

**Deep Learning Algorithms:**  
**Transformers, gans, encoders, cnns, rnns, and more**

by  
Ricardo A. Calix

**Galactic Backwater Media ©**  
**[www.galacticbackwater.com](http://www.galacticbackwater.com)**

## **DEDICATION**

To my family

## **ACKNOWLEDGEMENTS**

Thanks to my students and colleagues at Purdue University Northwest who helped me to develop a Big Data course which is the basis for this book. I especially want to thank Professor Keyuan Jiang for his support of my teaching and research endeavors. I want to thank Ravish Gupta and Matrika Gupta for their questions and comments that helped to improve the material discussed in this book. I would also like to thank my Fall 2018, 2019 deep learning students for their help in making this book a better instrument for learning.

## TABLE OF CONTENTS

DEDICATION .....	iii
ACKNOWLEDGEMENTS .....	iv
PREFACE (First edition) .....	x
PREFACE (Current edition) .....	xii
CHAPTER 1: INTRODUCTION .....	1
1.1 Setting up your Environment .....	3
1.1.1 Setting up your Environment with a VM or laptop .....	3
1.1.2 Setting up your Environment with a Physical Box .....	4
1.2 Background .....	5
1.3 Numpy Arrays, Tensors, and Linear Algebra .....	5
1.3.1 Numpy Arrays .....	6
1.3.2 Tensor Operations with Tensorflow .....	42
1.4 Summary .....	59
CHAPTER 2: TRADITIONAL MACHINE LEARNING .....	60
2.1 Code Issues .....	63
2.2 Performance Evaluation .....	67
2.3 Optimization .....	70
2.4 Logistic Regression .....	73
2.5 Neural Networks .....	74
2.6 KNN .....	75
2.7 Support Vector Machines (SVM) .....	76
2.8 The t-test for Machine Learning .....	79
2.9 Summary .....	80

CHAPTER 3: DATA LOADING AND PREPROCESSING.....	81
3.1 Loading the Data.....	81
3.2 Data and Feature Pre-Processing .....	83
3.3 One Hot Encoding.....	85
3.4 Features .....	87
3.4.1 Features from Text.....	88
3.4.2 Features from Images.....	89
3.5 Corpora .....	90
3.6 Summary .....	94
CHAPTER 4: DEEP LEARNING .....	95
4.1 Things to know about the code .....	95
4.2 Getting Started with Deep Leaning.....	99
4.3 Deep Leaning Definition.....	102
4.4 Tensorflow Basics.....	102
4.5 Loading your Data into your Tensorflow code.....	104
4.6 Linear Regression in Tensorflow .....	109
4.7 Linear Regression to Logistic Regression and Neural Nets.....	123
4.8 Logistic Regression in Tensorflow .....	133
4.8.1 A Simple Logistic Regression Example .....	146
4.9 Layers of the Neural Network in Tensorflow .....	152
4.10 Going Deep: An N layer Neural Network in Tensorflow .....	156
4.11 Evaluating your Deep Neural Network in Tensorflow .....	160
4.12 Summary .....	163
CHAPTER 5: SEMANTIC VECTOR SPACES .....	164
5.1 Vector Space Model.....	165

5.2 Latent Semantic Analysis (LSA) .....	166
5.3 Word Embeddings and Neural Nets.....	169
5.4 Word2vec code .....	174
5.5 Knowledge Enhanced Vector Spaces .....	192
5.6 Autoencoders .....	196
5.7 Deep Gramulator.....	204
5.8 Summary .....	210
CHAPTER 6: CONVOLUTIONAL NEURAL NETWORKS.....	212
6.1 Loading the Mnist Data .....	212
6.2 Convolution and CNNs Defined.....	213
6.3 Architecture for a Convolutional Neural Network.....	214
6.4 Code for a Convolutional Neural Network.....	218
6.4.1 Convolution and Maxpool Operations.....	219
6.4.2 Layer Definitions .....	220
6.4.3 Defining the CNN Architecture .....	223
6.4.4 Definition of the Loss, Optimization, and Evaluation Functions.....	225
6.4.5 The Core of the Deep Neural Network .....	226
6.4.6 Initializing Variables and the Session.....	227
6.4.7 The Main Loop .....	228
6.4.8 Classification Results for the CNN.....	229
6.5 CNNs for RGB Data .....	232
6.6 Summary .....	248
CHAPTER 7: RECURRENT NEURAL NETWORKS .....	249
7.1 Using a Recurrent Neural Network (RNN) on MNIST .....	249
7.1.1 Processing Parameters.....	250

