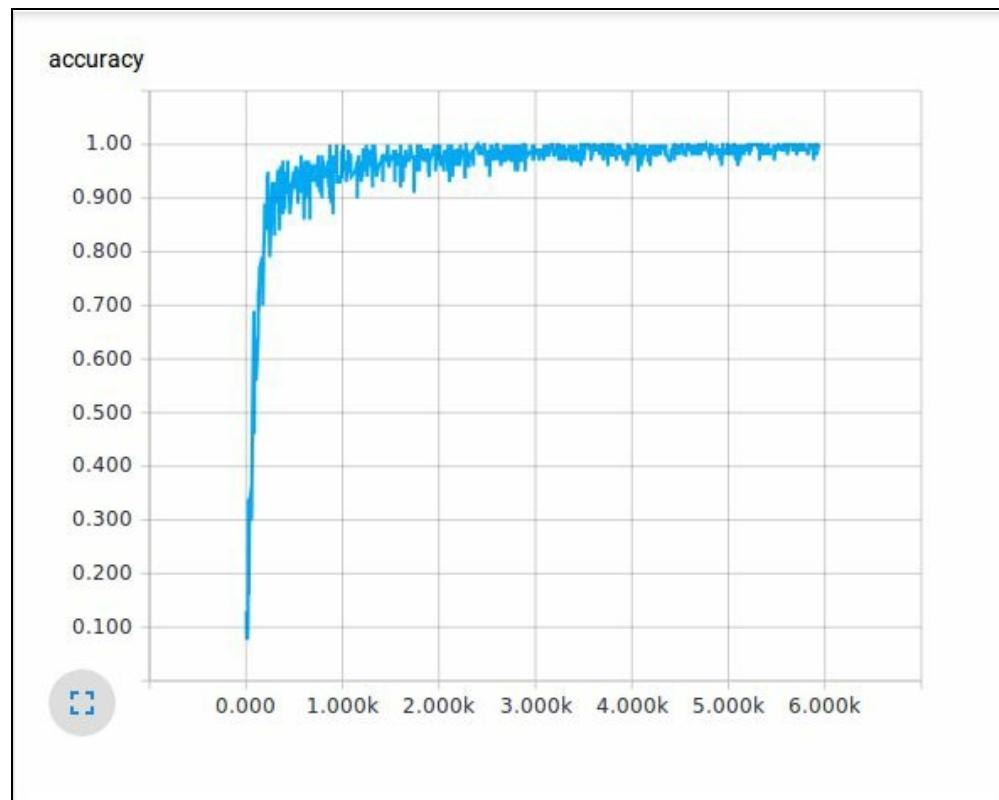

Visualization

As concerns, the TensorBoard analysis, from the folder where the source has been executed, you should digit:

```
| $> Tensorboard --logdir = 'log_simple_stats_5_layers_relu_softmax'
```

Then open the browser at localhost to visualize TensorBoard's starting page.

In the following figure, we show the trend's accuracy over the number of examples of the training set:



Accuracy function over train set

You can easily see how the accuracy, after a bad initial trend, begins a rapid progressive improvement after about 1000 examples.

Source code for the ReLU classifier

Here we have the entire code for the implemented ReLU classifier:

```
import mnist_data
import tensorflow as tf
import math

logs_path = 'log_simple_stats_5_layers_relu_softmax'
batch_size = 100
learning_rate = 0.5
training_epochs = 10

mnist = mnist_data.read_data_sets("data")

X = tf.placeholder(tf.float32, [None, 28, 28, 1])
Y_ = tf.placeholder(tf.float32, [None, 10])
lr = tf.placeholder(tf.float32)

# five layers and their number of neurons (the last layer has 10 softmax neurons)
L = 200
M = 100
N = 60
O = 30

W1 = tf.Variable(tf.truncated_normal([784, L], stddev=0.1))
B1 = tf.Variable(tf.ones([L])/10)
W2 = tf.Variable(tf.truncated_normal([L, M], stddev=0.1))
B2 = tf.Variable(tf.ones([M])/10)
W3 = tf.Variable(tf.truncated_normal([M, N], stddev=0.1))
B3 = tf.Variable(tf.ones([N])/10)
W4 = tf.Variable(tf.truncated_normal([N, O], stddev=0.1))
B4 = tf.Variable(tf.ones([O])/10)
W5 = tf.Variable(tf.truncated_normal([O, 10], stddev=0.1))
B5 = tf.Variable(tf.zeros([10]))

XX = tf.reshape(X, [-1, 784])
Y1 = tf.nn.relu(tf.matmul(XX, W1) + B1)
Y2 = tf.nn.relu(tf.matmul(Y1, W2) + B2)
Y3 = tf.nn.relu(tf.matmul(Y2, W3) + B3)
Y4 = tf.nn.relu(tf.matmul(Y3, W4) + B4)
Ylogits = tf.matmul(Y4, W5) + B5
Y = tf.nn.softmax(Ylogits)

cross_entropy = tf.nn.softmax_cross_entropy_with_logits(logits=Ylogits, labels=Y_)
cross_entropy = tf.reduce_mean(cross_entropy)

correct_prediction = tf.equal(tf.argmax(Y, 1), tf.argmax(Y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

train_step = tf.train.AdamOptimizer(lr).minimize(cross_entropy)

#tensorboard parameters
tf.summary.scalar("cost", cross_entropy)
tf.summary.scalar("accuracy", accuracy)
summary_op = tf.summary.merge_all()
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    writer = tf.summary.FileWriter(logs_path, graph=tf.get_default_graph())

    for epoch in range(training_epochs):
        batch_count = int(mnist.train.num_examples/batch_size)
        for i in range(batch_count):
            batch_x, batch_y = mnist.train.next_batch(batch_size)
            max_learning_rate = 0.003
            min_learning_rate = 0.0001
            decay_speed = 2000
            learning_rate = min_learning_rate+\n                (max_learning_rate - min_learning_rate)\n\n            sess.run(train_step, feed_dict={X: batch_x, Y_: batch_y, lr: learning_rate})\n\n            if i % 50 == 0:\n                summary = sess.run(summary_op, feed_dict={X: batch_x, Y_: batch_y})\n                writer.add_summary(summary, epoch * batch_count + i)\n\n            if i % 100 == 0:\n                print('Epoch', epoch, 'Batch', i, 'Cost', cross_entropy.eval(session=sess, feed_dict={X: batch_x, Y_: batch_y}))\n\n        print('Epoch', epoch, 'Accuracy', accuracy.eval(session=sess, feed_dict={X: mnist.validation.images, Y_: mnist.validation.labels}))\n\n    print('Final Accuracy', accuracy.eval(session=sess, feed_dict={X: mnist.validation.images, Y_: mnist.validation.labels}))
```

```
* math.exp(-i/decay_speed)
_, summary = sess.run([train_step, summary_op], \
                     {X: batch_x, Y_: batch_y, \
                      lr: learning_rate})
writer.add_summary(summary, \
                   epoch * batch_count + i)
print "Epoch: ", epoch

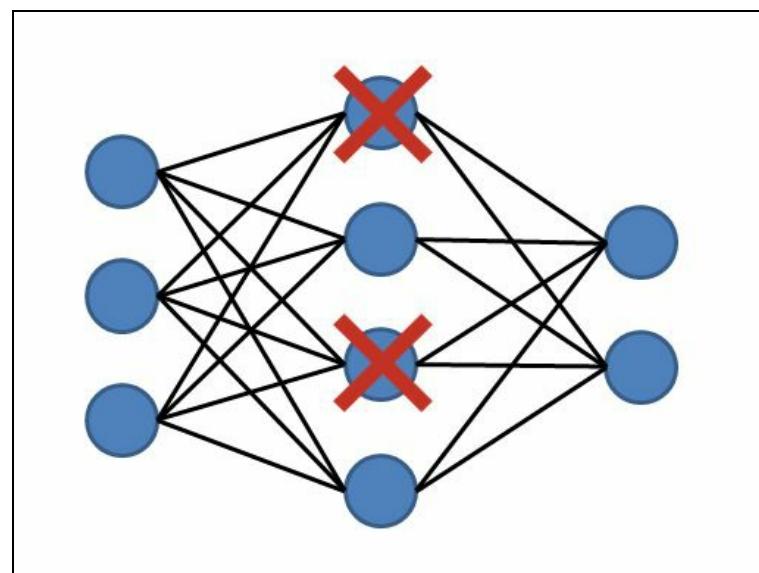
print "Accuracy: ", accuracy.eval\
      (feed_dict={X: mnist.test.images, Y_: mnist.test.labels})
print "done"
```


Dropout optimization

During the learning phase, the connections with the next layer can be limited to a subset of neurons to reduce the weights to be updated, this learning optimization technique is called **dropout**. The dropout is therefore a technique used to decrease the overfitting within a network with many layers and/or neurons. In general, the dropout layers are positioned after the layers that possess a large amount of trainable neurons.

This technique allows setting to 0, and then excluding the activation of a certain percentage of the neurons of the preceding layer. The probability that the neuron's activation is set to 0 is indicated by the dropout ratio parameter within the layer, via a number between 0 and 1: in practice the activation of a neuron is held with probability equal to the dropout ratio, otherwise it is discarded, that is, set to 0.

The neurons by this transaction do not affect, therefore, during the forward propagation and even during the next backward propagation of a certain input. In this way, for each input, the network owns an architecture slightly different from the previous, some connections are active and some are not, in a different way, every time, even if these architectures possess the same weights. The following figure shows how the dropout works: each hidden unit is randomly omitted from the network with a probability of p . One thing to notice though, is that the selected dropout units are different for each training instance, that's why this is more of a training problem, rather dropout can be seen as an efficient way to perform model averaging across a large number of different neural networks, where overfitting can be avoided with much less cost of computation than an architecture problem:



Dropout representation

The dropout reduces the possibility that a neuron relies on the presence of other neurons, in this way it is forced to learn more about robust features and that they are also useful with linkages to other different neurons.

The TensorFlow function that allows building a dropout layer is `tf.nn.dropout`.

The input of this function is the output of the previous layer and a dropout parameter; `tf.nn.dropout`, returns an output tensor of the same size of the input tensor.

The implementation of this model follows the same rules used for the 5-layer network; in this case, between one layer and another layer we must insert the dropout function:

```
dropout_ratio = tf.placeholder(tf.float32)

Y1 = tf.nn.relu(tf.matmul(XX, W1) + B1)
Y1d = tf.nn.dropout(Y1, dropout_ratio)

Y2 = tf.nn.relu(tf.matmul(Y1d, W2) + B2)
Y2d = tf.nn.dropout(Y2, dropout_ratio)

Y3 = tf.nn.relu(tf.matmul(Y2d, W3) + B3)
Y3d = tf.nn.dropout(Y3, dropout_ratio)

Y4 = tf.nn.relu(tf.matmul(Y3d, W4) + B4)
Y4d = tf.nn.dropout(Y4, dropout_ratio)

Ylogits = tf.matmul(Y4d, W5) + B5
Y = tf.nn.softmax(Ylogits)
```

The dropout optimization produces the following results:

```
>>>
Loading data/train-images-idx3-ubyte.mnist
Loading data/train-labels-idx1-ubyte.mnist
Loading data/t10k-images-idx3-ubyte.mnist
Loading data/t10k-labels-idx1-ubyte.mnist
Epoch:  0
Epoch:  1
Epoch:  2
Epoch:  3
Epoch:  4
Epoch:  5
Epoch:  6
Epoch:  7
Epoch:  8
Epoch:  9
Accuracy:  0.9666
done
>>>
```

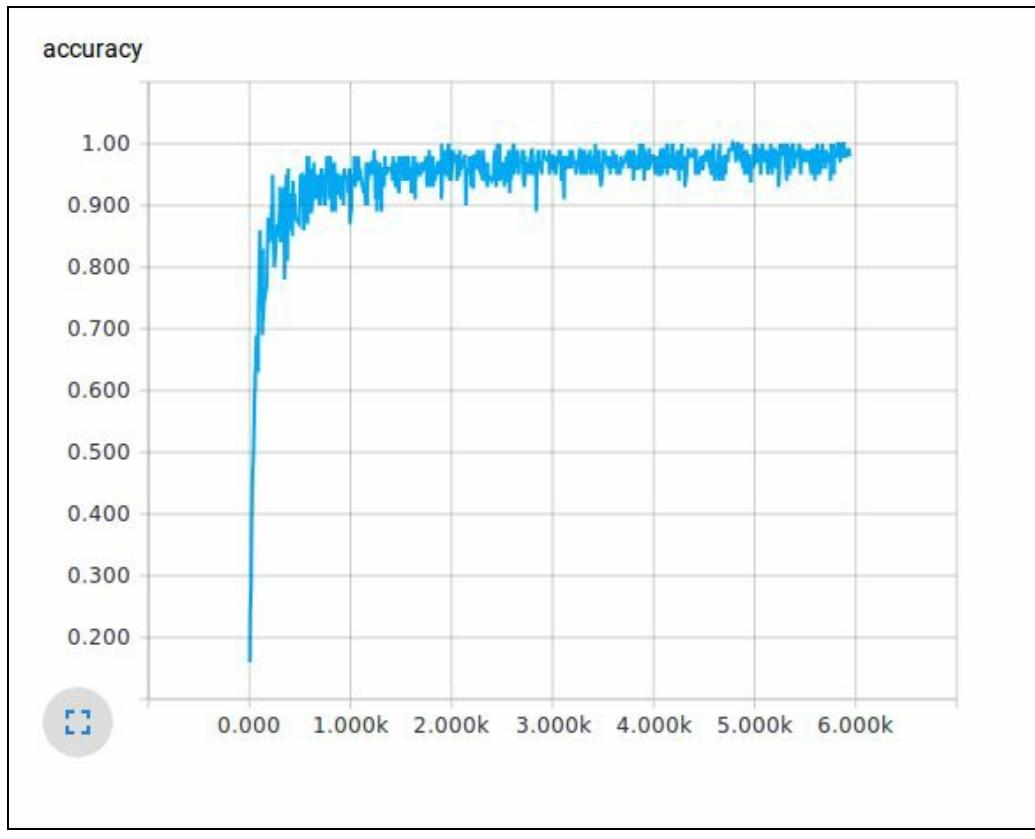
Despite this implementation, the previous ReLU network is still better, but you can try to change the network parameters to improve the model's accuracy.

Visualization

To start the TensorBoard analysis, just type the following:

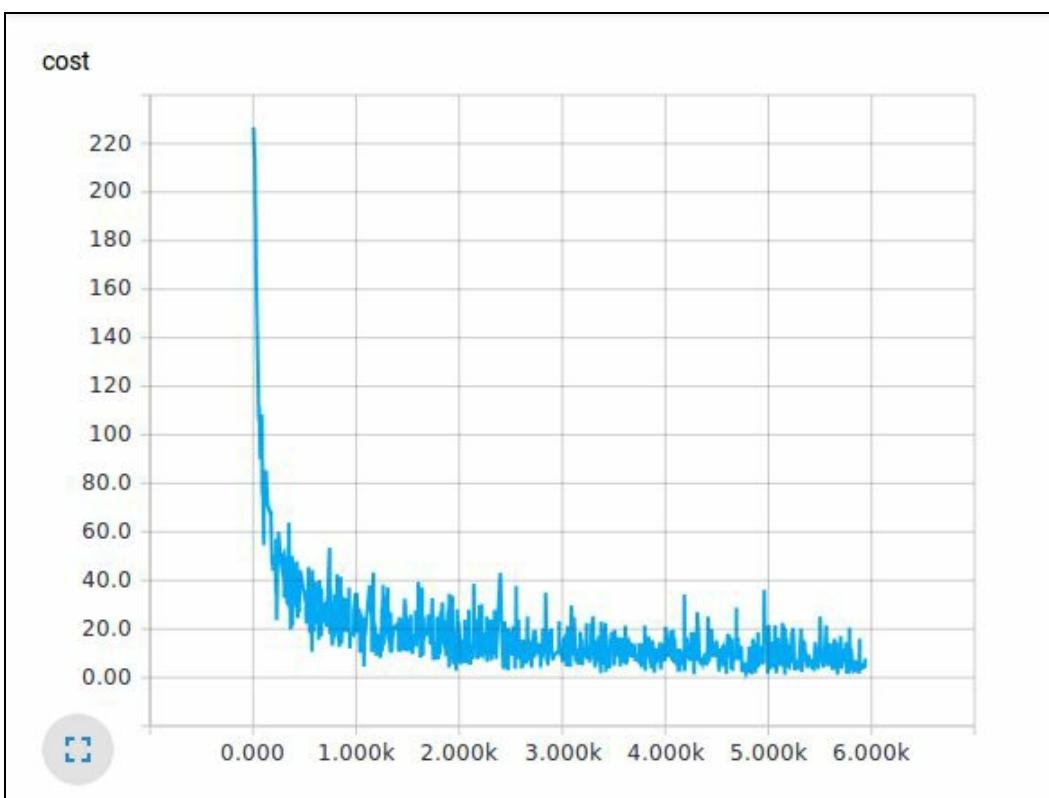
```
$> Tensorboard --logdir = 'log_simple_stats_5_lyers_dropout'
```

The following graph shows the accuracy cost function as a function of the training examples:



Accuracy in dropout optimization

In the following chart, we display the `cost` function as a function of the *training examples*:



The cost function over the training set

Both trends are what we expected: the accuracy increases with training examples, while the `cost` function decreases with increasing iterations.

Source code for dropout optimization

Dropout optimization completes our journey in feed-forward networks. Here is the source code for further analysis and implementations:

```
import mnist_data
import tensorflow as tf
import math

logs_path = 'log_simple_stats_5_layers_dropout'
batch_size = 100
learning_rate = 0.5
training_epochs = 10

mnist = mnist_data.read_data_sets("data")

X = tf.placeholder(tf.float32, [None, 28, 28, 1])
Y_ = tf.placeholder(tf.float32, [None, 10])
lr = tf.placeholder(tf.float32)
pkeep = tf.placeholder(tf.float32)

L = 200
M = 100
N = 60
O = 30

W1 = tf.Variable(tf.truncated_normal([784, L], stddev=0.1))
B1 = tf.Variable(tf.ones([L])/10)
W2 = tf.Variable(tf.truncated_normal([L, M], stddev=0.1))
B2 = tf.Variable(tf.ones([M])/10)
W3 = tf.Variable(tf.truncated_normal([M, N], stddev=0.1))
B3 = tf.Variable(tf.ones([N])/10)
W4 = tf.Variable(tf.truncated_normal([N, O], stddev=0.1))
B4 = tf.Variable(tf.ones([O])/10)
W5 = tf.Variable(tf.truncated_normal([O, 10], stddev=0.1))
B5 = tf.Variable(tf.zeros([10]))

# The model, with dropout at each layer
XX = tf.reshape(X, [-1, 28*28])

Y1 = tf.nn.relu(tf.matmul(XX, W1) + B1)
Y1d = tf.nn.dropout(Y1, pkeep)

Y2 = tf.nn.relu(tf.matmul(Y1d, W2) + B2)
Y2d = tf.nn.dropout(Y2, pkeep)

Y3 = tf.nn.relu(tf.matmul(Y2d, W3) + B3)
Y3d = tf.nn.dropout(Y3, pkeep)

Y4 = tf.nn.relu(tf.matmul(Y3d, W4) + B4)
Y4d = tf.nn.dropout(Y4, pkeep)

Ylogits = tf.matmul(Y4d, W5) + B5
Y = tf.nn.softmax(Ylogits)

cross_entropy = tf.nn.softmax_cross_entropy_with_logits(logits=Ylogits, labels=Y_)
cross_entropy = tf.reduce_mean(cross_entropy)

correct_prediction = tf.equal(tf.argmax(Y, 1), tf.argmax(Y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
train_step = tf.train.AdamOptimizer(lr).minimize(cross_entropy)

tf.summary.scalar("cost", cross_entropy)
tf.summary.scalar("accuracy", accuracy)
summary_op = tf.summary.merge_all()

init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    writer = tf.summary.FileWriter(logs_path,\
```

```
graph=tf.get_default_graph()

for epoch in range(training_epochs):
    batch_count = int(mnist.train.num_examples/batch_size)
    for i in range(batch_count):
        batch_x, batch_y = mnist.train.next_batch(batch_size)
        max_learning_rate = 0.003
        min_learning_rate = 0.0001
        decay_speed = 2000
        learning_rate = min_learning_rate+\n            (max_learning_rate - min_learning_rate)\\
            * math.exp(-i/decay_speed)
        _, summary = sess.run([train_step, summary_op],\
            {X: batch_x, Y_: batch_y,\n             pkeep: 0.75, lr: learning_rate})
        writer.add_summary(summary,\n            epoch * batch_count + i)
    print "Epoch: ", epoch

print "Accuracy: ", accuracy.eval\
    (feed_dict={X: mnist.test.images,\n               Y_: mnist.test.labels, pkeep: 0.75})
print "done"
```


Summary

We have seen how to implement a **feed-forward neural network (ffnn)** architecture for an image classification problem.

An ffnn is characterized by a set of input units, a set of output units, and one or more hidden units that connect the input level from that output. The connections between the levels are total and in a single direction: each unit receives a signal from all the units of the previous layer and transmits its output value, suitably weighed to all units of the next layer. For each layer a transfer function (sigmoid, softmax, ReLU) must be defined: the choice of the transfer function depends on the architecture and then the addressed problem.

Then we implemented four different ffnn models, the first model with a single hidden layer with softmax activation function, and then three other more complex models, with five hidden layers in total, but with different activation functions:

- Four sigmoid layers and one softmax layer
- Four ReLu layers and one softmax layer
- Four ReLu layers with dropout optimization and one softmax layer

In the next chapter, we will go even further in the complexity of the neural network models introducing the **Convolutional Neural Networks (CNNs)**, which may have a big impact in the deep learning techniques: we will study the main features and we will see some implementation examples.

TensorFlow on a Convolutional Neural Network

Convolutional Neural Networks (CNNs) are deep learning networks, which have achieved excellent results in many practical applications, and primarily in object recognition of images. CNN architecture is organized into a series of blocks. The first blocks are composed of two types of layers, convolutional layers and pooling layers; while the last blocks are fully-connected layers with softmax layers.

We'll develop two examples of CNN networks, for image classification problems. The first problem is the classic MNIST digit classification system. We'll see how to build a CNN that reaches 99 percent accuracy. The training set for the second example is taken from the Kaggle platform. The purpose here is to train a network on a series of facial images to classify their emotional stretch.

We'll evaluate the accuracy of the model and then we'll test it on a single image that does not belong to the original dataset.

The following topics are covered in the chapter:

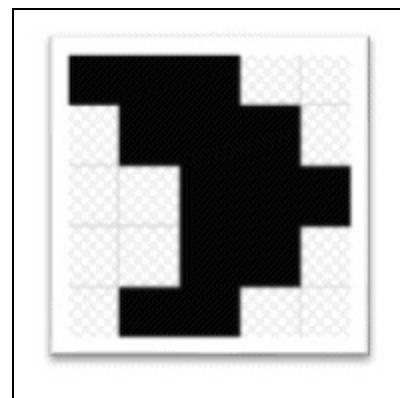
- Introducing CNN networks
- CNN architecture
- A model for CNNs - LeNet
- Building your first CNN
- Emotion recognition with CNNs

Introducing CNNs

In recent years, **Deep Neural Networks (DNNs)** have contributed a new impetus to research as well as industry and are therefore been used increasingly. A special type of a DNN is a **Convolutional Neural Network (CNN)**, which has been used with great success in image classification problems.

Before diving into the implementation of an image classifier based on CNN, we'll introduce some basic concepts in image recognition, such as feature detection and convolution.

It's well known that a real image is associated with a grid composed of a high number of small squares, called **pixels**. The following figure represents a black and white image related to a 5x5 grid of pixels:

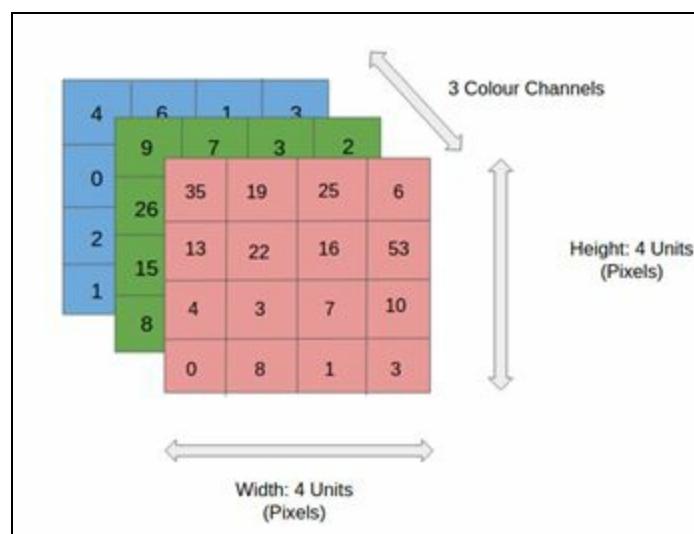


Black and white image

Each element of the grid corresponds to a pixel and, in the case of a black and white image, it assumes either a value of 1, which is associated with black color or the value 0, which is associated with white image. In a grayscale image, the values for each grid element are in the range [0-255], where 0 is associated with black and 255 with white.

Finally, a color image is represented by a group of three matrices, each corresponding to one color channel (red, green, blue); and each element of each matrix can vary over an entire interval [0-255] that specifies the brightness of the fundamental color (or base color).

This representation is shown in the following figure where each represented matrix is a 4x4 size and the number of color channels is three:



Color image representation

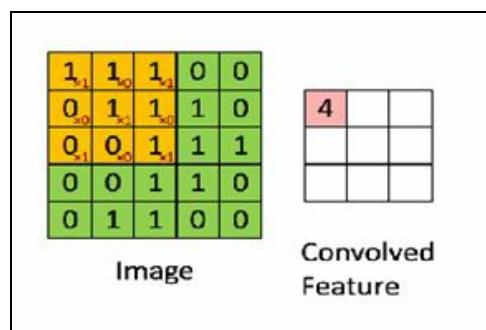
Let's focus now on the black and white image (5x5 matrix), and suppose to flow on this matrix, top to bottom and from left to right, a second matrix of lower dimensions, for example a 3x3 matrix, as shown in the following figure:

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

Kernel filter

This flowing matrix is called a **kernel filter** or **feature detector**. While the kernel filter moves along the input matrix (or input image), it performs a scalar product, between the kernel values and those of the matrix portion to which it is applied. The result is a new matrix, called a **convolution matrix**.

The following figure points out the convolution procedure, the convolved feature (the resulting 3x3 matrix) is generated by the convolution operation, flowing the kernel filter (the 3x3 matrix) on the input image (the 5x5 matrix):



Input image, kernel filter and convolved feature

CNN architecture

Taking as an example the input matrix 5x5 as shown earlier, a CNN consists of an input layer consisting of 25 neurons ($5 \times 5 = 25$) whose task is to acquire the input value corresponding to each pixel and transfer it to the next hidden layer.

In a multilayer network, the outputs of all neurons of the input layer would be connected to each neuron of the hidden layer (fully-connected layer).

In CNN networks, the connection scheme that defines the convolutional layer that we are going to describe is significantly different.

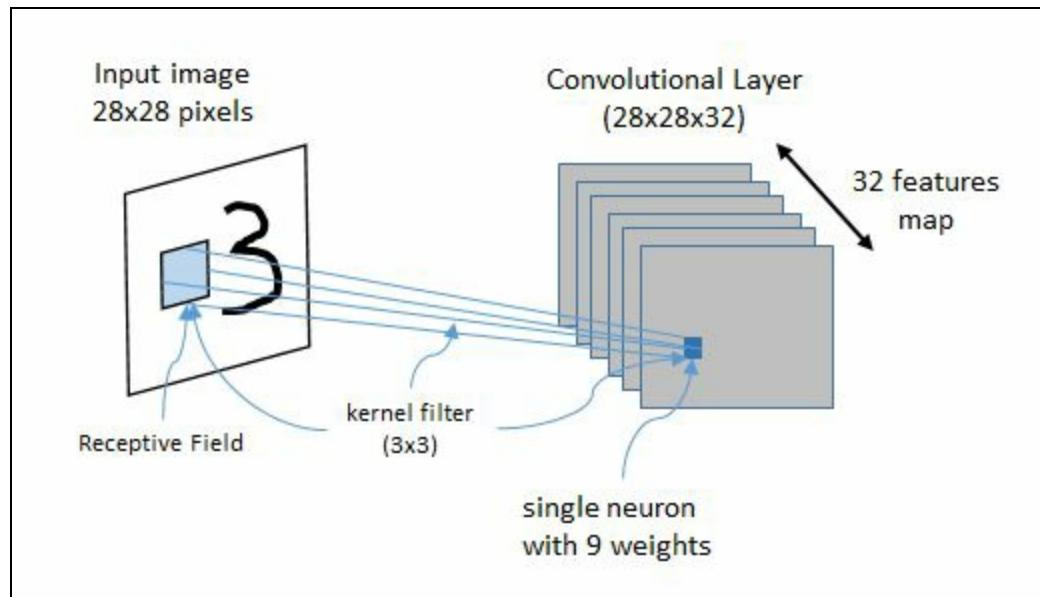
As you can probably guess, this is the main type of layer; the use of one or more of these layers in a CNN is indispensable.

In a convolutional layer, each neuron is connected to a certain region of the input area called the **receptive field**.

For example, using a 3x3 kernel filter, each neuron will have a bias and 9=3x3 weights connected to a single receptive field. Of course, to effectively recognize an image, we need different kernel filters applied to the same receptive field, because each filter should recognize a different feature's image.

The set of neurons that identify the same feature define a single feature map.

The following figure shows a CNN architecture in action; the input image of a size of 28x28 will be analyzed by a convolutional layer composed of a 32 features map with a size of 28x28. The figure also shows a receptive field and the kernel filter of a 3x3 size:



CNN in action

A CNN may consist of several convolution layers connected in a cascade. The output of each convolution layer is a set of feature maps (each generated by a single kernel filter), and all these matrices define a new input that will be used by the next layer.

Usually in a CNN, each neuron produces an output, after an activation threshold, proportional to the input and not bounded; generally, the activation function used is the ReLU function that we introduced in [Chapter 3, Using TensorFlow on a Feed-Forward Neural Network](#).

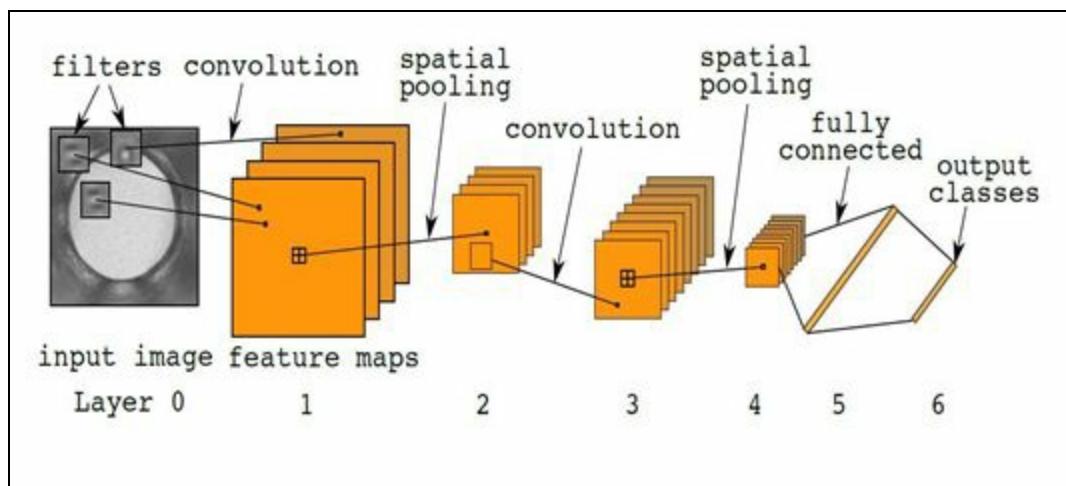
CNNs also use pooling layers positioned immediately after the convolutional layers. A pooling layer divides a convolutional region into sub regions and selects a single representative value (max-pooling or average pooling) to reduce the computational time of subsequent layers and increase the robustness of the feature with respect to its spatial position.

The last hidden layer of a convolutional network is generally a fully-connected network with a softmax activation function for the output layer.

A model for CNNs - LeNet

Convolutional and max-pooling layers are at the heart of the **LeNet** family models. It is a family of multilayered feed-forward networks specialized on visual pattern recognition.

While the exact details of the model will vary greatly, the following figure points out the graphical schema of a LeNet network:



LeNet network

In a LeNet model, the lower layers are composed of an alternating convolution and max-pooling, while the last layers are fully-connected and correspond to a traditional feed-forward network (fully-connected + softmax layer).

The input to the first fully-connected layer is the set of all feature maps at the layer below.

From a TensorFlow implementation point of view, this means lower layers operate on 4D tensors. These are then flattened to a 2D matrix to be compatible with a feed forward implementation.



See <http://yann.lecun.com/exdb/lenet/index.html> for basic references to LeNet family models.

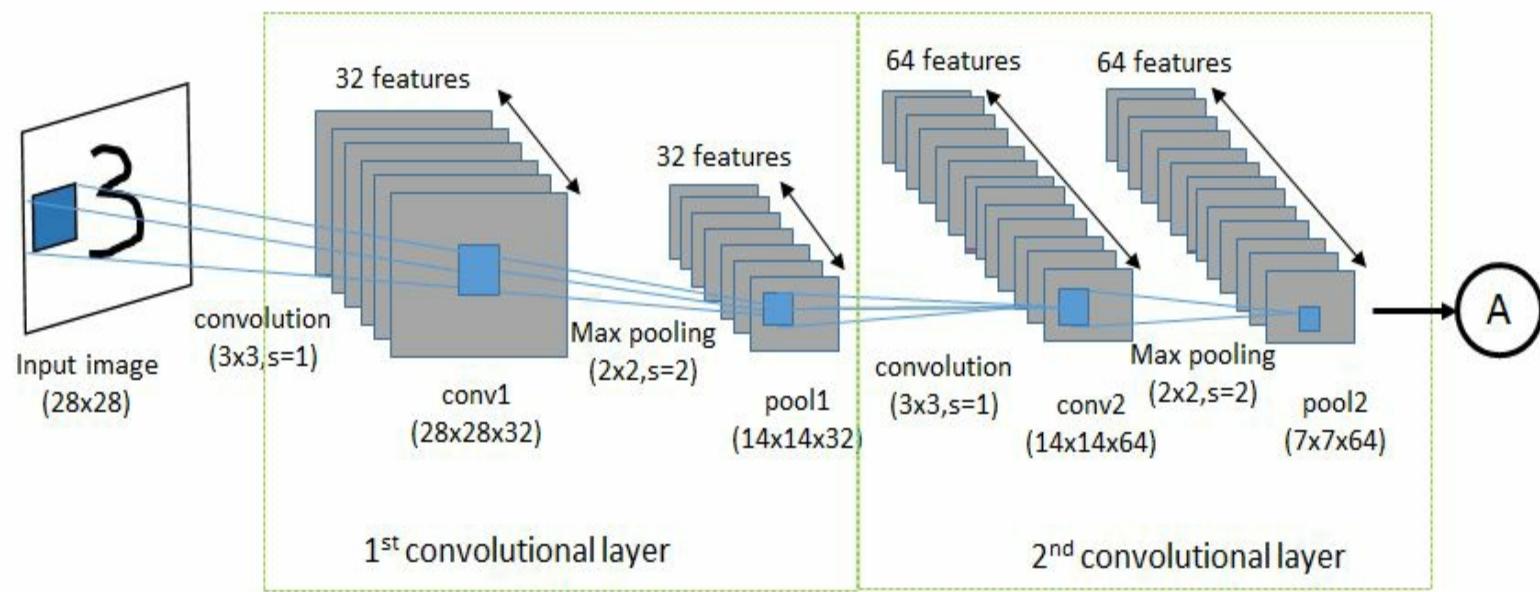
Building your first CNN

In this section, we will learn how to build a CNN to classify images of the MNIST dataset. In the previous chapter, we saw that a simple softmax model provides about 92% classification accuracy for recognizing hand written digits in the MNIST.

Here we'll implement a CNN which has a classification accuracy of about 99%.

The following figure shows how the data flows in the first two convolutional layer. The input image is processed in the first convolutional layer using the filter-weights. This results in 32 new images, one for each filter in the convolutional layer. The images are also downsampled with the pooling operation so the image resolution is decreased from 28x28 to 14x14.

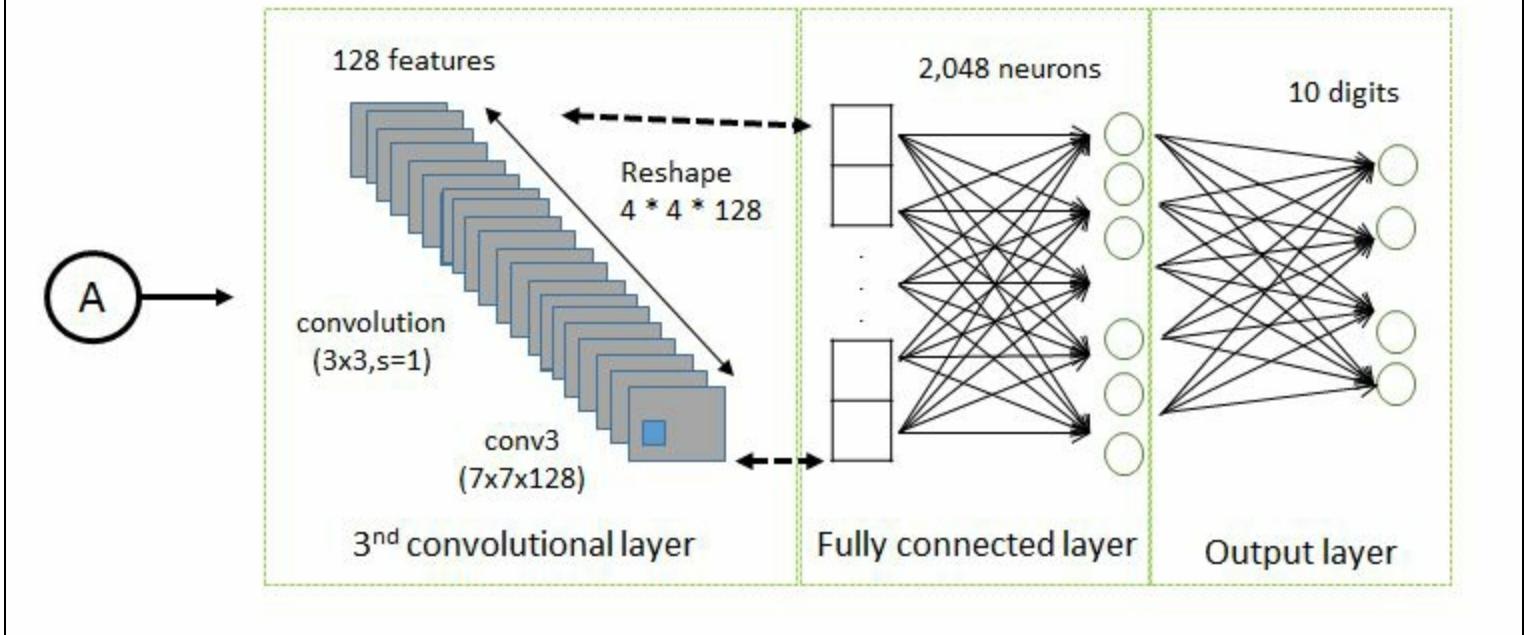
These 32 smaller images are then processed in the second convolutional layer. We need filter weights again for each of these 32 features, and we need filter-weights for each output channel of this layer. The images are again downsampled with a pooling operation so that the image resolution is decreased from 14x14 to 7x7. The total number of features for this convolutional layer is 64:



Data flow of the first two convolutional layers

The 64 resulting images are filtered again by a (3×3) third convolutional layer. We don't apply a pooling operation for this layer. The output of the second convolutional layer is 128 images of 7×7 pixels each. These are then flattened to a single vector of length 4×4×128, which is used as the input to a fully-connected layer with 128 neurons (or elements).

This feeds into another fully-connected layer with 10 neurons, one for each of the classes, which is used to determine the class of the image, that is, which number is depicted in the following image:



Data flow of the last three convolutional layers

The convolutional filters are initially chosen at random. The error between the predicted and actual class of the input image is measured as the so-called **cost function** which generalizes our network beyond the training data. The optimizer then automatically propagates this error back through the convolutional network and updates the filter-weights to improve the classification error.

This is done iteratively thousands of times until the classification error is sufficiently low.

Now let's see in detail how to code our first CNN.

Let's start by importing the TensorFlow libraries for our implementation:

```
import tensorflow as tf
import numpy as np
from tensorflow.examples.tutorials.mnist import input_data
```

Set the following parameters, that indicate the number of samples to consider respectively for the training phase (₁₂₈) and then the test phase (₂₅₆):

```
batch_size = 128
test_size = 256
```

We define the following parameter, the value is ₂₈ because a MNIST image is ₂₈ pixels in height and width:

```
| img_size = 28
```

Regarding the number of classes, the value ₁₀ means that we'll have one class for each ₁₀ digits:

```
| num_classes = 10
```

A placeholder variable, x , is defined for the input images. The data type for this tensor is set to `float32` and the shape is set to `[None, img_size, img_size, 1]`, where `None` means that the tensor may hold an

arbitrary number of images:

```
| x = tf.placeholder("float", [None, img_size, img_size, 1])
```

Then we set another placeholder variable, y , for the true labels associated with the images that were input data in the placeholder variable x .

