Deep Learning with Python

A Hands-on Introduction

Nikhil Ketkar

Apress®

Deep Learning with Python

A Hands-on Introduction

Nikhil Ketkar

Apress®

Deep Learning with Python: A Hands-on Introduction

Nikhil Ketkar
Bangalore, Karnataka, India

ISBN-13 (pbk): 978-1-4842-2765-7 ISBN-13 (electronic): 978-1-4842-2766-4
DOI 10.1007/978-1-4842-2766-4

Library of Congress Control Number: 2017939734

Managing Director: Welmoed Spahr
Editorial Director: Todd Green
Acquisitions Editor: Celestin Suresh John
Development Editor: Matthew Moodie and Anila Vincent
Technical Reviewer: Jojo Moolayail
Coordinating Editor: Prachi Mehta
Copy Editor: Larissa Shmailo
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global
Cover image designed by Freepik

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit http://www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484227657. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

To Aditi.

Contents at a Glance

Contents

About the Author

Nikhil Ketkar currently leads the Machine Learning Platform team at Flipkart, India's largest e-commerce company. He received his PhD from Washington State University. Following that, he conducted postdoctoral research at University of North Carolina at Charotte, which was followed by a brief stint in high frequency trading at TransMarket in Chicago. More recently, he led the data mining team in Guavus, a startup doing big data analytics in the telecom domain and Indix, a startup doing data science in the e-commerce domain. His research interests include machine learning and graph theory.

About the Technical Reviewer

Jojo Moolayil is a data scientist and author of *Smarter Decisions—The Intersection of Internet of Things and Decision Science*. With over four years of industrial experience in data science, decision science, and IoT, he has worked with industry leaders on high-impact and critical projects across multiple verticals. He is currently associated with General Electric, a pioneer and leader in data science for industrial IoT, and lives in Bengaluru, the Silicon Valley of India.

He was born and raised in Pune, India and graduated from the University of Pune with a major in information technology engineering. He started his career with Mu Sigma, the world's largest pure play analytics provider, and worked with the leaders of many Fortune 50 clients. One of the early enthusiasts to venture into IoT analytics, he now focuses on solving decision science problems for industrial IoT use cases. As a part of his role at GE, he also develops data science and decision science products and platforms for industrial IoT.

Acknowledgments

I would like to thank my colleagues at Flipkart and Indix, and the technical reviewers for their feedback and comments.

CHAPTER 1

Introduction to Deep Learning

This chapter gives a broad overview and a historical context around the subject of deep learning. It also gives the reader a roadmap for navigating the book, the prerequisites, and further reading to dive deeper into the subject matter.

Historical Context

The field of Artificial Intelligence (AI), which can definitely be considered to be the parent field of deep learning, has a rich history going back to 1950. While we will not cover this history in much detail, we will go over some of the key turning points in the field, which will lead us to deep learning.

Tasks that AI focused on in its early days were tasks that could be easily described formally, like the game of checkers or chess. This notion of *being able to easily describe the task formally* is at the heart of what can or cannot be done easily by a computer program. For instance, consider the game of chess. The formal description of the game of chess would be the representation of the board, a description of how each of the pieces move, the starting configuration, and a description of the configuration wherein the game terminates.

With these notions formalized, it's relatively easy to model a chess-playing AI program as a search and, given sufficient computational resources, it's possible to produces a relatively good chess-playing AI.

The first era of AI focused on such tasks with a fair amount of success. At the heart of the methodology was a symbolic representation of the domain and the manipulation of symbols based on given rules (with increasingly sophisticated algorithms for searching the solution space to arrive at a solution).

It must be noted that the formal definitions of such rules were done manually. However, such early AI systems were fairly general purpose task/problem solvers in the sense that any problem that could be described formally could be solved with the generic approach.

The key limitation about such systems is that the game of chess is a relatively easy problem for AI simply because the problem setting is relatively simple and can be easily formalized. This is not the case with many of the problems human beings solve on a day-to-day basis (natural intelligence). For instance, consider diagnosing a disease (as a physician does) or transcribing human speech to text. These tasks, like most other tasks human beings master easily, are hard to describe formally and presented a challenge in the early days of AI.

Human beings address such tasks by leveraging a large amount of knowledge about the task/problem domain. Given this observation, subsequent AI systems relied on a large knowledge base which captured the knowledge about the problem/task domain. One point to be noted is the term used here is knowledge, not information or data. By knowledge we simply mean data/information that a program/algorithm can reason about. An example of this could be a graph representation of a map with edges labeled with distances and about traffic (which is being constantly updated), which allows a program to reason about the shortest path between points.

N. Ketkar, *Deep Learning with Python*, DOI 10.1007/978-1-4842-2766-4_1

Such knowledge-based systems wherein the knowledge was compiled by experts and represented in a way which allowed algorithms/programs to reason about it represent the second generation of AI. At the heart of such approaches were increasingly sophisticated approaches for representing and reasoning about knowledge to solve tasks/problems which required such knowledge. Examples of such sophistication include the use of first order logic to encode knowledge and probabilistic representations to capture and reason where uncertainty is inherent to the domain.

One of the key challenges that such systems faced and addressed to some extent was the uncertainty inherent in many domains. Human beings are relatively good at reasoning in environments with unknowns and uncertainty. One key observation here is that even the knowledge we hold about a domain is not black or white but gray. A lot of progress was made in this era on representing and reasoning about unknowns and uncertainty. There were some limited successes in tasks like diagnosing a disease, which relied on leveraging and reasoning using a knowledge base in the presence of unknowns and uncertainty.

The key limitation of such systems was the need to hand compile the knowledge about the domain from experts. Collecting, compiling, and maintaining such knowledge bases rendered such systems unpractical. In certain domains, it was extremely hard to even collect and compile such knowledge (for instance, transcribing speech to text or translating documents from one language to another). While human beings can easily learn to do such tasks, it's extremely challenging to hand compile and encode the knowledge related to the tasks (for instance, the knowledge of the English language and grammar, accents, and subject matter).

Human beings address such tasks by acquiring knowledge about a task/problem domain, a process which is referred to as learning. Given this observation, the focus of subsequent work in AI shifted over a decade or two to algorithms that improved their performance based on data provided to them. The focus of this subfield was to develop algorithms that acquired relevant knowledge for a task/problem domain given data. It is important to note that this knowledge acquisition relied on labeled data and a suitable representation of labeled data as defined by a human being.

For instance, consider the problem of diagnosing a disease. For such a task, a human expert would collect a lot of cases where a patient had and did not have the disease in question. Then, the human expert would identify a number of features that would aid making the prediction like, say, the age of the patient, the gender, and results from a number of diagnostic tests like blood pressure, blood sugar, etc. The human expert would compile all this data and represent it in a suitable way like scaling/normalizing the data, etc. Once this data was prepared, a machine learning algorithm can learn how to infer whether the patient has the disease or not by generalizing from the labeled data. Note that the labeled data consisted of patients that both have and do not have the disease. So, in essence, the underlying ML algorithm is essentially doing the job of finding a mathematical function that can produce the right outcome (disease or no disease) given the inputs (features like age, gender, data from diagnostic tests, etc.). Finding the simplest mathematical function that predicts the outputs with required level of accuracy is at the heart of the field of ML. Specific questions like how many examples are required to learn a task or the time complexity of the algorithm, etc., are specific questions on which the field of ML has provided answers with theoretical justification. The field has matured to a point where, given enough data, computer resources, and human resources to engineer features, a large class of problems are solvable.

The key limitation of mainstream ML algorithms is that applying them to a new problem domain requires a massive amount of feature engineering. For instance, consider the problem of recognizing objects in images. Using traditional ML techniques, such a problem will require a massive feature engineering effort wherein experts would identify and generate features which would be used by the ML algorithm. In a sense, the true intelligence is in the identification of features and what the ML algorithm is doing is simply learning how to combine these features to arrive at the correct answer. This identification of features or the representation of data which domain experts do before ML algorithms are applied is both a conceptual and practical bottleneck in AI.

It's a conceptual bottleneck because if features are being identified by domain experts, and the ML algorithm is simply learning to combine and draw conclusions from this, is this really AI? It's a practical bottleneck because the process of building models via traditional ML is bottlenecked by the amount of feature engineering required; there are limits to how much human effort can be thrown at the problem.

Human beings learn concepts starting from raw data. For instance, a child shown a few examples/instances of a particular animal (like, say, cats) will soon learn to identify cats. The learning process does not involve a parent identifying features like does it have whiskers or does it have fur or does it have a tail. Human learning goes from raw data to a conclusion without the explicit step where features are identified and provided to the learner. In a sense, human beings learn the appropriate representation of data from the data itself. Furthermore, they organize concepts as a hierarchy where complicated concepts are expressed using primitive concepts.

The field of deep learning has its primary focus on learning appropriate representations of data such that these could be used to draw conclusions. The word *deep* in deep learning refers to the idea of learning the hierarchy of concepts directly from raw data. A more technically appropriate term for deep learning would be representation learning, and a more practical term for the same would be automated feature engineering.

Advances in Related Fields

It is important to make a note of advances in other fields that have played a key role in the recent interest and success of deep learning. The following points are to be noted.

1. The ability to collect, store, and operate over large amounts of data has greatly advanced over the last decade (for instance, the Apache Hadoop Ecosystem).
2. The ability to generate supervised training data (which is basically data with labels—an example of this would be pictures annotated with the objects in the picture) has improved a lot with the availability of crowd-sourcing services (like Amazon Mechanical Turk).
3. The massive improvements in computational horsepower brought about by Graphical Processor Units.
4. The advances in both theory and software implementation of automatic differentiation (like Theano).

While these advancements are peripheral to deep learning, they have played a big role in enabling advances in deep learning.

Prerequisites

The key prerequisites for reading this book are a working knowledge of Python and some course work on linear algebra, calculus, and probability. It is recommended that readers refer to the following in case they need to cover these prerequisites.

1. *Dive Into Python* by Mark Pilgrim for Python.
2. *Linear Algebra* by Gilbert Strang for linear algebra.
3. *Calculus* by Gilbert Strang for calculus.
4. *All of Statistics* by Larry Wasserman for probability (Section 1, Chapters 1-5).

Overview of Subsequent Chapters

We now provide an overall outline of the subsequent chapters for the reader. It is important to note that each of the chapters covers either the concepts or the skills (or in certain cases both) with respect to deep learning. We highlight these below so that the readers can ensure that they have internalized these concepts and skills. It is highly recommended that the readers not only read the chapters but also work out the mathematical details (using pen and paper) and play with the source code provided in each of the chapters.

1. Chapter 2 covers the basics of Machine Learning. The key take home point for this chapter is the concept of generalizing over unseen examples, the ideas of over-fitting and under-fitting the training data, the capacity of the model, and the notion of regularization.
2. Chapter 3 covers Feed Forward Neural Networks and serves as the conceptual foundation for the entire book. Concepts like the overall structure of the neural network, the input, hidden and output layers, cost functions and their basis on the principle of Maximum Likelihood are the important concepts in this chapter.
3. Chapter 4 provides a hands-on introduction to the Theano library. It covers how to define networks as computational graphs, automatically derive gradients for complicated networks, and train neural networks.
4. Chapter 5 covers Convolutional Neural Networks, which are perhaps the most successful application of deep learning.
5. Chapter 6 covers Recurrent Neural Networks and Long Short Term Memory (LSTM) networks, which are another successful application of deep learning.
6. Chapter 7 provides a hands-on introduction to the Keras library. The Keras library provides a number of high-level abstractions over the Theano library and is probably the ideal go-to tool when it comes to building deep learning applications.
7. Chapter 8 introduces the reader to Stochastic Gradient Descent (SGD), which is the most common procedure used to train Neural Networks. This chapter also covers the shortcomings of SGD and a number of variations to SGD that address these shortcomings.
8. Chapter 9 introduces the reader to Automatic Differentiation (commonly referred to as backpropagation), which is a standard technique used to derive gradients (required for SGD) for arbitrarily complicated networks.
9. Chapter 10 introduces the reader to Graphical Processing Units (GPUs) and GPU-based computation, which has acted as a key enabling technology for deep learning.

Installing the Required Libraries

There are a number of libraries that the reader will need to install in order to run the source code for the examples in the chapters. We recommend that the reader install Anaconda Python Distribution (`https://www.continuum.io/downloads`), which would make the process of installing the required packages significantly easy (using either conda or pip). The list of packages the reader would need includes Scikit-learn, Theano, Autograd, Keras, and PyOpenCL.

CHAPTER 2

Machine Learning Fundamentals

Deep Learning is a branch of Machine Learning and in this chapter we will cover the fundamentals of Machine Learning. While machine learning as a subject is inherently mathematical in nature, we will keep mathematics to the basic minimum required to develop intuition about the subject. Prerequisites for the subject matter covered in this chapter would be linear algebra, multivariable calculus, and basic probability theory.

Intuition

As human beings we are intuitively aware of the concept of learning: it simply means to get better at a task over a period of time. The task could be physical (like learning to drive a car) or intellectual (like learning a new language). The subject of machine learning focuses on development of algorithms that can learn as humans do; that is, they get better at a task over a period over time, with experience.

The first question to ask is why we would be interested in development of algorithms that improve their performance over time, with experience. After all, there are many algorithms that are developed and implemented to solve real world problems that don't improve over time, they simply are developed by humans and implemented in software and they get the job done. From banking to e-commerce and from navigation systems in our cars to landing a spacecraft on the moon, algorithms are everywhere, and, a majority of them do not improve over time. These algorithms simply perform the task they are intended to perform, with some maintenance required from time to time. Why do we need machine learning?

The answer to this question is that for certain tasks it is easier to develop an algorithm that learns/improves its performance with experience than to develop an algorithm manually. While this might seem unintuitive to the reader at this point, we will build intuition for this during the course of this chapter.

Binary Classification

In order to further discuss the matter at hand, we need to be precise about some of the terms we have been intuitively using, like task, learning, experience, and improvement. We will start with the task of binary classification.

Consider an abstract problem domain where we have data of the form

$$D = \{(x_1, y_1), (x_2, y_2), \ldots (x_n, y_n)\}$$

where $x \in \mathbb{R}^n$ and $y = \pm 1$. We do not have access to all such data but only a subset $S \in D$. Using S, our task is to generate a computational procedure that implements the function $f : x \rightarrow y$ such that we can use f to make predictions over unseen data $(x_i, y_i) \notin S$ that are correct, $f(x_i) = y_i$. Let us denote $U \in D$ as the set of

N. Ketkar, *Deep Learning with Python*, DOI 10.1007/978-1-4842-2766-4_2

unseen data, that is, $(x_i, y_i) \notin S$ and $(x_i, y_i) \in U$. We measure performance over this task as the error over unseen data,

$$E(f, D, U) = \frac{\sum_{(x_i, y_i) \in U} \left[f(x_i) \neq y_i \right]}{|U|}.$$

We now have a precise definition of the task, which is to categorize data into one of two categories ($y = \pm 1$) based on some seen data S by generating f. We measure performance (and improvement in performance) using the error $E(f, D, U)$ over unseen data U. The size of the seen data $|S|$ is the conceptual equivalent of experience. In this context, we want to develop algorithms that generate such functions f (which are commonly referred to as a model). In general, the field of machine learning studies the development of such algorithms that produce models that make predictions over unseen data for such algorithms, and other formal tasks (we will be introducing multiple such tasks later in the chapter). Note that the x is commonly referred to as the input/input variable and y is referred to as the output/output variable.

As with any other discipline in computer science, the computational characteristics of such algorithms is an important facet, but in addition to that we also would like to have a model f that achieves a lower error $E(f, D, U)$ with as small a $|S|$ as possible.

Let us now relate this abstract but precise definition to a real world problem so that our abstractions are grounded. Let us say an e-commerce web site wants to customize the landing page for registered users to show them the products the users might be interested in buying. The web site has historical data on users and would like to implement this as a feature so as to increase sales. Let us now see how this real world problem maps onto the abstract problem of binary classification we described earlier.

The first thing that one might notice is that given a particular user and a particular product, one wants to predict whether the user will buy the product. Since this is the value to be predicted, it maps onto $y = \pm 1$, where we will let the value of $y = +1$ denote the prediction that the user will buy the product and we will denote $y = -1$ as the prediction that the user does not buy the product. Note that that there is no particular reason for picking these values; we could have swapped this (let $y = +1$ denote the does not buy case and $y = -1$ denote the buy case) and there would be no difference. We just use $y = \pm 1$ to denote the two classes of interest to categorize data. Next, let us assume that we can somehow represent the attributes of the product and the user's buying and browsing history as $x \in \mathbb{R}^n$. This step is referred to as feature engineering in machine learning and we will cover it later in the chapter. For now, it suffices to say that we are able to generate such a mapping. Thus, we have historical data of what the users browsed and bought, attributes of a product and whether the user bought the product or not mapped onto $\{(x_1, y_1), (x_2, y_2), \ldots (x_n, y_n)\}$. Now, based on this data, we would like to generate a function or a model $f : x \to y$ which we can use to determine which products a particular user will buy and use this to populate the landing page for users. We can measure how well the model is doing on unseen data by populating the landing page for users and see whether they buy the products or not and evaluate the error $E(f, D, U)$.

Regression

Let us introduce another task, namely regression. Here, we have data of the form $D = \{(x_1, y_1), (x_2, y_2), \ldots (x_n, y_n)\}$ where $x \in \mathbb{R}^n$ and $y \in \mathbb{R}$ and our task is to generate a computational procedure that implements the function $f : x \to y$. Note that instead of the prediction being a binary class label $y = \pm 1$ like in binary classification, we have real valued prediction. We measure performance over this task as the root mean squared error (RMSE) over unseen data,

$$E(f,D,U)=\left(\frac{\sum_{(x_i,y_i)\in U}\left(y_i-f(x_i)\right)^2}{|U|}\right)^{\frac{1}{2}}.$$

■ **Note** That the RMSE is simply taking the difference between the predicted and actual value, squaring it so as to account for both positive and negative differences, taking the mean so as to aggregate over all the unseen data and, finally, taking the square root so as to counterbalance the square operation.

A real world problem that corresponds to the abstract task of regression is to predict the credit score for an individual based on their financial history, which can be used by a credit card company to extend the line of credit.

Generalization

Let us now cover what is the single most important intuition in machine leaning, which is that we want to develop/generate models that have good performance over unseen data. In order to do that, let us introduce a toy data set for a regression task first.

We generate the toy dataset by generating 100 values equidistantly between -1 and 1 as the input variable (x). We generate the output variable (y) based on $y=2+x+2x^2+\epsilon$ where $\epsilon \sim \mathcal{N}(0,0.1)$ is noise (random variation) from a normal distribution with 0 mean and 0.1 being the standard deviation. Code for this is presented in Listing 2-1 and the data is plotted in Figure 2-1.

Listing 2-1. Generalization vs. Rote Learning

```
#Generate Toy Dataset
import pylab
import numpy
x = numpy.linspace(-1,1,100)
signal = 2 + x + 2 * x * x
noise = numpy.random.normal(0, 0.1, 100)
y = signal + noise
pylab.plot(signal,'b');
pylab.plot(y,'g')
pylab.plot(noise, 'r')
pylab.xlabel("x")
pylab.ylabel("y")
pylab.legend(["Without Noise", "With Noise", "Noise"], loc = 2)
x_train = x[0:80]
y_train = y[0:80]

# Model with degree 1
pylab.figure()
degree = 2
X_train = numpy.column_stack([numpy.power(x_train,i) for i in xrange(0,degree)])
```

```
model = numpy.dot(numpy.dot(numpy.linalg.inv(numpy.dot(X_train.transpose(),X_train)),X_
train.transpose()),y_train)
pylab.plot(x,y,'g')
pylab.xlabel("x")
pylab.ylabel("y")
predicted = numpy.dot(model, [numpy.power(x,i) for i in xrange(0,degree)])
pylab.plot(x, predicted,'r')
pylab.legend(["Actual", "Predicted"], loc = 2)
train_rmse1 = numpy.sqrt(numpy.sum(numpy.dot(y[0:80] - predicted[0:80], y_train -
predicted[0:80])))
test_rmse1 = numpy.sqrt(numpy.sum(numpy.dot(y[80:] - predicted[80:], y[80:] -
predicted[80:])))
print("Train RMSE (Degree = 1)", train_rmse1)
print("Test RMSE (Degree = 1)", test_rmse1)

# Model with degree 2
pylab.figure()
degree = 3
X_train = numpy.column_stack([numpy.power(x_train,i) for i in xrange(0,degree)])
model = numpy.dot(numpy.dot(numpy.linalg.inv(numpy.dot(X_train.transpose(),X_train)),
X_train.transpose()),y_train)
pylab.plot(x,y,'g')
pylab.xlabel("x")
pylab.ylabel("y")
predicted = numpy.dot(model, [numpy.power(x,i) for i in xrange(0,degree)])
pylab.plot(x, predicted,'r')
pylab.legend(["Actual", "Predicted"], loc = 2)
train_rmse1 = numpy.sqrt(numpy.sum(numpy.dot(y[0:80] - predicted[0:80],
y_train - predicted[0:80])))
test_rmse1 = numpy.sqrt(numpy.sum(numpy.dot(y[80:] - predicted[80:],
y[80:] - predicted[80:])))
print("Train RMSE (Degree = 2)", train_rmse1)
print("Test RMSE (Degree = 2)", test_rmse1)

# Model with degree 8
pylab.figure()
degree = 9
X_train = numpy.column_stack([numpy.power(x_train,i) for i in xrange(0,degree)])
model = numpy.dot(numpy.dot(numpy.linalg.inv(numpy.dot(X_train.transpose(),X_train)),
X_train.transpose()), y_train)
pylab.plot(x, y,'g')
pylab.xlabel("x")
pylab.ylabel("y")
predicted = numpy.dot(model, [numpy.power(x,i) for i in xrange(0,degree)])
pylab.plot(x, predicted,'r')
pylab.legend(["Actual", "Predicted"], loc = 3)
train_rmse2 = numpy.sqrt(numpy.sum(numpy.dot(y[0:80] - predicted[0:80],
y_train - predicted[0:80])))
test_rmse2 = numpy.sqrt(numpy.sum(numpy.dot(y[80:] - predicted[80:],
y[80:] - predicted[80:])))
print("Train RMSE (Degree = 8)", train_rmse2)
print("Test RMSE (Degree = 8)", test_rmse2)
```

```
# Output
Train RMSE (Degree = 1) 3.50756834691
Test RMSE (Degree = 1) 7.69514326946
Train RMSE (Degree = 2) 0.91896252959
Test RMSE (Degree = 2) 0.446173435392
Train RMSE (Degree = 8) 0.897346255079
Test RMSE (Degree = 8) 14.1908525449
```

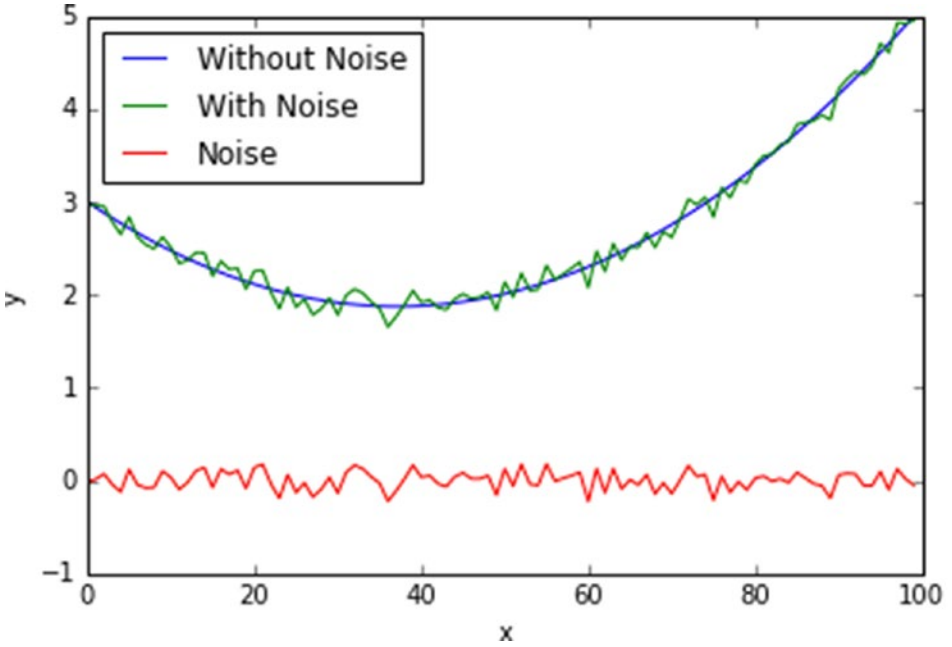


Figure 2-1. *Generate a toy problem dataset for regression*

In order to simulate seen and unseen data, we use the first 80 data points as seen data and the rest we treat as unseen data. That is, we build the model using only the first 80 data points and use the rest for evaluating the model.

Next, we use a very simple algorithm to generate a model, commonly referred to as Least Squares. Given a data set of the form $D = \{(x_1, y_1), (x_2, y_2), \ldots (x_n, y_n)\}$ where $x \in \mathbb{R}^n$ and $y \in \mathbb{R}$, the least squares model takes the form $y = \beta x$ where β is a vector such that $\|X\beta - y\|_2^2$ is minimized. Here X is a matrix where each row is an x, thus $X \in \mathbb{R}^{m \times n}$ with m being the number of examples (in our case 80). The value of β can be derived using the closed form $\beta = \left(X^T X\right)^{-1} X^T y$. We are glossing over a lot of important details of the least squares method but those are secondary to the current discussion. The more pertinent detail is how we transform the input variable to a suitable form. In our first model, we will transform x to be a vector of values $[x^0, x^1, x^2]$. That is, if $x = 2$, it will be transformed to $[1, 2, 4]$. Post this transformation, we can generate a least squares model β using the formula described above. What is happening under the hood is that we are approximating the given data with a second order polynomial (degree = 2) equation, and the least squares algorithm is simply curve fitting or generating the coefficients for each of $[x^0, x^1, x^2]$.

We can evaluate the model on the unseen data using the RMSE metric. We can also compute the RMSE metric on the training data. The actual and predicted values are plotted in Figure 2-2 and listing 2-1 shows the source code for generating the model.

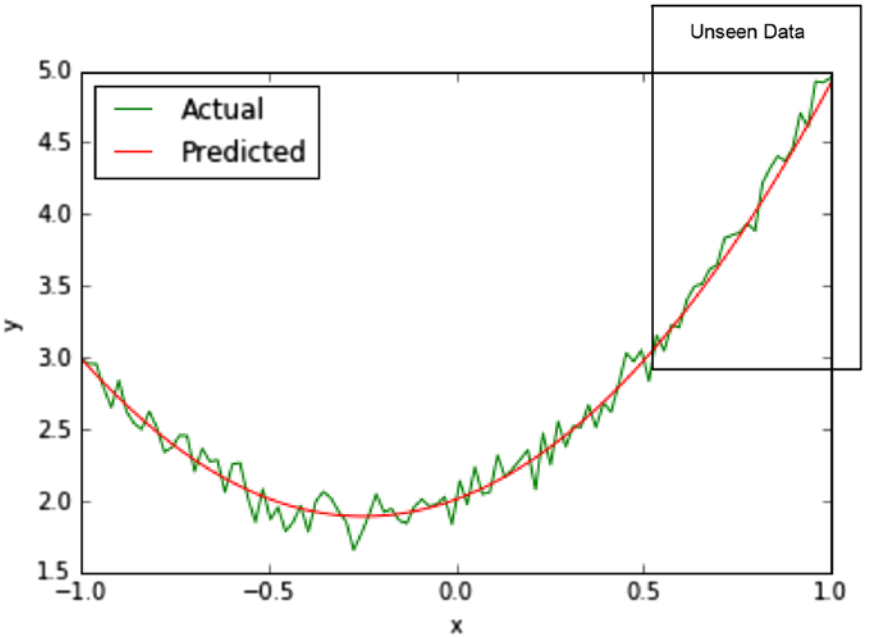


Figure 2-2. *Actual and predicted values for model with degree = 2*

Next, we generate another model with the least squares algorithm but we will transform x to $[x^0, x^1, x^2, x^3, x^4, x^5, x^6, x^7, x^8]$. That is, we are approximating the given data with a polynomial with degree = 8. The actual and predicted values are plotted in Figure 2-3 and listing 2-1 shows the source code for generating the model. As the last step we generate a model with degree = 1.

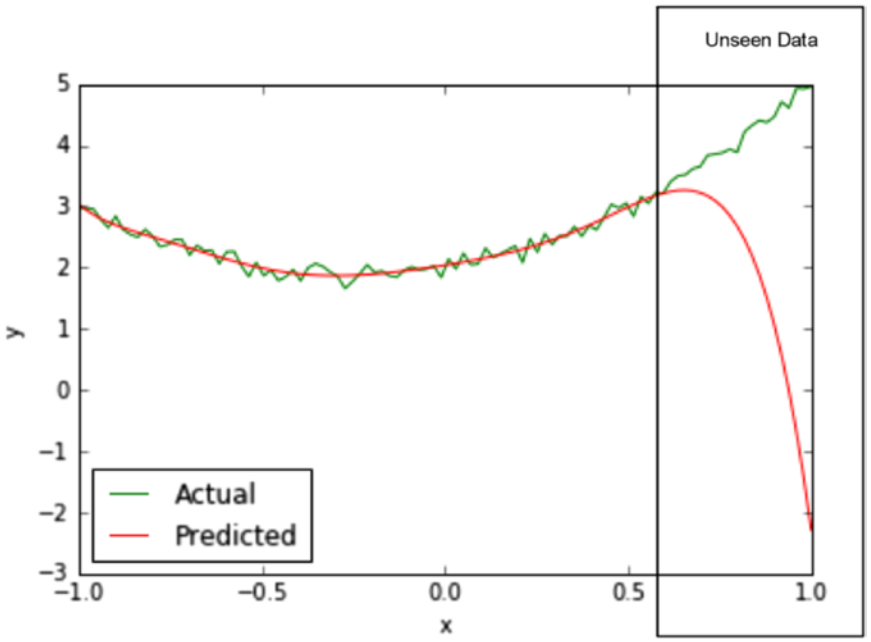


Figure 2-3. *Actual and predicted values for model with degree = 8*

The actual and predicted values are plotted in Figure 2-4 and listing 2-1 shows the source code for generating the model.

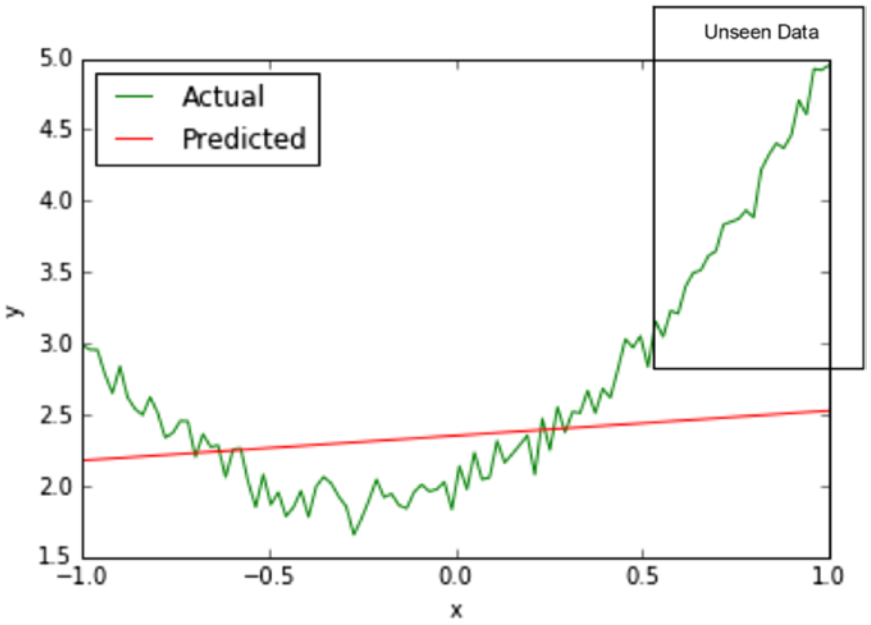


Figure 2-4. *Actual and predicted values for model with degree = 1*

We now have all the details in place to discuss the core concept of generalization. The key question to ask is which is the better model? The one with degree = 2 or the one with degree = 8 or the one with degree = 1?

Let us start by making a few observations about the three models. The model with degree = 1 performs poorly on both the seen as well as unseen data as compared to the other two models. The model with degree = 8 performs better on seen data as compared to the model with degree = 2. The model with degree = 2 performs better than the model with degree = 8 on unseen data. Table 2-1 visualizes this in table form for easy interpretation.

Table 2-1. *Comparing the performance of the 3 different models*

Degree	1	2	8
Seen Data	Worst	Worst	Better
Unseen Data	Worst	Better	Worst

Let us now understand the important concept of model capacity, which corresponds to the degree of the polynomial in this example. The data we generated was using a second order polynomial (degree = 2) with some noise. Then, we tried to approximate the data using three models of degree: 1, 2, and 8, respectively. The higher the degree, the more expressive is the model. That is, it can accommodate more variation. This ability to accommodate variation corresponds to the notion of capacity. That is, we say that the model with degree = 8 has a higher capacity that the model with degree = 2, which in turn has a higher capacity than the model with degree = 1. Isn't having higher capacity always a good thing? It turns out it is not, when we consider that all real world datasets contain some noise and a higher capacity model will end up just fitting the noise in addition to the signal in the data. This is why we observe that the model with degree = 2 does better on the unseen data as compared to the model with degree = 8. In this example, we knew how the data was generated (with a second order polynomial (degree = 2) with some noise); hence, this observation is quite trivial. However, in the real world, we don't know the underlying mechanism by which the data is generated. This leads us to the fundamental challenge in machine learning, which is, does the model truly generalize? And the only true test for that is the performance over unseen data.

In a sense the concept of capacity corresponds to the simplicity or parsimony of the model. A model with high capacity can approximate more complex data. This is how many free variables/coefficients the model has. In our example, the model with degree = 1 does not have capacity sufficient to approximate the data and this is commonly referred to as under fitting. Correspondingly, the model with degree = 8 has extra capacity and it over fits the data.

As a thought experiment, consider what would happen if we had a model with degree equal to 80. Given that we had 80 data points as training data, we would have an 80-degree polynomial that would perfectly approximate the data. This is the ultimate pathological case wherein there is no learning at all. The model has 80 coefficients and can simply memorize the data. This is referred to as rote learning, the logical extreme of overfitting. This is why the capacity of the model needs to be tuned with respect to the amount of training data we have. If the dataset is small, we are better off training models with lower capacity.

Regularization

Building on the idea of model capacity, generalization, over fitting, and under fitting, let us now cover the idea of regularization. The key idea here is to penalize complexity of the model. A regularized version of least squares takes the form $y = \beta x$, where β is a vector such that $\|X\beta - y\|_2^2 + \lambda \|\beta\|_2^2$ is minimized and λ is a user-defined parameter that controls the complexity. Here, by introducing the term $\lambda \|\beta\|_2^2$, we are penalizing complex models. To see why this is the case, consider fitting a least square model using a polynomial of degree 10, but the values in the vector β has 8 zeros; 2 are non-zeros. Against this, consider the case where

all values in the vector β are non-zeros. For all practical purposes, the former model is a model with degree = 2 and has a lower value of $\lambda \|\beta\|_2^2$. The λ term allows us to balance accuracy over the training data with the complexity of the model. Lower values of λ imply a simpler model.

We can compute the value of β using the closed form $\beta = \left(X^T X - \lambda I\right)^{-1} X^T y$. We illustrate keeping the degree fixed at a value of 80 and varying the value of λ in listing 2-2.

Listing 2-2. Regularization

```
import pylab
import numpy
x = numpy.linspace(-1,1,100)
signal = 2 + x + 2 * x * x
noise = numpy.random.normal(0, 0.1, 100)
y = signal + noise
x_train = x[0:80]
y_train = y[0:80]

train_rmse = []
test_rmse = []
degree = 80
lambda_reg_values = numpy.linspace(0.01,0.99,100)

for lambda_reg in lambda_reg_values:
    X_train = numpy.column_stack([numpy.power(x_train,i) for i in xrange(0,degree)])
    model = numpy.dot(numpy.dot(numpy.linalg.inv(numpy.dot(X_train.transpose(),X_train) +
lambda_reg * numpy.identity(degree)),X_train.transpose()),y_train)
    predicted = numpy.dot(model, [numpy.power(x,i) for i in xrange(0,degree)])
    train_rmse.append(numpy.sqrt(numpy.sum(numpy.dot(y[0:80] - predicted[0:80],
y_train - predicted[0:80]))))
    test_rmse.append(numpy.sqrt(numpy.sum(numpy.dot(y[80:] - predicted[80:],
y[80:] - predicted[80:]))))

pylab.plot(lambda_reg_values, train_rmse)
pylab.plot(lambda_reg_values, test_rmse)
pylab.xlabel(r"$\lambda$")
pylab.ylabel("RMSE")
pylab.legend(["Train", "Test"], loc = 2)
```

The training RMSE (seen data) and test RMSE (unseen data) is plotted in Figure 2-5.

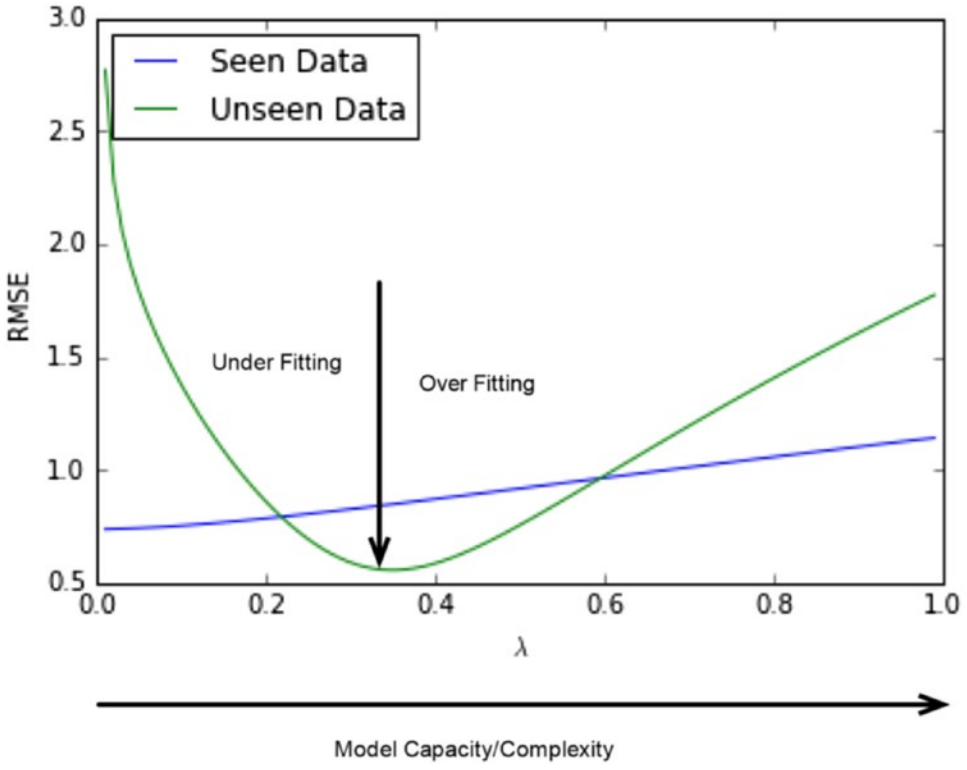


Figure 2-5. *Regularization*

Summary

In this chapter we covered the basics of Machine Learning. The key take-home points for this chapter are the concepts of generalizing over unseen examples, over-fitting and under-fitting the training data, the capacity of the model, and the notion of regularization. The reader is encouraged to try out the examples (in the source code listings). In the next chapter, we will build on these concepts and cover neural networks.

CHAPTER 3

Feed Forward Neural Networks

In this chapter we will cover some key concepts around feedforward neural networks. These concepts will serve as a foundation as we cover more technical topics in depth.

At an abstract level, a Neural Network can be thought of as a function $f_\theta : x \to y$, which takes an input $x \in \mathbb{R}^n$ and produces an output $y \in \mathbb{R}^m$, and whose behavior is parameterized by $\theta \in \mathbb{R}^p$. So for instance, f_θ could be simply $y = f_\theta(x) = \theta \cdot x$.

We will start by looking at the structure of a neural network, followed by how they are trained and used for making predictions.

Unit

A unit is the basic building of a neural network; refer to Figure 3-1. The following points should be noted:

1. A unit is a function that takes as input a vector $x \in \mathbb{R}^n$ and produces a scalar.
2. A unit is parameterized by a weight vector $w \in \mathbb{R}^n$ and a bias term denoted by b.
3. The output of the unit can be described as

$$f\left(\sum_{i=1}^{n} x_i \cdot w_i + b\right)$$

   where $f : \mathbb{R} \to \mathbb{R}$ is referred to as an activation function.
4. A variety of activation functions may be used, as we shall see later in the chapter; generally, it's a non-linear function.

N. Ketkar, *Deep Learning with Python*, DOI 10.1007/978-1-4842-2766-4_3

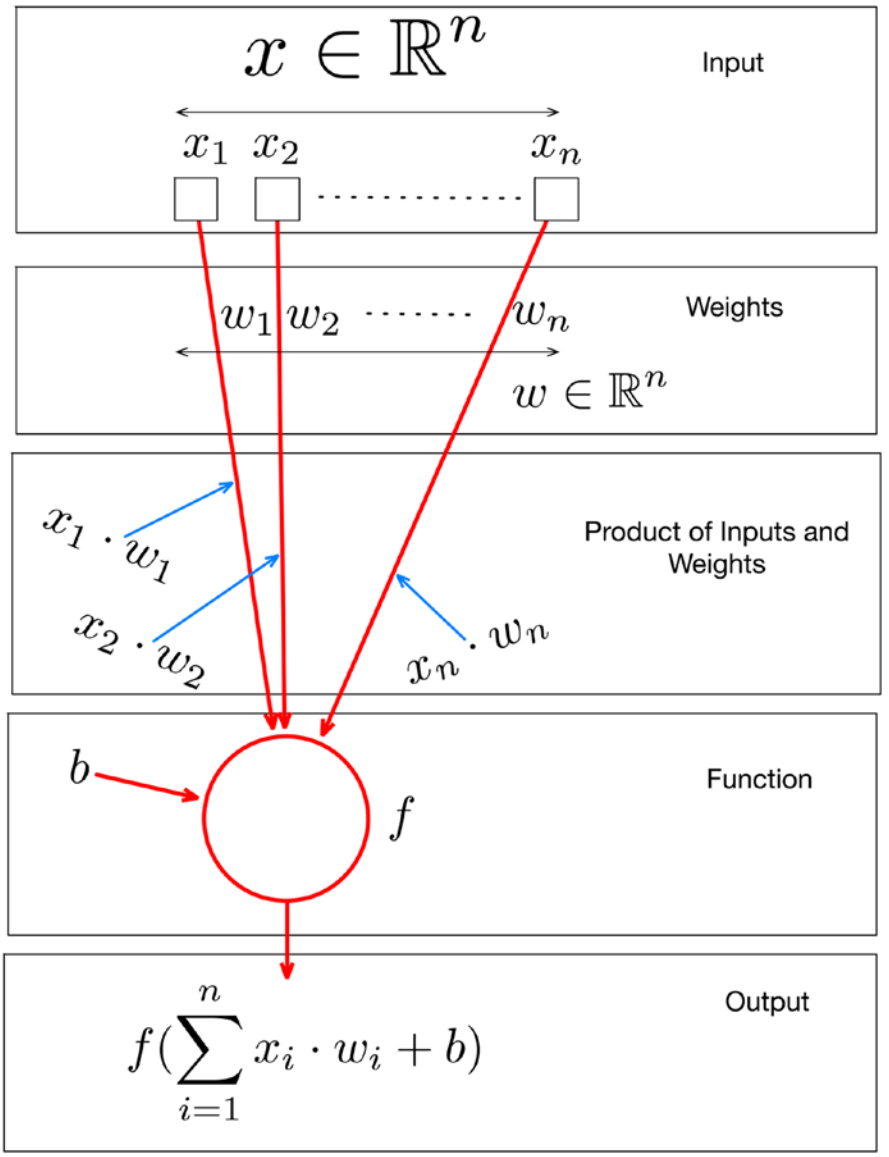


Figure 3-1. *A unit in a neural network*

Overall Structure of a Neural Network

Neural Networks are constructed using the unit as a basic building block (introduced earlier); refer to Figure 3-2.

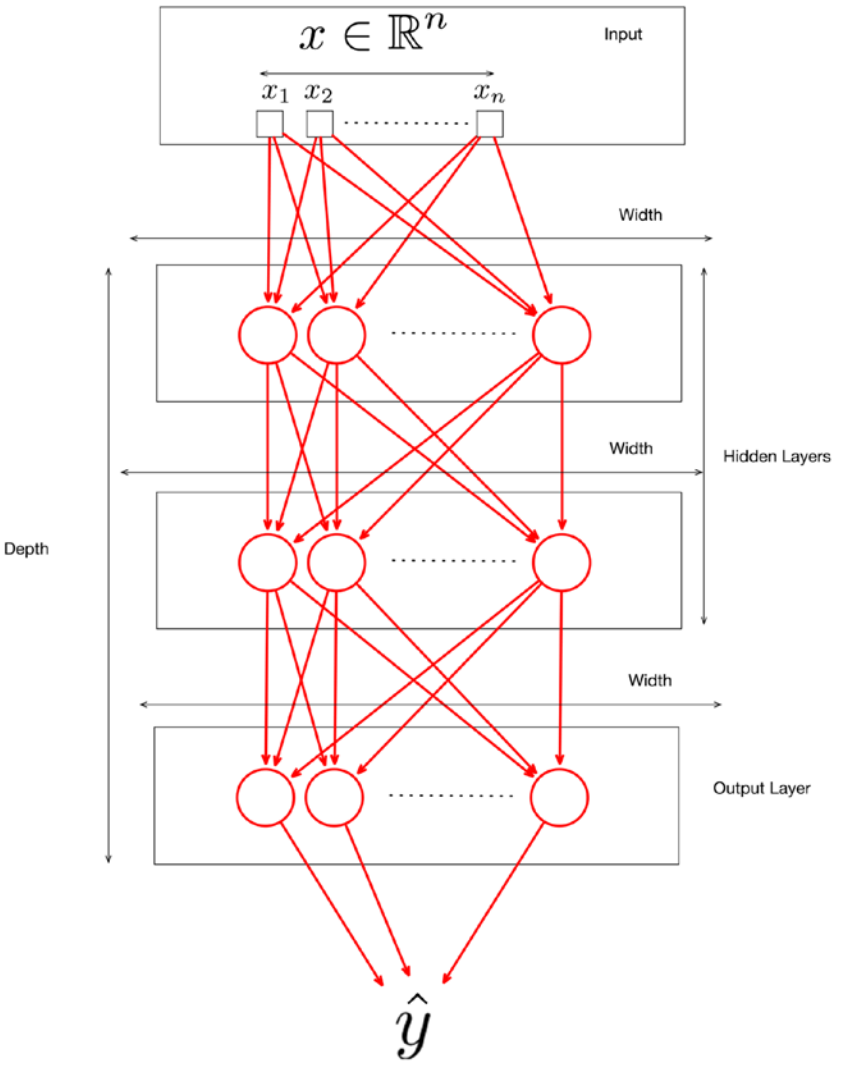


Figure 3-2. *Structure of a Neural Network*

The following points are to be noted:

1. Units are organized as layers, with every layer containing one or more units.
2. The last layer is referred to as the output layer.
3. All layers before the output layers are referred to as hidden layers.
4. The number of units in a layer is referred to as the width of the layer.