7.1.2 Algorithm and Architecture Parameters.....	250
7.1.3 Import the Data .....	253
7.1.4 Calling the Core Functions .....	253
7.1.5 The Loss Function.....	254
7.1.6 The Optimizer Function.....	255
7.1.7 The Evaluation Function.....	255
7.1.8 The RNN Function and Architecture .....	255
7.1.9 Creating the Labels and Data Tensors .....	259
7.1.10 Initializing the Session and Variables .....	260
7.1.11 The Main Loop.....	260
7.1.12 Results.....	262
7.2 RNNs for NLP .....	263
7.3 LSTM.....	272
7.4 Transformers.....	273
7.5 Summary .....	273
CHAPTER 8: GENERATIVE ADVERSARIAL NETWORKS.....	274
8.1 GAN code .....	276
8.2 Generating MNIST digits with GANs .....	283
8.3 Some Uses of GANs .....	295
8.4 Summary .....	296
CHAPTER 9: REINFORCEMENT LEARNING .....	297
9.1 Q-Learning using a Table .....	300
9.2 Q-Learning using a Neural Network.....	305
9.3 Q-Learning using a Neural Network and Randomness.....	317
9.4 Summary .....	320



CHAPTER 10: TRANSFORMERS.....	321
10.1 Encoder Decoder with Multi-Head Attention.....	321
10.1.1 The Main Ideas of the Transformer .....	325
10.1.2 Code .....	332
10.2 Summary.....	397
CHAPTER 11: CONCLUSIONS AND FINAL THOUGHTS .....	398
11.1 Benchmarking Tensorflow.....	398
11.2 Conclusions.....	399
11.3 Summary.....	402
REFERENCES .....	403
APPENDIX A: FULL DEEP LEARNING CODE.....	407
APPENDIX B: USEFUL TENSORFLOW FUNCTIONS.....	408
VITA.....	412

## **PREFACE (FIRST EDITION)**

Ever since 2007 with the explosion in the use of parallel based hardware, the field of machine learning has become more exciting and more promising. It seems that the dream of true AI is finally just around the corner. Certainly, there are many companies that are starting to rely heavily on AI for their products. These include companies in search like Facebook, Google, as well as retailers and multimedia companies like Amazon and Netflix. But more recently many others in the health-care and cyber security industries are also interested in what AI and machine learning can do for them.

Some of these machine learning technologies such as Tensorflow (which came about around 2015) are new and not widely understood. In this book I hope to provide basic discussions of machine learning and in particular deep learning to help readers to quickly get started in using these technologies and methodologies.

The book is not a comprehensive survey on deep learning. There are many topics I do not cover here as too much material can be overwhelming to the un-initiated. There are many good books that cover all the theory in depth and I will mention some of them in the book (e.g. Goodfellow et. al 2016). Instead, the goal is to help people new to deep learning to quickly get started with these concepts using python and Tensorflow. Therefore, a lot of detail is spent on helping the reader to write his or her first deep network classifier. Additionally, I will try to connect several elements in machine learning which I think are related and are very important for data analysis and automatic classification.

In general, I prefer python and I will try to present all examples using this great language. I will also use the more common libraries and the Linux or mac development environment. Many people use SKlearn and I have therefore tried to

use this library in the Tensorflow examples so that the focus is mainly on creating the deep layer network architectures.

In this book I cover the following topics: linear regression, logistic regression, neural networks, deep neural networks, word embeddings and word2vec, convolutional neural networks, recurrent neural networks, generative adversarial networks, and reinforcement learning.

## **PREFACE (CURRENT EDITION)**

This was going to be the second edition of my original book “Getting Started with Deep learning”. However, so much has changed about the book and the field since its initial publishing that I decided that a new name was more appropriate.