The shape of this placeholder variable is `[None, num_classes]` which means it may hold an arbitrary number of labels and each label is a vector of the length `num_classes` which is `10` in this case:

```
| y = tf.placeholder("float", [None, num_classes])
```

We collect the `mnist` data which will be copied into the `data` folder:

```
| mnist = mnist_data.read_data_sets("data/")
```

We build the datasets for training (`trX, trY`) and testing the network (`teX, teY`):

```
| trX, trY, teX, teY = mnist.train.images,\n|     mnist.train.labels,\n|     mnist.test.images,\n|     mnist.test.labels
```

The `trX` and `teX` image sets must be reshaped according the input shape:

```
| trX = trX.reshape(-1, img_size, img_size, 1)\n| teX = teX.reshape(-1, img_size, img_size, 1)
```

We shall now proceed to define the network's weights.

The `init_weights` function builds new variables in the given shape and initializes the network's weights with random values:

```
| def init_weights(shape):\n|     return tf.Variable(tf.random_normal(shape, stddev=0.01))
```

Each neuron of the first convolutional layer is convoluted to a small subset of the input tensor, with a dimension of $3 \times 3 \times 1$, while the value `32` is just the number of feature maps we are considering for this first layer. The weight w is then defined:

```
| w = init_weights([3, 3, 1, 32])
```

The number of inputs is then increased of `32`, which means that each neuron of the second convolutional layer is convoluted to $3 \times 3 \times 32$ neurons of the first convolution layer. The w_2 weight is:

```
| w2 = init_weights([3, 3, 32, 64])
```

The value `64` represents the number of obtained output features.

The third convolutional layer is convoluted to $3 \times 3 \times 64$ neurons of the previous layer, while `128` are the resulting features:

```
| w3 = init_weights([3, 3, 64, 128])
```

The fourth layer is fully-connected. It receives 128x4x4 inputs, while the output is equal to 625 :

```
| w4 = init_weights([128 * 4 * 4, 625])
```

The output layer receives 625 inputs, while the output is the number of classes:

```
| w_o = init_weights([625, num_classes])
```

Note that these initializations are not actually done at this point; they are merely being defined in the TensorFlow graph:

```
| p_keep_conv = tf.placeholder("float")
| p_keep_hidden = tf.placeholder("float")
```

It's time to define the network model. As we did for the network's weight definition, it will be a function.

It receives as input, the `x` tensor, the weights tensors, and the dropout parameters for convolution and fully-connected layers:

```
| def model(x, w, w2, w3, w4, w_o, p_keep_conv, p_keep_hidden):
```

The `tf.nn.conv2d()` function executes the TensorFlow operation for the convolution. Note that the strides are set to `1` in all dimensions.

Indeed, the first and last stride must always be `1`, because the first is for the image number and the last is for the input channel. The `padding` parameter is set to '`SAME`' which means the input image is padded with zeroes so that the size of the output is the same:

```
|     conv1 = tf.nn.conv2d(x, w, strides=[1, 1, 1, 1],
|                           padding='SAME')
```

Then we pass the `conv1` layer to a `relu` layer. It calculates the `max(x, 0)` function for each input pixel `x`, adding some non-linearity to the formula and allows us to learn more complicated functions:

```
|     conv1 = tf.nn.relu(conv1)
```

The resulting layer is then pooled by the `tf.nn.max_pool` operator:

```
|     conv1 = tf.nn.max_pool(conv1, ksize=[1, 2, 2, 1]
|                           ,strides=[1, 2, 2, 1],
|                           padding='SAME')
```

It is a 2×2 max-pooling, which means that we are considering 2×2 windows and select the largest value in each window. Then we move two pixels to the next window.

We try to reduce the overfitting, via the `tf.nn.dropout()` function, passing the `conv1` layer and the `p_keep_conv` probability value:

```
|     conv1 = tf.nn.dropout(conv1, p_keep_conv)
```

As you can see, the next two convolutional layers, `conv2`, `conv3`, are defined in the same way as `conv1`:

```
conv2 = tf.nn.conv2d(conv1, w2,
                     strides=[1, 1, 1, 1],
                     padding='SAME')
conv2 = tf.nn.relu(conv2)
conv2 = tf.nn.max_pool(conv2, ksize=[1, 2, 2, 1],
                      strides=[1, 2, 2, 1],
                      padding='SAME')
conv2 = tf.nn.dropout(conv2, p_keep_conv)

conv3 = tf.nn.conv2d(conv2, w3,
                     strides=[1, 1, 1, 1]
                     ,padding='SAME')

conv3_a = tf.nn.relu(conv3)
```

Two fully-connected layers are added to the network. The input of the first `FC_layer` is the convolution layer from the previous convolution:

```
FC_layer = tf.nn.max_pool(conv3, ksize=[1, 2, 2, 1],
                           strides=[1, 2, 2, 1],
                           padding='SAME')

FC_layer = tf.reshape(FC_layer, [-1, w4.get_shape().as_list()[0]])
```

A `dropout` function is again used to reduce the overfitting:

```
|   FC_layer = tf.nn.dropout(FC_layer, p_keep_conv)
```

The output layer receives the input as `FC_layer` and the `w4` weight tensor. A `relu` and a `dropout` operator are respectively applied:

```
|   output_layer = tf.nn.relu(tf.matmul(FC_layer, w4))
|   output_layer = tf.nn.dropout(output_layer, p_keep_hidden)
```

The `result` variable is a vector of length `10` for determining which one of the `10` classes for the input image belongs to:

```
|   result = tf.matmul(output_layer, w_o)
|   return result
```

The **cross-entropy** is the performance measure we used in this classifier. The cross-entropy is a continuous function that is always positive and is equal to zero, if the predicted output exactly matches the desired output. The goal of this optimization is therefore to minimize the cross-entropy so it gets as close to zero as possible by changing the variables of the network layers.

TensorFlow has a built-in function for calculating the cross-entropy. Note that the function calculates the softmax internally so we must use the output of `py_x` directly:

```
| py_x = model(X, w, w2, w3, w4, w_o, p_keep_conv, p_keep_hidden)
| Y_ = tf.nn.softmax_cross_entropy_with_logits(logits=py_x, labels=Y)
```

Now that we have defined the cross-entropy for each classified image, we have a measure of how well the model performs on each image individually. But using the cross-entropy to guide the optimization of the networks's variables we need a single scalar value, so we simply take the average

of the cross-entropy for all the classified images:

```
| cost = tf.reduce_mean(Y_)
```

To minimize the evaluated cost, we must define an optimizer. In this case, we adopt the implemented `RMSPropOptimizer` function which is an advanced form of gradient descent.

The `RMSPropOptimizer` function implements the `RMSProp` algorithm, that is an unpublished, adaptive learning rate method proposed by Geoff Hinton in Lecture 6e of his coursera class.

You can find Geoff Hinton's course at <https://www.coursera.org/learn/neural-networks>.



The `RMSPropOptimizer` function also divides the learning rate by an exponentially decaying average of squared gradients. Hinton suggests setting the decay parameter to `0.9`, while a good default value for the learning rate is `0.001`:

```
| optimizer = tf.train.RMSPropOptimizer(0.001, 0.9).minimize(cost)
```

Basically, the common **Stochastic Gradient Descent (SGD)** algorithm has a problem in that learning rates must scale with $1/T$ to get convergence, where T is the iteration number. `RMSProp` tries to get around this by automatically adjusting the step size so that the step is on the same scale as the gradients, as the average gradient gets smaller, the coefficient in the SGD update gets bigger to compensate.

*An interesting reference about this algorithm can be found here:
http://www.cs.toronto.edu/%7Etijmen/csc321/slides/lecture_slides_lec6.pdf.*

Finally, we define `predict_op` that is the index with the largest value across dimensions from the output of the mode:

```
| predict_op = tf.argmax(py_x, 1)
```

Note that optimization is not performed at this point. Nothing is calculated at all; we'll just add the optimizer object to the TensorFlow graph for later execution.

We now come to define the network's running session: There are 55,000 images in the training set, so it takes a long time to calculate the gradient of the model using all these images. Therefore, we'll use a small batch of images in each iteration of the optimizer. If your computer crashes or becomes very slow because you run out of RAM, then you can try and lower this number, but you may then need to perform more optimization iterations.

Now we can proceed to implement a TensorFlow session:

```
| with tf.Session() as sess:
```

```
sess.run(tf.global_variables_initializer())
for i in range(100):
```

We get a batch of training examples, the `training_batch` tensor now holds a subset of images and corresponding labels:

```
training_batch = zip(range(0, len(trX), batch_size),
                     range(batch_size,
                           len(trX)+1,
                           batch_size))
```

Put the batch into `feed_dict` with the proper names for placeholder variables in the graph. We run the optimizer using this batch of training data, TensorFlow assigns the variables in a feed to the placeholder variables and then runs the optimizer:

```
for start, end in training_batch:
    sess.run(optimizer, feed_dict={X: trX[start:end],
                                    Y: trY[start:end],
                                    p_keep_conv: 0.8,
                                    p_keep_hidden: 0.5})
```

At the same time, we get a shuffled batch of test samples:

```
test_indices = np.arange(len(teX))
np.random.shuffle(test_indices)
test_indices = test_indices[0:test_size]
```

For each iteration, we display the accuracy evaluated on the batch set:

```
print(i, np.mean(np.argmax(teY[test_indices], axis=1) ==
                 sess.run
                 (predict_op,
                  feed_dict={X: teX[test_indices],
                            Y: teY[test_indices],
                            p_keep_conv: 1.0,
                            p_keep_hidden: 1.0})))
```

Training a network can take several hours depending on how much computational resources it uses. The results on my machine are as follows:

```
Successfully downloaded train-images-idx3-ubyte.gz 9912422 bytes.
Successfully extracted to train-images-idx3-ubyte.mnist 9912422 bytes.
Loading ata/train-images-idx3-ubyte.mnist
Successfully downloaded train-labels-idx1-ubyte.gz 28881 bytes.
Successfully extracted to train-labels-idx1-ubyte.mnist 28881 bytes.
Loading ata/train-labels-idx1-ubyte.mnist
Successfully downloaded t10k-images-idx3-ubyte.gz 1648877 bytes.
Successfully extracted to t10k-images-idx3-ubyte.mnist 1648877 bytes.
Loading ata/t10k-images-idx3-ubyte.mnist
Successfully downloaded t10k-labels-idx1-ubyte.gz 4542 bytes.
Successfully extracted to t10k-labels-idx1-ubyte.mnist 4542 bytes.
Loading ata/t10k-labels-idx1-ubyte.mnist
(0, 0.95703125)
(1, 0.98046875)
(2, 0.9921875)
(3, 0.99609375)
(4, 0.99609375)
(5, 0.98828125)
(6, 0.99609375)
(7, 0.99609375)
(8, 0.98828125)
(9, 0.98046875)
(10, 0.99609375)
```

```
.  
(90, 1.0)  
(91, 0.9921875)  
(92, 0.9921875)  
(93, 0.99609375)  
(94, 1.0)  
(95, 0.98828125)  
(96, 0.98828125)  
(97, 0.99609375)  
(98, 1.0)  
(99, 0.99609375)
```

After 10,000 iterations, the model has an accuracy of about 99% no bad!!

Source code for a handwritten classifier

For a better understanding, we reported the entire source code for the CNN previously discussed:

```
import tensorflow as tf
import numpy as np
from tensorflow.examples.tutorials.mnist import input_data

batch_size = 128
test_size = 256
img_size = 28
num_classes = 10

def init_weights(shape):
    return tf.Variable(tf.random_normal(shape, stddev=0.01))

def model(X, w, w2, w3, w4, w_o, p_keep_conv, p_keep_hidden):
    conv1 = tf.nn.conv2d(X, w,
                        strides=[1, 1, 1, 1],
                        padding='SAME')

    conv1_a = tf.nn.relu(conv1)
    conv1 = tf.nn.max_pool(conv1_a, ksize=[1, 2, 2, 1],
                          strides=[1, 2, 2, 1],
                          padding='SAME')
    conv1 = tf.nn.dropout(conv1, p_keep_conv)

    conv2 = tf.nn.conv2d(conv1, w2,
                        strides=[1, 1, 1, 1],
                        padding='SAME')
    conv2_a = tf.nn.relu(conv2)
    conv2 = tf.nn.max_pool(conv2_a, ksize=[1, 2, 2, 1],
                          strides=[1, 2, 2, 1],
                          padding='SAME')
    conv2 = tf.nn.dropout(conv2, p_keep_conv)

    conv3 = tf.nn.conv2d(conv2, w3,
                        strides=[1, 1, 1, 1]
                        ,padding='SAME')

    conv3 = tf.nn.relu(conv3)

    FC_layer = tf.nn.max_pool(conv3, ksize=[1, 2, 2, 1],
                              strides=[1, 2, 2, 1],
                              padding='SAME')

    FC_layer = tf.reshape(FC_layer, [-1, w4.get_shape().as_list()[0]])
    FC_layer = tf.nn.dropout(FC_layer, p_keep_conv)

    output_layer = tf.nn.relu(tf.matmul(FC_layer, w4))
    output_layer = tf.nn.dropout(output_layer, p_keep_hidden)

    result = tf.matmul(output_layer, w_o)
    return result

mnist = input_data.read_data_sets("MNIST_data", one_hot=True)
trX, trY, teX, teY = mnist.train.images,
                     mnist.train.labels,
                     mnist.test.images,
                     mnist.test.labels

trX = trX.reshape(-1, img_size, img_size, 1) # 28x28x1 input img
teX = teX.reshape(-1, img_size, img_size, 1) # 28x28x1 input img

X = tf.placeholder("float", [None, img_size, img_size, 1])
Y = tf.placeholder("float", [None, num_classes])

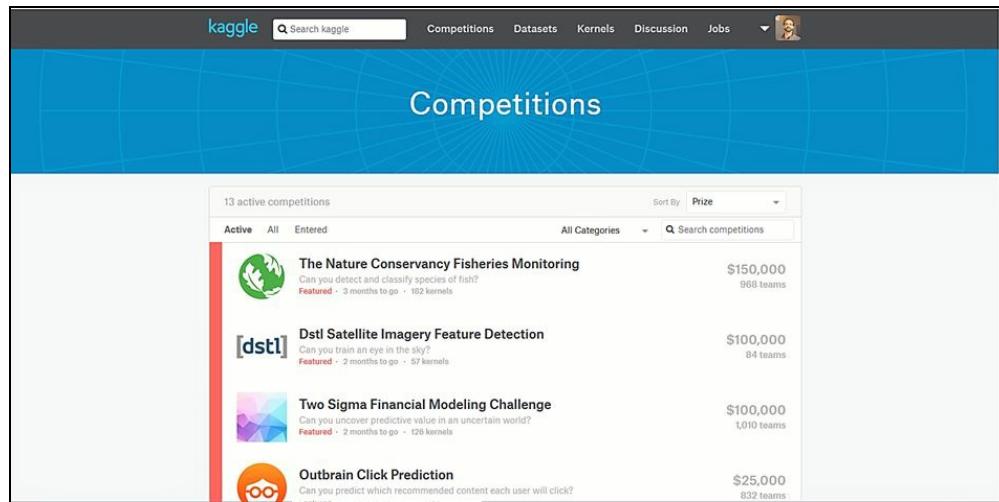
w = init_weights([3, 3, 1, 32])
```


Emotion recognition with CNNs

One of the hardest problems to solve in deep learning has nothing to do with neural nets, it's the problem of getting the right data in the right format. However, a valuable assistant to find new problems, and new datasets to study, comes from the Kaggle platform (<https://www.kaggle.com/>).

The Kaggle platform was founded in 2010 as a platform for predictive modeling and analytics competitions on which companies and researchers post their data and statisticians and data miners from all over the world compete to produce the best models.

In this section, we show how to make a CNN for emotion detection from facial images. The train and test set of this example can be downloaded from <https://inclass.kaggle.com/c/facial-keypoints-detector/data>. Please note that you can login and download the data using Facebook, Google+ or Yahoo. Alternatively, you will have to create an account and you can download the dataset.



The Kaggle competition page

The train set consists of 3,761 grayscale images of 48x48 pixels in size and a 3,761 label set of seven elements each.

Each element encodes an emotional stretch, 0 = anger, 1 = disgust, 2 = fear, 3 = happy, 4 = sad, 5 = surprise, 6 = neutral.

In a classic Kaggle competition, the set of labels obtained from the test set must be evaluated by subjecting it to the platform. In this example, we will train a neural network from a training set, after which we will evaluate the model on a single image.

Before starting the CNN implementation, we'll look at the downloaded data by implementing a simple procedure.

Import the libraries with the following code:

```
| import numpy as np
```

```
| from matplotlib import pyplot as plt  
| import EmotionDetectorUtils
```

Please note that there is a dependency on `EmotionDetectorUtils` and `EmotionDetectorUtils` uses `pandas` package to execute these codes. Now to install the `pandas` package use the following command on terminal on Ubuntu:

```
| sudo apt-get update  
| sudo apt-get install python-pip  
| sudo pip install numpy  
| sudo pip install pandas  
| sudo apt-get install python-pandas
```

The `read_data` function allows you to build all the datasets starting with the downloaded data. You can find it in the `EmotionDetectorUtils` library, which you can download in the code repository for this book:

```
| FLAGS = tf.flags.FLAGS  
| tf.flags.DEFINE_string("data_dir", "EmotionDetector/", "Path to data files")  
  
| train_images, train_labels, valid_images, valid_labels, test_images =  
|     EmotionDetectorUtils.read_data(FLAGS.data_dir)
```

Then print the shape of the training images set and test sets:

```
| print "train images shape = ",train_images.shape  
| print "test labels shape = ",test_images.shape
```

Display the first image of the training set and its correct label:

```
| image_0 = train_images[0]  
| label_0 = train_labels[0]  
| print "image_0 shape = ",image_0.shape  
| print "label set = ",label_0  
| image_0 = np.resize(image_0, (48,48))  
  
| plt.imshow(image_0, cmap='Greys_r')  
| plt.show()
```

There are 3761 grayscale images of 48x48 pixels size:

```
| train images shape = (3,761, 48, 48, 1)
```

There are 3761 class labels, each class contains seven elements:

```
| train labels shape = (3,761, 7)
```

The test set is formed by 1312 grayscale images of 48x48 pixel size:

```
| test labels shape = (1,312, 48, 48, 1)
```

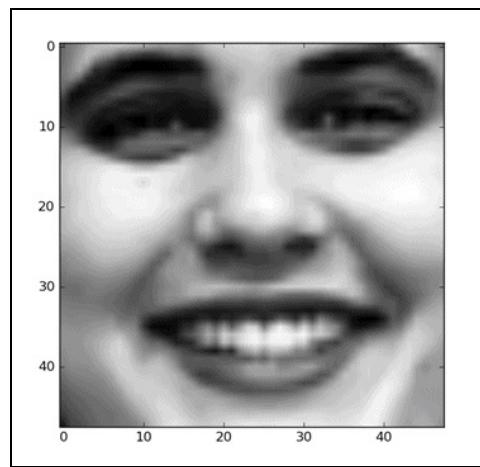
A single image has the following shape:

```
| image_0 shape = (48, 48, 1)
```

The label set for the first image is:

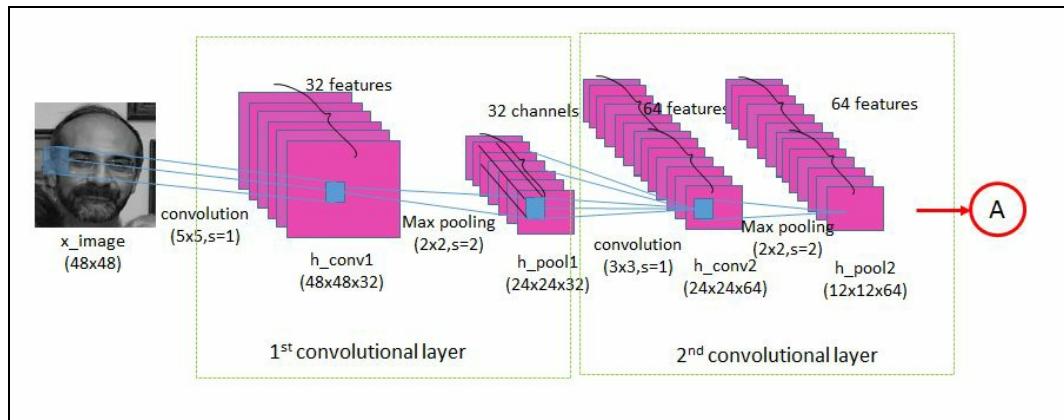
```
| label set = [ 0.  0.  0.  1.  0.  0.  0.]
```

It corresponds to a happy emotional stretch, that we visualize in the following matplotlib figure:



The first image from the emotion detection face dataset

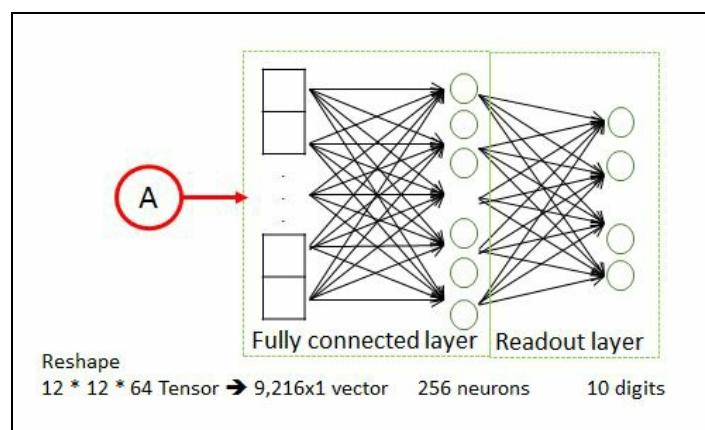
We shall now proceed to study the CNN architecture. The following figure shows how the data flows in the CNN that will be implemented:



The first two convolutional layers of the implemented CNN

The network has two convolutional layers, two fully-connected layers and finally a softmax classification layer. The input image (48 pixel) is processed in the first convolutional layer using 5x5 convolutional kernels. This results in 32 features, one for each filter used. The images are also downsampled by a max-pooling operation, to decrease the images from 48x48 to 24x24 pixels. These 32 smaller images are then processed by a second convolutional layer; this results in 64 new features (see the preceding figure). The resulting images are downsampled again to 12x12 pixels, by a second pooling operation.

The output of this second pooling layer is formed by 64 images of 12x12 pixels each. These are then flattened to a single vector of length $12 \times 12 \times 64 = 9,126$, which is used as the input to a fully-connected layer with 256 neurons. This feeds into another fully-connected layer with 10 neurons, one for each of the classes, which is used to determine the class of the image, that is, which decodes the emotion in depicted in the image.



The last two layers of the implemented CNN

Let's go onto the `weights` and `bias` definition. The following data structure, represents the definition of the network's weights and summarizes what we have previously described:

```
weights = {
    'wc1': weight_variable([5, 5, 1, 32], name="W_conv1"),
    'wc2': weight_variable([3, 3, 32, 64], name="W_conv2"),
    'wf1': weight_variable(
        [(IMAGE_SIZE / 4) * (IMAGE_SIZE / 4) * 64,
         256], name="W_fc1"),
    'wf2': weight_variable([256, NUM_LABELS], name="W_fc2")
}
```

Note again, that the convolutional filters are randomly chosen, so the classification is done randomly:

```
def weight_variable(shape, stddev=0.02, name=None):
    initial = tf.truncated_normal(shape, stddev=stddev)
    if name is None:
        return tf.Variable(initial)
    else:
        return tf.get_variable(name, initializer=initial)
```

In a similar way, we have defined `bias_variable`:

```
biases = {
    'bc1': bias_variable([32], name="b_conv1"),
    'bc2': bias_variable([64], name="b_conv2"),
    'bf1': bias_variable([256], name="b_fc1"),
    'bf2': bias_variable([NUM_LABELS], name="b_fc2")
}

def bias_variable(shape, name=None):
    initial = tf.constant(0.0, shape=shape)
    if name is None:
        return tf.Variable(initial)
    else:
        return tf.get_variable(name, initializer=initial)
```

An optimizer must propagate the error back through the CNN using the chain rule of differentiation and update the filter weights to improve the classification error. The error between the predicted and true class of the input image is measured by the implemented `loss` function. It takes in the input the predicted output of the `pred` model to the desired `label` output:

```
def loss(pred, label):
    cross_entropy_loss=
        tf.nn.softmax_cross_entropy_with_logits(pred, label)
    cross_entropy_loss= tf.reduce_mean(cross_entropy_loss)
```

```
| reg_losses = tf.add_n(tf.get_collection("losses"))
| return cross_entropy_loss + REGULARIZATION * reg_losses
```

The `tf.nn.softmax_cross_entropy_with_logits(pred, label)` function computes the `cross_entropy_loss` of the result after applying the `softmax` function (but it does it altogether in a more mathematically careful way). It's like the result of:

```
| a = tf.nn.softmax(x)
| b = cross_entropy(a)
```

We calculate the `cross_entropy_loss` function for each of the classified images so that we'll have a measure of how well the model performs on each image individually.

We take the cross-entropy's average for the classified images:

```
| cross_entropy_loss= tf.reduce_mean(cross_entropy_loss)
```

To prevent overfitting, we use L2 regularization that consists of inserting an additional term to the `cross_entropy_loss` function:

```
| reg_losses = tf.add_n(tf.get_collection("losses"))
| return cross_entropy_loss + REGULARIZATION * reg_losses
```

Where:

```
| def add_to_regularization_loss(W, b):
|     tf.add_to_collection("losses", tf.nn.l2_loss(W))
|     tf.add_to_collection("losses", tf.nn.l2_loss(b))
```



See <http://www.kdnuggets.com/2015/04/preventing-overfitting-neural-networks.html/2> for further reference.

We built the network's *weights* and *bias* and their optimization procedure. However, like all the implemented networks, we must start the implementation by importing all necessary libraries:

```
import tensorflow as tf
import numpy as np
import os, sys, inspect
from datetime import datetime
import EmotionDetectorUtils
```

We set the paths for storing the dataset on the computer, and the network parameters with the following code:

```
FLAGS = tf.flags.FLAGS
tf.flags.DEFINE_string("data_dir",
                      "EmotionDetector/", "Path to data files")
tf.flags.DEFINE_string("logs_dir", "logs/EmotionDetector_logs/",
                      "Path to where log files are to be saved")
tf.flags.DEFINE_string("mode", "train", "mode: train (Default)/ test")

BATCH_SIZE = 128
LEARNING_RATE = 1e-3
MAX_ITERATIONS = 1001
REGULARIZATION = 1e-2
IMAGE_SIZE = 48
```

```
| NUM_LABELS = 7
| VALIDATION_PERCENT = 0.1
```

The `emotion_cnn` function implements our model:

```
def emotion_cnn(dataset):
    with tf.name_scope("conv1") as scope:
        tf.summary.histogram("W_conv1", weights['wc1'])
        tf.summary.histogram("b_conv1", biases['bc1'])
        conv_1 = tf.nn.conv2d(dataset, weights['wc1'],
                             strides=[1, 1, 1, 1], padding="SAME")
        h_conv1 = tf.nn.bias_add(conv_1, biases['bc1'])
        h_1 = tf.nn.relu(h_conv1)
        h_pool1 = max_pool_2x2(h_1)
        add_to_regularization_loss(weights['wc1'], biases['bc1'])

    with tf.name_scope("conv2") as scope:
        tf.summary.histogram("W_conv2", weights['wc2'])
        conv_2 = tf.nn.conv2d(h_pool1, weights['wc2'],
                             strides=[1, 1, 1, 1], padding="SAME")
        h_conv2 = tf.nn.bias_add(conv_2, biases['bc2'])
        h_2 = tf.nn.relu(h_conv2)
        h_pool2 = max_pool_2x2(h_2)
        add_to_regularization_loss(weights['wc2'], biases['bc2'])

    with tf.name_scope("fc_1") as scope:
        prob=0.5
        image_size = IMAGE_SIZE / 4
        h_flat = tf.reshape(h_pool2, [-1, image_size * image_size * 64])
        tf.summary.histogram("W_fc1", weights['wf1'])
        tf.summary.histogram("b_fc1", biases['bf1'])
        h_fc1 = tf.nn.relu(tf.matmul(
            (h_flat, weights['wf1']) + biases['bf1']))
        h_fc1_dropout = tf.nn.dropout(h_fc1, prob)

    with tf.name_scope("fc_2") as scope:
        tf.summary.histogram("W_fc2", weights['wf2'])
        tf.summary.histogram("b_fc2", biases['bf2'])
        pred = tf.matmul(h_fc1_dropout, weights['wf2']) + biases['bf2']
    return pred
```

We defined a `main` function where we'll define the dataset, the input and output placeholder variables and the main session to start the training procedure:

```
| def main(argv=None):
```

The first operation in this function is to load the dataset for the training and validation phase. We'll use the training set to teach the classifier to recognize the to-be-predicted labels, and the we'll use the validation set to estimate the classifier performance:

```
train_images,
        train_labels,
        valid_images,
        valid_labels,
        test_images =
        EmotionDetectorUtils.read_data(FLAGS.data_dir)
print "Train size: %s" % train_images.shape[0]
print 'Validation size: %s' % valid_images.shape[0]
print "Test size: %s" % test_images.shape[0]
```

We define the placeholder variable for the input images. This allows us to change the images that are input to the TensorFlow graph. The datatype is set to `float32` and the shape is set to `[None, IMG_SIZE, IMAGE_SIZE, 1]`, where `None` means that the tensor may hold an arbitrary number of images with each image being `img_size` pixels high and `img_size` pixels wide and `1` is the number of color channels:

```
input_dataset = tf.placeholder(tf.float32,
                               [None,
                                IMAGE_SIZE,
                                IMAGE_SIZE, 1], name="input")
```

Next, we have the placeholder variable for the true labels associated with the images that were input in the `input_dataset` placeholder variable. The shape of this placeholder variable is `[None, NUM_LABELS]` which means it may hold an arbitrary number of labels and each label is a vector of length `NUM_LABELS`, which is 7 in this case:

```
input_labels = tf.placeholder(tf.float32,
                             [None, NUM_LABELS])
```

The `global_step` variable keeps track of the number of optimization iterations performed so far. We want to save this variable with all the other TensorFlow variables in the checkpoints. Note that `trainable=False` which means that TensorFlow will not try to optimize this variable:

```
| global_step = tf.Variable(0, trainable=False)
```

And the following variable, `dropout_prob`, for dropout optimization:

```
| dropout_prob = tf.placeholder(tf.float32)
```

Now create the neural network for the test phase. The `emotion_cnn()` function returns the predicted `pred` class labels for the `input_dataset` variable:

```
| pred = emotion_cnn(input_dataset)
```

The `output_pred` variable is the predictions for the test and validation, which we'll compute in the running session:

```
| output_pred = tf.nn.softmax(pred, name="output")
```

The `loss_val` variable contains the error between the `pred` predicted class and the true class of the input image (`input_labels`):

```
| loss_val = loss(pred, input_labels)
```

The `train_op` variable defines the optimizer used to minimize the `cost` function. In this case again we use `AdamOptimizer`:

```
| train_op = tf.train.AdamOptimizer
          (LEARNING_RATE).minimize
          (loss_val, global_step)
```

And `summary_op` for TensorBoard visualizations:

```
| summary_op = tf.merge_all_summaries()
```

Once the graph has been created, we need to create a TensorFlow session which is used to execute the graph:

```
| with tf.Session() as sess:
```

```
sess.run(tf.global_variables_initializer())
summary_writer = tf.summary.FileWriter(FLAGS.logs_dir, sess.graph_def)
```

We define a `saver` variable to restore the model:

```
saver = tf.train.Saver()
ckpt = tf.train.get_checkpoint_state(FLAGS.logs_dir)
if ckpt and ckpt.model_checkpoint_path:
    saver.restore(sess, ckpt.model_checkpoint_path)
    print "Model Restored!"
```

Get a batch of training examples, `batch_image` now holds a batch of images and `batch_label` are the true labels for those images:

```
for step in xrange(MAX_ITERATIONS):
    batch_image, batch_label = get_next_batch(train_images,
                                                train_labels,
                                                step)
```

We put the batch into a `dict` variable with the proper names for placeholder variables in the TensorFlow graph:

```
feed_dict = {input_dataset: batch_image,
            input_labels: batch_label}
```

We run the optimizer using this batch of training data; TensorFlow assigns the variables in `feed_dict_train` to the placeholder variables and then runs the optimizer:

```
sess.run(train_op, feed_dict=feed_dict)
if step % 10 == 0:
    train_loss,
    summary_str =
        sess.run([loss_val, summary_op],
                feed_dict=feed_dict)
    summary_writer.add_summary(summary_str,
                                global_step=step)
    print "Training Loss: %f" % train_loss
```

When the running step is a multiple of `100`, we run the trained model on the validation set:

```
if step % 100 == 0:
    valid_loss =
        sess.run(loss_val,
                feed_dict={input_dataset:
                           valid_images,
                           input_labels:
                           valid_labels})
```

Print the `loss` value:

```
print "%s Validation Loss: %f"
      % (datetime.now(), valid_loss)
```

At the end of the training session the model is saved:

```
saver.save(sess, FLAGS.logs_dir
           + 'model.ckpt',
           global_step=step)

if __name__ == "__main__":
```

Now we report the resulting output. As you can see the loss function decreased during the following simulation:

```
>>>
Train size: 3761
Validation size: 417
Test size: 1312
2016-11-05 22:39:36.645682 Validation Loss: 1.962719
2016-11-05 22:42:58.951699 Validation Loss: 1.822431
2016-11-05 22:46:55.144483 Validation Loss: 1.335237
2016-11-05 22:50:17.677074 Validation Loss: 1.111559
2016-11-05 22:53:30.999141 Validation Loss: 0.999061
2016-11-05 22:56:53.256991 Validation Loss: 0.931223
2016-11-05 23:00:06.530139 Validation Loss: 0.911489
2016-11-05 23:03:15.351156 Validation Loss: 0.818303
2016-11-05 23:06:26.575298 Validation Loss: 0.824178
2016-11-05 23:09:40.136353 Validation Loss: 0.803449
2016-11-05 23:12:50.769527 Validation Loss: 0.851074
>>>
```

However, the model can be improved by acting on hyperparameters or changing its architecture. In the next section, we will see how to effectively test the model on your own images.

Source code for emotion classifier

In the following code, we have quoted the source code of the implemented emotion classifier:

```
import tensorflow as tf
import numpy as np
import os, sys, inspect
from datetime import datetime
import EmotionDetectorUtils

FLAGS = tf.flags.FLAGS
tf.flags.DEFINE_string("data_dir", "EmotionDetector/", "Path to data files")
tf.flags.DEFINE_string("logs_dir", "logs/EmotionDetector_logs/", "Path to where log files are to be saved")
tf.flags.DEFINE_string("mode", "train", "mode: train (Default)/ test")

BATCH_SIZE = 128
LEARNING_RATE = 1e-3
MAX_ITERATIONS = 1001
REGULARIZATION = 1e-2
IMAGE_SIZE = 48
NUM_LABELS = 7
VALIDATION_PERCENT = 0.1

def add_to_regularization_loss(W, b):
    tf.add_to_collection("losses", tf.nn.l2_loss(W))
    tf.add_to_collection("losses", tf.nn.l2_loss(b))

def weight_variable(shape, stddev=0.02, name=None):
    initial = tf.truncated_normal(shape, stddev=stddev)
    if name is None:
        return tf.Variable(initial)
    else:
        return tf.get_variable(name, initializer=initial)

def bias_variable(shape, name=None):
    initial = tf.constant(0.0, shape=shape)
    if name is None:
        return tf.Variable(initial)
    else:
        return tf.get_variable(name, initializer=initial)

def conv2d_basic(x, W, bias):
    conv = tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding="SAME")
    return tf.nn.bias_add(conv, bias)

def max_pool_2x2(x):
    return tf.nn.max_pool(x, ksize=[1, 2, 2, 1],
                          strides=[1, 2, 2, 1], padding="SAME")

def emotion_cnn(dataset):
    with tf.name_scope("conv1") as scope:
        tf.summary.histogram("W_conv1", weights['wc1'])
        tf.summary.histogram("b_conv1", biases['bc1'])
        conv_1 = tf.nn.conv2d(dataset, weights['wc1'],
                             strides=[1, 1, 1, 1], padding="SAME")
        h_conv1 = tf.nn.bias_add(conv_1, biases['bc1'])
        h_1 = tf.nn.relu(h_conv1)
        h_pool1 = max_pool_2x2(h_1)
        add_to_regularization_loss(weights['wc1'], biases['bc1'])

    with tf.name_scope("conv2") as scope:
        tf.summary.histogram("W_conv2", weights['wc2'])
        tf.summary.histogram("b_conv2", biases['bc2'])
        conv_2 = tf.nn.conv2d(h_pool1, weights['wc2'],
                             strides=[1, 1, 1, 1], padding="SAME")
        h_conv2 = tf.nn.bias_add(conv_2, biases['bc2'])
        #h_2 = conv2d_basic(h_pool1, weights['wc2'], biases['bc2'])
        h_2 = tf.nn.relu(h_conv2)
        h_pool2 = max_pool_2x2(h_2)
        add_to_regularization_loss(weights['wc2'], biases['bc2'])
```

```

with tf.name_scope("fc_1") as scope:
    prob=0.5
    image_size = IMAGE_SIZE / 4
    h_flat = tf.reshape(h_pool2, [-1, image_size * image_size *
                                    64])
    tf.summary.histogram("W_fc1", weights['wf1'])
    tf.summary.histogram("b_fc1", biases['bf1'])
    h_fc1 = tf.nn.relu(tf.matmul(h_flat, weights['wf1']) +
                       biases['bf1'])
    h_fc1_dropout = tf.nn.dropout(h_fc1, prob)

with tf.name_scope("fc_2") as scope:
    tf.summary.histogram("W_fc2", weights['wf2'])
    tf.summary.histogram("b_fc2", biases['bf2'])
    #pred = tf.matmul(h_fc1, weights['wf2']) + biases['bf2']
    pred = tf.matmul(h_fc1_dropout, weights['wf2']) + biases['bf2']

return pred

weights = {
    'wc1': weight_variable([5, 5, 1, 32], name="W_conv1"),
    'wc2': weight_variable([3, 3, 32, 64], name="W_conv2"),
    'wf1': weight_variable([(IMAGE_SIZE / 4) *
                           (IMAGE_SIZE / 4) * 64, 256], name="W_fc1"),
    'wf2': weight_variable([256, NUM_LABELS], name="W_fc2")
}

biases = {
    'bc1': bias_variable([32], name="b_conv1"),
    'bc2': bias_variable([64], name="b_conv2"),
    'bf1': bias_variable([256], name="b_fc1"),
    'bf2': bias_variable([NUM_LABELS], name="b_fc2")
}

def loss(pred, label):
    cross_entropy_loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=pred, labels=label))
    tf.summary.scalar('Entropy', cross_entropy_loss)
    reg_losses = tf.add_n(tf.get_collection("losses"))
    tf.scalar_summary('Reg_loss', reg_losses)
    return cross_entropy_loss + REGULARIZATION * reg_losses

def train(loss, step):
    return tf.train.AdamOptimizer(LEARNING_RATE).minimize(loss, global_step=step)

def get_next_batch(images, labels, step):
    offset = (step * BATCH_SIZE) % (images.shape[0] - BATCH_SIZE)
    batch_images = images[offset: offset + BATCH_SIZE]
    batch_labels = labels[offset:offset + BATCH_SIZE]
    return batch_images, batch_labels

def main(argv=None):
    train_images,
        train_labels,
        valid_images,
        valid_labels,
        test_images =
            EmotionDetectorUtils.read_data(FLAGS.data_dir)
    print "Train size: %s" % train_images.shape[0]
    print 'Validation size: %s' % valid_images.shape[0]
    print "Test size: %s" % test_images.shape[0]

    global_step = tf.Variable(0, trainable=False)
    dropout_prob = tf.placeholder(tf.float32)
    input_dataset = tf.placeholder(tf.float32,
                                  [None,
                                   IMAGE_SIZE,
                                   IMAGE_SIZE, 1], name="input")
    input_labels = tf.placeholder(tf.float32,
                                 [None, NUM_LABELS])

    pred = emotion_cnn(input_dataset)
    output_pred = tf.nn.softmax(pred, name="output")
    loss_val = loss(pred, input_labels)

```

```

train_op = train(loss_val, global_step)
summary_op = tf.summary.merge_all()
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    summary_writer = tf.summary.FileWriter(FLAGS.logs_dir,
    sess.graph_def)
saver = tf.train.Saver()
ckpt = tf.train.get_checkpoint_state(FLAGS.logs_dir)
if ckpt and ckpt.model_checkpoint_path:
    saver.restore(sess, ckpt.model_checkpoint_path)
    print "Model Restored!"

for step in xrange(MAX_ITERATIONS):
    batch_image, batch_label = get_next_batch(train_images,
                                                train_labels,
                                                step)
    feed_dict = {input_dataset: batch_image,
                input_labels: batch_label}

    sess.run(train_op, feed_dict=feed_dict)
    if step % 10 == 0:
        train_loss,
        summary_str = sess.run([loss_val,
                               summary_op],
                               feed_dict=feed_dict)
        summary_writer.add_summary(summary_str,
                                   global_step=step)
        print "Training Loss: %f" % train_loss

    if step % 100 == 0:
        valid_loss =
            sess.run(loss_val,
                     feed_dict={input_dataset:
                                valid_images,
                     input_labels:
                                valid_labels})
        print "%s Validation Loss: %f"
        % (datetime.now(), valid_loss)
    saver.save(sess, FLAGS.logs_dir
               + 'model.ckpt',
               global_step=step)

if __name__ == "__main__":
    tf.app.run()

```


Testing the model on your own image

The dataset we use is standardized. All faces are pointed exactly at the camera and the emotional expressions are exaggerated, and even comical in some situations. Now let's see what happens if we use a more natural image. First, we need to make sure that there is no text overlaid on the face, the emotion is recognizable, and the face is pointed mostly at the camera.

