Neural Network Programming with TensorFlow

Unleash the power of TensorFlow to train efficient neural networks
Neural Network Programming with TensorFlow

Unleash the power of TensorFlow to train efficient neural networks
## Credits

<table>
<thead>
<tr>
<th>Authors</th>
<th>Copy Editor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manpreet Singh Ghotra</td>
<td>Tasneem Fatehi</td>
</tr>
<tr>
<td>Rajdeep Dua</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Reviewer</th>
<th>Project Coordinator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Giancarlo Zaccone</td>
<td>Manthan Patel</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Commissioning Editor</th>
<th>Proofreader</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amey Varangaonkar</td>
<td>Safis Editing</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Acquisition Editor</th>
<th>Indexer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Varsha Shetty</td>
<td>Rekha Nair</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Content Development Editor</th>
<th>Graphics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tejas Limkar</td>
<td>Tania Dutta</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Technical Editor</th>
<th>Production Coordinator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sagar Sawant</td>
<td>Aparna Bhagat</td>
</tr>
</tbody>
</table>
About the Authors

**Manpreet Singh Ghotra** has more than 15 years of experience in software development for both enterprise and big data software. He is currently working on developing a machine learning platform/API's using open source libraries and frameworks like TensorFlow, Keras, Apache Spark and PredictionIO at Salesforce. He has worked on various machine learning systems like sentiment analysis, spam detection, image moderation and anomaly detection. He was part of the machine learning group at one of the largest online retailers in the world, working on transit time calculations using R and Apache Mahout. With a master's and a postgraduate degree in machine learning, he has contributed to and worked for the machine learning community.

His GitHub profile is [https://github.com/badlogicmanpreet](https://github.com/badlogicmanpreet) and you can find him on LinkedIn at [https://in.linkedin.com/in/msghotra](https://in.linkedin.com/in/msghotra).

*This book is dedicated to my parents, Amrik Singh and Nirmal Kaur. I would like to thank my family for bearing with me while I spent time writing this book.*

**Rajdeep Dua** has over 18 years of experience in the cloud, big data, and machine learning space. He has worked in the advocacy team for Google's big data tools, BigQuery. He has worked on the Greenplum big data platform at VMware in the developer evangelist team. He also worked closely with a team on porting Spark to run on VMware's public and private cloud as a feature set. He has taught Spark and big data at some of the most prestigious tech schools in India--IIIT Hyderabad, ISB, IIIT Delhi, and College of Engineering, Pune.
Currently, he leads the developer relations team at Salesforce India. He has published numerous blogs and hands-on labs for Einstein, a machine learning library from Salesforce and Apache Prediction IO.

He has published several big data and Spark tutorials at cluddatalab. He has also presented BigQuery and Google App Engine at the W3C conference in Hyderabad. He has led the developer relations teams at Google, VMware, and Microsoft, and he has spoken at hundreds of other conferences on the cloud.

His contributions to the open source community are for Docker, Kubernetes, Android, OpenStack, Prediction IO, and cloudfoundry projects.

*This book is dedicated to my parents. I would like to thank my wife, Manjula, and sons, Navtej and Kairav, for bearing with me while I spent time writing this book.*
About the Reviewer

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

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

Giancarlo holds a master's degree in physics from the Federico II of Naples and a second-level postgraduate master's degree in scientific computing from La Sapienza of Rome.

He is the author of the following Packt books: *Python Parallel Programming Cookbook*, *Getting Started with TensorFlow*, and *Deep Learning with TensorFlow*.

You can contact him at https://it.linkedin.com/in/giancarlozaccone.
Why subscribe?

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

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at https://www.amazon.in/dp/1788390393. If you'd like to join our team of regular reviewers, you can email us at customerreviews@packtpub.com. We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products!
Table of Contents

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

1. Maths for Neural Networks
  Understanding linear algebra
    Environment setup
      Setting up the Python environment in Pycharm
    Linear algebra structures
      Scalars, vectors, and matrices
      Tensors
    Operations
      Vectors
      Matrices
      Matrix multiplication
      Trace operator
      Matrix transpose
      Matrix diagonals
      Identity matrix
      Inverse matrix
    Solving linear equations
      Singular value decomposition
      Eigenvalue decomposition
      Principal Component Analysis
  Calculus
    Gradient
    Hessian
    Determinant
2. Deep Feedforward Networks

- Defining feedforward networks
- Understanding backpropagation
- Implementing feedforward networks with TensorFlow
- Analyzing the Iris dataset
  - Code execution
- Implementing feedforward networks with images
  - Analyzing the effect of activation functions on the feedforward networks accuracy

Summary

3. Optimization for Neural Networks

- What is optimization?
- Types of optimizers
- Gradient descent
  - Different variants of gradient descent
  - Algorithms to optimize gradient descent
- Which optimizer to choose
  - Optimization with an example

Summary

4. Convolutional Neural Networks

- An overview and the intuition of CNN
  - Single Conv Layer Computation
  - CNN in TensorFlow
    - Image loading in TensorFlow
- Convolution operations
  - Convolution on an image
  - Strides
- Pooling
  - Max pool
  - Example code
    - Average pool
- Image classification with convolutional networks
  - Defining a tensor for input images and the first convolution layer
    - Input tensor
    - First convolution layer
    - Second convolution layer
    - Third convolution layer
Flatten the layer
Fully connected layers
Defining cost and optimizer
Optimizer
First epoch
Plotting filters and their effects on an image

Summary

5. Recurrent Neural Networks
   Introduction to RNNs
      RNN implementation
         Computational graph
      RNN implementation with TensorFlow
         Computational graph
   Introduction to long short term memory networks
      Life cycle of LSTM
      LSTM implementation
         Computational graph
   Sentiment analysis
      Word embeddings
      Sentiment analysis with an RNN
         Computational graph

Summary

6. Generative Models
   Generative models
      Discriminative versus generative models
      Types of generative models
         Autoencoders
         GAN
      Sequence models
   GANs
      GAN with an example
      Types of GANs
         Vanilla GAN
         Conditional GAN
         Info GAN
         Wasserstein GAN
         Coupled GAN

Summary

7. Deep Belief Networking
   Understanding deep belief networks
DBN implementation
   Class initialization
   RBM class
      Pretraining the DBN
Model training
Predicting the label
Finding the accuracy of the model
DBN implementation for the MNIST dataset
   Loading the dataset
   Input parameters for a DBN with 256-Neuron RBM layers
   Output for a DBN with 256-neuron RBN layers
Effect of the number of neurons in an RBM layer in a DBN
   An RBM layer with 512 neurons
   An RBM layer with 128 neurons
   Comparing the accuracy metrics
DBNs with two RBM layers
Classifying the NotMNIST dataset with a DBN
Summary