5. The width of each layer need not be the same, but the dimension should be aligned, as we shall see later in the chapter.
6. The number of layers is referred to as the depth of the network. This is where the notion of *deep* (as in deep learning) comes from.
7. Every layer takes as input the output produced by the previous layer, except for the first layer, which consumes the input.
8. The output of the last layer is the output of the network and is the prediction generated based on the input.
9. As we have seen earlier, a neural network can be seen as a function $f_\theta : x \rightarrow y$, which takes as input $x \in \mathbb{R}^n$ and produces as output $y \in \mathbb{R}^m$ and whose behavior is parameterized by $\theta \in \mathbb{R}^p$. We can now be more precise about θ; it's simply a collection of all the weights w for all the units in the network.
10. Designing a neural network involves, amongst other things, defining the overall structure of the network, including the number of layers and the width of these layers.

Expressing the Neural Network in Vector Form

Let us now take a look at the layers of a Neural Network and their dimensions in a bit more detail. Refer to Figure 3-3; the following points should be noted:

1. If we assume that the dimensionality of the input is $x \in \mathbb{R}^n$ and the first layer has p_1 units, then each of the units has $w \in \mathbb{R}^n$ weights associated with them. That is, the weights associated with the first layer are a matrix of the form $w_1 \in \mathbb{R}^{n \times p_1}$. While this is not shown in the diagram, each of the p_1 units also has a bias term associated with it.
2. The first layer produces an output $o_1 \in \mathbb{R}^{p_1}$ where $o_i = f\left(\sum_{k=1}^{n} x_k \cdot w_k + b_i\right)$. Note that the index k corresponds to each of the inputs/weights (going from $1 \ldots n$) and the index i corresponds to the units in the first layer (going from $1..p_1$).
3. Let us now look at the output of the first layer in a vectorised notation. By vectorised notation we simply mean linear algebraic operations like vector matrix multiplications and computation of the activation function on a vector producing a vector (rather than scalar to scalar). The output of the first layer can be represented as $f(x \cdot w_1 + b_1)$. Here we are treating the input $x \in \mathbb{R}^n$ to be of dimensionality $1 \times n$, the weight matrix w_1 to be of dimensionality $n \times p_1$, and the bias term to be a vector of dimensionality $1 \times p_1$. Notice then that $x \cdot w_1 + b$ produces a vector of dimensionality $1 \times p_1$ and the function f simply transforms each element of the vector to produce $o_1 \in \mathbb{R}^{p_1}$.
4. A similar process follows for the second layer that goes from $o_1 \in \mathbb{R}^{p_1}$ to $o_2 \in \mathbb{R}^{p_2}$. This can be written in vectorised form as $f(o_1 \cdot w_2 + b_2)$. We can also write the entire computation up to layer 2 in vectorised form as $f(f(x \cdot w_1 + b_1) \cdot w_2 + b_2)$.

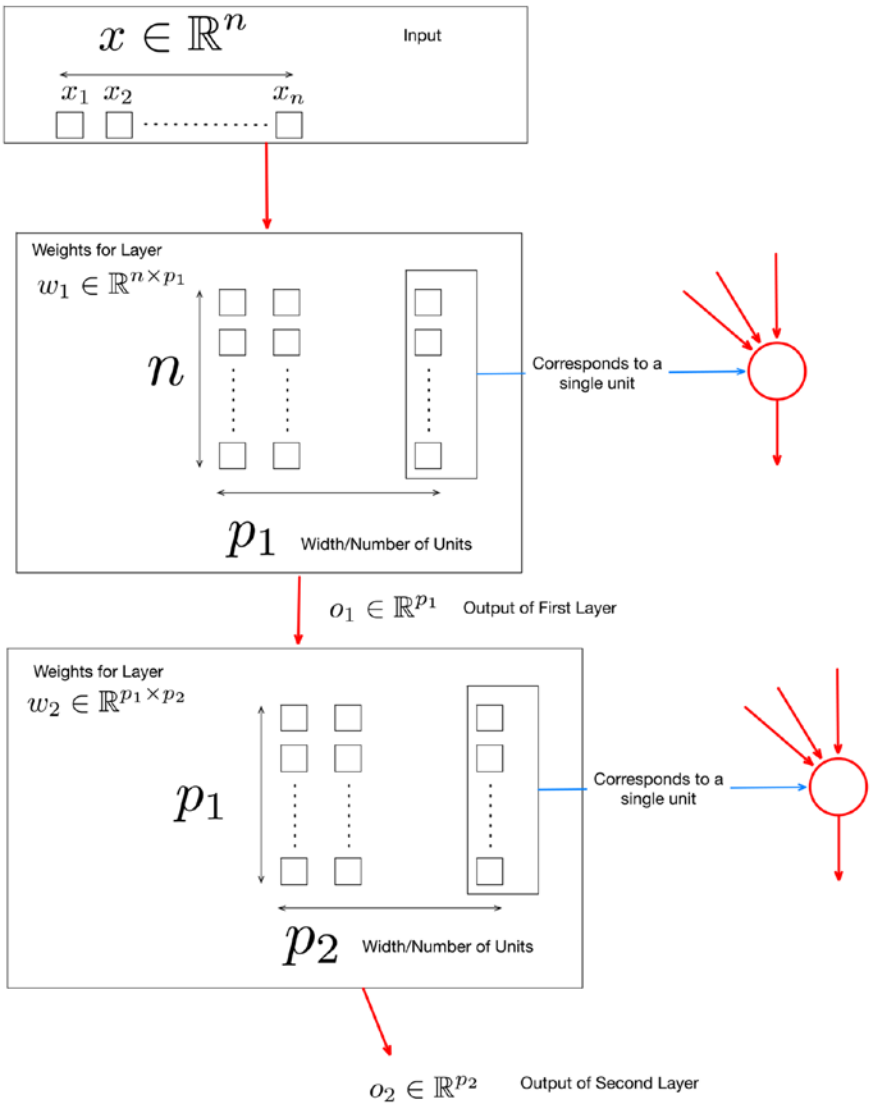


Figure 3-3. *Expressing the Neural Network in Vector Form*

Evaluating the output of the Neural Network

Now that we have looked at the structure of a Neural Network, let's look at how the output of the neural network can be evaluated against labeled data. Refer to Figure 3-4. The following points are to be noted:

1. We can assume that our data has the form $D = \{(x_1, y_1), (x_2, y_2), \ldots (x_n, y_n)\}$ where $x \in \mathbb{R}^n$ and $y \in \{0, 1\}$, which is the target of interest (currently this is binary, but it may be categorical or real valued depending on whether we are dealing with a multi-class or regression problem, respectively).
2. For a single data point we can compute the output of the Neural Network, which we denote as $\hat{y}$.

3. Now we need to compute how good the prediction of our Neural Network $\hat{y}$ is as compared to y. Here comes the notion of a loss function.
4. A loss function measures the disagreement between $\hat{y}$ and y which we denote by l. There are a number of loss functions appropriate for the task at hand: binary classification, multi-classification, or regression, which we shall cover later in the chapter (typically derived using Maximum Likelihood).
5. A loss function typically computes the disagreement between $\hat{y}$ and y over a number of data points rather than a single data point.

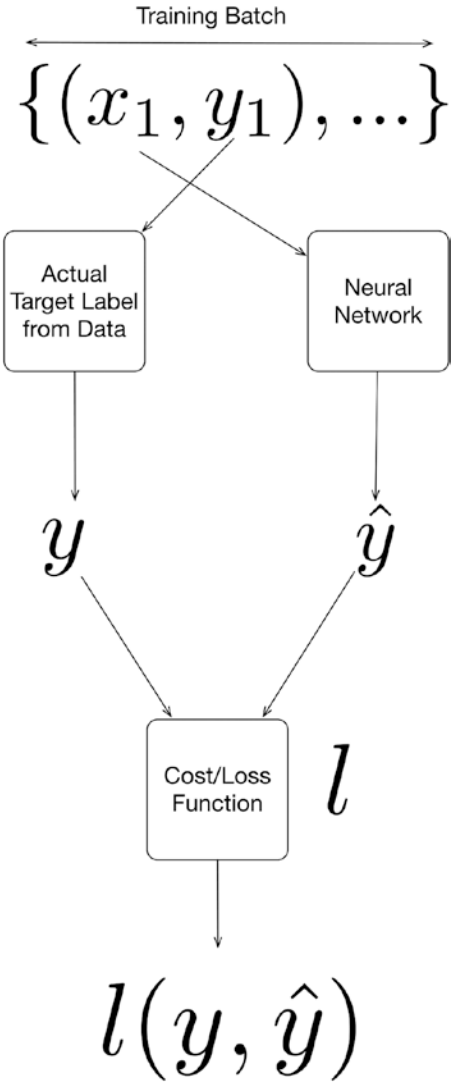


Figure 3-4. *Loss/Cost function and the computation of cost/loss w.r.t, a neural network*

Training the Neural Network

Let us now look at how the Neural Network is Trained. Refer to Figure 3-5. The following points are to be noted:

1. Assuming the same notation as earlier, we denote by θ the collection of all the weights and bias terms of all the layers of the network. Let us assume that θ has been initialized with random values. We denote by f_{NN} the overall function representing the Neural Network.
2. As we have seen earlier, we can take a single data point and compute the output of the Neural Network $\hat{y}$. We can also compute the disagreement with the actual output y using the loss function $l(\hat{y}, y)$ that is $l(f_{NN}(x, \theta), y)$.
3. Let us now compute the gradient of this loss function and denote it by $\nabla l\left(f_{NN}(x,\theta), y\right)$.
4. We can now update θ using steepest descent as $\theta_s = \theta_{s-1} - \alpha \cdot l\left(f_{NN}(x,\theta), y\right)$ where s denotes a single step (we can take many such steps over different data points in our training set over and over again until we have a reasonably good value for $l(f_{NN}(x, \theta), y)$.

■ **Note** For now, we will stay away from the computation of gradients of loss functions $\nabla l\left(f_{NN}(x,\theta), y\right)$. These can be generated using automatic differentiation (covered elsewhere in the book) quite easily (even for arbitrary complicated loss functions) and need not be derived manually. Also, the method of steepest descent and stochastic gradient descent is covered in a separate chapter.

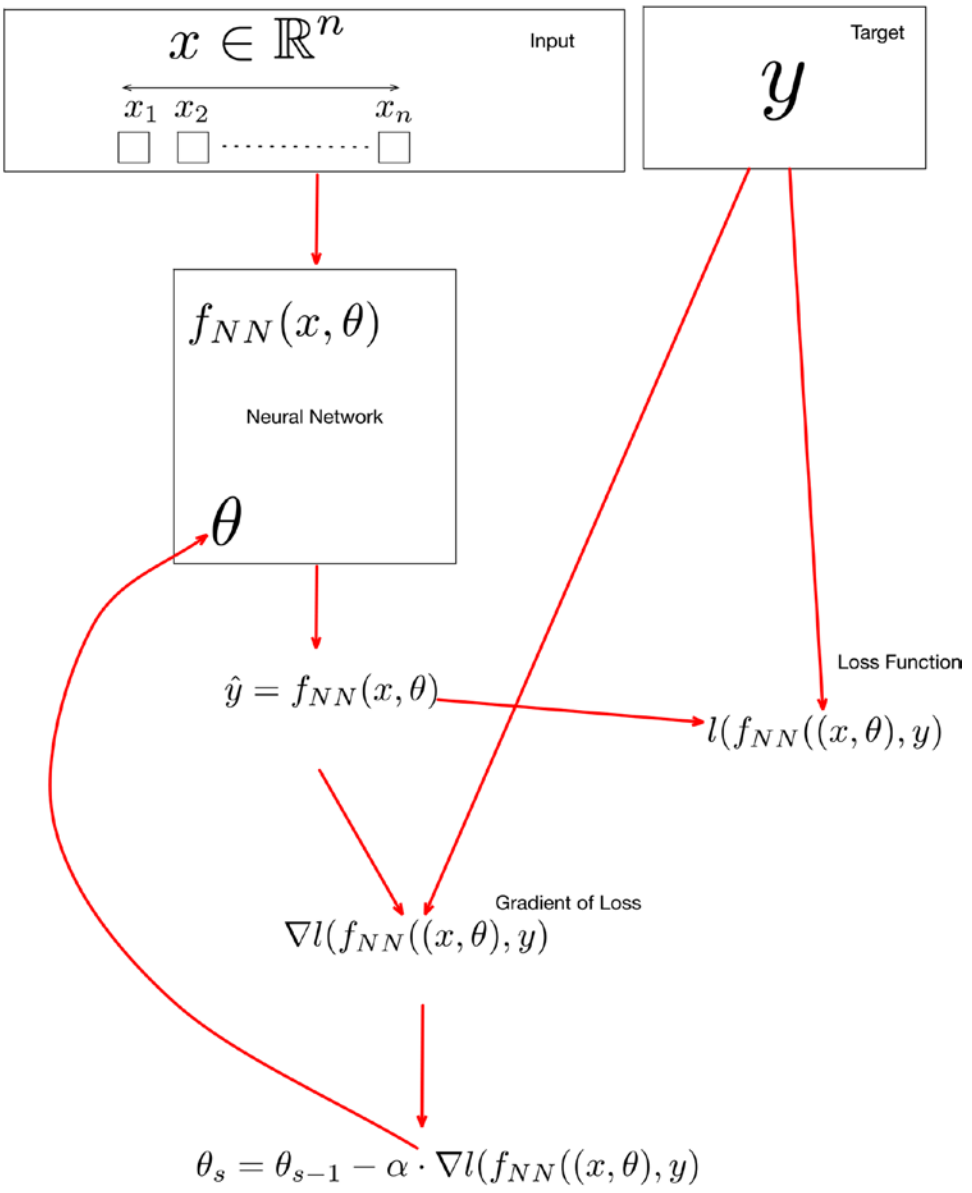


Figure 3-5. *Training a Neural Network*

Deriving Cost Functions using Maximum Likelihood

We will now look into how loss functions are derived using Maximum Likelihood. Specifically, we will see how commonly used loss functions in deep learning like binary cross entropy, cross entropy, and squared error can be derived using the Maximum Likelihood principle.

Binary Cross Entropy

Instead of starting with the general idea of Maximum Likelihood, let's directly jump to the binary classification problem. We have some data consisting of $D = \{(x_1, y_1), (x_2, y_2), \ldots (x_n, y_n)\}$ where $x \in \mathbb{R}^n$ and $y \in \{0, 1\}$, which is the target of interest (currently this is binary, but it may be categorical or real valued depending on whether we are dealing with a multi-class or regression problem, respectively).

Let us assume that we have somehow generated a model that predicts the probability of *y* given *x*. Let us denote this model by $f(x, \theta)$ where θ represents the parameters of the model. The idea behind Maximum Likelihood is to find a θ that maximizes $P(D|\theta)$. Assuming a Bernoulli distribution and given that each of the examples $\{(x_1, y_1), (x_2, y_2), \ldots (x_n, y_n)\}$ are independent, we have the following expression:

$$P(D|\theta) = \prod_{i=1}^{n} f(x_i, \theta)^{y_i} \cdot \left(1 - f(x_i, \theta)\right)^{(1-y_i)}$$

We can take a logarithm operation on both sides to arrive at the following:

$$\log P(D|\theta) = \log \prod_{i=1}^{n} f(x_i, \theta)^{y_i} \cdot \left(1 - f(x_i, \theta)\right)^{(1-y_i)}$$

which simplifies to the expression below:

$$\log P(D|\theta) = \sum_{i=1}^{n} y_i \log f(x_i, \theta) + (1 - y_i) \log\left(1 - f(x_i, \theta)\right)$$

Instead of maximizing the RHS, we minimize its negative value as follows:

$$-\log P(D|\theta) = -\sum_{i=1}^{n} y_i \log f(x_i, \theta) + (1 - y_i) \log\left(1 - f(x_i, \theta)\right)$$

This leads us to the binary cross entropy function as below:

$$-\sum_{i=1}^{n} y_i \log f(x_i, \theta) + (1 - y_i) \log\left(1 - f(x_i, \theta)\right)$$

The idea of Maximum Likelihood thus allows us to derive the binary cross entropy function which can be used as a loss function in the context of binary classification.

Cross Entropy

Building on the idea of binary cross entropy, let us now consider deriving the cross entropy loss function to be used in the context of multi-classification. Let us assume in this case that $y \in \{0, 1, ..k\}$, which are the classes. We also denote $n_1, n_2 \cdots n_k$ to be the observed counts of each of the *k* classes. Observe that $\sum_{i=1}^{k} n_i = n$.

In this case, too, let us assume that we have somehow generated a model that predicts the probability of *y* given *x*. Let us denote this model by $f(x, \theta)$ where θ represents the parameters of the model. Let us again use the idea behind Maximum Likelihood, which is to find a θ that maximizes $P(D|\theta)$. Assuming a Multinomial distribution and given that each of the examples $\{(x_1, y_1), (x_2, y_2), \ldots (x_n, y_n)\}$ are independent, we have the following expression:

$$P(D|\theta) = \frac{n!}{n_1! \cdot n_2! \cdots n_k!} \prod_{i=1}^{n} f(x_i, \theta)^{y_i}$$

We can take a logarithm operation on both sides to arrive at the following:

$$\log P(D|\theta) = \log n! - \log n_1! \cdot n_2! \cdots n_k! + \log \prod_{i=1}^{n} f(x_i, \theta)^{y_i}$$

This can be simplified to the following:

$$\log P(D|\theta) = \log n! - \log n_1! \cdot n_2! \cdots n_k! + \sum_{i=1}^{n} y_i \log f(x_i, \theta)$$

The terms $\log n!$ and $\log n_1! \cdot n_2! \cdots n_k!$ are not parameterized by θ and can be safely ignored as we try to find a θ that maximizes $P(D|\theta)$. Thus we have the following:

$$\log P(D|\theta) = \sum_{i=1}^{n} y_i \log f(x_i, \theta)$$

As before, instead of maximizing the RHS we minimize its negative value as follows:

$$-\log P(D|\theta) = -\sum_{i=1}^{n} y_i \log f(x_i, \theta)$$

This leads us to the binary cross entropy function as below:

$$-\sum_{i=1}^{n} y_i \log f(x_i, \theta)$$

The idea of Maximum Likelihood thus allows us to derive the cross entropy function, which can be used as a loss function in the context of multi-classification.

Squared Error

Let us now look into deriving the squared error to be used in the context of regression using Maximum Likelihood. Let us assume in this case that $y \in \mathbb{R}$. Unlike the previous cases where we assumed that we had a model that predicted a probability, here we will assume that we have a model that predicts the value of *y*. To apply the Maximum Likelihood idea, we assume that the difference between the actual *y* and the predicted $\hat{y}$ has a Gaussian distribution with zero mean and a variance of σ^2. Then it can be shown that minimizing

$$\sum_{i=1}^{n} \left(y - \hat{y}\right)^2$$

leads to the minimization of $-\log P(D|\theta)$.

Summary of Loss Functions

We now summarize three key points with respect to loss functions and the appropriateness of a particular loss function given the problem at hand.

1. The Binary Cross entropy given by the expression

$$-\sum_{i=1}^{n} y_i \log f(x_i, \theta) + (1 - y_i) \log\left(1 - f(x_i, \theta)\right)$$

   is the recommended loss function for binary classification. This loss function should typically be used when the Neural Network is designed to predict the probability of the outcome. In such cases, the output layer has a single unit with a suitable sigmoid as the activation function.

2. The Cross entropy function given by the expression

$$-\sum_{i=1}^{n} y_i \log f(x_i, \theta)$$

   is the recommended loss function for multi-classification. This loss function should typically be used with the Neural Network and is designed to predict the probability of the outcomes of each of the classes. In such cases, the output layer has softmax units (one for each class).

3. The squared loss function given by $\sum_{i=1}^{n} \left(y - \hat{y}\right)^2$ should be used for regression problems. The output layer in this case will have a single unit.

Types of Units/Activation Functions/Layers

We will now look at a number of Units/Activation Functions/Layers commonly used for Neural Networks.

Let's start by enumerating a few properties of interest for activation functions.

1. In theory, when an activation function is non-linear, a two-layer Neural Network can approximate any function (given a sufficient number of units in the hidden layer). Thus, we do seek non-linear activation functions in general.

2. A function that is continuously differentiable allows for gradients to be computed and gradient-based methods to be used for finding the parameters that minimize our loss function over the data. If a function is not continuously differentiable, gradient-based methods cannot make progress.

3. A function whose range is finite (as against infinite) leads to a more stable performance w.r.t gradient-based methods.

4. Smooth functions are preferred (empirical evidence) and Monolithic functions for a single layer lead to convex error surfaces (this is typically not a consideration w.r.t deep learning).

5. Are symmetric around the origin and behave like identity functions near the origin ($f(x) = x$).

Linear Unit

The Linear unit is the simplest unit which transforms the input as $y = w.x + b$. As the name indicates, the unit does not have a non-linear behavior and is typically used to generate the mean of a conditional Gaussian distribution. Linear units make gradient-based learning a fairly straightforward task.

Sigmoid Unit

The Sigmoid unit transforms the input as follows:

$$y = \frac{1}{1+e^{-(wx+b)}}.$$

The underlying activation function (refer to Figure 3-6) is given by

$$f(x) = \frac{1}{1+e^{-x}}.$$

Sigmoid units can be used in the output layer in conjunction with binary cross entropy for binary classification problems. The output of this unit can model a Bernoulli distribution over the output y conditioned over x.

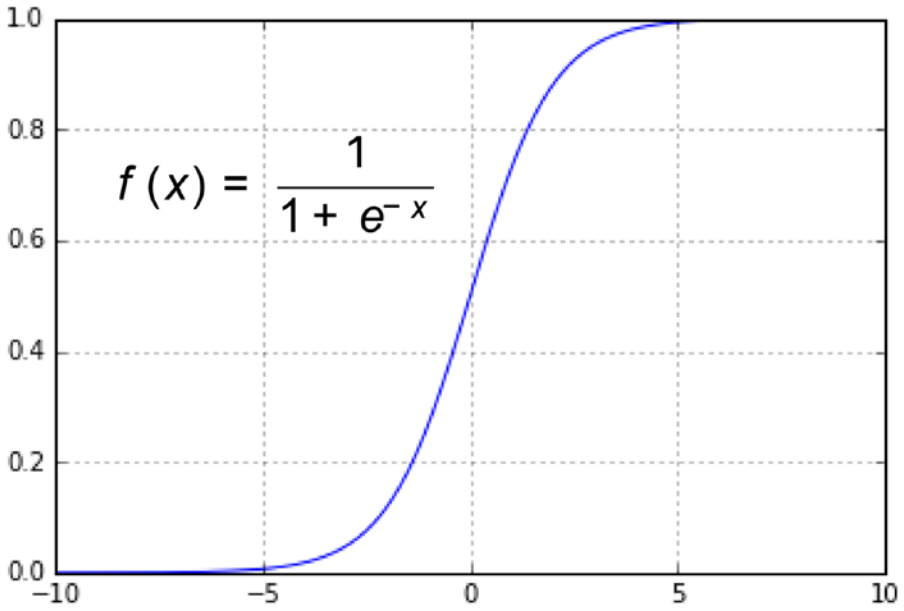


Figure 3-6. *Sigmoid Function*

Softmax Layer

The Softmax layer is typically used as an output layer for multi-classification tasks in conjunction with the Cross Entropy loss function. Refer to Figure 3-7. The Softmax layer normalizes outputs of the previous layer so that they sum up to one. Typically, the units of the previous layer model an un-normalized score of how likely the input is to belong to a particular class. The softmax layer normalized this so that the output represents the probability for every class.

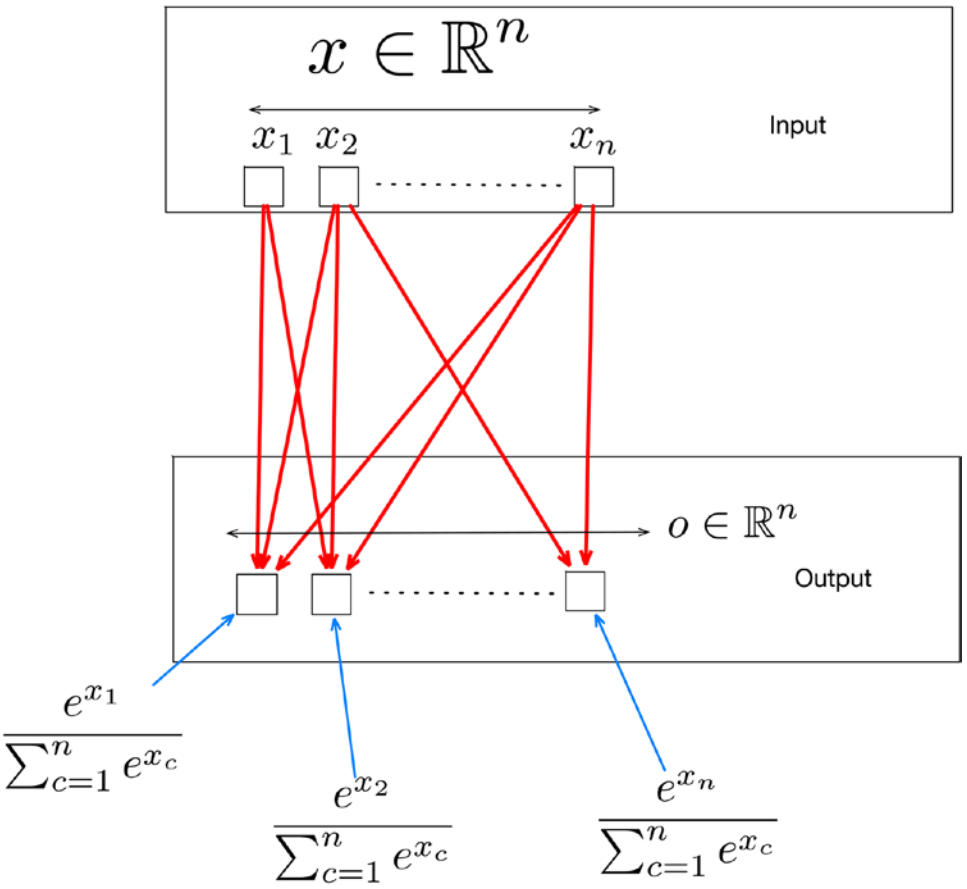


Figure 3-7. *Softmax Layer*

Rectified Linear Unit (ReLU)

Rectified Linear Unit used in conjunction with a linear transformation transforms the input as

$$f(x) = \max(0, wx + b) .$$

The underlying activation function is $f(x) = \max(0, x)$; refer to Figure 3-8. The ReLU unit is more commonly used as a hidden unit in recent times. Results show that ReLU units lead to large and consistent gradients, which helps gradient-based learning.

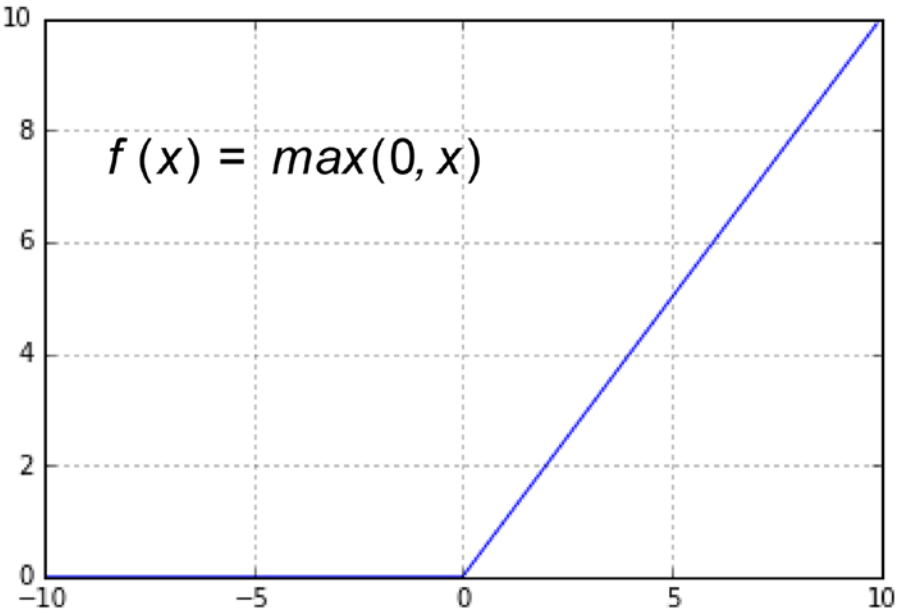


Figure 3-8. *ReLU*

Hyperbolic Tangent

The Hyperbolic Tangent unit transforms the input (used in conjunction with a linear transformation) as follows:

$$y = \tanh(wx + b).$$

The underlying activation function (refer to Figure 3-9) is given by

$$f(x) = \tanh(x).$$

The hyperbolic tangent unit is also commonly used as a hidden unit.

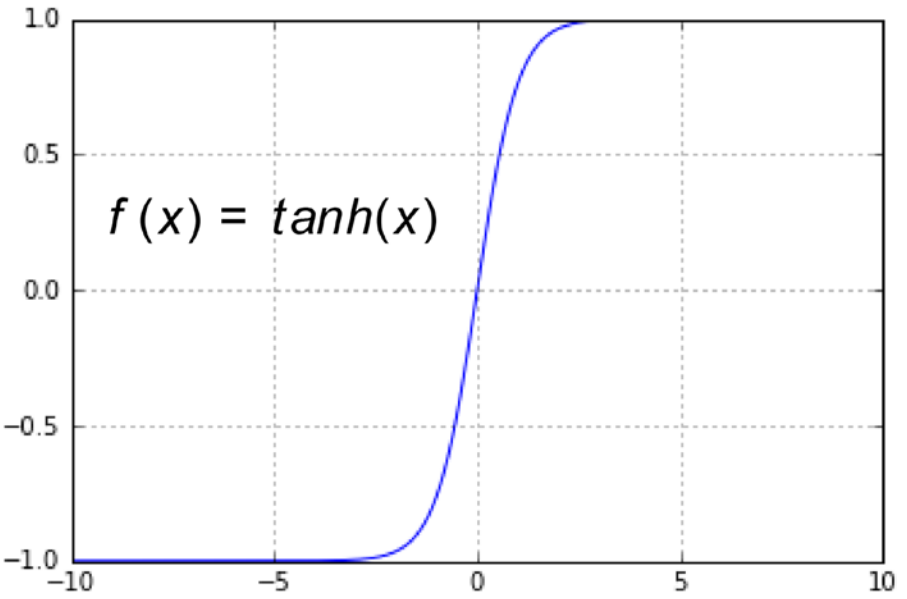


Figure 3-9. *The tanh activation function*

Listing 3-1. A 2-Layer Neural Network for Regression

```
import autograd.numpy as np
import autograd.numpy.random as npr
from autograd import grad
import sklearn.metrics
import pylab

# Generate Dataset
examples = 1000
features = 100
D = (npr.randn(examples, features), npr.randn(examples))

# Specify the network
layer1_units = 10
layer2_units = 1
w1 = npr.rand(features, layer1_units)
b1 = npr.rand(layer1_units)
w2 = npr.rand(layer1_units, layer2_units)
b2 = 0.0
theta = (w1, b1, w2, b2)

# Define the loss function
def squared_loss(y, y_hat):
    return np.dot((y - y_hat),(y - y_hat))

# Output Layer
def binary_cross_entropy(y, y_hat):
    return np.sum(-((y * np.log(y_hat)) + ((1-y) * np.log(1 - y_hat))))
```

```
# Wraper around the Neural Network
def neural_network(x, theta):
    w1, b1, w2, b2 = theta
    return np.tanh(np.dot((np.tanh(np.dot(x,w1) + b1)), w2) + b2)

# Wrapper around the objective function to be optimised
def objective(theta, idx):
    return squared_loss(D[1][idx], neural_network(D[0][idx], theta))

# Update
def update_theta(theta, delta, alpha):
    w1, b1, w2, b2 = theta
    w1_delta, b1_delta, w2_delta, b2_delta = delta
    w1_new = w1 - alpha * w1_delta
    b1_new = b1 - alpha * b1_delta
    w2_new = w2 - alpha * w2_delta
    b2_new = b2 - alpha * b2_delta
    new_theta = (w1_new,b1_new,w2_new,b2_new)
    return new_theta

# Compute Gradient
grad_objective = grad(objective)

# Train the Neural Network
epochs = 10
print "RMSE before training:", sklearn.metrics.mean_squared_error(D[1],neural_network(D[0],
theta))
rmse = []
for i in xrange(0, epochs):
    for j in xrange(0, examples):
        delta = grad_objective(theta, j)
        theta = update_theta(theta,delta, 0.01)

rmse.append(sklearn.metrics.mean_squared_error(D[1],neural_network(D[0], theta)))
print "RMSE after training:", sklearn.metrics.mean_squared_error(D[1],neural_network(D[0],
theta))

pylab.plot(rmse)
pylab.show()

#Output
#RMSE before training: 1.88214665439
#RMSE after training: 0.739508975012
```

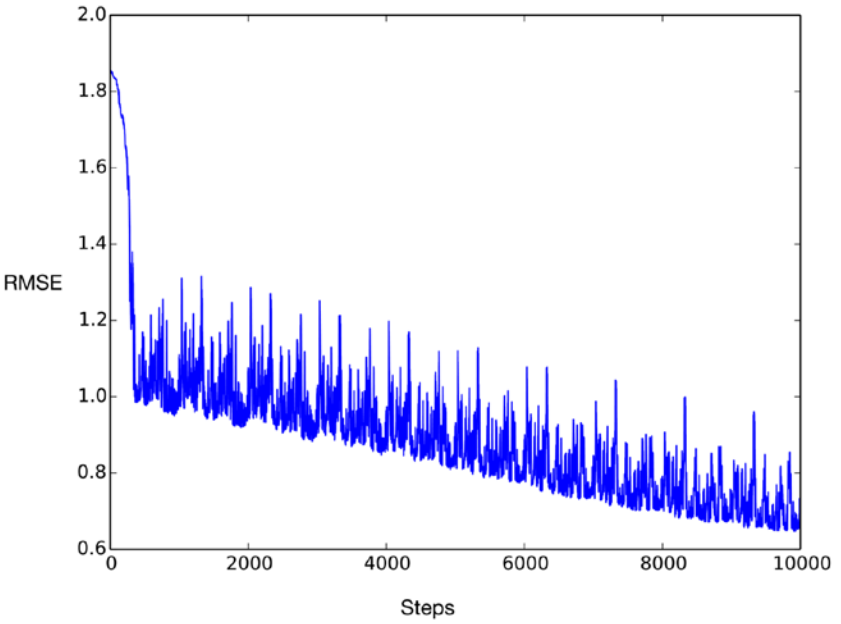


Figure 3-10. *RMSE over training steps*

Neural Network Hands-on with AutoGrad

We will now build a simple Neural Network from scratch (refer to Listing 3-1). The only external library we will be using is Autograd. Autograd is an automatic differentiation library that will allow us to compute gradients for arbitrary functions written with Numpy.

■ **Note** Autograd is covered in more detail in the chapter on automatic differentiation.

Summary

In this chapter we covered feed forward neural networks, which will serve as the conceptual foundation for the rest of the chapters. The key concepts we covered were the overall structure of the neural network, the input, hidden and output layers, cost functions, and their basis on the principle of Maximum Likelihood. We encourage the reader to try out the example in the source code listing; although it is a toy example, it will help reinforce the concepts. The next chapter will provide a hands-on introduction to the reader on Theano, which will enable the reader to implement full-fledged neural networks.

CHAPTER 4

Introduction to Theano

In this chapter we introduce the reader to Theano, which is a Python library for defining mathematical functions (operating over vectors and matrices), and computing the gradients of these functions. Theano is the foundational layer on which many deep learning packages like Keras are based.

What is Theano

As we have seen before, building deep learning models fundamentally involves optimizing loss functions using stochastic gradient descent (SGD), requiring the computation of the gradient of loss function. As loss functions in deep learning are complicated, it is not convenient to manually derive such gradients. This is where Theano comes in handy. Theano allows the user to define mathematical expressions that encode loss functions and, once these are defined, Theano allows the user to compute the gradients of these expressions.

A typical workflow for using Theano is as follows:

1. Write symbolic expressions in Python that implement the loss function of the model to be built. This usually amounts to only a few lines of code because of the expressiveness of the Python language and a tight integration with Numpy that allows the user to quickly and elegantly define mathematical expressions involving vectors and matrices.
2. Use the symbolic/automatic differentiation capabilities in Theano to generate an expression that produces the gradient of the loss function.
3. Pass this gradient function as a parameter to a SGD optimization routine to optimize the loss function.

Theano allows a user to focus on the model rather than on the mechanics of deriving the gradient. The specific capabilities of Theano that make it a great toolbox for building deep learning models are as follows:

1. a very seamless integration with Numpy that allows the user to use Numpy objects (vectors and matrices) in the definition of loss functions
2. Theano can generate optimized code for both CPU as well as GPU under the hood without the user having to rewrite the code, which defines the loss function.
3. optimized automatic/symbolic differentiation
4. numerical stability for the generated code via automatic/symbolic differentiation

N. Ketkar, *Deep Learning with Python*, DOI 10.1007/978-1-4842-2766-4_4

Theano Hands-On

Let us now start getting our hands dirty with Theano. We will start with simple examples, which will serve as conceptual building blocks for more complicated examples. It is recommended that the reader go over the source code listing and corresponding computational graph for each example and give careful consideration to how the source code translates to the computational graph.

In our first example in Listing 4-1 (and the corresponding computational graph in Figure 4-1) we will define a simple function with scalars. The reader should note the following:

1. Scalars are defined before they can be used in a mathematical expression.
2. Every scalar is given a unique name.
3. Once defined, the scalar can be operated upon with operations like +, -, * and /.
4. The function construct in Theano allows one to relate inputs and outputs. So, in the example from Listing 4-1, we have defined a function with the name g, which takes a, b, c, d, and e as input and produces f as the output.
5. We can now compute the result of the function g given the input and check that it evaluates exactly as the non-Theano expression.
6. While this seems like a trivial example, the non-trivial bit is that now we will be able to easily compute the gradient of such a function g quite easily using Theano, as we will see soon.

Listing 4-1. Functions with Scalars

```
import theano.tensor as T
from theano import function

a = T.dscalar('a')
b = T.dscalar('b')
c = T.dscalar('c')
d = T.dscalar('d')
e = T.dscalar('e')

f = ((a - b + c) * d )/e

g = function([a, b, c, d, e], f)

print "Expected: ((1 - 2 + 3) * 4)/5.0 = ", ((1 - 2 + 3) * 4)/5.0
print "Via Theano: ((1 - 2 + 3) * 4)/5.0 = ",  g(1, 2, 3, 4, 5)

# Expected: ((1 - 2 + 3) * 4)/5.0 =  1.6
# Via Theano: ((1 - 2 + 3) * 4)/5.0 =  1.6
```

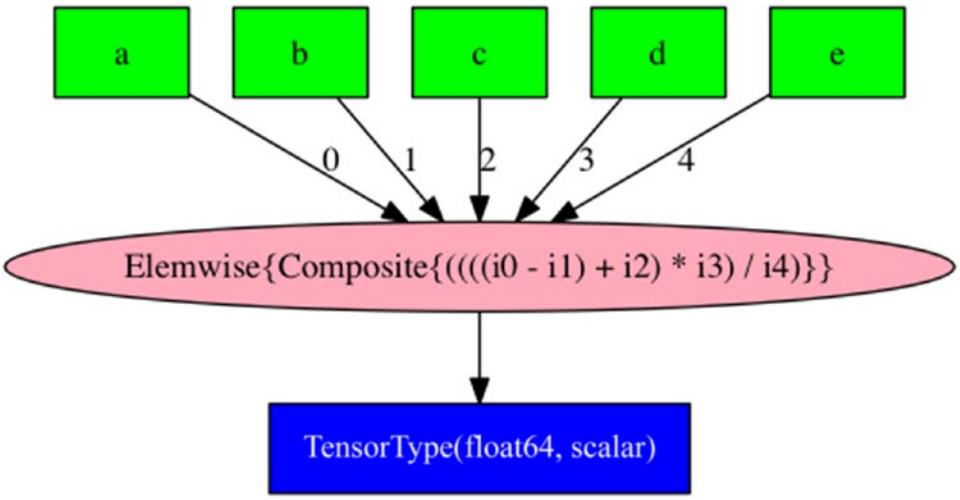


Figure 4-1. *Functions with scalars*

In our second example in Listing 4-2 (and the corresponding computational graph in Figure 4-2) we will define a simple function with vectors. The reader should note the following:

1. Vectors/Matrices are defined before they can be used in a mathematical expression.
2. Every Vector/Matrix is given a unique name.
3. The dimensions of the Vectors/Matrices are not specified.
4. Once Vectors/Matrices are defined, the user can define operations like matrix addition, subtraction, and multiplication. The user should take care that vector/matrix operations respect the intended dimensionality.
5. As before, the user can define a function based on the defined expressions. In this case we define a function f that takes a, b, c, and d as input and produces e as output.
6. The use can pass Numpy arrays to the function and compute the output. The user should take care that vector/matrix inputs respect the intended dimensionality.

Listing 4-2. Functions with Vectors

```
import numpy
import theano.tensor as T
from theano import function

a = T.dmatrix('a')
b = T.dmatrix('b')
c = T.dmatrix('c')
d = T.dmatrix('d')
```

```
e = (a + b - c) * d

f = function([a,b,c,d], e)

a_data = numpy.array([[1,1],[1,1]])
b_data = numpy.array([[2,2],[2,2]])
c_data = numpy.array([[5,5],[5,5]])
d_data = numpy.array([[3,3],[3,3]])

print "Expected:", (a_data + b_data - c_data) * d_data
print "Via Theano:", f(a_data,b_data,c_data,d_data)

# Expected: [[-6 -6]
#  [-6 -6]]
# Via Theano: [[-6. -6.]
#  [-6. -6.]]
```

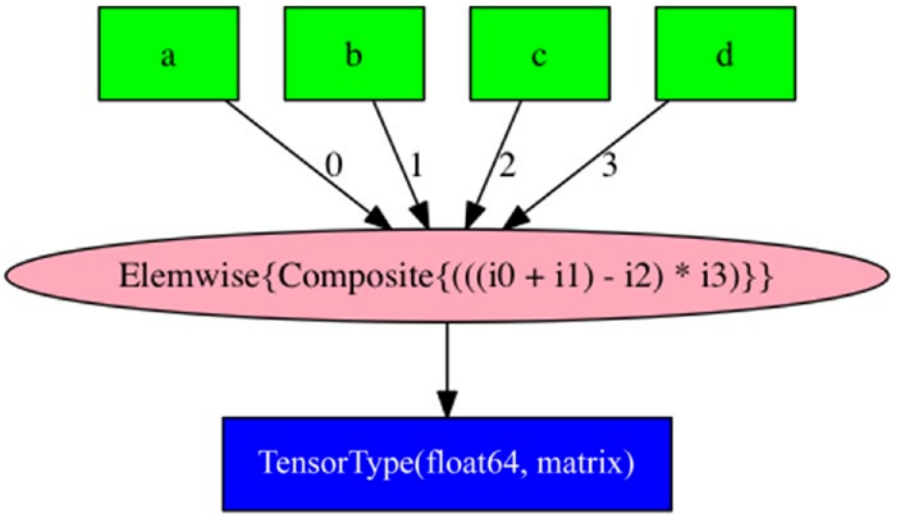


Figure 4-2. *Functions with Vectors*

In our next example in Listing 4-3 (and the corresponding computational graph in Figure 4-3) we will define a function with both scalars and vectors. The reader should note the following:

1. Scalars and vectors/matrices can be used together in expressions.
2. The user needs to take care that vector/matrices respect the dimensionality both while defining the expressions as well as passing inputs to the expressions.

Listing 4-3. Functions with Scalars and Vectors

```
import numpy
import theano.tensor as T
from theano import function
```

```
a = T.dmatrix('a')
b = T.dmatrix('b')
c = T.dmatrix('c')
d = T.dmatrix('d')

p = T.dscalar('p')
q = T.dscalar('q')
r = T.dscalar('r')
s = T.dscalar('s')
u = T.dscalar('u')

e = (((a * p) + (b - q) - (c + r )) * d/s) * u

f = function([a,b,c,d,p,q,r,s,u], e)

a_data = numpy.array([[1,1],[1,1]])
b_data = numpy.array([[2,2],[2,2]])
c_data = numpy.array([[5,5],[5,5]])
d_data = numpy.array([[3,3],[3,3]])

print "Expected:", (((a_data * 1.0) + (b_data - 2.0) - (c_data + 3.0 )) * d_data/4.0) * 5.0
print "Via Theano:", f(a_data,b_data,c_data,d_data,1,2,3,4,5)

# Expected: [[-26.25 -26.25]
#  [-26.25 -26.25]]
# Via Theano: [[-26.25 -26.25]
#  [-26.25 -26.25]]
```

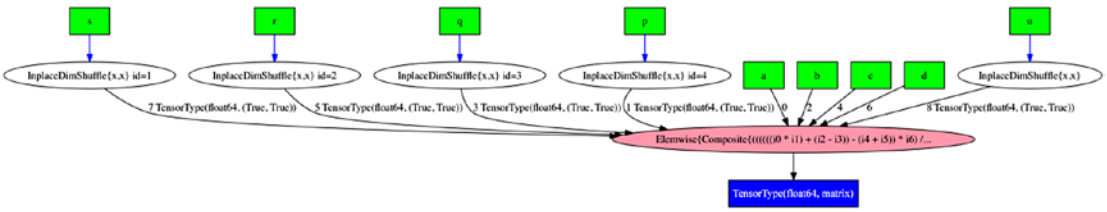


Figure 4-3. *Functions with Scalars and Vectors*

In our next example in Listing 4-4 (and the corresponding computational graph in Figure 4-4) we will define a few activation functions with Theano. The nnet package in Theano defines a number of common activation functions.