I started with this `.jpg` image (it's a color image, as you can download it from the book's code repository):



The input image for our test

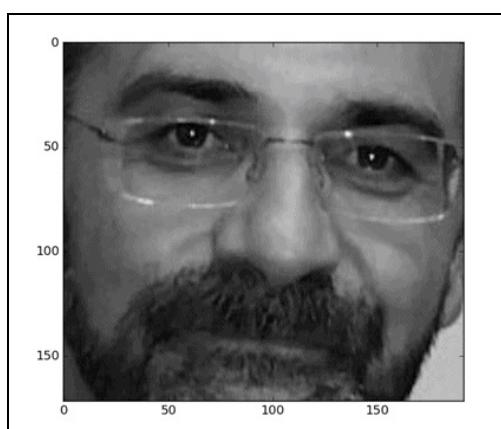
Using matplotlib and other NumPy Python libraries, we convert the input color image into a valid input for the network, that is, a grayscale image:

```
img = mpimg.imread('author_image.jpg')
gray = rgb2gray(img)
```

The conversion function is:

```
def rgb2gray(rgb):
    return np.dot(rgb[...,:3], [0.299, 0.587, 0.114])
```

The result is shown in the following figure:



The grayscaled input image

Finally, we can feed the network with this image; but first we must define a TensorFlow running

session:

```
| sess = tf.InteractiveSession()
```

Then we can recall the previously saved model:

```
new_saver = tf.train.  
            import_meta_graph('logs/model.ckpt-1000.meta')  
new_saver.restore(sess, 'logs/model.ckpt-1000')  
tf.get_default_graph().as_graph_def()  
x = sess.graph.get_tensor_by_name("input:0")  
y_conv = sess.graph.get_tensor_by_name("output:0")
```

To test an image, we must reshape it into a 48x48x1 valid format for the network:

```
| image_test = np.resize(gray, (1, 48, 48, 1))
```

We evaluated the same picture several times ($_{1000}$) in order to build a percentage of possible emotional stretches of the input image:

```
tResult = testResult()  
num_evaluations = 1000  
for i in range(0,num_evaluations):  
    result = sess.run(y_conv, feed_dict={x:image_test})  
    label = sess.run(tf.argmax(result, 1))  
    label = label[0]  
    label = int(label)  
    tResult.evaluate(label)  
  
tResult.display_result(num_evaluations)
```

After a few seconds a result like this should appear:

```
>>>  
anger = 0.1%  
disgust = 0.1%  
fear = 29.1%  
happy = 50.3%  
sad = 0.1%  
surprise = 20.0%  
neutral = 0.3%  
>>>
```

The maximum percentage confirms ($_{\text{happy} = 50.3\%}$) that we are on the right track, but of course this doesn't mean that our model is accurate. Possible improvements can result from a greater and more diverse training set, intervening in the network's parameters, or modifying the network architecture.

Source code

The second part of the implemented classifier is listed here:

```
from scipy import misc
import numpy as np
import matplotlib.cm as cm
import tensorflow as tf
from matplotlib import pyplot as plt
import matplotlib.image as mpimg
import EmotionDetectorUtils
from EmotionDetectorUtils import testResult

def rgb2gray(rgb):
    return np.dot(rgb[...,:3], [0.299, 0.587, 0.114])

img = mpimg.imread('author_image.jpg')
gray = rgb2gray(img)
plt.imshow(gray, cmap = plt.get_cmap('gray'))
plt.show()

sess = tf.InteractiveSession()
new_saver = tf.train.import_meta_graph('logs/model.ckpt-1000.meta')
new_saver.restore(sess, 'logs/model.ckpt-1000')
tf.get_default_graph().as_graph_def()
x = sess.graph.get_tensor_by_name("input:0")
y_conv = sess.graph.get_tensor_by_name("output:0")

image_test = np.resize(gray, (1,48,48,1))
tResult = testResult()
num_evaluations = 1000
for i in range(0,num_evaluations):
    result = sess.run(y_conv, feed_dict={x:image_test})
    label = sess.run(tf.argmax(result, 1))
    label = label[0]
    label = int(label)
    tResult.evaluate(label)

tResult.display_result(num_evaluations)
```

We implement the `testResult` Python class to display the resulting percentages. It can be found in the `EmotionDetectorUtils.py` file.

Here we report the implementation of this class:

```
class testResult:

    def __init__(self):
        self.anger = 0
        self.disgust = 0
        self.fear = 0
        self.happy = 0
        self.sad = 0
        self.surprise = 0
        self.neutral = 0

    def evaluate(self,label):

        if (0 == label):
            self.anger = self.anger+1
        if (1 == label):
            self.disgust = self.disgust+1
        if (2 == label):
            self.fear = self.fear+1
        if (3 == label):
```

```
    self.happy = self.happy+1
if (4 == label):
    self.sad = self.sad+1
if (5 == label):
    self.surprise = self.surprise+1
if (6 == label):
    self.neutral = self.neutral+1

def display_result(self,evaluations):
    print("anger = " + str((self.anger/float(evaluations))*100) + "%")
    print("disgust = " + str((self.disgust/float(evaluations))*100) + "%")
    print("fear = " + str((self.fear/float(evaluations))*100) + "%")
    print("happy = " + str((self.happy/float(evaluations))*100) + "%")
    print("sad = " + str((self.sad/float(evaluations))*100) + "%")
    print("surprise = " + str((self.surprise/float(evaluations))*100) + "%")
    print("neutral = " + str((self.neutral/float(evaluations))*100) + "%")
```


Summary

In this chapter, we introduced **Convolutional Neural Networks (CNNs)**.

We have seen how the architecture of these networks yield CNNs, which are particularly suitable for image classification problems, making the training phase faster and the test phase more accurate.

We have therefore implemented an image classifier, testing it on MNIST dataset, where have achieved a 99 percent accuracy.

Finally, we built a CNN to classify emotions starting from a dataset of images; we tested the network on a single image and we evaluated the limits and the goodness of our model.

The next chapter describes autoencoders, these algorithms are useful for dimensionality reduction, classification, regression, collaborative filtering, feature learning and topic modeling. We will carry out further data analysis using autoencoders and measure classification performance using image datasets.

Optimizing TensorFlow Autoencoders

A big problem that plagues all supervised learning systems is the so-called curse of dimensionality; a progressive decline in performance with an increase in the input space dimension. This occurs because the number of necessary samples to obtain a sufficient sampling of the input space increases exponentially with the number of dimensions. To overcome these problems, some optimizing networks have been developed.

The first are **autoencoder** networks, these are designed and trained for transforming an input pattern in itself, so that, in the presence of a degraded or incomplete version of an input pattern, it is possible to obtain the original pattern. The network is trained to create output data, like those presented in the entrance, and the hidden layer stores the data compressed, that is, a compact representation that captures the fundamental characteristics of the input data.

The second optimizing networks are **Boltzmann machines**. These types of networks consist of an input/output visible layer, and one hidden layer. The connections between the visible layer and the hidden one are non-directional: data can travel in both directions, visible-hidden and hidden-visible, and the different neuronal units can be fully or partially connected.

Autoencoders can be compared with **Principal Component Analysis (PCA)**, which is used to represent a given input using fewer dimensions than originally present. In this chapter, we'll focus only on autoencoders.

The topics covered are:

- Introducing autoencoders
- Implementing an autoencoder
- Improving autoencoder robustness
- Building denoising autoencoders
- Convolutional autoencoders

Introducing autoencoders

An autoencoder is a network with three or more layers, where the input layer and the output have the same number of neurons, and those intermediate (hidden) layers have a lower number of neurons. The network is trained to simply reproduce in output, for each input data, the same pattern of activity in the input.

The remarkable aspect of the problem is that, due to the lower number of neurons in the hidden layer, if the network can learn from examples, and can generalize to an acceptable extent, it performs data compression: the status of the hidden neurons provide, for each example, a compressed version of the input and output common states.

In the first examples of such networks, in the 1980s, a compression of simple images was obtained in this way. This was not far for services to that obtainable with standard methods and more complicated.

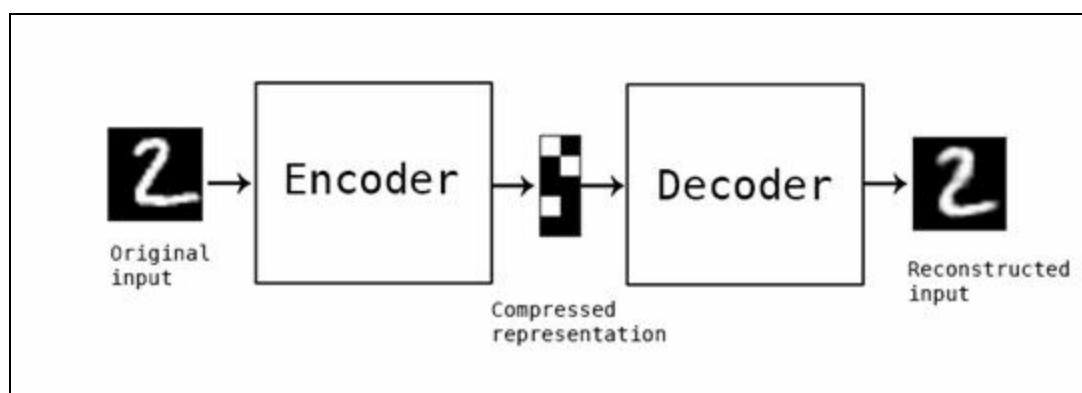
Interest in autoencoders was recently revived by some authors who have developed an efficient strategy to improve the learning process in this type of network (usually very slow and not always effective), through a prelearning procedure that provides a good initial condition of the weights for the learning procedure.



See the paper by G.E. Hinton and R. Salakhutdinov, Reducing the Dimensionality of Data with Neural Networks, 2006. Please refer <https://www.cs.toronto.edu/~hinton/science.pdf>

Useful applications of autoencoders are data denoising and dimensionality reduction for data visualization.

The following figure shows how an autoencoder typically works: it reconstructs the received input through two phases, an encoding phase, that corresponds to a dimensional reduction for the original input, and a decoding phase, capable of reconstructing the original input from the encoded (compressed) representation:



Encoder and decoder phase in autoencoder

Implementing an autoencoder

Training an autoencoder is basically a simple process. It is a neural network whose output is same as the input. The basic architecture of the autoencoder is as follows.

There is an input layer, which is followed by a few hidden layers, and then, after a certain depth, the hidden layers follow the reverse architecture until we reach a point where the final layer is the same as the input layer. We pass data into the network whose embedding we wish to learn.

In this example, we use the images input by the MNIST dataset. We begin our implementation by importing all the main libraries:

```
| import tensorflow as tf  
| import numpy as np  
| import matplotlib.pyplot as plt  
| import mnist_data
```

We then prepare the MNIST dataset. We use the `input_data` function to load and set up the data:

```
| from tensorflow.examples.tutorials.mnist import input_data  
| mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
```

Then configure the network parameters:

```
| learning_rate = 0.01  
| training_epochs = 10  
| batch_size = 256  
| display_step = 1  
| examples_to_show = 4
```

The sizes of the hidden features are as follows:

```
| n_hidden_1 = 256  
| n_hidden_2 = 128
```

The size of the input images is as follows:

```
| n_input = 784
```

The final size corresponds to $28 \times 28 = 784$ pixels.

Define a placeholder variable for the input images. The data type for this tensor is set to `float` and the shape is set to `[None, n_input]`. Defining the `None` parameter, the tensor may hold an arbitrary number of images:

```
| x = tf.placeholder("float", [None, n_input])
```

Then, we can define the weights and biases of the network. The weights data structure contains the definition of the weights for the encoder and decoder. Notice that weights are chosen using `tf.random_normal`, which returns random values with a normal distribution:

```

weights = {
    'encoder_h1': tf.Variable(
        (tf.random_normal([n_input, n_hidden_1]))),
    'encoder_h2': tf.Variable(
        (tf.random_normal([n_hidden_1, n_hidden_2]))),
    'decoder_h1': tf.Variable(
        (tf.random_normal([n_hidden_2, n_hidden_1]))),
    'decoder_h2': tf.Variable(
        (tf.random_normal([n_hidden_1, n_input]))),
}

```

Similarly, we have defined the network's bias:

```

biases = {
    'encoder_b1': tf.Variable(
        (tf.random_normal([n_hidden_1]))),
    'encoder_b2': tf.Variable(
        (tf.random_normal([n_hidden_2]))),
    'decoder_b1': tf.Variable(
        (tf.random_normal([n_hidden_1]))),
    'decoder_b2': tf.Variable(
        (tf.random_normal([n_input]))),
}

```

We split the network modeling into two complementary, fully connected networks: an encoder and a decoder.

The encoder encodes the data; it takes as input an image, x , of the MNIST dataset, and performs the data encoding:

```

encoder_in = tf.nn.sigmoid(tf.add(
    (tf.matmul(x, \
               weights['encoder_h1']), \
               biases['encoder_b1']))

```

The input data encoding is simply a matrix multiplication operation. The input data, x , of dimension 784 is reduced to a lower dimension, 256, using a matrix multiplication:

```
| (W*x + b) = encoder_in
```

Here, w is the weight tensor, encoder_h1 , and b is the bias tensor, encoder_b1 .

Through this operation, we have coded the initial image into a useful input to the autoencoder. The second step of the encoding procedure consists of data compression.

The data represented by the input encoder_in tensor is reduced to a smaller size by means of a second matrix multiplication operation:

```

encoder_out = tf.nn.sigmoid(tf.add(
    (tf.matmul(encoder_in, \
               weights['encoder_h2']), \
               biases['encoder_b2']))

```

The input data, encoder_in , of dimension 256, is then compressed to a lower tensor of size 128:

```
| (W * encoder_in + b) = encoder_out
```

Here, w stands for the weight tensor, encoder_h2 , while b stands for the bias tensor, encoder_b2 .

Notice that the activation function we used for the encoder phase is the sigmoid.

The decoder performs the inverse operation of the encoder. It decompresses the input to obtain an output of the same size of the network input. The first step of the procedure is to transform the `encoder_out` tensor of size 128 to a tensor of intermediate representation, of size 256:

```
| decoder_in = tf.nn.sigmoid(tf.add  
|     (tf.matmul(encoder_out,\n|         weights['decoder_h1']),\n|         biases['decoder_b1']))
```

In the formulas, this means the following:

```
|     (W * encoder_out + b) = decoder_in
```

Here, `W` is the weight tensor, `decoder_h1`, of size 256x128, and `b` is the bias tensor, `decoder_b1`, of size 256.

The final operation of decoding is to decompress the data from its intermediate representation (of size 256), to a final representation (of size 784), which, as you will recall, is the size of the original data:

```
| decoder_out = tf.nn.sigmoid(tf.add \  
|     (tf.matmul(decoder_in,\n|         weights['decoder_h2']),\n|         biases['decoder_b2']))
```

The `y_pred` parameter is set equal to `decoder_out`:

```
| y_pred = decoder_out
```

The network will learn if the input data, `x`, is equal to the decoded data, so we define the following:

```
| y_true = x
```

The point of the autoencoder is to create a reduction matrix that is good at reconstructing the original data. Thus, we want to minimize the `cost` function. Then we define the `cost` function as the mean squared error between `y_true` and `y_pred`:

```
| cost = tf.reduce_mean(tf.pow(y_true - y_pred, 2))
```

To optimize the `cost` function, we use following the `RMSPropOptimizer` class:

```
| optimizer = tf.train.RMSPropOptimizer(learning_rate).minimize(cost)
```

Then we prepare to launch the session:

```
| init = tf.global_variables_initializer()  
| with tf.Session() as sess:  
|     sess.run(init)
```

Set the size of batch images to train the network:

```
|     total_batch = int(mnist.train.num_examples/batch_size)
```

Start with the training cycle (the number of `training_epochs` is set to 10):

```
| for epoch in range(training_epochs):
```

While looping over all batches:

```
|   for i in range(total_batch):
|     batch_xs, batch_ys =\
|       mnist.train.next_batch(batch_size)
```

We run the optimization procedure, feeding the execution graph with the batch set, `batch_xs`:

```
|   _, c = sess.run([optimizer, cost], \
|                   feed_dict={X: batch_xs})
```

Display results per epoch step:

```
|   if epoch % display_step == 0:
|     print("Epoch:", '%04d' % (epoch+1),
|           "cost=", "{:.9f}".format(c))
|   print("Optimization Finished!")
```

Finally, we test the model, applying the encode or decode procedure. We feed the model with a subset of images, where the value of `example_to_show` is set to 4:

```
| encode_decode = sess.run(y_pred, feed_dict=\n|   {X: mnist.test.images[:examples_to_show]})
```

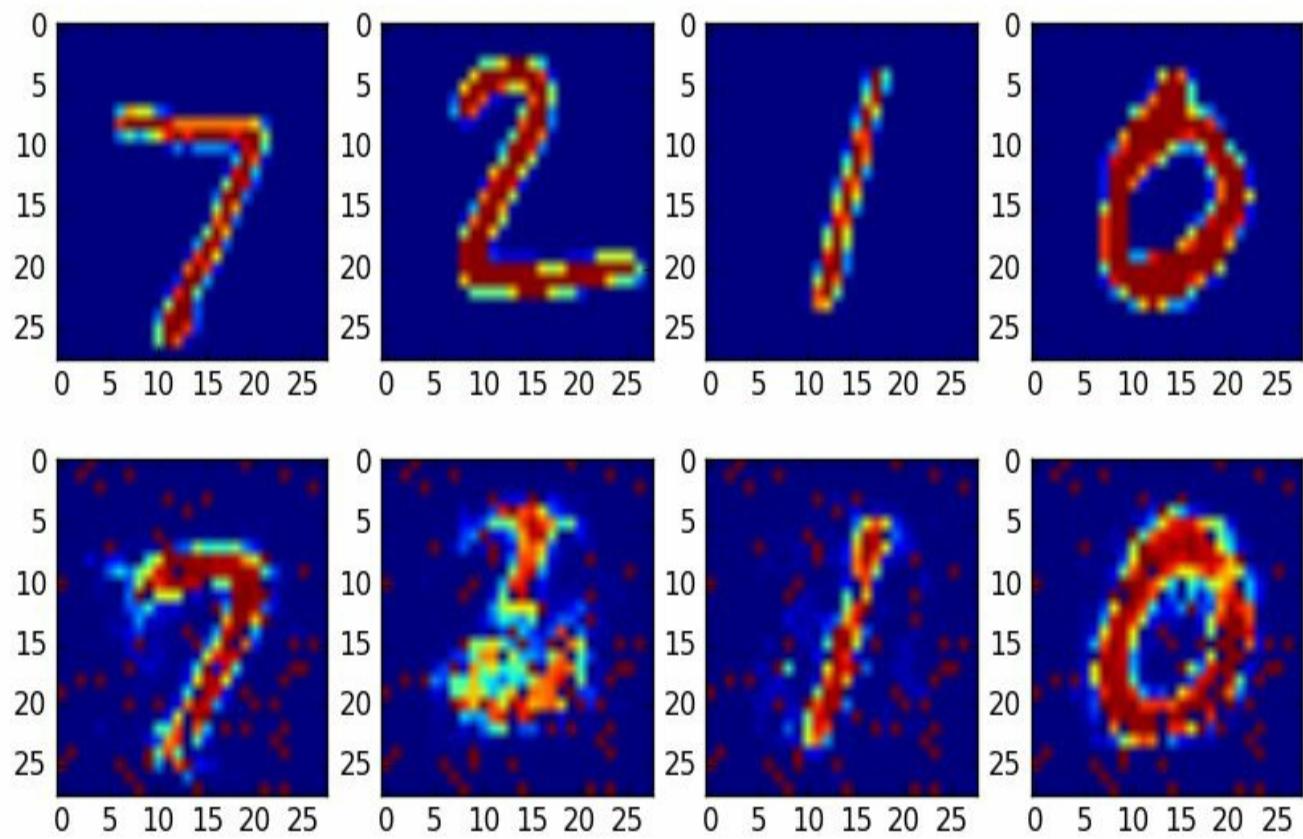
We compare the original images with their reconstructions, using matplotlib capabilities:

```
| f, a = plt.subplots(2, 4, figsize=(10, 5))
| for i in range(examples_to_show):
|   a[0][i].imshow(np.reshape(mnist.test.images[i], (28, 28)))
|   a[1][i].imshow(np.reshape(encode_decode[i], (28, 28)))
| f.show()
| plt.draw()
| plt.show()
```

Running the session, we should have an output as follows:

```
| Extracting MNIST_data/train-images-idx3-ubyte.gz
| Extracting MNIST_data/train-labels-idx1-ubyte.gz
| Extracting MNIST_data/t10k-images-idx3-ubyte.gz
| Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
| ('Epoch:', '0001', 'cost=', '0.196781039')
| ('Epoch:', '0002', 'cost=', '0.157454371')
| ('Epoch:', '0003', 'cost=', '0.139842913')
| ('Epoch:', '0004', 'cost=', '0.132784918')
| ('Epoch:', '0005', 'cost=', '0.123214975')
| ('Epoch:', '0006', 'cost=', '0.117614307')
| ('Epoch:', '0007', 'cost=', '0.111050725')
| ('Epoch:', '0008', 'cost=', '0.111332968')
| ('Epoch:', '0009', 'cost=', '0.107702859')
| ('Epoch:', '0010', 'cost=', '0.106899358')
| Optimization Finished!
```

Then we display the results; the first row has the original images, while the second row has the decoded images:



Original and decoded images

As you can see, the digit 2 differs from the original one (it looks more like the digit 3). We can increase the number of epochs or change the network parameters to improve the result.

Source code for the autoencoder

The following is the complete source code for the implemented autoencoder:

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
mnist = mnist.train.read_data_sets("data/")
learning_rate = 0.01
training_epochs = 10
batch_size = 256
display_step = 1
examples_to_show = 10
n_hidden_1 = 256 # 1st layer num features
n_hidden_2 = 128 # 2nd layer num features
n_input = 784 # MNIST data input (img shape: 28*28)
X = tf.placeholder("float", [None, n_input])
weights = {
    'encoder_h1': tf.Variable(
        tf.random_normal([n_input, n_hidden_1])),
    'encoder_h2': tf.Variable(
        tf.random_normal([n_hidden_1, n_hidden_2])),
    'decoder_h1': tf.Variable(
        tf.random_normal([n_hidden_2, n_hidden_1])),
    'decoder_h2': tf.Variable(
        tf.random_normal([n_hidden_1, n_input])),
}
biases = {
    'encoder_b1': tf.Variable(
        tf.random_normal([n_hidden_1])),
    'encoder_b2': tf.Variable(
        tf.random_normal([n_hidden_2])),
    'decoder_b1': tf.Variable(
        tf.random_normal([n_hidden_1])),
    'decoder_b2': tf.Variable(
        tf.random_normal([n_input])),}
encoder_in = tf.nn.sigmoid(tf.add(
    (tf.matmul(X, \
        weights['encoder_h1']), \
        biases['encoder_b1'])))
encoder_out = tf.nn.sigmoid(tf.add(
    (tf.matmul(encoder_in, \
        weights['encoder_h2']), \
        biases['encoder_b2'])))
decoder_in = tf.nn.sigmoid(tf.add(
    (tf.matmul(encoder_out, \
        weights['decoder_h1']), \
        biases['decoder_b1'])))
decoder_out = tf.nn.sigmoid(tf.add(
    (tf.matmul(decoder_in, \
        weights['decoder_h2']), \
        biases['decoder_b2'])))
y_pred = decoder_out
y_true = X
cost = tf.reduce_mean(tf.pow(y_true - y_pred, 2))
optimizer = tf.train.RMSPropOptimizer(learning_rate).minimize(cost)
init = tf.initialize_all_variables()
with tf.Session() as sess:
    sess.run(init)
    total_batch = int(mnist.train.num_examples/batch_size)

    for epoch in range(training_epochs):

        for i in range(total_batch):
            batch_xs, batch_ys =\
                mnist.train.next_batch(batch_size)
            _, c = sess.run([optimizer, cost], \
                feed_dict={X: batch_xs})
```

```
if epoch % display_step == 0:  
    print("Epoch:", '%04d' % (epoch+1),  
          "cost=", "{:.9f}".format(c))  
print("Optimization Finished!")  
encode_decode = sess.run(  
    y_pred, feed_dict=  
    {X: mnist.test.images[:examples_to_show]})  
  
f, a = plt.subplots(2, 4, figsize=(10, 5))  
for i in range(examples_to_show):  
    a[0][i].imshow(np.reshape(mnist.test.images[i], (28, 28)))  
    a[1][i].imshow(np.reshape(encode_decode[i], (28, 28)))  
f.show()  
plt.draw()  
plt.show()
```


Improving autoencoder robustness

A successful strategy we can take to improve the model's robustness is to introduce a noise in the encoding phase. Indeed, we call a denoising autoencoder a stochastic version of an autoencoder, in which the input is stochastically corrupted, but the uncorrupted version of the same input is used as a target for the decoding phase.

Intuitively, a denoising autoencoder does two things: first, it tries to encode the input, preserving the concerning information, and then it seeks to nullify the effect of the corruption process applied to the same input.

In the following section, we'll show an implementation of a denoising autoencoder.

Building a denoising autoencoder

The network architecture is very simple. An input image, of size 784 pixels, is stochastically corrupted, and then it is dimensionally reduced by an encoding network layer. The reduction step is from 784 to 256 pixels.

In the decoding phase, we prepare the network for output, re-changing the original image size from 256 to 784 pixels.

As usual, we start loading all the necessary libraries to our implementation:

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
from tensorflow.examples.tutorials.mnist import input_data
```

Set the basic network parameters:

```
n_input      = 784
n_hidden_1   = 256
n_hidden_2   = 256
n_output     = 784
```

We also set the session's parameters:

```
epochs       = 110
batch_size   = 100
disp_step    = 10
```

We build the training and test sets. We again use the `input_data` feature imported from the `tensorflow.examples.tutorials.mnist` library present in the installation package:

```
print ("PACKAGES LOADED")
mnist = input_data.read_data_sets('data/', one_hot=True)
trainimg  = mnist.train.images
trainlabel = mnist.train.labels
testimg   = mnist.test.images
testlabel  = mnist.test.labels
print ("MNIST LOADED")
```

Let's define a `placeholder` variable for the input images. The data type is set to `float` and the shape is set to `[None, n_input]`. The `None` parameter means that the tensor may hold an arbitrary number of images, and the size per image is `n_input`:

```
| x = tf.placeholder("float", [None, n_input])
```

Next, we have a `placeholder` variable for the true labels associated with the images that were input in the `placeholder` variable `x`. The shape of this `placeholder` variable is `[None, n_output]`, which means it may hold an arbitrary number of labels, and each label is a vector of length `n_output`, which is 10 in this case:

```
| y = tf.placeholder("float", [None, n_output])
```

To reduce overfitting, we'll apply a dropout before the encoding and decoding procedure, so we must define a `placeholder` for the probability that a neuron's output is kept during the dropout:

```
| dropout_keep_prob = tf.placeholder("float")
```

On these definitions, we fix the weights and network's biases:

```
weights = {
    'h1': tf.Variable(tf.random_normal([n_input, n_hidden_1])),
    'h2': tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2])),
    'out': tf.Variable(tf.random_normal([n_hidden_2, n_output]))
}
biases = {
    'b1': tf.Variable(tf.random_normal([n_hidden_1])),
    'b2': tf.Variable(tf.random_normal([n_hidden_2])),
    'out': tf.Variable(tf.random_normal([n_output]))
}
```

Weights and biases are chosen using `tf.random_normal`, which returns random values with a normal distribution.

The encoding phase takes as its input an image from the MNIST dataset, and then performs the data compression, applying a matrix multiplication operation:

```
encode_in = tf.nn.sigmoid\
    (tf.add(tf.matmul\
        (x, weights['h1']),\
        biases['b1']))
encode_out = tf.nn.dropout\
    (encode_in, dropout_keep_prob)
```

In the decoding phase, we apply the same procedure:

```
decode_in = tf.nn.sigmoid\
    (tf.add(tf.matmul\
        (encode_out, weights['h2']),\
        biases['b2']))
```

The over-fitting reduction is made by a dropout procedure:

```
decode_out = tf.nn.dropout(decode_in, \
    dropout_keep_prob)
```

Finally, we are ready to build the prediction tensor, `y_pred`:

```
y_pred = tf.nn.sigmoid\
    (tf.matmul(decode_out, \
        weights['out']) + \
        biases['out'])
```

We then define a `cost` measure that is used to guide the variables optimization procedure:

```
| cost = tf.reduce_mean(tf.pow(y_pred - y, 2))
```

We will minimize the `cost` function, using the `RMSPropOptimizer` class:

```
| optimizer = tf.train.RMSPropOptimizer(0.01).minimize(cost)
```

Finally, we can initialize the defined variables:

```
| init = tf.initialize_all_variables()
```

Set TensorFlow's running session:

```
| with tf.Session() as sess:  
|     sess.run(init)  
|     print ("Start Training")  
|     for epoch in range(epochs):  
|         num_batch = int(mnist.train.num_examples/batch_size)  
|         total_cost = 0.  
|         for i in range(num_batch):
```

For each training epoch, we select a smaller batch set from the training dataset:

```
|     batch_xs, batch_ys =\  
|         mnist.train.next_batch(batch_size)
```

Here is the focal point: we randomly corrupt the `batch_xs` set using the `random` function by the `numpy` package previously imported:

```
|     batch_xs_noisy = batch_xs +\  
|         0.3*np.random.randn(batch_size, 784)
```

We use these sets to feed the execution graph, and then to run the session (`sess.run`):

```
|     feeds = {x: batch_xs_noisy,\  
|               y: batch_xs,\  
|               dropout_keep_prob: 0.8}  
|     sess.run(optmizer, feed_dict=feeds)  
|     total_cost += sess.run(cost, feed_dict=feeds)
```

Every 10 epochs, the `average cost` value will be shown:

```
|     if epoch % disp_step == 0:  
|         print ("Epoch %02d/%02d average cost: %.6f"  
|             % (epoch, epochs, total_cost/num_batch))
```

Finally, we start to test the trained model:

```
|     print ("Start Test")
```

To make this, we randomly select an image from the test set:

```
|     randidx = np.random.randint\  
|                     (testimg.shape[0], size=1)  
|     orgvec = testimg[randidx, :]  
|     testvec = testimg[randidx, :]  
|     label = np.argmax(testlabel[randidx, :], 1)  
|     print ("Test label is %d" % (label))  
|     noisyvec = testvec + 0.3*np.random.randn(1, 784)
```

We then run the trained model on the selected image:

```
|     outvec = sess.run(y_pred, feed_dict={x: noisyvec,\  
|                                         dropout_keep_prob: 1})
```

As we'll see, the following `plotresult` function will display the original image, the noisy image, and the

predicted image:

```
| plotresult(orgvec,noisyvec,outvec)
| print ("restart Training")
```

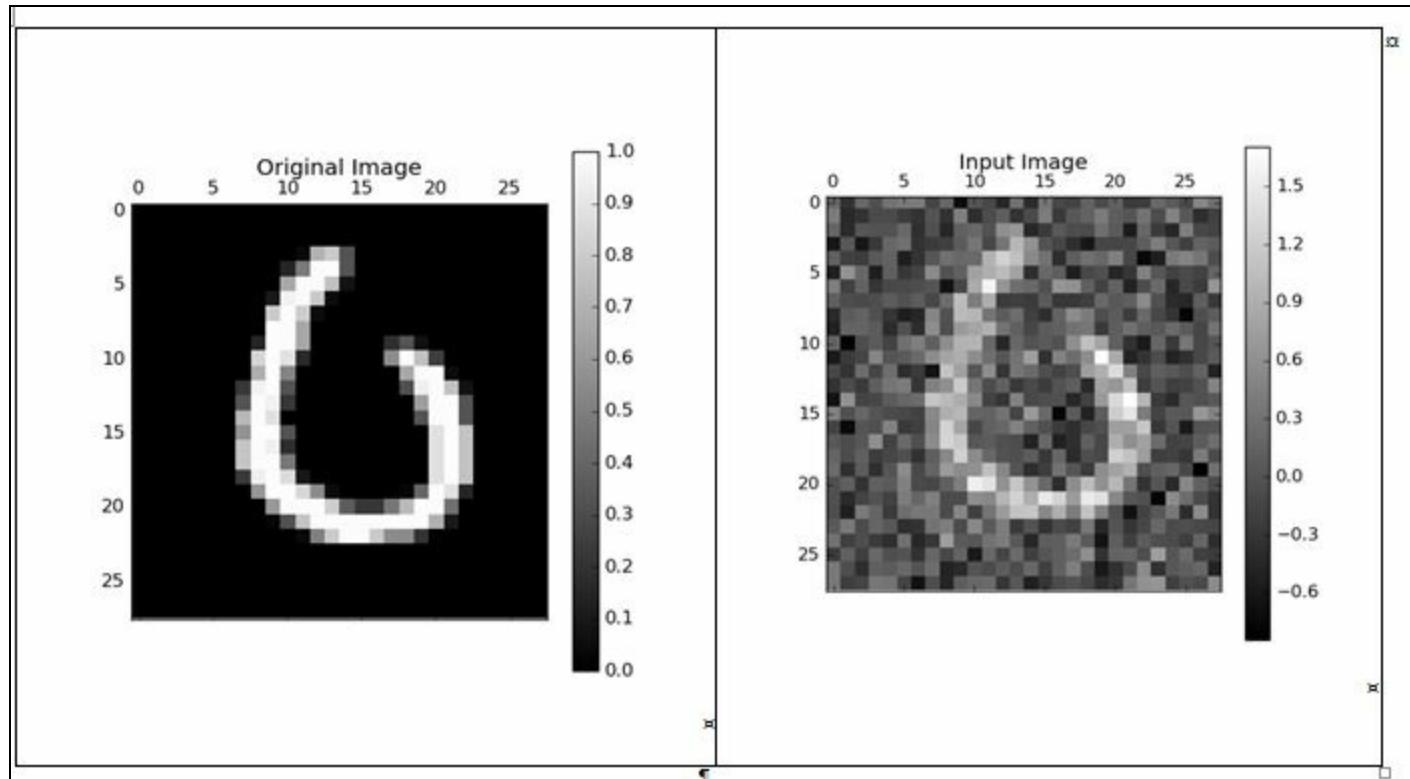
Running the session, we should see a result like the following:

```
| PACKAGES LOADED
| Extracting data/train-images-idx3-ubyte.gz
| Extracting data/train-labels-idx1-ubyte.gz
| Extracting data/t10k-images-idx3-ubyte.gz
| Extracting data/t10k-labels-idx1-ubyte.gz
| MNIST LOADED
| Start Training
```

For the sake of brevity, we only report the results after 10 epochs and after 100 epochs:

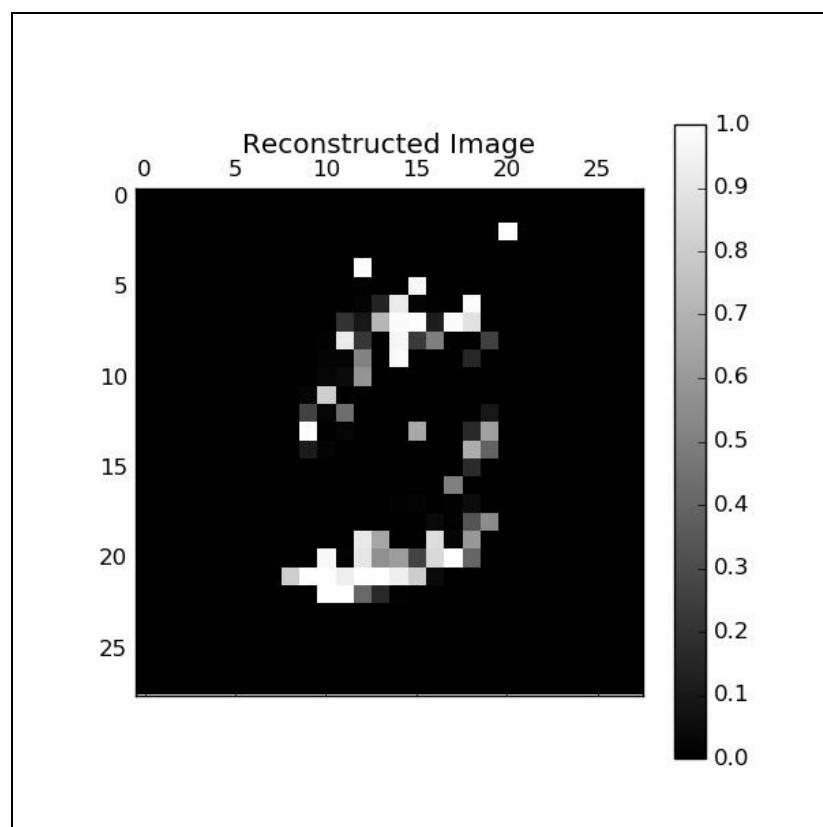
```
| Epoch 00/100 average cost: 0.212313
| Start Test
| Test label is 6
```

The following are the original and the noisy images (the number 6, as you can see):



Original and noisy images

The following shows a poorly reconstructed image:

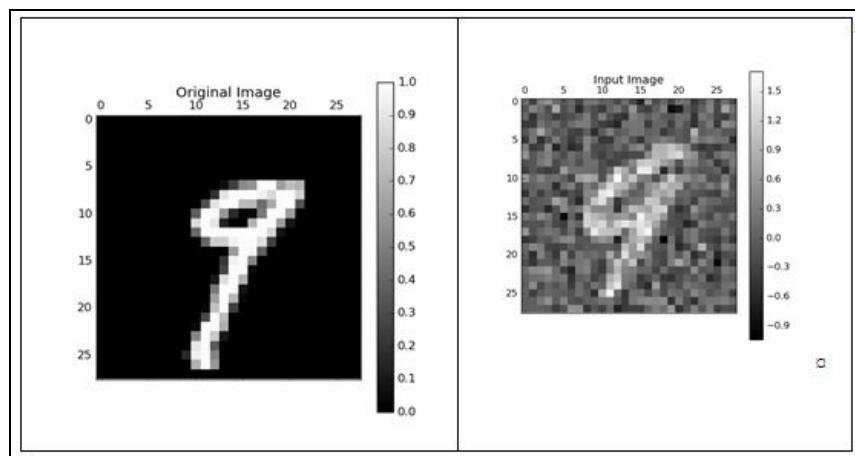


Reconstructed image

After 100 epochs, we have a better result:

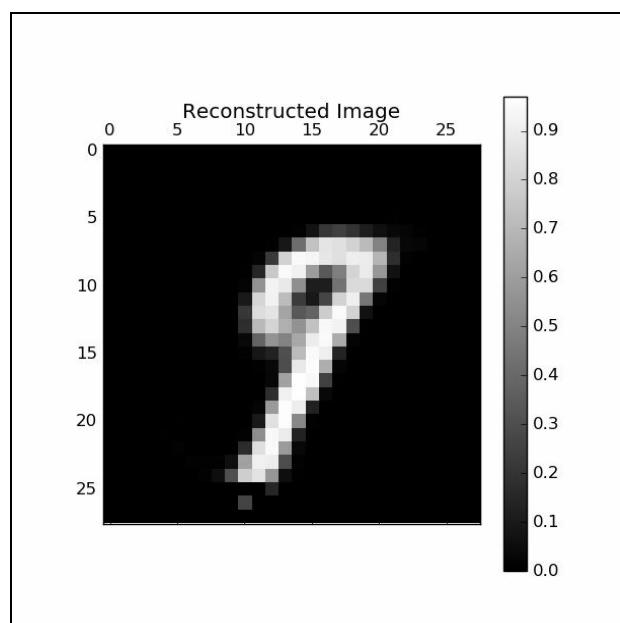
```
Epoch 100/100 average cost: 0.018221
Start Test
Test label is 9
```

Again, the original and the noisy images:



Original and noisy images

Followed by a good reconstructed image:



Reconstructed image

Source code for the denoising autoencoder

Here is the complete source code for the implemented autoencoder:

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
from tensorflow.examples.tutorials.mnist import input_data
#Plot function
def plotresult(org_vec,noisy_vec,out_vec):
    plt.matshow(np.reshape(org_vec, (28, 28)),\
                cmap=plt.get_cmap('gray'))
    plt.title("Original Image")
    plt.colorbar()
    plt.matshow(np.reshape(noisy_vec, (28, 28)),\
                cmap=plt.get_cmap('gray'))
    plt.title("Input Image")
    plt.colorbar()

    outimg = np.reshape(out_vec, (28, 28))\
    plt.matshow(outimg, cmap=plt.get_cmap('gray'))
    plt.title("Reconstructed Image")
    plt.colorbar()
    plt.show()

# NETOWORK PARAMETERS
n_input      = 784
n_hidden_1   = 256
n_hidden_2   = 256
n_output     = 784

epochs       = 110
batch_size   = 100
disp_step    = 10

print ("PACKAGES LOADED")

mnist = input_data.read_data_sets('data/', one_hot=True)
trainimg   = mnist.train.images
trainlabel = mnist.train.labels
testimg    = mnist.test.images
testlabel  = mnist.test.labels
print ("MNIST LOADED")

# PLACEHOLDERS
x = tf.placeholder("float", [None, n_input])
y = tf.placeholder("float", [None, n_output])
dropout_keep_prob = tf.placeholder("float")