8. Autoencoders
   Autoencoder algorithms
   Under-complete autoencoders
Dataset
Basic autoencoders
   Autoencoder initialization
   AutoEncoder class
   Basic autoencoders with MNIST data
      Basic autoencoder plot of weights
      Basic autoencoder recreated images plot
   Basic autoencoder full code listing
   Basic autoencoder summary
Additive Gaussian Noise autoencoder
   Autoencoder class
   Additive Gaussian Autoencoder with the MNIST dataset
      Training the model
      Plotting the weights
   Plotting the reconstructed images
   Additive Gaussian autoencoder full code listing
   Comparing basic encoder costs with the Additive Gaussian Noise autoencoder
   Additive Gaussian Noise autoencoder summary
Sparse autoencoder
  KL divergence
    KL divergence in TensorFlow
  Cost of a sparse autoencoder based on KL Divergence
  Complete code listing of the sparse autoencoder
  Sparse autoencoder on MNIST data
  Comparing the Sparse encoder with the Additive Gaussian Noise encoder

Summary

9. Research in Neural Networks
  Avoiding overfitting in neural networks
    Problem statement
    Solution
    Results
  Large-scale video processing with neural networks
    Resolution improvements
    Feature histogram baselines
    Quantitative results
  Named entity recognition using a twisted neural network
    Example of a named entity recognition
    Defining Twinet
    Results
  Bidirectional RNNs
    BRNN on TIMIT dataset

Summary

10. Getting started with TensorFlow
  Environment setup
  TensorFlow comparison with Numpy
  Computational graph
    Graph
    Session objects
    Variables
    Scope
    Data input
    Placeholders and feed dictionaries
  Auto differentiation
  TensorBoard
Preface

If you're aware of the buzz surrounding terms such as machine learning, artificial intelligence, or deep learning, you might know what neural networks are. Ever wondered how they help solve complex computational problems efficiently, or how to train efficient neural networks? This book will teach you both of these things, and more.

You will start by getting a quick overview of the popular TensorFlow library and see how it is used to train different neural networks. You will get a thorough understanding of the fundamentals and basic math for neural networks and why TensorFlow is a popular choice. Then, you will proceed to implement a simple feedforward neural network. Next, you will master optimization techniques and algorithms for neural networks using TensorFlow. Further, you will learn how to implement some more complex types of neural networks such as convolutional neural networks (CNNs), recurrent neural networks (RNNs), and Deep Belief Networks (DBNs). In the course of the book, you will be working on real-world datasets to get a hands-on understanding of neural network programming. You will also get to train generative models and will learn the applications of autoencoders.

By the end of this book, you will have a fair understanding of how to leverage the power of TensorFlow to train neural networks of varying complexities, without any hassle.
What this book covers

Chapter 1, *Maths for Neural Networks*, covers the basics of algebra, probability, and optimization techniques for neural networks.

Chapter 2, *Deep Feedforward Networks*, explains the basics of perceptrons, neurons, and feedforward neural networks. You will also learn about various learning techniques and mainly the core learning algorithm called backpropagation.

Chapter 3, *Optimization for Neural Networks*, covers optimization techniques that are fundamental to neural network learning.

Chapter 4, *Convolutional Neural Networks*, discusses the CNN algorithm in detail. CNNs and their application to different data types will also be covered.

Chapter 5, *Recurrent Neural Networks*, covers the RNN algorithm in detail. RNNs and their application to different data types are covered as well.

Chapter 6, *Generative Models*, explains the basics of generative models and the different approaches to generative models.

Chapter 7, *Deep Belief Networking*, covers the basics of deep belief networks, how they differ from the traditional neural networks, and their implementation.

Chapter 8, *Autoencoders*, provides an introduction to autoencoders, which have recently come to the forefront of generative modeling.

Chapter 9, *Deep Learning Research and Summary*, discusses the current and future research details on deep learning. It also points the readers to papers for reference reading.
What you need for this book

This book will guide you through the installation of all the tools that you need to follow the examples:

- Python 3.4 or above
- TensorFlow r.14 or above
Who this book is for

This book is meant for developers with a statistical background who want to work with neural networks. Though we will be using TensorFlow as the underlying library for neural networks, this book can be used as a generic resource to bridge the gap between the math and the implementation of deep learning. If you have some understanding of Tensorflow and Python and want to learn what happens at a level lower than the plain API syntax, this book is for you.
Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning. Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The next lines of code read the link and assign it to the BeautifulSoup function." A block of code is set as follows:

```python
#import packages into the project
from bs4 import BeautifulSoup
from urllib.request import urlopen
import pandas as pd
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
[default] exten
  => s,1,Dial(Zap/1130) exten
  => s,2,Voicemail(u100) exten
  => s,102,Voicemail(b100) exten
  => i,1,Voicemail(s0)
```

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

```
C:\Python34\Scripts> pip install -upgrade pip
C:\Python34\Scripts> pip install pandas
```

**New terms** and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "In order to download new modules, we will go to Files | Settings | Project Name | Project Interpreter."

*Warnings or important notes appear like this.*

*Tips and tricks appear like this.*
Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of. To send us general feedback, simply email feedback@packtpub.com, and mention the book's title in the subject of your message. If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.
Customer support

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

You can download the example code files for this book from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files emailed directly to you. You can download the code files by following these steps:

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

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

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

The code bundle for the book is also hosted on GitHub at https://github.com/PacktPublishing/Neural-Network-Programming-with-TensorFlow. We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!
Downloading the color images of this book

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

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

Piracy of copyrighted material on the internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the internet, please provide us with the location address or website name immediately so that we can pursue a remedy. Please contact us at copyright@packtpub.com with a link to the suspected pirated material. We appreciate your help in protecting our authors and our ability to bring you valuable content.
Questions

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

Neural network users need to have a fair understanding of neural network concepts, algorithms, and the underlying mathematics. Good mathematical intuition and understanding of many techniques is necessary for a solid grasp of the inner functioning of the algorithms and for getting good results. The amount of maths required and the level of maths needed to understand these techniques is multidimensional and also depends on interest. In this chapter, you will learn neural networks by understanding the maths used to solve complex computational problems. This chapter covers the basics of linear algebra, calculus, and optimization for neural networks.