Listing 4-4. Activiation Functions

```
import theano.tensor as T
from theano import function

# sigmoid
a = T.dmatrix('a')
f_a = T.nnet.sigmoid(a)
f_sigmoid = function([a],[f_a])
print "sigmoid:", f_sigmoid([[-1,0,1]])
```

```
# tanh
b = T.dmatrix('b')
f_b = T.tanh(b)
f_tanh = function([b],[f_b])
print "tanh:", f_tanh([[-1,0,1]])

# fast sigmoid
c = T.dmatrix('c')
f_c = T.nnet.ultra_fast_sigmoid(c)
f_fast_sigmoid = function([c],[f_c])
print "fast sigmoid:", f_fast_sigmoid([[-1,0,1]])

# softplus
d = T.dmatrix('d')
f_d = T.nnet.softplus(d)
f_softplus = function([d],[f_d])
print "soft plus:",f_softplus([[-1,0,1]])

# relu
e = T.dmatrix('e')
f_e = T.nnet.relu(e)
f_relu = function([e],[f_e])
print "relu:",f_relu([[-1,0,1]])

# softmax
f = T.dmatrix('f')
f_f = T.nnet.softmax(f)
f_softmax = function([f],[f_f])
print "soft max:",f_softmax([[-1,0,1]])
```

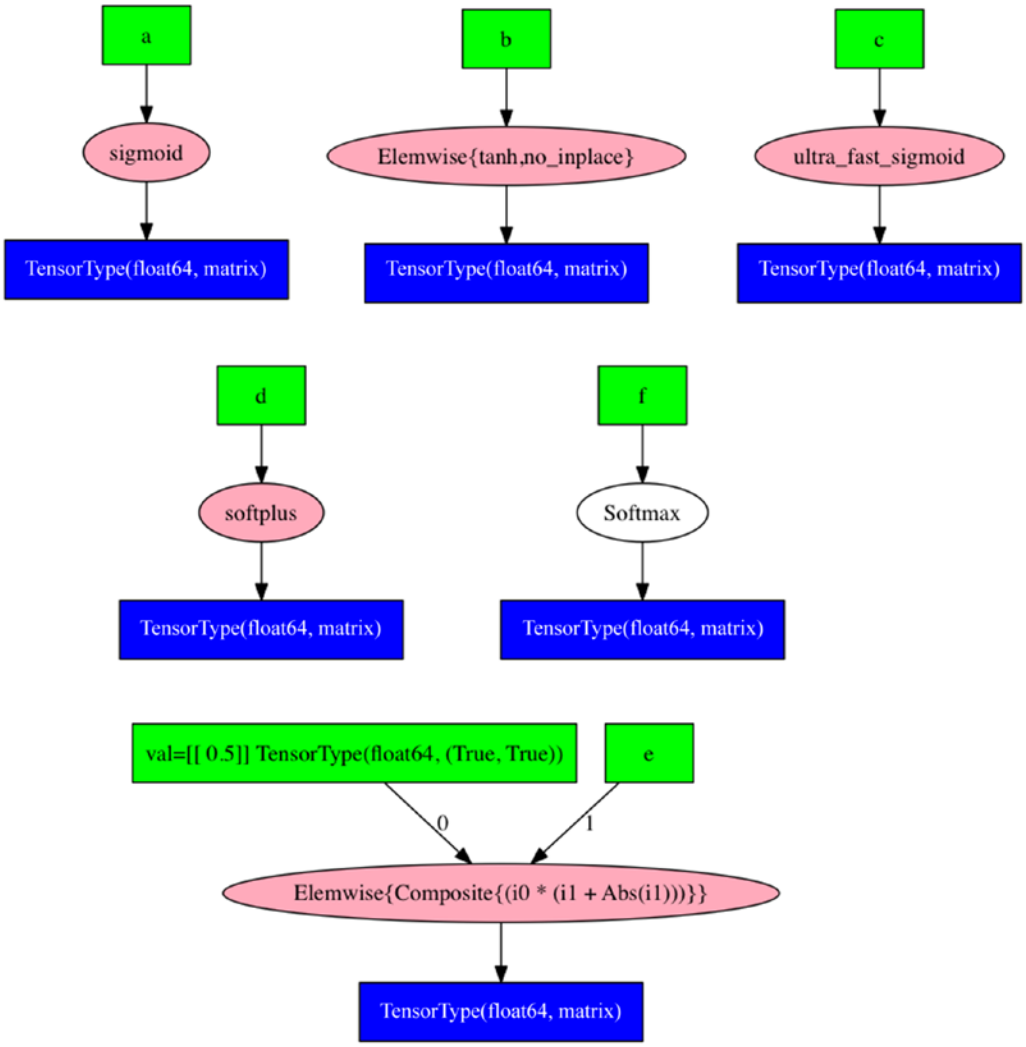


Figure 4-4. *Activation Functions*

In our next example in Listing 4-5 (and the corresponding computational graph in Figure 4-5) we will define a function with internal state. The reader should note the following:

1. All models (deep learning or otherwise) will involve defining functions with internal state, which will typically be weights that need to be learned or fitted.
2. A shared variable is defined using the shared construct in Theano.
3. A shared variable can be initialized with Numpy constructs.
4. Once the shared variable is defined and initialized, it can be used in the definition of expressions and functions in a manner similar to scalars and vectors/matrices, as we have seen earlier.
5. A user can get the value of the shared variable using the get_value method.

6. A user can set the value for the shared variable using the set_value method.
7. A function defined using the shared variable computes its output based on the current value of the shared variable. That is, as soon as the shared variable updates, a function defined using the shared variable will produce a different value for the same input.
8. A shared variable allows a user to define a function with internal state, which can be updated arbitrarily without needing to redefine the function defined using the shared variable.

Listing 4-5. Shared Variables

```
import theano.tensor as T
from theano import function
from theano import shared

import numpy

x = T.dmatrix('x')
y = shared(numpy.array([[4, 5, 6]]))
z = x + y
f = function(inputs = [x], outputs = [z])

print "Original Shared Value:", y.get_value()
print "Original Function Evaluation:", f([[1, 2, 3]])

y.set_value(numpy.array([[5, 6, 7]]))

print "Original Shared Value:", y.get_value()
print "Original Function Evaluation:", f([[1, 2, 3]])

# Couldn't import dot_parser, loading of dot files will not be possible.
# Original Shared Value: [[4 5 6]]
# Original Function Evaluation: [array([[ 5.,  7.,  9.]])]
# Original Shared Value: [[5 6 7]]
# Original Function Evaluation: [array([[  6.,   8.,  10.]])]
```

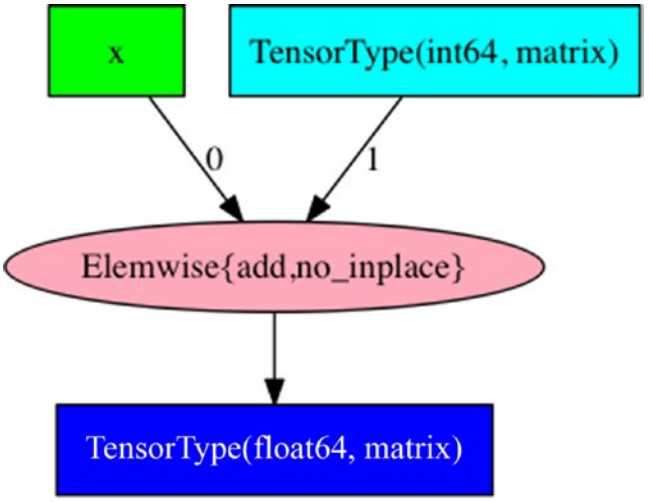


Figure 4-5. *Shared Variables*

In our next example in Listing 4-6 (and the corresponding computational graph in Figure 4-6) we will define a function and generate a function that computes its gradient. The reader should note the following:

1. A function needs to be defined using expressions before the gradient of the function can be generated.
2. The grad construct in Theano allows the user to generate the gradient of a function (as an expression). Users can then define a function over the expression, which gives them the gradient function.
3. Gradients can be computed for any set of expressions/functions as in the earlier examples. So, for instance, we could generate gradients for functions with a shared state. As the shared state updates, so do the function and the gradient function.

Listing 4-6. Gradients

```
import theano.tensor as T
from theano import function
from theano import shared

import numpy

x = T.dmatrix('x')
y = shared(numpy.array([[4, 5, 6]]))
z = T.sum(((x * x) + y) * x)

f = function(inputs = [x], outputs = [z])

g = T.grad(z,[x])
g_f = function([x], g)
```

```
print "Original:", f([[1, 2, 3]])
print "Original Gradient:", g_f([[1, 2, 3]])

y.set_value(numpy.array([[1, 1, 1]]))
print "Updated:", f([[1, 2, 3]])
print "Updated Gradient", g_f([[1, 2, 3]])

# Original: [array(68.0)]
# Original Gradient: [array([[  7.,  17.,  33.]])]
# Updated: [array(42.0)]
# Updated Gradient [array([[  4.,  13.,  28.]])]
```

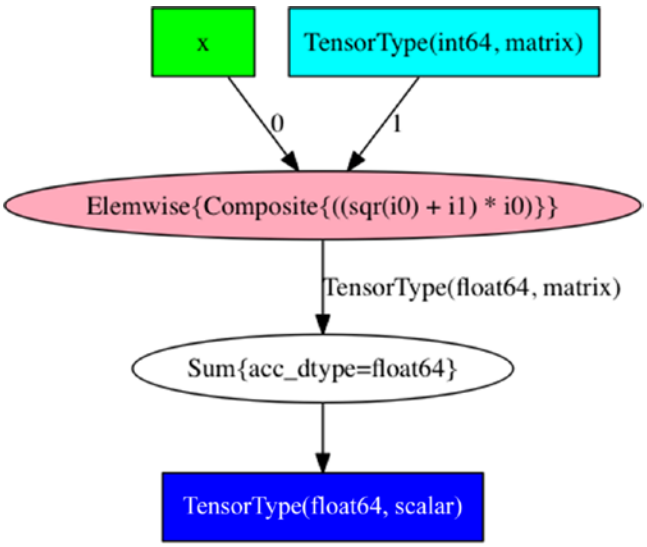


Figure 4-6. *Computing Gradients*

In our next example in Listing 4-7 (and the corresponding computational graph in Figure 4-7 (a) and Figure 4-7 (b)) we will define a few loss functions using Theano. The nnet package in Theano implements many standard loss functions.

Listing 4-7. Loss Functions

```
import theano.tensor as T
from theano import function

# binary cross entropy
a1 = T.dmatrix('a1')
a2 = T.dmatrix('a2')
f_a = T.nnet.binary_crossentropy(a1, a2).mean()
f_sigmoid = function([a1, a2],[f_a])
print "Binary Cross Entropy [[0.01,0.01,0.01]],[[0.99,0.99,0.01]]:",
f_sigmoid([[0.01,0.01,0.01]],[[0.99,0.99,0.01]])
```

```
# categorical cross entropy
b1 = T.dmatrix('b1')
b2 = T.dmatrix('b2')
f_b = T.nnet.categorical_crossentropy(b1, b2)
f_sigmoid = function([b1, b2],[f_b])
print "Categorical Cross Entropy [[0.01,0.01,0.01]],[[0.99,0.99,0.01]]:",
f_sigmoid([[0.01,0.01,0.01]],[[0.99,0.99,0.01]])

# squared error
def squared_error(x,y):
    return (x - y) ** 2

c1 = T.dmatrix('b1')
c2 = T.dmatrix('b2')
f_c = squared_error(c1, c2)
f_squared_error = function([c1, c2],[f_c])
print "Squared Error [[0.01,0.01,0.01]],[[0.99,0.99,0.01]]:",
f_sigmoid([[0.01,0.01,0.01]],[[0.99,0.99,0.01]])

# Binary Cross Entropy [[0.01,0.01,0.01]],[[0.99,0.99,0.01]]: [array(3.058146503109446)]
# Categorical Cross Entropy [[0.01,0.01,0.01]],[[0.99,0.99,0.01]]: [array([ 9.16428867])]
# Squared Error [[0.01,0.01,0.01]],[[0.99,0.99,0.01]]: [array([ 9.16428867])]
```

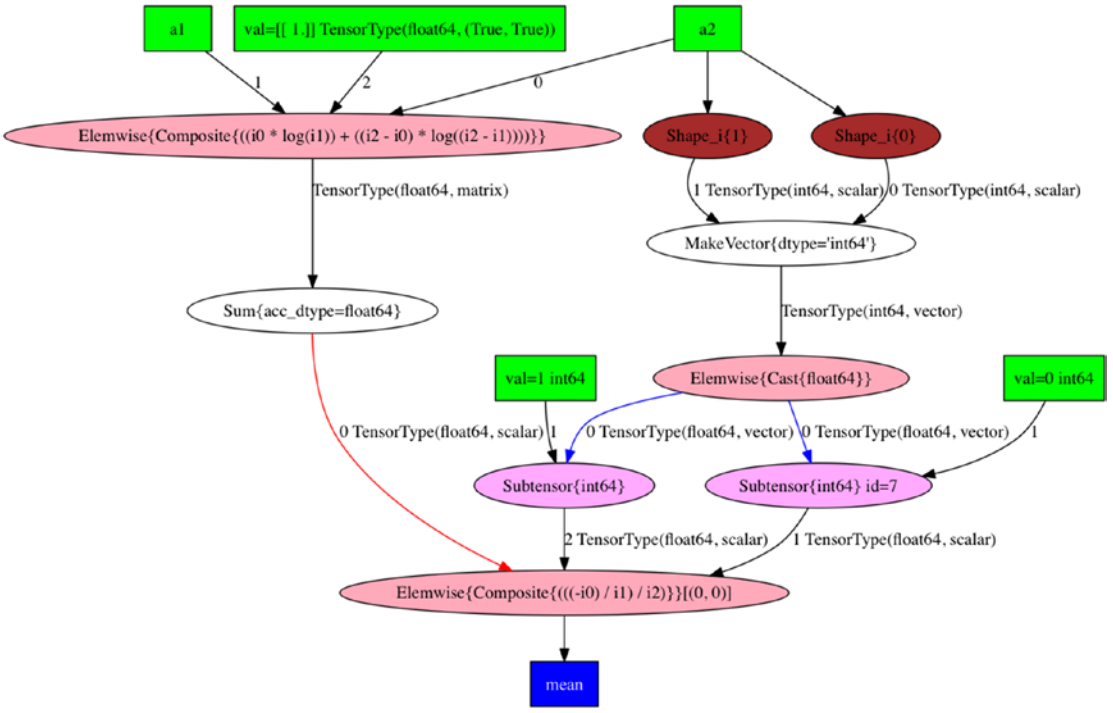


Figure 4-7 (a). *Loss Functions – Binary Cross Entropy*

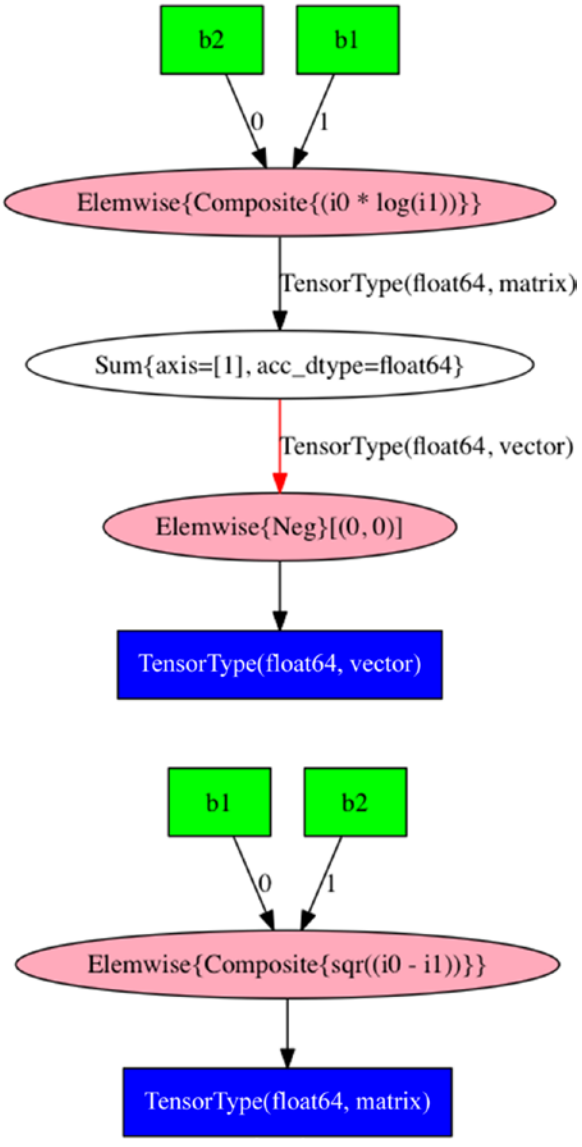


Figure 4-7 (b). *Loss Functions – Categorical Cross Entropy and Squared Error*

In our next example in Listing 4-8 (and corresponding computational graph in Figure 4-8) we will define L1 and L2 regularization using Theano.

Listing 4-8. Regularization

```
import theano.tensor as T
from theano import function

# L1 Regularization
def l1(x):
    return T.sum(abs(x))

# L2 Regularization
def l2(x):
    return T.sum(x**2)

a = T.dmatrix('a')
f_a = l1(a)
f_l1 = function([a], f_a)
print "L1 Regularization:", f_l1([[0,1,3]])

b = T.dmatrix('b')
f_b = l2(b)
f_l2 = function([b], f_b)
print "L2 Regularization:", f_l2([[0,1,3]])

# L1 Regularization: 4.0
# L2 Regularization: 10.0
```

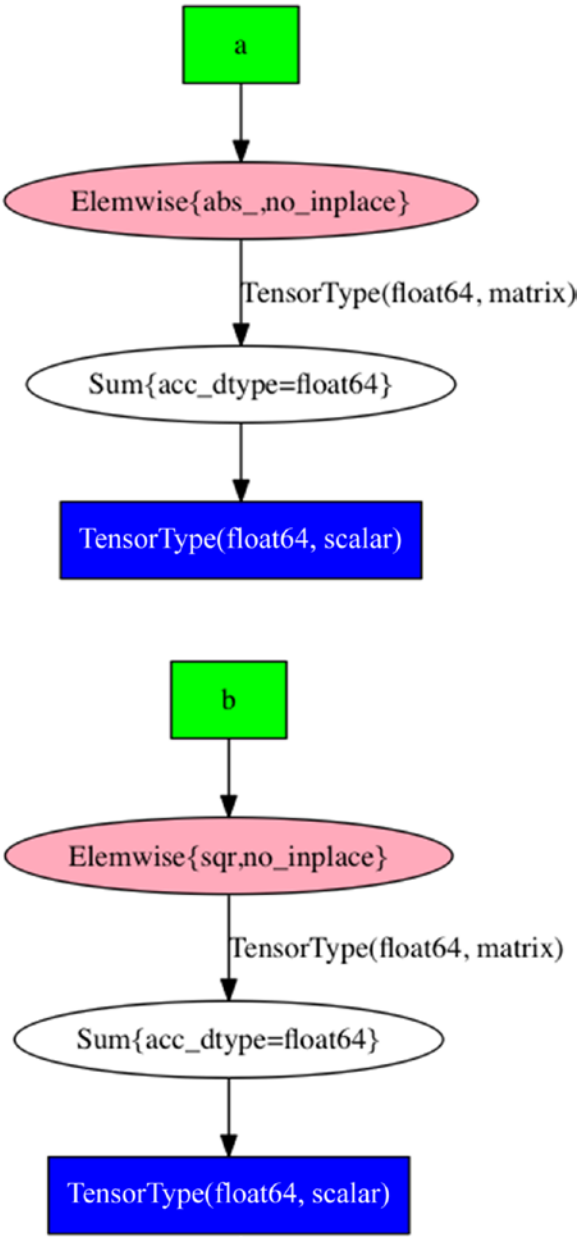


Figure 4-8. *Regularization*

In our next example in Listing 4-9 (and corresponding computational graph in Figure 4-9) we will define a function with a random variable. The reader should note the following:

1. There are cases/situations where we want to define functions having a random variable (for instance introducing minor corruptions in inputs).
2. Such a random element in the function is different from having an internal state, like in the case of shared variables.

3. Basically, the desired outcome in such cases/situations is that the user wants to define a function with a random variable with a particular distribution.
4. Theano provides a construct called RandomStreams, which allows the user to define functions with a random variable. RandomStreams is initialized with a seed.
5. The user defines a variable using RandomStreams and specifies a distribution by calling the appropriate function (in our case, normal).
6. Once defined, the random variable can be used in the definition of expression or functions in a manner similar to scalars and vectors/matrices.
7. Every invocation of the function defined with a random variable will internally draw a sample point from the set distribution (in our case, normal).

Listing 4-9. Random Streams

```
import theano.tensor as T
from theano import function
from theano.tensor.shared_randomstreams import RandomStreams
import numpy

random = RandomStreams(seed=42)

a = random.normal((1,3))
b = T.dmatrix('a')

f1 = a * b

g1 = function([b], f1)

print "Invocation 1:", g1(numpy.ones((1,3)))
print "Invocation 2:", g1(numpy.ones((1,3)))
print "Invocation 3:", g1(numpy.ones((1,3)))

# Invocation 1: [[ 1.25614218 -0.53793023 -0.10434045]]
# Invocation 2: [[ 0.66992188 -0.70813926  0.99601177]]
# Invocation 3: [[ 0.0724739  -0.66508406  0.93707751]]
```

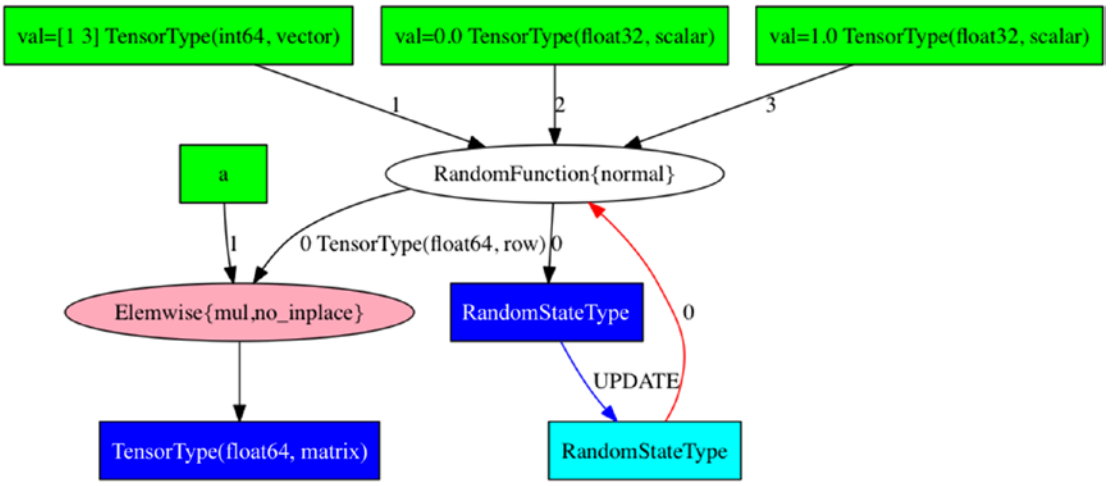


Figure 4-9. *Random Streams*

In our next example in Listing 4-10 (and corresponding computational graph in Figure 4-10) we will build a model for Logistic regression. The reader should note the following:

1. The example generates some artificial/toy data, fits a logistic regression model and computes the accuracy before and after training. This is a toy dataset for the purposes of illustration; the model is not generalizing/learning, as the data is generated randomly.
2. We define a function to compute L2 regularization as covered in listing 4-8 earlier.
3. We generate input data, which consists of 1000 vectors of dimensionality 100. Basically, 1000 examples with 100 features.
4. We generate random target/output labels as zeros and ones.
5. We define the expressions for logistic regression involving the data (denoted by x), the outputs (denoted by y), the bias term (denoted by b), and the weight vector (denoted by w). The weight vector and the bias term are shared variables.
6. We compute the prediction, the error, and the loss using binary cross entropy as introduced in listing 4-7 earlier.
7. Having defined these expressions, we can now use the grad construct in Theano (introduced in listing (4-6)) to compute the gradient.
8. We define a train function based on the gradient function. The train function defines the inputs, outputs, and how the internal state (shared variables) are to be updated.
9. The train function is invoked for 1000 steps; in each step the gradient is computed internally and the shared variables are updated.
10. Accuracy is computed before and after the training steps using sklearn.metrics

Listing 4-10. Logistic Regression

```
import numpy
import theano
import theano.tensor as T
import sklearn.metrics

def l2(x):
    return T.sum(x**2)

examples = 1000
features = 100

D = (numpy.random.randn(examples, features), numpy.random.randint(size=examples,
low=0, high=2))
training_steps = 1000

x = T.dmatrix("x")
y = T.dvector("y")
w = theano.shared(numpy.random.randn(features), name="w")
b = theano.shared(0., name="b")

p = 1 / (1 + T.exp(-T.dot(x, w) - b))
error = T.nnet.binary_crossentropy(p,y)
loss = error.mean() + 0.01 * l2(w)
prediction = p > 0.5
gw, gb = T.grad(loss, [w, b])

train = theano.function(inputs=[x,y],outputs=[p, error], updates=((w, w - 0.1 * gw),
(b, b - 0.1 * gb)))
predict = theano.function(inputs=[x], outputs=prediction)

print "Accuracy before Training:",sklearn.metrics.accuracy_score(D[1], predict(D[0]))

for i in range(training_steps):
    prediction, error = train(D[0], D[1])

print "Accuracy before Training:", sklearn.metrics.accuracy_score(D[1], predict(D[0]))

# Accuracy before Training: 0.481
# Accuracy before Training: 0.629
```

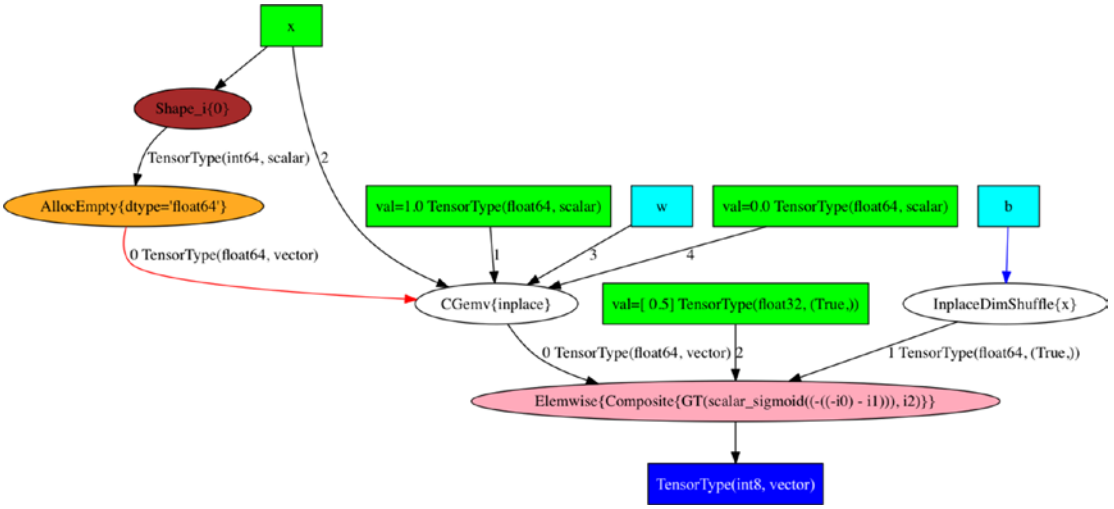


Figure 4-10. *Logistic Regression*

In our next example in Listing 4-11 (and corresponding computational graph in Figure 4-11) we will build a model for Linear regression. The reader should note the following:

1. The example generates some artificial/toy data, fits a linear regression model and computes the accuracy before and after training. This is a toy dataset for the purposes of illustration, the model is not generalizing/learning as the data is generated randomly.
2. We define a function to compute L2 regularization as covered in listing 4-8 earlier.
3. We define a function for squared error as covered in listing 4-7.
4. We generate input data, which consists of 1000 vectors of dimensionality 100. Basically, 1000 examples with 100 features.
5. We generate random target/output labels values between 0 and 1.
6. We define the expressions for linear regression involving the data (denoted by x), the outputs (denoted by y), the bias term (denoted by b) and the weight vector (denoted by w). The weight vector and the bias term are shared variables.
7. We compute the prediction, the error and the loss using squared error as introduced in listing 4-7 earlier.
8. Having defined these expressions, we can now use the grad construct in Theano (introduced in listing (4-6) to compute the gradient.
9. We define a train function based on the gradient function. The train function defines the inputs, outputs and how the internal state (shared variables) are to be updated.
10. The train function is invoked for a 1000 steps, in each step the gradient is computed internally and the shared variables are updated.
11. Root mean squared error (RMSE) is computed before and after the training steps using sklearn.metrics

Listing 4-11. Linear Regression

```
import numpy
import theano
import theano.tensor as T
import sklearn.metrics

def l2(x):
    return T.sum(x**2)

def squared_error(x,y):
    return (x - y) ** 2

examples = 1000
features = 100

D = (numpy.random.randn(examples, features), numpy.random.randn(examples))
training_steps = 1000

x = T.dmatrix("x")
y = T.dvector("y")
w = theano.shared(numpy.random.randn(features), name="w")
b = theano.shared(0., name="b")

p = T.dot(x, w) + b
error = squared_error(p,y)
loss = error.mean() + 0.01 * l2(w)
gw, gb = T.grad(loss, [w, b])

train = theano.function(inputs=[x,y],outputs=[p, error], updates=((w, w - 0.1 * gw),
(b, b - 0.1 * gb)))
predict = theano.function(inputs=[x], outputs=p)

print "RMSE before training:", sklearn.metrics.mean_squared_error(D[1],predict(D[0]))

for i in range(training_steps):
    prediction, error = train(D[0], D[1])

print "RMSE after training:", sklearn.metrics.mean_squared_error(D[1],predict(D[0]))

# RMSE before training: 90.4707491496
# RMSE after training: 0.915701676631
```

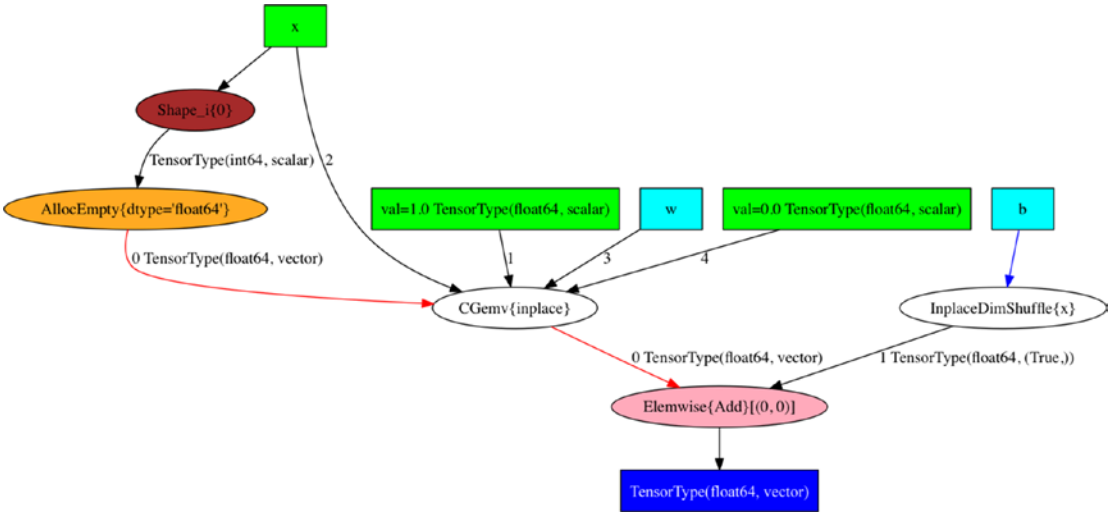


Figure 4-11. *Linear Regression*

In our next example in Listing 4-12 (and corresponding computational graph in Figure 4-12) we will build a neural network model with 2 layers. The reader should note the following:

1. The example generates some artificial/toy data, fits a logistic regression model and computes the accuracy before and after training. This is a toy dataset for the purposes of illustration; the model is not generalizing/learning, as the data is generated randomly.
2. We define a function to compute L2 regularization as covered in listing 4-8 earlier.
3. We generate input data, which consists of 1000 vectors of dimensionality 100. Basically, 1000 examples with 100 features.
4. We generate random target/output labels as zeros and ones.
5. We define the expressions for the 2-layer neural network involving the data (denoted by x), the outputs (denoted by y), the bias term of the first layer (denoted by b1), the weight vector of the first layer (denoted by w1), the bias term of the second layer (denoted by b2), and, the weight vector of the second layer (denoted by w2). The weight vectors and the bias terms are shared variables.
6. We use the tanh activation function as covered in listing 4-4 to encode the neural network.
7. We compute the prediction, the error, and the loss using binary cross entropy as introduced in listing 4-7 earlier.
8. Having defined these expressions, we can now use the grad construct in Theano (introduced in listing (4-6)) to compute the gradient.
9. We define a train function based on the gradient function. The train function defines the inputs, outputs, and how the internal state (shared variables) are to be updated.

10. The train function is invoked for 1000 steps; in each step the gradient is computed internally and the shared variables are updated.

11. Accuracy is computed before and after the training steps using sklearn.metrics

Listing 4-12. Neural Network

```
import numpy
import theano
import theano.tensor as T
import sklearn.metrics

def l2(x):
    return T.sum(x**2)

examples = 1000
features = 100
hidden = 10

D = (numpy.random.randn(examples, features), numpy.random.randint(size=examples,
low=0, high=2))
training_steps = 1000

x = T.dmatrix("x")
y = T.dvector("y")

w1 = theano.shared(numpy.random.randn(features, hidden), name="w1")
b1 = theano.shared(numpy.zeros(hidden), name="b1")

w2 = theano.shared(numpy.random.randn(hidden), name="w2")
b2 = theano.shared(0., name="b2")

p1 = T.tanh(T.dot(x, w1) + b1)
p2 = T.tanh(T.dot(p1, w2) + b2)

prediction = p2 > 0.5

error = T.nnet.binary_crossentropy(p2,y)

loss = error.mean() + 0.01 * (l2(w1) + l2(w2))
gw1, gb1, gw2, gb2 = T.grad(loss, [w1, b1, w2, b2])

train = theano.function(inputs=[x,y],outputs=[p2, error], updates=((w1, w1 - 0.1 * gw1),
(b1, b1 - 0.1 * gb1), (w2, w2 - 0.1 * gw2), (b2, b2 - 0.1 * gb2)))
predict = theano.function(inputs=[x], outputs=[prediction])

print "Accuracy before Training:", sklearn.metrics.accuracy_score(D[1], numpy.
array(predict(D[0])).ravel())

for i in range(training_steps):
     prediction, error = train(D[0], D[1])
```

```
print "Accuracy after Training:", sklearn.metrics.accuracy_score(D[1],
numpy.array(predict(D[0])).ravel())

# Accuracy before Training: 0.51
# Accuracy after Training: 0.716
```

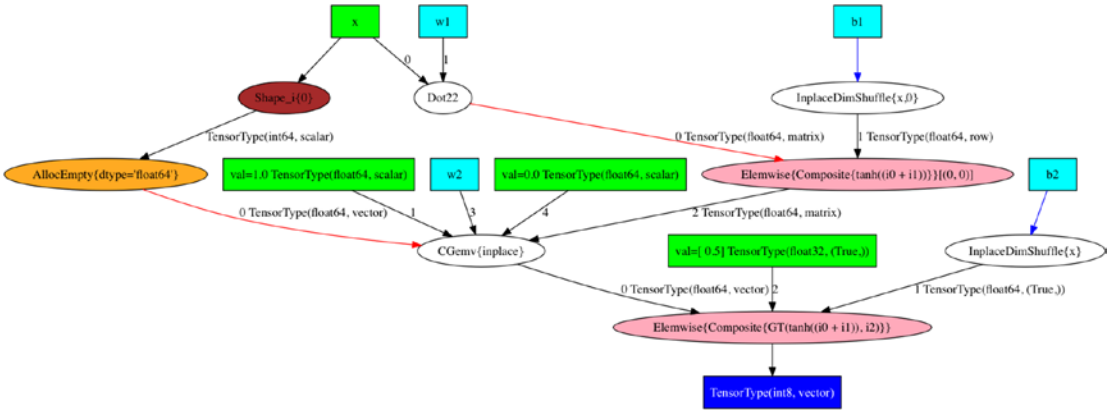


Figure 4-12. *Neural Network*

In our next example in Listing 4-13 (and corresponding computational graphs in Figures 4-13 (a), 4-13 (b) and 4-13 (c)) we will define a function using the if-else and switch construct. The reader should note the following:

1. Certain functions need an if-else (or switch) clause for their evaluation. For such cases Theano provides an if-else and switch constructs.
2. Expressions and functions can be defined using the if-else and switch constructs and gradients can be generated as with other expressions/constructs.
3. In the example we demonstrate the computation of the hinge lose using the if-else and switch construct and verify that it matches to the one defined with max.

Listing 4-13. Switch/If-Else

```
import numpy
import theano
import theano.tensor as T
from theano.ifelse import ifelse

def hinge_a(x,y):
    return T.max([0 * x, 1-x*y])

def hinge_b(x,y):
    return ifelse(T.lt(1-x*y,0), 0 * x, 1-x*y)

def hinge_c(x,y):
    return T.switch(T.lt(1-x*y,0), 0 * x, 1-x*y)
```

```
x = T.dscalar('x')
y = T.dscalar('y')

z1 = hinge_a(x, y)
z2 = hinge_b(x, y)
z3 = hinge_b(x, y)

f1 = theano.function([x,y], z1)
f2 = theano.function([x,y], z2)
f3 = theano.function([x,y], z3)

print "f(-2, 1) =",f1(-2, 1), f2(-2, 1), f3(-2, 1)
print "f(-1,1 ) =",f1(-1, 1), f2(-1, 1), f3(-1, 1)
print "f(0,1) =",f1(0, 1), f2(0, 1), f3(0, 1)
print "f(1, 1) =",f1(1, 1), f2(1, 1), f3(1, 1)
print "f(2, 1) =",f1(2, 1), f2(2, 1), f3(2, 1)

# f(-2, 1) = 3.0 3.0 3.0
# f(-1,1 ) = 2.0 2.0 2.0
# f(0,1) = 1.0 1.0 1.0
# f(1, 1) = 0.0 0.0 0.0
# f(2, 1) = 0.0 0.0 0.0
```

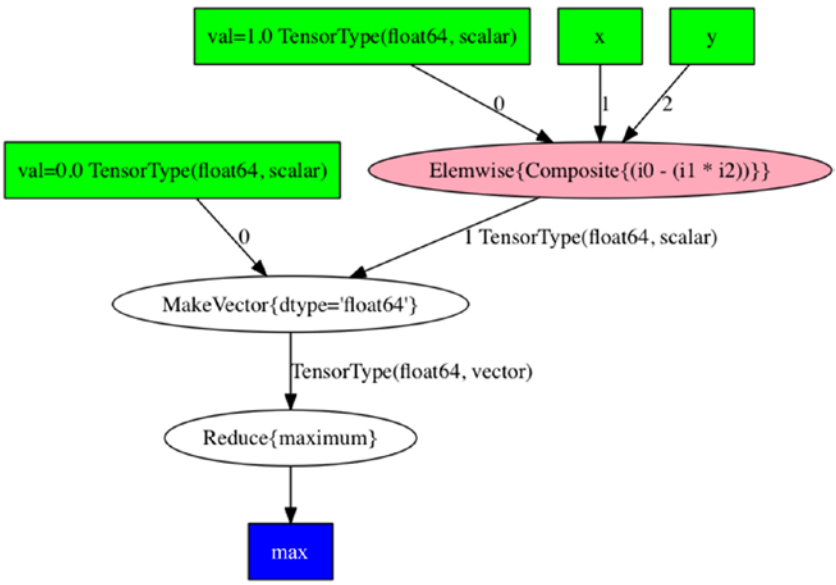


Figure 4-13 (a). *Hinge implemented using Max*

Figure 4-13 (a) illustrates the implementation of the hinge loss using the max operation that is $l(y) = \max(0, 1 - x \cdot y)$ where x is the correct/actual output and y is the output produced by the model. The computational graph closely corresponds to the formula/equation for hinge loss.

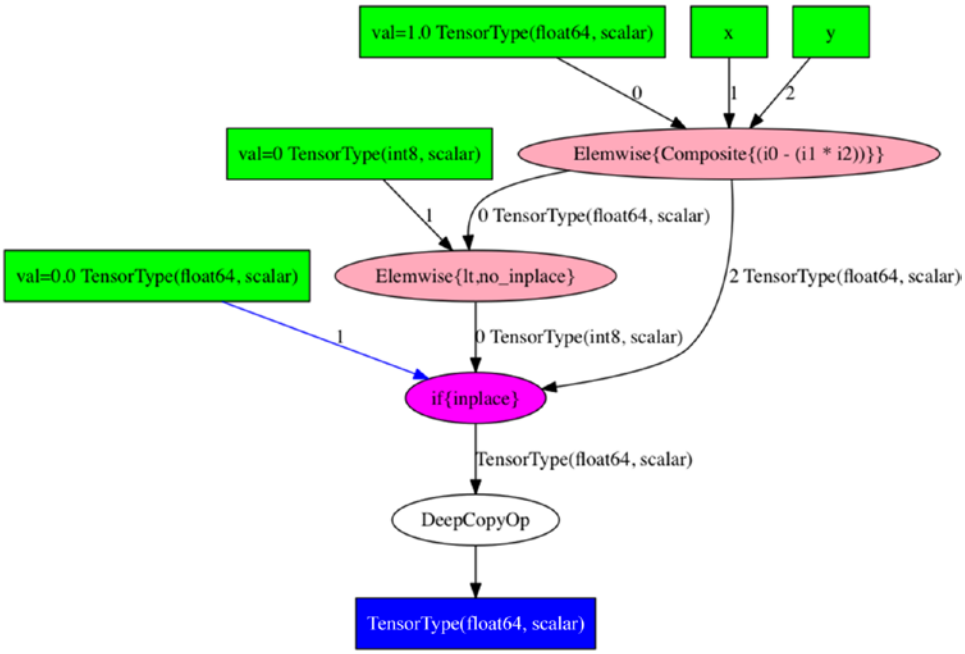


Figure 4-13 (b). *Hinge implemented using ifelse*

Figure 4-13 (b) illustrates the implementation of the hinge loss using the ifelse construct. Note how the computational graph implements the condition and the intended out for each of the condition.

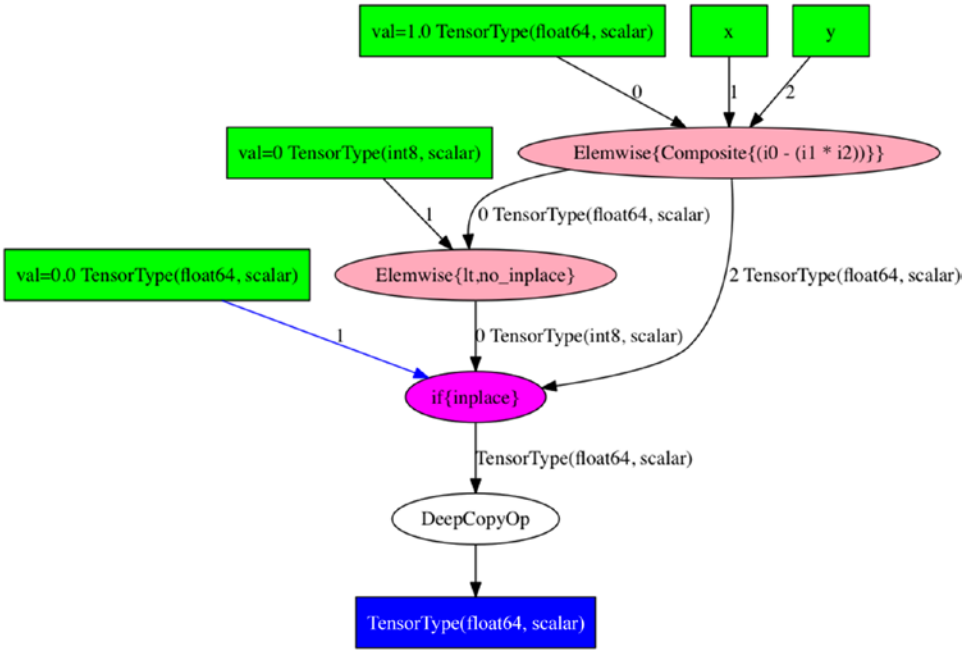


Figure 4-13 (c). *Hinge implemented using switch*

In our next example in listing 4-14 (and the corresponding computational graph in Figure 4-14) we illustrate the scan construct that allows the user to define functions involving iterative computation. The reader should note the following.

1. Computation of certain functions requires iterative constructs for which Theano provides the scan construct.
2. In our example we compute the power operation with the scan construct and match the output with using the standard operator for power.
3. Expressions and functions can be defined using the scan construct and gradients can be generated as with other expressions/constructs.

Listing 4-14. Scan

```
import theano
import theano.tensor as T
import theano.printing
k = T.iscalar("k")
a = T.dscalar("a")
result, updates = theano.scan(fn=lambda prior_result, a: prior_result * a, outputs_info=a,
non_sequences=a, n_steps=k-1)
final_result = result[-1]
a_pow_k = theano.function(inputs=[a,k], outputs=final_result, updates=updates)
print a_pow_k(2,5), 2 ** 5
print a_pow_k(2,5), 2 ** 5
# 32.0 32
```

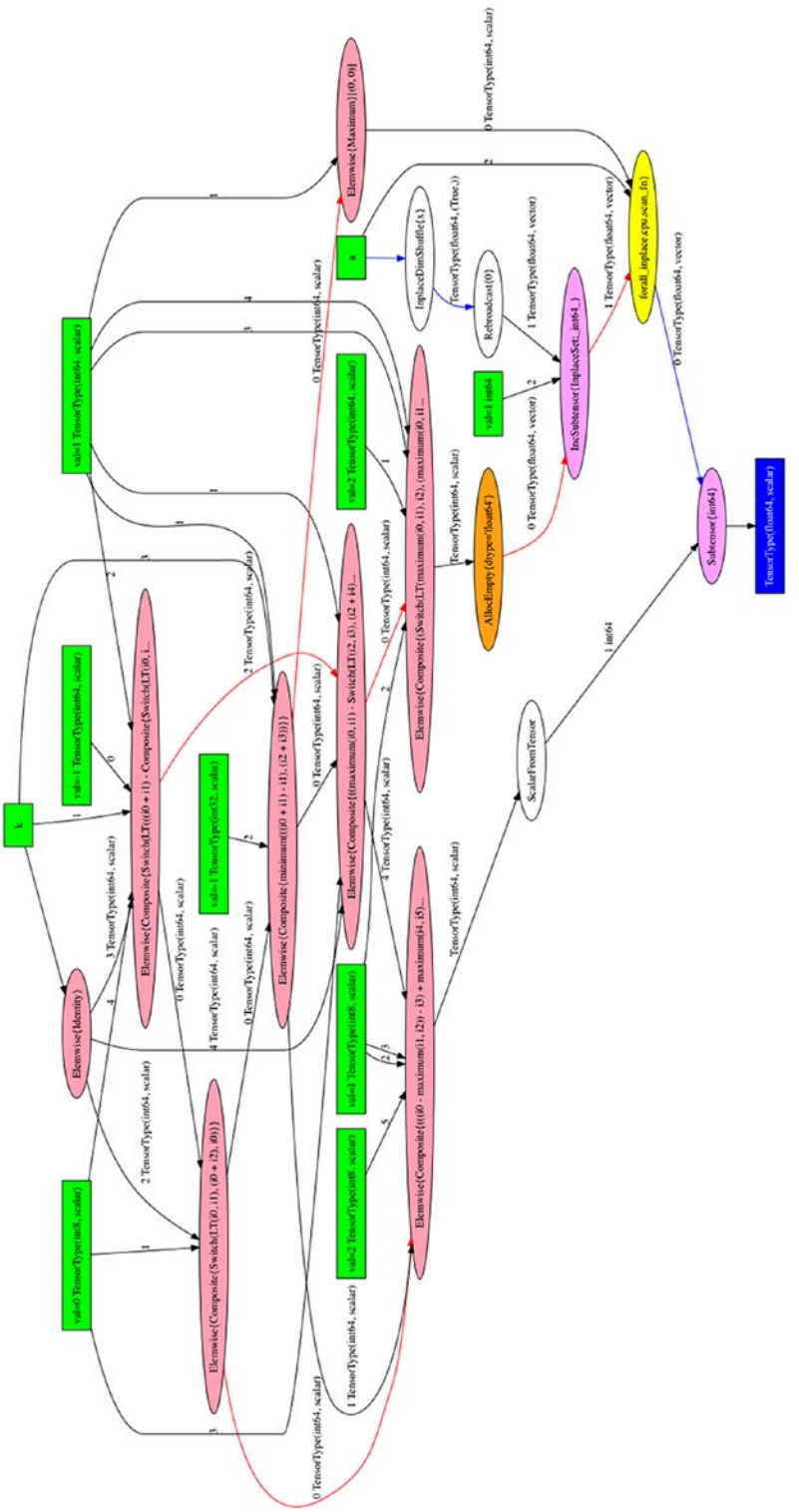


Figure 4-14. *Scan Operation*

Summary

In this chapter we covered the basics of Theano which is a low level library for building and training neural networks. Theano allows users to build computational graphs and compute gradients and is the ideal tool when it comes to building new architectures/networks as it gives the user a very fine grained control over the computational graph. A number of libraries like Keras (covered in Chapter 7) and Lasagne are built over Theano and provide higher level abstractions so that users need not build networks using computational graphs themselves. Such higher level libraries make the user much more productive, but the user does not have precise control over the network/architecture. In general, higher level libraries are recommended when the higher library provides a close enough implementation of the network/ architecture the user wants to build. In case, this is not available, it is recommended that the user build the network using Theano which will give him complete control. Basically using such high level libraries versus Theano is a tradeoff between productivity and control, much similar to programming in Python vs. programming in C.