# WEIGHTS
weights = {
    'h1': tf.Variable(tf.random_normal([n_input, n_hidden_1])),
    'h2': tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2])),
    'out': tf.Variable(tf.random_normal([n_hidden_2, n_output]))
}
biases = {
    'b1': tf.Variable(tf.random_normal([n_hidden_1])),
    'b2': tf.Variable(tf.random_normal([n_hidden_2])),
    'out': tf.Variable(tf.random_normal([n_output]))
}

encode_in = tf.nn.sigmoid\
    (tf.add(tf.matmul\
        (x, weights['h1']),\
        biases['b1']))

encode_out = tf.nn.dropout\
    (encode_in, dropout_keep_prob)

decode_in = tf.nn.sigmoid\
```

```

(tf.add(tf.matmul\
       (encode_out, weights['h2']), \
       biases['b2']))

decode_out = tf.nn.dropout(decode_in,\n                           dropout_keep_prob)

y_pred = tf.nn.sigmoid\
         (tf.matmul(decode_out,\n                    weights['out']) +\
         biases['out'])

# COST
cost = tf.reduce_mean(tf.pow(y_pred - y, 2))

# OPTIMIZER
optimizer = tf.train.RMSPropOptimizer(0.01).minimize(cost)

# INITIALIZER
init = tf.global_variables_initializer()

# Launch the graph
with tf.Session() as sess:
    sess.run(init)
    print ("Start Training")
    for epoch in range(epochs):
        num_batch = int(mnist.train.num_examples/batch_size)
        total_cost = 0.
        for i in range(num_batch):
            batch_xs, batch_ys = mnist.train.next_batch(batch_size)
            batch_xs_noisy = batch_xs\
                + 0.3*np.random.randn(batch_size, 784)
            feeds = {x: batch_xs_noisy,\n                    y: batch_xs,\n                    dropout_keep_prob: 0.8}
            sess.run(optimizer, feed_dict=feeds)
            total_cost += sess.run(cost, feed_dict=feeds)
    # DISPLAY
    if epoch % disp_step == 0:
        print ("Epoch %02d/%02d average cost: %.6f"
              % (epoch, epochs, total_cost/num_batch))

        # Test one
        print ("Start Test")
        randidx = np.random.randint\
                  (testimg.shape[0], size=1)
        orgvec = testimg[randidx, :]
        testvec = testimg[randidx, :]
        label = np.argmax(testlabel[randidx, :], 1)

        print ("Test label is %d" % (label))
        noisyvec = testvec + 0.3*np.random.randn(1, 784)
        outvec = sess.run(y_pred,\n                         feed_dict={x: noisyvec,\n                                     dropout_keep_prob: 1})

        plotresult(orgvec,noisyvec,outvec)
        print ("restart Training")

```


Convolutional autoencoders

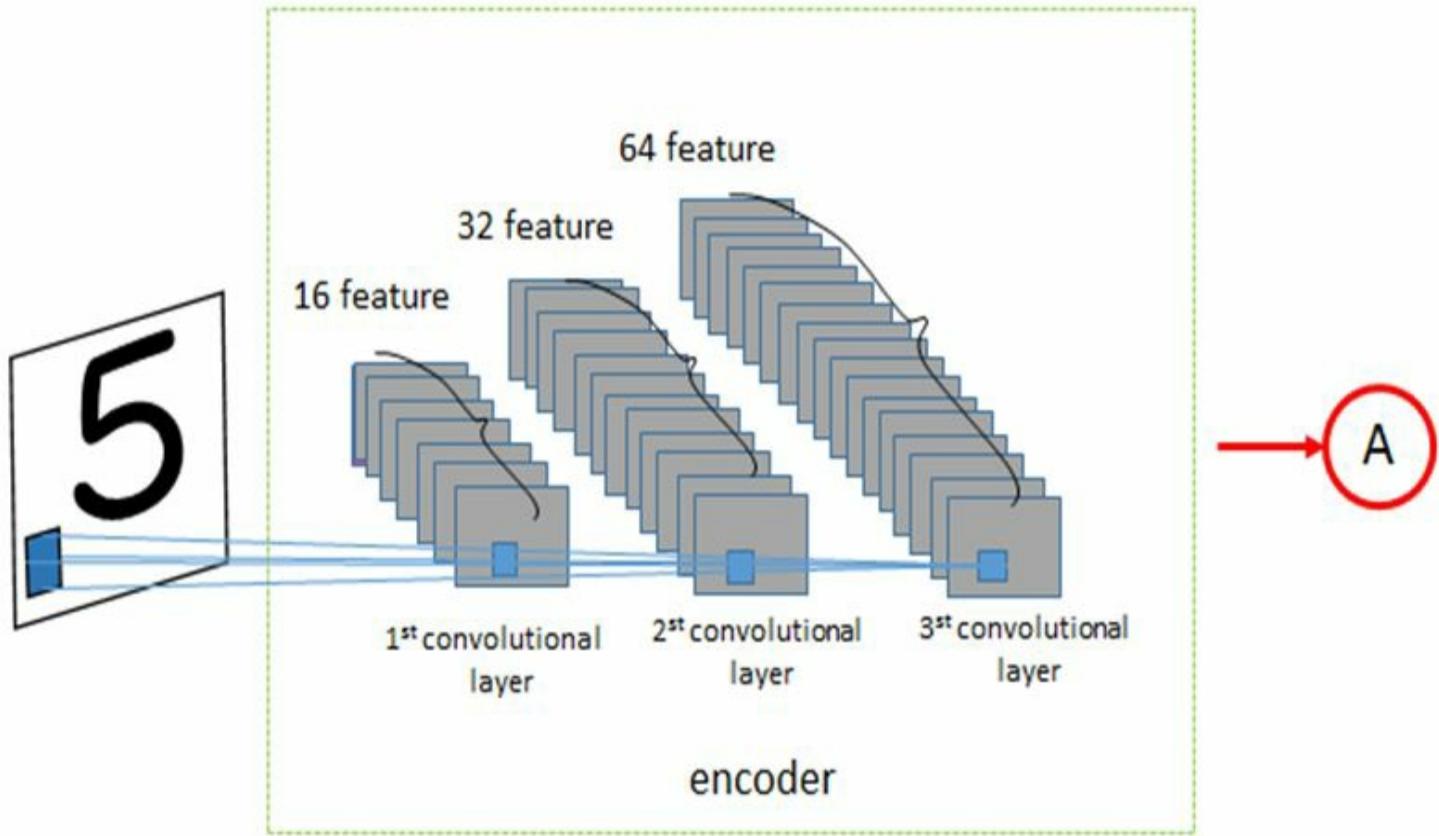
Until now, we have seen that autoencoder inputs are images. So, it makes sense to ask whether a convolutional architecture can work better than the autoencoder architectures discussed previously.

We will analyze how the encoder and decoder work in convolutional autoencoders.

Encoder

The encoder consists of three convolutional layers. The number of features changes from 1, the input data, to 16 for the first convolutional layer, then from 16 to 32 for the second layer, and finally, from 32 to 64 for the final convolutional layer.

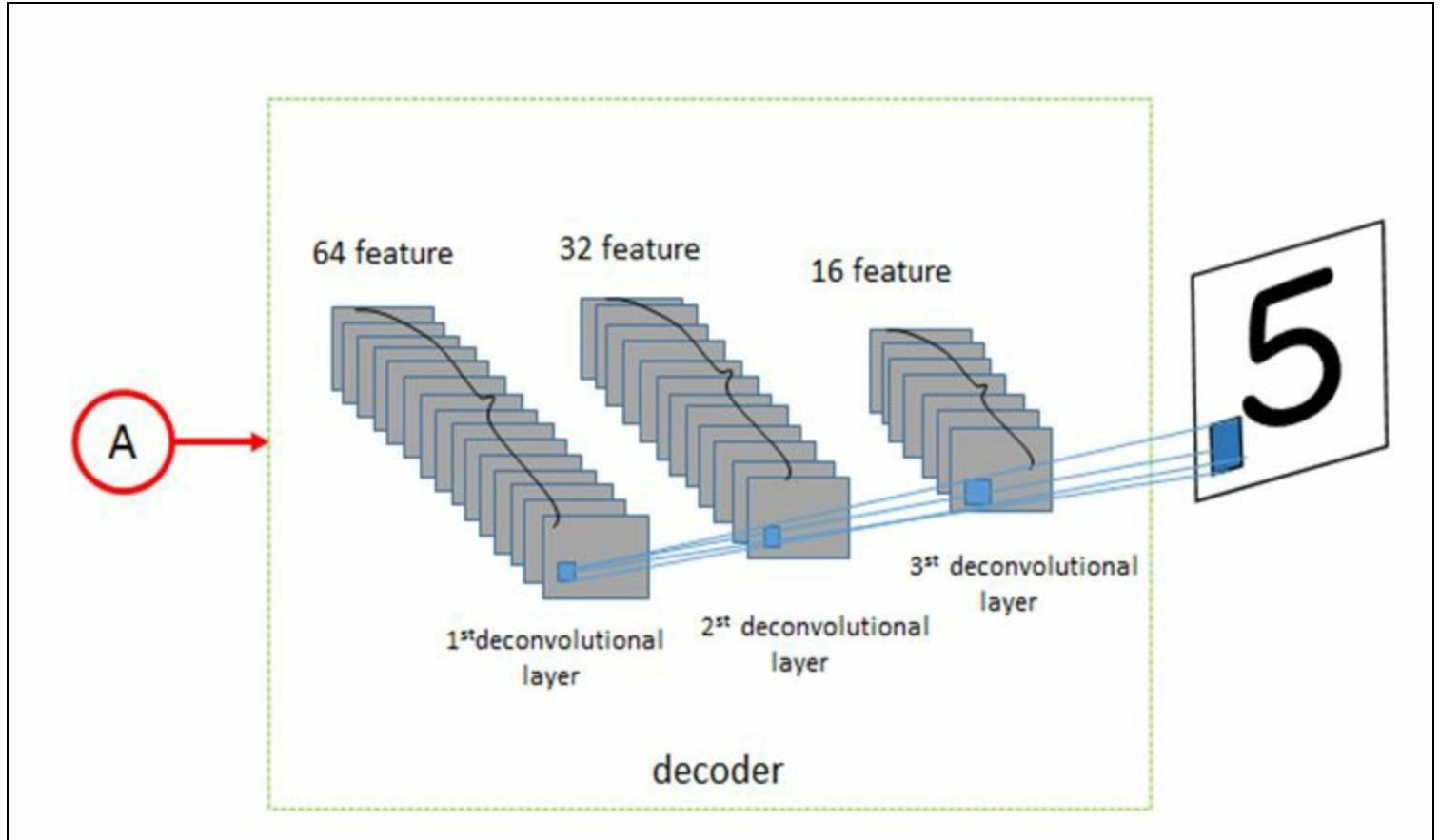
While transitioning from one convolutional layer to another, the shape undergoes an image compression:



Data flow of encoding phase

Decoder

The decoder consists of three deconvolution layers arranged in sequence. For each deconvolution operation, we reduce the number of features to obtain an image that must be the same size as the original image. In addition to reducing the number of features, deconvolution involves a shape transformation of the images:



Data flow of the decoding phase

We're ready to look at how to implement a convolutional autoencoder. The first step is to load the basic libraries:

```
import matplotlib.pyplot as plt
import numpy as np
import math
import tensorflow as tf
import tensorflow.examples.tutorials.mnist.input_data as input_data
```

Then build the training and test sets:

```
mnist = input_data.read_data_sets("data/", one_hot=True)
trainimgs = mnist.train.images
trainlabels = mnist.train.labels
testimgs = mnist.test.images
testlabels = mnist.test.labels
ntrain = trainimgs.shape[0]
ntest = testimgs.shape[0]
dim = trainimgs.shape[1]
nout = trainlabels.shape[1]
```

Define a `placeholder` variable for the input images:

```
| x = tf.placeholder(tf.float32, [None, dim])
```

The data type is set to `float32`, and the shape is set to `[None, dim]`, where `None` means that the tensor may hold an arbitrary number of images, with each image being a vector of length `dim`. Next, we have a `placeholder` variable for the output images. The shape of this variable is set to `[None, dim]`, the same input shape:

```
| y = tf.placeholder(tf.float32, [None, dim])
```

Then we define the `keepprob` variable, which is used to configure the dropout rate used during neural network training:

```
| keepprob = tf.placeholder(tf.float32)
```

Also, we have to define the number of nodes for each of the network's layers:

```
n1 = 16  
n2 = 32  
n3 = 64  
ksize = 5
```

The network contains a total of six layers. The first three layers are convolutional and belong to the encoding phase, while the last three layers are deconvolutional and define the decoding phase:

```
weights = {  
    'ce1': tf.Variable(tf.random_normal([ksize, ksize, 1, n1], stddev=0.1)),  
    'ce2': tf.Variable(tf.random_normal([ksize, ksize, n1, n2], stddev=0.1)),  
    'ce3': tf.Variable(tf.random_normal([ksize, ksize, n2, n3], stddev=0.1)),  
    'cd3': tf.Variable(tf.random_normal([ksize, ksize, n2, n3], stddev=0.1)),  
    'cd2': tf.Variable(tf.random_normal([ksize, ksize, n1, n2], stddev=0.1)),  
    'cd1': tf.Variable(tf.random_normal([ksize, ksize, 1, n1], stddev=0.1))  
}  
  
biases = {  
    'be1': tf.Variable(tf.random_normal([n1], stddev=0.1)),  
    'be2': tf.Variable(tf.random_normal([n2], stddev=0.1)),  
    'be3': tf.Variable(tf.random_normal([n3], stddev=0.1)),  
    'bd3': tf.Variable(tf.random_normal([n2], stddev=0.1)),  
    'bd2': tf.Variable(tf.random_normal([n1], stddev=0.1)),  
    'bd1': tf.Variable(tf.random_normal([1], stddev=0.1))  
}
```

The following function, `cae`, builds the convolutional autoencoder: the inputs passed are the image, `_x`, the data structure weights and bias, `_w`, `_b`, and the `_keepprob` parameter:

```
| def cae(_x, _w, _b, _keepprob):
```

The initial image of 784 pixels must be reshaped in a 28×28 matrix to be subsequently processed by the next convolutional layers:

```
| _input_r = tf.reshape(_X, shape=[-1, 28, 28, 1])
```

The first convolutional layer is `_ce1`, which has as its input the `_input_r` tensor relative to the input image:

```
| _ce1 = tf.nn.sigmoid\  
|   (tf.add(tf.nn.conv2d\  
|     (_input_r, _W['ce1'], \  
|       strides=[1, 2, 2, 1], \  
|       padding='SAME'), \  
|       _b['be1']))
```

Before moving to the second convolutional layer, we apply the dropout operation:

```
| _ce1 = tf.nn.dropout(_ce1, _keepprob)
```

In the following two encoding layers, we apply the same convolution and dropout operations:

```
| _ce2 = tf.nn.sigmoid\  
|   (tf.add(tf.nn.conv2d\  
|     (_ce1, _W['ce2'], \  
|       strides=[1, 2, 2, 1], \  
|       padding='SAME'), \  
|       _b['be2']))  
| _ce2 = tf.nn.dropout(_ce2, _keepprob)  
  
| _ce3 = tf.nn.sigmoid\  
|   (tf.add(tf.nn.conv2d\  
|     (_ce2, _W['ce3'], \  
|       strides=[1, 2, 2, 1], \  
|       padding='SAME'), \  
|       _b['be3']))  
| _ce3 = tf.nn.dropout(_ce3, _keepprob)
```

The number of features has increased from 1 (the input image) to 64, while the original shape image is reduced from 28×28 to 7×7 . In the decoding phase, the compressed (or encoded) and reshaped image must be as similar to the original as possible.

To obtain this, we used the TensorFlow function, `conv2d_transpose`, for the next (three) layers:

```
| tf.nn.conv2d_transpose(value, filter, output_shape, strides, padding='SAME')
```

This operation is sometimes called deconvolution; it is simply the transpose (gradient) of `conv2d`:

The arguments of this function are as follows:

- `value`: A 4-D tensor of type float and shape (batch, height, width, and `in_channels`).
- `filter`: A 4-D tensor with the same type as value and shape (height, width, `output_channels`, `in_channels`). The `in_channels` dimension must match that of value.
- `output_shape`: A 1-D tensor representing the output shape of the deconvolution op.
- `strides`: A list of ints. The stride of the sliding window for each dimension of the input tensor.
- `padding`: A string, either valid or same.

- `conv2d_transpose`: This will return a tensor with the same type as the `value` argument.

The first deconvolutional layer, `_cd3`, has the convolutional layer `_ce3` as the input. It returns the `_cd3` tensor, whose shape is (1, 7, 7, 32):

```
_cd3 = tf.nn.sigmoid\
    (tf.add(tf.nn.conv2d_transpose\
        (_ce3, _W['cd3'], \
            tf.pack([tf.shape(_x)[0], 7, 7, n2]), \
            strides=[1, 2, 2, 1], \
            padding='SAME'), \
            b['bd3']))
_cd3 = tf.nn.dropout(_cd3, _keepprob)
```

To the second deconvolutional layer, `_cd2`, we pass as the input the deconvolutional layer, `_cd3`. It returns the `_cd2` tensor, whose shape is (1, 14, 14, 16):

```
_cd2 = tf.nn.sigmoid\
    (tf.add(tf.nn.conv2d_transpose\
        (_cd3, _W['cd2'], \
            tf.pack([tf.shape(_x)[0], 14, 14, n1]), \
            strides=[1, 2, 2, 1], \
            padding='SAME'), \
            b['bd2']))
_cd2 = tf.nn.dropout(_cd2, _keepprob)
```

The third and final deconvolutional layer, `_cd1`, has the layer `_cd2` passed as the input. It returns the resulting `_out` tensor, whose shape is (1, 28, 28, 1), equal to the input image:

```
_cd1 = tf.nn.sigmoid\
    (tf.add(tf.nn.conv2d_transpose\
        (_cd2, _W['cd1'], \
            tf.pack([tf.shape(_x)[0], 28, 28, 1]), \
            strides=[1, 2, 2, 1], \
            padding='SAME'), \
            b['bd1']))
_cd1 = tf.nn.dropout(_cd1, _keepprob)
_out = _cd1
return _out
```

Then we define a `cost` function as the mean squared error between `y` and `pred`:

```
pred = cae(x, weights, biases, keepprob)
cost = tf.reduce_sum\
    (tf.square(cae(x, weights, biases, keepprob) \
        - tf.reshape(y, shape=[-1, 28, 28, 1])))
learning_rate = 0.001
```

To optimize the `cost`, we'll use the `AdamOptimizer`:

```
| optm = tf.train.AdamOptimizer(learning_rate).minimize(cost)
```

In the next step, we configure the running session for our network:

```
init = tf.global_variables_initializer()
print ("Functions ready")
sess = tf.Session()
sess.run(init)
mean_img = np.zeros((784))
```

The size of the batch is set as 128:

```
| batch_size = 128
```

The number of epochs is 5:

```
| n_epochs = 5
```

Start the loop session:

```
| for epoch_i in range(n_epochs):
```

For each epoch, we get a batch set, trainbatch:

```
| for batch_i in range(mnist.train.num_examples // batch_size):
|     batch_xs, _ = mnist.train.next_batch(batch_size)
|     trainbatch = np.array([img - mean_img for img in batch_xs])
```

We apply a random noise, just like denoising autoencoders, to ensure better learning:

```
| trainbatch_noisy = trainbatch + 0.3*np.random.randn(
|     trainbatch.shape[0], 784)
| sess.run(optm, feed_dict={x: trainbatch_noisy\
|                           , y: trainbatch, keepprob: 0.7})
| print ("[%02d/%02d] cost: %.4f" % (epoch_i, n_epochs
| , sess.run(cost, feed_dict={x: trainbatch_noisy
|                           , y: trainbatch, keepprob: 1.})))
```

For each training epoch, we randomly take 5 training examples:

```
| if (epoch_i % 1) == 0:
|     n_examples = 5
|     test_xs, _ = mnist.test.next_batch(n_examples)
|     test_xs_noisy = test_xs + 0.3*np.random.randn(
|         test_xs.shape[0], 784)
```

Then we test the trained model on a little subset:

```
| recon = sess.run(pred, feed_dict={x: test_xs_noisy,
|                                     keepprob: 1.})
| fig, axs = plt.subplots(2, n_examples, figsize=(15, 4))
| for example_i in range(n_examples):
|     axs[0][example_i].matshow(np.reshape(
|         test_xs_noisy[example_i, :], (28, 28))
|     , cmap=plt.get_cmap('gray'))
```

Finally, we can display the inputs and the learned set, by matplotlib:

```
| axs[1][example_i].matshow(np.reshape(
|     np.reshape(recon[example_i, ...], (784,)) +
|     mean_img, (28, 28)), cmap=plt.get_cmap('gray'))
| plt.show()
```

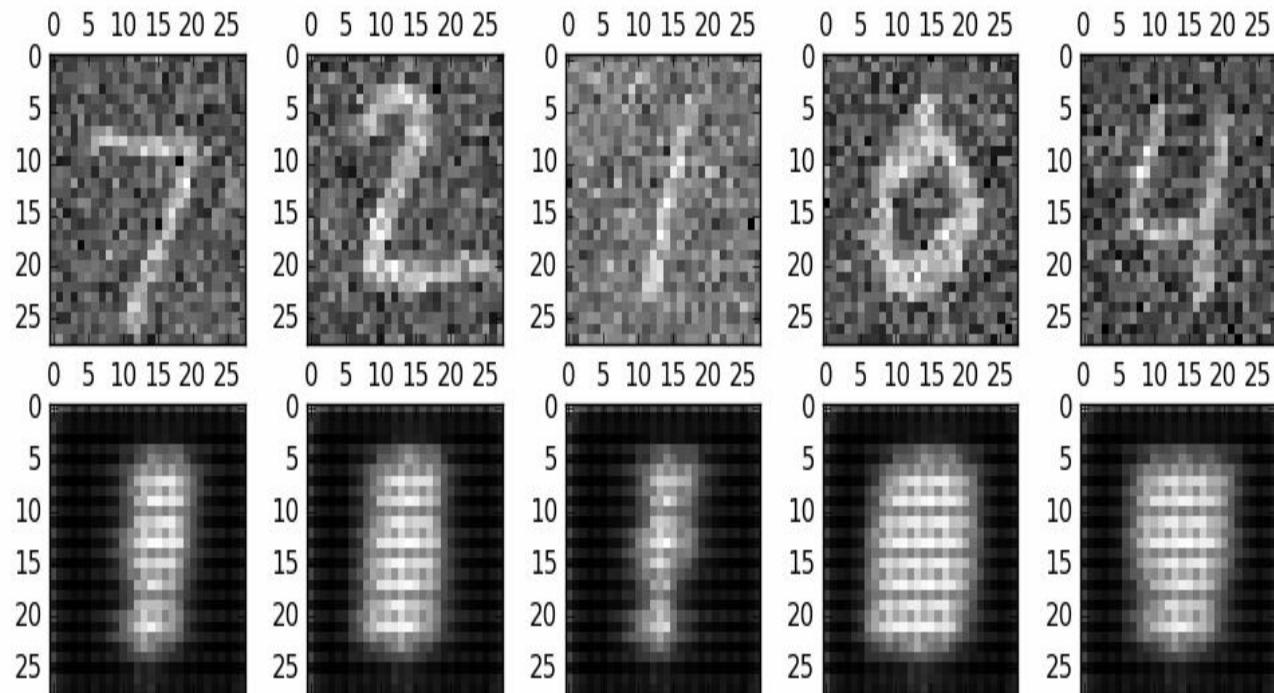
The execution will produce the following output:

```
>>>
Extracting data/train-images-idx3-ubyte.gz
Extracting data/train-labels-idx1-ubyte.gz
Extracting data/t10k-images-idx3-ubyte.gz
Extracting data/t10k-labels-idx1-ubyte.gz
Packages loaded
Network ready
Functions ready
```

```
Start training..  
[00/05] cost: 8049.0332  
[01/05] cost: 3706.8667  
[02/05] cost: 2839.9155  
[03/05] cost: 2462.7021  
[04/05] cost: 2391.9460  
>>>
```

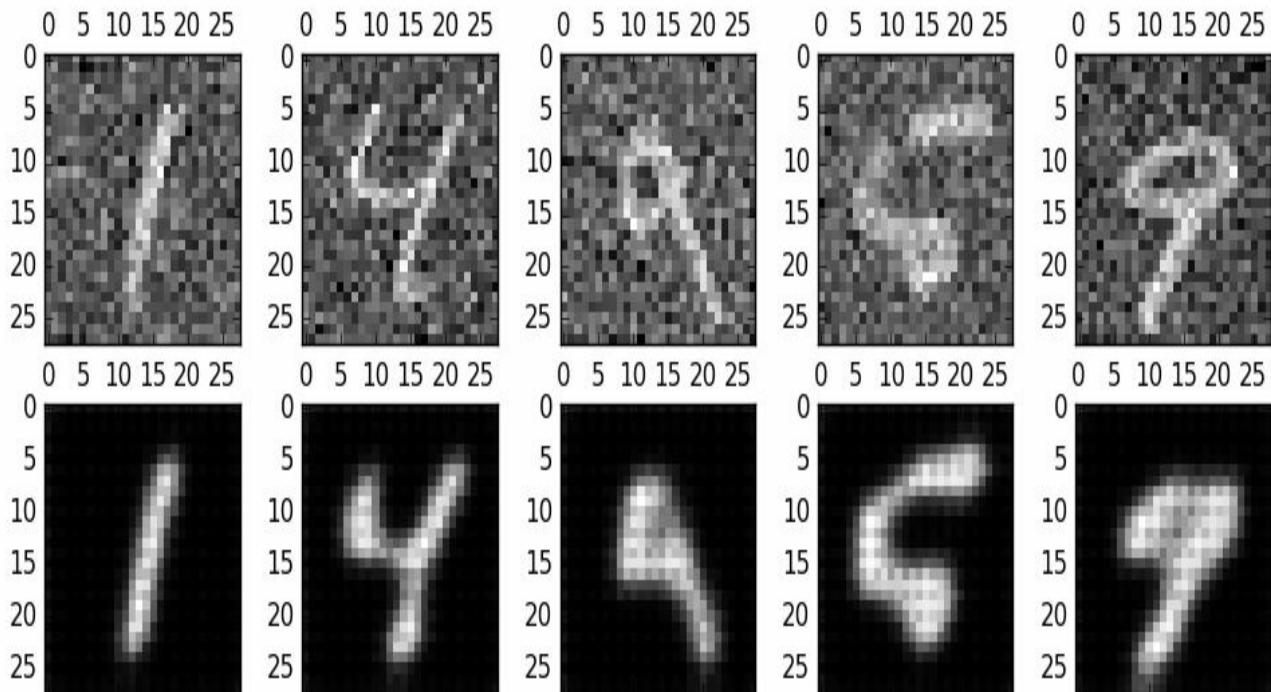
Note that, for each epoch, we'll visualize the input set and the corresponding learned set shown previously.

As you may note, for the first epoch, we have no idea what images are learned:



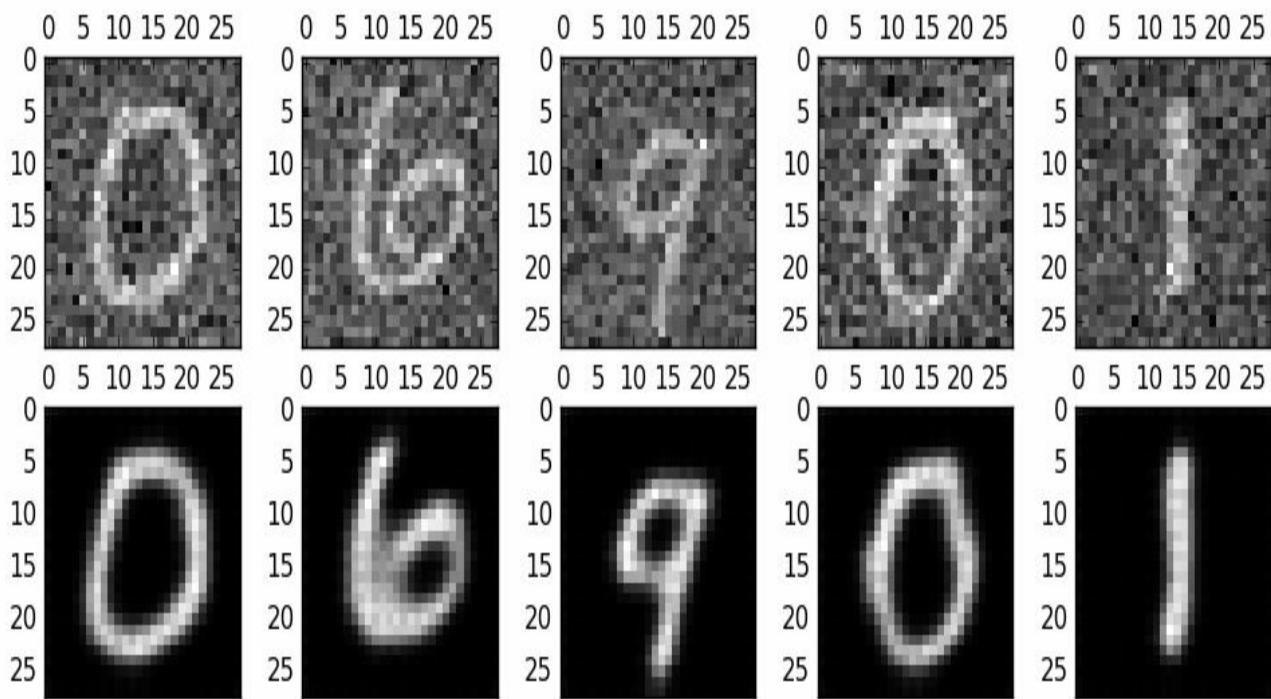
First epoch images

The idea becomes clearer in the second epoch:



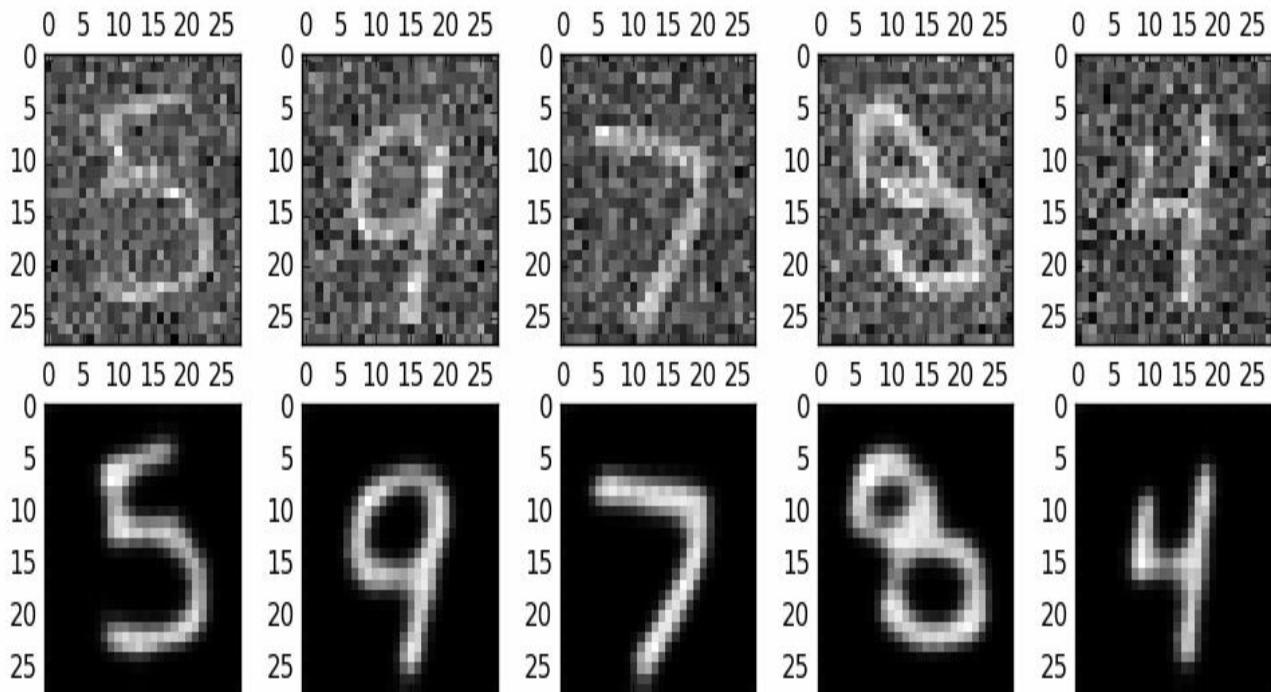
Second epoch images

The third epoch is shown in the following figure:



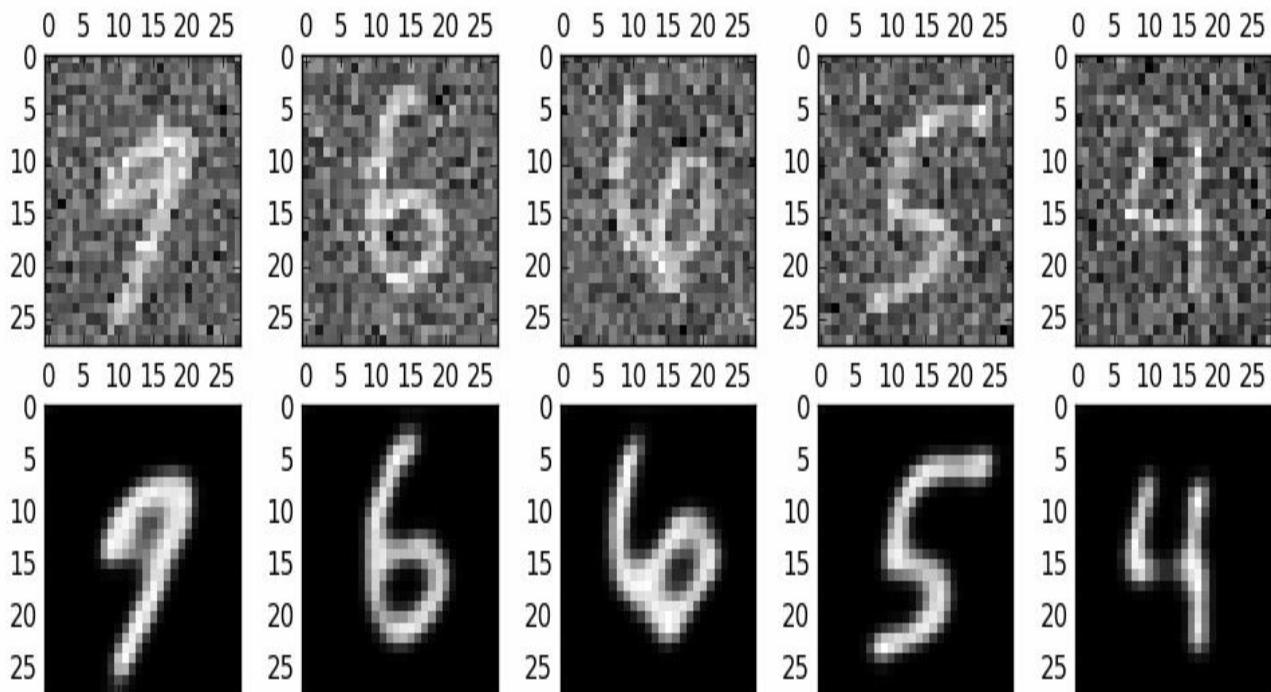
Third epoch images

They are still better in the fourth epoch:



Fourth epoch images

We could probably have stopped with the previous epoch, but this is the final, fifth epoch:



Fifth epoch images

Source code for convolutional autoencoder

Here is the source code for the example explained previously:

```
import matplotlib.pyplot as plt
import numpy as np
import math
import tensorflow as tf
import tensorflow.examples.tutorials.mnist.input_data as input_data

#LOAD PACKAGES
mnist = input_data.read_data_sets("data/", one_hot=True)
trainimgs = mnist.train.images
trainlabels = mnist.train.labels
testimgs = mnist.test.images
testlabels = mnist.test.labels
ntrain = trainimgs.shape[0]
ntest = testimgs.shape[0]
dim = trainimgs.shape[1]
nout = trainlabels.shape[1]
print ("Packages loaded")

#WEIGHT AND BIASES
n1 = 16
n2 = 32
n3 = 64
ksize = 5
weights = {
    'ce1': tf.Variable(tf.random_normal([
        [ksize, ksize, 1, n1], stddev=0.1))),
    'ce2': tf.Variable(tf.random_normal([
        [ksize, ksize, n1, n2], stddev=0.1))),
    'ce3': tf.Variable(tf.random_normal([
        [ksize, ksize, n2, n3], stddev=0.1])),
    'cd3': tf.Variable(tf.random_normal([
        [ksize, ksize, n2, n3], stddev=0.1])),
    'cd2': tf.Variable(tf.random_normal([
        [ksize, ksize, n1, n2], stddev=0.1])),
    'cd1': tf.Variable(tf.random_normal([
        [ksize, ksize, 1, n1], stddev=0.1)))
}
biases = {
    'be1': tf.Variable(
        (tf.random_normal([n1], stddev=0.1))),
    'be2': tf.Variable(
        (tf.random_normal([n2], stddev=0.1))),
    'be3': tf.Variable(
        (tf.random_normal([n3], stddev=0.1))),
    'bd3': tf.Variable(
        (tf.random_normal([n2], stddev=0.1))),
    'bd2': tf.Variable(
        (tf.random_normal([n1], stddev=0.1))),
    'bd1': tf.Variable(
        (tf.random_normal([1], stddev=0.1)))
}
def cae(_X, _W, _b, _keepprob):
    _input_r = tf.reshape(_X, shape=[-1, 28, 28, 1])
    # Encoder
    _ce1 = tf.nn.sigmoid(
        (tf.add(tf.nn.conv2d(
            (_input_r, _W['ce1']),
            strides=[1, 2, 2, 1],
            padding='SAME'),
            _b['be1'])))
    _ce1 = tf.nn.dropout(_ce1, _keepprob)
    _ce2 = tf.nn.sigmoid(
        (tf.add(tf.nn.conv2d(
            (_ce1, _W['ce2']),
```