The main purpose of this chapter is to set up the fundamentals of mathematics for the upcoming chapters.

Following topics will be covered in the chapter:

- Understanding linear algebra
- Understanding Calculus
- Optimization
Understanding linear algebra

Linear algebra is a key branch of mathematics. An understanding of linear algebra is crucial for deep learning, that is, neural networks. Throughout this chapter, we will go through the key and fundamental linear algebra prerequisites. Linear Algebra deals with linear systems of equations. Instead of working with scalars, we start working with matrices and vectors. Using linear algebra, we can describe complicated operations in deep learning.
Environment setup

Before we jump into the field of mathematics and its properties, it's essential for us to set up the development environment as it will provide us settings to execute the concepts we learn, meaning installing the compiler, dependencies, and IDE (Integrated Development Environment) to run our code base.
Setting up the Python environment in Pycharm

It is best to use an IDE like Pycharm to edit Python code as it provides development tools and built-in coding assistance. Code inspection makes coding and debugging faster and simpler, ensuring that you focus on the end goal of learning maths for neural networks.

The following steps show you how to set up local Python environment in Pycharm:

1. Go to Preferences and verify that the TensorFlow library is installed. If not, follow the instructions at [https://www.tensorflow.org/install/](https://www.tensorflow.org/install/) to install TensorFlow:

2. Keep the default options of TensorFlow and click on OK.
3. Finally, right-click on the source file and click on Run 'matrices':

```python
# convert matrices to tensor objects
import tensorflow as tf
import numpy as np

# Define a 2x2 matrix in 3 different ways
m1 = np.array([[1.0, 2.0], [3.0, 4.0]], dtype=np.float32)
m2 = np.array([[1.0, 2.0], [3.0, 4.0]], dtype=np.float32)
m3 = tf.constant([[1.0, 2.0], [3.0, 4.0]])

print(type(m1))
print(type(m2))
print(type(m3))

1 = tf.convert_to_tensor(m1, dtype=tf.float32)
2 = tf.convert_to_tensor(m2, dtype=tf.float32)
3 = tf.convert_to_tensor(m3, dtype=tf.float32)

print(type(1))
print(type(2))
print(type(3))```
Linear algebra structures

In the following section, we will describe the fundamental structures of linear algebra.
Scalars, vectors, and matrices

Scalars, vectors, and matrices are the fundamental objects of mathematics. Basic definitions are listed as follows:

- Scalar is represented by a single number or numerical value called magnitude.
- Vector is an array of numbers assembled in order. A unique index identifies each number. Vector represents a point in space, with each element giving the coordinate along a different axis.
- Matrices is a two-dimensional array of numbers where each number is identified using two indices \((i, j)\).
Tensors

An array of numbers with a variable number of axes is known as a tensor. For example, for three axes, it is identified using three indices \((i, j, k)\).

The following image summaries a tensor, it describes a second-order tensor object. In a three-dimensional Cartesian coordinate system, tensor components will form the matrix:

*Image reference is taken from tensor wiki* https://en.wikipedia.org/wiki/Tensor
Operations

The following topics will describe the various operations of linear algebra.
Vectors

The \textbf{Norm} function is used to get the size of the vector; the norm of a vector \( x \) measures the distance from the origin to the point \( x \). It is also known as the \( L^p \) norm, where \( p=2 \) is known as the \textbf{Euclidean norm}.

The following example shows you how to calculate the \( L^p \) norm of a given vector:

```python
import tensorflow as tf

vector = tf.constant([[4, 5, 6]], dtype=tf.float32)
eucNorm = tf.norm(vector, ord="euclidean")

with tf.Session() as sess:
    print(sess.run(eucNorm))
```

The output of the listing is 8.77496.
Matrices

A matrix is a two-dimensional array of numbers where each element is identified by two indices instead of just one. If a real matrix \( X \) has a height of \( m \) and a width of \( n \), then we say that \( X \in \mathbb{R}^{m \times n} \). Here, \( \mathbb{R} \) is a set of real numbers.

The following example shows how different matrices are converted to tensor objects:

```python
# convert matrices to tensor objects
import numpy as np
import tensorflow as tf

# create a 2x2 matrix in various forms
matrix1 = [[1.0, 2.0], [3.0, 40]]
matrix2 = np.array([[1.0, 2.0], [3.0, 40]], dtype=np.float32)
matrix3 = tf.constant([[1.0, 2.0], [3.0, 40]])

print(type(matrix1))
print(type(matrix2))
print(type(matrix3))

tensorForM1 = tf.convert_to_tensor(matrix1, dtype=tf.float32)
tensorForM2 = tf.convert_to_tensor(matrix2, dtype=tf.float32)
tensorForM3 = tf.convert_to_tensor(matrix3, dtype=tf.float32)

print(type(tensorForM1))
print(type(tensorForM2))
print(type(tensorForM3))
```

The output of the listing is shown in the following code:

- `<class 'list'>`
- `<class 'numpy.ndarray'>`
- `<class 'tensorflow.python.framework.ops.Tensor'>`
- `<class 'tensorflow.python.framework.ops.Tensor'>`
- `<class 'tensorflow.python.framework.ops.Tensor'>`
- `<class 'tensorflow.python.framework.ops.Tensor'>`
Matrix multiplication

Matrix multiplication of matrices $A$ and $B$ is a third matrix, $C$:

$$C = AB$$

The element-wise product of matrices is called a Hadamard product and is denoted as $A.B$.

The dot product of two vectors $x$ and $y$ of the same dimensionality is the matrix product $x$ transposing $y$. Matrix product $C = AB$ is like computing $C_{i,j}$ as the dot product between row $i$ of matrix $A$ and column $j$ of matrix $B$:

$$C_{i,j} = \sum_k A_{i,k}B_{k,j}$$

The following example shows the Hadamard product and dot product using tensor objects:

```python
import tensorflow as tf
mat1 = tf.constant([[4, 5, 6],[3,2,1]])
mat2 = tf.constant([[7, 8, 9],[10, 11, 12]])
# hadamard product (element wise)
mult = tf.multiply(mat1, mat2)
# dot product (no. of rows = no. of columns)
dotprod = tf.matmul(mat1, tf.transpose(mat2))
with tf.Session() as sess:
    print(sess.run(mult))
    print(sess.run(dotprod))
```

The output of the listing is shown as follows:

$$\begin{bmatrix}
28 & 40 & 54 \\
30 & 22 & 12
\end{bmatrix}
\begin{bmatrix}
122 & 167 \\
122 & 167
\end{bmatrix}$$
Trace operator

The trace operator $Tr(A)$ of matrix $A$ gives the sum of all of the diagonal entries of a matrix. The following example shows how to use a trace operator on tensor objects:

```python
import tensorflow as tf

mat = tf.constant([[0, 1, 2],
                   [3, 4, 5],
                   [6, 7, 8]], dtype=tf.float32)

# get trace ('sum of diagonal elements') of the matrix
mat = tf.trace(mat)

with tf.Session() as sess:
    print(sess.run(mat))
```

The output of the listing is 12.0.
Matrix transpose

Transposition of the matrix is the mirror image of the matrix across the main diagonal. A symmetric matrix is any matrix that is equal to its own transpose:

\[
A = \begin{bmatrix}
A_{1,1} & A_{1,2} \\
A_{2,1} & A_{2,2} \\
A_{3,1} & A_{3,2}
\end{bmatrix} \Rightarrow A^T
\]

The following example shows how to use a transpose operator on tensor objects:

```python
import tensorflow as tf
x = [[1,2,3],[4,5,6]]
x = tf.convert_to_tensor(x)
xtrans = tf.transpose(x)

y=[[1,2,3],[6,5,4],[4,5,6],[3,6,3]]
y = tf.convert_to_tensor(y)
ytrans = tf.transpose(y, perm=[0, 2, 1])

with tf.Session() as sess:
    print(sess.run(xtrans))
    print(sess.run(ytrans))
```

The output of the listing is shown as follows:

```
[[1 4]
 [2 5]
 [3 6]]
```
Matrix diagonals

Matrices that are diagonal in nature consist mostly of zeros and have non-zero entries only along the main diagonal. Not all diagonal matrices need to be square.

Using the diagonal part operation, we can get the diagonal of a given matrix, and to create a matrix with a given diagonal, we use the `diag` operation from `tensorflow`. The following example shows how to use diagonal operators on tensor objects:

```python
import tensorflow as tf

mat = tf.constant([[0, 1, 2], [3, 4, 5], [6, 7, 8]], dtype=tf.float32)
# get diagonal of the matrix
diag_mat = tf.diag_part(mat)
# create matrix with given diagonal
mat = tf.diag([1, 2, 3, 4])

with tf.Session() as sess:
    print(sess.run(diag_mat))
    print(sess.run(mat))
```

The output of this is shown as follows:

```
[[ 0.  4.  8.]]
[[[1 0 0 0] [0 2 0 0] [0 0 3 0] [0 0 0 4]]
```
Identity matrix

An identity matrix is a matrix $I$ that does not change any vector, like $V$, when multiplied by $I$.

The following example shows how to get the identity matrix for a given size:

```python
import tensorflow as tf
identity = tf.eye(3, 3)
with tf.Session() as sess:
    print(sess.run(identity))
```

The output of this is shown as follows:

```
[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]
```
Inverse matrix

The matrix inverse of $I$ is denoted as $I^{-1}$. Consider the following equation; to solve it using inverse and different values of $b$, there can be multiple solutions for $x$. Note the property:

\[
A^{-1} A = I
\]
\[
A x = b
\]
\[
A^{-1} A x = A^{-1} b
\]
\[
I_n x = A^{-1} b
\]
\[
x = A^{-1} b
\]

The following example shows how to calculate the inverse of a matrix using the `matrix_inverse` operation:

```python
import tensorflow as tf

mat = tf.constant([[2, 3, 4], [5, 6, 7], [8, 9, 10]], dtype=tf.float32)
print(mat)
inv_mat = tf.matrix_inverse(tf.transpose(mat))

with tf.Session() as sess:
    print(sess.run(inv_mat))
```
Solving linear equations

TensorFlow can solve a series of linear equations using the `solve` operation. Let's first explain this without using the library and later use the `solve` function.

A linear equation is represented as follows:

\[ ax + b = y \]
\[ y - ax = b \]
\[ y/b - a/b(x) = 1 \]

Our job is to find the values for \( a \) and \( b \) in the preceding equation, given our observed points. First, create the matrix points. The first column represents \( x \) values, while the second column represents \( y \) values.

Consider that \( X \) is the input matrix and \( A \) is the parameters that we need to learn; we set up a system like \( AX = B \), therefore, \( A = BX^{-1} \).

The following example, with code, shows how to solve the linear equation:

\[ 3x + 2y = 15 \]
\[ 4x - y = 10 \]

```python
import tensorflow as tf

# equation 1
x1 = tf.constant(3, dtype=tf.float32)
y1 = tf.constant(2, dtype=tf.float32)
point1 = tf.stack([x1, y1])

# equation 2
x2 = tf.constant(4, dtype=tf.float32)
y2 = tf.constant(-1, dtype=tf.float32)
point2 = tf.stack([x2, y2])

# solve for AX=C
X = tf.transpose(tf.stack([point1, point2]))
C = tf.ones((1, 2), dtype=tf.float32)
A = tf.matmul(C, tf.matrix_inverse(X))

with tf.Session() as sess:
    pass
```
X = sess.run(X)
print(X)

A = sess.run(A)
print(A)

b = 1 / A[0][1]
a = -b * A[0][0]
print("Hence Linear Equation is: y = {a}x + {b}".format(a=a, b=b))

The output of the listing is shown as follows:

| [[ 3. 4.][ 2. -1.]] |
| [[ 0.27272728 0.09090909]] |
Hence Linear Equation is: y = -2.9999999999999996x + 10.999999672174463

The canonical equation for a circle is \(x^2+y^2+dx+ey+f=0\); to solve this for the parameters \(d\), \(e\), and \(f\), we use TensorFlow's solve operation as follows:

```python
# canonical circle equation
# \(x^2+y^2+dx+ey+f = 0\)
# \(dx+ey+f=-(x^2+y^2)\) => AX = B
# we have to solve for \(d\), \(e\), \(f\)

points = tf.constant([[2, 1], [0, 5], [-1, 2]], dtype=tf.float64)
X = tf.constant([[2, 1, 1], [0, 5, 1], [-1, 2, 1]], dtype=tf.float64)
B = -tf.constant([[5], [25], [5]], dtype=tf.float64)
A = tf.matrix_solve(X, B)

with tf.Session() as sess:
    result = sess.run(A)
    D, E, F = result.flatten()
    print("Hence Circle Equation is: x**2 + y**2 + {D}x + {E}y + {F} = 0".format(**locals()))
```

The output of the listing is shown in the following code:

Hence Circle Equation is: x**2 + y**2 + -2.0x + -6.0y + 5.0 = 0
Singular value decomposition

When we decompose an integer into its prime factors, we can understand useful properties about the integer. Similarly, when we decompose a matrix, we can understand many functional properties that are not directly evident. There are two types of decomposition, namely eigenvalue decomposition and singular value decomposition.

All real matrices have singular value decomposition, but the same is not true for Eigenvalue decomposition. For example, if a matrix is not square, the Eigen decomposition is not defined and we must use singular value decomposition instead.

Singular Value Decomposition (SVD) in mathematical form is the product of three matrices $U$, $S$, and $V$, where $U$ is $m \times r$, $S$ is $r \times r$ and $V$ is $r \times n$:

$$A_{m \times n} = USV^{-1}$$

The following example shows SVD using a TensorFlow svd operation on textual data:

```python
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plts

path = "/neuralnetwork-programming/ch01/plots"

text = ["I", "like", "enjoy", "deep", "learning", "NLP", "flying", "."]
xMatrix = np.array([[0,2,1,0,0,0,0,0],
                    [1,0,0,0,0,0,0,0],
                    [0,1,0,0,1,0,0,0],
                    [0,0,0,1,0,0,0,1],
                    [0,1,0,0,0,0,0,1],
                    [0,0,1,0,0,0,0,1],
                    [0,0,0,0,1,1,1,0]], dtype=np.float32)

X_tensor = tf.convert_to_tensor(xMatrix, dtype=tf.float32)

# tensorflow svd
with tf.Session() as sess:
    s, U, Vh = sess.run(tf.svd(X_tensor, full_matrices=False))
```
for i in range(len(text)):
    plt.text(U[i,0], U[i,1], text[i])

plt.ylim(-0.8, 0.8)
plt.xlim(-0.8, 2.0)
plt.savefig(path + '/svd_tf.png')

# numpy svd
la = np.linalg
U, s, Vh = la.svd(xMatrix, full_matrices=False)
print(U)
print(s)
print(Vh)

# write matrices to file (understand concepts)
file = open(path + '/matx.txt', 'w')
file.write(str(U))
file.write('
')
file.write('============
')
file.write('
')
file.write(str(s))
file.close()

for i in range(len(text)):
    plt.text(U[i,0], U[i,1], text[i])

plt.ylim(-0.8, 0.8)
plt.xlim(-0.8, 2.0)
plt.savefig(path + '/svd_np.png')

The output of this is shown as follows:

```
[[ -5.24124920e-01  -5.72859168e-01   9.54463035e-02   3.83228481e-01
  -1.76963374e-01  -1.76092178e-01  -4.19185609e-01  -5.57702743e-02
  1.80462332e-01  -2.34777853e-01  1.29535481e-01  1.36813134e-01
 -2.56274015e-01   2.47912124e-01   3.54610324e-01  -7.31901303e-02
 -1.93139315e-01   3.84950386e-02  -5.00790417e-01  -4.28462476e-01
  3.47110212e-01  1.55483231e-01  -4.68663752e-01  -9.03576553e-01
 -3.05134684e-01  -2.3433599e-01  -1.91614250e-01  -3.83228481e-01
  1.27460942e-01  4.91219401e-01  2.09528904e-01  6.75753747e-01
 -1.82489842e-01  -1.61027774e-01  -3.97842437e-01  -3.83228481e-01
 -5.1293241e-01  -4.27574426e-01  4.1985699e-01  -1.8313827e-01
 -2.46898422e-01  1.57254755e-01  5.92991650e-01  -6.20076716e-01
 -3.21868137e-02  -2.31065609e-01  -2.5967063e-01  2.37966909e-01
  2.75726271  2.67824793  1.89221275  1.6103401  1.19154561  0.94833982
  0.61803401  0.56999218]]
```
```
Here is the plot for the SVD of the preceding dataset:

![Plot](image-url)
Eigenvalue decomposition

Eigen decomposition is one of the most famous decomposition techniques in which we decompose a matrix into a set of eigenvectors and eigenvalues.

For a square matrix, Eigenvector is a vector $v$ such that multiplication by $A$ alters only the scale of $v$:

$$Av = \lambda v$$

The scalar $\lambda$ is known as the eigenvalue corresponding to this eigenvector.

Eigen decomposition of $A$ is then given as follows:

$$A = V \text{diag} (\lambda) V^{-1}$$

Eigen decomposition of a matrix describes many useful details about the matrix. For example, the matrix is singular if, and only if, any of the eigenvalues are zero.
Principal Component Analysis

**Principal Component Analysis** (PCA) projects the given dataset onto a lower dimensional linear space so that the variance of the projected data is maximized. PCA requires the eigenvalues and eigenvectors of the covariance matrix, which is the product where $X$ is the data matrix.

SVD on the data matrix $X$ is given as follows:

$$X = U \Sigma V^T$$

$$XX^T = (U \Sigma V^T)(U \Sigma V^T)^T$$

$$XX^T = (U \Sigma V^T)(V \Sigma U^T)$$

$$XX^T = U \Sigma^2 U^T$$

The following example shows PCA using SVD:

```python
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import plotly.plotly as py
import plotly.graph_objs as go
import plotly.figure_factory as FF
import pandas as pd

path = "~/neuralnetwork-programming/ch01/plots"
logs = "~/neuralnetwork-programming/ch01/logs"

xMatrix = np.array([[0,2,1,0,0,0,0,0],
                    [2,0,0,1,0,1,0,0],
                    [1,0,0,0,0,1,0,0],
                    [0,1,0,0,1,0,0,0],
                    [0,0,0,1,0,0,0,1],
                    [0,1,0,0,0,0,0,1],
                    [0,0,1,0,0,0,0,1],
                    [0,0,0,0,1,1,1,0]], dtype=np.float32)

def pca(mat):
    mat = tf.constant(mat, dtype=tf.float32)
    mean = tf.reduce_mean(mat, 0)
    less = mat - mean
    s, u, v = tf.svd(less, full_matrices=True, compute_uv=True)
    s2 = s ** 2
    variance_ratio = s2 / tf.reduce_sum(s2)
```
with tf.Session() as session:
    run = session.run([variance_ratio])
    return run

if __name__ == '__main__':
    print(pca(xMatrix))