CHAPTER 5

Convolutional Neural Networks

Convolution Neural Networks (CNNs) in essence are neural networks that employ the convolution operation (instead of a fully connected layer) as one of its layers. CNNs are an incredibly successful technology that has been applied to problems wherein the input data on which predictions are to be made has a known grid like topology like a time series (which is a 1-D grid) or an image (which is a 2-D grid).

Convolution Operation

Let us start developing intuition for the convolution operation in one dimension. Given an input $I(t)$ and a kernel $K(a)$ the convolution operation is given by

$$s(t) = \sum_a I(a) \cdot K(t-a)$$

An equivalent form of this operation given commutativity of the convolution operation is as follows:

$$s(t) = \sum_a I(t-a) \cdot K(a)$$

Furthermore, the negative sign (flipping) can be replaced to get cross-correlation given as follows:

$$s(t) = \sum_a I(t+a) \cdot K(a)$$

N. Ketkar, *Deep Learning with Python*, DOI 10.1007/978-1-4842-2766-4_5

In deep learning literature and software implementations, convolution and cross-correlation are used interchangeably. The essence of the operation is that the Kernel is a much shorter set of data points as compared to the input, and the output of the convolution operation is higher when the input is similar to the kernel. Figures 5-1 and 5-2 illustrate this key idea. We take an arbitrary input and an arbitrary kernel, perform the convolution operation, and the highest value is achieved when the kernel is similar to a particular portion of the input.

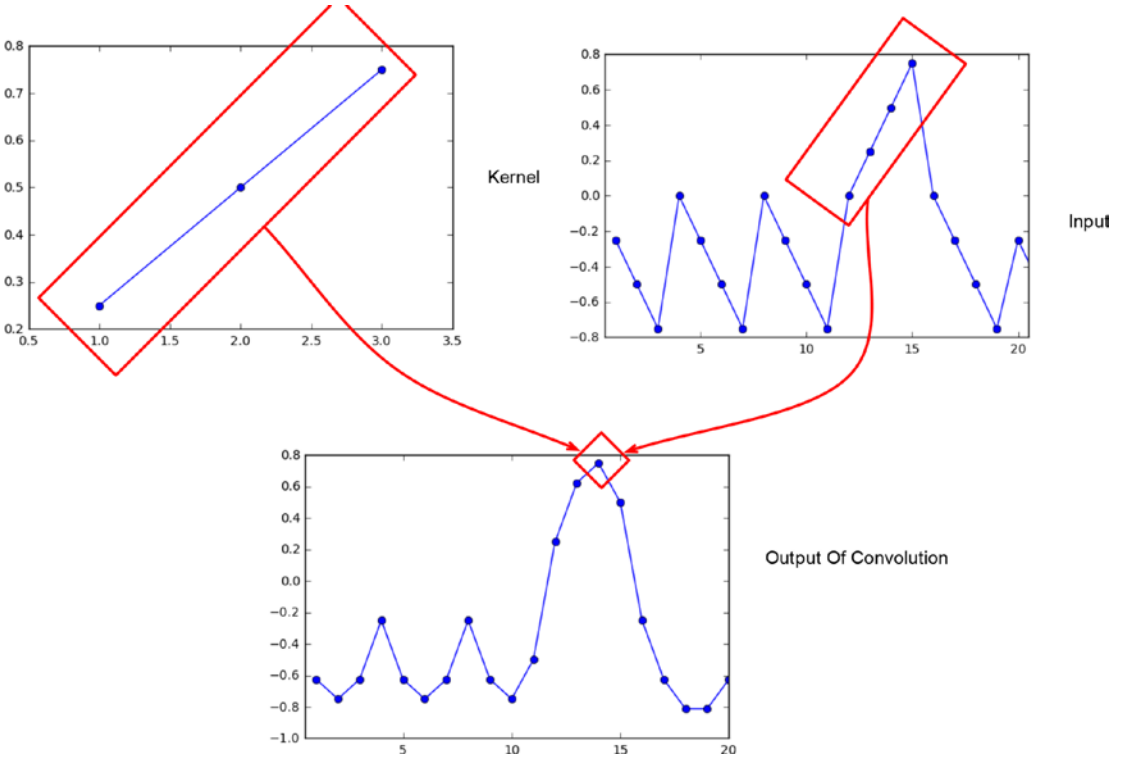


Figure 5-1. *Convolution operation – Intuition*

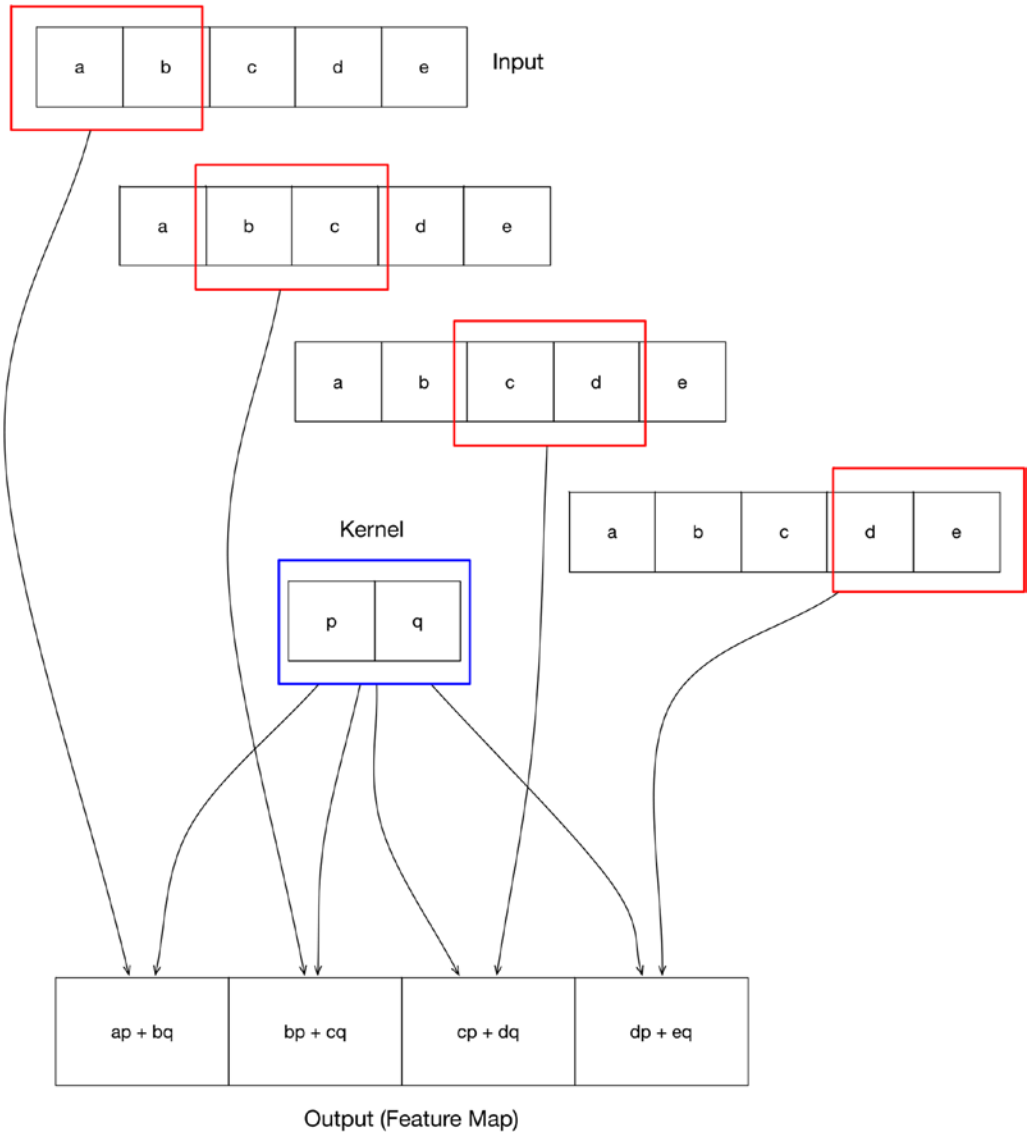


Figure 5-2. *Convolution operation – One Dimension*

Let us strengthen our intuition about convolution by observing Figures 5-1 and 5-2 and noting the following points:

1. The input is an arbitrary large set of data points.
2. The Kernel is a set of data points smaller in number to the input.
3. The convolution operation in a sense slides the kernel over the input and computes how similar the kernel is with the portion of the input.
4. The convolution operation produces the highest value where the Kernel is most similar with a portion of the input.

The convolution operation can be extended to two dimensions. Given an input $I(m, n)$ and a kernel $K(a, b)$ the convolution operation is given by

$$s(t) = \sum_a \sum_b I(a,b) \cdot K(m-a, n-b)$$

An equivalent form of this operation given commutativity of the convolution operation is as follows:

$$s(t) = \sum_a \sum_b I(m-a,n-b) \cdot K(a,b)$$

Furthermore, the negative sign (flipping) can be replaced to get cross-correlation given as follows:

$$s(t) = \sum_a \sum_b I(m+a,n+b) \cdot K(a,b)$$

Figure 5-3 illustrates the convolution operation in two dimensions. Note that this is simply extending the idea of convolution to two dimensions.

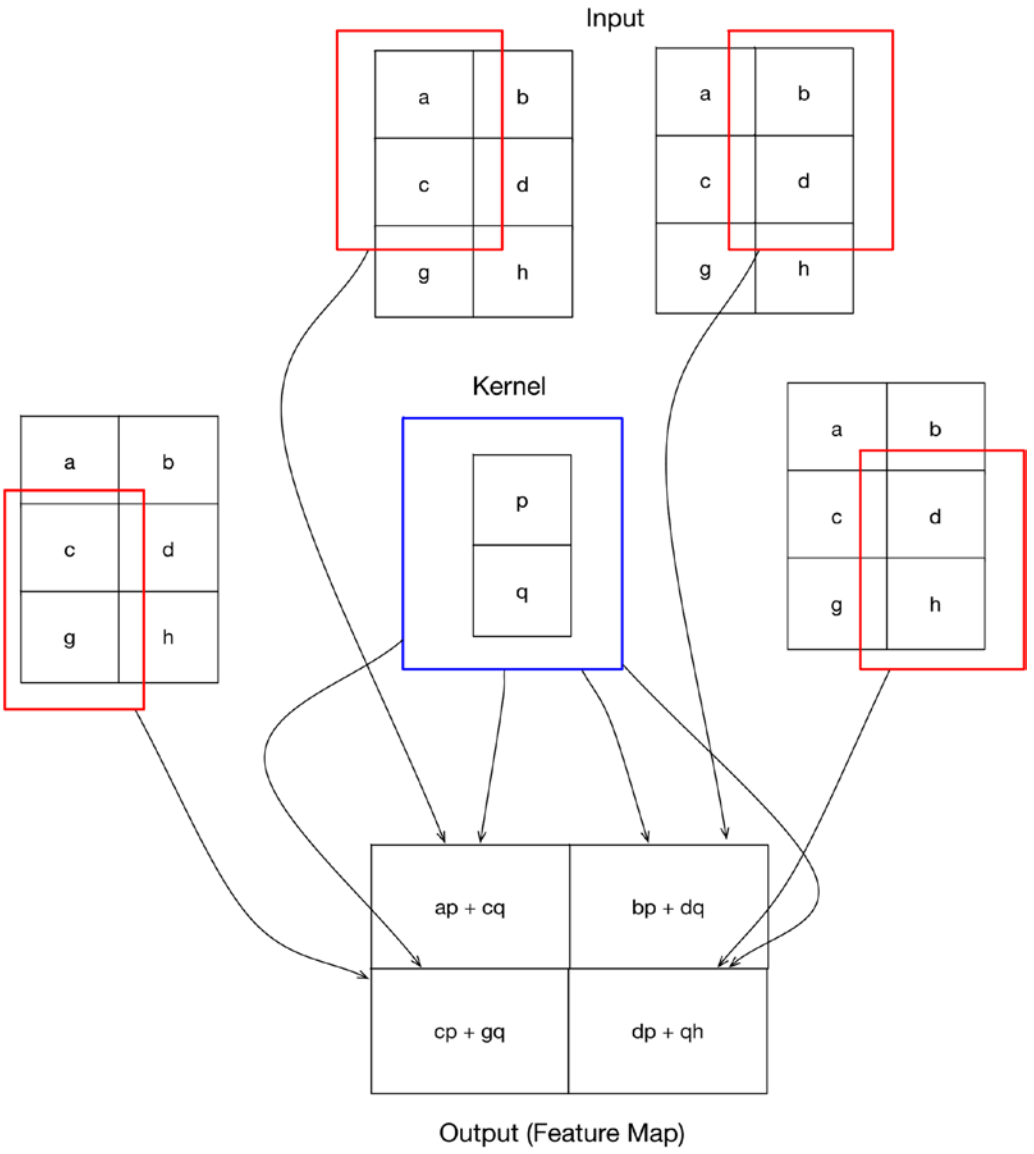


Figure 5-3. *Convolution operation – Two Dimensions*

Having introduced the convolution operation, we can now dive deeper into the key constituent parts of a CNN, were a convolution layer is used instead of a fully connected layer which involves a matrix multiplication. So, a fully connected layer can be described as $y = f(x \cdot w)$ where x is the input vector, y is the output vector, w is a set of weights, and f is the activation function. Correspondingly, a convolution layer can be described as $y = f(s(x \cdot w))$ where s denotes the convolution operation between the input and the weights.

Let us now contrast the fully connected layer with the convolution layer. Figure 5-4 illustrates a fully connected layer and Figure 5-5 illustrates a convolution layer, schematically. Figure 5-6 illustrates parameter sharing in a convolution layer and the lack of it in a fully connected layer. The following points should be noted:

1. For the same number of inputs and outputs, the fully connected layer has a lot more connections, and correspondingly weights that a convolution layer.
2. The interactions amongst inputs to produce outputs are fewer in convolution layers as compared to many interactions in the case of a fully connected layer. This is referred to as sparse interactions.
3. Parameters/weights are shared across the convolution layer, given that the kernel is much smaller than the input and the kernel slides across the input. Thus, there are a lot fewer unique parameters/weights in a convolution layer.

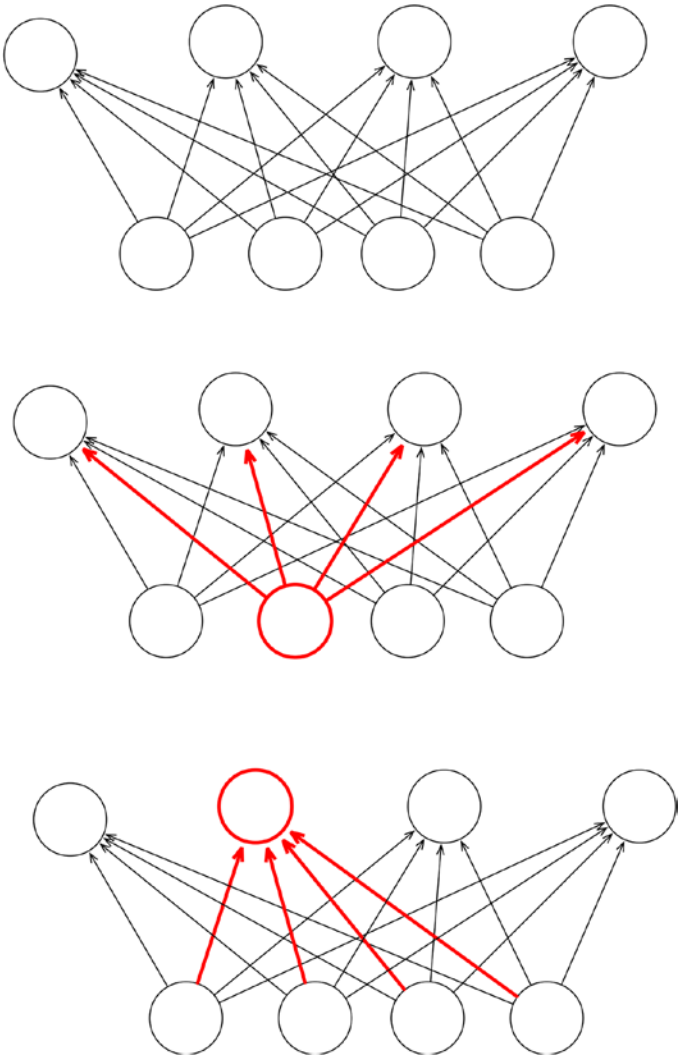


Figure 5-4. *Dense Interactions in Fully Connected Layers*

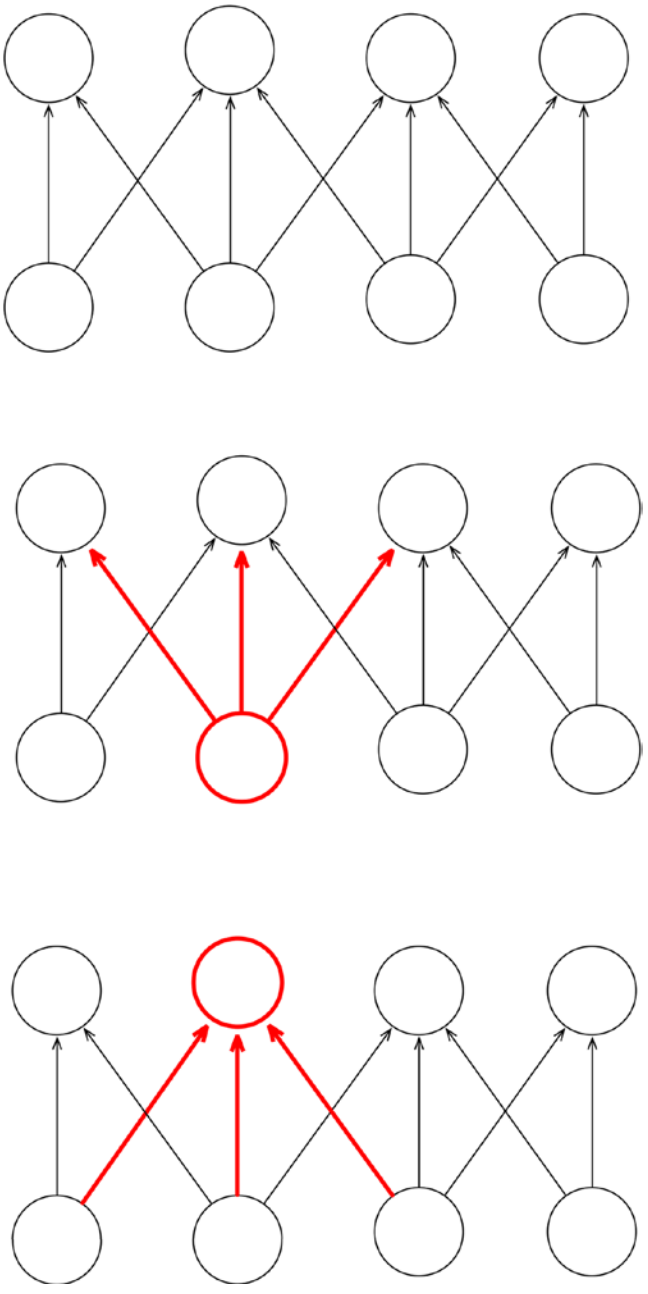


Figure 5-5. *Sparse Interactions in Convolution Layer*

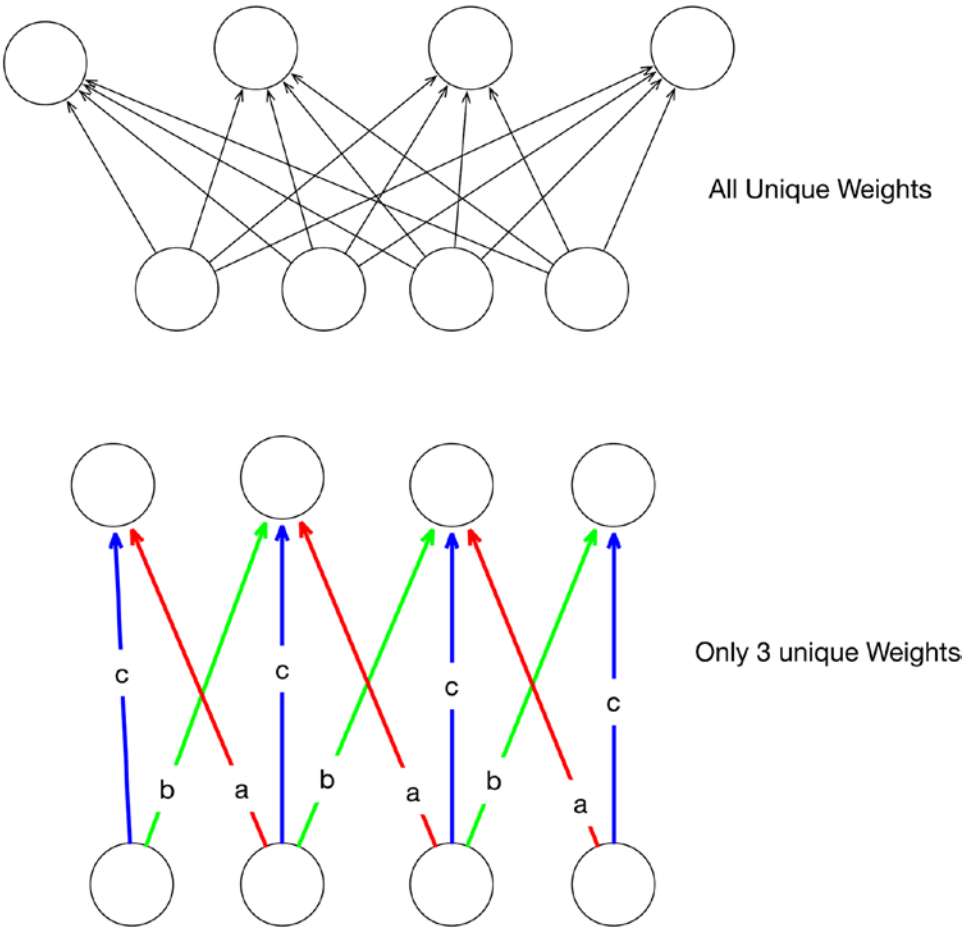


Figure 5-6. *Parameter Sharing Tied Weights*

Pooling Operation

Let us now look at the pooling operation which is almost always used in CNNs in conjunction with convolution. The intuition behind the pooling operation is that the exact location of the feature is not a concern if in fact it has been discovered. It simply provides translation invariance. So, for instance, assume that the task at hand is to learn to detect faces in photographs. Let us also assume that the faces in the photograph are tilted (as they generally are) and suppose that we have a convolution layer that detects the eyes. We would like to abstract the location of the eyes in the photograph from their orientation. The pooling operation achieves this and is an important constituent of CNNs.

Figure 5-7 illustrates the pooling operation for a two-dimensional input. The following points are to be noted:

1. Pooling operates over a portion of the input and applies a function *f* over this input to produce the output.

2. The function *f* is commonly the *max* operation (leading to max pooling), but other variants such as average or L_2 norm can be used as an alternative.

3. For a two-dimensional input, this is a rectangular portion.
4. The output produced as a result of pooling is much smaller in dimensionality as compared to the input.

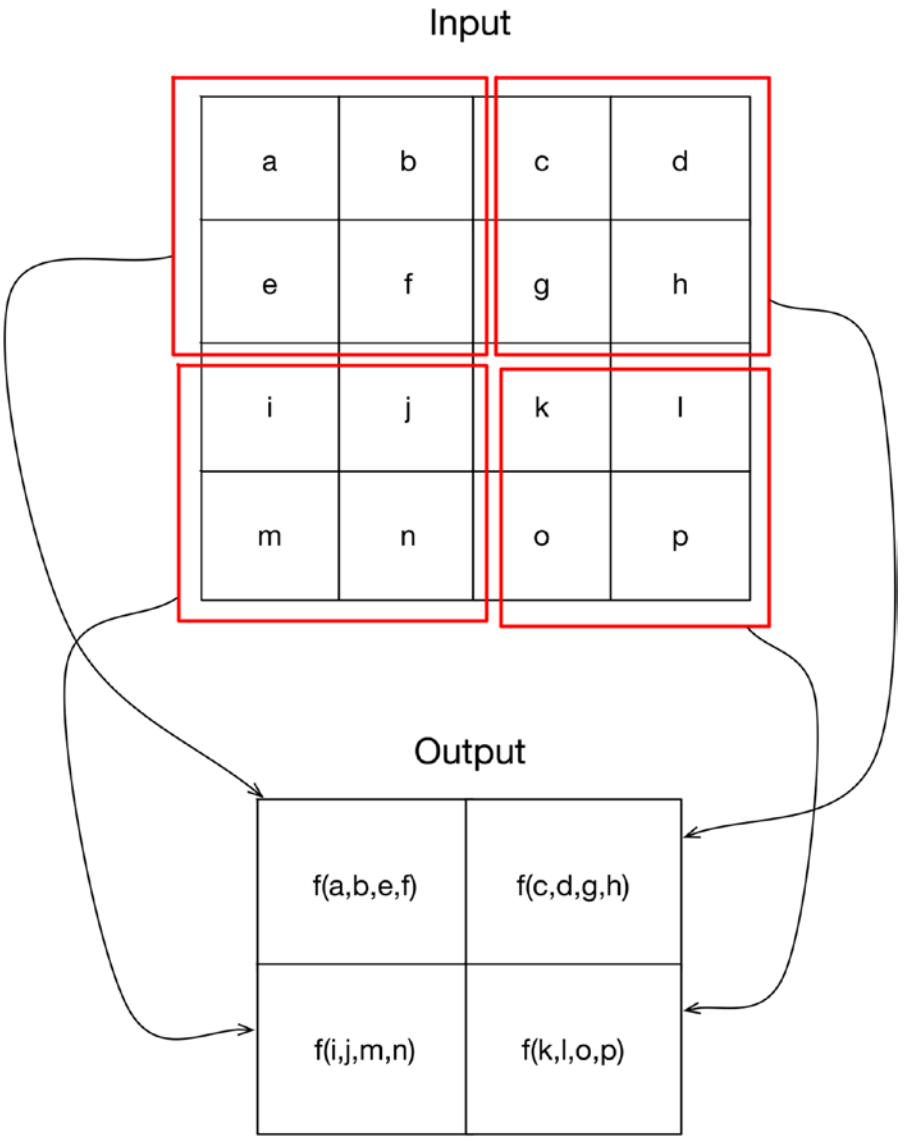


Figure 5-7. *Pooling or Subsampling*

Convolution-Detector-Pooling Building Block

Let us now look at the Convolution-Detector-Pooling block, which can be thought of as a building block of the CNN.

Let us now look at how all the operations we have covered earlier work in conjunction. Refer to Figure 5-8 and Figure 5-9. The following points are to be noted:

1. The detector stage is simply a non-linear activation function.
2. The convolution, detector, and pooling operations are applied in sequence to transform the input to the output. The output is referred to as a feature map.
3. The output typically is passed on to other layers (convolution or fully connected).
4. Multiple Convolution-Detector-Pooling blocks can be applied in parallel, consuming the same input and producing multiple outputs or feature maps.

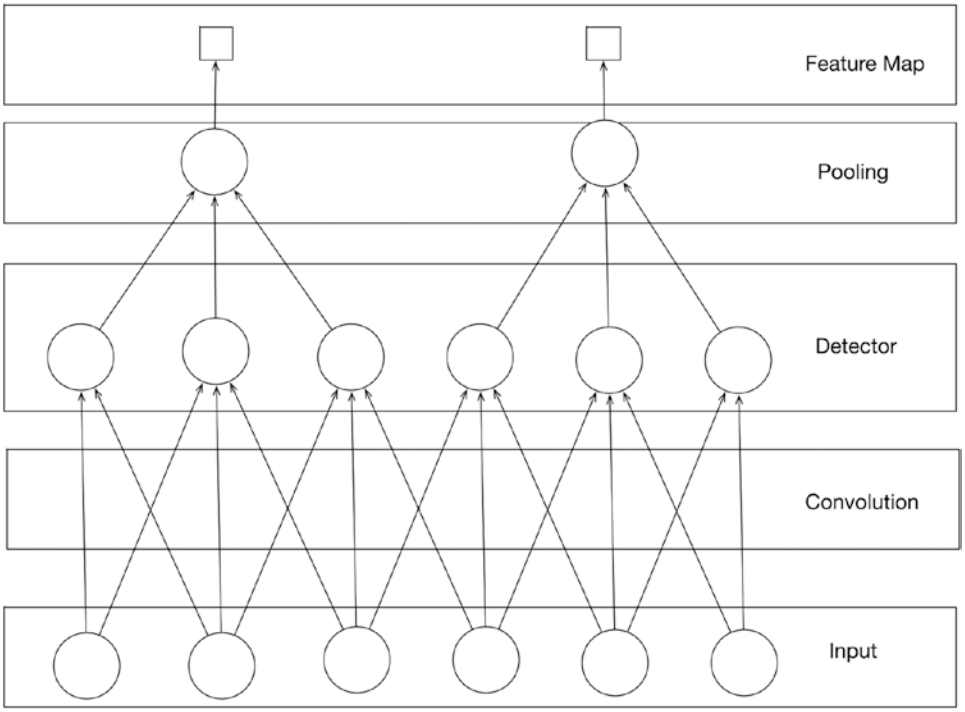


Figure 5-8. *Convolution followed by detector stage and pooling*

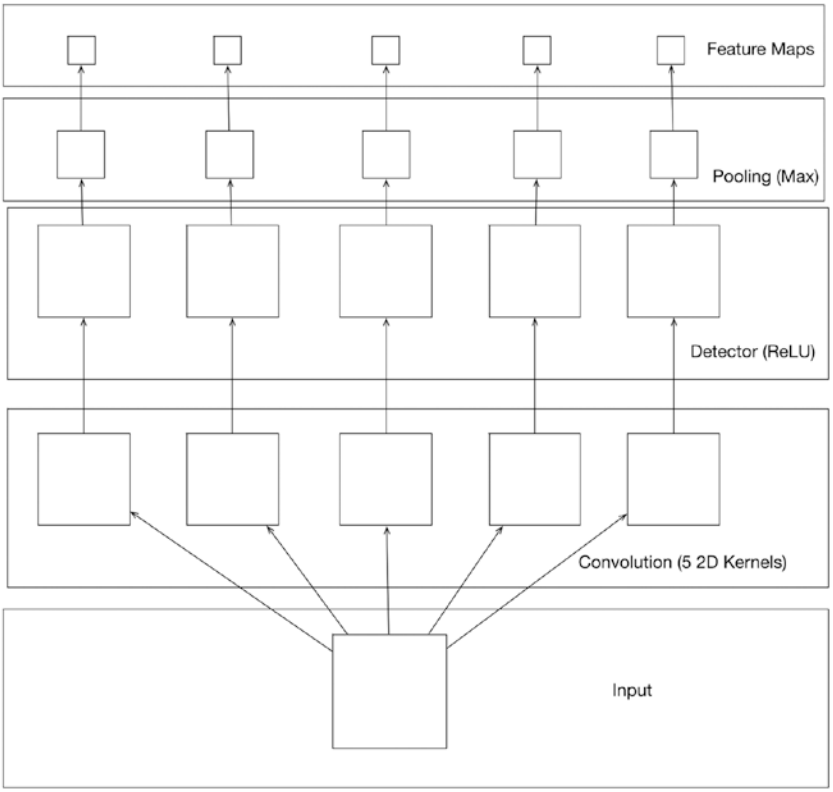


Figure 5-9. *Multiple Filters/Kernels giving Multiple Feature Maps*

In the case of image inputs, which consist of 3 channels, a separate convolution operation is applied to each channel and then outputs post the convolution are added up. This is illustrated in Figure 5-10.

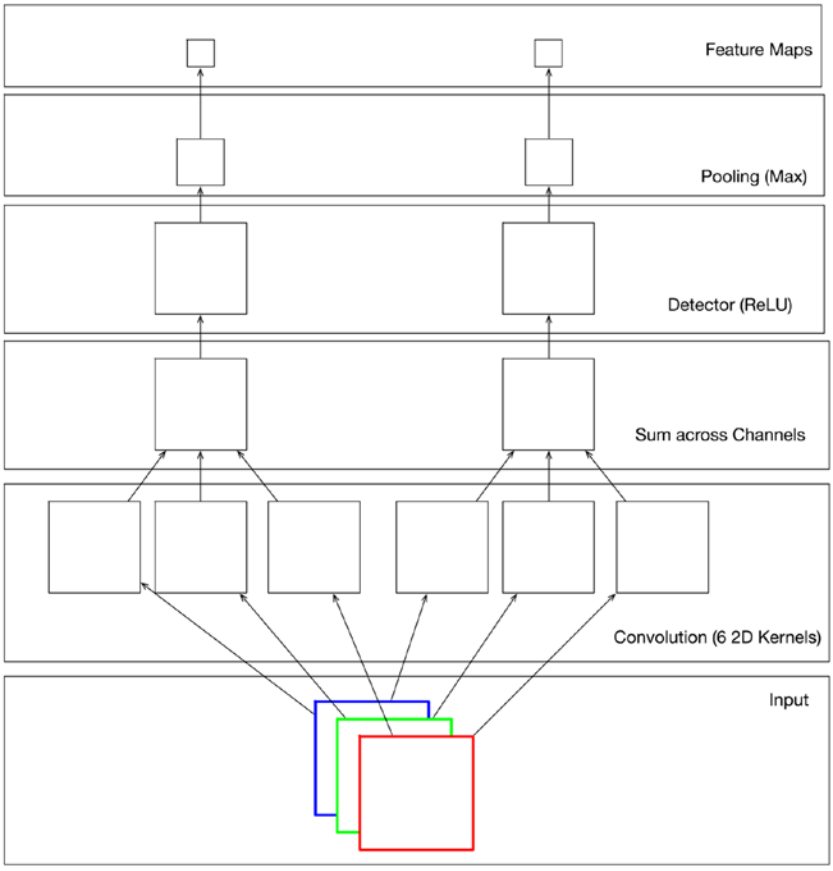


Figure 5-10. *Convolution with Multiple Channels*

Having covered all the constituent elements of CNNs, we can now look at an exemplar CNN in its entirety as illustrated in Figure 5-11. The CNN consists of two stages of convolution-detector-pooling blocks with multiple filters/kernels at each stage producing multiple feature maps. Post these two stages we have a fully connected layer which produces the output. In general, a CNN may have multiple stages of convolution-detector-pooling blocks (employing multiple filters) typically followed by a fully connected layer.

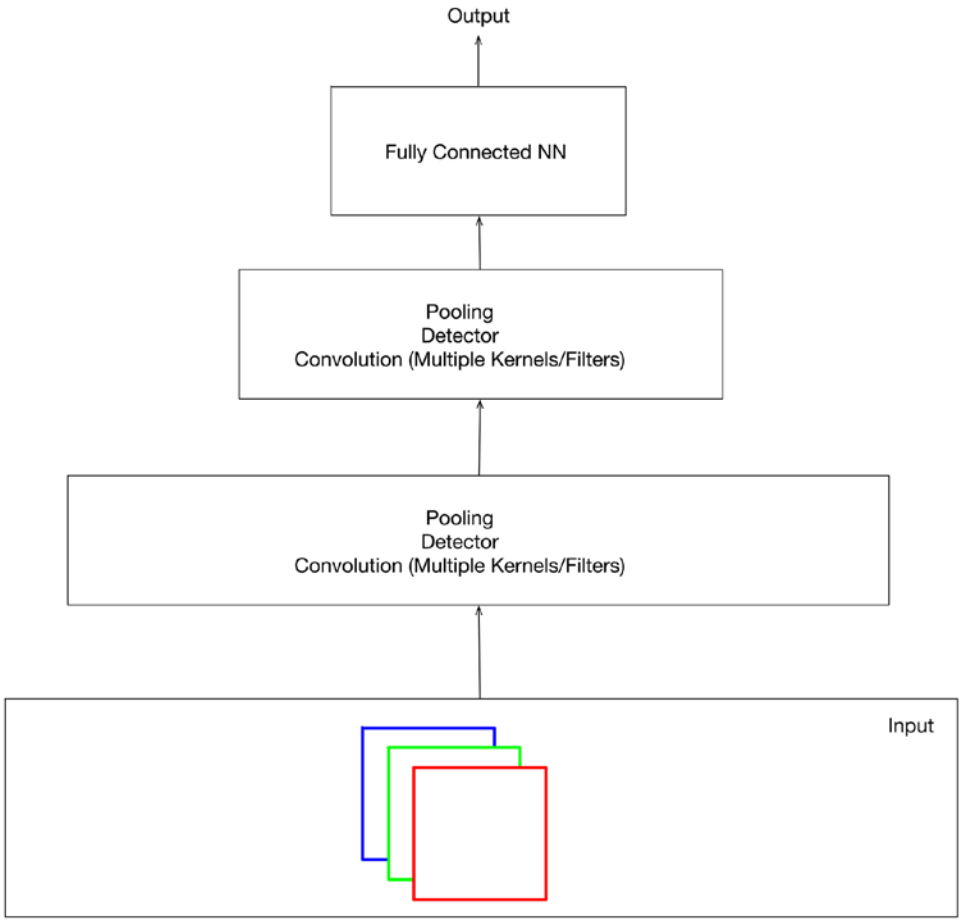


Figure 5-11. *A Complete Convolution Neural Network Architecture*

Convolution Variants

We will now cover some variations of convolution, illustrated in Figure 5-12. Strided convolution is a variant of the standard convolution where the kernel slides over the input by moving at a predefined stride. An alternative way of looking at this is that the standard convolution operated at a stride size equal to one. Another variation is tiled convolution where there are actually multiple kernels that are convolved with the input alternately.

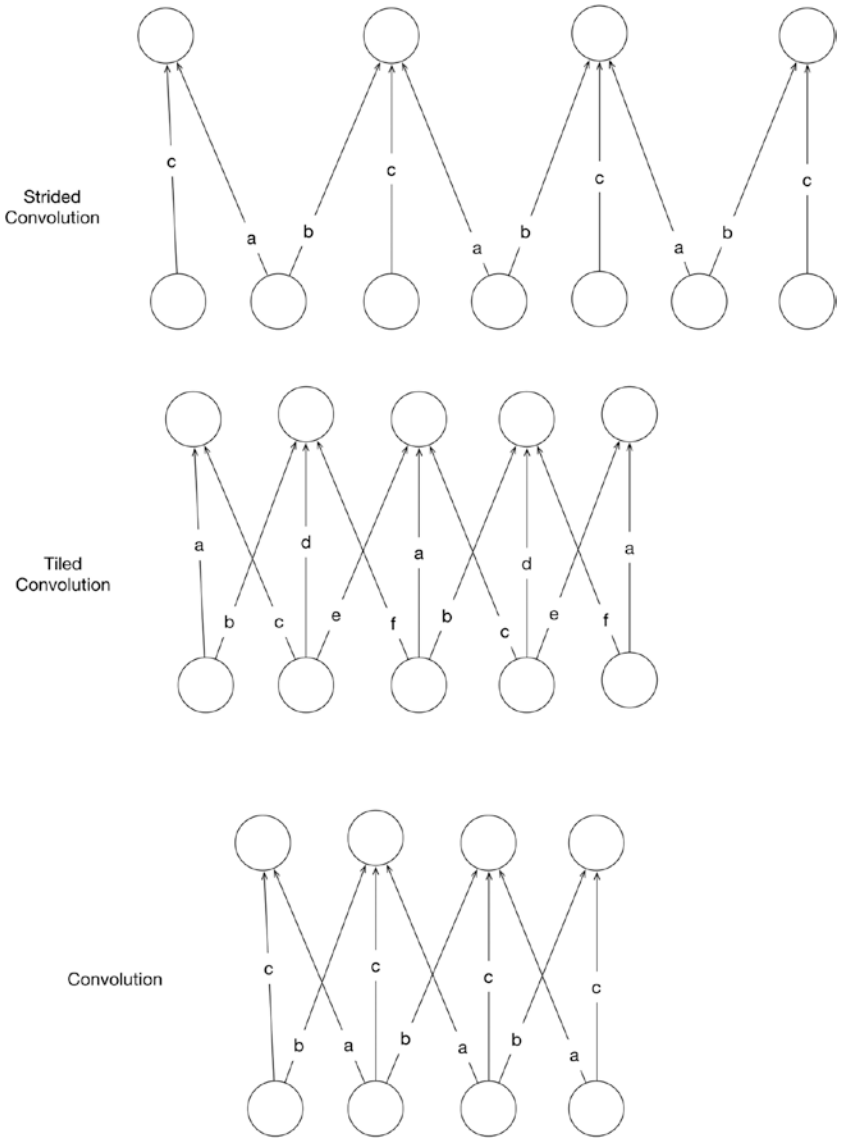


Figure 5-12. *Convolution, variation on the theme*

Another variation on the theme is locally connected layers which basically employ sparsity of interactions but do not employ parameter/weight sharing. This is illustrated in Figure 5-13.

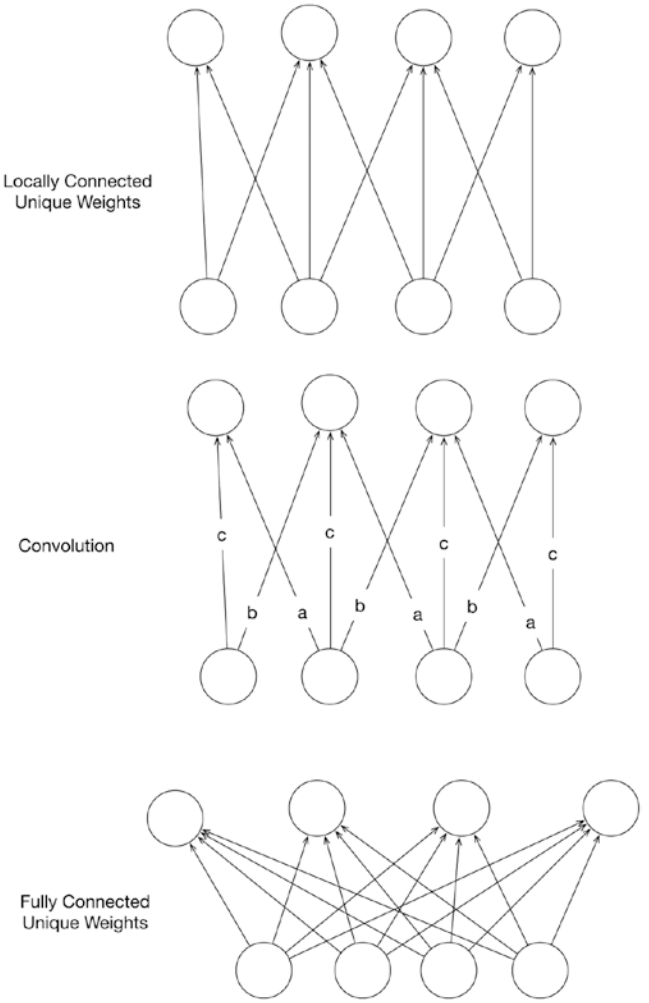


Figure 5-13. *Locally Connected Weights*

Intuition behind CNNs

So far in this chapter we have covered the key constituent concepts behind the CNN, namely the convolution operation, the pooling operation, and how they are used in conjunction. Let us now take a step back to internalize the intuition behind CNNs using these building blocks.

The first idea to consider is the capacity of CNNs (refer to Chapter 2 on the capacity of a machine learning model). CNNs, which replace at least one of the fully connected layers of a neural network by the convolution operation, have less capacity than a fully connected network. That is, there exist data sets that a fully connected network will be able to model that a CNN will not be. So, the first point to note is that CNNs achieve more by limiting the capacity, hence making the training efficient.

The second idea to consider is that learning the filters driving the convolution operation is, in a sense, representation learning. For instance, the learned filters might learn to detect edges, shapes, etc. The important point to consider here is that we are not manually describing the features to be extracted from the input data, but are describing an architecture that learns to engineer the features/representations.

The third idea to consider is the location invariance introduced by the pooling operation. The pooling operation separates the location of the feature from the fact that it is detected. A filter detecting straight lines might detect this filter in any portion of the image, but the pooling operation picks the fact that the feature is detected (max pooling).

The fourth idea is that of hierarchy. A CNN may have multiple convolution and pooling layers stacked up followed by a fully connected network. This allows the CNN to build a hierarchy of concepts wherein more abstract concepts are based on simpler concepts (refer to Chapter 1).

The fifth and last idea is the presence of a fully connected layer at the end of a series convolution and pooling layers. The idea is that the series of convolution and pooling layers generates the features and a standard neural network learns the final classification/regression function. It is important to distinguish this aspect of the CNN from traditional machine learning. In traditional machine learning, an expert would hand engineer features and feed them to a neural network. In the case of CNNs, these features/representations are being learned from data.

Summary

In this chapter we covered the basics of CNNs. The key takeaway points are the convolution operation, the pooling operation, how they are used in conjunction, and how features are not hand engineered but learned. CNNs are the most successful application of deep learning and embody the idea of learning features/representations rather than hand engineering them.