```
print ("[%02d/%02d] cost: %.4f" % (epoch_i, n_epochs
    , sess.run(cost, feed_dict={x: trainbatch_noisy
                                , y: trainbatch, keepprob: 1.})))
if (epoch_i % 1) == 0:
    n_examples = 5
    test_xs, _ = mnist.test.next_batch(n_examples)
    test_xs_noisy = test_xs + 0.3*np.random.randn(
        test_xs.shape[0], 784)
    recon = sess.run(pred, feed_dict={x: test_xs_noisy,
                                      keepprob: 1.})
fig, axs = plt.subplots(2, n_examples, figsize=(15, 4))
for example_i in range(n_examples):
    axs[0][example_i].matshow(np.reshape(
        test_xs_noisy[example_i, :], (28, 28))
        , cmap=plt.get_cmap('gray'))
    axs[1][example_i].matshow(np.reshape(
        np.reshape(recon[example_i, ...], (784,)) +
        mean_img, (28, 28)), cmap=plt.get_cmap('gray'))
plt.show()
```


Summary

In this chapter, we have implemented some optimizing networks, called autoencoders. An autoencoder is basically a data-compression network model.

It is used to encode a given input into a representation of a smaller dimension; then, a decoder can be used to reconstruct the input back from the encoded version. All the autoencoders we implemented contain an encoding, and a decoding, part.

We have also looked at how to improve the autoencoder's performance, introducing a noise during network training, and building a denoising autoencoder. Finally, we applied the concepts of the CNN networks introduced in [Chapter 4, TensorFlow on a Convolutional Neural Network](#), with the implementation of convolutional autoencoders.

In the next chapter, we'll examine **Recurrent Neural Networks (RNNs)**. We will start by describing the basic principles of these networks, and then we'll implement some interesting example applications of these architectures.

Recurrent Neural Networks

Deep learning architectures that are used widely nowadays are the so-called **Recurrent Neural Networks (RNNs)**. The basic idea of RNNs is to make use of sequential type information in the input.

These networks are recurrent because they perform the same computations for all the elements of a sequence of inputs, and the output of each element depends, in addition to the current input, from all the previous computations.

RNNs have proved to have excellent performance in problems such as predicting the next character in a text or, similarly, the prediction of the next word sequence in a sentence.

However, they are also used for more complex problems, such as **Machine Translation (MT)**. In this case, the network has as input a sequence of words in a source language, while the output will be the translated input sequence in a target language, finally, other applications of great importance in which the RNNs are widely used are speech recognition and also image recognition.

The main topics of this chapter are organized in the following way:

- RNN basic concepts
- RNNs at work
- Unfolding an RNN
- The vanishing gradient problem
- LSTM networks
- An image classifier with RNNs
- Bidirectional RNNs
- Text processing

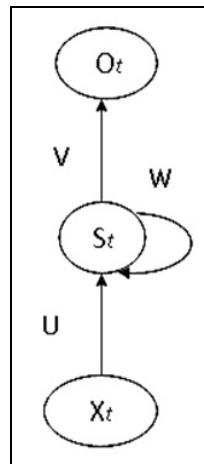
RNNs basic concepts

Human beings don't start thinking from scratch, human minds have the so-called persistence of memory, namely, the ability to associate the past with recent information. Traditional neural networks, instead, ignore past events. Taking as an example, a movie's scenes classifier, it's not possible that a neural network uses past scenes to classify the current ones.

Trying to solve this problem, RNNs have been developed, in contrast with the **Convolutional Neural Networks (CNNs)**, the RNNs are networks with a loop that allows the information to be persistent.

RNNs process a sequential input one at a time, updating a kind of vector state that contains information about all past elements of the sequence.

The following figure shows a neural network that takes as input a value of X_t , and then outputs an O_t value:



An RNN with its internal loop

S_t is a network's vector state that can be considered a kind of memory of the system, which contains information on all the previous elements of the input sequence. On the other hand, the depicted cycle allows information to move from each step of the network to the next.

RNNs at work

The state vector S_t is calculated starting from the current input and the state vector in previous time, through the U and W matrices:

$$S_t = f(U \cdot x_t + W \cdot S_{t-1})$$

f is a nonlinear function such as tanh or ReLU. As you can see, the two terms in the function are added together before being processed by the function itself.

Finally, O_t ; is the network output, calculated using the matrix V :

$$O_t = V \cdot S_t$$

Unfolding an RNN

The next figure shows an unfolded version of an RNN, obtained by unrolling the network structure for the entire input sequence, at different and discrete times. It is immediately clear that it is different from the typical multi-level neural networks, which use different parameters at each level; an RNN uses the same parameters, U , V , W , for each instant of time.

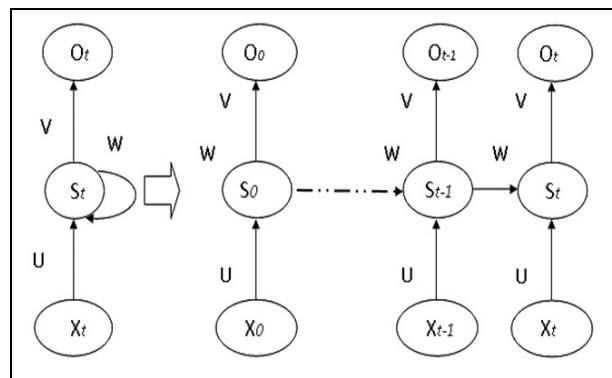
Indeed, RNNs perform the same computation at each instance, on different inputs of the same sequence. Sharing the same parameters, also, an RNN strongly reduces the number of parameters that the network must learn during the training phase, thus also improving the training times.

Regarding this unfolded version, it is evident how through the backpropagation algorithm with only a small change, you can train networks of this type.

In fact, because the parameters are shared for each instant time, the computed gradient depends on the current computation, but also on the previous ones.

For example, to calculate the gradient (at a time $t = 4$) it is necessary to back propagate the computed gradient for three instants of time preceding and then sum the gradients thus obtained. In fact, an entire input sequence is typically considered to be a single element of the training set. Therefore, if the total error is simply the sum of the errors at each instant of time (for each element of the input sequence) as a result, the error gradient turns out to be the sum of the error gradients at each instant of time.

This procedure is called the **Backpropagation Through Time (BPTT)**.



Unrolled version of an RNN

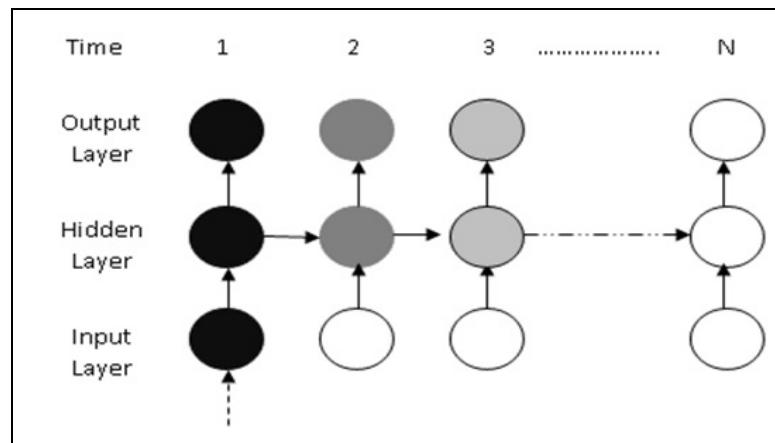
The vanishing gradient problem

In backpropagation algorithm, the weights are adjusted in proportion to the gradient error, and for the way in which the gradients are computed. Let's check the following:

- If the weights are small, it can lead to a situation called vanishing gradients where the gradient signal gets so small that learning either becomes very slow or stops working altogether. This is often referred to as **vanishing gradients**.
- If the weights in this matrix are large it can lead to a situation where the gradient signal is so large that it can cause learning to diverge. This is often referred to as **exploding gradients**.

The vanishing-exploding gradient problem also afflicts RNNs. In fact, the BPTT rolls out the RNN creating a very deep feed-forward neural network. The impossibility of having a long-term context by the RNN is due precisely to this phenomenon, if the gradient vanishes or explodes within a few layers, the network will not be able to learn high temporal distance relationships between the data.

The following figure shows schematically what happens; the computed and back propagated gradient tend to decrease (or increase) at each instant of time and then, after a certain number of instants of time, tend to converge to zero (or explode to infinity):



Vanishing gradient problem in RNNs

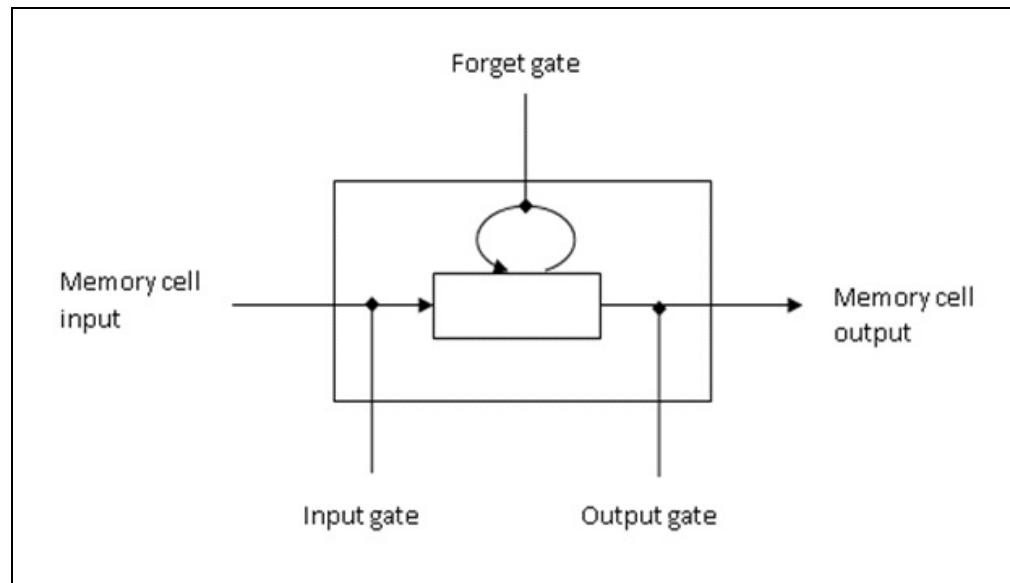
To overcome the vanishing-exploding problem, various extensions of the basic RNN model have been proposed, one of these, is represented, for example, by Long Short Term Memory (LSTM) networks that will be introduced in the next section.

LSTM networks

Long Short Term Memory (LSTM) is a special Recurrent Neural Network architecture, which was originally conceived by Hochreiter and Schmidhuber in 1997. This type of neural network has been recently rediscovered in the context of deep learning, because *it* is free from the problem of vanishing gradients, and offers excellent results and performance. The networks that are LSTM-based are ideal for prediction and classification of temporal sequences, and are replacing many traditional approaches to deep learning.

LSTM is a network that is composed of cells (LSTM blocks) linked to each other. Each LSTM block contains three types of gate: **Input gate**, **Output gate**, and **Forget gate**, respectively, which implement the functions of writing, reading, and resetting on the cell memory. These gates are not binary, but analogical (generally managed by a sigmoidal activation function mapped in the range $[0, 1]$, where 0 indicates total inhibition, and 1 shows the total activation).

The presence of these gates, allows LSTM cells to remember information for an indefinite time; in fact, if the following **Input gate** is the activation threshold, the cell will retain the previous state, and if the current state is enabled, it will be combined with the input value. As the name suggests, the **Forget gate** resets the current state of the cell (when its value is cleared to 0), and the **Output gate** decides whether the value in the cell must be carried out or not.

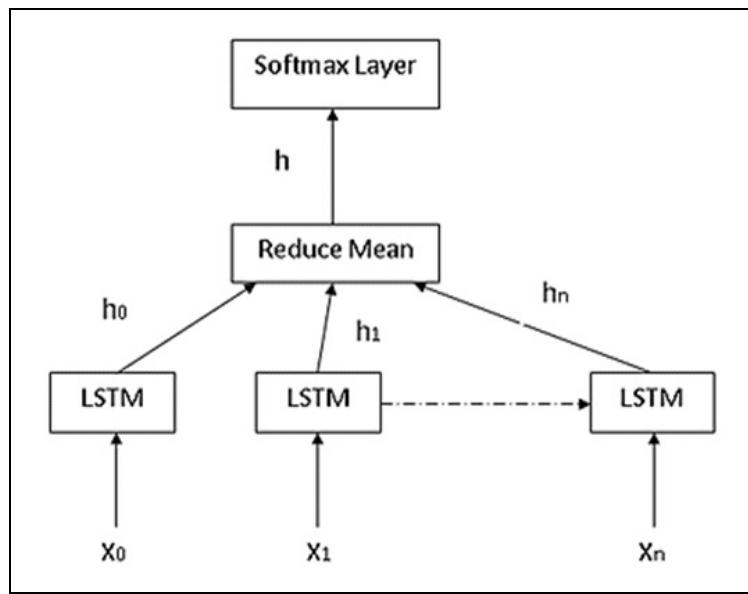


Block diagram of an LSTM cell

An image classifier with RNNs

At this point we introduce our implementation of a recurrent model including LSTMs blocks for an image classification problem. The dataset we used is the well known MNIST.

The implemented model is composed of a single LSTM layer followed by a reduce mean operation and a softmax layer, as illustrated in the following figure:



Dataflow in an RNN architecture

The following code computes the mean of elements across dimensions of a tensor and reduces `input_tensor` along the dimensions given in `axis`. Unless `keep_dims` is `true`, the rank of the tensor is reduced by `1` for each entry in `axis`. If `keep_dims` is `true`, the reduced dimensions are retained with length `1`:

`tf.reduce_mean(input_tensor, axis=None, keep_dims=False, name=None, reduction_indices=None)`

If `axis` has no entries, all dimensions are reduced, and a tensor with a single element is returned.

For example:

```
# 'x' is [[1., 1.]
#           [2., 2.]]
tf.reduce_mean(x)==> 1.5
tf.reduce_mean(x,0)==> [1.5,1.5]
tf.reduce_mean(x,1)==> [1.,2.]
```

Thus, starting from an input sequence x_0, x_1, \dots, x_n , the memory cells in the LSTM layer will produce a representation sequence h_0, h_1, \dots, h_n .

This representation sequence is then averaged over all time steps resulting in the final representation h . Finally, this representation is fed to a softmax layer whose target is the class label associated with the input sequence.

Let's begin the implementation, with the usual importing of all dependencies:

```
| import tensorflow as tf  
| from tensorflow.contrib import rnn
```

The imported `rnn` and `rnn_cell` are TensorFlow classes, described as follows:

The `rnn_cell` module provides a number of basic commonly used RNN cells, such as LSTM and a number of operators that allow us to add dropouts, projections, or embeddings for inputs

Then we load the MNIST dataset using the following library:

```
| from tensorflow.examples.tutorials.mnist import input_data  
| mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)
```

It might take minutes as it downloads the dataset from the Internet.

To classify the images using a Recurrent Neural Network, we must consider every image row as a sequence of pixels, because MNIST image shape is 28×28 pixels, we will then handle 28 sequences of 28 timesteps for every sample.

We then define the following parameters:

```
MNIST data input (image shape: 28x28)  
n_input = 28  
the timesteps  
n_steps = 28  
The number of features in the hidden layer:  
n_hidden = 128  
MNIST total classes (0-9 digits)  
n_classes = 10
```

Here we define our parameters that we will be using in the learning process:

```
learning_rate = 0.001  
training_iters = 100000  
batch_size = 128  
display_step = 10
```

Define our input data (the images) as `x`. The datatype for this tensor is set to `float` and the shape is set to `[None, n_steps, n_input]`. The `None` parameter means that the tensor may hold an arbitrary number of images:

```
| x = tf.placeholder("float", [None, n_steps, n_input])
```

Then we have the placeholder variable for the true labels associated with the images that were input in the placeholder variable `x`. The shape of this placeholder variable is `[None, n_classes]`, which means it may hold an arbitrary number of labels and each label is a vector of length `n_classes`, which is `10` in this case:

```
y = tf.placeholder("float", [None, n_classes])  
weights = {  
    'out': tf.Variable(tf.random_normal([n_hidden, n_classes]))}  
biases = {  
    'out': tf.Variable(tf.random_normal([n_classes]))}
```

We define the network through the `RNN` function:

```
| def RNN (x, weights, biases):
```

We set the input `x` data shape to correctly match the `RNN` function requirements. Notice the following:

- The current input data will be (`batch_size, n_steps, n_input`)
- The required shape is a `n_steps` tensors list of shape (`batch_size, n_input`)

In order to do this, we must perform some transformations on the `x` input tensor. The first operation is to permute a transposition of the current input data:

```
|     x = tf.transpose(x, [1, 0, 2])
```

This operation returns a $(28, 28, 128)$ tensor from the $(128, 28, 28)$ current input data. Then, reshape `x`:

```
|     x = tf.reshape(x, [-1, n_input])
```

It returns a `n_steps × batch_size, n_input` tensor. Split the `x` tensor to get the required list of `n_steps` tensors of shape (`batch_size, n_input`):

```
|     x = tf.split(axis=0, num_or_size_splits=n_steps, value=x)
```

To define our Recurrent Neural Network perform the following steps:

1. **Define a single LSTM cell:** The `BasicLSTMCell` method defines LSTM recurrent network cell. The `forget_bias` parameter is set to `1.0` to reduce the scale of forgetting in the beginning of the training:

```
|     lstm_cell = rnn_cell.BasicLSTMCell(n_hidden, forget_bias=1.0)
```

2. **Build the network:** The `rnn()` operation creates the compute nodes for a given amount of time steps:

```
|     outputs, states = rnn.static_rnn(lstm_cell, x,
|                                         dtype=tf.float32)
```

This operation returns the LSTM cell outputs, where:

- `outputs` is a length and `n_steps` is a list of outputs (one for each input)
- `states` are the cell final states

The resulting tensor of the `RNN` function is a vector of length `10` for determining which of the `10` classes the input image belongs to:

```
| return tf.matmul(outputs[-1], weights['out']) + biases['out']
```

We define the `cost` function and `optimizer` for the predictor:

```
| pred = RNN(x, weights, biases)
```

We used `softmax_cross_entropy_with_logits` as a performance measure and `reduce_mean` to take the average of the cross-entropy for all the image classifications:

```
| New: cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=pred, labels=y))
```

Then we apply the `AdamOptimizer` algorithm to minimize the cross-entropy so it gets as close to zero as possible by changing the variables of the network layers:

```
| optimizer = tf.train.AdamOptimizer(\n|     learning_rate=learning_rate).minimize(cost)
```

We define the accuracy that will be displayed during the computation:

```
| correct_pred = tf.equal(tf.argmax(pred,1), tf.argmax(y,1))\n| accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))
```

We then initialize the variables:

```
| init = tf.global_variables_initializer()
```

It's time to begin the training session--first, we make our session to use it to make our computations happen:

```
| with tf.Session() as sess:\n|     sess.run(init)\n|     step = 1
```

Build the batch sets until we reach the maximum training iterations:

```
|     while step * batch_size < training_iters:\n|         batch_x, batch_y = mnist.train.next_batch(batch_size)
```

Reshape data to get 28×28 sequences of 28 elements:

```
|     batch_x = batch_x.reshape((batch_size, n_steps, n_input))
```

Run through our data in the sequential manner, we break them into pieces and every piece is sized by the batch size that we defined, and then we take every piece and feed it to our optimizer and calculate our accuracy and error and repeat it by feeding new chunks, and so on. In this process, our accuracy gets better the more we feed it:

```
|     sess.run(optimizer, feed_dict={x: batch_x, y: batch_y})\n|     if step % display_step == 0:
```

Compute the accuracy using the following code:

```
|     acc = sess.run(accuracy, feed_dict={x: batch_x, y: batch_y})
```

The loss value, on the other hand, can be calculated as follows:

```
|     loss = sess.run(cost, feed_dict={x: batch_x, y: batch_y})
```

Then we can display the accuracy as follows:

```

print("Iter " + str(step*batch_size) + ", Minibatch Loss= " +\
      "{:.6f}".format(loss) + ", Training Accuracy= " +\
      "{:.5f}".format(acc))
step += 1
print("Optimization Finished!")

```

Finally, we test the RNN model, on a subset (or batch set) of images:

```

test_len = 128
test_data = mnist.test.images[:test_len]\n    .reshape((-1, n_steps, n_input))
test_label = mnist.test.labels[:test_len]
print("Testing Accuracy:", \
sess.run(accuracy, feed_dict={x: test_data, y: test_label}))

```

The resulting output is shown as follows:

```

>>>
Extracting /tmp/data/train-images-idx3-ubyte.gz
Extracting /tmp/data/train-labels-idx1-ubyte.gz
Extracting /tmp/data/t10k-images-idx3-ubyte.gz
Extracting /tmp/data/t10k-labels-idx1-ubyte.gz
Iter 1280, Minibatch Loss= 1.861236, Training Accuracy= 0.35156
Iter 2560, Minibatch Loss= 1.457468, Training Accuracy= 0.51562
Iter 3840, Minibatch Loss= 1.092437, Training Accuracy= 0.64062
Iter 5120, Minibatch Loss= 0.857512, Training Accuracy= 0.73438
Iter 6400, Minibatch Loss= 0.678605, Training Accuracy= 0.78125
Iter 7680, Minibatch Loss= 1.139174, Training Accuracy= 0.61719
Iter 8960, Minibatch Loss= 0.797665, Training Accuracy= 0.75781
Iter 10240, Minibatch Loss= 0.640586, Training Accuracy= 0.81250
Iter 11520, Minibatch Loss= 0.379285, Training Accuracy= 0.90625
Iter 12800, Minibatch Loss= 0.694143, Training Accuracy= 0.72656
. . . . .
Iter 85760, Minibatch Loss= 0.110027, Training Accuracy= 0.96094
Iter 87040, Minibatch Loss= 0.042054, Training Accuracy= 0.98438
Iter 88320, Minibatch Loss= 0.110460, Training Accuracy= 0.96875
Iter 89600, Minibatch Loss= 0.098120, Training Accuracy= 0.97656
Iter 90880, Minibatch Loss= 0.081780, Training Accuracy= 0.96875
Iter 92160, Minibatch Loss= 0.064964, Training Accuracy= 0.97656
Iter 93440, Minibatch Loss= 0.077182, Training Accuracy= 0.96094
Iter 94720, Minibatch Loss= 0.187053, Training Accuracy= 0.95312
Iter 96000, Minibatch Loss= 0.128569, Training Accuracy= 0.96094
Iter 97280, Minibatch Loss= 0.125085, Training Accuracy= 0.96094
Iter 98560, Minibatch Loss= 0.102962, Training Accuracy= 0.96094
Iter 99840, Minibatch Loss= 0.063063, Training Accuracy= 0.98438
Optimization Finished! Testing Accuracy: 0.960938
>>>

```


Source code for RNN image classifier

We listed the source code for the example previously described:

```
import tensorflow as tf
from tensorflow.contrib import rnn
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)

learning_rate = 0.001
training_iters = 100000
batch_size = 128
display_step = 10

n_input = 28
n_steps = 28
n_hidden = 128
n_classes = 10

x = tf.placeholder("float", [None, n_steps, n_input])
y = tf.placeholder("float", [None, n_classes])

weights = {
    'out': tf.Variable(tf.random_normal([n_hidden, n_classes]))
}
biases = {
    'out': tf.Variable(tf.random_normal([n_classes]))
}

def RNN(x, weights, biases):
    x = tf.transpose(x, [1, 0, 2])
    x = tf.reshape(x, [-1, n_input])
    x = tf.split(axis=0, num_or_size_splits=n_steps, value=x)
    lstm_cell = rnn_cell.BasicLSTMCell(n_hidden, forget_bias=1.0)
    outputs, states = rnn.rnn(lstm_cell, x, dtype=tf.float32)
    return tf.matmul(outputs[-1], weights['out']) + biases['out']

pred = RNN(x, weights, biases)
New: cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=pred, labels=y)) optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate).minimize(cost)

correct_pred = tf.equal(tf.argmax(pred,1), tf.argmax(y,1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))

init = tf.global_variables_initializer()
with tf.Session() as sess:
    sess.run(init)
    step = 1
    while step * batch_size < training_iters:
        batch_x, batch_y = mnist.train.next_batch(batch_size)
        batch_x = batch_x.reshape((batch_size, n_steps, n_input))
        sess.run(optimizer, feed_dict={x: batch_x, y: batch_y})
        if step % display_step == 0:
            acc = sess.run(accuracy, feed_dict={x: batch_x, y: batch_y})
            loss = sess.run(cost, feed_dict={x: batch_x, y: batch_y})
            print("Iter "+str(step*batch_size) + ", Minibatch Loss= " + \
                  "{:.6f}".format(loss) + ", Training Accuracy= " + \
                  "{:.5f}".format(acc))
        step += 1
    print("Optimization Finished!")

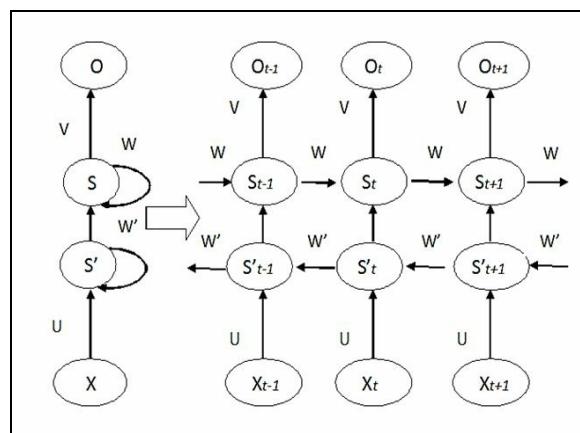
test_len = 128
test_data = mnist.test.images[:test_len].reshape((-1,n_steps,\n                n_input))
test_label = mnist.test.labels[:test_len]
print("Testing Accuracy:",\
sess.run(accuracy, feed_dict={x: test_data, y: test_label}))
```


Bidirectional RNNs

Bidirectional RNNs are based on the idea that the output at time t may depend on previous and future elements in the sequence. To realize this, the output of two RNN must be mixed--one executes the process in a direction and the second runs the process in the opposite direction.

The network splits neurons of a regular RNN into two directions, one for positive time direction (forward states), and another for negative time direction (backward states). By this structure, the output layer can get information from past and future states.

The unrolled architecture of B-RNN is depicted in the following figure:



Unrolled bidirectional RNN

Let's see now, how to implement a B-RNN for an image classification problem. We begin by importing the needed library, notice that `rnn` and `rnn_cell` are TensorFlow libraries:

```
import tensorflow as tf
from tensorflow.contrib import rnn
import numpy as np
```

The network will classify the MNIST images, so we have to load them:

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)
```

We then define the learning parameters:

```
learning_rate = 0.001
training_iters = 100000
batch_size = 128
display_step = 10
```

Next we configure the network's parameters:

```
n_input = 28
n_steps = 28
n_hidden = 128
n_classes = 10
```

Set up the placeholders, which we use to feed to our network. First, we define a placeholder variable for the input images. This allows us to change the images that are input to the TensorFlow graph. The datatype is set to `float` and the tensor's shape is set to `[None, n_steps, n_input]`, `None` stands for a tensor that may hold an arbitrary number of images:

```
| x = tf.placeholder("float", [None, n_steps, n_input])
```

Then we fix a second placeholder variable for the labels associated with the images that were input in the placeholder variable `x`. The shape of this placeholder variable is set to `[None, n_classes]`, which means that it may hold an arbitrary number of labels and each label is a vector of length `num_classes`, which is ¹⁰ in this case:

```
| y = tf.placeholder("float", [None, n_classes])
```

The first variable that must be optimized is `weights` and it is defined here as a TensorFlow variable that must be initialized with random uniform values and whose shape is `[2*n_hidden, n_classes]`.

Here's the `weights` definition:

```
| weights = {
|   'out': tf.Variable(tf.random_normal([2*n_hidden, n_classes]))
| }
```

Then we define the corresponding `biases`:

```
| biases = {
|   'out': tf.Variable(tf.random_normal([n_classes]))
| }
```

With the following `BiRNN` function we define the `weights` and network's `biases`:

```
| def BiRNN(x, weights, biases):
```

To achieve this purpose, we apply the following sequence of tensor transformations:

```
| x = tf.transpose(x, [1, 0, 2])
| x = tf.reshape(x, [-1, n_input])
| x = tf.split(axis=0, num_or_size_splits=n_steps, value=x)
```

Despite the previous model, we define two types of LSTM cells, a forward cell and a backward cell:

```
| lstm_fw_cell = rnn_cell.BasicLSTMCell(n_hidden, forget_bias=1.0)
| lstm_bw_cell = rnn_cell.BasicLSTMCell(n_hidden, forget_bias=1.0)
```

Then we build the bidirectional network using the following imported class `rnn.bidirectional_rnn()`. Similar to the unidirectional case, `rnn.bidirectional_rnn()` takes as input and builds independent forward and backward RNNs with the final forward and backward outputs depth-concatenated:

```
| try:
|   outputs, _, _ = rnn.static_bidirectional_rnn
|         (lstm_fw_cell, lstm_bw_cell, x, dtype=tf.float32)
| except Exception:
|   outputs, _, _ = rnn.bidirectional_rnn
```

```
| outputs = rnn.static_bidirectional_rnn  
|     (lstm_fw_cell, lstm_bw_cell, x, dtype=tf.float32)
```

The `input_size` of forward and backward cells must match. Notice that outputs will have the following format:

```
| [time] [batch] [cell_fw.output_size + cell_bw.output_size]
```

The `BiRNN` function returns an output tensor for determining which of the `10` classes the input image belongs to:

```
| return tf.matmul(outputs[-1], weights['out']) + biases['out']
```

The value returned by the `BiRNN` function will then be passed to the `pred` tensor:

```
| pred = BiRNN(x, weights, biases)
```

We have to compute the cross-entropy value for each classified image, because we must have a measure of how well the model works individually on each image.

Using the cross-entropy to guide the network's optimization procedure we need a single scalar value, so we simply take the cross-entropy average (`tf.reduce_mean`) evaluated for all the classified images:

```
| New: cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=pred, labels=y))
```

The obtained cost measure will be minimized by an `optimizer` variable. We use the `AdamOptimizer`, which is an advanced form of gradient descent:

```
| optimizer = tf.train.AdamOptimizer(\n|     learning_rate=learning_rate).minimize(cost)
```

We add performance measures to be able to display the progress during the training phase. It is a vector of Booleans whether the predicted class equals the true class of each image:

```
| correct_pred = tf.equal(tf.argmax(pred, 1), tf.argmax(y, 1))
```

The `correct_pred` variable is used here to compute the classification accuracy by first type-casting the vector of Booleans to `float`, so that `false` becomes `0` and `true` becomes `1`, and then calculating the average of these numbers:

```
| accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))
```

All the variables must be initialized before we start optimizing them:

```
| init = tf.global_variables_initializer()
```

We then create a `session`, which will execute the graph:

```
| with tf.Session() as sess:  
|     sess.run(init)  
|     step = 1
```

During the session, we get a batch of training examples:

```
| while step * batch_size < training_iters:
```

The `batch_x` variable now holds a subset of training images and `batch_y` is a subset of true labels for those images:

```
|     batch_x, batch_y = mnist.train.next_batch(batch_size)
|     batch_x = batch_x.reshape((batch_size, n_steps, n_input))
```

We put the batch sets into `feed_dict` with the proper names for the placeholder variables, then we run the optimizer through `sess.run`:

```
| sess.run(optimizer, feed_dict={x: batch_x, y: batch_y})
| if step % display_step == 0:
```

We calculate the accuracy and the `loss` values on these sets:

```
acc = sess.run(accuracy,
               feed_dict={x: batch_x, y: batch_y})

loss = sess.run(cost,
                feed_dict={x: batch_x, y: batch_y})
print("Iter " + str(step*batch_size) + \
      ", Minibatch Loss= " + \
      "{:.6f}".format(loss) + ", Training Accuracy= " + \
      "{:.5f}".format(acc))
step += 1
print("Optimization Finished!")
```

At the end of the training session we get a batch of testing examples:

```
test_len = 128
test_data = mnist.test.images\
            [:test_len].reshape((-1, n_steps, n_input))
test_label = mnist.test.labels[:test_len]
```

Finally, we can calculate and display the accuracy on this test set:

```
| print("Testing Accuracy:", \
| sess.run(accuracy, feed_dict={x: test_data, y: test_label}))
```

We show only an excerpt of the output. Here you can visualize the `loss` value and the accuracy evaluated on the batch sets:

```
Successfully downloaded train-images-idx3-ubyte.gz 9912422 bytes. Extracting /tmp/data/train-images-idx3-ubyte.
Successfully downloaded train-labels-idx1-ubyte.gz 28881 bytes. Extracting /tmp/data/train-labels-idx1-ubyte.gz
Successfully downloaded t10k-images-idx3-ubyte.gz 1648877 bytes. Extracting /tmp/data/t10k-images-idx3-ubyte.gz
Successfully downloaded t10k-labels-idx1-ubyte.gz 4542 bytes. Extracting /tmp/data/t10k-labels-idx1-ubyte.gz
Minibatch Loss= 1.877825, Training Accuracy= 0.34375
Iter 2560, Minibatch Loss= 1.582133, Training Accuracy= 0.45312
Iter 3840, Minibatch Loss= 1.172375, Training Accuracy= 0.53125
Iter 5120, Minibatch Loss= 0.942408, Training Accuracy= 0.67188
Iter 6400, Minibatch Loss= 0.678984, Training Accuracy= 0.73438
Iter 7680, Minibatch Loss= 1.089620, Training Accuracy= 0.64844
Iter 8960, Minibatch Loss= 0.658389, Training Accuracy= 0.79688
Iter 10240, Minibatch Loss= 0.576066, Training Accuracy= 0.82031
Iter 11520, Minibatch Loss= 0.404379, Training Accuracy= 0.92188
Iter 12800, Minibatch Loss= 0.627313, Training Accuracy= 0.79688
```

```
| Iter 14080, Minibatch Loss= 0.447121, Training Accuracy= 0.87500  
| . . . . .  
| Iter 90880, Minibatch Loss= 0.048776, Training Accuracy= 1.00000  
| Iter 92160, Minibatch Loss= 0.096100, Training Accuracy= 0.98438  
| Iter 93440, Minibatch Loss= 0.059382, Training Accuracy= 0.98438  
| Iter 94720, Minibatch Loss= 0.088342, Training Accuracy= 0.97656  
| Iter 96000, Minibatch Loss= 0.083945, Training Accuracy= 0.98438  
| Iter 97280, Minibatch Loss= 0.077618, Training Accuracy= 0.97656  
| Iter 98560, Minibatch Loss= 0.141791, Training Accuracy= 0.93750  
| Iter 99840, Minibatch Loss= 0.064927, Training Accuracy= 0.98438 Optimization Finished!
```

Finally, the evaluated accuracy on the test set is as follows:

```
| Testing Accuracy: 0.984375
```


Source code for the bidirectional RNN

Here we report the entire source code for the implemented bidirectional RNN:

```
import tensorflow as tf
from tensorflow.contrib import rnn
import numpy as np

from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)

learning_rate = 0.001
training_iters = 100000
batch_size = 128
display_step = 10

n_input = 28
n_steps = 28
n_hidden = 128
n_classes = 10

x = tf.placeholder("float", [None, n_steps, n_input])
y = tf.placeholder("float", [None, n_classes])

weights = {
    'out': tf.Variable(tf.random_normal([2*n_hidden, n_classes]))
}
biases = {
    'out': tf.Variable(tf.random_normal([n_classes]))
}

def BiRNN(x, weights, biases):
    x = tf.transpose(x, [1, 0, 2])
    x = tf.reshape(x, [-1, n_input])
    x = tf.split(axis=0, num_or_size_splits=n_steps, value=x)
    lstm_fw_cell = rnn_cell.BasicLSTMCell(n_hidden, forget_bias=1.0)
    lstm_bw_cell = rnn_cell.BasicLSTMCell(n_hidden, forget_bias=1.0)
    try:
        outputs, _, _ = rnn.static_bidirectional_rnn(lstm_fw_cell,
                                                    lstm_bw_cell, x, dtype=tf.float32)
    except Exception:# Old TensorFlow version returns output not states
        outputs = rnn.bidirectional_rnn(lstm_fw_cell, lstm_bw_cell,
                                       x, dtype=tf.float32)
    return tf.matmul(outputs[-1], weights['out']) + biases['out']

pred = BiRNN(x, weights, biases)
New: cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=pred, labels=y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(cost)
correct_pred = tf.equal(tf.argmax(pred,1), tf.argmax(y,1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))
init = tf.global_variables_initializer()
with tf.Session() as sess:
    sess.run(init)
    step = 1
    while step * batch_size < training_iters:
        batch_x, batch_y = mnist.train.next_batch(batch_size)
        batch_x = batch_x.reshape((batch_size, n_steps, n_input))
        sess.run(optimizer, feed_dict={x: batch_x, y: batch_y})
        if step % display_step == 0:
            acc = sess.run(accuracy, feed_dict={x: batch_x, y: batch_y})
            loss = sess.run(cost, feed_dict={x: batch_x, y: batch_y})
            print("Iter " + str(step*batch_size) + ", Minibatch Loss=\n" +
                  "{:.6f}".format(loss) + ", Training Accuracy= " +\
                  "{:.5f}".format(acc))
        step += 1
    print("Optimization Finished!")

test_len = 128
test_data = mnist.test.images[:test_len].reshape((-1, n_steps,
```

```
n_input))
test_label = mnist.test.labels[:test_len]
print("Testing Accuracy:", \
sess.run(accuracy, feed_dict={x: test_data, y: test_label}))
```


Text prediction

Language computational models based on RNNs are nowadays among the most successful techniques for statistical language modeling. They can be easily applied in a wide range of tasks, including automatic speech recognition and machine translation.

In this section, we'll explore an RNN model on a challenging task of language processing, guessing the next word in a sequence of text.



You'll find a complete reference for this example in the following page:

<https://www.tensorflow.org/versions/r0.8/tutorials/recurrent/index.html>.

You can download the source code for this example here (official TensorFlow project GitHub page):
<https://github.com/tensorflow/models/tree/master/tutorials/rnn/ptb>.

The files to download are as follows:

- `ptb_word_lm.py`: This file contains code to train the model on the PTB dataset
- `reader.py`: This file contains code to read the dataset

Here we just present only the main ideas.

Dataset

The dataset used is the **Penn Tree Bank (PTB)** language modeling dataset that must be downloaded from Tomas Mikolov's webpage <http://www.fit.vutbr.cz/~imikolov/rnnlm/simple-examples.tgz> and then extracted in your data folder. This dataset consists of 929k training words, 73k validation words, and 82k test words. It has 10k words in its vocabulary including the end-of-sentence marker and a special symbol <unk> for rare words.

Generally speaking, a treebank is a collection of sentences collected according to a syntactic annotation scheme that makes machine readable information concerning the linguistic text structure.

Perplexity

The metric used to measure the goodness of the network is the so-called perplexity. The precise definition requires a lot of math, but it is roughly the average number of choices that the model may have after each word; if we consider spoken English, about one million words, currently the best network models have perplexity 247, a really big number of branching for each word.

PTB model

The PTB model is implemented through the `PTBModel` class that you can find in the `ptb_word_lm.py` file. Here we analyze the basic pseudo code.

The network model consists of a `BasicLSTMCell` cell:

```
|     lstm = rnn_cell.BasicLSTMCell(lstm_size)
```

The memory state of the network is initialized with a vector of zeros; the data is processed in a mini batch set of size `batch_size`:

```
| state = tf.zeros([batch_size, lstm.state_size])
```

For each analyzed word, the probabilities for the continuations of the sentence are computed as follows:

```
| probabilities = []
| for current_batch_of_words in words_in_dataset:
```

The value of `state` is updated after processing each batch of words:

```
|     output, state = lstm(current_batch_of_words, state)
```

The LSTM output is then used to make predictions on the next word:

```
|     logits = tf.matmul(output, softmax_w) + softmax_b
|     probabilities.append(tf.nn.softmax(logits))
|     loss += loss_function(probabilities, target_words)
```

The `loss_function` function minimizes the average negative log probability of the target words. It computes the average per-word perplexity.

Running the example

The PTB model can support small, medium, and large dataset configurations. The small model should be able to reach perplexity below 120 on the test set and the large one below 80, though it might take several hours to train.

We executed the model on the small size dataset. To do this, simply type in your command prompt the following:

```
| python ptb_word_lm --data_path=/tmp/simple-examples/data/ --model small
```

This is where you extracted the previously downloaded PTB dataset in `/tmp/simple-examples/data/`.

The preceding perplexity value after eight hours of training and 13 training epochs are as follows:

```
Epoch: 1 Learning rate: 1.000
0.004 perplexity: 5263.762 speed: 391 wps
0.104 perplexity: 837.607 speed: 429 wps
0.204 perplexity: 617.207 speed: 442 wps
0.304 perplexity: 498.160 speed: 438 wps
0.404 perplexity: 430.516 speed: 436 wps
0.504 perplexity: 386.339 speed: 427 wps
0.604 perplexity: 348.393 speed: 431 wps
0.703 perplexity: 322.351 speed: 432 wps
0.803 perplexity: 301.630 speed: 431 wps
0.903 perplexity: 282.417 speed: 434 wps
Epoch: 1 Train Perplexity: 268.124
Epoch: 1 Valid Perplexity: 180.210
Epoch: 2 Learning rate: 1.000
0.004 perplexity: 209.082 speed: 448 wps
0.104 perplexity: 150.589 speed: 437 wps
0.204 perplexity: 157.965 speed: 436 wps
0.304 perplexity: 152.896 speed: 453 wps
0.404 perplexity: 150.299 speed: 458 wps
0.504 perplexity: 147.984 speed: 462 wps
0.604 perplexity: 143.367 speed: 462 wps
0.703 perplexity: 141.246 speed: 446 wps
0.803 perplexity: 139.299 speed: 436 wps
0.903 perplexity: 135.632 speed: 435 wps
Epoch: 2 Train Perplexity: 133.576
Epoch: 2 Valid Perplexity: 143.072
. . . . .
Epoch: 12 Learning rate: 0.008
0.004 perplexity: 57.011 speed: 347 wps
0.104 perplexity: 41.305 speed: 356 wps
0.204 perplexity: 45.136 speed: 356 wps
0.304 perplexity: 43.386 speed: 357 wps
0.404 perplexity: 42.624 speed: 358 wps
0.504 perplexity: 41.980 speed: 358 wps
0.604 perplexity: 40.549 speed: 357 wps
0.703 perplexity: 39.943 speed: 357 wps
0.803 perplexity: 39.287 speed: 358 wps
0.903 perplexity: 37.949 speed: 359 wps
Epoch: 12 Train Perplexity: 37.125
Epoch: 12 Valid Perplexity: 123.571
Epoch: 13 Learning rate: 0.004
0.004 perplexity: 56.576 speed: 365 wps
0.104 perplexity: 40.989 speed: 358 wps
0.204 perplexity: 44.809 speed: 358 wps
0.304 perplexity: 43.082 speed: 356 wps
0.404 perplexity: 42.332 speed: 356 wps
0.504 perplexity: 41.694 speed: 356 wps
0.604 perplexity: 40.275 speed: 357 wps
0.703 perplexity: 39.673 speed: 356 wps
```

```
0.803 perplexity: 39.021 speed: 356 wps
0.903 perplexity: 37.690 speed: 356 wps
Epoch: 13 Train Perplexity: 36.869
Epoch: 13 Valid Perplexity: 123.358
Test Perplexity: 117.171
```


Summary

In this chapter, we provided an overview of RNNs. These are a class of neural networks where the connections between the units form direct cycles, thus giving the possibility to manage temporal and sequential data. We have described the LSTM architecture. The basic idea of this architecture is to improve the RNN providing it with an explicit memory.

LSTM networks are equipped with special hidden units, said memory cells, whose behavior is to remember the previous input for a long time. These cells take in input, at each instant of time, the previous state, and the current input of the network. Combining them with the current contents of memory, and deciding by a gating mechanism by other units what to keep and which to delete things from memory, LSTM have proved very useful and effective learning of long-term dependency.

We have therefore implemented two models of neural networks--the LSTM for a classification problem of images from the MNIST dataset.

Finally, we saw TensorFlow in action on the complex task of predicting the next word in a sequence of text.

The next chapter will introduce the fundamental issue of GPU computing. This technology was in fact the main architect of development of deep learning technology in recent years. We will describe the main GPU's features, and we'll see how to configure TensorFlow for use with GPUs.

GPU Computing

Deep Neural Networks (DNNs) are structured in a very uniform manner, such that, at each layer of a network thousands of identical artificial neurons perform the same computation. Therefore, DNN's architecture fits quite well with the kinds of computation that a GPU can efficiently perform.

GPU have additional advantages over CPU; these include having more computational units and having a higher bandwidth to retrieve from memory.

Furthermore, in many deep learning applications that require a lot of computational effort, GPU graphics specific capabilities can be exploited to further speed up calculations.

This chapter is organized as follows:

- GPGPU computing
- GPGPU history
- The CUDA architecture
- GPU programming model
- TensorFlow GPU set up
- TensorFlow GPU management
- Assigning a single GPU on a multi-GPU system
- Using multiple GPUs

GPGPU computing

There are several reasons that have led to deep learning to be developed and placed at the center of attention in the field of machine learning only in recent decades.