The output of the listing is shown as follows:

```
array([[ 4.15949494e-01,  2.08390564e-01,  1.90929279e-01,  8.36438537e-02,  5.55494241e-02,  2.46047471e-02,  2.09326427e-02,  3.57540098e-16],
       [ 8.36438537e-02,  5.5494241e-02,  2.09326427e-02,  3.57540098e-16]], dtype=float32)
```
Calculus

Topics in the previous sections are covered as part of standard linear algebra; something that wasn't covered is basic calculus. Despite the fact that the calculus that we use is relatively simple, the mathematical form of it may look very complex. In this section, we present some basic forms of matrix calculus with a few examples.
Gradient

Gradient for functions with respect to a real-valued matrix $A$ is defined as the matrix of partial derivatives of $A$ and is denoted as follows:

$$\nabla_A f(A) \in \mathbb{R}^{m \times n} = \begin{bmatrix}
\frac{\partial f(A)}{\partial A_{11}} & \frac{\partial f(A)}{\partial A_{12}} & \cdots & \frac{\partial f(A)}{\partial A_{1n}} \\
\frac{\partial f(A)}{\partial A_{21}} & \frac{\partial f(A)}{\partial A_{22}} & \cdots & \frac{\partial f(A)}{\partial A_{2n}} \\
\vdots & \vdots & \ddots & \vdots \\
\frac{\partial f(A)}{\partial A_{m1}} & \frac{\partial f(A)}{\partial A_{m2}} & \cdots & \frac{\partial f(A)}{\partial A_{mn}}
\end{bmatrix}$$

$$\left(\nabla_A f(A)\right)_{ij} = \frac{\partial f(A)}{\partial A_{ij}}$$

TensorFlow does not do numerical differentiation; rather, it supports automatic differentiation. By specifying operations in a TensorFlow graph, it can automatically run the chain rule through the graph and, as it knows the derivatives of each operation we specify, it can combine them automatically.

The following example shows training a network using MNIST data, the MNIST database consists of handwritten digits. It has a training set of 60,000 examples and a test set of 10,000 samples. The digits are size-normalized.

Here backpropagation is performed without any API usage and derivatives are calculated manually. We get 913 correct out of 1,000 tests. This concept will be introduced in the next chapter.

The following code snippet describes how to get the `mnist` dataset and initialize weights and biases:

```python
import tensorflow as tf

# get mnist dataset
from tensorflow.examples.tutorials.mnist import input_data
```
data = input_data.read_data_sets("MNIST_data/", one_hot=True)

# x represents image with 784 values as columns (28*28), y represents output digit
x = tf.placeholder(tf.float32, [None, 784])
y = tf.placeholder(tf.float32, [None, 10])

# initialize weights and biases [w1,b1][w2,b2]
numNeuronsInDeepLayer = 30
w1 = tf.Variable(tf.truncated_normal([784, numNeuronsInDeepLayer]))
b1 = tf.Variable(tf.truncated_normal([1, numNeuronsInDeepLayer]))
w2 = tf.Variable(tf.truncated_normal([numNeuronsInDeepLayer, 10]))
b2 = tf.Variable(tf.truncated_normal([1, 10]))