CHAPTER 6

Recurrent Neural Networks

Recurrent Neural Networks (RNNs) in essence are neural networks that employ recurrence, which is basically using information from a previous forward pass over the neural network. Essentially, all RNN's can be described as a recurrence relationship. RNNs are suited and have been incredibly successful when applied to problems wherein the input data on which the predictions are to be made is in the form of a sequence (series of entities where order is important).

RNN Basics

Let us start by describing the moving parts of a RNN. First, we introduce some notation.

1. We will assume that input consists of a sequence of entities $x^{(1)}, x^{(2)}, \ldots, x^{(\tau)}$.
2. Corresponding to this input we either need to produce a sequence $y^{(1)}, y^{(2)}, \ldots, y^{(\tau)}$ or just one output for the entire input sequence y
3. To distinguish between what the RNN produces and what it is ideally expected to produce we will denote by $\hat{y}^{(1)}, \hat{y}^{(2)}, \ldots, \hat{y}^{(\tau)}$ or $\hat{y}$ the output the RNN produces. Note that this is distinct from what the RNN should ideally produce, which is denoted by $y^{(1)}, y^{(2)}, \ldots, y^{(\tau)}$ or y.

RNNs either produce an output for every entity in the input sequence or produce a single output for the entire sequence. Let us consider the case where an RNN produces one output for every entity in the input. The RNN can be described using the following equations:

$$h^{(t)} = \tanh\left(Ux^{(t)} + Wh^{(t-1)} + b\right)$$

$$\hat{y}^{(t)} = softmax\left(Vh^{(t)} + c\right)$$

The following points about the RNN equations should be noted:

1. The RNN computation involves first computing the hidden state for an entity in the sequence. This is denoted by $h^{(t)}$.
2. The computation of $h^{(t)}$ uses the corresponding input at entity $x^{(t)}$ and the previous hidden state $h^{(t-1)}$.
3. The output $\hat{y}^{(t)}$ is computed using the hidden state $h^{(t)}$.
4. There are weights associated with the input and the previous hidden state while computing the current hidden state. This is denoted by U and W respectively. There is also a bias term denoted by b.

N. Ketkar, *Deep Learning with Python*, DOI 10.1007/978-1-4842-2766-4_6

5. There are weights associated with the hidden state while computing the output; this is denoted by V. There is also a bias term, which is denoted by c.
6. The tanh activation function (introduced in earlier chapters) is used in the computation of the hidden state.
7. The softmax activation function is used in the computation of the output.
8. The RNN as described by the equations can process an arbitrarily large input sequence.
9. The parameters of the RNN, namely, U,W,V,b,c, etc. are shared across the computation of the hidden layer and output value (for each of the entities in the sequence).

Figure 6-1 illustrates an RNN. Note how the illustration depicts the recurrence relation with the self-loop at the hidden state.

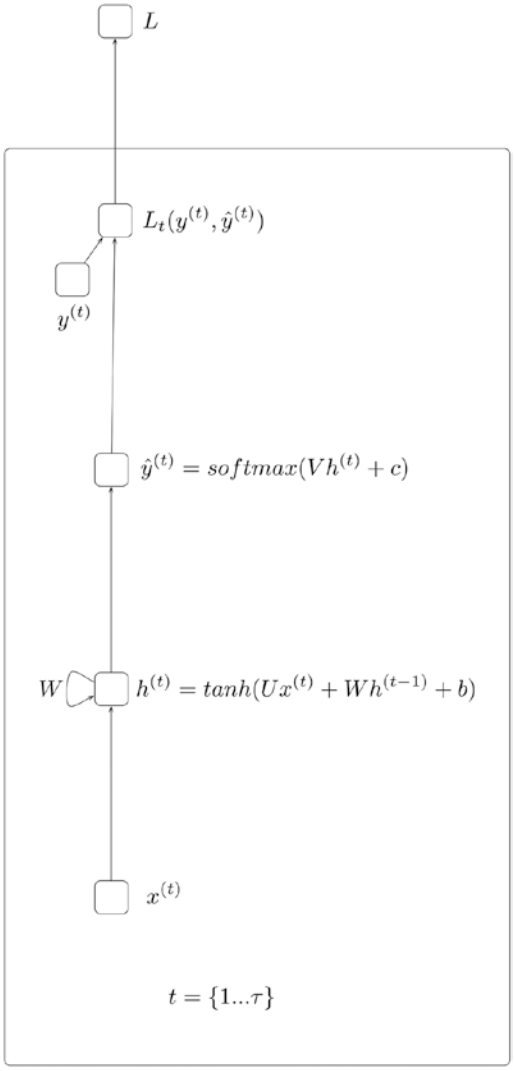


Figure 6-1. *RNN (Recurrence using the previous hidden state)*

The figure also depicts a loss function associated with each output associated with each input. We will refer back to it when we cover how RNNs are trained.

Presently, it's essential to internalize how an RNN is different from all the feedforward neural networks (including convolution networks) we have seen earlier. The key difference is the hidden state, which represents a summary of the entities seen in the past (for the same sequence).

Ignoring for the time being how RNNs are trained, it should be clear to the reader how a trained RNN could be used. For a given sequence of inputs, an RNN would produce an output for each entity in the input.

Let us now consider a variation in the RNN wherein instead of the recurrence using the hidden state, we have recurrence using the output produced in the previous state (refer to Figure 6-2).

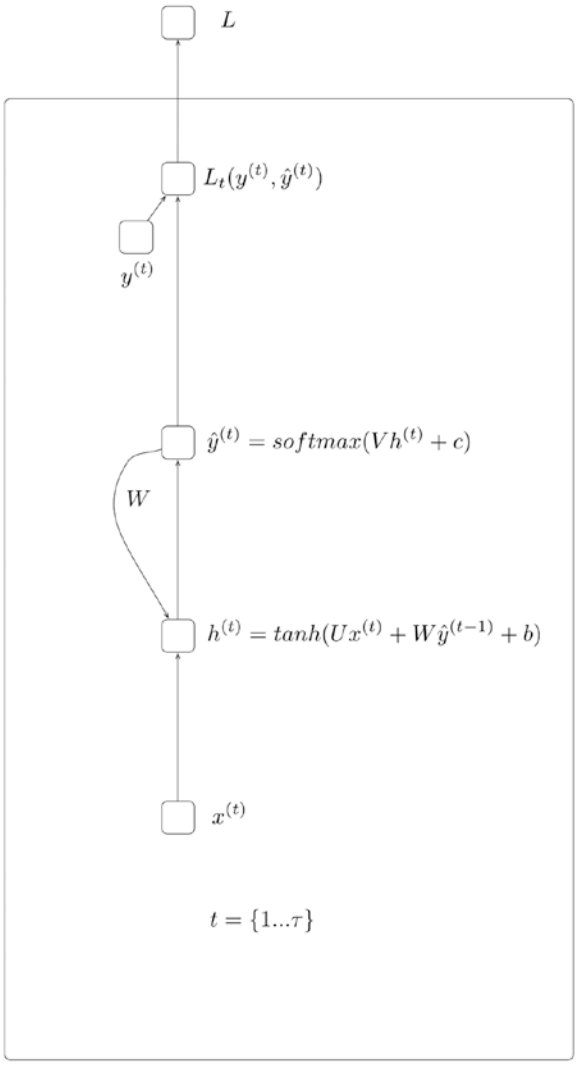


Figure 6-2. *RNN (Recurrence using the previous output)*

The equations describing such an RNN are as follows:

$$h^{(t)} = \tanh(Ux^{(t)} + W\hat{y}^{(t-1)} + b)$$

$$\hat{y}^{(t)} = softmax\left(Vh^{(t)} + c\right)$$

The following points are to be noted:

1. The RNN computation involves first computing the hidden state for an entity in the sequence. This is denoted by $h^{(t)}$.
2. The computation of $h^{(t)}$ uses the corresponding input at entity $x^{(t)}$ and the previous output $\hat{y}^{(t-1)}$.
3. The output $\hat{y}^{(t)}$ is computed using the hidden state $h^{(t)}$.
4. There are weights associated with the input and the previous output while computing the current hidden state. This is denoted by U and W respectively. There is also a bias term denoted by c.
5. There are weights associated with the hidden state while computing the output; this is denoted by V. There is also a bias term, which is denoted by c.
6. The tanh activation function (introduced in earlier chapters) is used in the computation of the hidden state.

The softmax activation function is used in the computation of the output.

Let us now consider a variation in the RNN where only a single output is produced for the entire sequence (refer to Figure 6-3). Such an RNN is described using the following equations:

$$h^{(t)} = \tanh(Ux^{(t)} + W\hat{y}^{(t-1)} + b)$$

$$\hat{y} = softmax\left(Vh^{(\tau)} + c\right)$$

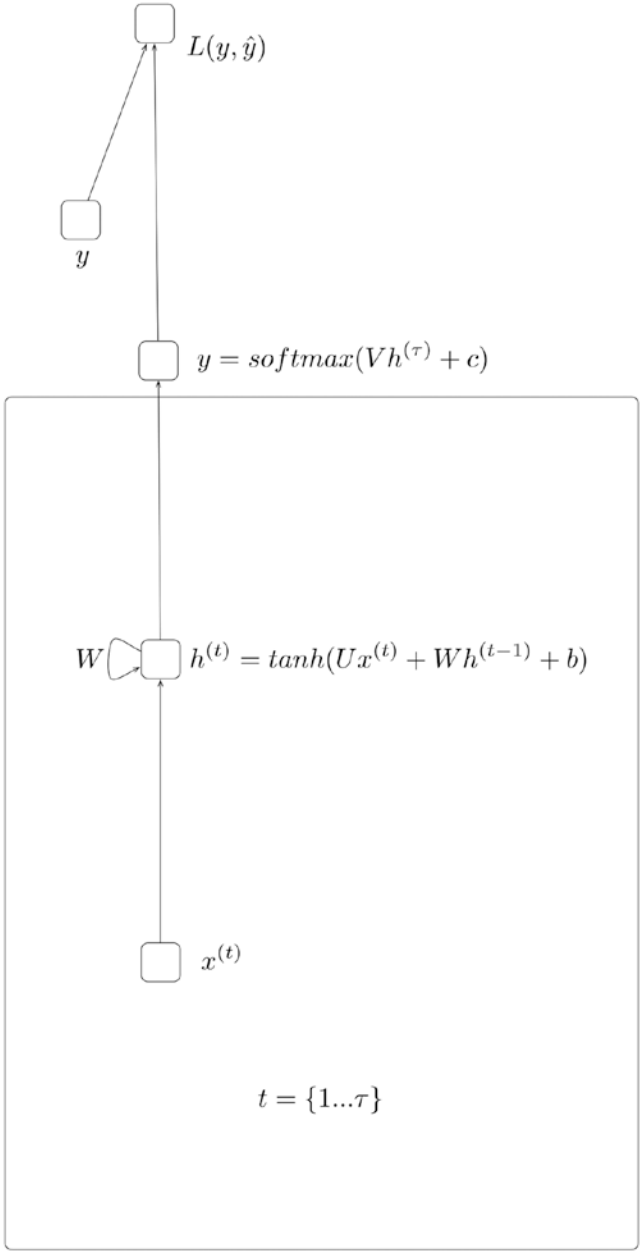


Figure 6-3. *RNN (Producing a single output for the entire input sequence)*

The following points are to be noted:

1. The RNN computation involves computing the hidden state for an entity in the sequence. This is denoted by $h^{(t)}$.
2. The computation of $h^{(t)}$ uses the corresponding input at entity $x^{(t)}$ and the previous hidden state $h^{(t-1)}$.
3. The computation of $h^{(t)}$ is done for each entity in the input sequence $x^{(1)}, x^{(2)}, \ldots, x^{(\tau)}$.
4. The output $\hat{y}$ is computed only using the last hidden state $h^{(\tau)}$.
5. There are weights associated with the input and the previous hidden state while computing the current hidden state. This is denoted by U and W respectively. There is also a bias term denoted by b.
6. There are weights associated with the hidden state while computing the output; this is denoted by V. There is also a bias term, which is denoted by c.
7. The tanh activation function (introduced in earlier chapters) is used in the computation of the hidden state.
8. The softmax activation function is used in the computation of the output.

Training RNNs

Let us now look at how RNNS are trained. To do this, we first need to look at how the RNN looks when we unroll the recurrence relation which is at the heart of the RNN.

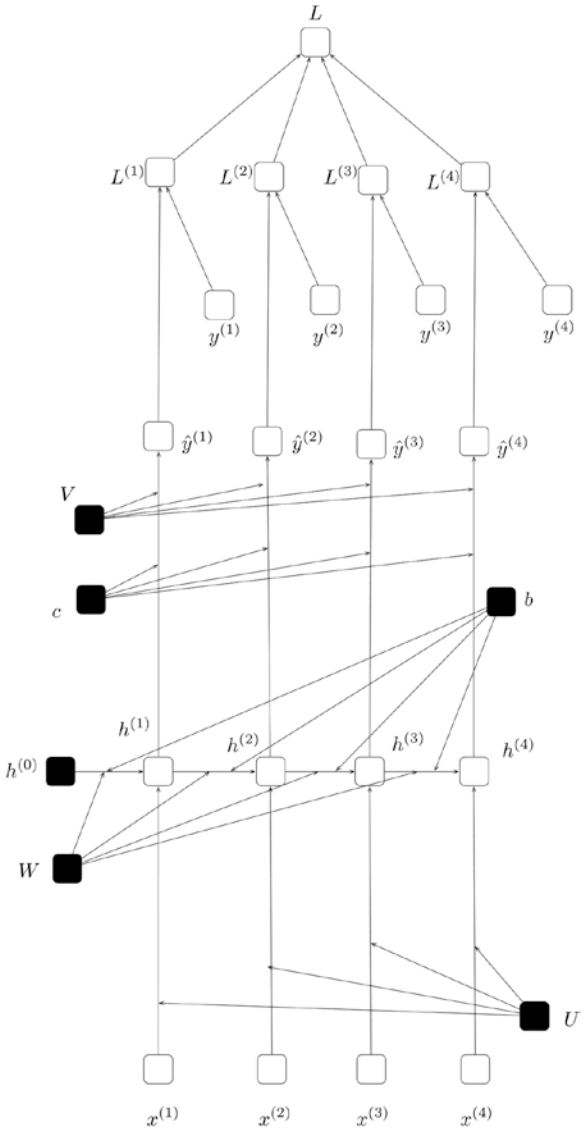


Figure 6-4. *Unrolling the RNN corresponding to Figure 6-1*

Unrolling the recurrence relation corresponding to RNN is simply writing out the equations by recursively substituting the value on which recurrence relation is defined. In the case of the RNN in Figure 6-1, this is $h^{(t)}$. That is, the value of $h^{(t)}$ is defined in terms of $h^{(t-1)}$, which in turn is defined in terms of $h^{(t-2)}$ and so on till $h^{(0)}$.

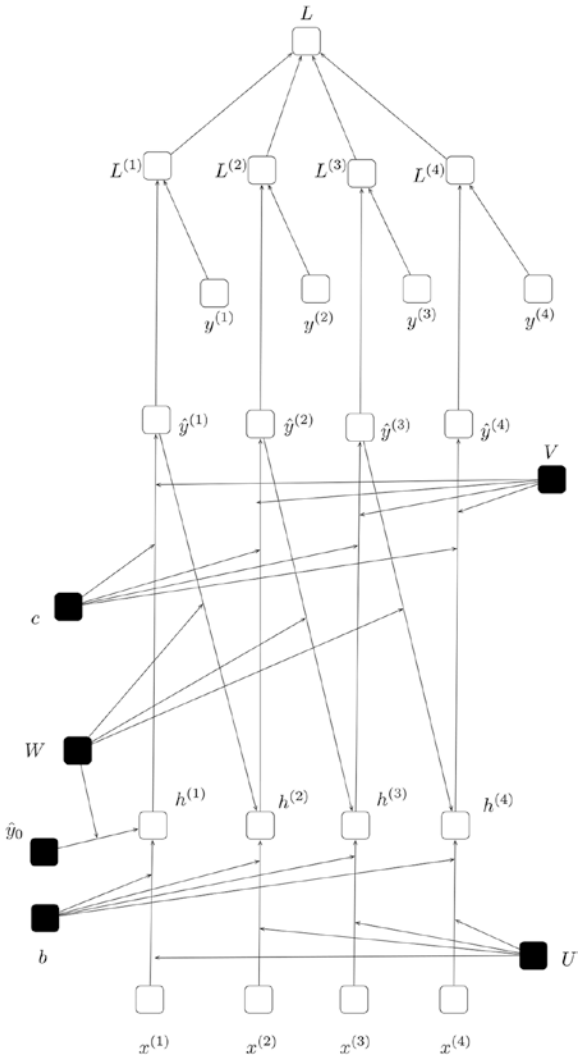


Figure 6-5. *Unrolling the RNN corresponding to Figure 6-2*

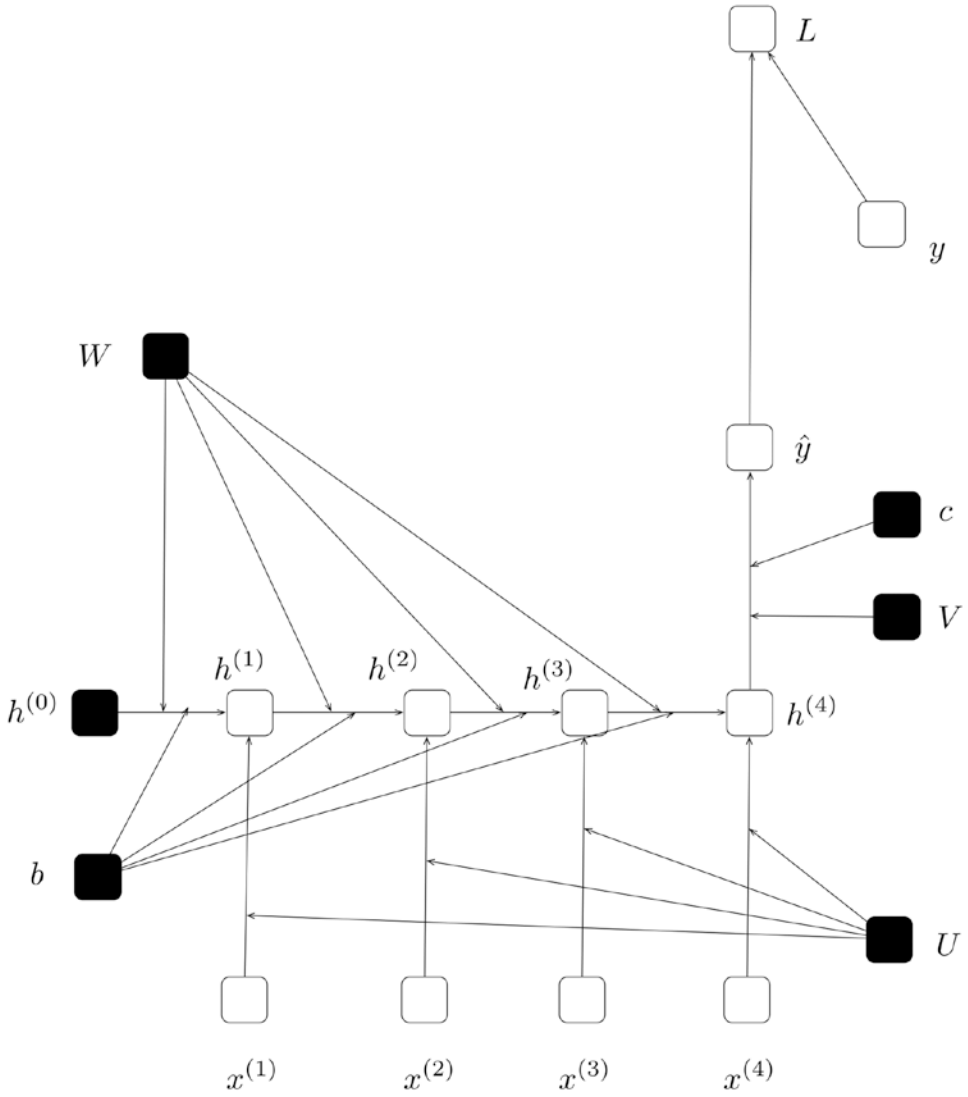


Figure 6-6. *Unrolling the RNN corresponding to Figure 6-3*

We will assume that $h^{(0)}$ is either predefined by the user, set to zero, or learned as another parameter/weight (learned like W, V, or b). Unrolling simply means writing out the equations describing the RNN in terms of $h^{(0)}$. Of course, in order to do so, we need fix the length of the sequence, which is denoted by τ. Figure 6-4 illustrates the unrolled RNN corresponding to the RNN in Figure 6-1 assuming an input sequence of size 4. Similarly, Figure 6-5 and 6-6 illustrate the unrolled RNNs corresponding to the RNNs in Figure 6-2 and 6-3 respectively. The following points are to be noted:

1. The unrolling process operates on the assumption that the length of the input sequence is known beforehand and based on the recurrence is unrolled.
2. Once unrolled, we essentially have a non-recurrent neural network.

3. The parameters to be learned, namely, U,W,V,b,c, etc. (denoted in dark in the diagram) are shared across the computation of the hidden layer and output value. We have seen such parameter sharing earlier in the context of CNNs.

4. Given an input and output of a given size, say τ (assumed to be 4 in Figures 6-4, 6-5, 6-6), we can unroll an RNN and compute gradients for the parameters to be learned with respect to a loss function (as we have seen in earlier chapters).

5. Thus, training an RNN is simply unrolling the RNN for a given size of input (and, correspondingly, the expected output) and training the unrolled RNN via computing the gradients and using stochastic gradient descent.

As mentioned earlier in the chapter, RNNs can deal with arbitrarily long inputs and correspondingly, they need to be trained on arbitrarily long inputs. Figure 6-7 illustrates how an RNN is unrolled for different sizes of inputs. Note that once the RNN is unrolled, the process of training the RNN is identical to training a regular neural network which we have covered in earlier chapters. In Figure 6-7 the RNN described in Figure 6-1 is unrolled for input sizes of 1,2,3 and 4.

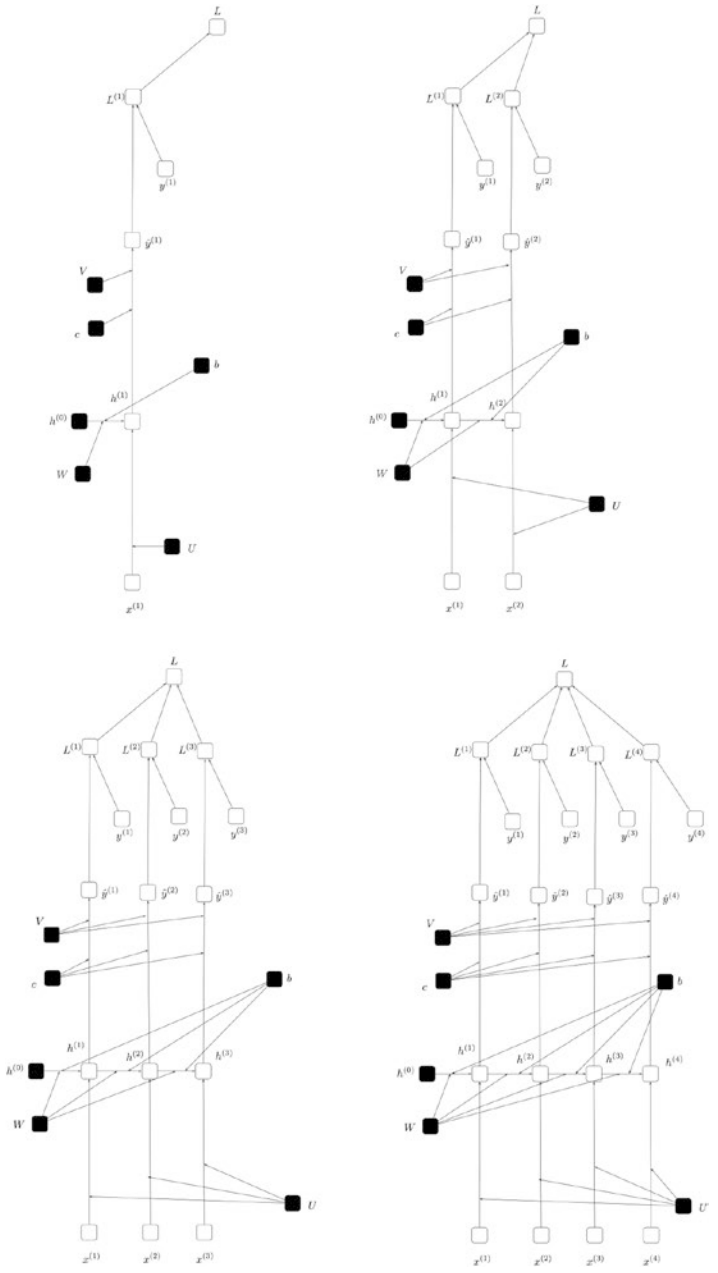


Figure 6-7. *Unrolling the RNN corresponding to Figure 6-1 for different sizes of inputs*

Given that the data set to be trained on consists of sequences of varying sizes, the input sequences are grouped so that the sequences of the same size fall in one group. Then for a group, we can unroll the RNN for the sequence length and train. Training for a different group will require the RNN to be unrolled for a different sequence length. Thus, it is possible to train the RNN on inputs of varying sizes by unrolling and training with the unrolling done based on the sequence length.

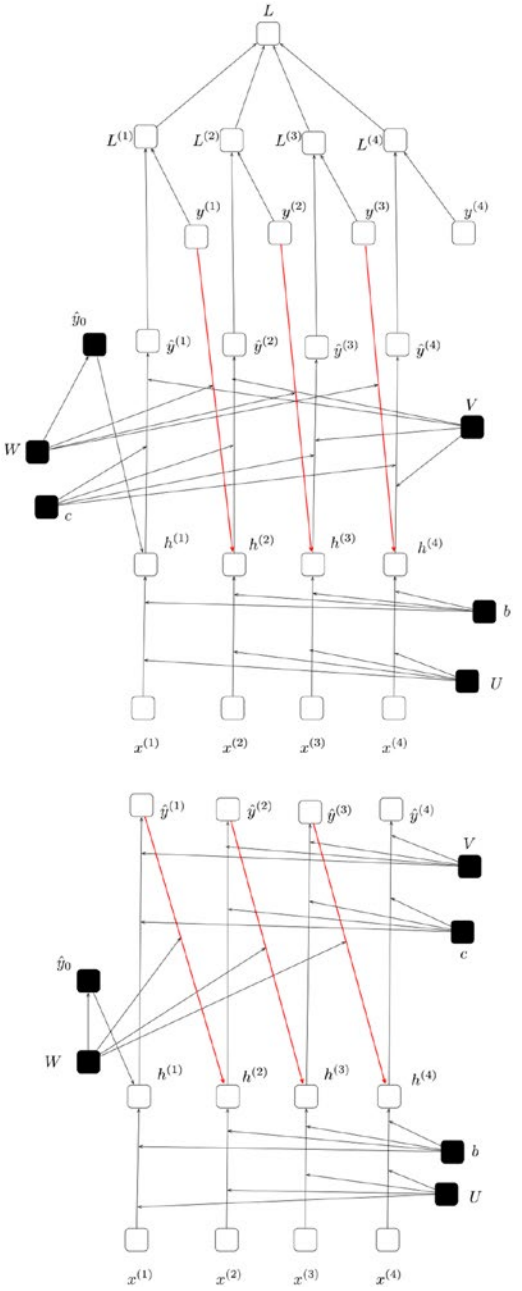


Figure 6-8. *Teacher Forcing (Top – Training, Bottom - Prediction)*

It must be noted that training the unrolled RNN (illustrated in Figure 6-1) is essentially a sequential process, as the hidden states are dependent on each other. In the case of RNNs wherein the recurrence is over the output instead of the hidden state (Figure 6-2), it is possible to use a technique called teacher forcing as illustrated in Figure 6-8. The key idea here is to use $y^{(t-1)}$ instead of $\hat{y}^{(t-1)}$ in the computation of $h^{(t)}$ while training. While making predictions (when the model is deployed for usage), however, $\hat{y}^{(t-1)}$ is used.

Bidirectional RNNs

Let us now take a look at another variation on RNNs, namely, the bidirectional RNN. The key intuition behind a bidirectional RNN is to use the entities that lie further in the sequence to make a prediction for the current entity. For all the RNNs we have considered so far we have been using the previous entities (captured by the hidden state) and the current entity in the sequence to make the prediction. However, we have not been using information concerning the entities that lie further in the sequence to make predictions. A bidirectional RNN leverages this information and can give improved predictive accuracy in many cases.

A bidirectional RNN can be described using the following equations:

$$h_f^{(t)} = \tanh\left(U_f x^{(t)} + W_f h^{(t+1)} + b_f\right)$$

$$h_b^{(t)} = \tanh\left(U_b x^{(t)} + W_b h^{(t-1)} + b_b\right)$$

$$\hat{y}^{(t)} = softmax\left(V_b h_b^{(t)} + V_f h_f^{(t)} + c\right)$$

The following points are to be noted:

1. The RNN computation involves first computing the forward hidden state and backward hidden state for an entity in the sequence. This is denoted by $h_f^{(t)}$ and $h_b^{(t)}$ respectively.
2. The computation of $h_f^{(t)}$ uses the corresponding input at entity $x^{(t)}$ and the previous hidden state $h_f^{(t-1)}$.
3. The computation of $h_b^{(t)}$ uses the corresponding input at entity $x^{(t)}$ and the previous hidden state $h_b^{(t-1)}$.
4. The output $\hat{y}^{(t)}$ is computed using the hidden state $h_f^{(t)}$ and $h_b^{(t)}$
5. There are weights associated with the input and the previous hidden state while computing the current hidden state. This is denoted by U_f, W_f, U_b, and W_b respectively. There is also a bias term denoted by b_f and b_b.
6. There are weights associated with the hidden state while computing the output; this is denoted by V_b and V_f. There is also a bias term, which is denoted by c.
7. The tanh activation function (introduced in earlier chapters) is used in the computation of the hidden state.
8. The softmax activation function is used in the computation of the output.
9. The RNN as described by the equations can process an arbitrarily large input sequence.
10. The parameters of the RNN, namely, U_f, U_b, W_f, W_b, V_b, V_f, b_f, b_b, c, etc. are shared across the computation of the hidden layer and output value (for each of the entities in the sequence).

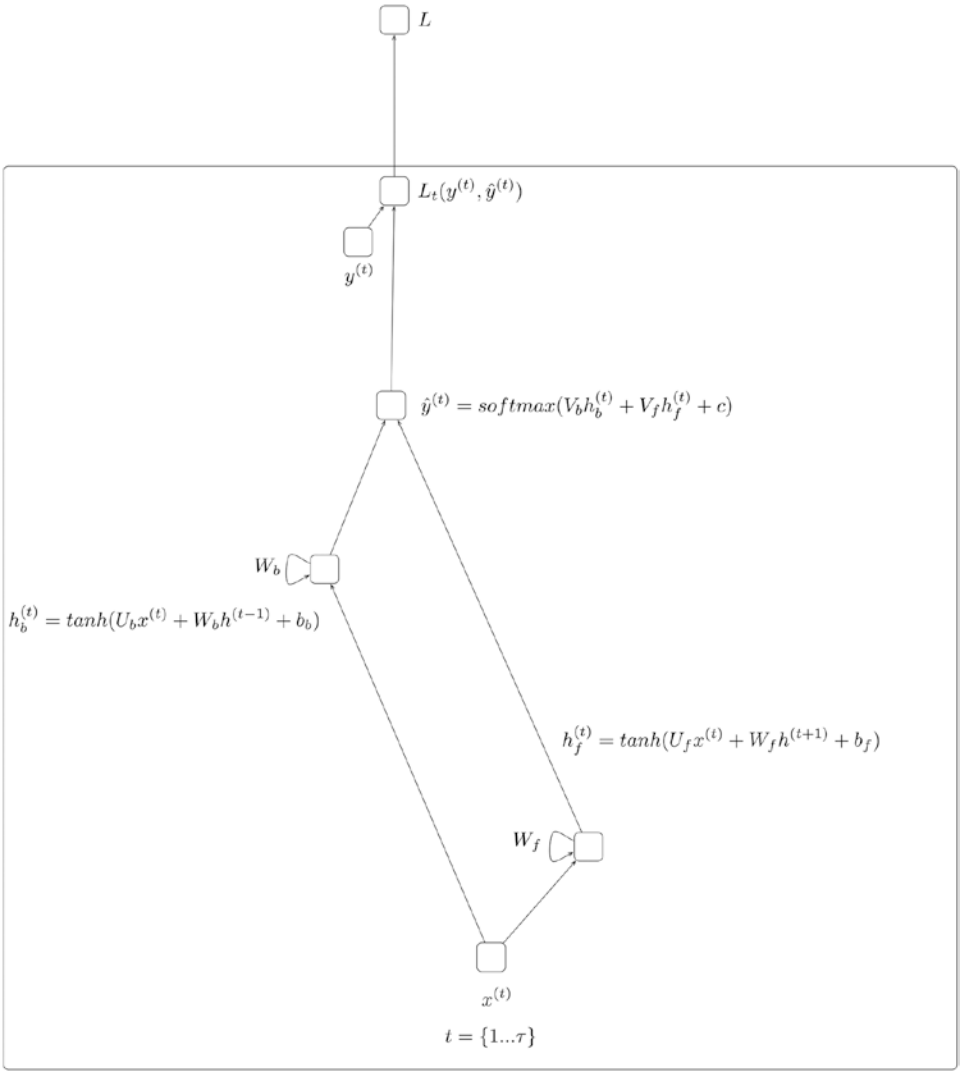


Figure 6-9. *Bidirectional RNN*

Gradient Explosion and Vanishing

Training RNNs suffers from the challenges of vanishing and explosion of gradients. Vanishing gradients means that, when computing the gradients on the unrolled RNNs, the value of the gradients can drop to a very small value (close to zero). Similarly, the gradients can increase to a very high value which is referred to as the exploding gradient problem. In both cases, training the RNN is a challenge.

Let us relook at the equations describing the RNN.

$$h^{(t)} = \tanh\left(Ux^{(t)} + Wh^{(t-1)} + b\right)$$

$$\hat{y}^{(t)} = softmax\left(Vh^{(t)} + c\right)$$

Let us derive the expression for the $\frac{\partial L}{\partial W}$ by applying the chain rule. This is illustrated in Figure 6-10.

$$\frac{\partial L}{\partial W} = \sum_{1 \le t \le \tau} \frac{\partial L^{(t)}}{\partial h^{(t)}} \left[\sum_{1 \le k \le t} \left[\prod_{k \le j \le t-1} \frac{\partial h^{(j+1)}}{\partial h^{(j)}} \right] \frac{\partial h^{(k)}}{\partial W} \right]$$

Let us now focus on the part of the expression $\prod_{k \le j \le t-1} \frac{\partial h^{(j+1)}}{\partial h^{(j)}}$ which involves a repeated matrix multiplication of W which contributes to both the vanishing and exploding gradient problems. Intuitively, this is similar to multiplying a real valued number over and over again, which might lead to the product shrinking to zero or exploding to infinity.

Gradient Clipping

One simple technique to deal with exploding gradients is to rescale the norm of gradient whenever it goes over a user-defined threshold. Specifically, if the gradient denoted by $\hat{g} = \frac{\partial L}{\partial W}$ and if $|\hat{g}| > c$ then we set $\hat{g} = \frac{c}{|\hat{g}|}\hat{g}$. This technique is both simple and computationally efficient but does introduce an extra hyperparameter.

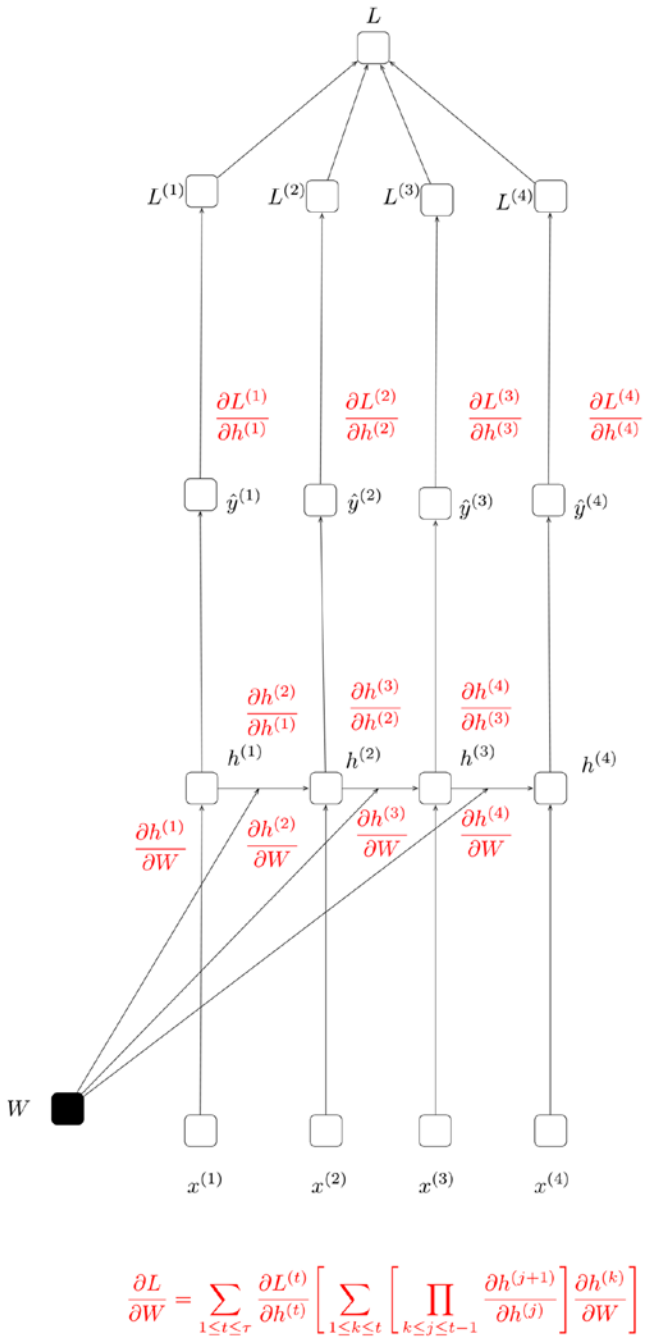


Figure 6-10. *Vanishing and Exploding Gradients*

Long Short Term Memory

Let us now take a look at another variation on RNNs, namely, the Long Short Term Memory (LSTM) Network. An LSTM can be described with the following set of equations. Note that the symbol $\odot$. notes pointwise multiplication of two vectors (if $a=[1,1,2]$ and $b=[0.5,0.5,0.5]$, then $a \odot b=[0.5,0.5,1]$, the functions σ, g and h are non-linear activation functions, all W and R are weight matrices, and all the b terms are bias terms).

$$z^{(t)} = g\left(W_z x^{(t)} + R_z \ \hat{y}^{(t-1)} + b_z\right)$$

$$i^{(t)} = \sigma\left(W_i x^{(t)} + R_i \ \hat{y}^{(t-1)} + p_i \odot c^{(t-1)} + b_i\right)$$

$$f^{(t)} = \sigma\left(W_f x^{t} + R_f \ \hat{y}^{(t-1)} + p_f \odot c^{(t-1)} + b_f\right)$$

$$c^{(t)} = i^{(t)} \odot z^{(t)} + f^{(t)} \odot c^{(t-1)}$$

$$o^{(t)} = \sigma\left(W_o x^{(t)} + R_o \ \hat{y}^{(t-1)} + p_o \odot c^{(t)} + b_o\right)$$

$$\hat{y}^{(t)} = o^{(t)} \odot h\left(c^{(t)}\right)$$

The following points are to be noted:

1. The most important element of the LSTM is the cell state denoted by $c^{(t)} = i^{(t)} \odot z^{(t)} + f^{(t)} \odot c^{(t-1)}$. The cell state is updated based on the block input $z^{(t)}$ and the previous cell state $c^{(t-1)}$. The input gate $i^{(t)}$ determines what fraction of the block input makes it into the cell state (hence called a gate). The forget gate $f^{(t)}$ determines how much of the previous cell state to retain.

2. The output $\hat{y}^{(t)}$ is determined based on the cell state $c^{(t)}$ and the output gate $o^{(t)}$, which determines how much the cell state affects the output.

3. The $z^{(t)}$ term is referred to as the block input and it produces a value based on the current input and the previous output.

4. The $i^{(t)}$ term is referred to as the input gate. It determines how much of the input to retain in the cell state $c^{(t)}$.

5. All the p terms are peephole connections, which allow for a faction of the cell state to factor into the computation of the term in question.

6. The computation of the cell state $c^{(t)}$ does not encounter the issue of the vanishing gradient (this is referred to as the constant error carousal). However, LSTMs are affected by exploding gradients and gradient clipping is used while training.

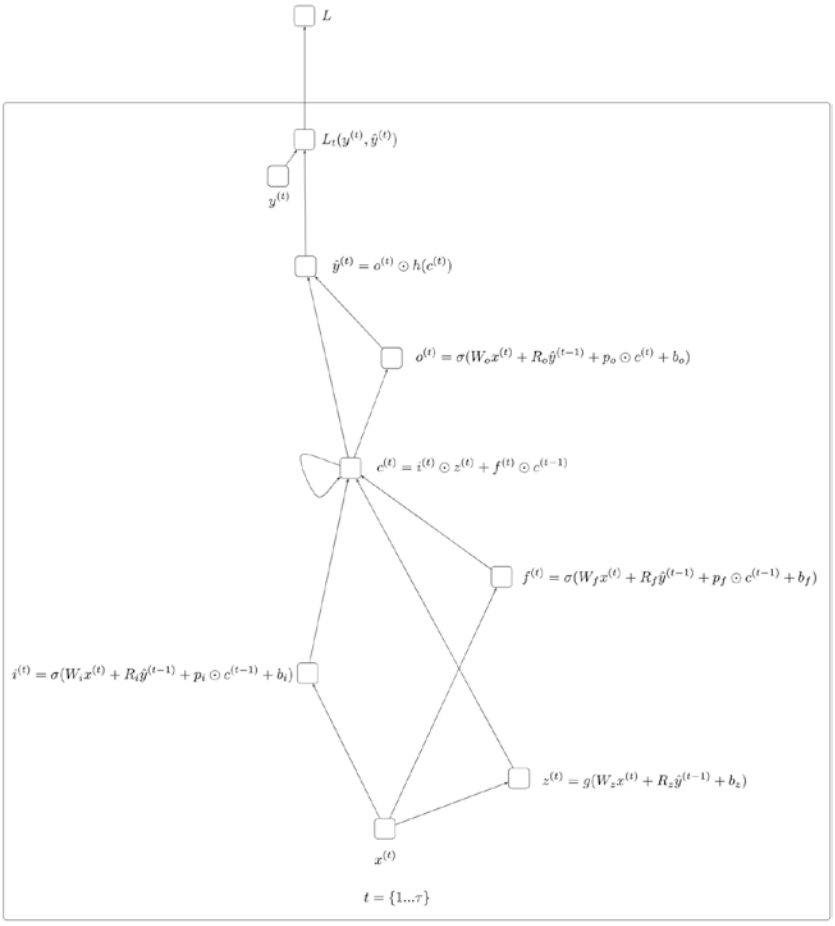


Figure 6-11. *Long Short Term Memory*

Summary

In this chapter we covered the basics of Recurrent Neural Networks (RNN). The key take-home points from this chapter are the notion of the hidden state, training RNNs via unrolling (backpropagation through time), the problem of vanishing and exploding gradients, and long short term memory networks. It is important to internalize how RNNs contain internal/hidden states that allow them to make predictions on a sequence of inputs, an ability that goes beyond conventional neural networks.

CHAPTER 7

Introduction to Keras

This chapter introduces the reader to Keras, which is a library that provides highly powerful and abstract building blocks to build deep learning networks. The building blocks Keras provides are built using Theano (covered earlier) as well as TensorFlow (which is an alternative to Theano for building computational graphs, automatically deriving gradients, etc.). Keras supports both CPU and GPU computation and is a great tool for quickly prototyping ideas.

We will introduce a number of key building blocks Keras provides, and then build a CNN and LSTM using Keras.

Let us start with a simple, single layer neural network. Listing 7-1 provides the code and Figure 7-1 gives the computational graph. The following points are to be noted:

1. A model is defined using the Sequential construct, which allows the user to add/configure layers.
2. Using this functionality, a user can add one or more layers and build the network. The Dense layer is basically a fully connected layer (leading to a vector-matrix or vector-vector product), which we have seen earlier.
3. The input and output dimensionality needs to be specified when the first layer is defined. In this case the model will take an input of dimensionality 500 and produce an output of dimensionality 1.
4. After this layer we add an activation function, in this case a sigmoid.
5. The model once defined needs to be explicitly compiled and, at this time, we provide the loss function, the optimization algorithm, and other metrics we want to calculate.
6. An appropriate loss function needs to be picked given the task at hand; in this case, given that we have a binary classification problem, we select binary cross-entropy.
7. An appropriate optimization algorithm needs to be picked, which typically is a variant of Stochastic Gradient Descent (coved in later chapters).
8. Once compiled we can fit the model by providing the data and evaluate the model.

N. Ketkar, *Deep Learning with Python*, DOI 10.1007/978-1-4842-2766-4_7

Listing 7-1. Single Layer Neural Network

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Activation
from keras.utils.visualize_util import plot

model = Sequential()
model.add(Dense(1, input_dim=500))
model.add(Activation(activation='sigmoid'))
model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['accuracy'])

data = np.random.random((1000, 500))
labels = np.random.randint(2, size=(1000, 1))

score = model.evaluate(data,labels, verbose=0)
print "Before Training:", zip(model.metrics_names, score)

model.fit(data, labels, nb_epoch=10, batch_size=32, verbose=0)

score = model.evaluate(data,labels, verbose=0)
print "After Training:", zip(model.metrics_names, score)
plot(model, to_file='s1.png', show_shapes=True)

# Before Training: [('loss', 0.76832762384414677), ('acc', 0.50700000000000001)]
# After Training: [('loss', 0.67270196056365972), ('acc', 0.56299999999999994)]
```

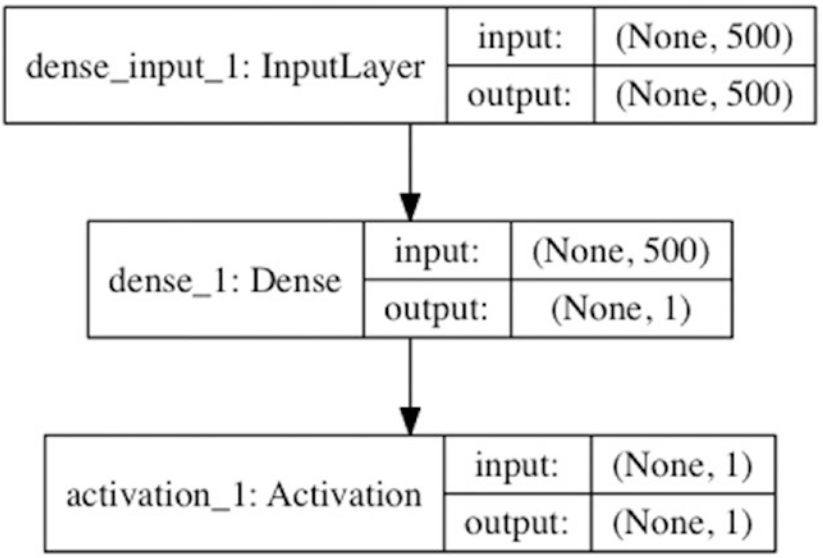


Figure 7-1. *Single Layer Neural Network (Binary Classification)*

Listing 7-2. Two Layer Neural Network

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Activation
from keras.utils.visualize_util import plot

model = Sequential()
model.add(Dense(32, input_dim=500))
model.add(Activation(activation='sigmoid'))
model.add(Dense(1))
model.add(Activation(activation='sigmoid'))
model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['accuracy'])

data = np.random.random((1000, 500))
labels = np.random.randint(2, size=(1000, 1))

score = model.evaluate(data,labels, verbose=0)
print "Before Training:", zip(model.metrics_names, score)

model.fit(data, labels, nb_epoch=10, batch_size=32, verbose=0)

score = model.evaluate(data,labels, verbose=0)
print "After Training:", zip(model.metrics_names, score)

plot(model, to_file='s2.png', show_shapes=True)

# Before Training: [('loss', 0.73012506151199341), ('acc', 0.51200000000000001)]
# After Training: [('loss', 0.6588478517532349), ('acc', 0.52700000000000002)]
```

Let us now look at a two-layer neural network. Listing 7-2 provides the code and Figure 7-2 gives the computational graph. The following points are to be noted:

1. A model is defined using the Sequential construct.
2. We add the first layer, using Dense, and specify the input dimensionality. In this case the model will take an input of dimensionality 500 and produce an output of dimensionality 32.
3. We define an activation function, selecting sigmoid.
4. We then define the second layer using Dense. Here we define the output dimensionality to be 1. Note, however, that we do not need to define the input dimensionality, as it is the same as the dimensionality of the output of the previous layer.
5. As before, we define the optimize and loss function, compile, train, and evaluate.