One reason, perhaps the main one, is surely represented by the progress in hardware, with the availability of new processors, such as **graphics processing units (GPUs)**, which have greatly reduced the time needed for training networks, lowering them to 10/20 times.

In fact, since the connections between the individual neurons have a weight numerically estimated, and that networks learn by calibrating the weights properly, we understand how the network's complexity requires a huge increase, in computing power, required for graphics processors used in the experiments.

GPGPU history

The **general purpose computing on graphics processing unit (GPGPU)** recognizes the trend to employ GPU technology for non-graphic applications. Until 2006, the graphics API OpenGL and DirectX standards were the only ways to program with the GPU. Any attempt to execute arbitrary calculations on the GPU was subject to the programming restrictions of those APIs.

The GPUs were designed to produce a color for each pixel on the screen using programmable arithmetic units called **pixel shaders**. The programmers realized that if the inputs were numerical data, with a different meaning from the pixel colors, then they could program the pixel shader to perform arbitrary computations.

The GPU was deceived by showing general tasks such as rendering tasks; this deception was intelligent, but also very convoluted.

There were memory limitations because the programs could only receive a handful of input color and texture units as input data. It was almost impossible to predict how a GPU would handle the floating-point data (if it was able to process it) so many scientific calculations could not use the GPU.

Anyone who wanted to resolve a numerical problem would have to learn OpenGL or DirectX, the only ways to communicate with the GPU.

The CUDA architecture

In 2006, NVIDIA was presented as the first GPU to support DirectX 10; the GeForce 8800GTX was also the first GPU to use the CUDA architecture. This architecture included several new components designed specifically for GPU computing and aimed to remove the limitations that prevented them that previous GPUs were used for non-graphical calculations. In fact, the execution units on the GPU could read and write arbitrary memory as well as access a cache maintained in software called **shared memory**. These architectural features were added to make a CUDA GPU that also excelled in general purpose calculations as well as in traditional graphics tasks.

The following figure summarizes the division of space between the various components of a **graphics processing unit (GPU)** and a **central processing unit (CPU)**. As you can see, a GPU devotes more transistors to data processing; it is a highly parallel, multithreaded, and many core processor:



CPU versus GPU architecture

*Almost all the space on the GPU chip is dedicated to the ALU, apart from cache and control, making it suitable for repetitive calculations on large amounts of data. The GPU accesses a local memory and is connected to the system, that is, the CPU via a bus--currently, the **Peripheral Component Interconnect Express (PCI Express)**.*

The graphics chip consists of a series of multiprocessors, the **Streaming Multiprocessor (SM)**.

The number of these multiprocessors depends on the specific characteristics and the performance class of each GPU.

Each multiprocessor is in turn formed by stream processors (or cores). Each of these processors can perform basic arithmetic operations on integer or floating-point numbers in single and double precision.

GPU programming model

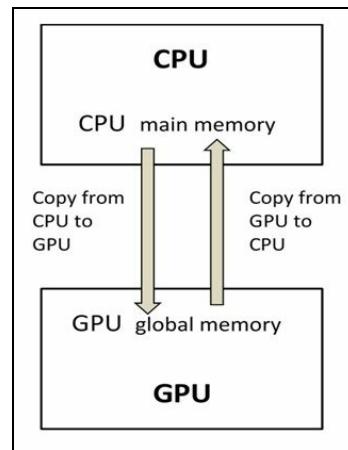
At this point it is necessary to introduce some basic concepts to understand the CUDA programming model. The first distinction is between host and device.

The code executed in the host side is the part of code executed on the CPU, and this will also include the RAM and the hard disk.

However, the code executed on the device is automatically loaded on the graphic card and run on the latter. Another important concept is the kernel; it stands for a function performed on the device and launched from the host.

The code defined in the kernel will be performed in parallel by an array of threads. The following figure summarizes how the GPU programming model works:

- The running program will have source code to run on CPU and code to run on GPU
- CPU and GPU have separated memories
- The data is transferred from CPU to GPU to be computed
- The data output from GPU computation is copied back to CPU memory



GPU programming model

TensorFlow GPU set up

The NVIDIA deep learning SDK offers powerful tools and libraries for the development of deep learning frameworks such as Caffe, CNTK, TensorFlow, Theano, and Torch. The NVIDIA deep learning SDK provides powerful tools and libraries for designing and deploying GPU-accelerated deep learning applications. It includes libraries for deep learning primitives, inference, video analytics, linear algebra, sparse matrices, and multi-GPU communications. The current implementation supports the following SDKs:

- **Deep learning primitives:** (<https://developer.nvidia.com/cudnn>) High-performance building blocks for deep neural network applications including convolutions, activation functions, and tensor transformations.
- **Deep learning inference engine:** (<https://developer.nvidia.com/tensorrt>) High-performance deep learning inference runtime for production deployment.
- **Deep learning for video analytics:** (<https://developer.nvidia.com/deepstream-sdk>) High-level C++ API and runtime for GPU-accelerated transcoding and deep learning inference.
- **Linear algebra:** (<https://developer.nvidia.com/cUBLAS>) GPU-accelerated BLAS functionality that delivers 6x to 17x faster performance than CPU-only BLAS libraries. The XLA (<https://www.tensorflow.org/performance/xla/>) is a domain-specific compiler for linear algebra that optimizes TensorFlow computations. Although this experimental (that is, under active development), however, the possible results are improvements in speed, memory usage, and portability on server and mobile platforms.
- **Sparse matrix operations:** (<https://developer.nvidia.com/cusparse>) GPU-accelerated linear algebra subroutines for sparse matrices that deliver up to 8x faster performance than CPU BLAS (MKL), ideal for applications such as natural language processing.
- **Multi-GPU communication:** (<https://github.com/NVIDIA/nccl>) Collective communication routines, such as all-gather, reduce, and broadcast that accelerate multi-GPU deep learning training on up to eight GPUs.

 *However, the deep learning SDK requires CUDA toolkit (<https://developer.nvidia.com/cuda-toolkit>), which offers a comprehensive development environment for building new GPU-accelerated deep learning algorithms, and dramatically increasing the performance of existing applications.*

To use TensorFlow with NVIDIA GPUs, the first step is to install the CUDA toolkit.



See more at: <https://developer.nvidia.com/cuda-downloads>.

Once the CUDA toolkit is installed, you must download the cuDNN v5.1 library from <https://developer.nvidia.com/cudnn> for Linux.



A more detailed installation including TensorFlow and Bazel for GPU computation using cuDNN, please refer to this URL <http://www.nvidia.com/object/gpu-accelerated-applications-tensorflow-installation.html>.

The cuDNN is a library that helps accelerate deep learning frameworks, such as TensorFlow or Theano. Here's a brief explanation from the NVIDIA website.

The NVIDIA **CUDA® Deep Neural Network library (cuDNN)** is a GPU-accelerated library of primitives for DNNs. The cuDNN provides highly tuned implementations for standard routines such as forward and backward convolution, pooling, normalization, and activation layers. The cuDNN is part of the NVIDIA deep learning SDK.

Before installing you'll need to register for NVIDIA's **Accelerated Computing Developer Program**. Once registered, login and download CudNN 5.1 to your local computer.



During the writing of this book, the version of cuDNN was 5.1 and released on Jan 20, 2017, for CUDA 8.0. For more details at <https://developer.nvidia.com/rdp/cudnn-download>. You should see options in the download page as shown in the following figure:

cuDNN Download

NVIDIA cuDNN is a GPU-accelerated library of primitives for deep neural networks.

I Agree To the Terms of the [cuDNN Software License Agreement](#)

Please check your framework documentation to determine the recommended version of cuDNN.
If you are using cuDNN with a Pascal (GTX 1080, GTX 1070), version 5 or later is required.

[Download cuDNN v5.1 \[Jan 20, 2017\], for CUDA 8.0](#)

[cuDNN User Guide](#)

[cuDNN Install Guide](#)

[cuDNN v5.1 Library for Linux](#)

[cuDNN v5.1 Library for Power8](#)

[cuDNN v5.1 Library for Windows 7](#)

[cuDNN v5.1 Library for Windows 10](#)

[cuDNN v5.1 Library for OSX](#)

[cuDNN v5.1 Release Notes](#)

[cuDNN v5.1 Runtime Library for Ubuntu14.04 \(Deb\)](#)

[cuDNN v5.1 Developer Library for Ubuntu14.04 \(Deb\)](#)

[cuDNN v5.1 Code Samples and User Guide \(Deb\)](#)

[cuDNN v5.1 Runtime Library for Ubuntu16.04 Power8 \(Deb\)](#)

[cuDNN v5.1 Developer Library for Ubuntu16.04 Power8 \(Deb\)](#)

[cuDNN v5.1 Code Samples and User Guide Power8 \(Deb\)](#)

[Download cuDNN v5.1 \[Jan 20, 2017\], for CUDA 7.5](#)

[Download cuDNN v5 \[May 27, 2016\], for CUDA 8.0](#)

[Download cuDNN v5 \[May 12, 2016\], for CUDA 7.5](#)

As shown in the preceding figure, you will have to select your platform/OS type. The following installation is for Linux. Now, once downloaded, uncompress the files and copy them into the CUDA toolkit directory (assumed here to be in `/usr/local/cuda/`):

```
| $ sudo tar -xvf cudnn-8.0-linux-x64-v5.1-rc.tgz -C /usr/local
```


Update TensorFlow

We're assuming you'll be using TensorFlow for building your DNN models. Simply update TensorFlow, via `pip` with the upgrade flag.

Here, we suppose you're currently using TensorFlow 1.0.1:

```
| pip install - upgrade https://storage.googleapis.com/tensorflow/linux/gpu/tensorflow-0.10.0rc0-cp27-none-linux_
```

Now you should have everything you need to run a model using your GPU.



For other versions Operating System, Python version or CPU only vs. GPU support, you should refer to this URL:

https://www.tensorflow.org/install/install_linux#the_url_of_the_tensorflow_python_package

TensorFlow GPU management

In TensorFlow the supported devices are represented as strings. For example:

- `/cpu:0`: The CPU of your machine
- `/gpu:0`: The GPU of your machine, if you have one
- `/gpu:1`: The second GPU of your machine, and so on

The execution flow gives priority when an operation is assigned to a GPU device.

Programming example

To use a GPU in your TensorFlow program, just type the following:

```
| with tf.device("/gpu:0"):
```

Followed by the setup operations. This line of code will create a new context manager, telling TensorFlow to perform those actions on the GPU.

Let's consider the following example, in which we want to execute the following sum of two matrices, $A^n + B^n$.

Define the basic imports:

```
| import numpy as np  
| import tensorflow as tf  
| import datetime
```

We can configure a program to find out which devices your operations and tensors are assigned. To realize this, we'll create a session with the following `log_device_placement` parameter set to `True`:

```
| log_device_placement = True
```

Then we fix the `n` parameter, that is, the number of multiplication to perform:

```
| n=10
```

Then we build two random large matrices. We use the NumPy `rand` function to perform this operation:

```
| A = np.random.rand(10000, 10000).astype('float32')  
| B = np.random.rand(10000, 10000).astype('float32')
```

`A` and `B` will be respectively, of size `10000x10000`.

The following array will be used to store results:

```
| c1 = []  
| c2 = []
```

Here, we define the kernel matrix multiplication function that will be performed by the GPU:

```
| def matpow(M, n):  
|     if n < 1:  
|         return M  
|     else:  
|         return tf.matmul(M, matpow(M, n-1))
```

As previously explained, we must configure the GPU and the GPU with the operations to perform.

The GPU will compute the A^n and B^n operations and store results in `c1`:

```
| with tf.device('/gpu:0'):
```

```
a = tf.placeholder(tf.float32, [10000, 10000])
b = tf.placeholder(tf.float32, [10000, 10000])
c1.append(matpow(a, n))
c1.append(matpow(b, n))
```



In case if the above code does not work use /job:localhost/replica:0/task:0/cpu:0 as the GPU device (that is, will be executed using CPU).

The addition of all elements in `c1`, that is, $A^n + B^n$, is performed by the CPU, so we define it as follows:

```
| with tf.device('/cpu:0'):
|   sum = tf.add_n(c1)
```

The `datetime` class permits to evaluate the computational time:

```
t1_1 = datetime.datetime.now()
with tf.Session(config=tf.ConfigProto(
    log_device_placement=log_device_placement)) as sess:
    sess.run(sum, {a:A, b:B})
t2_1 = datetime.datetime.now()
```

Computational time is then displayed using:

```
| print("GPU computation time: " + str(t2_1-t1_1))
```

I am using a GeForce 840M graphic card, the results are as follows:

```
| GPU computation time: 0:00:13.816644
```


Source code for GPU computation

Here is the full code for the previous example:

```
import numpy as np
import tensorflow as tf
import datetime

log_device_placement = True

n = 10

A = np.random.rand(10000, 10000).astype('float32')
B = np.random.rand(10000, 10000).astype('float32')

c1 = []
c2 = []

def matpow(M, n):
    if n < 1: #Abstract cases where n < 1
        return M
    else:
        return tf.matmul(M, matpow(M, n-1))

with tf.device('/gpu:0'):
    a = tf.placeholder(tf.float32, [10000, 10000])
    b = tf.placeholder(tf.float32, [10000, 10000])
    c1.append(matpow(a, n))
    c1.append(matpow(b, n))

with tf.device('/cpu:0'):
    sum = tf.add_n(c1) #Addition of all elements in c1, i.e. A^n + B^n

t1_1 = datetime.datetime.now()
with tf.Session(config=tf.ConfigProto(
    log_device_placement=log_device_placement)) as sess:
    sess.run(sum, {a:A, b:B})
t2_1 = datetime.datetime.now()
```



For the following case if the preceding code does not work or if there's no GPU support in your device, use /job:localhost/replica:0/task:0/cpu:0 as the CPU device.

GPU memory management

In some cases it is desirable for the process to only allocate a subset of the available memory, or to only grow the memory usage as it is needed by the process. TensorFlow provides two configuration options on the session to control this. The first is the `allow_growth` option, which attempts to allocate only as much GPU memory based on runtime allocations, it starts out allocating very little memory, and as sessions get run and more GPU memory is needed, we extend the GPU memory region needed by the TensorFlow process.

Note that we do not release memory, since that can lead to even worse memory fragmentation. To turn this option on, set the option in `ConfigProto` by:

```
config = tf.ConfigProto()
config.gpu_options.allow_growth = True
session = tf.Session(config=config, ...)
```

The second method is the `per_process_gpu_memory_fraction` option, which determines the fraction of the overall amount of memory that each visible GPU should be allocated.

For example, you can tell TensorFlow to only allocate 40% of the total memory of each GPU by:

```
config = tf.ConfigProto()
config.gpu_options.per_process_gpu_memory_fraction = 0.4
session = tf.Session(config=config, ...)
```

This is useful if you want to truly bind the amount of GPU memory available to the TensorFlow process.

Assigning a single GPU on a multi-GPU system

If you have more than one GPU in your system, the GPU with the lowest ID will be selected by default. If you would like to run on a different GPU, you will need to specify the preference explicitly.

For example, we can try to change the GPU assignation in the previous code:

```
with tf.device('/gpu:1'):
    a = tf.placeholder(tf.float32, [10000, 10000])
    b = tf.placeholder(tf.float32, [10000, 10000])
    c1.append(matpow(a, n))
    c1.append(matpow(b, n))
```

In this way, we are telling GPU to execute the kernel function.

If the device we have specified does not exist (as in my case), you will get the following error message on the console (or terminal):

```
InvalidArgumentException :
InvalidArgumentException (see above for traceback): Cannot assign a device to node 'Placeholder_1': Could not satisfy
[[Node: Placeholder_1 = Placeholder[dtype=DT_FLOAT, shape=[100,100], _device="/device:GPU:1"]()]]
```

If you would like TensorFlow to automatically choose an existing and supported device to run the operations in case the specified one doesn't exist, you can set `allow_soft_placement` to `True` in the configuration option when creating the session.

Again, we fix `/gpu:1` for the following node:

```
with tf.device('/gpu:1'):
    a = tf.placeholder(tf.float32, [10000, 10000])
    b = tf.placeholder(tf.float32, [10000, 10000])
    c1.append(matpow(a, n))
    c1.append(matpow(b, n))
```

Then we build a session with the following `allow_soft_placement` parameter set to `True`:

```
with tf.Session(config=tf.ConfigProto(
    allow_soft_placement=True,
    log_device_placement=log_device_placement)) \
    as sess:
```

In this way, when running the session `InvalidArgumentException` will not be displayed, but instead a correct result will be displayed, in this case, with a slight delay:

```
| GPU computation time: 0:00:15.006644
```


Source code for GPU with soft placement

We report, just for better understanding, the complete source code:

```
import numpy as np
import tensorflow as tf
import datetime

log_device_placement = True
n = 10

A = np.random.rand(10000, 10000).astype('float32')
B = np.random.rand(10000, 10000).astype('float32')

c1 = []

def matpow(M, n):
    if n < 1: #Abstract cases where n < 1
        return M
    else:
        return tf.matmul(M, matpow(M, n-1))

with tf.device('/gpu:1'):
    a = tf.placeholder(tf.float32, [10000, 10000])
    b = tf.placeholder(tf.float32, [10000, 10000])
    c1.append(matpow(a, n))
    c1.append(matpow(b, n))

with tf.device('/cpu:0'):
    sum = tf.add_n(c1)

t1_1 = datetime.datetime.now()
with tf.Session(config=tf.ConfigProto\
                  (allow_soft_placement=True, \
                   log_device_placement=log_device_placement))\
    as sess:
    sess.run(sum, {a:A, b:B})
t2_1 = datetime.datetime.now()
```


Using multiple GPUs

If you would like to run TensorFlow on multiple GPUs, you can construct your model assigning a specific chunk of code to a GPU. For example, having two GPUs, we can split the previous code in this way, assigning the first matrix computation to the first GPU as follows:

```
with tf.device('/gpu:0'):
    a = tf.placeholder(tf.float32, [10000, 10000])
    c1.append(matpow(a, n))
```

The second matrix computation to the second GPU as follows:

```
with tf.device('/gpu:1'):
    b = tf.placeholder(tf.float32, [10000, 10000])
    c1.append(matpow(b, n))
```

Finally, your CPU will manage the results; also note that we used the shared `c1` array to collect them:

```
with tf.device('/cpu:0'):
    sum = tf.add_n(c1)
    print(sum)
```


Source code for multiple GPUs management

The complete source code is fully listed here:

```
import numpy as np
import tensorflow as tf
import datetime

log_device_placement = True
n = 10

A = np.random.rand(10000, 10000).astype('float32')
B = np.random.rand(10000, 10000).astype('float32')

c1 = []

def matpow(M, n):
    if n < 1: #Abstract cases where n < 1
        return M
    else:
        return tf.matmul(M, matpow(M, n-1))

#FIRST GPU
with tf.device('/gpu:0'):
    a = tf.placeholder(tf.float32, [10000, 10000])
    c1.append(matpow(a, n))

#SECOND GPU
with tf.device('/gpu:1'):
    b = tf.placeholder(tf.float32, [10000, 10000])
    c1.append(matpow(b, n))

with tf.device('/cpu:0'):
    sum = tf.add_n(c1)

t1_1 = datetime.datetime.now()
with tf.Session(config=tf.ConfigProto(
                    (allow_soft_placement=True, \
                     log_device_placement=log_device_placement)) \
                  as sess:
    sess.run(sum, {a:A, b:B})
t2_1 = datetime.datetime.now()
```


Summary

GPUs are specialized hardware components that were originally developed for graphics applications. However, it was realized that GPUs could be used to perform computations in a DNN architecture.

In the second part of this chapter, we saw how to install a GPU-enabled TensorFlow and how to manage GPU devices.

In the next chapter, we will discuss some advanced programming features using TensorFlow by integrating with other third-party tools such as, Keras, PrettyTensor and TFLearn. In addition, we'll give an overview of some TensorFlow-based framework--Keras, PrettyTensor and TFLearn.

We'll show the main topics of these frameworks, with very interesting application examples.

Advanced TensorFlow Programming

Development of deep learning networks, especially when testing new models, may require rapid prototyping. For this reason, there have been developed several TensorFlow-based libraries, abstracting many programming concepts and providing higher-level building blocks.

In this chapter, we'll give an overview of the libraries such as, Keras, Pretty Tensor, and TFLearn.

For each library, we'll describe its main characteristics, with an application example.

The chapter is organized as follows:

- Introducing Keras
- Building deep learning models
- Sentiment classification of movie reviews
- Adding a convolutional layer
- Pretty Tensor
- Digit classifier
- TFLearn
- Titanic survival predictor

Introducing Keras

Keras is a minimalist, high-level neural networks library, capable of running on top of TensorFlow. It was developed with a focus on enabling easy and fast prototyping and experimentation. Keras runs on Python 2.7 or 3.5, and can seamlessly execute on GPUs and CPUs given the underlying frameworks. It is released under the MIT license.

Keras was developed and maintained by François Chollet, a Google engineer, following these design principles:

- **Modularity:** A model is understood as a sequence or a graph of the standalone, fully configurable modules that can be plugged together with as few restrictions as possible. Neural layers, cost functions, optimizers, initialization schemes, and activation functions are all standalone modules that can be combined to create new models.
- **Minimalism:** Each module must be short (few lines of code) and simple. The source code should be transparent upon the first reading.
- **Extensibility:** New modules are simple to add (as new classes and functions), and existing modules provide examples. To be able to easily create new modules allows for total expressiveness, making Keras suitable for advanced research.
- **Python:** No separate model configuration files in a declarative format. Models are described in Python code, which is compact, easier to debug, and allows for ease of extensibility.

The following screenshot shows the Keras homepage:

The screenshot shows the Keras Documentation homepage. The header features a red bar with the Keras logo and the text "Keras Documentation". Below the header is a search bar labeled "Search docs". The main content area has a white background. At the top right, there are links for "Docs » Home" and "Edit on GitHub". The main title is "Keras: Deep Learning library for Theano and TensorFlow". Below the title, a sub-headline says "You have just found Keras.". The text explains that Keras is a high-level neural networks library written in Python and capable of running on top of either TensorFlow or Theano. It emphasizes the principle of "Being able to go from idea to result with the least possible delay is key to doing good research." Further down, it states "Use Keras if you need a deep learning library that:" followed by a bulleted list of features. At the bottom of the page, there are links to "Read the documentation at Keras.io." and "Keras is compatible with: Python 2.7-3.5."

The Keras home page

Installation

To install Keras, you must also have an installation of TensorFlow on your system already. Keras can be installed easily using `pip`, as follows:

```
| sudo pip install keras
```

For Python 3+ use the following command:

```
| sudo pip3 install keras
```

During the writing of this book, the most recent version of Keras is version 2.0.2. You can check your version of Keras on the command line, using the following snippet:

```
| python -c "import keras; print keras.__version__"
```

Running the preceding script, you will see the following output:

```
| 2.0.2
```

You can also upgrade your installation of Keras using the same method:

```
| sudo pip install --upgrade keras
```


Building deep learning models

The core data structure of Keras is a model, which is a way to organize layers. There are two types of model:

- **Sequential:** The main type of model. It is simply a linear stack of layers.
- **Keras functional API:** These are used for more complex architectures.

You define a sequential model as follows:

```
| from keras.models import Sequential  
| model = Sequential()
```

Once a model is defined, you can add one or more layers. The stacking operation is provided by the `add()` statement:

```
| from keras.layers import Dense, Activation
```

For example, add a first fully connected NN layer and the `Activation` function:

```
| model.add(Dense(output_dim=64, input_dim=100))  
| model.add(Activation("relu"))
```

Then add a second `softmax` layer:

```
| model.add(Dense(output_dim=10))  
| model.add(Activation("softmax"))
```

If the model looks fine, you must compile the model by using the `model.compile` function, specifying the `loss` function and the `optimizer` function to be used:

```
| model.compile(loss='categorical_crossentropy', \  
|                 optimizer='sgd', \  
|                 metrics=['accuracy'])
```

You may configure your optimizer. Keras tries to make programming reasonably simple, allowing the user to be fully in control when they need to be. Once compiled, the model must be fitted to the data:

```
| model.fit(X_train, Y_train, nb_epoch=5, batch_size=32)
```

Alternatively, you can feed batches to your model manually:

```
| model.train_on_batch(X_batch, Y_batch)
```

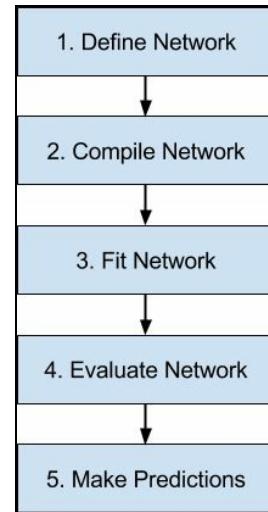
Once trained, you can use your model to make predictions on new data:

```
| classes = model.predict_classes(X_test, batch_size=32)  
| proba = model.predict_proba(X_test, batch_size=32)
```

We can summarize the construction of deep learning models in Keras as follows:

1. **Define your model:** Create a sequence and add layers.
2. **Compile your model:** Specify loss functions and optimizers.
3. **Fit your model:** Execute the model using data.
4. **Evaluate the model:** Keep an evaluation of your training dataset.
5. **Make predictions:** Use the model to generate predictions on new data.

The following figure depicts the preceding processes:



Keras programming model

In the following section, we'll look at how to use the Keras sequential model to study the sentiment classification problem of movie reviews.

Sentiment classification of movie reviews

Sentiment analysis is the capability to decipher the opinions contained in a written or spoken text. The main purpose of this technique is to identify the sentiment (or polarity) of a lexical expression, which may have a neutral, positive, or negative connotation.

The problem we want to resolve is the IMDB movie review sentiment classification problem. Each movie review is a variable sequence of words, and the sentiment (positive or negative) of each movie review must be classified.

This problem is very complex, because the sequences can vary in length; they can also be part of a large vocabulary of input symbols.

The solution requires the model to learn long-term dependencies between symbols in the input sequence.

The IMDB dataset contains 25,000 highly polarized movie reviews (good or bad) for training and the same amount again for testing. The data was collected by Stanford researchers, and was used in a 2011 paper, where a split of 50/50 of the data was used for training and testing. In this paper, an accuracy of 88.89% was achieved.

Once we define our problem, we are ready to develop an LSTM model to classify the sentiment of movie reviews. We can quickly develop an LSTM for the IMDB problem and achieve good accuracy.

Let's start off by importing the classes and functions required for this model, and initializing the random number generator to a constant value, to ensure we can easily reproduce the results:

```
import numpy
from keras.datasets import imdb
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers.embeddings import Embedding
from keras.preprocessing import sequence
numpy.random.seed(7)
```

We load the IMDB dataset. We are constraining the dataset to the top 5,000 words. We also split the dataset into training (50%) and testing (50%) sets.



Keras provides access to the IMDb dataset (<http://www.imdb.com/interfaces>) built-in. alternatively, you also can download the IMDb dataset from Kaggle website at <https://www.kaggle.com/deepmatrix/imdb-5000-movie-dataset>.

The `imdb.load_data()` function allows you to load the dataset in a format that is ready for use in neural network and deep learning models. The words have been replaced by integers, which indicate the ordered frequency of each word in the dataset. The sentences in each review are therefore comprised of a sequence of integers.

Here's the code:

```
top_words = 5000\
(X_train, y_train), (X_test, y_test) =\
    imdb.load_data(nb_words=top_words)
```

Next, we need to truncate and pad the input sequences so that they are all the same length for modeling. The model will learn the zero values that carry no information because, although the sequences are not the same length in terms of content, same length vectors are required to perform the computation in Keras. The sequence length in each review varies, so we constrained each review to 500 words, truncating long reviews and padding the shorter reviews with zero values:

Let's see:

```
max_review_length = 500\
X_train = sequence.pad_sequences\
    (X_train, maxlen=max_review_length)
X_test = sequence.pad_sequences\
    (X_test, maxlen=max_review_length)
```

We can now define, compile, and fit our LSTM model.

To resolve the sentiment classification problem, we'll use the word embedding technique, which consists of representing words in a continuous vector space, that is, an area in which the words that are semantically similar are mapped in neighboring points. Word embedding is based on the distributional hypothesis, that is, the words that appear in a given context must share the same semantic meaning. Each movie review will then be mapped into a real vector domain, where the similarity between words, in terms of meaning, translates to closeness in the vector space. Keras provides a convenient way to convert positive integer representations of words into word embedding by an embedding layer.

Here, we define the length of the embedding vector and the model:

```
embedding_vector_length = 32
model = Sequential()
```

The first layer is the embedded layer, which uses 32 length vectors to represent each word:

```
model.add(Embedding(top_words,
    embedding_vector_length,
    input_length=max_review_length))
```

The next layer is the LSTM layer, with 100 memory units. Finally, because this is a classification problem, we use a Dense output layer with a single neuron and a sigmoid activation function to make 0 or 1 predictions for the two classes (good and bad) in the problem:

```
model.add(LSTM(100))
model.add(Dense(1, activation='sigmoid'))
```

Because it is a binary classification problem, the binary_crossentropy function is used as a loss function, while the optimizer function used here is the adam optimization algorithm (we also encountered it in a

previous TensorFlow implementation):

```
model.compile(loss='binary_crossentropy', \
               optimizer='adam', \
               metrics=['accuracy'])
print(model.summary())
```

We fit only three epochs, because the problem quickly overfits. A batch size of 64 reviews is used to space out weight updates:

```
model.fit(X_train, y_train, \
           validation_data=(X_test, y_test), \
           nb_epoch=3, \
           batch_size=64)
```

Then, we estimate the model's performance on unseen reviews:

```
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

Running this example produces the following output:

```
Epoch 1/3
16750/16750 [=====] - 107s - loss: 0.5570 - acc: 0.7149
Epoch 2/3
16750/16750 [=====] - 107s - loss: 0.3530 - acc: 0.8577
Epoch 3/3
16750/16750 [=====] - 107s - loss: 0.2559 - acc: 0.9019
Accuracy: 86.79%
```

You can see that this simple LSTM, with little tuning, achieves near state of the art results on the IMDB problem. Importantly, this is a template that you can use to apply LSTM networks to your own sequence classification problems.

Source code for the Keras movie classifier

Here is the complete source code; you can see how short the number of lines of code as shown in the following. However, if you experience error saying that there's no module called `keras.datasets` and so on, you should install `keras` package using the following command:

```
| $ sudo pip install keras
```



Alternatively, download the source code of Keras from <https://pypi.python.org/pypi/Keras>, unzip the file and run the source code using Python 3 (within the `keras` folder) as follows:

```
python keras_movie_classifier_1.py
```

```
import numpy
from keras.datasets import imdb
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers.embeddings import Embedding
from keras.preprocessing import sequence

# fix random seed for reproducibility
numpy.random.seed(7)

# load the dataset but only keep the top n words, zero the rest
top_words = 5000
(X_train, y_train), (X_test, y_test) = \
    imdb.load_data(nb_words=top_words)
# truncate and pad input sequences
max_review_length = 500
X_train = sequence.pad_sequences(X_train, maxlen=max_review_length)
X_test = sequence.pad_sequences(X_test, maxlen=max_review_length)

# create the model
embedding_vector_length = 32
model = Sequential()
model.add(Embedding(top_words, embedding_vector_length, \
                   input_length=max_review_length))
model.add(LSTM(100))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', \
               optimizer='adam', \
               metrics=['accuracy'])
print(model.summary())

model.fit(X_train, y_train, \
          validation_data=(X_test, y_test), \
          nb_epoch=3, batch_size=64)

# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)

print("Accuracy: %.2f%%" % (scores[1]*100))
```


Adding a convolutional layer

We can add one-dimensional CNN and max-pooling layers after the embedding layer, which will then feed the consolidated features to the LSTM.

Here is our embedding layer:

```
| model = Sequential()
| model.add(Embedding(top_words,
|                     embedding_vector_length,
|                     input_length=max_review_length))
```

We can apply a convolution layer with a small kernel filter (`filter_length`) of size `3`, with `32` output features (`nb_filter`):

```
| model.add(Conv1D(padding="same", activation="relu", kernel_size=3, num_filter=32))
```

Next, we add a pooling layer; the size of the region to which max pooling is applied is equal to `2`:

```
| model.add(GlobalMaxPooling1D())
```

The next layer is a `LSTM` layer, with `100` memory units:

```
| model.add(LSTM(100))
```

The final layer is a `Dense` output layer, with a single neuron and a `sigmoid` activation function, to make `0` or `1` predictions for the two classes (good and bad) in the problem (that is, binary classification problem):

```
| model.add(Dense(1, activation='sigmoid'))
```

Running this example provides the following output:

```
Epoch 1/3
16750/16750 [=====] - 58s - loss: 0.5186 - acc: 0.7263
Epoch 2/3
16750/16750 [=====] - 58s - loss: 0.2946 - acc: 0.8825
Epoch 3/3
16750/16750 [=====] - 58s - loss:
0.2291 - acc: 0.9126
Accuracy: 86.36%
```

The result obtained is a slight improvement on the accuracy of our model.

Source code for movie classifier with convolutional layer

The complete source code for the previous example is as follows:

```
import numpy
from keras.datasets import imdb
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers.embeddings import Embedding
from keras.preprocessing import sequence
from keras.layers import Conv1D, GlobalMaxPooling1D

# fix random seed for reproducibility
numpy.random.seed(7)

# load the dataset but only keep the top n words, zero the rest
top_words = 5000
(X_train, y_train), (X_test, y_test) = \
    imdb.load_data(num_words=top_words)

# truncate and pad input sequences
max_review_length = 500
X_train = sequence.pad_sequences(X_train, maxlen=max_review_length)
X_test = sequence.pad_sequences(X_test, maxlen=max_review_length)

# create the model
embedding_vector_length = 32
model = Sequential()
model.add(Embedding(top_words, embedding_vector_length, \
                   input_length=max_review_length))
model.add(Conv1D(padding="same", activation="relu", kernel_size=3, num_filter=32))
model.add(GlobalMaxPooling1D ())

model.add(LSTM(100))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', \
               optimizer='adam', \
               metrics=['accuracy'])
print(model.summary())

model.fit(X_train, y_train, \
          validation_data=(X_test, y_test), \
          nb_epoch=3, batch_size=64)

# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)

print("Accuracy: %.2f%%" % (scores[1]*100))
```


Pretty Tensor

Pretty Tensor allows the developer to wrap TensorFlow operations, to quickly chain any number of layers to define neural networks.

The following is a simple example of the Pretty Tensor capabilities. We wrap a standard TensorFlow object, `pretty`, into a library compatible object, then we feed it through three fully connected layers, to finally output a softmax distribution:

```
pretty = tf.placeholder([None, 784], tf.float32)
softmax = (prettensor.wrap(examples)
    .fully_connected(256, tf.nn.relu)
    .fully_connected(128, tf.sigmoid)
    .fully_connected(64, tf.tanh)
    .softmax(10))
```

The Pretty Tensor installation is very simple; just use the `pip` installer:

```
| sudo pip install prettentor
```


Chaining layers

Pretty Tensor has three modes of operation, which share the ability to chain methods.

Normal mode

In normal mode, every time a method is called, a new Pretty Tensor is created. This allows for easy chaining, and yet you can still use any particular object multiple times. This makes it easy to branch your network.

Sequential mode

In sequential mode, an internal variable, the head, keeps track of the most recent output tensor, thus allowing for the breaking of call chains into multiple statements.

Here is a quick example:

```
seq = pretty_tensor.wrap(input_data).sequential()
seq.flatten()
seq.fully_connected(200, activation_fn=tf.nn.relu)
seq.fully_connected(10, activation_fn=None)
result = seq.softmax(labels, name=softmax_name)
```


Branch and join

Complex networks can be built using the first class methods of branch and join:

- Branch creates a separate Pretty Tensor object that points to the current head when it is called, and this allows the user to define a separate tower, which either ends in a regression target, output, or rejoins the network. Rejoining allows the user to define composite layers, such as inception.
- Join is used to join multiple inputs or to rejoin a composite layer.

Digit classifier

In this example, we'll define and train either a two-layer model or a convolutional model in the style of LeNet 5:

```
from six.moves import xrange
import tensorflow as tf
import prettyn tensor as pt
from prettyn tensor.tutorial import data_utils

tf.app.flags.DEFINE_string(
    'save_path', None, 'Where to save the model checkpoints.')
FLAGS = tf.app.flags.FLAGS

BATCH_SIZE = 50
EPOCH_SIZE = 60000 // BATCH_SIZE
TEST_SIZE = 10000 // BATCH_SIZE
```

Since we are feeding our data as numpy arrays, we need to create placeholders in the graph. These must then be fed using the feed dict.