We now define a two-layered network with a nonlinear sigmoid function; a squared loss function is applied and optimized using a backward propagation algorithm, as shown in the following snippet:

```python
# non-linear sigmoid function at each neuron
def sigmoid(x):
    sigma = tf.div(tf.constant(1.0), tf.add(tf.constant(1.0), tf.exp(tf.negative(x))))
    return sigma

# starting from first layer with wx+b, then apply sigmoid to add non-linearity
z1 = tf.add(tf.matmul(x, w1), b1)
a1 = sigmoid(z1)
z2 = tf.add(tf.matmul(a1, w2), b2)
a2 = sigmoid(z2)

# calculate the loss (delta)
loss = tf.subtract(a2, y)

# derivative of the sigmoid function der(sigmoid)=sigmoid*(1-sigmoid)
def sigmaprime(x):
    return tf.multiply(sigmoid(x), tf.subtract(tf.constant(1.0), sigmoid(x)))

# backward propagation
dz2 = tf.multiply(loss, sigmaprime(z2))
db2 = dz2
dw2 = tf.matmul(tf.transpose(a1), dz2)
da1 = tf.matmul(dz2, tf.transpose(w2))
dz1 = tf.multiply(da1, sigmaprime(z1))
db1 = dz1
dw1 = tf.matmul(tf.transpose(x), dz1)

# finally update the network
eta = tf.constant(0.5)
step = [
    tf.assign(w1, tf.subtract(w1, tf.multiply(eta, dw1)))]
tf.subtract(b2, tf.multiply(eta,
    tf.reduce_mean(db2, axis=[0])))

acct_mat = tf.equal(tf.argmax(a2, 1), tf.argmax(y, 1))
acct_res = tf.reduce_sum(tf.cast(acct_mat, tf.float32))
sess = tf.InteractiveSession()
sess.run(tf.global_variables_initializer())
for i in range(10000):
    batch_xs, batch_ys = data.train.next_batch(10)
    sess.run(step, feed_dict={x: batch_xs,
                               y: batch_ys})
    if i % 1000 == 0:
        res = sess.run(acct_res, feed_dict={
            x: data.test.images[:1000],
            y: data.test.labels[:1000]})
        print(res)

The output of this is shown as follows:

Extracting MNIST_data
125.0
814.0
870.0
874.0
889.0
897.0
906.0
903.0
922.0
913.0

Now, let's use automatic differentiation with TensorFlow. The following example demonstrates the use of GradientDescentOptimizer. We get 924 correct out of 1,000 tests.

import tensorflow as tf

# get mnist dataset
from tensorflow.examples.tutorials.mnist import input_data
data = input_data.read_data_sets("MNIST_data/", one_hot=True)

# x represents image with 784 values as columns (28*28), y represents output digit
x = tf.placeholder(tf.float32, [None, 784])
y = tf.placeholder(tf.float32, [None, 10])

# initialize weights and biases [w1,b1][w2,b2]
numNeuronsInDeepLayer = 30
w1 = tf.Variable(tf.truncated_normal([784, numNeuronsInDeepLayer]))
b1 = tf.Variable(tf.truncated_normal([1, numNeuronsInDeepLayer]))
w2 = tf.Variable(tf.truncated_normal([numNeuronsInDeepLayer, 10]))
b2 = tf.Variable(tf.truncated_normal([1, 10]))

# non-linear sigmoid function at each neuron
def sigmoid(x):
    sigma = tf.div(tf.constant(1.0), tf.add(tf.constant(1.0), tf.exp(tf.negative(x))))
    return sigma

# starting from first layer with wx+b, then apply sigmoid to add non-linearity
z1 = tf.add(tf.matmul(x, w1), b1)
a1 = sigmoid(z1)
z2 = tf.add(tf.matmul(a1, w2), b2)
a2 = sigmoid(z2)

# calculate the loss (delta)
loss = tf.subtract(a2, y)

# derivative of the sigmoid function der(sigmoid)=sigmoid*(1-sigmoid)
def sigmaprime(x):
    return tf.multiply(sigmoid(x), tf.subtract(tf.constant(1.0), sigmoid(x)))

# automatic differentiation
cost = tf.multiply(loss, loss)
step = tf.train.GradientDescentOptimizer(0.1).minimize(cost)
acct_mat = tf.equal(tf.argmax(a2, 1), tf.argmax(y, 1))
acct_res = tf.reduce_sum(tf.cast(acct_mat, tf.float32))
sess = tf.InteractiveSession()
sess.run(tf.global_variables_initializer())

for i in range(10000):
    batch_xs, batch_ys = data.train.next_batch(10)
sess.run(step, feed_dict={x: batch_xs, y: batch_ys})
    if i % 1000 == 0:
        res = sess.run(acct_res, feed_dict={x: data.test.images[:1000],
                                              y: data.test.labels[:1000]})
        print(res)

The output of this is shown as follows:

96.0
777.0
862.0
870.0
889.0
901.0
911.0
905.0
914.0
924.0

The following example shows linear regression using gradient descent:

import tensorflow as tf
import numpy
import matplotlib.pyplot as plt
rndm = numpy.random
# config parameters

learningRate = 0.01
trainingEpochs = 1000
displayStep = 50

# create the training data

trainX = numpy.asarray([3.3, 4.4, 5.5, 6.71, 6.93, 4.168, 9.779, 6.182, 7.59, 2.167,
                        7.042, 10.791, 5.313, 7.997, 5.654, 9.27, 3.12])
trainY = numpy.asarray([1.7, 2.76, 2.09, 3.19, 1.694, 1.573, 3.366, 2.596, 2.53, 1.221,
                        2.827, 3.465, 1.65, 2.904, 2.42, 2.94, 1.34])

nSamples = trainX.shape[0]

# tf inputs

X = tf.placeholder("float")
Y = tf.placeholder("float")

# initialize weights and bias
W = tf.Variable(rnd.randn(), name="weight")
b = tf.Variable(rnd.randn(), name="bias")

# linear model
linearModel = tf.add(tf.multiply(X, W), b)

# mean squared error
loss = tf.reduce_sum(tf.pow(linearModel-Y, 2))/(2*nSamples)

# Gradient descent
opt = tf.train.GradientDescentOptimizer(learningRate).minimize(loss)

# initializing variables
init = tf.global_variables_initializer()

# run with tf.Session() as sess:
sess.run(init)

    # fitting the training data
    for epoch in range(trainingEpochs):
        for (x, y) in zip(trainX, trainY):
            sess.run(opt, feed_dict={X: x, Y: y})

        # print logs
        if (epoch+1) % displayStep == 0:
            c = sess.run(loss, feed_dict={X: trainX, Y: trainY})
            print("Epoch is:", '{:.4f}'.format(epoch+1), "loss=", '{:.9f}'.format(c),
                   "W=", sess.run(W), "b=", sess.run(b))

            print("optimization done...")
            trainingLoss = sess.run(loss, feed_dict={X: trainX, Y: trainY})
            print("Training loss=", trainingLoss, "W=", sess.run(W), "b=", sess.run(b),
                   \n")

    # display the plot
    plt.plot(trainX, trainY, 'ro', label='Original data')
    plt.plot(trainX, sess.run(W) * trainX + sess.run(b), label='Fitted line')
    plt.legend()
    plt.show()

# Testing example, as requested (Issue #2)
testX = numpy.asarray([6.83, 4.668, 8.9, 7.91, 5.7, 8.7, 3.1, 2.1])
testY = numpy.asarray([1.84, 2.273, 3.2, 2.831, 2.92, 3.24, 1.35, 1.03])
print("Testing... (Mean square loss Comparison)")

testing_cost = sess.run(
    tf.reduce_sum(tf.pow(linearModel - Y, 2)) / (2 * testX.shape[0]),
    feed_dict={X: testX, Y: testY})

print("Testing cost=", testing_cost)

print("Absolute mean square loss difference:", abs(trainingLoss - testing_cost))

plt.plot(testX, testY, 'bo', label='Testing data')
plt.plot(trainX, sess.run(W) * trainX + sess.run(b), label='Fitted line')
plt.legend()
plt.show()

The output of this is shown as follows:

Epoch is: 0050 loss= 0.141912043 W= 0.10565 b= 1.8382
Epoch is: 0100 loss= 0.134377643 W= 0.11413 b= 1.7772
Epoch is: 0150 loss= 0.127711013 W= 0.122106 b= 1.71982
Epoch is: 0200 loss= 0.121811897 W= 0.129609 b= 1.66585
Epoch is: 0250 loss= 0.116592340 W= 0.136666 b= 1.61508
Epoch is: 0300 loss= 0.111973859 W= 0.143304 b= 1.56733
Epoch is: 0350 loss= 0.107887231 W= 0.149547 b= 1.52241
Epoch is: 0400 loss= 0.104270900 W= 0.15542 b= 1.48017
Epoch is: 0450 loss= 0.101070963 W= 0.160945 b= 1.44043
Epoch is: 0500 loss= 0.098239250 W= 0.166141 b= 1.40305
Epoch is: 0550 loss= 0.095733419 W= 0.171029 b= 1.36789
Epoch is: 0600 loss= 0.093516059 W= 0.175626 b= 1.33481
Epoch is: 0650 loss= 0.091553882 W= 0.179951 b= 1.3037
Epoch is: 0700 loss= 0.089817807 W= 0.184018 b= 1.27445
Epoch is: 0750 loss= 0.088281371 W= 0.187843 b= 1.24692
Epoch is: 0800 loss= 0.086921677 W= 0.191442 b= 1.22104
Epoch is: 0850 loss= 0.085718453 W= 0.194827 b= 1.19669
Epoch is: 0900 loss= 0.084653646 W= 0.198011 b= 1.17378
Epoch is: 0950 loss= 0.083711281 W= 0.201065 b= 1.15224
Epoch is: 1000 loss= 0.082877308 W= 0.203822 b= 1.13190

optimization done...
Training loss= 0.0828773 W= 0.203822 b= 1.13198
Testing... (Mean square loss Comparison)
Testing cost= 0.0957726
Absolute mean square loss difference: 0.0128952

The plots are as follows:
The following image shows the fitted line on testing data using the model:
**Hessian**

Gradient is the first derivative for functions of vectors, whereas hessian is the second derivative. We will go through the notation now:

\[
\nabla^2 f(x) \in \mathbb{R}^{n \times n} = \begin{bmatrix}
\frac{\partial^2 f(x)}{\partial x_1^2} & \frac{\partial^2 f(x)}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f(x)}{\partial x_1 \partial x_n} \\
\frac{\partial^2 f(x)}{\partial x_2 \partial x_1} & \frac{\partial^2 f(x)}{\partial x_2^2} & \cdots & \frac{\partial^2 f(x)}{\partial x_2 \partial x_n} \\
\vdots & \vdots & \ddots & \vdots \\
\frac{\partial^2 f(x)}{\partial x_n \partial x_1} & \frac{\partial^2 f(x)}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f(x)}{\partial x_n^2}
\end{bmatrix}
\]

Similar to the gradient, the hessian is defined only when \( f(x) \) is real-valued.

*The algebraic function used* is \( q(x) = x_1^2 + 2x_1x_2 + 3x_2^2 + 4x_1 + 5x_2 + 6 \).

The following example shows the hessian implementation using TensorFlow:

```python
import tensorflow as tf
import numpy as np

X = tf.Variable(np.random.random_sample(), dtype=tf.float32)
y = tf.Variable(np.random.random_sample(), dtype=tf.float32)

def createCons(x):
    return tf.constant(x, dtype=tf.float32)

function = tf.pow(X, createCons(2)) + createCons(2) * X * y + createCons(3) * tf.pow(y, createCons(2)) + createCons(4) * X + createCons(5) * y + createCons(6)

# compute hessian
def hessian(func, varbles):
    matrix = []
    for v_1 in varbles:
        tmp = []
        for v_2 in varbles:
            # calculate derivative twice, first w.r.t v2 and then w.r.t v1
            tmp.append(tf.gradients(tf.gradients(func, v_2)[0], v_1)[0])
        tmp = [createCons(0) if t == None else t for t in tmp]
        tmp = tf.stack(tmp)
        matrix.append(tmp)
        matrix = tf.stack(matrix)
    return matrix

hessian = hessian(function, [X, y])

sess = tf.Session()
sess.run(tf.initialize_all_variables())
print(sess.run(hessian))
```
The output of this is shown as follows:

| [[ 2.  2. ] [ 2.  6.]] |
Determinant

Determinant shows us information about the matrix that is helpful in linear equations and also helps in finding the inverse of a matrix.

For a given matrix \( X \), the determinant is shown as follows:

\[
X = \begin{bmatrix}
a & b & c \\
d & e & f \\
g & h & i \\
\end{bmatrix}
\]

\[
\text{det}(X) = a(ef - fh) - b(di - fg) - c(dh - eg)
\]

The following example shows how to get a determinant using TensorFlow:

```python
import tensorflow as tf
import numpy as np

x = np.array([[10.0, 15.0, 20.0], [0.0, 1.0, 5.0], [3.0, 5.0, 7.0]], dtype=np.float32)
det = tf.matrix_determinant(x)

with tf.Session() as sess:
    print(sess.run(det))
```

The output of this is shown as follows:

```
-15.0
```
Optimization

As part of deep learning, we mostly would like to optimize the value of a function that either minimizes or maximizes $f(x)$ with respect to $x$. A few examples of optimization problems are least-squares, logistic regression, and support vector machines. Many of these techniques will get examined in detail in later chapters.
Optimizers

We will study AdamOptimizer here; TensorFlow AdamOptimizer uses Kingma and Ba's Adam algorithm to manage the learning rate. Adam has many advantages over the simple GradientDescentOptimizer. The first is that it uses moving averages of the parameters, which enables Adam to use a larger step size, and it will converge to this step size without any fine-tuning.

The disadvantage of Adam is that it requires more computation to be performed for each parameter in each training step. GradientDescentOptimizer can be used as well, but it would require more hyperparameter tuning before it would converge as quickly. The following example shows how to use AdamOptimizer:

- tf.train.Optimizer creates an optimizer
- tf.train.Optimizer.minimize(loss, var_list) adds the optimization operation to the computation graph

Here, automatic differentiation computes gradients without user input:

```python
import numpy as np
import seaborn
import matplotlib.pyplot as plt
import tensorflow as tf

# input dataset
xData = np.arange(100, step=.1)
yData = xData + 20 * np.sin(xData/10)

# scatter plot for input data
plt.scatter(xData, yData)
plt.show()

# defining data size and batch size
nSamples = 1000
batchSize = 100

# resize
xData = np.reshape(xData, (nSamples, 1))
yData = np.reshape(yData, (nSamples, 1))

# input placeholders
x = tf.placeholder(tf.float32, shape=(batchSize, 1))
y = tf.placeholder(tf.float32, shape=(batchSize, 1))
```
# init weight and bias
with tf.variable_scope("linearRegression"):  
    W = tf.get_variable("weights", (1, 1), 
    initializer = tf.random_normal_initializer())  
    b = tf.get_variable("bias", (1,), initializer = tf.constant_initializer(0.0))  

    y_pred = tf.matmul(x, W) + b  
    loss = tf.reduce_sum((y - y_pred)**2 / nSamples)

# optimizer
opt = tf.train.AdamOptimizer().minimize(loss)
with tf.Session() as sess:  
    sess.run(tf.global_variables_initializer())

    # gradient descent loop for 500 steps
    for _ in range(500):  
        # random minibatch
        indices = np.random.choice(nSamples, batchSize)

        X_batch, y_batch = xData[indices], yData[indices]

        # gradient descent step
        _, loss_val = sess.run([opt, loss], feed_dict={x: X_batch, y: y_batch})

Here is the scatter plot for the dataset:

![Scatter plot for the dataset](image1)

This is the plot of the learned model on the data:

![Learned model plot](image2)
Summary

In this chapter, we've introduced the mathematical concepts that are key to the understanding of neural networks and reviewed some maths associated with tensors. We also demonstrated how to perform mathematical operations within TensorFlow. We will repeatedly be applying these concepts in the following chapters.