It is now 2020 and Deep Learning is still going strong. In fact, I believe it is accelerating in its evolution. The techniques are widely used by companies now and the algorithms are starting to do things that are truly amazing. As is necessary with progress, the algorithms are also more complicated, with deeper and more resource intensive networks. This is best exemplified by one of the newest deep learning algorithms: The Transformer. Transformers are, for me, the first algorithm I was not able to run on a laptop. They truly require a machine learning “war machine”. Lots of GPU power and memory, etc. The algorithms are much more complicated too. A little bit too much in fact and the programming languages are starting to abstract too much of the code. Something I am not crazy about as I like writing the code from scratch and I never use a deep learning algorithm until I understand every detail about it. My quest for understanding always makes me gravitate away from abstracting libraries and over simplifications. As such, I have great admiration for the computational static graph and the Tensorflow low level API. I feel that I can only understand a deep learning algorithm when I implement it in the low level API with a static graph. As such, all algorithms discussed in this book are implemented in this way. Therefore, the goal of this book is to learn, and to better understand, how to write deep learning algorithms from scratch (as much as is possible using Tensorflow) using the Tensorflow low level API and the static graph. This is a book for everyone from those starting in deep learning to those with more advanced knowledge. The book starts with basic linear regression and builds on every chapter while progressing to the more advanced algorithms like

CNNs, GANs, and Transformers, to name a few. I hope you enjoy reading the book. I sure have enjoyed writing it.

## CHAPTER 1: INTRODUCTION

This book is very sequential and I recommend that you start from chapter 1 and read sequentially, especially if you are new to deep learning. If you are experienced in deep learning, then you can probably skip around the chapters. In this book I will discuss some aspects of the theory of deep learning while providing as much intuition as possible. I will use Linux or mac, python, Sklearn, and the Tensorflow low level API with static computational graphs.

Chapter 1 will briefly discuss some of the background required or recommended for deep learning. In chapter 2, I will address some of the general machine learning topics. This will be a brief review of some of the traditional (non-deep learning based) machine learning algorithms for context. I will introduce the Sklearn library to provide code examples on how to use the different classifiers and other tools. I will quickly move through logistic regression to arrive at other algorithms. There are many books on the theory of these traditional machine learning algorithms (Witten and Frank, etc.); so, instead of discussing the theory, I will concentrate on the practical aspects of using the machine learning algorithms in Python. More importantly, I will show how Sklearn can later be used with Tensorflow for deep learning. Finally, I will discuss evaluation tasks and performance metrics.

In chapter 3, I will address the very important issue of the data and data pre-processing. To me and most practitioners, data collection and data processing are some of the most important issues in machine learning/deep learning. There are many issues that must be addressed when dealing with data. These include: getting the data, cleaning data, pre-processing data, building a corpus and annotating it, performing inter-annotator agreement analysis, etc. Some of these issues will be discussed in chapter 3. In chapter 4, I will introduce the topic of deep learning for the first time and, in particular, how to create deep neural networks. My goal here

is to help students and practitioners alike to better understand the computational graph. In particular, I want the reader to be able to get data and pre-process it in the appropriate format for deep learning. This chapter also covers how to load data to your program, and how to perform simple classification tasks with plain vanilla deep neural networks of 2 or more hidden layers.

In chapter 5, I will discuss semantic vector spaces. I will discuss the vector space model and then address latent semantic analysis (LSA). I will also address the newer vector space based models such as Word2vec. Again, the focus will be on the intuition. Many companies today base their products on these very simple but powerful semantic spaces. There are many different types of vector space based models that have been used to represent data. However, the exciting aspect is that now deep learning methods allow you to process massive amounts of data so it can be represented by these vector spaces. In chapter 6, I will discuss Convolutional Neural Networks (CNNs). This chapter covers one of the most powerful techniques in deep learning. I will discuss how to implement simple CNNs with 2 convolutional layers for 2D image processing, as well as more complex CNNs for RGB data. In chapter 7, I will discuss Recurrent Neural Networks (RNNs). This chapter covers one of the most powerful techniques in deep learning for natural language processing. I will discuss how to implement simple RNNs in this chapter and how they can be applied to natural language processing (NLP). In chapter 8, I will discuss Generative Adversarial Networks (GANs). This chapter covers one of the most interesting new techniques in deep learning. I will discuss aspects of GANs as well as some of the code. In chapter 9, I will discuss Reinforcement Learning. This chapter covers an amazing technique in machine learning now enhanced by deep learning that has been applied extensively to AI and games. I will discuss aspects of Q-Learning as well as some of the code. This chapter will rely on Python Gym which is a set of games that can be implemented using the