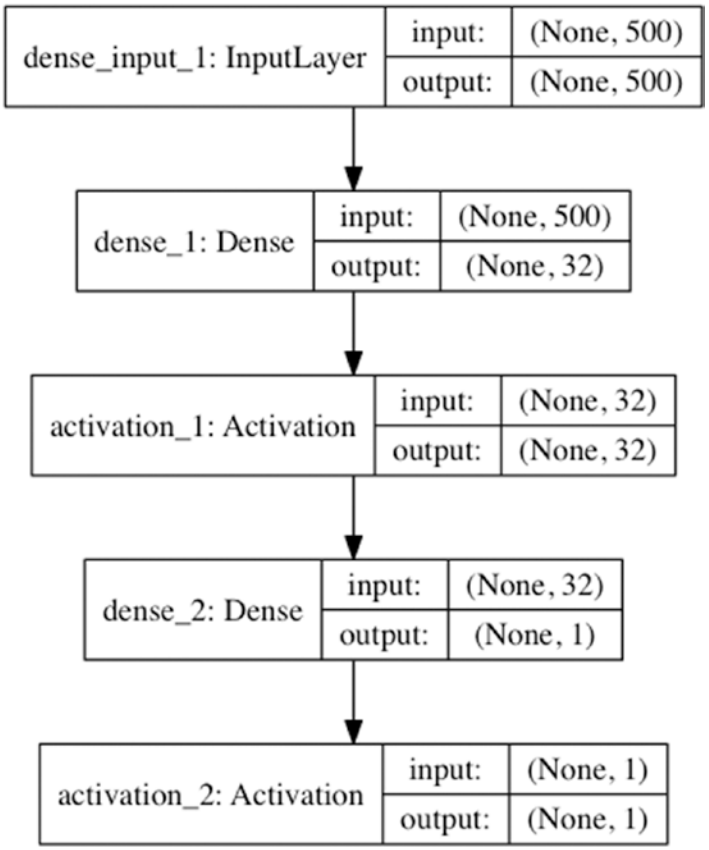


Figure 7-2. *Double Layer Neural Network (Binary Classification)*

Let us now look at a two-layer neural network for multiclass classification. Listing 7-3 provides the code and Figure 7-3 gives the computational graph. The following points are to be noted:

1. A model is defined using the Sequential construct.
2. We add the first layer, using Dense, and specify the input dimensionality. In this case the model will take an input of dimensionality 500 and produce an output of dimensionality 32.
3. We define an activation function, selecting sigmoid.
4. We then define the second layer using Dense. Here we define the output dimensionality to be 10. Note that this is exactly equal to the number of classes we have in our dataset.
5. Next we use the softmax activation and the categorical entropy as the loss function (an earlier chapter covers why this is a good choice).
6. We compile, train, and evaluate the model as before.

Listing 7-3. Multiclass Classification

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Activation
from keras.utils.np_utils import to_categorical
from keras.utils.visualize_util import plot

model = Sequential()
model.add(Dense(32, input_dim=500))
model.add(Activation(activation='relu'))
model.add(Dense(10))
model.add(Activation(activation='softmax'))
model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['categorical_
accuracy'])

data = np.random.random((1000, 500))
labels = to_categorical(np.random.randint(10, size=(1000, 1)))

score = model.evaluate(data,labels, verbose=0)
print "Before Training:", zip(model.metrics_names, score)

model.fit(data, labels, nb_epoch=10, batch_size=32, verbose=0)

score = model.evaluate(data,labels, verbose=0)
print "After Training:", zip(model.metrics_names, score)

plot(model, to_file='s3.png', show_shapes=True)

# Before Training: [('loss', 2.4697211952209472), ('categorical_accuracy',
0.092999999999999999)]
# After Training: [('loss', 2.1891849689483642), ('categorical_accuracy',
0.19400000000000001)]
```

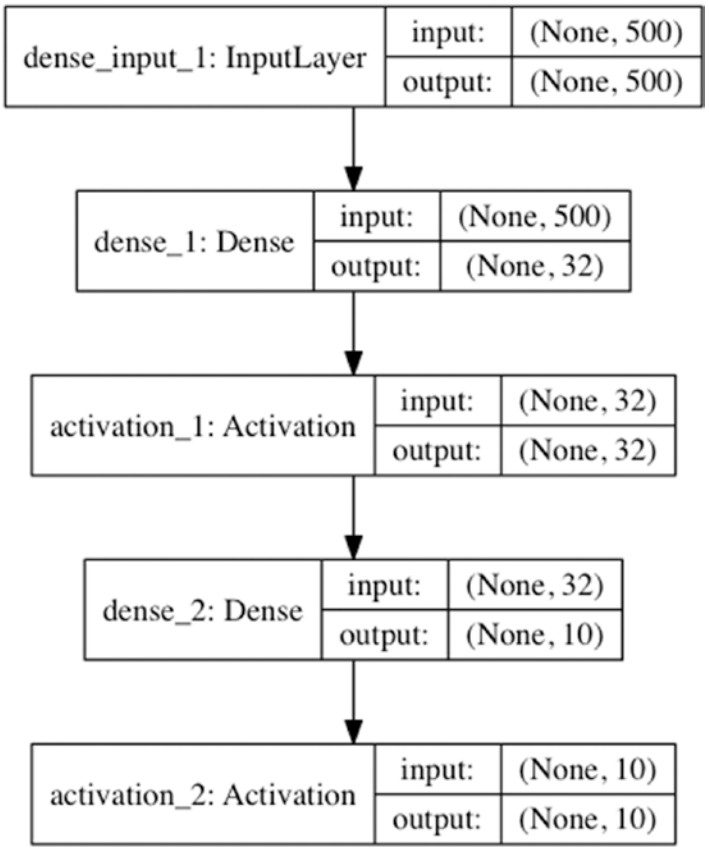


Figure 7-3. *Multiclass Classification*

Let us now look at a two-layer neural network for regression. Listing 7-4 provides the code and Figure 7-4 gives the computational graph. The following points are to be noted:

1. A model is defined using the Sequential construct.
2. We add the first layer, using Dense, and specify the input dimensionality. In this case the model will take an input of dimensionality 500 and produce an output of dimensionality 32.
3. We define an activation function, selecting sigmoid.
4. We then define the second layer using Dense, producing an output of dimensionality 1.
5. We select the activation as sigmoid and select mean squared error, which is appropriate for regression.
6. We compile, train, and evaluate the model.

Listing 7-4. Regression

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Activation
from keras.utils.visualize_util import plot

model = Sequential()
model.add(Dense(32, input_dim=500))
model.add(Activation(activation='sigmoid'))
model.add(Dense(1))
model.add(Activation(activation='sigmoid'))
model.compile(optimizer='rmsprop', loss='mse', metrics=['mean_squared_error'])

data = np.random.random((1000, 500))
labels = np.random.randint(2, size=(1000, 1))

score = model.evaluate(data,labels, verbose=0)
print "Before Training:", zip(model.metrics_names, score)

model.fit(data, labels, nb_epoch=10, batch_size=32, verbose=0)

score = model.evaluate(data,labels, verbose=0)
print "After Training:", zip(model.metrics_names, score)

plot(model, to_file='s4.png', show_shapes=True)

# Before Training: [('loss', 0.26870122766494753), ('mean_squared_error',
0.26870122766494753)]
# After Training: [('loss', 0.22180086207389832), ('mean_squared_error',
0.22180086207389832)]
```

Let us now take a pause and look at how Keras allows for quick iteration of ideas.

1. New models can be quickly defined, trained, and evaluated using the sequential construct.
2. The parameters of the layers input/output dimensionality can be easily modified.
3. We can compare multiple choices of activation functions easily. Listing 7-6 illustrates how we can compare the effects of activation functions.
4. We can compare multiple choices of optimization algorithms easily. Listing 7-5 illustrates how we can compare the effects of different choices of activation algorithms.

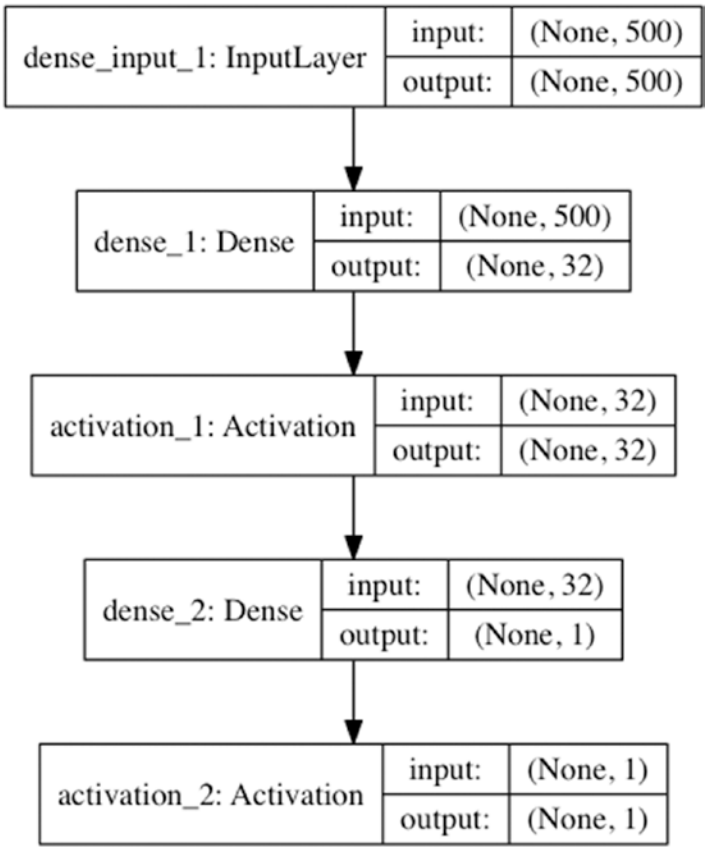


Figure 7-4. *Regression*

Listing 7-5. Optimisers

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Activation

def train_given_optimiser(optimiser):
    model = Sequential()
    model.add(Dense(1, input_dim=500))
    model.add(Activation(activation='sigmoid'))
    model.compile(optimizer=optimiser, loss='binary_crossentropy', metrics=['accuracy'])

    data = np.random.random((1000, 500))
    labels = np.random.randint(2, size=(1000, 1))

    score = model.evaluate(data,labels, verbose=0)
    print "Optimiser: ", optimiser
    print "Before Training:", zip(model.metrics_names, score)
```

```
    model.fit(data, labels, nb_epoch=10, batch_size=32, verbose=0)

    score = model.evaluate(data,labels, verbose=0)
    print "After Training:", zip(model.metrics_names, score)

train_given_optimiser("sgd")
train_given_optimiser("rmsprop")
train_given_optimiser("adagrad")
train_given_optimiser("adadelta")
train_given_optimiser("adam")
train_given_optimiser("adamax")
train_given_optimiser("nadam")

# Optimiser:  sgd
# Before Training: [('loss', 0.76416229248046874), ('acc', 0.51800000000000002)]
# After Training: [('loss', 0.6759231286048889), ('acc', 0.56899999999999995)]
# Optimiser:  rmsprop
# Before Training: [('loss', 0.77773557662963866), ('acc', 0.52600000000000002)]
# After Training: [('loss', 0.727150842666626), ('acc', 0.53500000000000003)]
# Optimiser:  adagrad
# Before Training: [('loss', 0.9275067367553711), ('acc', 0.49099999999999999)]
# After Training: [('loss', 0.66770141410827633), ('acc', 0.57599999999999996)]
# Optimiser:  adadelta
# Before Training: [('loss', 0.76523585319519039), ('acc', 0.48799999999999999)]
# After Training: [('loss', 0.70753741836547857), ('acc', 0.51700000000000002)]
# Optimiser:  adam
# Before Training: [('loss', 0.76974405097961429), ('acc', 0.51100000000000001)]

# After Training: [('loss', 0.66079518222808842), ('acc', 0.59399999999999997)]
# Optimiser:  adamax
# Before Training: [('loss', 0.76244759178161625), ('acc', 0.49399999999999999)]
# After Training: [('loss', 0.67273861455917361), ('acc', 0.58499999999999996)]
# Optimiser:  nadam
# Before Training: [('loss', 0.71690645027160649), ('acc', 0.50600000000000001)]
# After Training: [('loss', 0.62006913089752203), ('acc', 0.68799999999999994)]
```

Keras implements a number of optimisers, namely Stocastic Gradient Descent (SGD), RMSProp, AdaGrad, AdataDelta, Adam, Adamax, and Nadam. Chapter 8 covers these (SGD and its variants) in much detail, explaining the intuition for each. For the context of this chapter it suffices to say that Keras makes it easy for users to experiment with these optimisers with very little coding effort.

Listing 7-6. Activation Functions

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Activation

def train_given_activation(activation):
    model = Sequential()
    model.add(Dense(1, input_dim=500))
    model.add(Activation(activation=activation))
    model.compile(optimizer="sgd", loss='binary_crossentropy', metrics=['accuracy'])
```

```
    data = np.random.random((1000, 500))
    labels = np.random.randint(2, size=(1000, 1))

    score = model.evaluate(data,labels, verbose=0)
    print "Activation: ", activation
    print "Before Training:", zip(model.metrics_names, score)

    model.fit(data, labels, nb_epoch=10, batch_size=32, verbose=0)

    score = model.evaluate(data,labels, verbose=0)
    print "After Training:", zip(model.metrics_names, score)

train_given_activation("relu")
train_given_activation("tanh")
train_given_activation("sigmoid")
train_given_activation("hard_sigmoid")
train_given_activation("linear")

# Activation:  relu
# Before Training: [('loss', 2.6973885402679443), ('acc', 0.48899999999999999)]
# After Training: [('loss', 7.7373054656982418), ('acc', 0.505)]
# Activation:  tanh
# Before Training: [('loss', 5.0640698051452633), ('acc', 0.41699999999999998)]
# After Training: [('loss', 7.6523446731567386), ('acc', 0.52000000000000002)]
# Activation:  sigmoid
# Before Training: [('loss', 0.70816111516952518), ('acc', 0.52500000000000002)]
# After Training: [('loss', 0.67464308834075926), ('acc', 0.58199999999999996)]
# Activation:  hard_sigmoid
# Before Training: [('loss', 0.70220352411270137), ('acc', 0.52100000000000002)]
# After Training: [('loss', 0.67294596910476689), ('acc', 0.58099999999999996)]
# Activation:  linear
# Before Training: [('loss', 3.5439299507141113), ('acc', 0.47799999999999998)]
# After Training: [('loss', 8.2581552581787108), ('acc', 0.0)]
```

Keras implements a number of activation functions, namely, tanh, sigmoid, hard_sigmoid, linear, and relu (rectified linear unit). Activation functions and their appropriateness given a task (classification, multiclassification, regression, etc.) are covered in much detail in Chapter 3. For the context of this chapter, it suffices to say that Keras makes it easy for users to experiment with these activation functions with very little coding effort.

Let us now look at the constructs Keras provides to build the Convolution Neural Networks introduced in Chapter 5. The data set we will be using is the MNIST data set, which is a commonly used benchmark data set for deep learning. The data set consists of handwritten digits (60,000 training examples and 10,000 test examples). The task at hand is to predict the digit given the image, so this is a multiclassifcation problem with ten classes.

Listing 7-7. CNN

```
import numpy as np
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Convolution2D, MaxPooling2D
```

```
from keras.utils import np_utils
from keras.utils.visualize_util import plot

# Image Size
img_rows, img_cols = 28, 28

# Filter
nb_filters = 32

# Pooling
pool_size = (2, 2)

# Kernel
kernel_size = (3, 3)

# Prepare dataset
(X_train, y_train), (X_test, y_test) = mnist.load_data()
X_train = X_train.reshape(X_train.shape[0], img_rows, img_cols, 1)
X_test = X_test.reshape(X_test.shape[0], img_rows, img_cols, 1)
input_shape = (img_rows, img_cols, 1)
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train /= 255
X_test /= 255
nb_classes = 10
Y_train = np_utils.to_categorical(y_train, nb_classes)
Y_test = np_utils.to_categorical(y_test, nb_classes)

# CNN
model = Sequential()
model.add(Convolution2D(nb_filters, kernel_size[0], kernel_size[1], border_mode='valid',
input_shape=input_shape))
model.add(Activation('relu'))
model.add(Convolution2D(nb_filters, kernel_size[0], kernel_size[1]))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=pool_size))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(nb_classes))
model.add(Activation('softmax'))

# Compilation
model.compile(loss='categorical_crossentropy', optimizer='adadelta', metrics=['accuracy'])

# Training
batch_size = 128
nb_epoch = 1
model.fit(X_train, Y_train, batch_size=batch_size, nb_epoch=nb_epoch, verbose=1, validation_
data=(X_test, Y_test))
```

```
# Evaluation
score = model.evaluate(X_test, Y_test, verbose=0)
print "Test Metrics:", zip(model.metrics_names, score)
plot(model, to_file='s7.png', show_shapes=True)

# Output
# Train on 60000 samples, validate on 10000 samples
# Epoch 1/1
# 60000/60000 [==============================] - 128s - loss: 0.3964 - acc: 0.8776 - val_
loss: 0.0929 - val_acc: 0.9712
# Test Metrics: [('loss', 0.092853568810969594), ('acc', 0.97119999999999995)]
```

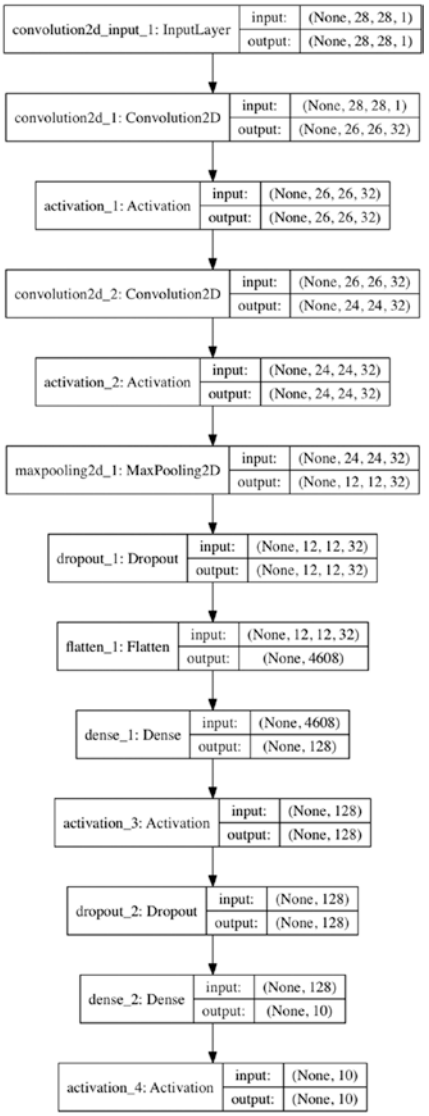


Figure 7-5. *CNN*

Listing 7-7 presents the source code for the convolution neural network and Figure 7-5 illustrates the computation graph. The following points are to be noted:

1. The overall network consists of two convolution-detector blocks, followed by a max pooling layer which in turn is followed by a two layer fully connected network (refer to Chapter 5).
2. The size of the kernel is 3×3.
3. The pooling operation is done over sections of dimensionality 2×2.
4. There are a number of dropout layers, which are basically a form of regularization (refer to Chapter 1) and operate by randomly turning off a certain number of units. The parameter 0.25 indicates the faction of inputs that will be randomly dropped.
5. The flatten layers convert the input of any dimensionality to a dimensionality of $1 \times n$. So, for instance, an input of dimensionality $2 \times 2 \times 3$ gets converted to a directionality of 1×12.
6. The output layer is softmax and the loss function is categorical entropy, as is appropriate for a multiclassification problem (refer to Chapter 3).
7. The model is fit using adadelta (refer to Chapter 8) and, for the purposes of illustration, we set the epochs to 1 (ideally it's set to much more than that).

Listing 7-8. LSTM

```
import numpy as np
from keras.preprocessing import sequence
from keras.utils import np_utils
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Embedding
from keras.layers import LSTM
from keras.datasets import imdb
from keras.utils.visualize_util import plot

max_features = 20000
maxlen = 80
batch_size = 32

# Prepare dataset
(X_train, y_train), (X_test, y_test) = imdb.load_data(nb_words=max_features)
X_train = sequence.pad_sequences(X_train, maxlen=maxlen)
X_test = sequence.pad_sequences(X_test, maxlen=maxlen)

# LSTM
model = Sequential()
model.add(Embedding(max_features, 128, dropout=0.2))
model.add(LSTM(128, dropout_W=0.2, dropout_U=0.2))
model.add(Dense(1))
model.add(Activation('sigmoid'))

# Compile
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

```
# Training
model.fit(X_train, y_train, batch_size=batch_size, verbose=1, nb_epoch=1, validation_
data=(X_test, y_test))

# Evalaution
score = model.evaluate(X_test, y_test, batch_size=batch_size)
print "Test Metrics:", zip(model.metrics_names, score)
plot(model, to_file='s8.png', show_shapes=True)

# Output
# Train on 25000 samples, validate on 25000 samples
# Epoch 1/1
# 25000/25000 [==============================] - 165s - loss: 0.5286 - acc: 0.7347 - val_
loss: 0.4391 - val_acc: 0.8076
# 25000/25000 [==============================] - 33s
# Test Metrics: [('loss', 0.43908300422668456), ('acc', 0.80759999999999998)]
```

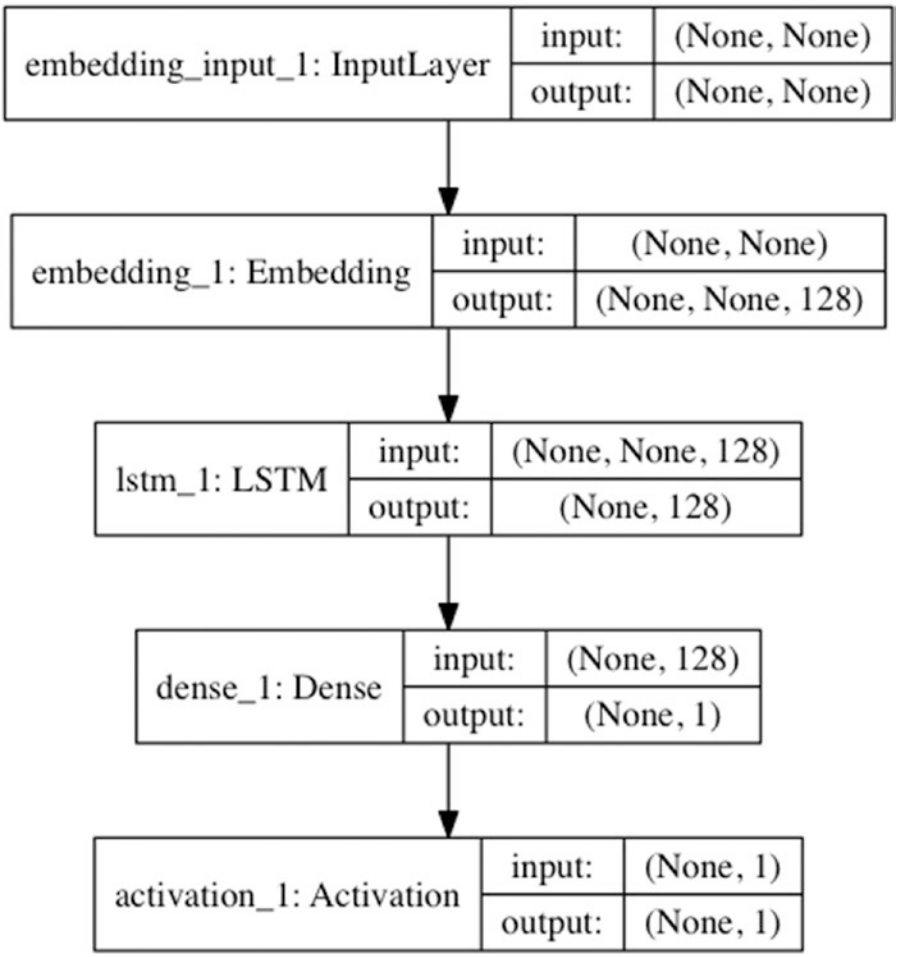


Figure 7-6. *LSTM*

Let us now look at the construct Keras provides to build LSTM Networks, introduced in Chapter 6 (refer to Listing 7-8 and Figure 7-6). The data set we will be using is from IMDB and represents 25,000 reviews from IMDD categorized as positive or negative, making this a binary sequence classification problem. The data is preprocessed to contain only frequent words (the words are actually represented as integers). Listing 7-8 presents the source code and Figure 7-6 illustrates the computational graph. Keras provides a fairly high-level construct for LSTMs, which allows users to construct LSTM models.

Summary

In this chapter we covered the basics of using Keras, using a number of small and simple examples. We encourage the reader to experiment with the examples. Keras has extensive documentation, which is recommended further reading.

CHAPTER 8

Stochastic Gradient Descent

An essential step in building a deep learning model is solving the underlying optimization problem, as defined by the loss function. This chapter covers Stochastic Gradient Descent (SGD), which is the most commonly used algorithm for solving such optimization problems. We also cover a breadth of algorithmic variations published in academic literature which improve on the performance of SGD, and present a bag of largely undocumented tricks which will allow the user to go an extra mile. Lastly, we cover some ground on parallel/distributed SGD and touch on Second Order Methods for completeness.

Most of the examples presented in the accompanying code for this chapter are based on a Python package called downhill. Downhill implements SGD with many of its variations and is an excellent choice for experimenting.

Optimization Problems

Simply put, an optimization problem involves finding the parameters which minimize a mathematical function.

For example, given the function $f(x) = 2x$, finding the value of x which minimizes the function is an optimization problem (refer to Figure 8-1).

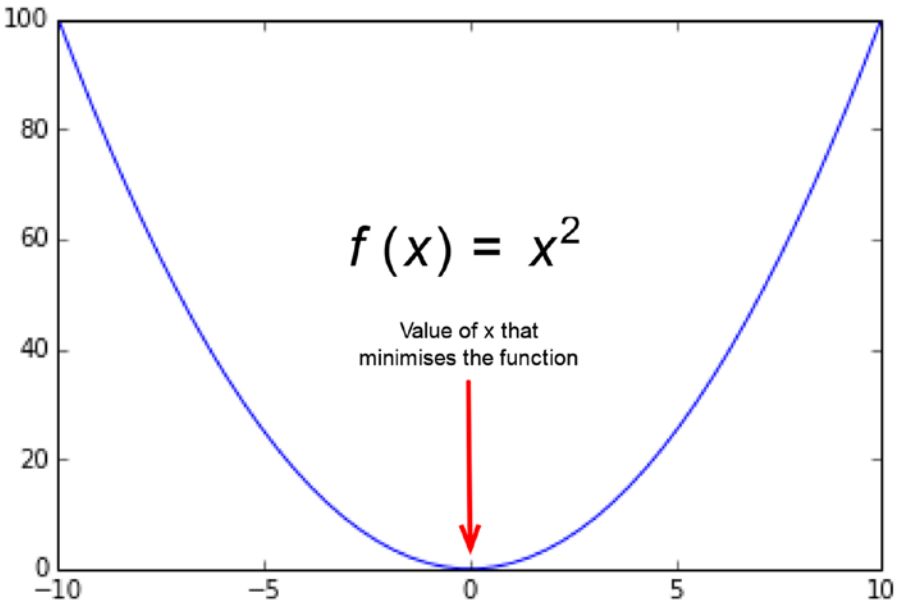


Figure 8-1. *An optimization problem involves finding the parameters that minimize a given function*

N. Ketkar, *Deep Learning with Python*, DOI 10.1007/978-1-4842-2766-4_8

While the functions we want to minimize while building deep learning models are way more complicated (involving multiple parameters which may be scalars, vectors, or matrices), conceptually it's simply finding the parameters that minimize the function.

The function one wants to optimize for while building a deep learning model is referred to as the loss function. The loss function may have a number of scalar/vector/matrix-valued parameters but always has a scalar output. This scalar output represents the goodness of the model. Goodness, typically, means a combination of how well the model predicts and how simple the model is.

■ **Note** For now, we will stay away from the statistical/machine learning aspects of a loss function (covered elsewhere in the book) and focus purely on solving such optimization problems. That is, we assume that we have been presented with a loss function $L(x)$ where x represents the parameters of the model and the job at hand is to find the values for x which minimize $L(x)$.

Method of Steepest Descent

Let us now look at a simple mathematical idea, which is the intuition behind SGD. For the sake of simplicity, let us assume that x is just one vector. Given that we want to minimize $L(x)$, we want to change or update x such that $L(x)$ reduces. Let u represent the unit vector or direction in which x should be ideally changed and let α denote the magnitude (scalar) of this step. A higher value of α implies a larger step in the direction u, which is not desired. This is because u is evaluated for the current value of x and it will be different for a different x.

Thus, we want to find a u such that

$$\lim_{\alpha \to 0} L(x + \alpha \mathrm{u})$$

is minimized. It follows that

$$\lim_{\alpha \to 0} L(\mathrm{x} + \alpha \mathrm{u}) = u^T \nabla_{\mathrm{x}} L(x).$$

Thus, we basically want to find a u such that

$$u^T \nabla_{\mathrm{x}} L(x)$$

is minimized. Note that $\nabla_{\mathrm{x}} L(x)$ is the gradient of $L(x)$.

Given that both u^T and $\nabla_{\mathrm{x}} L(x)$ are vectors, it follows that

$$u^T \nabla_{\mathrm{x}} L(x) = |u| \cdot |\nabla_{\mathrm{x}} L(x)| \cos\theta,$$

where θ is the angle between the two vectors (refer to Figure 8-2).

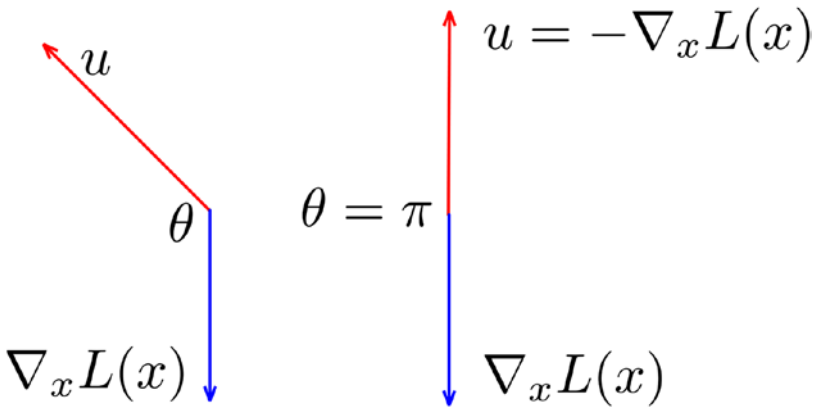


Figure 8-2. *Finding the desired direction of update*

The value of cos θ would be minimized at $\theta = \pi$, that is to say the vectors are pointing in the opposite direction. Thus, it follows that setting the direction $u = -\nabla_x L(x)$ would achieve our desired objective. This leads to a simple iterative algorithm as follows:

Input: α, n

Initialize x to a random value.
For n steps do:

Update x = $x - \alpha \nabla_x L(x)$

Output: x

Batch, Stochastic (Single and Mini-batch) Descent

So far, we were denoting our loss function as $L(x)$. The training data (examples) were implicit in this notation. For the purpose of this discussion, we need to make them explicit. Let us denote our training data as $D = \{d_1, d_2, \ldots d_n\}$. Our loss function should now be denoted as $L_D(x)$. This simply means that the loss function is being evaluated with parameters x and with respect to a set of data points D. Let T be a subset of examples in D; then $L_T(x)$ denotes the loss function evaluated over the set of examples T. Similarly, $\nabla_x L_D(x)$ and $\nabla_x L_T(x)$ denote the gradients of the loss function $L(x)$ computed over the sets of training data D and T, respectively.

■ **Note** For now, we will stay away from the computation of gradients of loss functions over subsets of data. These can be generated using automatic differentiation (covered elsewhere in the book) quite easily (even for arbitrary complicated loss functions) and need not be derived manually.

Armed with this notation, we can now discuss three variants of the steepest descent approach we discussed earlier.

Batch

In this approach, the entire dataset D is used in the update step. That is, x is updated as $x = x - \nabla_x L_D(x)$. Two things are apparent about this approach. First, that it is expensive as it requires a computation over the entire dataset. Second, the update direction is the most accurate one, given our entire dataset.

Stochastic Single Example

In this approach, only a single example from D is used in the update step. That is, x is updated as $x = x - \nabla_x L_S(x)$ where $|S| = 1$. Note that we use a different example in each iteration chosen randomly (hence the term stochastic, which simply means having a random nature). Two things are apparent about this approach, too. First, that it is quite cheap as it requires a computation of the gradient over only a single example. Second, the update direction is not as accurate as we are only using a very small fraction of the training dataset.

Stochastic Mini-batch

In this approach, only a small subset of examples from D is used in the update step. That is, x is updated as $x = x - \nabla_x L_S(x)$ where $|S| < |D|$. Note that we use a different set of examples in each iteration chosen randomly (hence the term stochastic).

Batch vs. Stochastic

In practice, stochastic approaches dominate over batch approaches and are much more commonly used. A seemingly nonintuitive fact about stochastic approaches is that not only is the gradient over few examples cheaper to compute, not getting the exact direction (using only a small number of examples) actually leads to better solutions. This is particularly true for large datasets with redundant information wherein the examples are not too different from each other. Another reason for stochastic approaches performing better is the presence of multiple local minima with different depths. In a sense, the noise in the direction allows for jumping across the trenches while a batch approach will converge in the trench it started with.

Challenges with SGD

Having covered the conceptual description of SGD, let us now consider the challenges of applying it to solving real world problems while build deep learning models.

Local Minima

Local minima are suboptimal solutions (refer to Figure 8-3) that trap any steepest descent approach and prevent the iterative procedure for making progress towards a better solution.

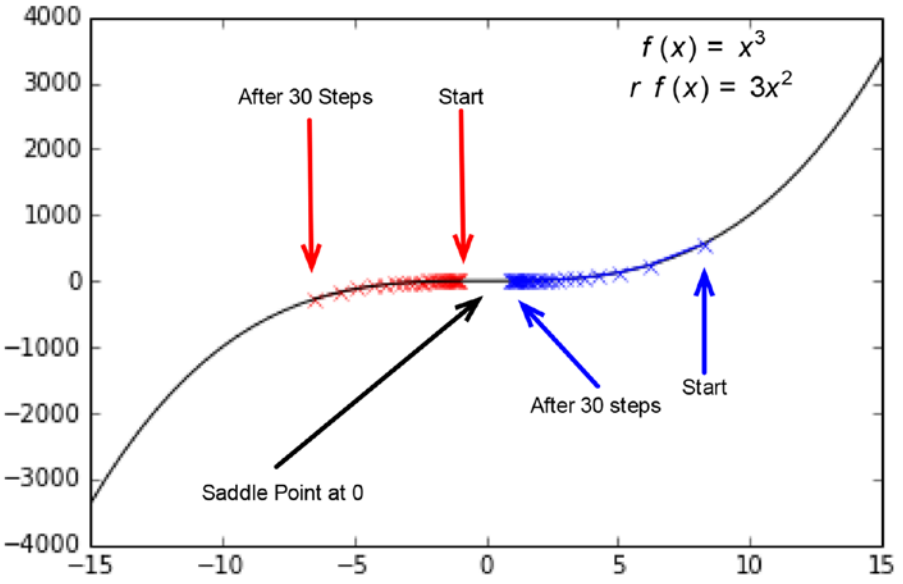


Figure 8-3. *Local Minima*

Saddle Points

Saddle points are points (refer to Figure 8-4) where the gradient evaluates to zero but the point is not a local minimum. Saddle points are any steepest descent approach and prevent the iterative procedure for making progress towards a better solution. There is good empirical evidence that saddle points are much more common than local minima when it comes to optimization problems in a high number of dimensions (which is always the case when it comes to building deep learning models).

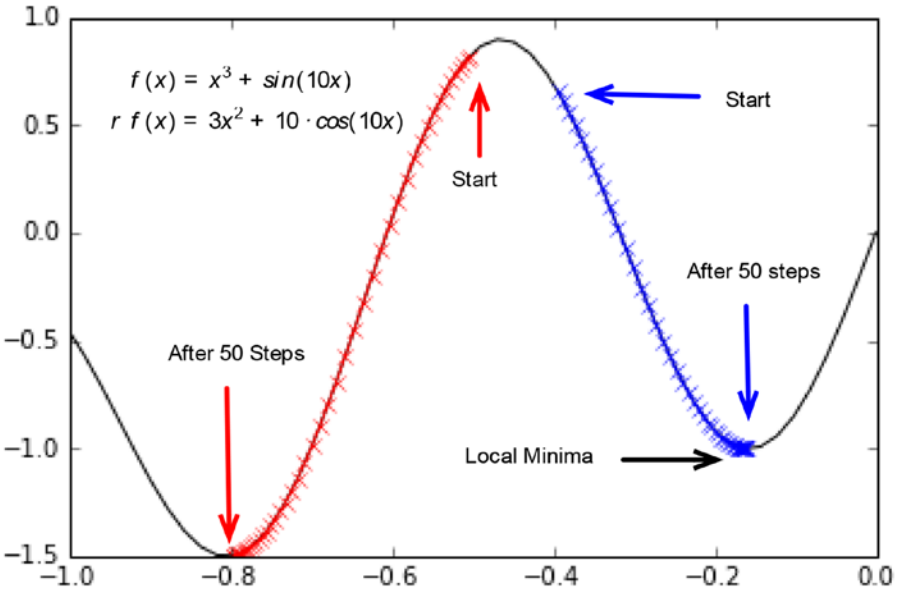


Figure 8-4. *Saddle point*

Selecting the Learning Rate

The α is the earlier discussion which represents the magnitude (scalar) of the step (in the direction u) is taken in each iteration so as to update *x*. This α is commonly referred to as the learning rate and it has a big impact on finding good solutions to the optimization problem (refer to Figure 8-5). Too high a learning rate can cause the solution to bounce around and too low a learning rate means slow convergence (implying not getting to a good solution in a given number of iterations). When it comes to loss functions with many parameters trained on sparse datasets, a single global learning rate for all parameters makes the problem of choosing a learning rate even more challenging.

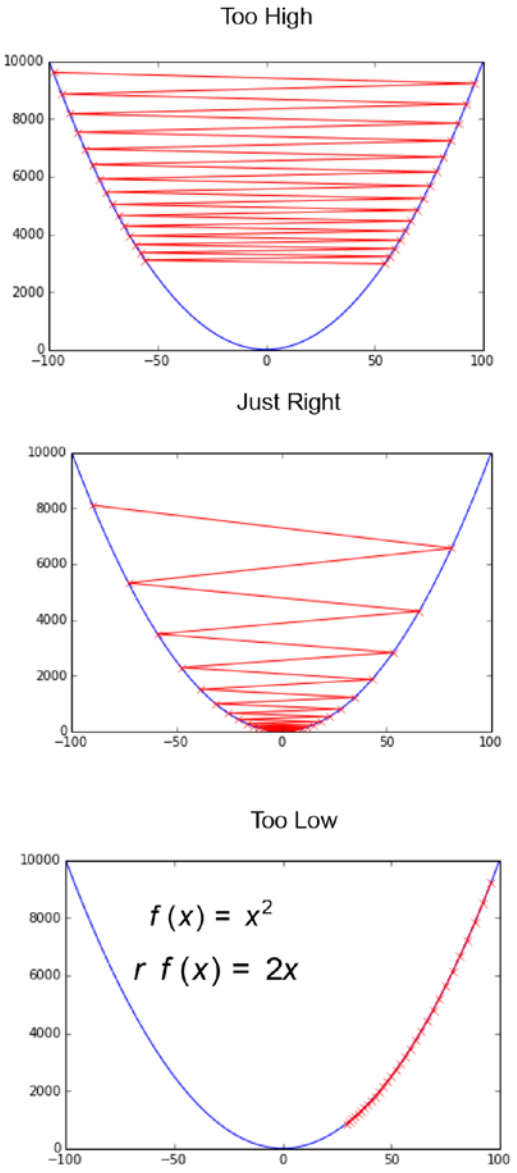


Figure 8-5. *Learning rate needs to be set properly*

Slow Progress in Narrow Valleys

Another problem inherent to steepest descent is the slow progress in narrow valleys generated due to badly scaled datasets. Progress slows down drastically as we get closer to the solution (refer to Figure 8-6).

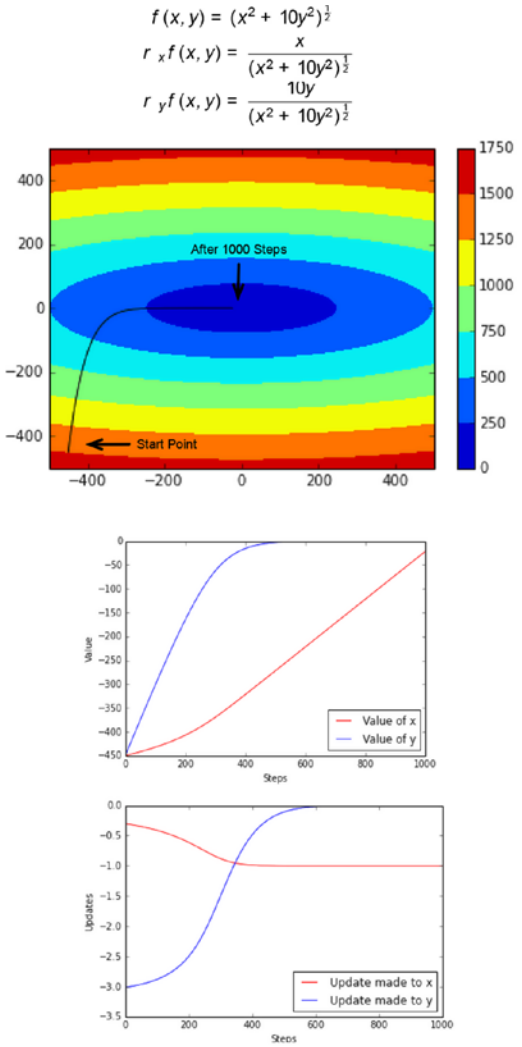


Figure 8-6. *Slow progress in narrow valleys*

Algorithmic Variations on SGD

We will now cover a number of algorithmic variations for SGD proposed in academic literature that address the challenges discussed earlier.

Momentum

Consider the update step for SGD described earlier,

$$x = \mathrm{x} - \alpha \nabla_{\mathrm{x}} L(x).$$

The intuition behind momentum is to use a fraction of the previous update for the current update. That is, let u_s denote the update to the parameters x in step s. Similarly, let u_{s-1} denote the update in the previous step. Now, let us update x with

$$\mathrm{u}_s = \gamma u_{s-1} + \alpha \nabla_{\mathrm{x}} L(x).$$

That is, we update

$$\mathrm{x} = \mathrm{x} - \mathrm{u}_s$$

instead of

$$\mathrm{x} = \mathrm{x} - \alpha \nabla_{\mathrm{x}} L(x).$$

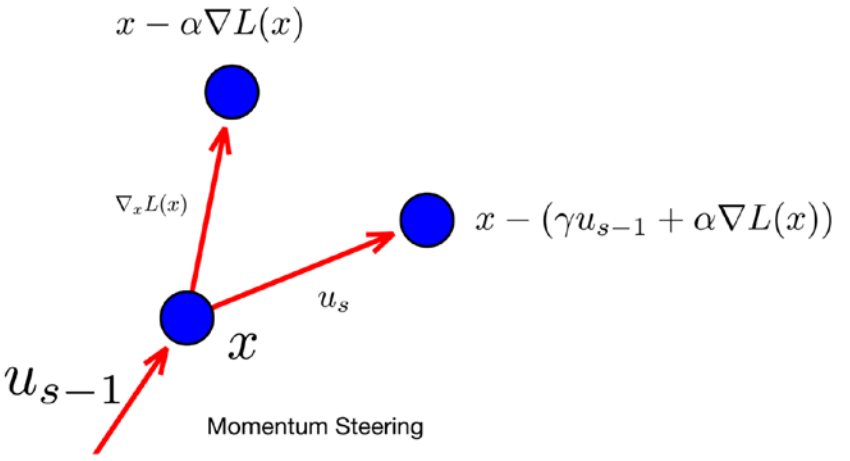


Figure 8-7. *Momentum steering*

Simply put, we have used a faction of the update in the previous step for the current update. This idea is referred to as momentum, as it is akin to the momentum acquired by a ball rolling downhill. A ball that has picked up momentum will bounce out of small ditches (local minima) along the way and reach the bottom of the hill. It will also keep up somewhat with the speed of previous downhill movement even if the hill has a much reduced slope (because it has picked up momentum). The momentum term basically causes new step direction to be biased by the step previous direction (refer to Figure 8-7). Use of momentum has been empirically shown to cause reduced oscillation and faster convergence.

Nesterov Accelerated Gradient (NAS)

Using the same notation as before, the Nesterov accelerated gradient is basically updating x with

$$u_s = \gamma u_{s-1} + \alpha \nabla_x L(x - \gamma u_{s-1}).$$

That is, we update $x = x - u_s$.

The intuition behind this is looking one step ahead. That is, we first take a step in the direction of the accumulated gradient and do an adaptation. The intuition behind NAS is looking ahead and anticipating, which leads to better solutions (refer to Figure 8-8).

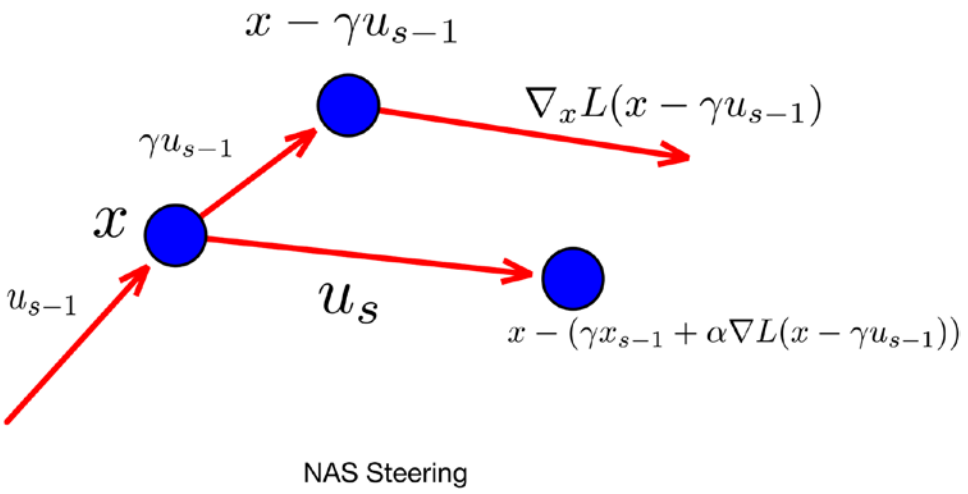


Figure 8-8. *NAS steering*

Annealing and Learning Rate Schedules

When gradient descent approaches a minimum, we have seen how a bad learning rate can cause it to oscillate around the minima (see Figure 8-5). Annealing refers to reducing the learning rate as it approaches the minima. This can be done manually (stopping the gradient descent and restarting from the same point with a reduced learning rate) or via learning rate schedules which introduce a number of user-controlled hyper parameters which dictate how the learning rate is reduced based on the number of steps taken. However, it must be noted that we are using the same learning rate for all the parameters, which may not be appropriate; a per-parameter learning rate adjustment is desired.

Adagrad

The Adagrad algorithm adjusts the learning rate for each parameter. So far, we have been denoting the parameters of the loss function as x. Note that x is actually a large number of parameters, each of which is being updated with the same learning rate. Let x_i denote one of the parameters and let g_i^s denote the gradient for x_i at step s. For steps $0,1,\ldots,S-1$ we have a corresponding series of gradients $g_i^0, g_i^1,\ldots,g_i^s$.

Let

$$G=\left(g_i^0\right)^2+\left(g_i^1\right)^2+\ldots+\left(g_i^{S-1}\right)^2,$$

which is basically the sum of squares of the gradients for each step up to the previous step. The update rule in Adagrad is

$$x_i=x_i-\frac{\alpha}{G^{\frac{1}{2}}}g_i^S.$$

The α term is the global learning rate, which gets adapted for each parameter based on the previous gradients. It must also be noted that as G accumulates, the learning rate slows down for each parameter and eventually no progress can be made, which is a weakness of Adagrad.

RMSProp

The RMSProp algorithm improves on the Adagarad algorithm's weakness of completely halted progress beyond a certain number of iterations. The intuition here is to use a window of fixed size over the gradients computed at each step rather than use the full set of gradients. That is, compute G over only the past W steps. Now, it's conceptually equivalent but computationally cheaper to treat the computation of

$$G=\frac{\left(g_i^{S-W}\right)^2+\left(g_i^{S-w+1}\right)^2+\ldots+\left(g_i^{S-1}\right)^2}{W}$$

as the accumulation of exponentially decaying average of square of gradients rather than store all values of

$$g_i^{-w},g_i^{-w+1},\ldots,g_i^{-2}g_i^{-1}$$

and compute G at each step. That is, we compute

$$E\left[\left(g_i\right)^2\right]^S=\rho E\left[\left(g_i\right)^2\right]^{S-1}+\left(1-\rho\right)\left(g_i^S\right)^2$$

where ρ is the decay.

Now, note that in Adagrad we were computing the update as $x_i=x_i-\frac{\alpha}{G^{\frac{1}{2}}}g_i^S$. Consider the value of $G^{\frac{1}{2}}$. We can see that this is simply the root-mean-square of g_i, that is

$$RMS\left[g_i\right]=G^{\frac{1}{2}}=\sqrt{E\left[\left(g_i\right)^2\right]^S}.$$

Thus, we can compute the update as

$$x_i=x_i-\frac{\alpha}{RMS\left[g_i\right]}g_i^S.$$

Adadelta

The intuition behind Adadelta is to consider whether the unit of the parameter and the update to the parameter is the same. The author of the Adadelta argues that this is not the same in the case of any first order methods like steepest descent (but is the same in the case of second order methods like Newton's method). In order to fix this issue, the proposed update rule of Adadelta is

$$x_i = x_i - \frac{RMS\left[\Delta x_i\right]^{S-1}}{RMS\left[g_i\right]^{S}} g_i^{S}$$

where $RMS\left[\Delta x\right]^{S-1}$ is the root-mean-square of the actual updates to x. Note that $RMS\left[\Delta x_i\right]^{S-1}$ lags behind $RMS[g_i]^S$ by one step.

Adam

Adam computes the updates by maintaining the exponentially weighted averages of both g_i and $(g_i)^2$ for each parameter (denoted by the subscript i). The update rule for Adam is

$$x_i = x_i - \frac{\alpha}{E\left[\left(g_i^{S-1}\right)\right]^2} E\left[g_i^{S-1}\right].$$

It is important to note that $E\left[g_i^{S-1}\right]$ and $E\left[\left(g_i^{S-1}\right)^2\right]$ are biased towards zero in the initial steps for small decay rates (there are two decay rates here—one for $E\left[g_i^{S-1}\right]$ and one for $E\left[\left(g_i^{S-1}\right)^2\right]$—which we denote by ρ_1 and ρ_2 respectively). This bias can be corrected by computing

$$E\left[g_i^{S-1}\right] = \frac{E\left[g_i^{S-1}\right]}{1-\rho_1}$$

and

$$E\left[\left(g_i^{S-1}\right)^2\right] = \frac{E\left[\left(g_i^{S-1}\right)^2\right]}{1-\rho_2}$$

respectively.

Resilient Backpropagation

The intuition behind Resilient Backpropagation is that the sign of the gradient switches back and forth between positive and negative when the learning rate is too high (refer to Figure 7). The key idea is to keep track of the sign of the previous gradient and match it with the current gradient. If the sign is the same, use a higher learning rate and, if different, use a lower learning rate. Note that this is done for every parameter. The hyper parameters include the amounts to increase and decrease the learning rate (in case the sign matches or does not match, respectively).

Equilibrated SGD

Equilibrated SGD aims to address issues SGD experiences with saddle points. The key idea here is that we need second order information (second derivatives of the loss function) to get out of the trap of a saddle point. The update rule for Equilibrated SGD is given by $x_i = x_i - \frac{\alpha}{\sqrt{D_i^S}} g_i^S$ where $D_i^S = \rho D_i^{S-1} + (1-\rho)(H_d)^2$ (exponentially weighted average) and H_d is the diagonal of the Hessian matrix of $L(x)$ (computed symbolically) evaluated at $x \in \mathcal{N}(0,1)$ (normal distribution with 0 mean and standard deviation of 1).

Tricks and Tips for using SGD

We will now cover a number of tricks and tips for SGD proposed in academic literature that address the challenges discussed earlier.

Preprocessing Input Data

It is of utmost importance that data is scaled well so as to ease the optimization (refer to Figure 8-6). A good rule of thumb is to standardize the data by subtracting the mean and divide by the standard deviation to scale the data. So, if $X = \{X_1, X_2 \ldots X_n\}$ is one of the input variables, we transform the data so that

$$X_i = \frac{X_i - \mu}{\sigma}.$$

In case of sparse data (most of X_i are equal to zero), the standardization process will cause the data to become dense, max-abs scaling where

$$X_i = \frac{|X_i|}{\max(X)}.$$

Scaling the feature to have a unit norm

$$X_i = \frac{X_i}{|X_{i2}|}$$

is another approach to scaling data.

It is also a good practice to remove linear correlations amongst input variables from input data using Principal Component Analysis.

Choice of Activation Function

Common examples of activation function are the standard logistic function $f(x) = \frac{1}{1+e^{-x}}$ and the hyperbolic tangent function $f(x) = \tanh(x)$. A recommended approach is to use an activation function that is symmetrical around 0 (rather than only positive or negative), for instance, $f(x) = 1.7159 \tanh\left(\frac{2}{3}x\right)$. It is also recommended that a small linear term be added to prevent flat sports, $f(x) = \tanh(x) + ax$.

Preprocessing Target Value

While target values can be binary (0,1) in many cases, it is advisable to transform the target variable to values that lie within the range (not asymptotically, but practically) of the activation functions used to define the loss function. Not doing so leads to parameters being updated to higher and higher values without achieving any effect (on the output label). However, if the value of the target label is being used as a measure of confidence, then the labels which have been unnecessarily pushed to higher values are bad estimates of the confidence. While 0 and 1 may be extreme values for the activation function, it is also important not to choose targets that lie in the linear region of the activation function. A recommended approach is to choose values that maximize the second derivative of the activation function, for instance, ±1 for $f(x)=1.7159\tanh\left(\frac{2}{3}x\right)$.

Initializing Parameters

It is a recommended practice to initialize parameters randomly (normal distribution, zero-mean, unit variance). Another recipe for neural networks where the activation function is $f(x)=1.7159\tanh\left(\frac{2}{3}x\right)$ (and the data is standardized) is to set weights to $m^{-\frac{1}{2}}$ where m is the fan-in (the number of connections feeding into the node).

Shuffling Data

It is recommended practice to shuffle input data, as it may be in a particular order and hence might bias the SGD. A rare exception to this is Curriculum Learning wherein examples are presented in a meaningful order (increasing difficulty of prediction).

Batch Normalization

While our parameters are initialized in a way such that they are normalized (set randomly, normal distribution, zero-mean, and unit variance), they do not remain normalized over the update steps. Batch normalization renormalizes parameters after each batch (refer to batch SGD).

Early Stopping

Early stopping basically involves measuring the loss on an unseen (unused for SGD) subset of training data (called validation data) and stopping when there is no change observed in the loss. Typically, there are two hyper parameters introduced: one which determines whether the change is significant (any change in loss less than this value is treated as not a change) and a patience parameter, which is the number of times a no change step can be taken before the iterative procedure terminates.

Gradient Noise

The gradient noise trick introduces a mean centered noise, $\mathcal{N}(0,\sigma)$, in every update step. Here, σ is a hyper parameter and the gradient is computed as $g_i = g_i + \mathcal{N}(0,\sigma)$.

Parallel and Distributed SGD

We now cover two parallel and distributed approaches for SGD. SGD in its basic form is a sequential algorithm and convergence can be very slow on large data sets and models with a large number of parameters. Parallel and distributed approaches have a great impact when it comes to dealing with large volumes of training data (in the order of billions) and a large number of model parameters (in the order of millions).

Hogwild

Consider the update step of the SGD procedure. What makes the algorithm inherently sequential is update step $x = x - \alpha \nabla_x L(x)$. Let's say that we want to employ multiple threads of computation to make the iteration faster. Since we want only one thread to do this (do the update one step at a time), we would place a lock around this step (to prevent a race condition). Once we do that, this essentially becomes a sequential algorithm; no matter how many cores and threads we devote to the process, only one thread is actually doing the work, while all others are waiting on a lock.

The intuition behind Hogwild is that the race condition caused by not placing a lock on the update step does not lead to much inconsistency in updates when the optimization problem is sparse. This is simply because each update step touches only a few parameters. The authors of Hogwild provide strong theoretical and empirical evidence for this finding and the gains on large datasets are significant. Hogwild is easy to implement on multi-core CPUs and GPUs.

Downpour

Downpour is a distributed algorithm for SGD that consists of two key moving parts: model replica and parameter server (refer to Figure 8-9). A model replica is a set of machines that operates on a subset of data, where every machine operates only on a subset of parameters. There are many such model replicas, each operating on a different subset of a large dataset. The parameter server is a set of machines that maintains a common global state of the model. Model replicas retrieve the global state from the parameter server, update the model based on the subset of data and update the global state. Note that the fetch and update of the global state does not happen at every iteration. There are two levels of distribution with Downpour. First, the model parameters (what we have been denoting as x so far) are split across multiple machines in each model replica. Second, the data is split amongst model replicas. So, essentially, each machine is doing the gradient update step on a subset of model parameters, using a subset of data. The global state is updated asynchronously. In spite of the apparent inconsistencies introduced by Downpour, it has been found to be very effective when it comes to training a large model with large amounts of data.

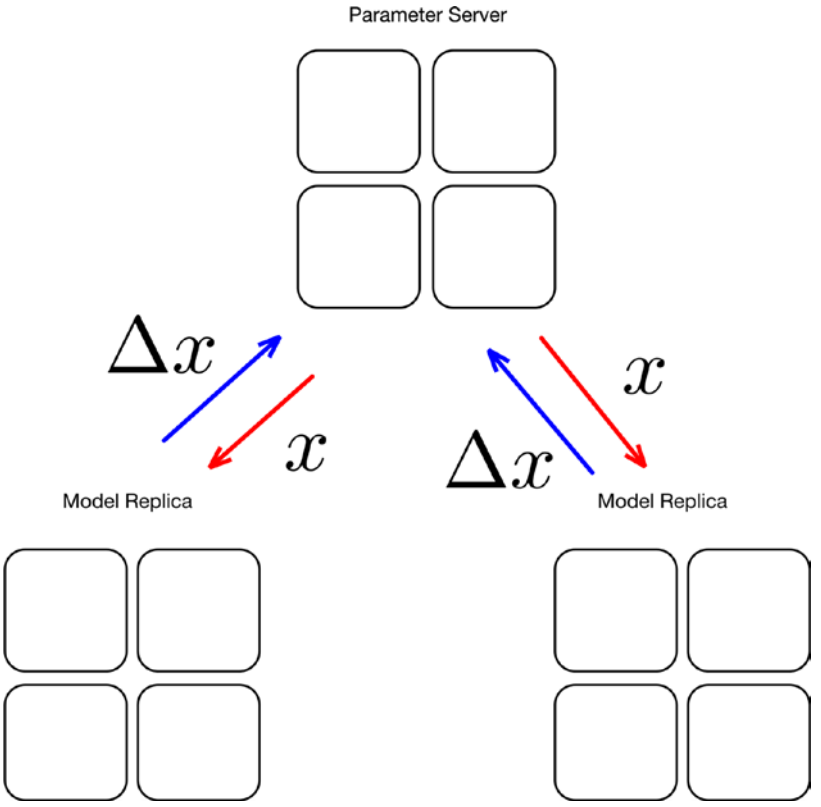


Figure 8-9. *Downpour*

Hands-on SGD with Downhill

We will now a hands-on exercise with SGD using a Python package called Downhill. Downhill implements SGD with many of its variants. It operates on loss functions defined in Theano, which makes it a very convenient tool to play with SGD variants on arbitrary loss functions defined in Theano. Let us start with generating a dataset for our exercise (Listing 8-1, Figure 8-10).

Listing 8-1. Generating data for our excercise