```
image_placeholder = tf.placeholder\
    (tf.float32, [BATCH_SIZE, 28, 28, 1])
labels_placeholder = tf.placeholder\
    (tf.float32, [BATCH_SIZE, 10])

tf.app.flags.DEFINE_string('model', 'full',
    'Choose one of the models, either
        full or conv')

FLAGS = tf.app.flags.FLAGS
```

We created the following function, `multilayer_fully_connected`. The first two layers are fully connected (100 neurons), and the final layer is a softmax result layer. Note that the chaining layer is a very simple operation:

```
def multilayer_fully_connected(images, labels):
    images = pt.wrap(images)
    with pt.defaults_scope\
        (activation_fn=tf.nn.relu, l2loss=0.00001):

        return (images.flatten() .\
            fully_connected(100) .\
            fully_connected(100) .\
            softmax_classifier(10, labels))
```

In the following, we'll build a multilayer convolutional network; the architecture is similar to that defined in LeNet 5. Please change this to experiment with other architectures:

```
def lenet5(images, labels):
    images = pt.wrap(images)
    with pt.defaults_scope\
        (activation_fn=tf.nn.relu, l2loss=0.00001):

        return (images.conv2d(5, 20) .\
            max_pool(2, 2) .\
            conv2d(5, 50) .\
            max_pool(2, 2) .\
            flatten() .\
            fully_connected(500) .\
            softmax_classifier(10, labels))
```

Since we are feeding our data as numpy arrays, we need to create placeholders in the graph. These must then be fed using the feed dict:

```
def main(_=None):
    image_placeholder = tf.placeholder(
        (tf.float32, [BATCH_SIZE, 28, 28, 1]))
    labels_placeholder = tf.placeholder(
        (tf.float32, [BATCH_SIZE, 10]))
```

Depending on `FLAGS.model`, we may have a two-layer classifier or a convolutional classifier, previously defined:

```
def main(_=None):

    if FLAGS.model == 'full':
        result = multilayer_fully_connected(
            (image_placeholder, labels_placeholder))
    elif FLAGS.model == 'conv':
        result = lenet5(image_placeholder, labels_placeholder)
    else:
        raise ValueError(
            'model must be full or conv: %s' % FLAGS.model)
```

Then we define the `accuracy` function for the evaluated classifier:

```
accuracy = result.softmax.evaluate_classifier(
    (labels_placeholder, phase=pt.Phase.test))
```

Next, we build the training and test sets:

```
train_images, train_labels = data_utils.mnist(training=True)
test_images, test_labels = data_utils.mnist(training=False)
```

We will use a gradient descent optimizer procedure and apply it to the graph. The `pt.apply_optimizer` function adds regularization losses and sets up a step counter:

```
optimizer = tf.train.GradientDescentOptimizer(0.01) \
train_op = pt.apply_optimizer(
    (optimizer, losses=[result.loss]))
```

We can set `save_path` in the running session to automatically checkpoint every so often. Otherwise, at the end of the session, the model will be lost:

```
runner = pt.train.Runner(save_path=FLAGS.save_path)
with tf.Session():
    for epoch in xrange(10):
```

Shuffle the training data:

```
train_images, train_labels = \
    data_utils.permute_data(
        ((train_images, train_labels)))

runner.train_model(train_op, result.\
    loss, EPOCH_SIZE, \
    feed_vars=(image_placeholder, \
        labels_placeholder), \
    feed_data=pt.train.\
    feed_numpy(BATCH_SIZE, \
        train_images, \
        train_labels), \
```

```

        print_every=100)

classification_accuracy = runner.evaluate_model\
    (accuracy,\n     TEST_SIZE,\n     feed_vars=(image_placeholder,\n                 labels_placeholder), \n     feed_data=pt.train.\n     feed_numpy(BATCH_SIZE,\n                 test_images,\n                 test_labels))

print('epoch' , epoch + 1)
print('accuracy', classification_accuracy )

if __name__ == '__main__':
    tf.app.run()

```

Running this example provides the following output:

```

>>>
Extracting /tmp/data/train-images-idx3-ubyte.gz
Extracting tmp/data/train-labels-idx1-ubyte.gz
Extracting /tmp/data/t10k-images-idx3-ubyte.gz
Extracting /tmp/data/t10k-labels-idx1-ubyte.gz
epoch = 1
Accuracy [0.8994]
epoch = 2
Accuracy [0.91549999]
epoch = 3
Accuracy [0.92259997]
epoch = 4
Accuracy [0.92760003]
epoch = 5
Accuracy [0.9303]
epoch = 6
Accuracy [0.93870002]
epoch = 7
epoch = 8
Accuracy [0.94700003]
epoch = 9
Accuracy [0.94910002]
epoch = 10
Accuracy [0.94980001]

```


Source code for digit classifier

The following is the full source code for the digit classifier previously described:

```
from six.moves import xrange
import tensorflow as tf
import prettyn tensor as pt
from prettyn tensor.tutorial import data_utils

tf.app.flags.DEFINE_string('save_path', None, 'Where to save the model checkpoints.')
FLAGS = tf.app.flags.FLAGS

BATCH_SIZE = 50
EPOCH_SIZE = 60000 // BATCH_SIZE
TEST_SIZE = 10000 // BATCH_SIZE

image_placeholder = tf.placeholder\
    (tf.float32, [BATCH_SIZE, 28, 28, 1])
labels_placeholder = tf.placeholder\
    (tf.float32, [BATCH_SIZE, 10])

tf.app.flags.DEFINE_string('model', 'full', 'Choose one of the models, either full or conv')
FLAGS = tf.app.flags.FLAGS

def multilayer_fully_connected(images, labels):
    images = pt.wrap(images)
    with pt.defaults_scope(activation_fn=tf.nn.relu, l2loss=0.00001):
        return (images.flatten() .\
            fully_connected(100) .\
            fully_connected(100) .\
            softmax_classifier(10, labels))

def lenet5(images, labels):
    images = pt.wrap(images)
    with pt.defaults_scope\
        (activation_fn=tf.nn.relu, l2loss=0.00001):
        return (images.conv2d(5, 20) .\
            max_pool(2, 2) .\
            conv2d(5, 50) .\
            max_pool(2, 2) .\
            flatten() .\
            fully_connected(500) .\
            softmax_classifier(10, labels))

def main(_=None):
    image_placeholder = tf.placeholder\
        (tf.float32, [BATCH_SIZE, 28, 28, 1])
    labels_placeholder = tf.placeholder\
        (tf.float32, [BATCH_SIZE, 10])

    if FLAGS.model == 'full':
        result = multilayer_fully_connected\
            (image_placeholder, \
            labels_placeholder)
    elif FLAGS.model == 'conv':
        result = lenet5(image_placeholder, \
            labels_placeholder)
    else:
        raise ValueError\
            ('model must be full or conv: %s' % FLAGS.model)

    accuracy = result.softmax.\
        evaluate_classifier\
            (labels_placeholder, phase=pt.Phase.test)

    train_images, train_labels = data_utils.mnist(training=True)
    test_images, test_labels = data_utils.mnist(training=False)
    optimizer = tf.train.GradientDescentOptimizer(0.01)
    train_op = pt.apply_optimizer(optimizer, losses=[result.loss])
    runner = pt.train.Runner(save_path=FLAGS.save_path)
```

```
with tf.Session():
    for epoch in xrange(10):
        train_images, train_labels =\
            data_utils.permute_data\
            ((train_images, train_labels))

        runner.train_model(train_op,result.\
                           loss,EPOCH_SIZE,\.
                           feed_vars=(image_placeholder,\.
                                      labels_placeholder),\.
                           feed_data=pt.train.\.
                           feed_numpy(BATCH_SIZE,\.
                                      train_images,\.
                                      train_labels),\.
                           print_every=100)
        classification_accuracy = runner.evaluate_model\
            (accuracy,\.
             TEST_SIZE,\.
             feed_vars=(image_placeholder,\.
                        labels_placeholder),\.
             feed_data=pt.train.\.
             feed_numpy(BATCH_SIZE,\.
                        test_images,\.
                        test_labels))

        print('epoch' , epoch + 1)
        print('accuracy', classification_accuracy )

if __name__ == '__main__':
    tf.app.run()
```


TFLearn

TFLearn is a library that wraps a lot of new APIs by TensorFlow with the nice and familiar scikit-learn API.

TensorFlow is all about building and executing a graph. This is a very powerful concept, but it is also cumbersome to start with.

Looking under the hood of TFLearn, we used just three parts:

- **Layers**: This is a set of advanced TensorFlow functions that allows you to easily build complex graphs, from fully connected layers, convolution, and batch norm, to losses and optimization.
- **Graph actions**: This is a set of tools to perform training and evaluating, and run inference on TensorFlow graphs.
- **Estimator**: This packages everything in a class that follows the scikit-learn interface, and provides a way to easily build and train custom TensorFlow models. Subclasses of Estimator, such as linear classifier, linear regressor, DNN classifier, and so on, are pre-packaged models similar to scikit-learn logistic regression that can be used in one line.

TFLearn installation

To install TFLearn, the easiest way is to run the following:

```
| pip install git+https://github.com/tflearn/tflearn.git
```

For the latest stable version, use the following command:

```
| pip install tflearn
```

Alternatively, you can also install from source by running (from the source folder) the following:

```
| python setup.py install
```



A more details discussion on installation of TFLearn can be found at <https://github.com/tflearn/tflearn>.

Titanic survival predictor

In this tutorial, we will learn to use TFLearn and TensorFlow to model the survival chance of titanic passengers using their personal information (such as gender, age, and so on). To tackle this classic machine learning task, we are going to build a DNN classifier.

Let's take a look at the dataset (TFLearn will automatically download it for you).

For each passenger, the following information is provided:

```
survivedSurvived (0 = No; 1 = Yes)
pclass          Passenger Class (1 = st; 2 = nd; 3 = rd)
name  Name
sex   Sex
age   Age
sibsp  Number of Siblings/Spouses Aboard
parch  Number of Parents/Children Aboard
ticket Ticket Number
fare   Passenger Fare
```

Here are some samples extracted from the dataset:

survived	pclass	name	sex	age	sibsp	parch	ticket	fare
1	1	Aubart, Mme. Leontine Pauline	Female	24	0	0	PC 17477	69.3000
0	2	Bowenur, Mr. Solomon	Male	42	0	0	211535	13.0000
1	3	Baclini, Miss. Marie Catherine	Female	5	2	1	2666	19.2583
0	3	Youseff, Mr. Gerious	Male	45.5	0	0	2628	7.2250

There are two classes in our task: *not survived* (`class = 0`) and *survived* (`class = 1`), and the passenger data has eight features.

The `titanic` dataset is stored in a CSV file, so we can use the TFLearn `load_csv()` function to load the data from the file into a Python list. We specify the `target_column` argument to indicate that our labels (survived or not) are located in the first column (ID is `0`). The functions will return a tuple (`data, labels`).

Let's start by importing the numpy and TFLearn libraries:

```
| import numpy as np  
| import tflearn
```

Download the `titanic` dataset:

```
| from tflearn.datasets import titanic  
| titanic.download_dataset('titanic_dataset.csv')
```

Load the CSV file, indicating that the first column represents `labels`:

```
| from tflearn.data_utils import load_csv  
| data, labels = load_csv('titanic_dataset.csv', target_column=0,  
|                         categorical_labels=True, n_classes=2)
```

The data needs some preprocessing to be ready to be used in our DNN classifier. Indeed, we must delete the columns, fields that don't help in our analysis. We discard the name and ticket fields, because we estimate that passenger name and ticket are not correlated with their chance of surviving:

```
| def preprocess(data, columns_to_ignore):
```

The preprocessing phase starts by descending the `id` and delete columns:

```
| for id in sorted(columns_to_ignore, reverse=True):  
|     [r.pop(id) for r in data]  
| for i in range(len(data)):
```

The `sex` field is converted to `float` (to be manipulated):

```
|     data[i][1] = 1. if data[i][1] == 'female' else 0.  
| return np.array(data, dtype=np.float32)
```

As already described, the fields `name` and `ticket` will be ignored by the analysis:

```
| to_ignore=[1, 6]
```

Here, we call the `preprocess` procedure:

```
| data = preprocess(data, to_ignore)
```

First of all, we specify the shape of our input data. The input sample has a total of `6` features, and we will process samples per batch to save memory, so our data input shape is `[None, 6]`. The `None` parameter means an unknown dimension, so we can change the total number of samples that are processed in a batch:

```
| net = tflearn.input_data(shape=[None, 6])
```

Finally, we build a three-layer neural network with this simple sequence of statements:

```
net = tflearn.fully_connected(net, 32)
net = tflearn.fully_connected(net, 32)
net = tflearn.fully_connected(net, 2, activation='softmax')
net = tflearn.regression(net)
```

TFLearn provides a model wrapper DNN that can automatically perform neural network classifier tasks:

```
| model = tflearn.DNN(net)
```

We will run it for 10 epochs, with a batch size of 16:

```
| model.fit(data, labels, n_epoch=10, batch_size=16, show_metric=True)
```

Running the model, you should have an output as follows:

```
Training samples: 1309
Validation samples: 0
--
Training Step: 82 | total loss: 0.64003
| Adam | epoch: 001 | loss: 0.64003 - acc: 0.6620 -- iter: 1309/1309
--
Training Step: 164 | total loss: 0.61915
| Adam | epoch: 002 | loss: 0.61915 - acc: 0.6614 -- iter: 1309/1309
--
Training Step: 246 | total loss: 0.56067
| Adam | epoch: 003 | loss: 0.56067 - acc: 0.7171 -- iter: 1309/1309
--
Training Step: 328 | total loss: 0.51807
| Adam | epoch: 004 | loss: 0.51807 - acc: 0.7799 -- iter: 1309/1309
--
Training Step: 410 | total loss: 0.47475
| Adam | epoch: 005 | loss: 0.47475 - acc: 0.7962 -- iter: 1309/1309
--
Training Step: 492 | total loss: 0.51677
| Adam | epoch: 006 | loss: 0.51677 - acc: 0.7701 -- iter: 1309/1309
--
Training Step: 574 | total loss: 0.48988
| Adam | epoch: 007 | loss: 0.48988 - acc: 0.7891 -- iter: 1309/1309
--
Training Step: 656 | total loss: 0.55073
| Adam | epoch: 008 | loss: 0.55073 - acc: 0.7427 -- iter: 1309/1309
--
Training Step: 738 | total loss: 0.50242
| Adam | epoch: 009 | loss: 0.50242 - acc: 0.7854 -- iter: 1309/1309
--
Training Step: 820 | total loss: 0.41557
| Adam | epoch: 010 | loss: 0.41557 - acc: 0.8110 -- iter: 1309/1309
--
```

The model accuracy is around 81%, which means that it can predict the correct outcome (survived or not) for 81% of the total passengers.

Finally, evaluate the model to get the final accuracy:

```
| accuracy = model.evaluate(data, labels, batch_size=16)
print('Accuracy: ', accuracy)
```

The following is the output:

```
| Accuracy: [0.78456837289473591]
```


Source code for titanic classifier

The full code for the implemented classifier is as follows:

```
from tflearn.datasets import titanic
titanic.download_dataset('titanic_dataset.csv')
from tflearn.data_utils import load_csv
data, labels = load_csv('titanic_dataset.csv', target_column=0,
                       categorical_labels=True, n_classes=2)

def preprocess(data, columns_to_ignore):
    for id in sorted(columns_to_ignore, reverse=True):
        [r.pop(id) for r in data]
    for i in range(len(data)):
        data[i][1] = 1. if data[i][1] == 'female' else 0.
    return np.array(data, dtype=np.float32)

to_ignore=[1, 6]
data = preprocess(data, to_ignore)
net = tflearn.input_data(shape=[None, 6])

net = tflearn.fully_connected(net, 32)
net = tflearn.fully_connected(net, 32)
net = tflearn.fully_connected(net, 2, activation='softmax')
net = tflearn.regression(net)
model = tflearn.DNN(net)
model.fit(data, labels, n_epoch=10, batch_size=16, show_metric=True)

# Evaluate the model
accuracy = model.evaluate(data, labels, batch_size=16)
print('Accuracy: ', accuracy)
```


Summary

In this chapter, we discovered three TensorFlow-based libraries for deep learning research and development.

We gave an overview of Keras, which is designed for minimalism and modularity, allowing the user to quickly define deep learning models.

Using Keras, we have learned how to develop a simple single layer LSTM model for the IMDB movie review sentiment classification problem.

Then, we briefly introduced Pretty Tensor; it allows the developer to wrap TensorFlow operations to chain any number of layers.

We implemented a convolutional model in the style of LeNet to quickly resolve the handwritten classification model.

The final library we looked at was TFLearn; it wraps a lot of TensorFlow APIs. In the example application, we learned to use TFLearn to estimate the survival chance of titanic passengers. To tackle this task, we built a deep neural network classifier.

The next chapter introduces reinforcement learning. We'll explore the basic principles and algorithms. We'll also look at some example applications, using TensorFlow and the OpenAI Gym framework, which is a powerful toolkit for developing and comparing reinforcement learning algorithms.

Advanced Multimedia Programming with TensorFlow

In this chapter, we will discuss advanced multimedia programming using TensorFlow. Some emerging research problem like Deep Neural Networks for scalable object detection and deep learning on Android using TensorFlow with an example will be discussed.

The following topics will be discussed in this chapter:

- Introduction to multimedia analysis
- Deep learning for Scalable Object Detection
- Accelerated Linear Algebra
- TensorFlow and Keras
- Deep learning on Android

Introduction to multimedia analysis

Image and video analysis is the extraction of meaningful information and patterns from images or videos. A huge amount of images and videos are being generated every day and being able to analyze this huge amount of data has a great potential towards providing services that are based on this analysis.

In this chapter we will go through deep learning examples that cover image analysis as well as video analysis (just a demonstration of what video analysis and other complex deep learning examples will look like after the integration of TensorFlow and Keras) and see how can we get meaningful information out of them.

Deep learning for Scalable Object Detection

In this section, we will learn how to make image recognition using TensorFlow. Also, we will be using transfer learning which is the ability to use the network weights of a pre-trained model that was trained on large dataset like ImageNet), usually people use transfer learning when they have small datasets. So we will be starting with a pre-trained model and use it on another problem. We will retrain this model on a similar problem because if we started from scratch then it will take days.



Figure 1: Flowers dataset (Image by TesnorFlow, Source: <https://www.tensorflow.org/images/daisies.jpg>)

There are lots of pre-trained networks that you can start using and they already packed with TensorFlow. In this section, we will be using Inception V3 network which is trained for the ImageNet (<http://image-net.org/>). This network can differentiate between 1000 different classes.

Initially, you'll need a set of images to teach the network about the new classes you want to recognize. TensorFlow team have created an archive of creative-commons licensed flower photos to use initially.

Initially, you will need a dataset which is a bunch of images in order to teach the network about your new classes that you want them to be recognized. We will use `flower` dataset that is provided by TensorFlow.

You need to issue the following commands in order to download and set up the directories for the `flower` dataset:

```
# return to the home directory  
cd $HOME
```

```
mkdir tensorflow_files  
cd tensorflow_files  
curl -O http://download.tensorflow.org/example_images/flower_photos.tgz  
tar xzf flower_photos.tgz
```

After downloading this `flower` dataset which is about 218MiB, you should now have a copy of the flower photos available in your home directory.

Next, you need to clone the TensorFlow repository by issuing the following command:

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

After cloning TensorFlow, you will have a trainer and we have data, so let's start the training process by using the Inception V3 network.

As we mentioned in the beginning that Inception V3 network is trained initially on ImageNet and it can differentiate between 1000 different classes and hence it comes with millions of parameters so we will only train the final layer of Inception V3 network and it will not take much time to accomplish this task.

So let's start the retraining process, by issuing the following command from your root directory of TensorFlow:

```
#From the root directory of TensorFlow  
python tensorflow/examples/image_retraining/retrain.py \  
--bottleneck_dir=~/tensorflow_files/bottlenecks \  
--model_dir=~/tensorflow_files/inception \  
--output_graph=~/tensorflow_files/retrained_graph.pb \  
--output_labels=~/tensorflow_files/retrained_labels.txt \  
--image_dir ~/tensorflow_files/flower_photos
```

So the script will load the pre-trained Inception v3 mode, removes the old final layer and finally trains a new model on the flower photo that we have provided.

Inception V3 network was not originally trained on any of these flower species, but the kinds of information that makes it plausible for Inception V3 to differentiate between 1000 different classes are also useful to discriminate between other objects. By using any pre-trained network, we are using that information as input to our final classification layer that distinguishes our flower classes.

Bottlenecks

You can see in the previous command we mentioned something called bottleneck, so what is a bottleneck?

So the first step that will be taken in order to get our retrained model is to analyse all the images that we have provided and calculate the bottleneck values for each one of them.

As we mentioned that we are using Inception V3 as a pre-trained model. Inception V3 has many layers that are lying on top of each other. These layers are already pre-trained and have information that can be used to discriminate between the images. We are only training the last fully-connected layer (the one just before the output layer) and all the other previous layers will retain their trained state.

So what's a bottleneck? It is a term that TensorFlow uses to refer to the layer just before the final layer that is actually responsible for the classification. So any image in the training set is used multiple times during the training and calculating the layers behind the bottleneck takes large time for each image. So by caching the outputs of these lower layers on the disk we are avoiding wasting a significant amount of time. By default, the bottleneck is stored in `/tmp/bottleneck` directory.

Using the retrained model

The retraining script that we just used will write out a custom version of Inception V3 network with the final layer retrained to our `flower` dataset to `tensorflow_files/output_graph.pb` and you will find a text file that contains the labels in `tensorflow_files/output_labels.txt`.

The following are the steps to classifying an image:

1. We are going to write a Python script that can use the previous network and use it to classify an image.

```
| import tensorflow as tf, sys
```

2. You will be sending the image to be classified as a parameter:

```
| provided_image_path = sys.argv[1]
```

3. Then we will read the image data:

```
| provided_image_data = tf.gfile.FastGFile(provided_image_path,
| 'rb').read()
```

4. Load labels file:

```
label_lines = [line.rstrip() for line
               in
               tf.gfile.GFile("tensorflow_files/retrained_labels.txt")]

# Unpersists graph from file
with tf.gfile.FastGFile("tensorflow_files/retrained_graph.pb",
                       'rb') as f:
    graph_def = tf.GraphDef()
    graph_def.ParseFromString(f.read())
    _ = tf.import_graph_def(graph_def, name='')

with tf.Session() as sess:
    # pass the provided_image_data as input to the graph
    softmax_tensor =
        sess.graph.get_tensor_by_name('final_result:0')

    network_predictions = sess.run(softmax_tensor, \
                                   {'DecodeJpeg/contents:0': provided_image_data})

    # Sort the result by confidence to show the flower labels
    # accordingly
    top_predictions = network_predictions[0].argsort()[-len(network_predictions[0]):][::-1]

    for prediction in top_predictions:
        flower_type = label_lines[prediction]
        score = network_predictions[0][prediction]
        print('%s (score = %.5f)' % (flower_type, score))
```

You can create the previous Python script into your `tensorflow_files` directory. Let's call it `classify_image.py`.

Then you can issue the following command from your `tensorflow_files` directory in order to classify a

daisy photo:

```
| python classify_image.py flower_photos/daisy/21652746_cc379e0eea_m.jpg
```

You might get an output like the following:

```
daisy (score = 0.99071)
sunflowers (score = 0.00595)
dandelion (score = 0.00252)
roses (score = 0.00049)
tulips (score = 0.00032)
```

Note that the output scores that you will get could be slightly different from this because if you train the same algorithm multiple times on the same data, it can give slightly different results.

Accelerated Linear Algebra

The **Accelerated Linear Algebra (XLA)** is a domain specific compiler developed by TensorFlow for optimizing its computations. By this, you will get improvements in speed, memory usage and even portability on mobile platforms.

Initially, you won't see much benefit from XLA because it's still experimental but you can try it by using the just-in-time compilation or ahead-of-time compilations.

First, we are going to briefly mention the key strengths of TensorFlow and see how TensorFlow team had the challenge to keep and increase these key strengths.

Key strengths of TensorFlow

The following are the key strengths of TensorFlow:

- **Flexible:** This Flexibility is coming from TensorFlow's nature of being interpreted. Also, you can see it in the name that it uses a data flow programming mode. The way TensorFlow works, you give the program a graph then it tries to find a node in that graph that's ready to execute, subsequently, it fetches this node and runs it. After this sequence of operations, another set of nodes in the graph become ready to be executed. So TensorFlow does the same with them, fetches them and then executes them. This process of getting the nodes that are ready to be executed and running them is called the **Interpreter Loop**.
- **Expressive:** TensorFlow is dynamic as it has some pythonic relations. So you have your full expressiveness and freedom to define your own graph and you are not constrained to anything that can limit you from doing so. Also, TensorFlow is stateful it's similar to the way you program in other environments as you get to use variables.
- **Extensible:** One of the great advantages of TensorFlow is that it's blackbox modular. So you can come up with totally new data flow operations and attach it to TensorFlow without having the integration headache.

So after mentioning three of the key strengths of TensorFlow, how can TensorFlow team keep these strengths and add more speed by using the **just-in-time (JIT)** compilation via XLA?

Just-in-time compilation via XLA

The way TensorFlow is going to increase the speed of its programs and incorporate more devices that can run TensorFlow is with this JIT compilation via XLA.

The way XLA is working is summarized in the following figure:

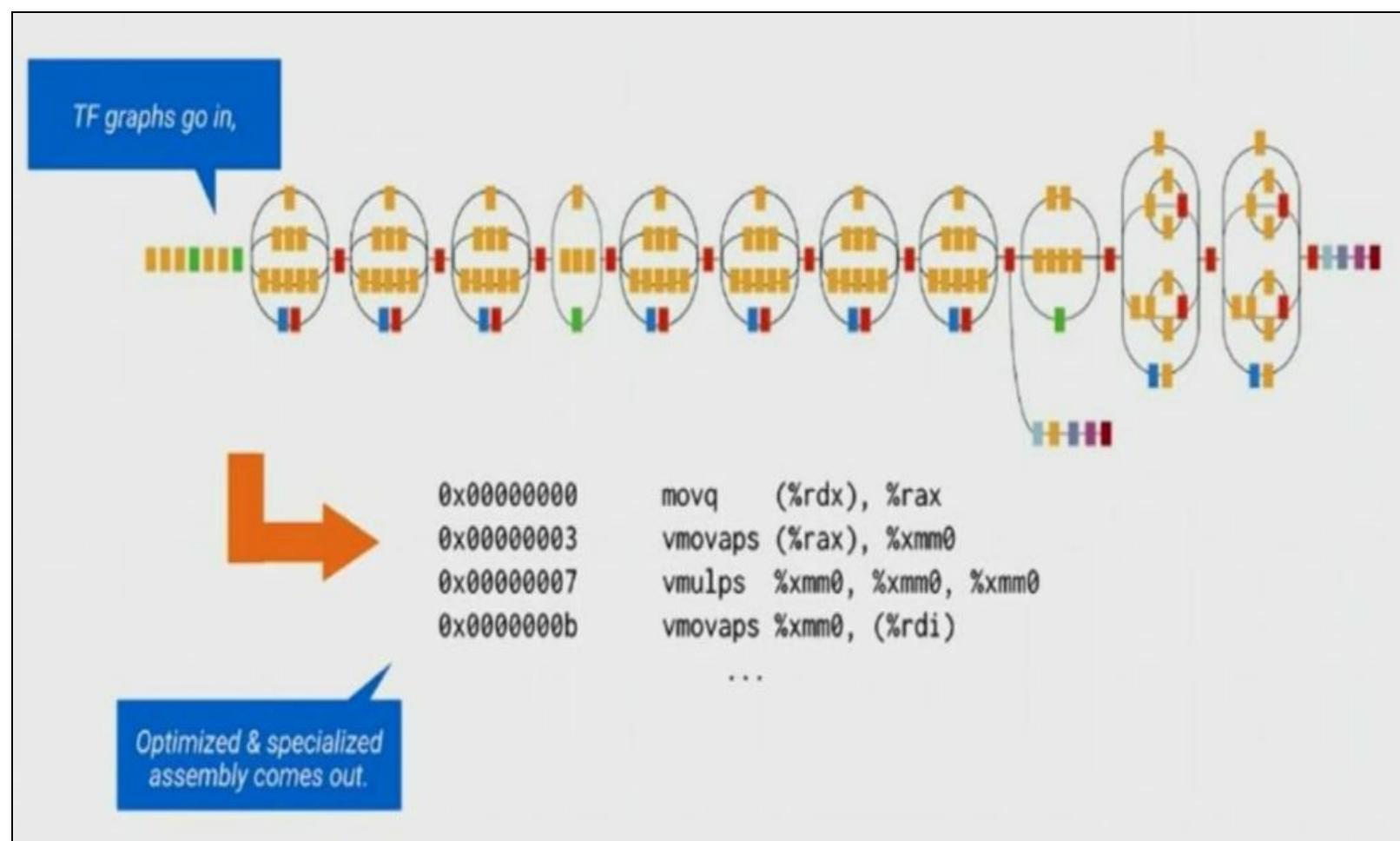


Figure 2: XLA life cycle (Image by TensorFlow, Source: http://img.ctolib.com/uploadImg/20170307/20170307064228_586.png)

TensorFlow team has developed this compiler infrastructure such that you can hand it a TensorFlow graph and then an optimized and specialized assembly comes out.

This is the compiler infrastructure that TensorFlow team have developed such that it can take a TensorFlow graph and spit out optimized and specialized assembly for that graph. This is a great feature added by the TensorFlow team as it enables you to produce compiled code that's not architecture specific rather it will be optimized and specialized for the underlying architecture that you are using.

In order to get XLA working on your machine, you need to manually specify this while configuring TensorFlow.

In order to clone the most update repository of TensorFlow, issue the following command:

```
| $ git clone https://github.com/tensorflow/tensorflow
```

Then you need to configure TensorFlow by issuing the following commands:

```
| $ cd tensorflow  
| $ ./configure
```

While configuring, you will be asked to enable XLA and you should answer yes to this question in order to be able to use XLA in the next example.

To show how XLA is working, let's demonstrate this by using the TensorFlow shell.

You need to open a TensorFlow shell in order to run the following code snippet. First, you can choose the paste mode by issuing the following command:

```
| %cpaste
```

Next, you can paste the following example:

```
| with tf.Session() as sess:  
|     x = tf.placeholder(tf.float32, [4])  
|     with tf.device("device:XLA_CPU:0"):  
|         y = x*x  
|     result = sess.run(y,{x:[1.5,0.5,-0.5,-1.5]})
```

You need to pass the following parameter to TensorFlow shell:

```
| --xla_dump_assembly=true
```

Here we are passing this flag to say spit out the XLA assembly that that's produced.

The output of the previous code will be the following:

```
| 0x00000000      movq    (%rdx), %rax  
| 0x00000003      vmovaps (%rax), %xmm0  
| 0x00000007      vmulps  %xmm0, %xmm0, %xmm0  
| 0x0000000b      vmovaps  %xmm0, (%rdi)  
| 0x0000000f      retq
```

Now we are going to elaborate more on this example in order to understand the code snippet and the assembly output.

The previous example is just taking four floating point numbers and multiplying them. What's special about this example is that we are putting it explicitly onto the XLA CPU device so the compiler is exposed as a device in one mode inside of TensorFlow.

After running the previous code snippet you will see just a couple of assembly instructions get emitted and the special thing about these instructions is that there are no loops because XLA knew that you are only going to multiply four numbers. So the emitted assembly instructions is specialized and optimized for the graph or the program that you fed in with your TensorFlow expression.

Also, you put the previous code snippet explicitly onto the XLA GPU device, but we are not going to cover this here. So as we mentioned that XLA can work for CPU and GPU in a standard TensorFlow shell.

JIT compilation

So we mentioned just-in-time compilation, so what's it all about?

The main thing about JIT compilation is that the programs you get the program to be built at runtime so when you type a TensorFlow expression and press enter, you don't want to sit for a while to figure out how to compile so you will also get low-overhead compilation. As you saw in the previous example that you go the assembly instructions generated just by pressing *Enter*.

Another advantage of JIT is that you can bind variables later in the code.

For example, you don't have to specify the batch size at the very beginning of your code. You want to able to specify the batch size right when you find out what you want it to be.

So if we looked at the basic TensorFlow level block diagram, we can see the TensorFlow core and XLA comes into the TensorFlow ecosystem in the bottom right.

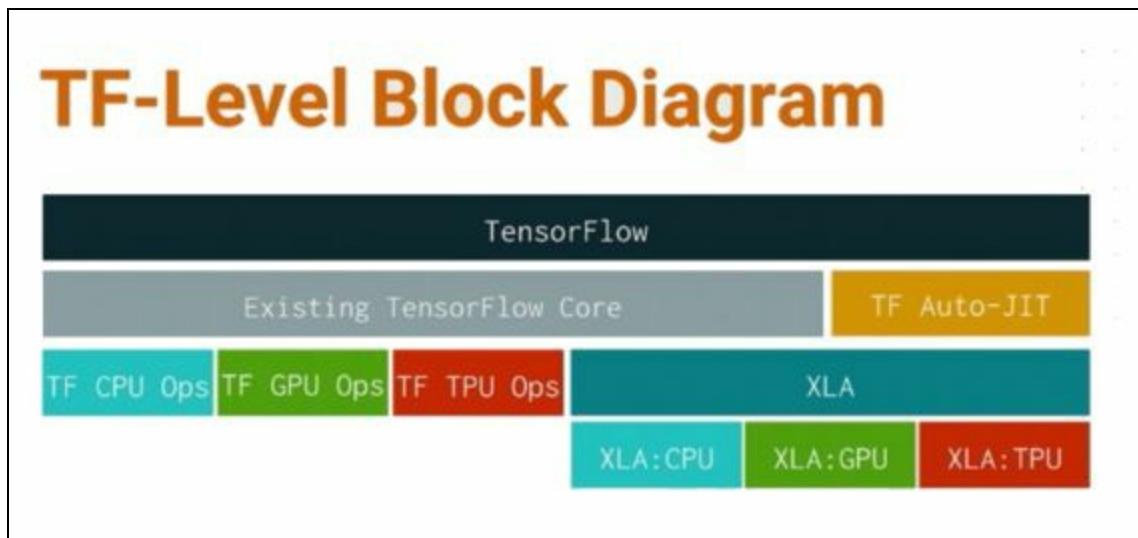


Figure 3: TensorFlow level block diagram (Source: TensorFlow Talk)

Existence and advantages of XLA

Next we will explain why TensorFlow enthusiasts are so excited about this new feature of just-in-time compilation via XLA.

The following are the key strengths for this feature:

- **Server side speedups:** Through this JIT compilation and specialization that we have mentioned, we see that some in-house models that TensorFlow team has won up to 60%. Also, SyntaxNet gets latency reductions from around 200 microseconds to 5 microseconds. And the reason for this was SyntaxNet had a lot of small operations in its graph so the interpreter has to go grab each little operation and this process of going and grabbing these small operations and running those causes some latency overhead but by compiling you're actually able to eliminate all that latency overhead away.
- **Improve memory usage:** By eliminating many intermediate storage buffers, TensorFlow managed to improve the memory usage and hence you can target more architecture like mobile architecture that has limited capabilities.
- **Mobile footprint reductions:** With XLA's ahead-of-time compilation you can go through a build process to turn models into executables if you want to do that from the beginning. This executable you can run it on the command line and it's able to eliminate much of TensorFlow runtime and by this; you will get to reduce the binary size of your program. So TensorFlow team has tried this feature on an **long short-term memory (LSTM)** model for mobile and they were able to reduce the binary size from 2.6MiB to less than 600KiB which means 4 times reduction in the deployment footprint and that is because of using XLA and following some best practices for writing TensorFlow code (https://www.tensorflow.org/performance/performance_guide).
- **Whole program analysis made easy:** The general thing that is exciting about this XLA approach is that analyzing a whole graph or program is made easy by this compiler infrastructure so we have this thing that's called XLA's high-level optimizer that able to look at a linear algebra level graph and create a reusable toolkit of global optimizations to work on it. So even though you compile for different platforms CPU, GPU or other devices, TensorFlow has parameterized out the things that are specific to a given platform at this high-level optimization toolkit level.

Under the hood working of XLA

So XLA takes an input language called HLO IR or just **High-Level Optimizer (HLO)**. So XLA takes a graph defined in this HLO and then compiles it into machine instructions for different architectures.

The following diagram shows the compilation process in XLA:

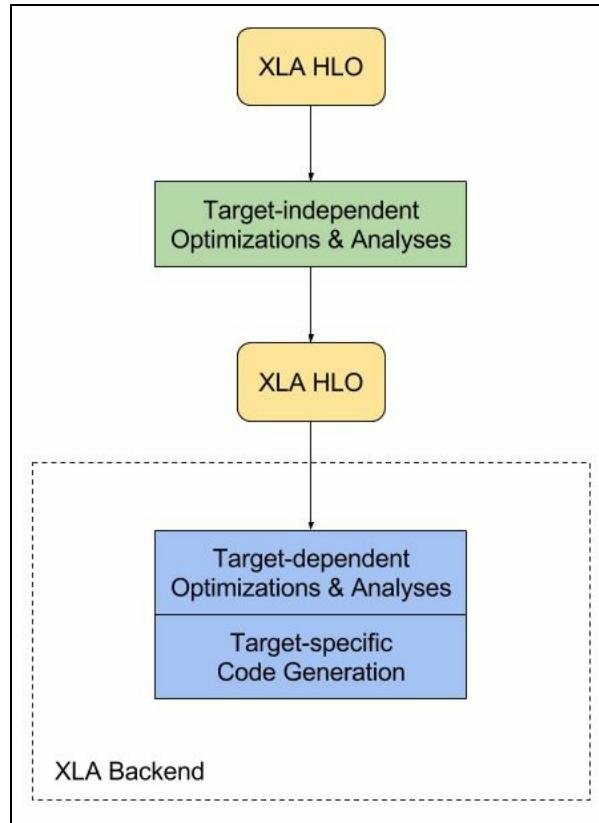


Figure 4: XLA compilation process (Source: <https://www.tensorflow.org/images/how-does-xla-work.png>)

As we discussed earlier, that TensorFlow uses HLO in order to provide target-independent code, so in this step TensorFlow is only trying to optimize your program without any target constraints, then TensorFlow uses another HLO in order to emit target dependent optimized and analyzed code that will be finally fed to XLA backend for target-specific code generation.

Still experimental

When writing this book status of TensorFlow is that not all of the TensorFlow operations compile as it's still the early days for this JIT compilation via XLA. With the growth of TensorFlow community we are expecting this feature to be fully functional and supported soon.

Supported platforms

Currently, XLA supports JIT compilation on x86-64 and NVIDIA GPUs, and also supports AOT compilation for x86-64 and ARM.

More experimental material

For more experimental material about XLA JIT compilation and how to turn it on during your session, refer to the following URL: <https://www.tensorflow.org/performance/xla/jit> by TensorFlow.

TensorFlow and Keras

In this section we are going to address a very important feature for all data scientists and machine learning enthusiasts which is the integration of TensorFlow and Keras. Having this feature on board, you will be able to build a very complex deep learning systems with very few lines of code.

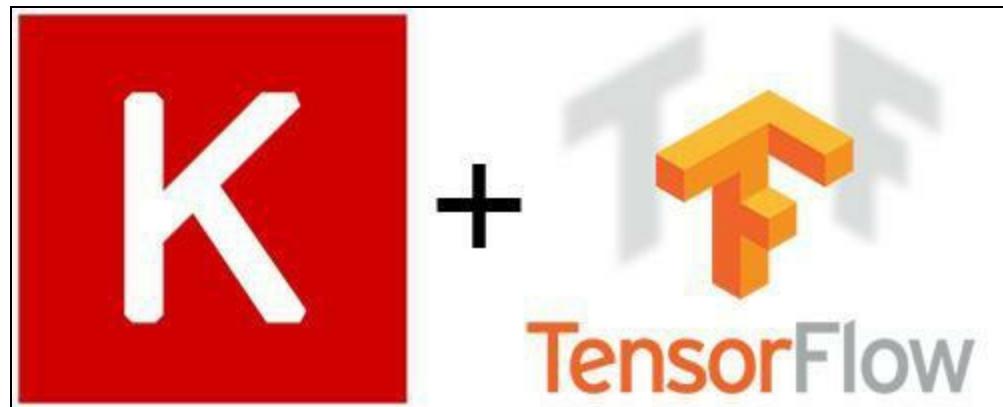


Figure 5: TensorFlow and Keras Integration (Source: <https://blog.keras.io/img/keras-tensorflow-logo.jpg>)

What is Keras?

Keras is an API that makes using and building deep learning models easier and faster. So it's a deep learning toolbox that's all about:

- Ease of use
- Reducing complexity
- Reducing cognitive load

And by making deep learning easier to use what happens is that you are making it accessible to more people. So the key design concept behind Keras is that to put deep learning into the hands of everyone.

So Keras is more like an API that has several different implementations. There's the Theano implementation which was originally released with Keras and you also have the TensorFlow implementations, and more will be coming in the future.

What TensorFlow has done is bringing the Keras API into the TensorFlow project and it's a great to combine the advantages of both TensorFlow and Keras and making deep learning accessible to everyone.

Effects of having Keras on board

As we said TensorFlow had brought the Keras API into the TensorFlow project and by doing so you get the following:

1. Keras compatibility `tf.keras` module is introduced in TensorFlow as an implementation of the Keras specifications that's built from the ground up for TensorFlow
2. Also, they are introducing into core TensorFlow new data structures like layers
3. Also, models which are containers for the graph of layers used to be Keras data structures and now they will be shared between core TensorFlow and this `tf.keras` module
4. Finally, it's fully compatible with all the TensorFlow advanced features like the experiment API.

So if it happens that you are a Keras a TensorFlow user, how will you get affected by such a move?

By integrating Keras directly into the core of TensorFlow, you are getting the following advantages as a Keras user:

- You can easily mix and match pure TensorFlow and Keras functionality.
- Also, you gain access to more advanced and powerful features like the following:
 - Distributed training
 - Cloud ML
 - Hyperparameter tuning
 - TF-serving to deploy your TensorFlow models for production

And of course, you are a TensorFlow user, so the following are the key advantages:

1. It gives you access to the full scope of the Keras API to make you development life easier and that's without having to leave your existing TensorFlow workflow.
2. You start using the Keras API without any loss of flexibility as you don't have to be familiar or to use all of the Keras, you can just adopt the layers you need.
3. Also, it gives you access to all the existing open source code of Keras. So you can just go and grab any piece of Keras code, drop it into your code base and by just changing the imports it will work right away.

So to make things clear and exciting, we are going to go through an example of what your workflow will look like when using Keras with TensorFlow:

Video question answering system

The following example will be focusing on building a video question answering model, and we will be using Keras to define the model.

In order to solve this problem, we will retrain it using high-level TensorFlow training in a distributed setting.



Figure 6: Video Question Answering

As we can see that we have videos which are sampled 4 frames per second and it's roughly 10 seconds per video so we have about 40 frames total per video. And we are asking questions about the video contents, just like the ones that are shown in figure 6.

So we are going to build a deep learning model that will take as an input:

- **Video:** Which will be represented as a sequence of frames, so it will be about 40 frames in order.
- **Question:** A sequence of words asking about the video contents.

The model will output an answer to this question.

This is very interesting and challenging problem, because if you tried to only take one frame and train CNN over them you will only model the visual information of this frame which might not be representative for the whole video. By using the entire video frames or even a sample, you will get to model these frames and understand the context by combining the different source of information which is the frames. So you are helping deep learning model to be able to leverage this order to correctly answer the correct question.

This kind of problem was very difficult a few years ago and it wasn't accessible to lots of researchers but with TensorFlow as a platform and Keras as an API, this problem's solution is accessible to anyone with just basic python scripting capabilities.

So the following is the model that we are going to follow and explain in details:

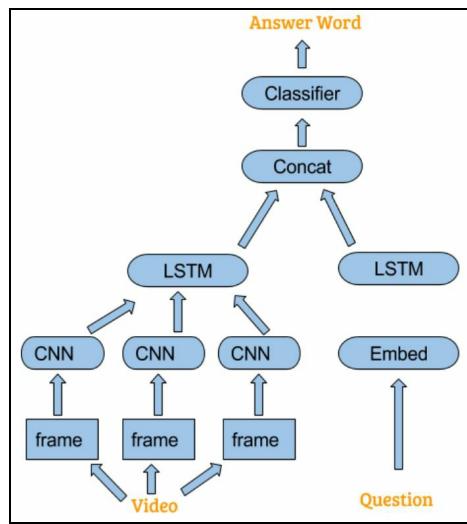


Figure 7: General video question answering architecture

At the high level, we have two main branches in this proposed architecture. The first branch is meant to encode the video frames into a single vector and the other branch is meant to encode the question, which is a sequence of words, into a single vector. So we have one vector encoding information about the entire video and another one for encoding information about the entire question, then we concatenate these two vectors to give us one single vector that encodes information about the entire problem.

What is really interesting about this deep learning architecture is that you are taking video as an input and also we are taking the semantic meaning that's represented in the question. Then we are representing this video and semantic meaning into the geometric space by turning them into vectors and then deep learning gets to learn interesting transformations of this geometric space. Then we will take this vector for concatenates the encoding for both the video and the question and pass it to a fully connected network which will end up with a softmax over a predefined vocabulary and then we pick the word has the highest probability in this vocabulary to be the answer to the question.

The following is more elaboration about the proposed infrastructure:

So for the video branch, we are starting out with the video as a sequence of frames and each frame is just an RGB image. Then we pass each frame to a CNN to transform it into a single vector and we are using the pre-trained network as a base for CNN. So After passing all the video frames through a series of CNNs, we will get the video encoded as a sequence of vectors. Then you take this sequence of vectors and run it through LSTM (a recurrent type of network that can process sequences and takes order into account) which will output a single vector to represent the video.

For the question we are following a very similar process, we represent the question as a sequence of integers in which each integer standing for a word and then we map each word to a word vector via the embedding process. So we get a sequence of vectors out of this sequence of words. Finally, we run it through a different LSTM that will encode the entire question as a single vector.

So let's see the representation of the previous architecture in Keras:

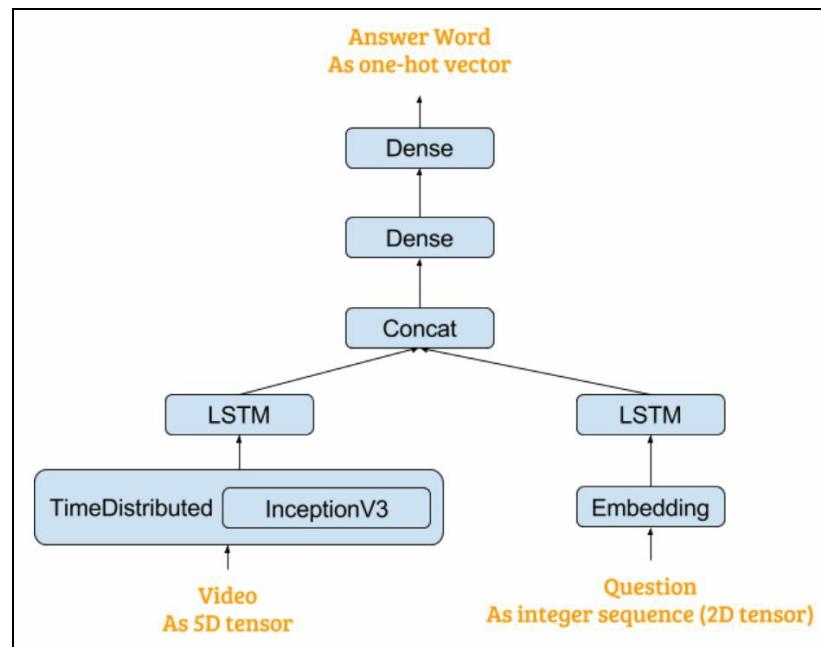


Figure 8: Keras video question answering architecture

The Keras architecture is very similar to the previous one. So for the video encoder, we are starting by representing the video as a 5D tensor, the first dimension/axis will be the batch axis, then you have the time axis and finally, you have a 3D tensor which encodes the frame.

So we will apply InceptionV3 network, which is pre-trained on ImageNet, to every frame of the 5D tensor to extract one vector per frame. So out of this, we will get a sequence of feature vectors which will be fed to an LSTM layer to produce a single vector for the video.

For the question part, we are simply using an embedding layer to map our question and we will also run this through an LSTM to produce a single vector.

At the top, we are using a concat operation to bring these two vectors together and then we stack a bunch of dense layers and finally, end up with a softmax over a predefined vocabulary and we are also training that with a target answer word encoded as one hot vector.

So what does the implementation look like?