Python language. In this chapter, I also show how to create your own environment to interface Q learning with game engines or simulators. Chapter 10 will introduce you to the very broad topic of Transformers and the mechanism of Attention. This is one of the latest developments in deep learning as of 2017. Transformers are very powerful and offer a lot of promise for NLP. Finally, in the final chapter, I will discuss some final loose ends as well as present my final thoughts and conclusions.

My goal for this book is to present intuition over equations and to use code to convey how things work. This is how I like to learn. To quote Richard Feynman: “What I cannot create, I do not understand”. Some equations can still be useful, however, so I will use them when appropriate. Additionally, all the code used in this book can be obtained from GitHub at <https://github.com/rcalix1> and any other complimentary materials about the book such as some of the figures in color can be obtained from the book website at [www.galacticbackwater.com](http://www.galacticbackwater.com).

## **1.1 Setting up your Environment**

In this section I will discuss how to set up your environment to get started with deep learning programming. The two main environments I will discuss are using a linux virtual machine and building the hardware with a CPU and GPU.

### **1.1.1 Setting up your Environment with a VM or laptop**

The code discussed in this book will work on a linux virtual machine or standard physical Linux or Mac operating systems. You can use the latest version of Linux to run your code. I used Ubuntu 16.04 to 18.04 (64 bit) and Mac to test the examples in this book. There may be some issues with memory if you do not allocate enough memory to your virtualization environment when using a VM. In



my case, I have used VMware. The instructions for installing the software on a VM are the same as for installing the software on a physical Linux box. You can get the detailed instructions to install Tensorflow from ([www.tensorflow.org](http://www.tensorflow.org)) and for Sklearn from (<http://scikit-learn.org/>). I suggest you use Conda for library and package installation. Some algorithms are not really meant to run on a VM. The virtual machine or even your laptop are for writing the code, debugging, etc. To get the best out of deep learning using deep architectures and massive amounts of data you will need to use a GPU enabled machine. I suggest using the cloud or building your own rig.

### **1.1.2 Setting up your Environment with a Physical Box**

If you want to build a physical box, I will give you some of the specifications of a machine I have used for this code. In general, the more powerful the machine the better it will be at processing large amounts of data. Here are the specifications I have used:

- A GPU GeForce RTX2080 Ti or better (P100)
- A CPU such as the AMD 12 CORE
- Power supply EVGA SuperNOVA 1600 W P2 220
- Motherboard for multiple GPU and CPU
- More than 64 GB of RAM (DDR4)
- SSD hard drive 2 TB
- A case with cooling

The total cost for 1 device with just 1 CPU and 1 GPU may be between \$2,500 and \$3,500.

## 1.2 Background

So, what background should you have to get the best out of deep learning? This is a question that often comes up and it is very important. Depending on what you will do in deep learning I would recommend a course in statistics that covers probability and linear regression, programming up to data structures with python and C/C++, a course in optimization, and a course on linear algebra. Sometimes a course in computer graphics using something like OpenGL where you have to manipulate meshes via linear algebra operation can also be very helpful to visualize and better understand matrices and vectors. Of all of these, matrix and vector operations from linear algebra are absolutely essential for writing algorithms from scratch. The name Tensor in Tensorflow means matrix of any dimension. So, even the name reminds you of the importance of linear algebra for deep learning. While this is not a linear algebra book, I will provide in the next section a quick introduction to the topic and some useful code examples that might help you later as you progress through the different deep learning algorithms. For a better treatment of linear algebra and optimization I recommend “Linear Algebra and Optimization for Machine Learning: A Textbook” by Charu C. Aggarwal.