```
#Specifiy the number of examples we need (5000) and the noise level
train_X, train_y = sklearn.datasets.make_moons(5000, noise=0.1)

#One hot encode the target values
train_y_onehot = numpy.eye(2)[train_y]

#Plot the data
pylab.scatter(train_X[:-1000, 0], train_X[:-1000, 1], c=train_y[:-1000], cmap=pylab.
cm.Spectral)
```

This should produce a plot of the dataset we generated. The objective at hand is to train a model to distinguish between the red and blue dots.

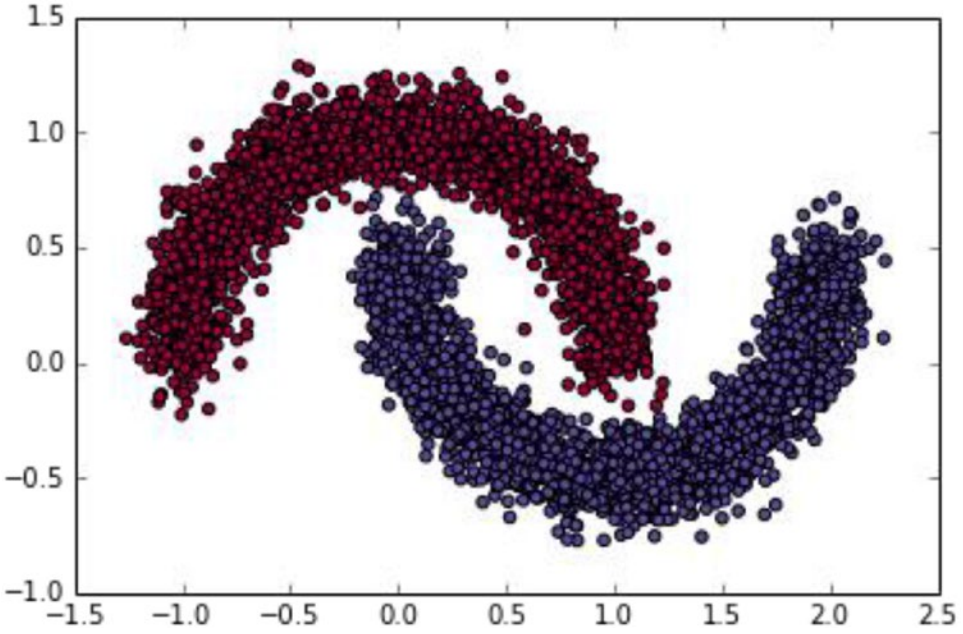


Figure 8-10. *Dataset for our experiments*

Next let's define a loss function with Theano.

Listing 8-2. Defining the loss function

```
#Set Seed
numpy.random.seed(0)

num_examples = len(train_X)

#Our Neural Network
nn_input_dim = 2
nn_hdim = 1000
nn_output_dim = 2

#Regularization
reg_lambda = numpy.float64(0.01)

#Weights and bias terms
W1_val = numpy.random.randn(nn_input_dim, nn_hdim)
b1_val = numpy.zeros(nn_hdim)
W2_val = numpy.random.randn(nn_hdim, nn_output_dim)
b2_val = numpy.zeros(nn_output_dim)

X = T.matrix('X')
```

```
y = T.matrix('y')
W1 = theano.shared(W1_val, name='W1')
b1 = theano.shared(b1_val, name='b1')
W2 = theano.shared(W2_val, name='W2')
b2 = theano.shared(b2_val, name='b2')

batch_size = 1

#Our Loss function
z1 = X.dot(W1) + b1
a1 = T.tanh(z1)
z2 = a1.dot(W2) + b2
y_hat = T.nnet.softmax(z2)
loss_reg = 1./batch_size * reg_lambda/2 * (T.sum(T.sqr(W1)) + T.sum(T.sqr(W2)))
loss = T.nnet.categorical_crossentropy(y_hat, y).mean() + loss_reg

prediction = T.argmax(y_hat, axis=1)
predict = theano.function([X], prediction)
```

■ **Note** For now, we will stay away from the details of defining loss functions with Theano (covered elsewhere in the book).

Next we set up a simple SGD using Downhill. We use all default parameters and want do 10K iterations. The patience parameter is set to 10K also so that early stopping (described earlier in the chapter) does not kick in.

Listing 8-3. SGD

```
#Store the training and vlidation loss
train_loss = []
validation_loss = []

opt = downhill.build('sgd', loss=loss)

#Set up training and validation dataset splits, use only one example in a batch #and use
only one batch per step/epoc

#Use everything except last 1000 examples for training
train = downhill.Dataset([train_X[:-1000], train_y_onehot[:-1000]], batch_size=batch_size,
iteration_size=1)

#Use last 1000 examples for valudation
valid = downhill.Dataset([train_X[-1000:], train_y_onehot[-1000:]])

#SGD
iterations = 0
for tm, vm in opt.iterate(train, valid, patience=10000):
    iterations += 1

    # Record the training and validation loss
    train_loss.append(tm['loss'])
    validation_loss.append(vm['loss'])
```

```
    if iterations > 10000:
            break
```

We can now visualize the decision boundary over the training (Figure 8-11) and validation (Figure 8-12) sets, and the loss (Figure 8-13).

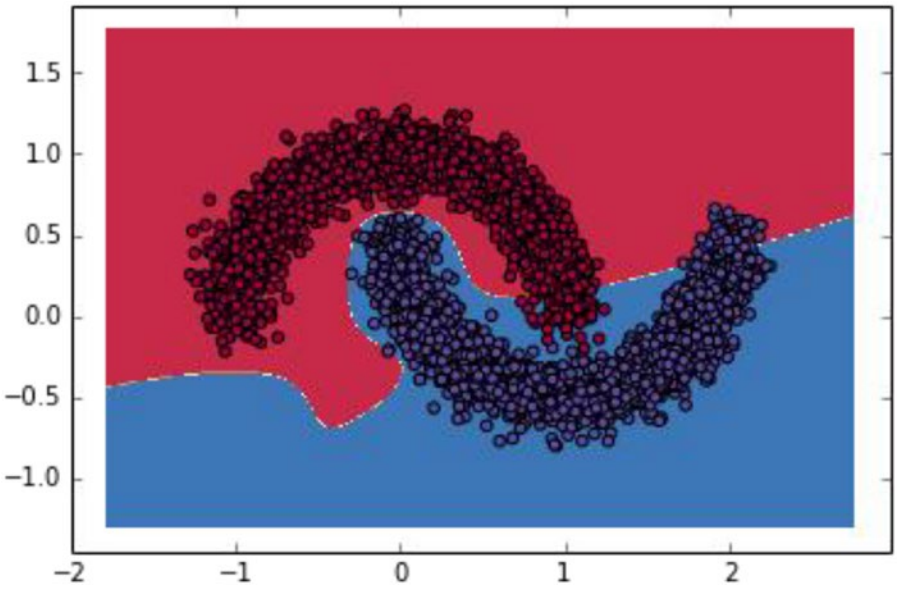


Figure 8-11. *Decision boundary over training set*

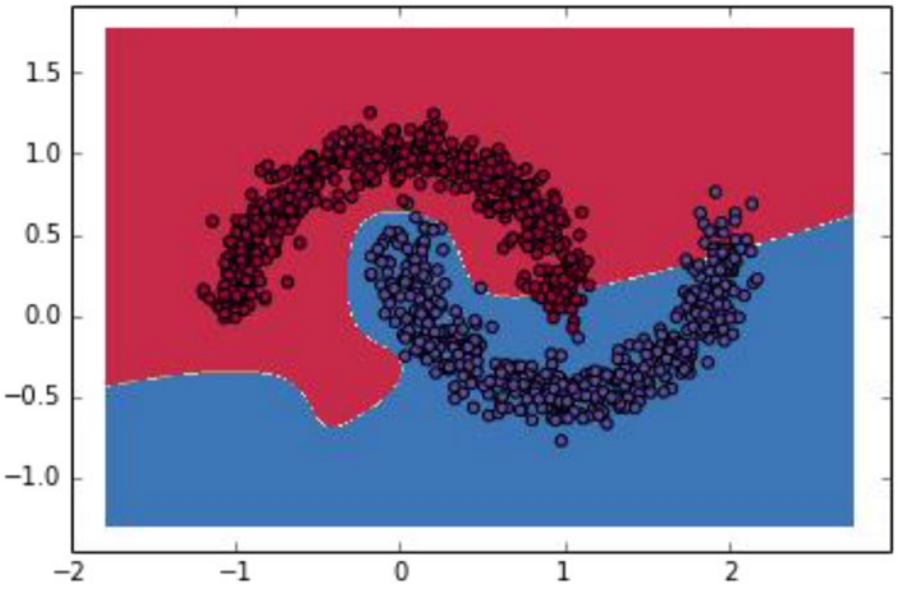


Figure 8-12. *Decision boundary over validaton set*

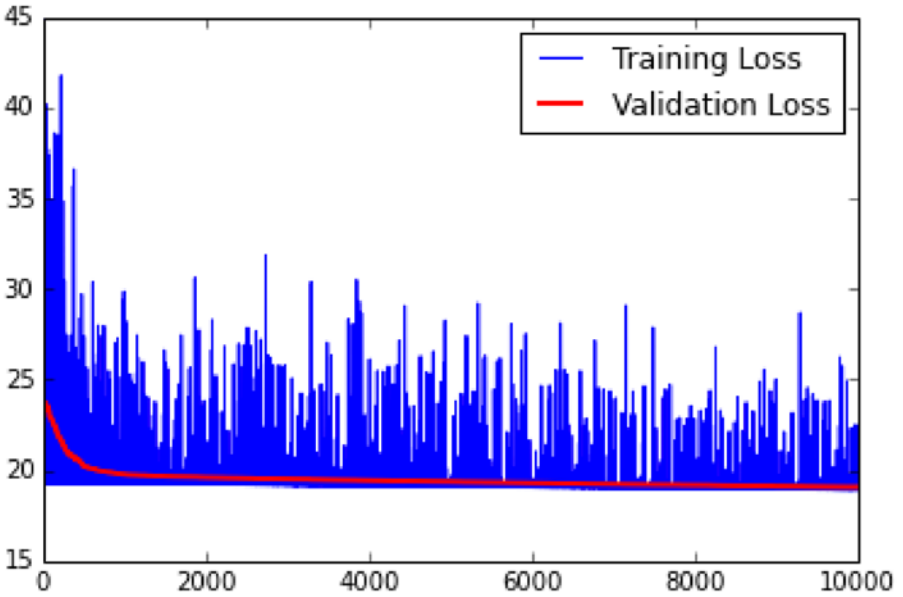


Figure 8-13. *Training and Validation Loss over 10K iterations*

Listing 8-4. Using SGD variants implemented in Downhill

```
def build_model(algo):
    loss_value = []

    W1.set_value(W1_val)
    b1.set_value(b1_val)
    W2.set_value(W2_val)
    b2.set_value(b2_val)

    opt = downhill.build(algo, loss=loss)

    train = downhill.Dataset([train_X[:-1000], train_y_onehot[:-1000]], batch_size=1,
    iteration_size=1)
    valid = downhill.Dataset([train_X[-1000:], train_y_onehot[-1000:]])
    iterations = 0
    for tm, vm in opt.iterate(train, valid, patience=1000):
        iterations += 1
        loss_value.append(vm['loss'])
        if iterations > 1000:
            break
    return loss_value
```

```
algo_names = ['adadelta', 'adagrad', 'adam', 'nag', 'rmsprop', 'rprop', 'sgd']
losses = []
for algo_name in algo_names:
    print algo_name
    vloss = build_model(algo_name)
    losses.append(numpy.array(vloss))
```

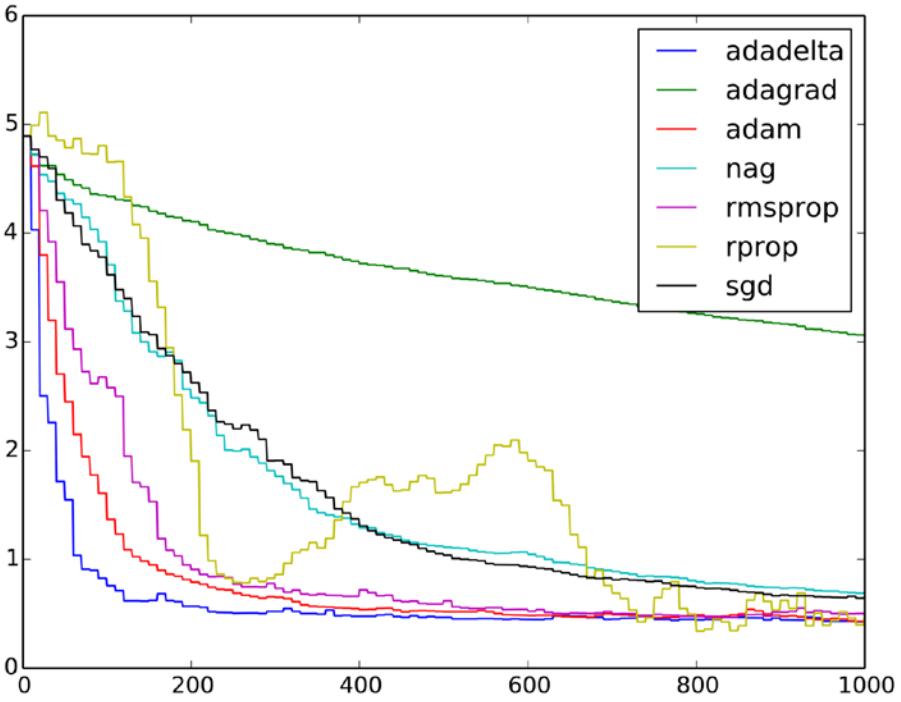


Figure 8-14. *Learning curves (over validation data)*

Let us now try out a number of SGD variants implemented in Downhill (Listing 8-4) and visualize the learning curves (Figure 8-14). We will be running Adadelta, Adagrad, Adam, Nesterov Accelerated Gradient (NAG), RMSProp, Resilient Backpropagation, and vanilla SGD as before. We run all these algorithms with default parameters for 1000 steps.

Summary

In this chapter we covered Stochastic Gradient Descent (SGD), the weaknesses of SGD, a number of algorithmic variations to address these weaknesses, and a number of tricks to make SGD effective. SGD is the most common approach to train deep learning models. The reader is advised to go over the examples in the source code listings and also look at the implementations of SGD and its variants in the Downhill package for further clarity and perspective.

One important aspect that we did not cover in this chapter is how gradients for arbitrary loss functions (required for SGD) are computed. This is covered in the next chapter on automatic differentiation.

CHAPTER 9

Automatic Differentiation

In the chapter on Stochastic Gradient Descent, we treated the computation of gradients of the loss function $\nabla_x L(x)$ as a black box. In this chapter we open this black box and cover the theory and practice of automatic differentiation. Automatic differentiation is a mature technology that allows for the effortless and efficient computation of gradients of arbitrarily complicated loss functions. This is critical when it comes to minimizing loss functions of interest; at the heart of building any deep learning model lies an optimization problem, which is invariably solved using stochastic gradient descent, which in turn requires one to compute gradients.

Most of the applications in this chapter are based on the Python package Autograd, which provides a mature set of capabilities for automatic differentiation.

Automatic differentiation is distinct from both numerical and symbolic differentiation, and we start by covering enough about both of these so that distinction becomes clear. For the purposes of illustration, assume that our function of interest is $f : \mathbb{R} \to \mathbb{R}$ and we intend to find the derivative of f denoted by $f'(x)$.

Numerical Differentiation

Numerical differentiation in its basic form follows from the definition of derivative/gradient. So, given that

$$f'(x) = \frac{df}{dx} = \lim_{\Delta x \to 0} \frac{f(x+\Delta x) - f(x)}{\Delta x}$$

we can compute the $f'(x)$ using the forward difference method as

$$f'(x) = D_+(h) = \frac{f(x+h) - f(x)}{h},$$

setting a suitably small value for h. Similarly, we can compute $f'(x)$ using the backward difference method as

$$f'(x) = D_-(h) = \frac{f(x) - f(x-h)}{h},$$

N. Ketkar, *Deep Learning with Python*, DOI 10.1007/978-1-4842-2766-4_9

again by setting a suitably small value for h. A more symmetric form is the central difference approach, which computes f' as

$$f'(x) = D_0(h) = \frac{f(x+h) - f(x-h)}{2h}.$$

A further development over this idea is Richardson's extrapolation

$$f'(x) = \frac{4D_0(h) - D_0(2h)}{3}.$$

The approximation errors for forward and backward differences are in the order of h, that is, $O(h)$, while those for central difference and Richardson's approximation are $O(h^2)$ and $O(h^4)$ respectively.

The key problems with numerical differentiation are the computational cost, which grows with the number of parameters in the loss function, the truncation errors, and the round off errors. The truncation error is the inaccuracy we have in the computation of $f'(x)$ due to h not being zero. The round off error is inherent to using floating point numbers and floating point arithmetic (as against using infinite precision numbers, which would be prohibitive expensive).

Numerical differentiation is thus not a feasible approach for computing gradients while building deep learning models. The only place where numerical differentiation comes in handy is quickly checking if gradients are being computed correctly. This is highly recommended when you have computed gradients manually or with a new/unknown automatic differentiation library. Ideally, this check should be put in as an automated check/assertion before staring SGD.

■ **Note** Numerical differentiation is implemented in a Python package called Scipy. We do not cover it here, as it is not directly relevant to deep learning.

Symbolic Differentiation

Symbolic differentiation in its basic form is a set of symbol rewriting rules applied to the loss function to arrive at the derivatives/gradients. Consider two of such simple rules:

$$\frac{d}{dx}\left(f(x) + g(x)\right) = \frac{d}{dx}f(x) + \frac{d}{dx}g(x)$$

and

$$\frac{d}{dx}x^n = nx^{(n-1)}.$$

Now, given a function like $f(x) = 2x^3 + x^2$, we can successively apply the symbol writing rules to first arrive at

$$f'(x)=\frac{d}{dx}\left(2x^3\right)+\frac{d}{dx}\left(x^2\right)$$

by applying the first rewriting rule and $f'(x)=6x^2+2x$ by applying the second rule. Symbolic differentiation is thus automating what we do when we derive gradients manually. Of course, the number of such rules can be large, and more sophisticated algorithms can be leveraged to make this symbol rewriting more efficient. However, in its essence, symbolic differentiation is simply the application of a set of symbol rewriting rules. The key advantage of symbolic differentiation is that it generates a legible mathematical expression for the derivative/gradient that can be understood and analyzed.

The key problem with symbolic differentiation is that it is limited to the symbolic differentiation rules already defined, which can cause us to hit roadblocks when trying to minimize complicated loss functions. An example of this is when your loss function involves an if-else clause or a for/while loop. In a sense, symbolic differentiation is differentiating a (closed form) mathematical expression; it is not differentiating a given computational procedure.

Another problem with automatic differentiation is that a naïve application of symbol rewriting rules, in some cases, can lead to an explosion of symbolic terms (expression swell) and make the process computationally unfeasible. Typically, a fair amount of compute effort is required to simplify such expressions and produce a closed form expression of the derivative.

■ **Note** Symbolic differentiation is implemented in a Python package called SymPy. We do not cover it here, as it is not directly relevant to deep learning.

Automatic Differentiation Fundamentals

The first key intuition behind automatic differentiation is that all functions of interest (which we intend to differentiate) can be expressed as compositions of elementary functions for which corresponding derivative functions are known. Composite functions, thus can be differentiated by applying the chain rule for derivatives. This intuition is also at the basis of symbolic differentiation.

The second key intuition behind automatic differentiation is that, rather than storing and manipulating intermediate symbolic forms of derivatives of primitive functions, one can simply evaluate them (for a specific set of input values) and thus address the issue of expression swell. Since intermediate symbolic forms are being evaluated, we do not have the burden of simplifying the expression. Note that this prevents us from getting a closed form mathematical expression of the derivate like the one symbolic differentiation gives us; what we get via automatic differentiation is the evaluation of the derivative for a given set of values.

The third key intuition behind automatic differentiation is that, because we are evaluating derivatives of primitive forms, we can deal with arbitrary computational procedures and not just closed form mathematical expressions. That is, our function can contain if-else statements, for-loops, or even recursion. The way automatic differentiation deals with any computational procedure is to treat a single evaluation of the procedure (for a given set of inputs) as a finite list of elementary function evaluations over the input variables to produce one or more output variables. While there might be control flow statements (if-else statements, for-loops, etc.), ultimately, there is a specific list of function evaluations that transform the given input to the output. Such a list/evaluation trace is referred to as a Wengert list.

Let us set the stage for discussing automatic differentiation by introducing a simple function $f(x_1, x_2)=\left(x_1^2+x_2^2\right)^{\frac{1}{2}}$. Figure 9-1 shows the computational graph for the function. Note that we also introduce some intermediate variables for convenience.

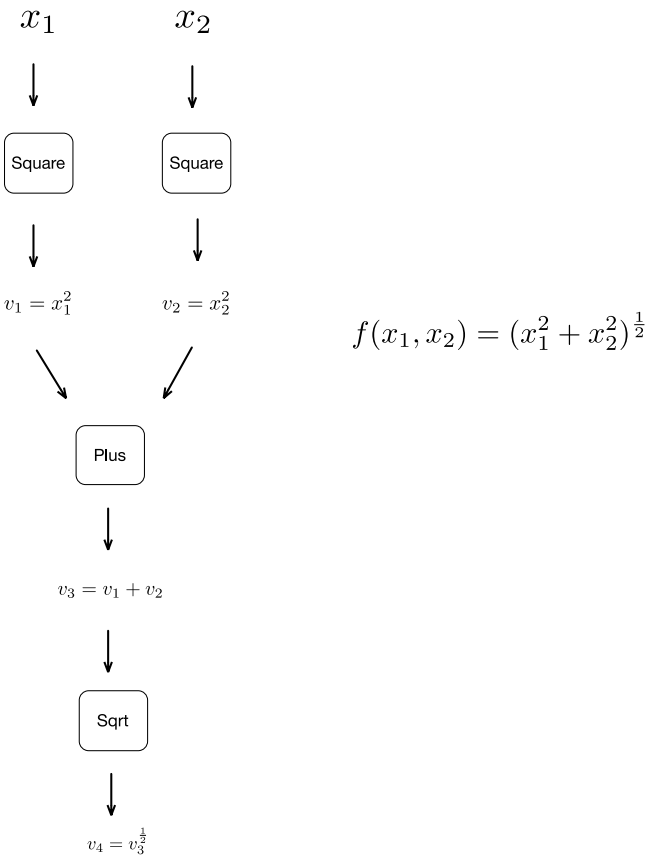


Figure 9-1. *A simple function and its computational graph*

Forward/Tangent Linear Mode

The forward mode (also called tangent linear mode) of automatic differentiation associates each intermediate variable in the computational graph with a derivative. More formally, we have $\dot{v}_i = \frac{\partial v_i}{\partial x}$ for all values of *i* where $\dot{v}_i$ is the derivative of the intermediate variable v_i with respect to an input variable/other intermediate variable *x*. Figure 9-2 illustrates this for the example function we introduced earlier.

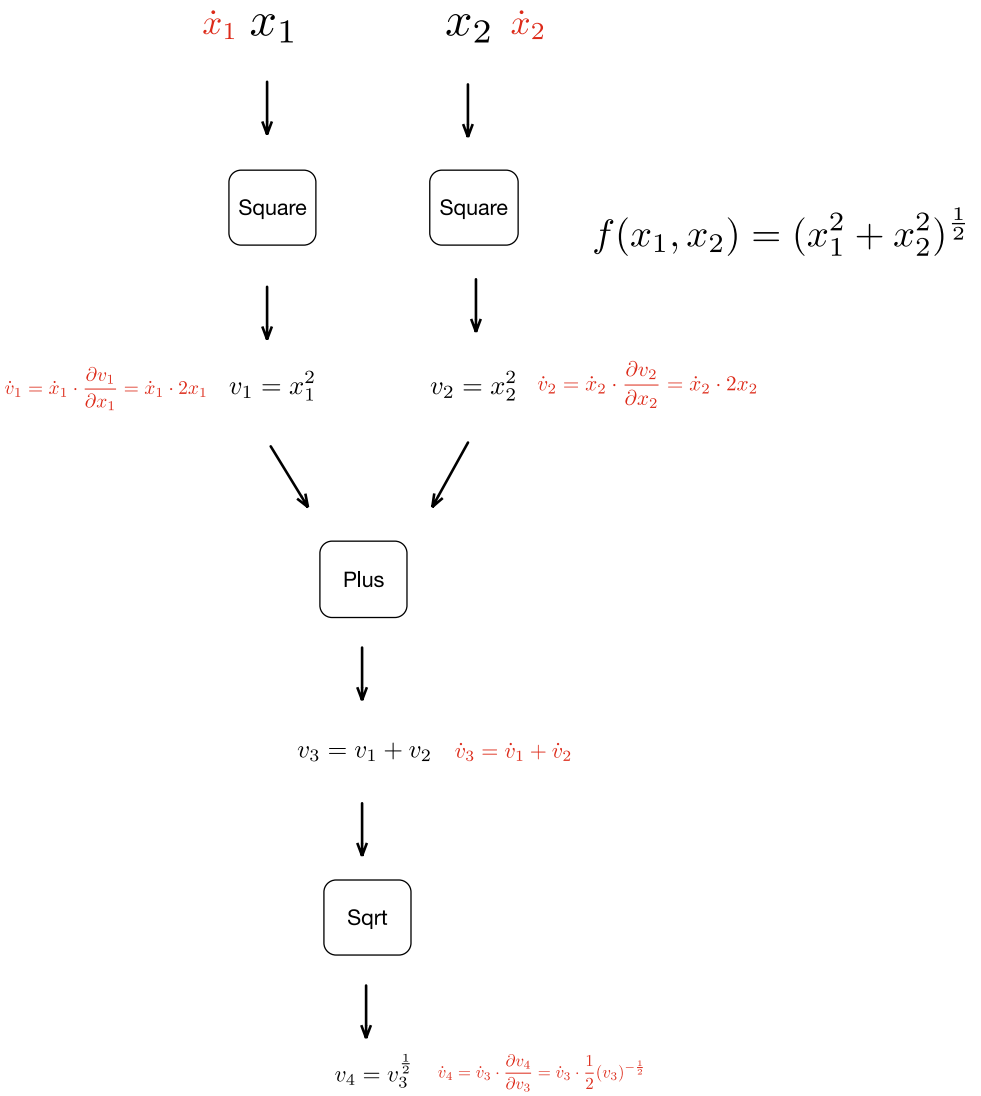
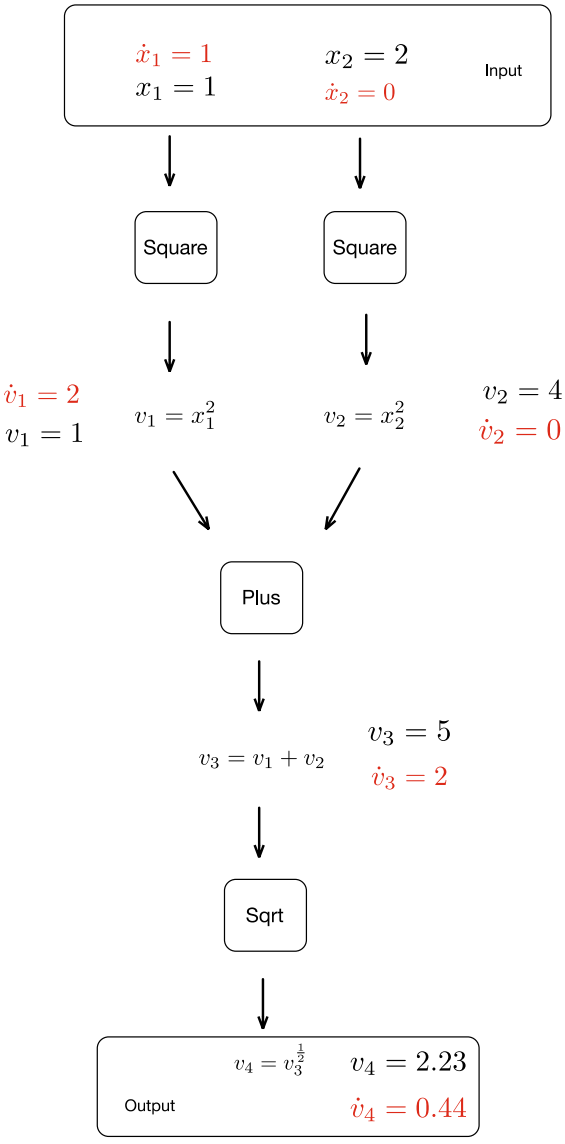


Figure 9-2. *Associating every intermediate variable with a derivative in forward mode automatic differentation*

Note that there are a number of ways by which such a computational graph can be constructed and the derivatives for the intermediate variables can be associated to the nodes. This can be done explicitly by parsing the given function (to be differentiated) or implicitly by using operator overloading. For the purposes of this discussion it suffices to say that the given function can be decomposed into its elementary functions and, using the derivatives of the elementary functions and the chain rule, we can associate intermediate variables with their corresponding derivatives.

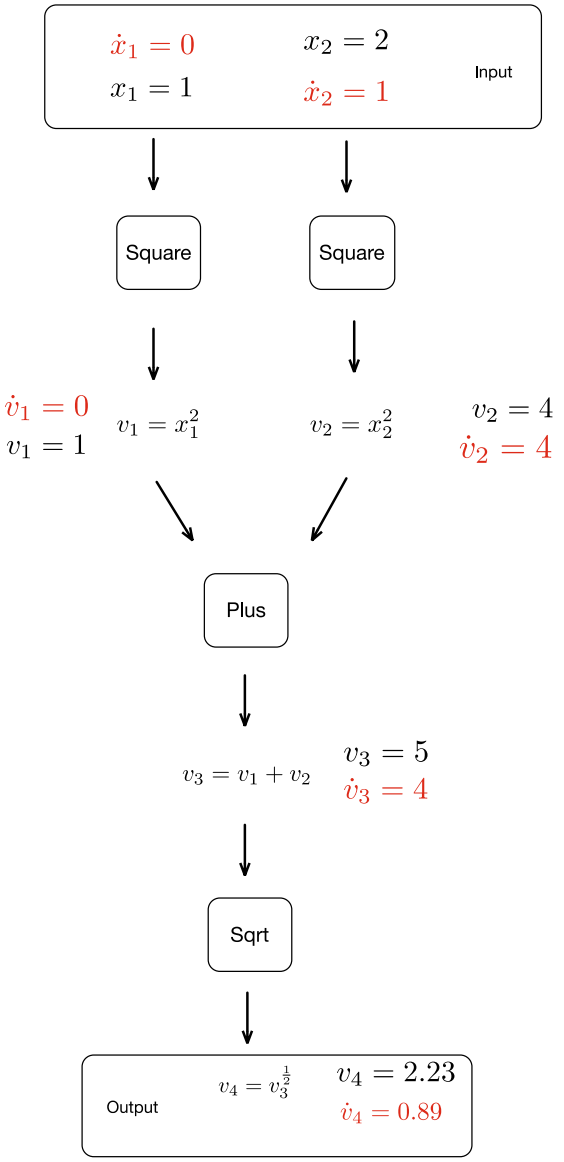
Given such an augmented computational graph, we can evaluate the value of the (partial) derivative of the given function with respect to a particular variable (and a set of inputs) by evaluating the expressions associated with the augmented computational graph. For this evaluation, we have values for all the input variables and we set all the values of the derivatives of the input variables to 0, except for the variable for which we intend to evaluate the partial derivative, which we set to 1. Figures 9-3 and 9-4 illustrate such an evaluation of the computational graph.



$$f(x_1, x_2) = (x_1^2 + x_2^2)^{\frac{1}{2}}$$

$$\nabla_{x_1} f(x_1, x_2) = x_1 (x_1^2 + x_2^2)^{-\frac{1}{2}}$$

Figure 9-3. *Computing the derivative (partial with respect to x_1) for a particular set of values of x_1 and x_2*



$$f(x_1, x_2) = (x_1^2 + x_2^2)^{\frac{1}{2}}$$

$$\nabla_{x_1} f(x_1, x_2) = x_1(x_1^2 + x_2^2)^{-\frac{1}{2}}$$

Figure 9-4. *Computing the derivative (partial with respect to x_2) for a particular set of values of x_1 and x_2*

It must be noted that with forward mode automatic differentiation we need to perform one evaluation of the augmented computational graph for computing the partial derivative with respect to each input variable. It follows that for computing the gradient for a function with respect to n input variable we would require n evaluations. Thus, forward mode automatic differentiation is expensive when it comes to computing gradients for functions with a lot of input variables, which is a common case in deep learning where loss functions consist of many input variables and a single output variable.

It is also clear that forward mode automatic differentiation is a fairly straightforward application of the chain rule and can be implemented easily using operator overloading. Forward mode automatic differentiation is often implemented by the use of dual numbers, which are defined as a truncated Taylor series of the form $v + \dot{v}\epsilon$. Arithmetic on dual numbers can be defined using $\epsilon^2 = 0$ and treating a non-dual number as $v + 0\dot{v}$. Dual numbers, in a sense, carry the derivative with them throughout their lifetime. Thus, given that we have a complete implementation of dual numbers, the derivatives can simply be computed as a side/parallel effect of the operations on the dual component.

Reverse/Cotangent/Adjoint Linear Mode

The reverse mode (also called cotangent linear mode or adjoint mode) of automatic differentiation also associates each intermediate variable in the computational graph with a derivative computed backward from the output. This bears a striking resemblance to backpropagation. More formally, we have $\bar{v}_i = \frac{\partial y_i}{\partial v_i}$ for all values of i where $\bar{v}_i$ is the derivative of the output/intermediate variable y_i for all values of i. Figure 9-5 illustrates this for the example function we introduced earlier.

To evaluate the derivative, we first do a forward pass over the augmented computational graph as shown in Figure 9-6. This is followed by a reverse pass in which the derivatives are computed, which is illustrated in Figure 9-7.

Reverse mode automatic differentiation computes all the partial derivatives in a single forward pass and a single reverse pass and thus scales well with respect to functions with many input variables common to loss functions in deep learning.

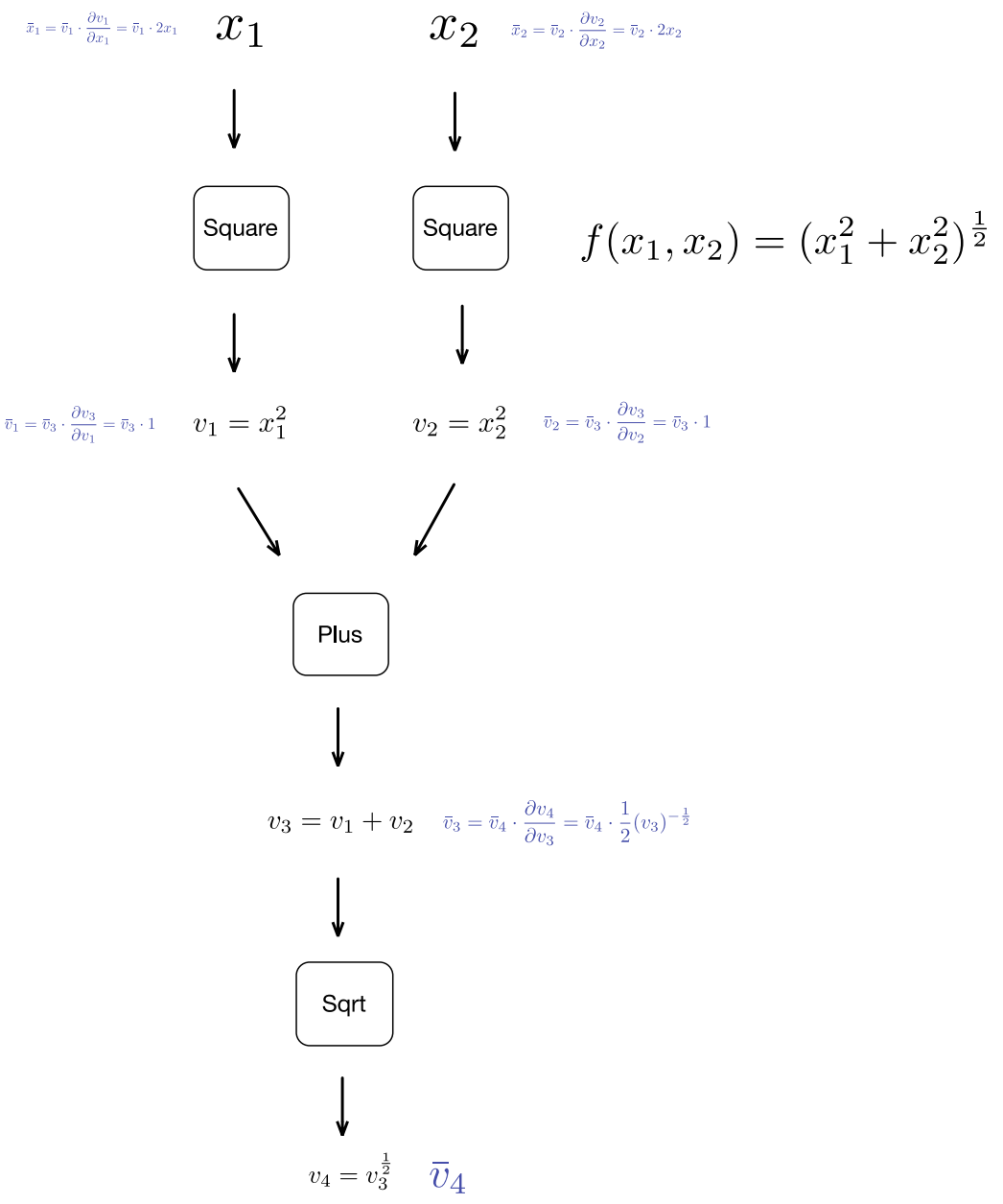


Figure 9-5. *Associating every intermediate variable with a derivative in reverse mode automatic differentiation*

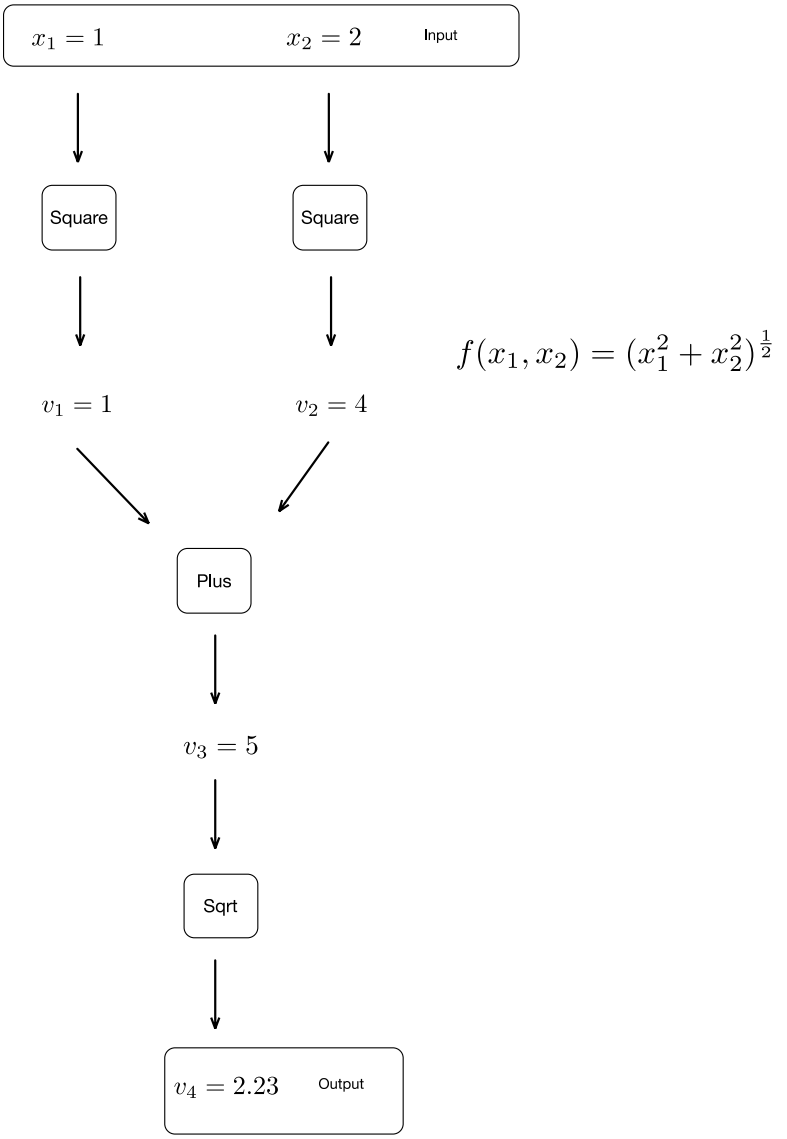


Figure 9-6. *Forward pass of reverse mode automatic differentiation*

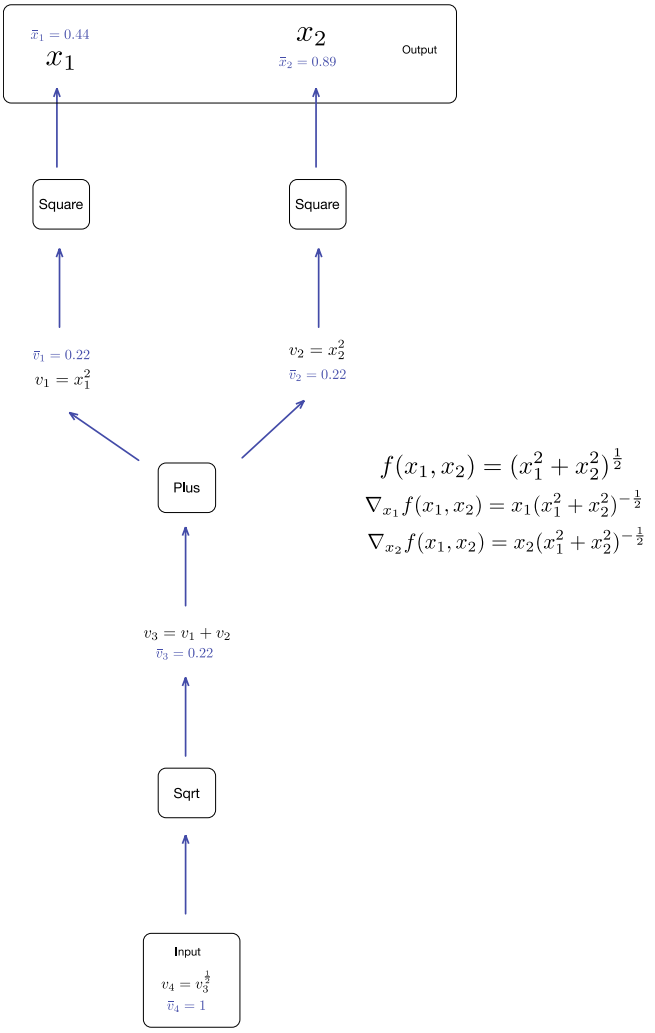


Figure 9-7. *Backward pass of reverse mode automatic diffrentiation*

Implementation of Automatic Differentiation

Let us now take a look at how Automatic Differentiation is commonly implemented. The three key approaches are using source code transformation and operator overloading (explicit or implicit dual number implementation).

Source Code Transformation

The source code transformation approach involves the user implementing the loss function in a regular programming language and then using an automatic differentiation tool to generate the corresponding gradient function. These two can then be compiled by the standard build tool chain to be used as part of a larger application. Refer to Figure 9-8.

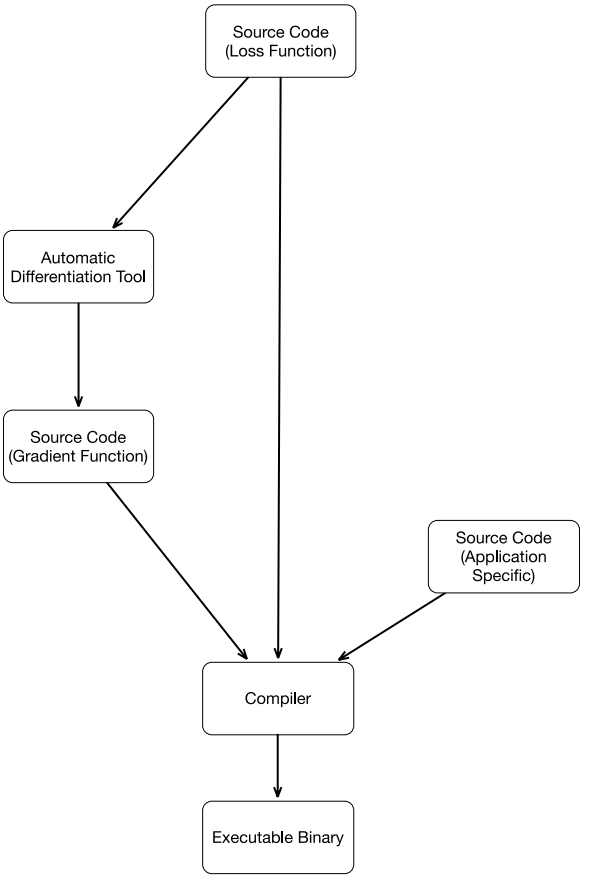


Figure 9-8. *Source Code Transformation*

Operator Overloading

The operator overloading approach basically is an explicit/implicit implementation of the dual number approach wherein the corresponding differentiation operation is implemented for every primitive operation of interest. Users implement their loss functions using the primitive operations and the computation of the gradients happen by the invocation of the overloaded method implementing the differentiation operation. Refer to Figure 9-9.

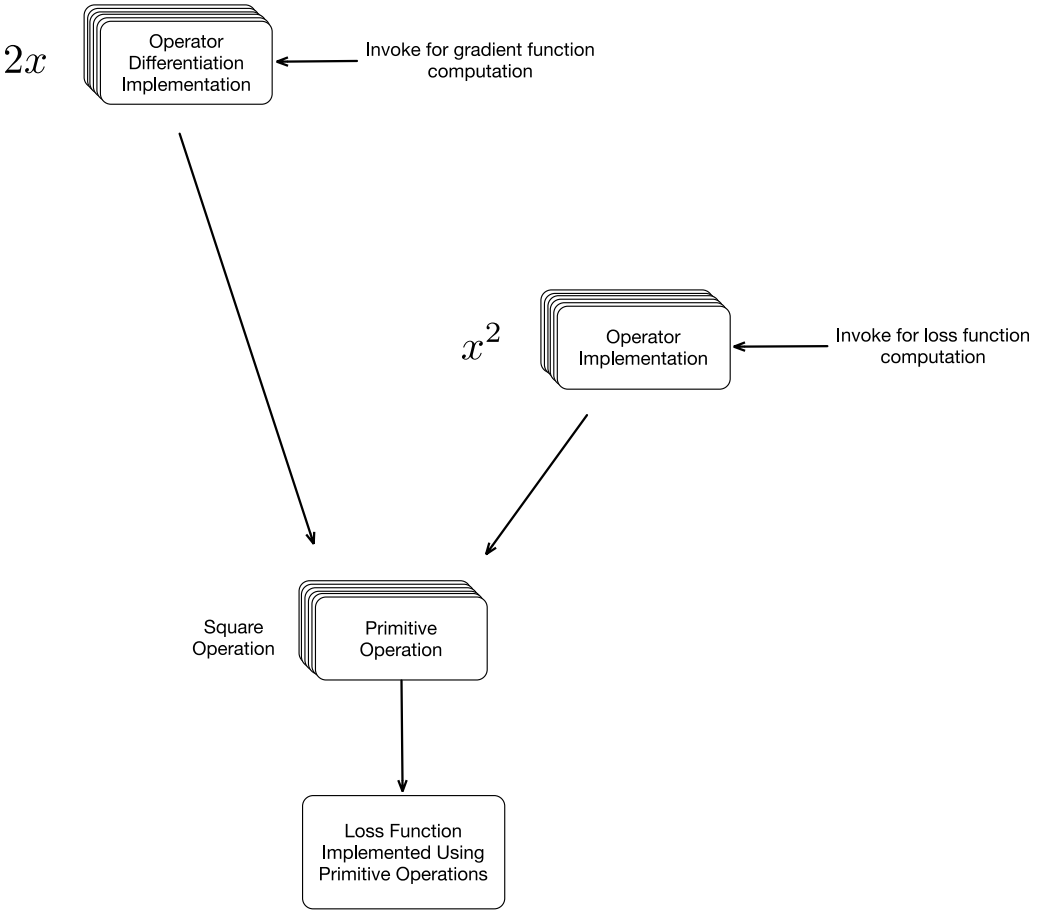


Figure 9-9. *Operator Overloading*

■ **Note** One key implementation detail surrounding the operator overloading approach is whether the operators in question are those already implemented in established library/core language or if the automatic differentiation tool provides its own operators. Autograd is an example an automatic differentiation tool that overloads the established Numpy Library whereas Theano provides its own operators for which corresponding differential operations are implemented.

Hands-on Automatic Differentiation with Autograd

We will now do a hands-on exercise with Automatic Differentiation using a Python package called Autograd. Autograd implements Reverse mode automatic differentiation and can compute derivatives for arbitrary Python and Numpy code.

Listing 9-1. Finding gradient for $f(x_1, x_2) = \left(x_1^2 + x_2^2\right)^{\frac{1}{2}}$