```
video = tf.keras.layers.Input(shape=(None, 150, 150, 3))
cnn = tk.keras.applications.InceptionV3(weights='imagenet',
                                         include_top=False,
                                         pool='avg')
cnn.trainable = False
encodedframes = tk.keras.TimeDistributed(cnn)(video)
encoded_vid = tf.layers.LSTM(256)(encoded_frames)
```

The previous code snippet represents the video encoding in just 5 lines of code.

During the first line, we are just specifying the shape of the video input. So it is a 5D tensor with the shape argument and you do not actually explicitly mention the batch size. The first axis which is set to

none is the time axis and it is set to none because we want to be variable to able to encode videos with a different number of frames. Then the shape Argument is just 150 by 150 RGB image.

The second line, we are instantiating an Inception V3 network that will automatically load pre-trained weights (was trained no ImageNet) and we configure this network to work as a feature extraction layer. So we will not by including the classifier part of Inception V3 network because we only want the convolutional base. Finally, we apply average pooling on top of the bottleneck layer. The output of this line will be a single vector per image/frame.

Maybe someone will ask, we are using a pre-trained InceptionV3 model? And the reason for that is we are dealing with a small dataset and this dataset will not have enough data to allow you to learn to extract interesting visual features.

So in order to get this network to actually work well, you really need to be leveraging these pre-trained weights.

In the third line, we are setting the CNN to be non-trainable which means that during training we will not be updating the weights because it's pre-trained model and if we updated its weights while training on this new problem of question answering then we will likely be destroying the representations that this model has already learned on ImageNet.

In the fourth line, we use a time distributed layer to essentially take this CNN and apply it to every step of the time axis of the video. And what comes out of this is just a 3D tensor representing a sequence of visual vectors extracted from the frames.

Finally, in the fifth line, we run this sequence tensor through an LSTM layer and this gives us one single vector encoding of the entire video.

As you can notice, while instantiating the Keras LSTM layer, you only need to specify one parameter which is the number of units in the LSTM layer. So you do not have to go into the complex details of LSTMs. And one principle of Keras is that best practices are included, so every keras layer has a well-optimized default configuration that takes into account all these best practises. So you can rely on keras defaults to be a good one.

For the question part, we are going to encode the question in the following three lines of code:

```
question = tk.keras.layers.Input(shape=(100), dtype='int32')
x = tf.keras.layers.Embedding(10000, 256, mask_zero=True)(question)
encoded_q = tf.keras.layers.LSTM(128)(x)
```

In the first line, we specify the input tensor of the question. So every question will be just a sequence of 100 integers and a result we will be limited to questions that are at most 100 words long.

In the second line, we embed every integer into word vector via the embedding layer and we are masking this embedding layer which means that, if the question is not 100 words long, then the embedding layer will pad the rest with zeros to get to 100.

In the third line, we are propagating this to the LSTM layer to encode this set of word vectors into one vector.

And finally this is how you end up with the answer word:

```
x = tk.keras.layers.concatenate([encoded_vid, encoded_q])
x = tf.keras.layers.Dense(128, activation=tf.nn.relu)(x)
outputs = tf.keras.layers.Dense(1000)(x)
```

So for the first line, you are taking the video vector and the question vector and concatenating them with just a concat operation and finally, you are applying a couple of dense layers. And we'll end up with 1000 units. So we will have a vocabulary that is just 1000 different words.

And here's the step at which you are specifying the training configuration:

```
model = tk.keras.models.Model(), outputs)
model.compile(optimizer=tf.AdamOptimizer(),
              loss=tf.softmax_crossentropy_with_logits)
```

So you are just instantiating a model which is a container for a graph of layers, and you're instantiating them by just specifying what are the inputs of the model, what are the outputs and you are telling the model that it should use `AdamOptimizer` during training and use this loss softmax cross entropy with logits

You can notice that when we specified our classification layer with 1000 units, we did not specify any activation so it's actually a purely linear layer. The softmax activation will be included with the loss

To sum up, this is the entire code which is about 15 lines. So it's very, very short. So we are essentially turning this very complex architecture including loading pre-trained weights into just a few lines of code.

```
video = tf.keras.layers.Input(shape=(None, 150, 150, 3))
cnn = tk.keras.applications.InceptionV3(weights='imagenet',
                                         include_top=False,
                                         pool='avg')

cnn.trainable = False
encodedframes = tk.keras.TimeDistributed(cnn)(video)
encoded_vid = tf.layers.LSTM(256)(encoded_frames)

question = tk.keras.layers.Input(shape=(100), dtype='int32')
x = tf.keras.layers.Embedding(10000, 256, mask_zero=True)(question)
encoded_q = tf.keras.layers.LSTM(128)(x)

x = tk.keras.layers.concatenate([encoded_vid, encoded_q])
x = tf.keras.layers.Dense(128, activation=tf.nn.relu)(x)
outputs = tf.keras.layers.Dense(1000)(x)

model = tk.keras.models.Model(), outputs)
model.compile(optimizer=tf.AdamOptimizer(),
              loss=tf.softmax_crossentropy_with_logits)
```

As we mentioned that because this implementation of Keras is built from the ground up for TensorFlow so it is fully compatible with things like estimators and experiments so in just one line,

you can instantiate a TensorFlow experiment and this gives you access to the distributed training to train on Could ML and so on.

So you can start running your experiment as you did with your question answering model reading your video data, question and answer data from pandas data frame and you can start running it on in a cluster of GPUs in just a few lines.

```
train_panda_dataframe = pandas.read_hdf(...)

train_inputs = tf.inputs.pandas_input_fn(
    train_panda_dataframe,
    batch_size=32,
    shuffle=True,
    target_column='answer')

eval_inputs = tf.inputs.pandas_input_fn(...)

exp = tf.training.Experiment(
    model,
    train_input_fn=train_inputs,
    eval_input_fn=eval_inputs)

exp.run(...)
```


Not runnable code!

All the previous that we just mentioned is not runnable at the moment but these modules (`tf.keras`) will be available soon and you will be able to do video analysis in just 15 lines of code.

You can check the availability of `tf.keras` and `tf.contrib` by referring to Spring 2017 roadmap of Keras (<https://github.com/fchollet/keras/issues/5299>)

Deep learning on Android

One will ask, why do we want to do that? Is not deep learning and TensorFlow something that you do on massive clusters of GPU machines in big data centres? The answer to this question is obviously yes, but it's also an amazing way of offering unique user experiences that you have never been able to offer it to people before and that is by running interactively on mobile devices.

Mobile applications have lots of potential because people use it on daily basis. You can offer dozens of apps the depend on deep learning from real time translation, to predicting next words on keyboards, to help people scan their old photos with photo scan, to all the amazing stuff that you know SnapChat is doing in real time. You can also develop medical mobile applications that can help diagnosing diseases and more.

At the moment not all devices are supported by TensorFlow but we are going to only cover the Android part of it. TensorFlow at the moment supports:

- Android
- IOS
- Raspberry PI

But with the growing community of TensorFlow, you can expect the more is coming.

TensorFlow demo examples

The Android examples that are included in TensorFlow are designed to help you think about some of the different things that you can actually do using TensorFlow. So if you are interested in more ideas to improve your apps, then they are a good place to start with.

The demos provided by TensorFlow are designed to give straightforward samples of using TensorFlow in mobile applications. The following are the current samples that you can directly use on your Android device:

- **TF Classify:** This is the classification example and what this is doing is it taking a live camera and tries to label images using the InceptionV3 network. This sample works well for the objects that are known in the ImageNet (<http://www.image-net.org/>). One key thing about this is that you can actually use this to train your own model using Inception V3 on images that you actually care about and then you can take that model and drop it directly into the sample, and have your own custom image recognition app with very little coding involved. You might notice with default ImageNet model if you point it at people it will come up with results that are confusing because there is no person label in the ImageNet class set. So it will show you things that it often sees together with people like seat belts. But if you want to recognize people then you need to use the following sample.



Figure 9: Sample result from running TF Classify on Android(source: https://github.com/tensorflow/tensorflow/blob/master/tensorflow/examples/android/sample_images/classify1.jpg)

- **TF Detect:** What this is doing is drawing bounding boxes around any people that actually get detected in the live camera view. Tracking is used to give fast frame rate and identify objects frame-to-frame.

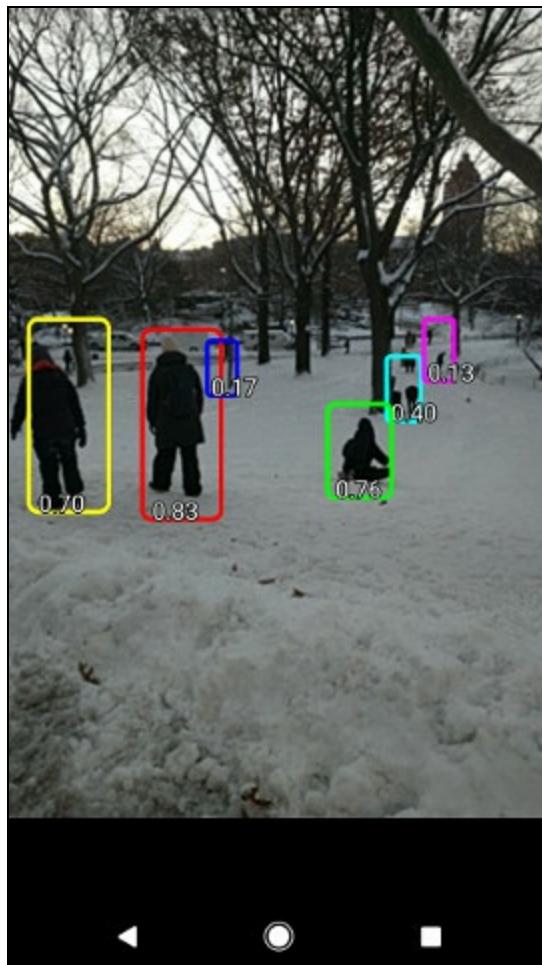


Figure 10: Sample result from running TF Detect on Android(source: https://github.com/tensorflow/tensorflow/blob/master/tensorflow/examples/android/sample_images/detect1.jpg)

- **TF Stylize:** The Magenta Group is about stylization and style transfer. You can actually run this live on the phone using the stylized example and you can pick and mix different style using the sliders. Because it's part of the magenta model set you can just grab the model directly from there and train it on any styles that you want to do Training at https://github.com/tensorflow/magenta/tree/master/magenta/models/image_stylization

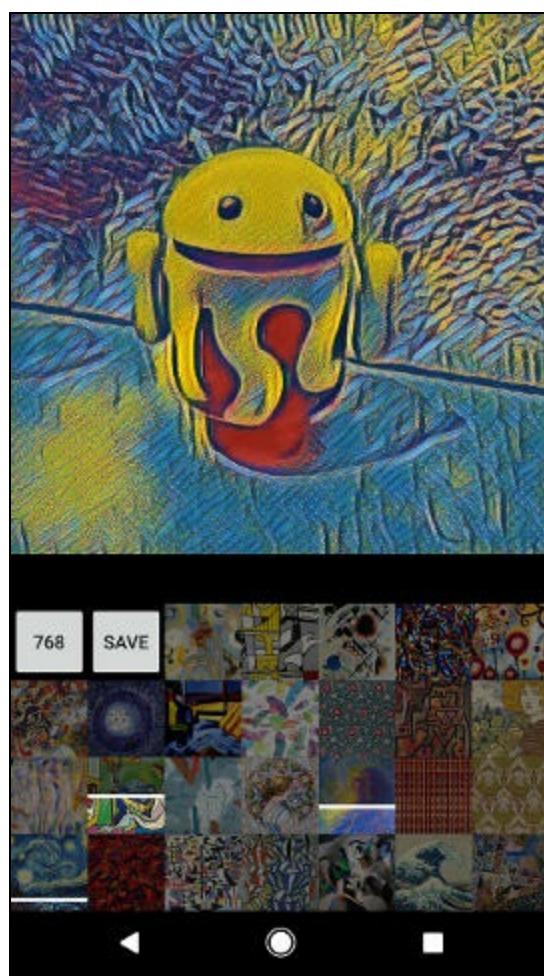


Figure 11: Sample result from running TF Stylize on Android(source: https://github.com/tensorflow/tensorflow/raw/master/tensorflow/examples/android/sample_images/stylize1.jpg)

Getting started with Android

We mentioned that we are only going to cover the Android part. So we are going first to explain the general procedure that one may follow in order to get the Android samples running, then we will dive into a concrete example.

Architecture requirements

As mentioned by TensorFlow *A device running Android 5.0 (API 21) or higher is required to run the demo due to the use of the camera2 API, although the native libraries themselves can run on API >= 14 devices.*

Prebuilt APK

If you want to right away try the demo, then you can directly download the nightly build from this <https://ci.tensorflow.org/view/Nightly/job/nightly-android/>. Go to Last Successful Artifacts then go to the `out` folder to find the `tensorflow_demo.apk` file. You can Also drop use it as a precompiled native library into your own applications.

Look at the following URL for more details:

<https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/android/README.md>

Running the demo

Once you downloaded and installed the app, it can be started via these icons (TF classify, TF detect, or TF stylize)

While running the activities, you can press the volume keys on your device which will toggle debug visualizations on and off to render additional info to the screen that will be useful for your development purposes.

Building with Android studio

If you are an Android developer, you are probably using Android Studio. TensorFlow can be built with just a few simple steps using Android Studio directly.

All the Android samples in TensorFlow rely on Bazel (which is the underlying build system that TensorFlow uses to build for Android) under the hood.

In order to get the Android samples to work in Android Studio, you need to do the following:

1. Alter the build script to let TensorFlow know where Bazel is. You name of this build script is `build.gradle` and you can find it in `tensorflow/examples/android/build.gradle`.
2. Then you just add the sample folder as a new project in Android Studio and then go ahead and build it.

By following the previous two steps, you will get your `.apk` file out of Android Studio and then you can use it right away on any Android device.

Going deeper - Building with Bazel

This the time for doing a concrete example from the beginning till having the .apk files up and running on your device. We are going to follow the Bazel approach for that.

First, you need to clone tensorflow repository, by issuing the following command:

```
| git clone --recurse-submodules https://github.com/tensorflow/tensorflow.git
```



The `reurse` submodules is needed in order to prevent some issues during compilation.

Next, you need to install Bazel (Which is the underlying build system that TensorFlow uses to build their programs, <https://bazel.build/>) and Android prerequisites:

1. Install the latest version of Bazel by following the instructions mentioned in this URL: <https://bazel.build/versions/master/docs/install.html>
2. To build the C++ part, Android NDK is required. The recommended version for it is 12b and you can download it from the following URL: https://developer.android.com/ndk/downloads/older_releases.html#ndk-12b-downloads
3. Also, you need to install the Android SDK and build tools which you can get from this (<https://developer.android.com/studio/releases/build-tools.html>.) You can also use it as part of Android Studio (<https://developer.android.com/studio/index.html>). Build tools API ≥ 23 is required to build the TF Android demo (though it will run on API ≥ 21 devices).

As mentioned above you need to provide information about the SDK and NDK by editing the WORKSPACE (you can find it under the root folder of TensorFlow) files and you need to uncomment the following lines and update the paths accordingly:

```
# Uncomment and update the paths in these entries to build the Android demo.  
#android_sdk_repository(  
#    name = "androidsdk",  
#    api_level = 23,  
#    # Ensure that you have the build_tools_version below installed in the  
#    # SDK manager as it updates periodically.  
#    build_tools_version = "25.0.2",  
#    # Replace with path to Android SDK on your system  
#    path = "<PATH_TO_SDK>",  
#)  
#  
# Android NDK r12b is recommended (higher may cause issues with Bazel)  
#android_ndk_repository(  
#    name="androidndk",  
#    path="<PATH_TO_NDK>",  
#    # This needs to be 14 or higher to compile TensorFlow.  
#    # Note that the NDK version is not the API level.  
#    api_level=14)
```

If you did not uncomment the previous lines in the workspace an error such as: "The external label `//external: android/sdk` is not bound to anything" will be reported.

Also, you need to edit the API levels for the SDK in the WORKSPACE file to the highest level installed in your SDK. As mentioned this must be ≥ 23 and the NDK API level may remain at 14.

Then you need to build the APK by issuing the following command from the workspace root:

```
| bazel build -c opt //tensorflow/examples/android:tensorflow_demo
```

As mentioned by TensorFlow *if you get build errors about protocol buffers, run git submodule update--init and make sure that you've modified your workspace file as instructed, then try building again.*

Finally you can install the .apk file but first, make sure that adb debugging is enabled on your Android 5.0 (API 21) or later device. Issue the following command from your workspace root in order to install the .apk file:

```
| adb install -r bazel-bin/tensorflow/examples/android/tensorflow_demo.apk
```

Now you can enjoy running deep learning algorithms on your Android mobile device by using TensorFlow as a platform.

Summary

Now you have covered the most recent changes in TensorFlow 1.0. Also, you knew about image and video analysis and how the integration between TensorFlow and Keras will make it even easier. Also, you learned about running deep learning that is supported by TensorFlow as a platform for your Android mobile device.

The next chapter introduces Reinforcement Learning. We'll explore the basic principles and algorithms; also we'll see some application examples, using TensorFlow and the OpenAI Gymframework, that is a powerful toolkit for developing and comparing reinforcement learning algorithms.

Reinforcement Learning

Reinforcement Learning is based on an interesting psychological theory:

Applying a reward immediately after the occurrence of a response increases its probability of reoccurring, while providing punishment after the response will decrease the probability (Thorndike, 1911).

A reward, received immediately after the execution of a correct behavior, increases the likelihood that this behavior will be repeated; while, following an undesired behavior, the application of a punishment decreases the likelihood of that error reoccurring. Therefore, once a goal has been established, Reinforcement Learning seeks to maximize the rewards received, to achieve the designated goal.

RL finds applications in different contexts in which supervised learning is inefficient.

A very short list includes the following:

- Advertising helps in learning rank, using one-shot learning for emerging items, and new users will bring more money
- Teaching robots new tasks, while retaining prior knowledge
- Deriving complex hierarchical schemes, from chess gambits to trading strategies
- Routing problems, for example, management of a shipping fleet, which trucks/truckers to assign to which cargo

A general idea that is being pursued right now is creating an algorithm that doesn't need anything apart from a description of its task. When this kind of performance is achieved, it will be applied virtually everywhere. This chapter covers the basic concepts of RL, and we will experience the Q-learning algorithm, which is one of the most popular Reinforcement Learning algorithms.

This chapter is structured as follows:

- Basic concepts of Reinforcement Learning
- Q-learning algorithm
- Introducing the OpenAI Gym framework
- Implementing FrozenLake-v0
- Q-learning with TensorFlow

Basic concepts of Reinforcement Learning

Reinforcement Learning (RL) aims to create systems that will learn and, at the same time, adapt to changes in the environment in which they are located, using a reward that is assigned to each action performed.

Software systems that process information in this way are called **intelligent agents**.

These agents decide to take an action based on the following:

- State of the system
- Learning algorithm used

To change the system state and maximize its long term rewards, and agent selects the action to be performed by continuously monitoring its environment.

To obtain a large reward and, therefore, optimize the Reinforcement Learning procedure, the agent must prefer actions that, in the past, have produced a good reward.

The actions are discovered, proving those never selected first. Therefore, the agent must exploit what it already knows, both to obtain the maximum reward, and also to explore and select the best actions for the future.

To do that, the agent should try a variety of actions and progressively favor those which seem to be the best. It then proceeds stochastically, where every action must be tried many times to get a reliable estimate of the expected reward.

Here, we describe the four main sub-elements of a Reinforcement Learning system.

The first one is the **policy**, which defines the way in which the agent must behave at a given time. In other words, a policy is a mapping between the states perceived by the environment and the actions to take when the agent is in those specific states. The policy is the heart of a Reinforcement Learning agent, because it determines the behavior that the agent must take.

The second main sub-element defines the goal of a Reinforcement Learning problem. This is the **reward** function. Each state is mapped with its own reward, indicating the degree of desire to be in that state. As mentioned previously, the goal of a Reinforcement Learning agent will be to maximize the total reward that it receives in the long run.

The third main sub-element is the **value** function. This function specifies what is good in the long run. In other words, the value of a state is the total amount of reward that an agent can expect to accumulate for the future, starting from that state. While the rewards determine an immediate desirability of the states, the values indicate the long-term desirability of states, taking into account the states that may follow and the available rewards in those states. The value function is specified

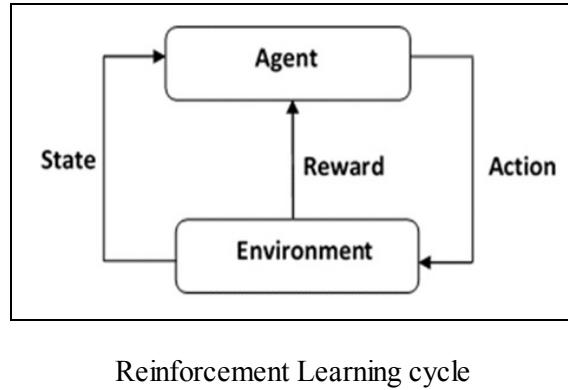
with respect to the chosen policy.

During the learning phase, an agent tries actions that determine the states with the highest value, because these actions will get the best amount of reward in the long run. While the rewards are derived directly from the environment, the values must be constantly estimated, observing an agent throughout its lifetime.

In fact, the most important component of a Reinforcement Learning algorithm is a method for efficiently estimating the values.

Finally, we have the last main sub-element, which is the **environment** (or model). This is an agent internal representation that stimulates the behavior of the environment. For example, given a state and an action, the model predicts what the next resulting state and next reward will be.

The following figure summarizes the Reinforcement Learning cycle:



An agent receives sensory information about the environment's state. Based on this information and the defined policies, the agent performs an action in the environment. This results in a reward signal. Unlike the sensory information, which may be a large feature vector, or the action, which may also have many components, the reward is a single real-valued scalar; a number.

In addition, the performed action changes the environment, leading it in a new state, where the agent can perform a new action, and so on.

The goal of learning is the maximization of the reward received, that is, this does not mean maximizing the immediate reward, but the cumulative reward received over time.

Q-learning algorithm

Solving a Reinforcement Learning problem during the learning process estimates an evaluation function. This function must be able to assess, through the sum of the rewards, the convenience or, otherwise, a policy. The basic idea of Q-learning is that the algorithm learns the optimal evaluation function on the whole space of states and actions ($S \times A$).

The so-called Q-function provides a match in the form $Q: S \times A \Rightarrow V$, where V is the value of future rewards of an action, $a \in A$, executed in the state $s \in S$.

Once it has learned the optimal function, Q , the agent will of course be able to recognize what action will lead to the highest future reward in a s state.

One of the most used examples for implementing the Q-learning algorithm involves the use of a table. Each cell of the table is a value, $Q(s; a) = V$, initialized to 0.

The agent can perform any action $a \in A$, where A is the total set of actions known by the agent. The basic idea of the algorithm is the training rule, which updates a table element, $Q(s; a)$.

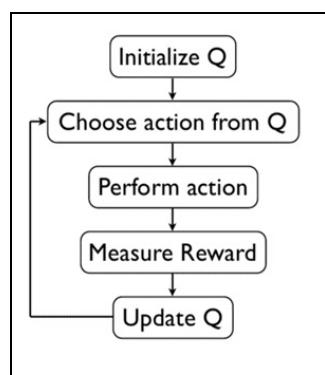
The algorithm follows these basic steps:

```
Initialize Q(s; a) arbitrarily
Repeat (for each episode)
    Initialize s
    Repeat (for each step of episode):
        Choose an action a ∈ A from s ∈ S using policy
        derived from Q
        Take an action a, observe r, s'
        Q(s; a) ← Q(s; a) + α · (r + γ · max Q(s'; a) - Q(s; a))
        s' := s - s
    Until s is terminal
```

The parameters used in the Q-value update process are as follows:

- α is the learning rate, set between 0 and 1. Setting it to 0 means that the Q-values are never updated, hence nothing is learned. Setting a high value, such as 0.9, means that learning can occur quickly.
- γ is the discount factor, also set between 0 and 1. This models the fact that future rewards are worth less than immediate rewards. Mathematically, the discount factor needs to be set to less than 0 for the algorithm to converge.
- $\max Q(s'; a)$ is the maximum reward attainable in the state following the current one, that is, the reward for taking the optimal action thereafter.

For better understanding, we have depicted the algorithm in the following figure:



Q-learning algorithm

Introducing the OpenAI Gym framework

To implement a Q-learning algorithm we'll use the **OpenAI Gym** framework, which is a TensorFlow compatible toolkit for developing and comparing Reinforcement Learning algorithms.

OpenAI Gym consists of two main parts:

- **The Gym open source library:** A collection of problems and environments that can be used to test Reinforcement Learning algorithms. All these environments have a shared interface, allowing you to write RL algorithms.
- **The OpenAI Gym service:** A site and API allowing people to meaningfully compare the performance of their trained agents.

See more references at <https://gym.openai.com>.



To get started, you'll need to have Python 2.7 or Python 3.5. To install Gym, use the `pip` installer:

```
| sudo pip install gym.
```

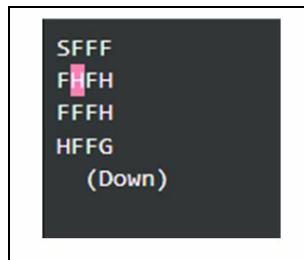
Once installed, you can list Gym's environments as follows:

```
|>>>from gym import envs  
|>>>print(envs.registry.all())
```

The output list is very long; the following is just an excerpt:

```
[EnvSpec(PredictActionsCartpole-v0),  
EnvSpec(AsteroidsramDeterministic-v0),  
EnvSpec(Asteroids-ramDeterministic-v3),  
EnvSpec(Gopher-ramDeterministic-v3),  
EnvSpec(Gopher-ramDeterministic-v0),  
EnvSpec(DoubleDunk-ramDeterministic-v3),  
EnvSpec(DoubleDunk-ramDeterministic-v0),  
EnvSpec(Carnival-v0),  
EnvSpec(FrozenLake-v0),....,  
EnvSpec(SpaceInvaders-ram-v3),  
EnvSpec(CarRacing-v0), EnvSpec(SpaceInvaders-ram-v0), ....,  
EnvSpec(Kangaroo-v0)]
```

Each `EnvSpec` defines a task to resolve, for example, the `FrozenLake-v0` representation is given in the following figure. The agent controls the movement of a character in a 4x4 grid world (see the following figure). Some tiles of the grid are walkable, and others lead to the agent falling into the water. Additionally, the movement direction of the agent is uncertain, and only partially depends on the chosen direction. The agent is rewarded for finding a walkable path to a goal tile:



A representation of the FrozenLake v0 grid word

The surface shown previously is described using a grid, such as the following:

SFFF	(S: starting point, safe)
FHFH	(F: frozensurface, safe)
FFFF	(H: hole, fall to your doom)
HFFG	(G: goal, where the frisbee islocated)

The episode ends when we reach the goal or fall in a hole. We receive a reward of one for reaching the goal, and zero otherwise.

FrozenLake-v0 implementation problem

Here we report a basic Q-learning implementation for the FrozenLake-v0 problem.

Import the following two basic libraries:

```
| import gym  
| import numpyasnp
```

Then, we load the `FrozenLake-v0` environment:

```
| environment = gym.make('FrozenLake-v0')
```

Then, we build the Q-learning table; it has the dimensions $S \times A$, where s is the dimension of the observation space, s , while a is the dimension of the action space, a :

```
| S = environment.observation_space.n  
| A = environment.action_space.n
```

The FrozenLake environment provides a state for each block, and four actions (that is, the four directions of movement), giving us a 16×4 table of Q-values to initialize:

```
| Q = np.zeros([S,A])
```

Then, we define the α parameter for the training rule and the discount γ factor:

```
| alpha = .85  
| gamma = .99
```

We fix the total number of episodes (trials):

```
| num_episodes = 2000
```

Then, we initialize the `rList`, where we'll append the cumulative reward to evaluate the algorithm's score:

```
| rList = []
```

Finally, we start the Q-learning cycle:

```
| for i in range(num_episodes):
```

Initialize the environment and other parameters:

```
| s = environment.reset()  
| cumulative_reward = 0  
| d = False  
| j = 0  
| while j < 99:  
|     j+=1
```

Randomically, we take an action from the space A :

```
| a = np.argmax(Q[s, :] + np.random.randn(1, A) * (1. / (i+1)))
```

We evaluate the action, a , by the function, `environment.step()`, getting the reward and the state s_1 :

```
| s1, reward, d, _ = env.step(a)
```

Update the $Q(s, a)$ table with the training rule:

```
| Q[s, a] = Q[s, a] + alpha * (reward + gamma * np.max(Q[s1, :]) - Q[s, a])
| cumulative_reward += reward
```

Set the state for the next learning cycle:

```
| s = s1
| if d == True:
|     break
| rList.append(cumulative_reward)
```

Print the score over time and the resulting Q-table:

```
| print "Score over time: " + str(sum(rList)/num_episodes)
| print "Final Q-TableValues"
| print Q
```

The average reward is about 0.54 over 100 consecutive trials as shown in the following figure:

```
[2017-03-23 12:22:49,913] Making new env: FrozenLake-v0
Score over time: 0.3585
Final Q-Table Values
[[ 4.90034838e-03  1.23733520e-02  5.04857351e-01  1.18572787e-02]
 [ 6.14009765e-04  1.34354386e-03  1.39327124e-03  5.88345699e-01]
 [ 2.42003179e-03  2.53712381e-03  1.27103632e-03  3.36417875e-01]
 [ 1.60332674e-03  6.60331077e-04  6.50987843e-04  1.96388199e-01]
 [ 6.38172447e-01  1.23434831e-03  1.35672865e-03  8.99709408e-05]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 1.78445198e-01  1.27421388e-04  2.70432817e-05  7.55201005e-12]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 5.85462465e-05  1.52400799e-03  6.22678642e-05  3.00741687e-01]
 [ 3.15488045e-03  6.66874039e-02  0.00000000e+00  4.21513681e-04]
 [ 7.99666157e-01  9.87928455e-04  2.11361272e-04  2.11179559e-04]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 1.20525081e-04  0.00000000e+00  9.20956992e-01  0.00000000e+00]
 [ 0.00000000e+00  0.00000000e+00  9.91561828e-01  0.00000000e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]]
```

Technically, we didn't resolve it. Indeed, `FrozenLake-v0` defines solving as getting an average reward of 0.78 over 100 consecutive trials; we could improve this result by tuning the configuration parameters, but this is out of the scope of this section.

Source code for the FrozenLake-v0 problem

The following is the implementation of the Q-learning algorithm for the `FrozenLake-v0` problem:

```
import gym
import numpy as np

env = gym.make('FrozenLake-v0')

#Initialize table with all zeros
Q = np.zeros([env.observation_space.n,env.action_space.n])
# Set learning parameters
lr = .85
gamma = .99
num_episodes = 2000

#create lists to contain total rewards and steps per episode
rList = []
for i in range(num_episodes):
    #Reset environment and get first new observation
    s = env.reset()
    rAll = 0
    d = False
    j = 0

    #The Q-Table learning algorithm
    while j < 99:
        j+=1

        #Choose an action by greedily (with noise) picking from Q table
        a=np.argmax(Q[s,:]+\n            np.random.randn(1,env.action_space.n)*(1./(i+1)))

        #Get new state and reward from environment
        s1,r,d,_ = env.step(a)

        #Update Q-Table with new knowledge
        Q[s,a] = Q[s,a] + lr*(r + gamma *np.max(Q[s1,:]) - Q[s,a])
        rAll += r
        s = s1
        if d == True:
            break

    rList.append(rAll)

print "Score over time: " + str(sum(rList)/num_episodes)
print "Final Q-Table Values"
print Q
```


Q-learning with TensorFlow

In the previous example, we saw how it is relatively simple, using a 16×4 grid, to update the Q-table at each step of the learning process. It is easy to imagine that the use of this table can serve for simple problems, but in real-world problems, we need a more sophisticated mechanism to update the system state. This is the point where deep learning steps in. Neural networks are exceptionally good at coming up with good features for highly structured data.

In this final section, we'll look at how to manage a Q-function with a neural network, which takes the state and action as input, and outputs the corresponding Q-value.

To do that, we'll build a one layer network that takes the state, encoded in a $[1 \times 16]$ vector, which learns the best move (action), mapping the possible actions in a vector of length four.



A recent application of deep Q-networks has been successful at playing some Atari 2600 games at expert human levels. Preliminary results were presented in 2014, with a paper published in February 2015, in Nature.

In the following, we describe our TensorFlow-based implementation of a Q-learning neural network for the `FrozenLake-v0` problem.

Import all the libraries with the help of the following code:

```
import gym
import numpy as np
import random
import tensorflow as tf
import matplotlib.pyplot as plt
```

To install matplotlib, you should execute the following commands on terminal:



```
$ apt-cache search python3-matplotlib
```

If you find it like its available then you can install it from:

```
$ sudo apt-get install python3-matplotlib
```

Load and set the environment to test:

```
| env = gym.make('FrozenLake-v0')
```

The input network is a state, encoded in a tensor of shape $[1, 16]$. For this reason, we define the `inputs1` placeholder:

```
| inputs1 = tf.placeholder(shape=[1,16], dtype=tf.float32)
```

The network weights are initially chosen randomly by the `tf.random_uniform` function:

```
| w = tf.Variable(tf.random_uniform([16,4],0,0.01))
```

The network output is given by the product of the `inputs1` placeholder and the weights:

```
| Qout = tf.matmul(inputs1,W)
```

The `argmax` evaluated on `Qout` will give the predicted value:

```
| predict = tf.argmax(Qout,1)
```

The best move (`Qtarger`) is encoded in a `[1, 4]` tensor shape:

```
| Qtarger = tf.placeholder(shape=[1,4],dtype=tf.float32)
```

Next, we must define a `loss` function to optimize for the backpropagation procedure. The `loss` function is as follows:

$$loss = \sum(Q-target - Q)^2$$

Where the difference between the current predicted Q-values and the target value is computed, and the gradients are passed through the network:

```
| loss = tf.reduce_sum(tf.square(Qtarger- Qout))
```

The optimizing function, is the well-known `GradientDescentOptimizer`:

```
| trainer = tf.train.GradientDescentOptimizer(learning_rate=0.1)
| updateModel = trainer.minimize(loss)
```

Reset and initialize the computational graph:

```
| tf.reset_default_graph()
| init = tf.global_variables_initializer()
```

Following this, we set the parameter for the Q-learning training procedure:

```
| gamma = .99
| e = 0.1
| num_episodes = 6000
|
| jList = []
| rList = []
```

We carry out the running session, in which the network will have to learn the best possible sequence of moves:

```
with tf.Session() as sess:
    sess.run(init)
    for i in range(num_episodes):
        s = env.reset()
        rAll = 0
        d = False
        j = 0

        while j < 99:
            j+=1
```

The input state is used here to feed the network:

```
a,allQ = sess.run([predict,Qout],\n                 feed_dict=\n                     {inputs1:np.identity(16)[s:s+1]})
```

A random state is chosen from the output tensor, `a`:

```
if np.random.rand(1) < e:\n    a[0] = env.action_space.sample()
```

Evaluate the action, `a[0]`, using the function `env.step()`, obtaining the reward, `r`, and the state, `s1`:

```
| s1,r,d,_ = env.step(a[0])
```

This new state `s1` is used to update the Q-tensor:

```
Q1 = sess.run(Qout,feed_dict=\n              {inputs1:np.identity(16)[s1:s1+1]})\nmaxQ1 = np.max(Q1)\ntargetQ = allQ\ntargetQ[0,a[0]] = r + y*maxQ1
```

Of course, the weights must be updated for the backpropagation procedure:

```
| _,W1 = sess.run([updateModel,W],\n                  feed_dict=\n                      {inputs1:np.identity(16)[s:s+1],nextQ:targetQ})
```

The `rAll` parameter, here, defines the total reward that will be incremented during the session. Let's recall that the goal of a Reinforcement Learning agent will be to maximize the total reward that it receives in the long run:

```
| rAll += r
```

Update the state of the environment for the next step:

```
s = s1\nif d == True:\n    e = 1./((i/50) + 10)\n    break\njList.append(j)\nrList.append(rAll)
```

When the computation ends, the percent of successful episodes will be displayed:

```
| print "Percent of succesful episodes: " +\\n| str(sum(rList)/num_episodes) + "%"\\n|
```

Running the model, you should have a result like the following, which can be improved by tuning the network parameters:

```
>>>\n[2017-03-23 12:36:19,986] Making new env: FrozenLake-v0\nPercent of successful episodes: 0.558%\n>>>
```


Source code for the Q-learning neural network

The following is the full code for the example shown previously:

```
import gym
import numpy as np
import random
import tensorflow as tf
import matplotlib.pyplot as plt

#Define the FrozenLake enviroment
env = gym.make('FrozenLake-v0')

#Setup the TensorFlow placeholders and variabiles
tf.reset_default_graph()
inputs1 = tf.placeholder(shape=[1,16],dtype=tf.float32)
W = tf.Variable(tf.random_uniform([16,4],0,0.01))
Qout = tf.matmul(inputs1,W)
predict = tf.argmax(Qout,1)
nextQ = tf.placeholder(shape=[1,4],dtype=tf.float32)

#define the loss and optimization functions
loss = tf.reduce_sum(tf.square(nextQ - Qout))
trainer = tf.train.GradientDescentOptimizer(learning_rate=0.1)
updateModel = trainer.minimize(loss)

#initilize the vabiiables
init = tf.global_variables_initializer()

#prepare the q-learning parameters
gamma = .99
e = 0.1
num_episodes = 6000
jList = []
rList = []

#Run the session
with tf.Session() as sess:
    sess.run(init)
#Start the Q-learning procedure
    for i in range(num_episodes):
        s = env.reset()
        rAll = 0
        d = False
        j = 0
        while j < 99:
            j+=1
            a,allQ = sess.run([predict,Qout],\
                feed_dict= \
                    {inputs1:np.identity(16) [s:s+1]})

            if np.random.rand(1) < e:
                a[0] = env.action_space.sample()
            s1,r,d,_ = env.step(a[0])
            Q1 = sess.run(Qout,feed_dict=\
                {inputs1:np.identity(16) [s1:s1+1]})

            maxQ1 = np.max(Q1)
            targetQ = allQ
            targetQ[0,a[0]] = r + gamma *maxQ1
            _,W1 = sess.run([updateModel,W],\
                feed_dict=\
                    {inputs1:np.identity(16) [s:s+1],nextQ:targetQ})

#cumulate the total reward
        rAll += r
        s = s1
        if d == True:
            e = 1./((i/50) + 10)
            break
        jList.append(j)
        rList.append(rAll)
```

```
| #print the results
|   print "Percent of successful episodes: " +\
|         str(sum(rList)/num_episodes) + "%"
```


Summary

This chapter covers the basic principles of Reinforcement Learning and the fundamental Q-learning algorithm.

The distinctive feature of Q-learning is its capacity to choose between immediate rewards and delayed rewards. Q-learning at its simplest uses tables to store data. This very quickly loses viability as the state/action space of the system it is monitoring/controlling increases.

We can overcome this problem by using a neural network as a function approximator, which takes the state and action as input, and outputs the corresponding Q-value.

Following this idea, we implemented a Q-learning neural network using the TensorFlow framework and the OpenAI Gym toolkit for developing and comparing Reinforcement Learning algorithms.

Our journey into *Deep Learning with TensorFlow* ends here.

Deep learning is a very productive research area; there are many books, courses, and online resources that may help you to go deeper into its theory and programming. In addition, TensorFlow provides a rich set of tools for working with deep learning models, and so on.

We really hope for you to be a part of the TensorFlow community, which is very active and expects enthusiastic people to join in!