## 1.3 Numpy Arrays, Tensors, and Linear Algebra

In this section, I will present some of the most useful techniques used in this book, or in the deep learning field as a whole, for dealing with data. In general, we want to be very efficient in our processing of the data so we do not use python lists and “for” loops. Instead, we use numpy arrays and tensors. These are treated as vectors and matrices in linear algebra. And more generally are referred to as tensors. The advantage of this approach is that we can perform a lot of linear algebra based

math operations like the matrix multiplication, the transpose, etc on our data using python's numpy or tensorflow libraries.

The best way to learn about this approach is to do a series of exercises. I will start with examples of numpy array operations and then move to tensor operations with tensorflow. Along the way I will point out some terminology or concepts from linear algebra.

### 1.3.1 Numpy Arrays

Okay, so let's get started. First we need to import the numpy and tensorflow libraries.

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

Once we import the libraries we can start writing some code. Let's begin with some warm up examples using numpy.

To declare an array in numpy we can write

```
a = np.array([4,5,2,6,8])
print(a)
```

This results in

```
[4 5 2 6 8]
```

We can also declare a numpy array with different data types. For instance, an array as float.

```
a = np.array([1, 3, 2, 5], dtype='float32' )  
print(a)
```

This gives us

```
[1. 3. 2. 5.]
```

A numpy matrix (2D np array) can be declared like so

```
list_of_lists = [[1, 2, 3], [4, 4, 5], [6, 2, 11]]  
b = np.array(list_of_lists)  
print(b)
```

This gives us a 3x3 matrix

```
[[ 1  2  3]  
 [ 4  4  5]  
 [ 6  2 11]]
```

Numpy has special functions to initialize a matrix. For example

```
b = np.zeros(10, dtype=int)
print(b)
```

The previous code generates a numpy array of size 10 made up of all zeros.

```
[0 0 0 0 0 0 0 0 0 0]
```

We can also create a matrix of all 1s with the following function:

```
b = np.ones((4, 6), dtype=float)
print(b)
```

which produces a 4x6 matrix of type float

```
[[1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1.]]
```

For a matrix made up of just one value we can write

```
b = np.full((3, 3), 42)
print(b)
```

This generates a 3x3 matrix where all values are 42

```
[[42 42 42]
 [42 42 42]
 [42 42 42]]
```

Sometimes we need numpy arrays with different data in them. Numpy has quick ways of creating arrays with data in them. The **np.arange** function is useful for this. For example

```
b = np.arange(1, 30, 3)
print(b)
```

The previous code gives us a numpy array with 10 values in the range from 1 to 28 with a step of size 3.

```
[ 1  4  7 10 13 16 19 22 25 28]
```

The function `linspace` is another way of generating numpy arrays with data in them. Here we generate 20 data points from 0 to 1 spaced by a step size of around 0.05.

```
b = np.linspace(0, 1, 20)
print(b)
```

The output looks like this

```
[0.      0.05263158 0.10526316 0.15789474 0.21052632 0.26315789
 0.31578947 0.36842105 0.42105263 0.47368421 0.52631579 0.57894737
 0.63157895 0.68421053 0.73684211 0.78947368 0.84210526 0.89473684
 0.94736842 1.      ]
```

Yet another useful function is `np.random.random`

```
b = np.random.random((4, 4))
print(b)
```

With this function we can generate random data like the following. Here we have a 4x4 matrix with random values.

```
[[0.52467069 0.68216617 0.79782109 0.33720887]
 [0.67956722 0.04082517 0.31311017 0.72985649]
 [0.64533659 0.83448976 0.37986602 0.60524177]
 [0.98868748 0.36999339 0.33000013 0.04157917]]
```

For random data with a mean of 0 and standard deviation of 1 we can write

```
## mean 0 and standard deviation 1

b = np.random.normal(0, 1, (4,4))
print(b)
```