```
#Wrapper Around Numpy
import autograd.numpy as numpy

#Function to generate gradients
from autograd import grad

#Define the function
def f(x1, x2): return numpy.sqrt(x1 * x1 + x2 * x2)

#Compute the gradient w.r.t the first input variable x1
g_x1_f = grad(f,0)

#Compute the gradient w.r.t the second input variable x2
g_x2_f = grad(f,1)

#Evaluate and print the value of the function at x1=1, x2=2
print f(1,2)
#Produces 2.23

#Evaluate and print the value of the gradient w.r.t x1 at x1=1, x2=2
print g_x1_f(1,2)
#Produces 0.44

#Evaluate and print the value of the gradient w.r.t x2 at x1=1, x2=2
print g_x2_f(1,2)
#Produces 0.89
```

Let us get started by taking the function that we have used for discussion throughout this chapter, $f(x_1, x_2) = \left(x_1^2 + x_2^2\right)^{\frac{1}{2}}$, and finding the gradient. As will be apparent, Autograd makes this really easy. Listing 9-1 illustrates this.

Autograd provides a utility function to check the correctness of the computed gradients. Listing 9-2 illustrates this. It is a good idea to conduct such checks, especially when we are computing gradients for complicated loss functions involving control flow statements.

Listing 9-2. Checking the gradient for $f(x_1, x_2) = \left(x_1^2 + x_2^2\right)^{\frac{1}{2}}$

```
from autograd.util import quick_grad_check

#Define the function
def f(x1, x2): return numpy.sqrt(x1 * x1 + x2 * x2)

#Computes and checks the gradient for the given values

quick_grad_check(f,1.0,extra_args=[2.0])

#Output
#
#Checking gradient of <function f at 0x10504bed8> at 1.0
#Gradient projection OK
#(numeric grad: 0.447213595409, analytic grad: 0.4472135955)
```

Listing 9-3. Logistic Regression using Autograd

```
import pylab
import sklearn.datasets
import autograd.numpy as np
from autograd import grad

# Generate the data
train_X, train_y = sklearn.datasets.make_moons(500, noise=0.1)

# Define the activation, prediction and loss functions for Logistic Regression
def activation(x):
    return 0.5*(np.tanh(x) + 1)

def predict(weights, inputs):
    return activation(np.dot(inputs, weights))

def loss(weights):
    preds = predict(weights, train_X)
    label_probabilities = preds * train_y + (1 - preds) * (1 - train_y)
    return -np.sum(np.log(label_probabilities))

# Compute the gradient of the loss function
gradient_loss = grad(loss)

# Set the initial weights
weights = np.array([1.0, 1.0])

# Steepest Descent
loss_values = []
learning_rate = 0.001
for i in range(100):
    loss_values.append(loss(weights))
    step = gradient_loss(weights)
    weights -= step * learning_rate

# Plot the decision boundary
x_min, x_max = train_X[:, 0].min() - 0.5, train_X[:, 0].max() + 0.5
y_min, y_max = train_X[:, 1].min() - 0.5, train_X[:, 1].max() + 0.5
x_mesh, y_mesh = np.meshgrid(np.arange(x_min, x_max, 0.01), np.arange(y_min, y_max, 0.01))
Z = predict(weights, np.c_[x_mesh.ravel(), y_mesh.ravel()])
Z = Z.reshape(x_mesh.shape)
cs = pylab.contourf(x_mesh, y_mesh, Z, cmap=pylab.cm.Spectral)
pylab.scatter(train_X[:, 0], train_X[:, 1], c=train_y, cmap=pylab.cm.Spectral)
pylab.colorbar(cs)

# Plot the loss over each step
pylab.figure()
pylab.plot(loss_values)
pylab.xlabel("Steps")
pylab.ylabel("Loss")
pylab.show()
```

Let us now compute gradient for something a bit more complicated, the loss function for logistic regression. Let's also fit the model using steepest descent. Listing 9-1 shows the code for the same, and Figures 9-10 and 9-11 show the decision boundary and the loss over the steepest descent steps.

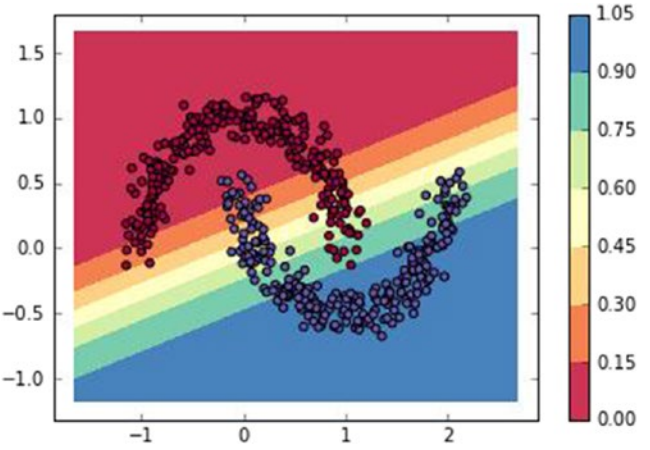


Figure 9-10. *Decision boundary and training data for Logistic Regression*

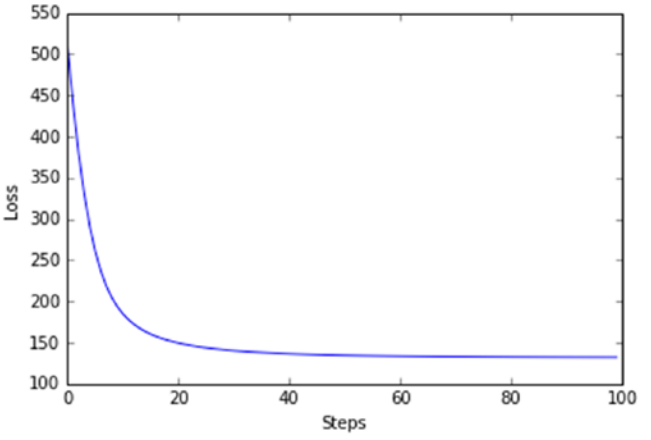


Figure 9-11. *Loss over steps for Logistic Regression*

Summary

In this chapter we covered the basics of Automatic Differentiation, which is commonly referred to as backpropagation in the Neural Network Community. The key take-away for the reader in this chapter is that automatic differentiation enables the computation of gradients for arbitrarily complex loss functions and is one of the key enabling technologies for deep learning. The reader should also internalize the concepts of automatic differentiation and how it is different from both symbolic and numerical differentiation.

CHAPTER 10

Introduction to GPUs

This chapter introduces the reader to GPU (Graphics Processing Unit)-based computation, which has played and will continue to play a big role in the successful application of Deep Learning in a variety of application domains. Typically, a deep learning practitioner is working with high-level libraries like Keras or Theano, which automatically translates the computation to be performed seamlessly to CPU or GPU. While in a majority of the cases, a practitioner of deep learning is not required to understand the internal workings of the GPU (as many high-level libraries are available), it is essential to be aware of the basics.

The essence of GPU-based computation is the notion of Single Instruction, Multiple Data (SIMD), wherein the same computation is being performed in parallel (over may cores) on multiple data points. This computational paradigm is very suitable for compute heavy linear algebra operations. As we have seen in earlier chapters, the core computation involved in training deep learning models is the computation of gradients and updating the parameters based on these gradients. At the heart of this lie basic linear algebraic operations (dot products, vector matrix multiplications, etc.) and this GPU-based computation is quite suitable for training (and making predictions) using the same.

Let us start by describing the key elements of such a GPU-based computation. Figure 10-1 schematically illustrates these key elements.

N. Ketkar, *Deep Learning with Python*, DOI 10.1007/978-1-4842-2766-4_10

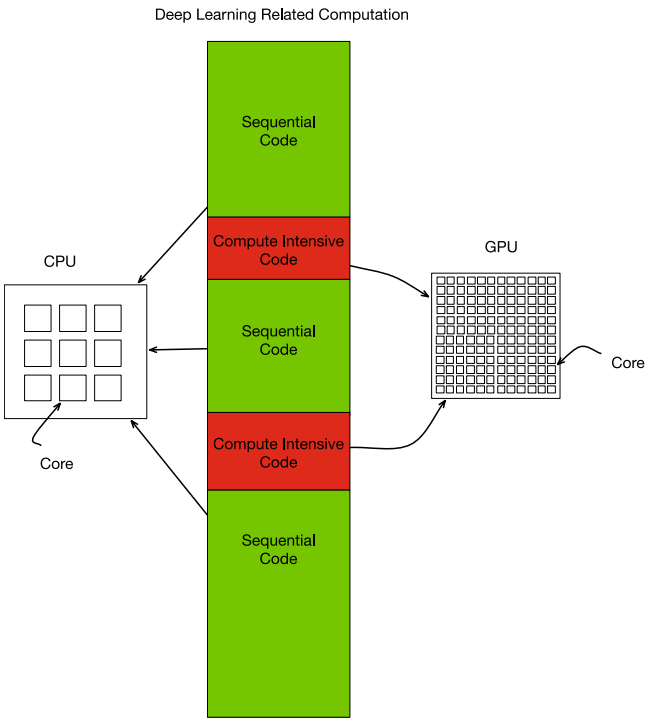


Figure 10-1. *GPU-based Computation*

The following points are to be noted:

1. Deep learning-related computation involves some code to be executed sequentially and some compute-intensive code which can be parallelized.
2. Typically, the sequential code involves loading the data from disk, etc., which is handled by the CPU.
3. The computationally heavy code typically involves computing the gradients and updating the parameters. Data for this computation is first transferred to the GPU memory, and this computation then happens on the GPU.
4. Next, the results are brought back to the main memory for further sequential processing.
5. There might be multiple blocks of such computationally heavy code interlaced with sequential code.

■ **Note** There are two main ecosystems built around GPUs: one is CUDA, which is specific to Nvidia, and OpenCL, which is vendor-neutral. We will be covering concepts around GPU computation in the context of OpenCL.

Let us start by looking at the overall programming model for GPU-based computation as described by OpenCL. OpenCL is a vendor neural framework for heterogeneous computation involving CPUs, GPUs, DSPs (Digital Signal Processors), and FPGAs (Field Programmable Gate Arrays), etc. Figure 10-2 illustrates the physical view of the system.

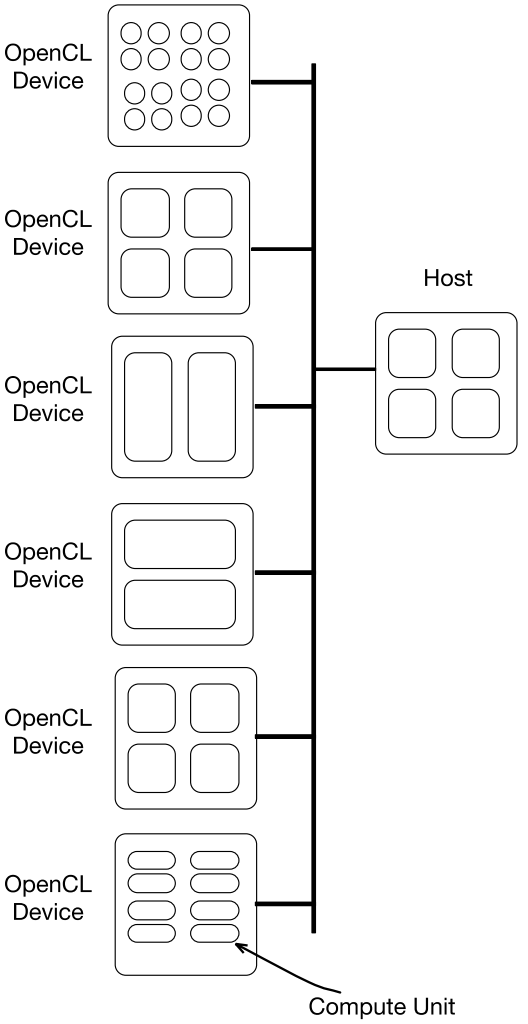


Figure 10-2. *OpenCL System Physical View*

The following points are to be noted:

1. The overall system consists of the Host and a number of OpenCL devices.
2. The host refers to the CPU running the OS, which can communicate with a number of OpenCL devices.
3. OpenCL devices are heterogeneous, as in, different. They might be involving CPUs, GPUs, DSPs (Digital Signal Processors), and FPGAs (Field Programmable Gate Arrays), etc.
4. OpenCL devices contain one or more compute units.

Let's now look at the logical view of an OpenCL system illustrated in Figure 10-3.

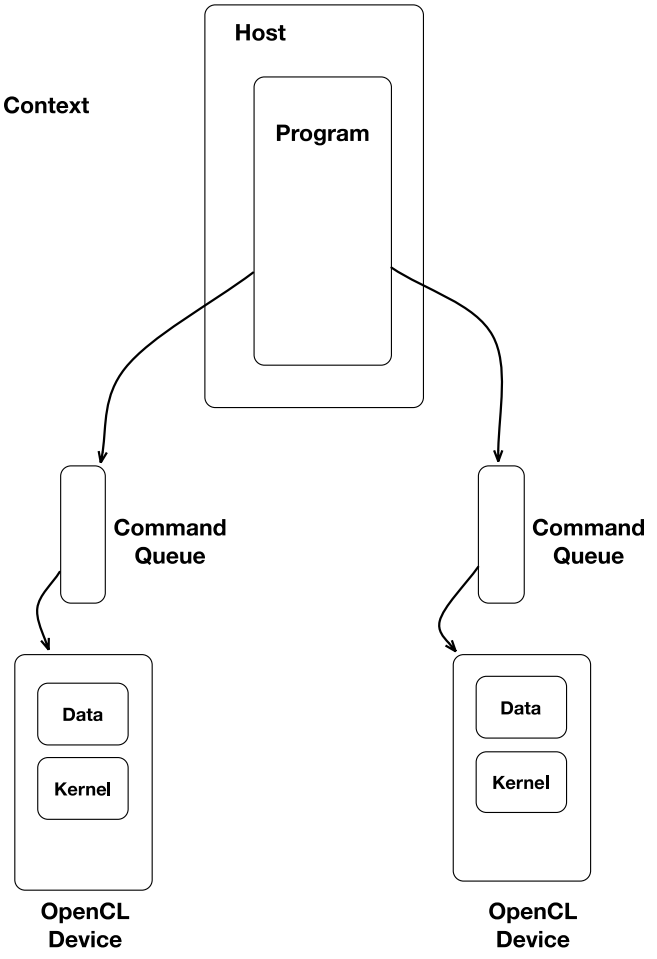


Figure 10-3. *OpenCL System Logical View*

The following points are to be noted:

1. An OpenCL program runs on the host system.
2. The OpenCL program communicates with OpenCL devices using command queues. Each OpenCL device has a separate command queue.
3. Each OpenCL device houses data in its memory, sent to it by the program running on the host.
4. Each OpenCL device runs code sent to it by the host program, referred to as the kernel.
5. The host program, the command queues, the data, and the kernels together constitute an execution context.
6. The execution context essentially is the logical envelopment of the heterogeneous computation. The host program orchestrates this computation by sending data and code to be executed to the OpenCL devices and getting the results.

Let's now take a look at the logical memory layout on an OpenCL device. Figure 10-4 illustrates the same.

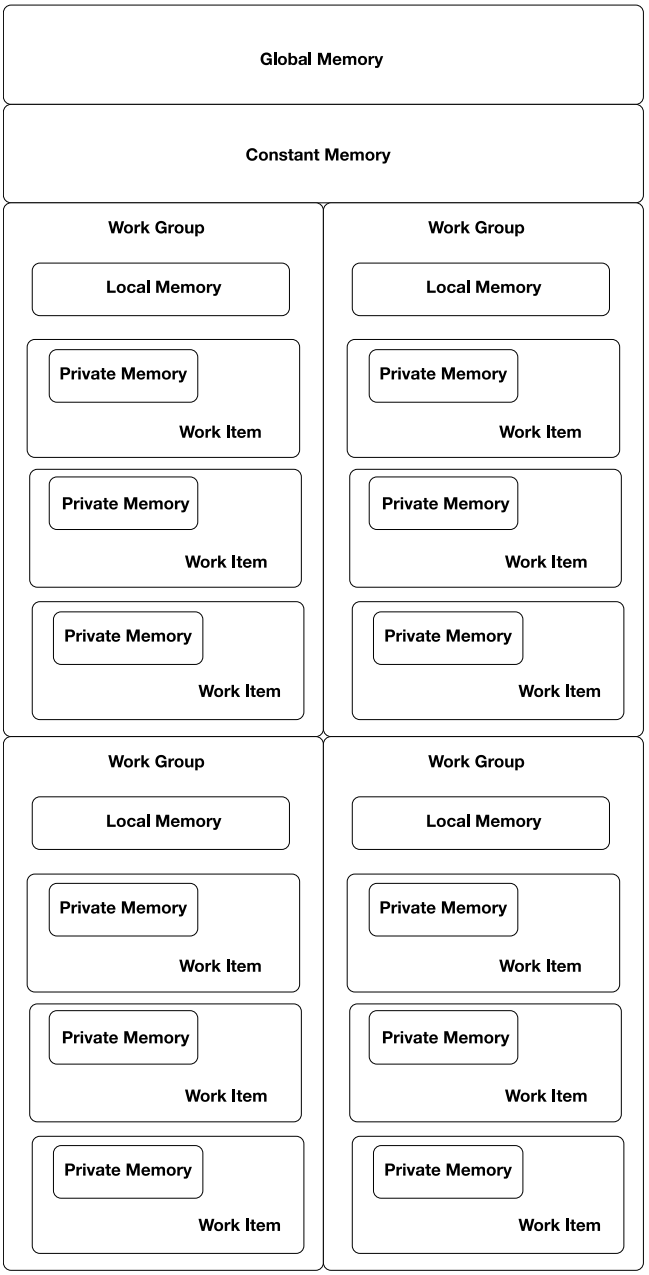


Figure 10-4. *Device Memory*

The following points are to be noted:

1. An OpenCL device has a global memory, which is accessible to the host program as well as all the running kernels on the device.

2. An OpenCL device has a constant memory, which is just like global memory but it is read-only for an executing kernel.
3. A Work Item is the logical unit of parallelism and it has its own private memory. Only the kernel code corresponding to this particular work item is aware of this memory.
4. A Work Group is the logical unit of synchronization and it contains a number of Work Items. Note that any synchronization can only be done within a Work Group.
5. A Work group has its own Local Memory that can only be accessed from within the Work Group.

Let us now take a look at the programming model with respect to an OpenCL Device (Figure 10-5).

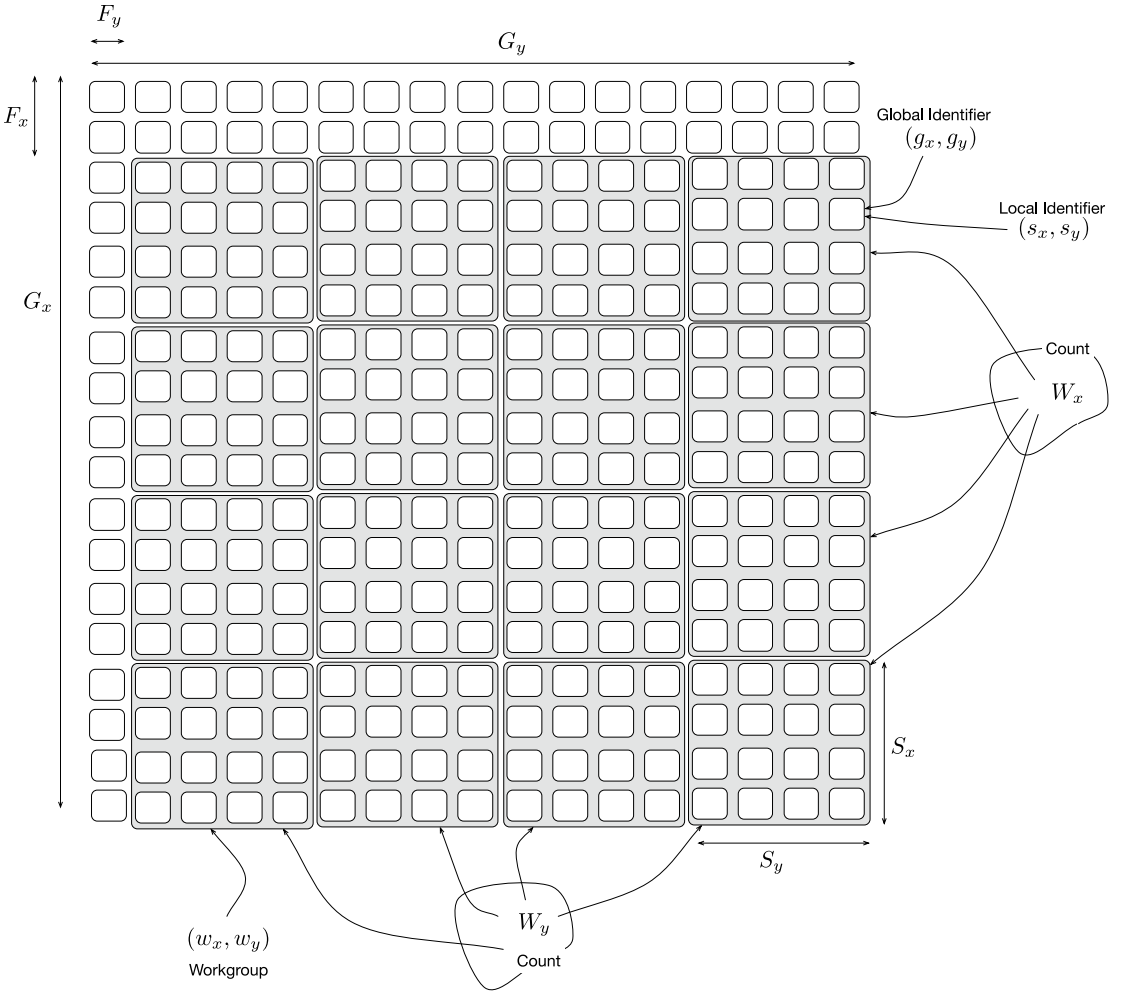


Figure 10-5. *Two-Dimensional NDRange Index Space*

The following points are to be noted:

1. An OpenCL kernel is launched to perform work on data already transferred to the device memory. While launching, the number of work groups and the numbers of work items in each work group is logically specified.
2. The kernel is invoked in parallel for each work item in a work group. Work groups execute in no particular order and a kernel can find out the current work item identifier and work group identifier.
3. Synchronization can happen only within a work group.
4. A work item identifier can be 1-, 2-, or 3- dimensional (NDRange). This basically makes it easy to write kernels for 1-D (time series), 2-D (images), and 3-D (volumes) data sets.

Let us now introduce some notation that will allow us to describe the indexing. We will assume the indexing is 2-D, but the same reasoning applies to 1-D or 3-D data. We denote by (G_x, G_y) the global indexing space. Let us now look at how this global indexing space gets broken into work groups and work items. For convenience, we define offsets (F_x, F_y) which define the portion of the indexing that is not broken into work items and work groups. Let (S_x, S_y) define the size of the work group and (W_x, W_y) define the number of work groups. Along similar lines, let (g_x, g_y) denote the global identifiers, (s_x, s_y) denote the local identifiers, and let (w_x, w_y) denote the work group identifiers. Then, the relationship between local and global identifiers is described as $(g_x, g_y) = (w_x S_x + s_x + F_x, w_y S_y + s_y + F_y)$.

Writing a kernel for a given computation basically involves leveraging this identifier mechanism and the parallel invocations of the kernels over work items and work groups to perform the task at hand. Listings 10-2 and 10-3 illustrate this for vector addition and matrix multiplication, respectively. Listing 10-1 simply prints out the details of the OpenCL system that the reader can use to determine the details of their system.

Listing 10-1. Getting Information on GPUs

```
import pyopencl as cl

print "OpenCL Platforms and Devices"
for platform in cl.get_platforms():
    print "Platform Name: ", platform.name
    print "Platform Vendor", platform.vendor
    print "Platform Version:", platform.version
    print "Platform Profile:", platform.profile
    for device in platform.get_devices():
        print "\n"
        print "\tDevice Name ", device.name
        print "\tDevice Type ",  cl.device_type.to_string(device.type)
        print "\tDevice Max Clock Speed ", "{0} Mhz".format(device.max_clock_frequency)
        print "\tDevice Compute Units ",  "{0}".format(device.max_compute_units)
        print "\tDevice Local Memory ",  "{0:.0f} KB".format(device.local_mem_size/1024.0)
        print "\tDevice Constant Memory ", "{0:.0f} KB".format(device.max_constant_buffer_
size/1024.0)
        print "\tDevice Global Memory " "{0:.0f} GB".format(device.global_mem_size/
(1024*1024*1024.0))

# OpenCL Platforms and Devices
# Platform Name:  Apple
# Platform Vendor Apple
```

```
# Platform Version: OpenCL 1.2 (Nov 18 2015 20:45:47)
# Platform Profile: FULL_PROFILE
#
#
#       Device Name  Intel(R) Core(TM) i7-4770HQ CPU @ 2.20GHz
#       Device Type  CPU
#       Device Max Clock Speed  2200 Mhz
#       Device Compute Units  8
#       Device Local Memory  32 KB
#       Device Constant Memory  64 KB
#       Device Global Memory 16 GB
#
#
#       Device Name  Iris Pro
#       Device Type  GPU
#       Device Max Clock Speed  1200 Mhz
#       Device Compute Units  40
#       Device Local Memory  64 KB
#       Device Constant Memory  64 KB
#       Device Global Memory 2 GB
```

Listing 10-2. Vector Addition

```
import numpy as np
import pyopencl as cl
import time

vector1 = np.random.random(5000000).astype(np.float32)
vector2 = np.random.random(5000000).astype(np.float32)

cl_context = cl.create_some_context()
queue = cl.CommandQueue(cl_context)
mf = cl.mem_flags
vector1_in_gpu = cl.Buffer(cl_context, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=vector1)
vector2_in_gpu = cl.Buffer(cl_context, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=vector2)
result_in_gpu = cl.Buffer(cl_context, mf.WRITE_ONLY, vector1.nbytes)

cl_program = cl.Program(cl_context, """
__kernel void sum(
    __global const float *vector1, __global const float *vector2, __global float *result)
{
  int i = get_global_id(0);
  result[i] = vector1[i] + vector2[i];
}
""").build()

t0 = time.time()
cl_program.sum(queue, vector1.shape, None, vector1_in_gpu, vector2_in_gpu, result_in_gpu)
t1 = time.time()
gpu_time = t1 - t0
print "GPU Time", gpu_time
```

```
result_in_numpy = np.empty_like(vector1)
cl.enqueue_copy(queue, result_in_numpy, result_in_gpu)

t0 = time.time()
cpu_result = vector1 + vector2
t1 = time.time()
cpu_time = t1 - t0
print "CPU Time", cpu_time

print "Norm of Difference", np.linalg.norm(result_in_numpy - cpu_result)

# GPU Time 0.00202608108521
# CPU Time 0.00995397567749
# Norm of Difference 0.0
```

Listing 10-3. Matrix Multiplication

```
import numpy as np
import pyopencl as cl
import time

matrix1 = np.random.random((500,500)).astype(np.float32)
matrix2 = np.random.random((500,500)).astype(np.float32)

cl_context = cl.create_some_context()
queue = cl.CommandQueue(cl_context)
mf = cl.mem_flags
matrix1_in_gpu = cl.Buffer(cl_context, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=matrix1)
matrix2_in_gpu = cl.Buffer(cl_context, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=matrix2)
result_in_gpu = cl.Buffer(cl_context, mf.WRITE_ONLY, matrix1.nbytes)

cl_program = cl.Program(cl_context, """
__kernel void product(
    int size, __global const float *matrix1, __global const float *matrix2, __global float
*result)
{
    int i = get_global_id(0);
    int j = get_global_id(1);
    result[i + size * j] = 0;
    for (int k = 0; k < size; k++)
    {
        result[i + size * j] += matrix1[k + size * i] * matrix2[j + size * k];
    }
}
""").build()

t0 = time.time()
cl_program.product(queue, matrix1.shape, None, np.int32(len(matrix1)), matrix1_in_gpu,
matrix2_in_gpu, result_in_gpu)
t1 = time.time()
gpu_time = t1 - t0
print "GPU Time", gpu_time
```

```
result_in_numpy = np.empty_like(matrix1)
cl.enqueue_copy(queue, result_in_numpy, result_in_gpu)

t0 = time.time()
cpu_result = np.dot(matrix1, matrix2)
t1 = time.time()
cpu_time = t1 - t0
print "CPU Time", cpu_time

print "Norm of Difference", np.linalg.norm(result_in_numpy - cpu_result.T)

# GPU Time 0.00202608108521
# CPU Time 0.00995397567749
# Norm of Difference 0.0
```

Summary

In this chapter we have introduced the reader to GPU-based computation, which is one of the key enabling technologies for Deep Learning.

One key point to note is that, in this chapter, we have covered the basics of GPU computation using OpenCL, which is vendor-neutral. The concepts apply with minor variation to other vendor-specific GPU computation libraries like CUDA, which is both older and popular as compared to OpenCL. The reader is advised to try out the examples in the source code listings in the chapter following, while reading up on the documentation of CUDA, which would be much easier to follow given that the reader has internalized the foundations. Libraries like cuDNN, which is a CUDA-based library for deep learning, is recommended further reading. It must be noted, however, that in many cases, when it comes to applying deep learning to a real-world problem, it suffices to use high-level libraries like Theano and Keras, which generate GPU code.

The second key point to note is the importance of GPU-based computation for Deep Learning. The single instruction, multiple data paradigm (SIMD) is ideal for deep learning, as most of computation with respect to deep learning boils down to stochastic gradient descent (SGD). At the heart of SGD we have the computation of gradients, which are essentially linear algebraic (vector/matrix) operations. As data sets and the sizes of the parameters grow, it becomes essential to perform SGD in a scalable way, and GPUs currently are the best suited computational paradigm.

Index

N. Ketkar, *Deep Learning with Python*, DOI 10.1007/978-1-4842-2766-4

F

G

H

I, J

K

L

M

N

O

P, Q

R

S

T

U

V

W, X, Y, Z