And the data looks like this where we get a 4x4 matrix of random data with mean 0 and standard deviation 1.

```
[[ 0.82064949 -0.95219825 -1.27123377 -1.01187383]
 [ 0.44419588  0.17695603 -0.75775624 -0.14476445]
 [ 0.59233303  1.27530445  0.77260354 -0.80240966]
 [-0.58009786 -1.04106833 -1.27650071  0.28198804]]
```

Sometimes when doing linear algebra operations you need special matrices like the identity matrix. You can generate this matrix with numpy using the following code:

```
## identity matrix

b = np.eye(5)
print(b)
```

The previous code generates a 5x5 identity matrix like the following

```
[[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]]
```

Okay. So, now let's practice generating some matrices and looking at their dimensions.



```
b1 = np.random.randint(20, size=6)
b2 = np.random.randint(20, size=(3,4))
b3 = np.random.randint(20, size=(2,4,6))

print(b2)
print(b3)
print("b2 dims ", b2.ndim)
print("b3 shape ", b3.shape)
print("b2 size ", b2.size)
print("data type of b3 ", b3.dtype)
```

The previous code generates the following matrices with dimensions:

A matrix of size [3, 3]

```
[[ 7 13 16 13]
 [11  6 17 11]
 [ 0 12  6  4]]
```

A matrix of size [2, 4, 6]

```
[[[12 10 15  3 14  4]
   [ 9  7  4  0 16 10]
   [11 16  9  0  5 12]
   [ 4 12  6  9  3  5]]

 [[ 7 17  5 18  0 15]
   [ 9  3  4  4  7  0]
   [15  1  4 12 10 17]
   [ 9 14  1 14 19  8]]]
```

We can use the methods `ndim`, `shape`, `size`, `dtype` to look at the properties of our matrices like so

```
b2 dims 2
b3 shape (2, 4, 6)
b2 size 12
data type of b3 int64
```

Knowing how to index a numpy array is very important. Here we have an example of how to extract values by index.

```
## indexing

a = np.array([1, 3, 2, 5], dtype='float32' )
print(a)
print("first ", a[0])
print("third ", a[2])
print("last ", a[-1])
print("before last ", a[-2])
```

The results of the previous indexing examples are as follows:

```
[1. 3. 2. 5.]
First value 1.0
Third value 2.0
Last value 5.0
before last value 2.0
```

We can also do indexing on 2D matrices as follows:

```

## indexing

a = np.array([[1, 2, 3, 4],
              [5, 6, 7, 8],
              [9, 10, 11, 12]])

print(a)
print("first ", a[0,0])

print("last ", a[2, -1])

```

The previous indexing examples give us the following results for the give matrix

```

[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]

```

We get the forst and las values

```

first 1
last 12

```

One important concept when dealing with numpy arrays or tensors is slicing. Slicing helps us to extract slices of data from a matrix like extracting 2 middle column vectors in a matrix. The following are some examples of slicing

```

## slicing

x = np.arange(15)
print(x)
print("first 4 elemets ", x[:4])
print("all after 3 ", x[3:])
print("even indeces ", x[::2]) ## starts at 0 ## 2 is the step size
print("uneven indeces ", x[1::2]) ## starts at 1
print("reverse ", x[::-1]) ## step value is negative starts at last element

```

Given a numpy array “x” with 15 values

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]
```

The previous code gives us the following slicing results

```
first 4 elemets  [0 1 2 3]
all after 3      [ 3  4  5  6  7  8  9 10 11 12 13 14]
even indeces    [ 0  2  4  6  8 10 12 14]
uneven indeces  [ 1  3  5  7  9 11 13]
reverse         [14 13 12 11 10  9  8  7  6  5  4  3  2  1  0]
```

Slicing a sub matrix from a larger matrix can be done as follows

```
## slicing

a = np.array([[1, 2, 3, 4],
              [5, 6, 7, 8],
              [9, 10, 11, 12]])

print(a)
print(a[:2,:2])
```

Here, from a 3x4 matrix

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

we get a 2x2 matrix after slicing

```
[[1 2]
 [5 6]]
```

Whenever we want the last value of a matrix, vector, or tensor, we can just use the -1 index. For example,

```

a = np.array([[1, 2, 3, 4],
              [5, 6, 7, 8],
              [9, 10, 11, 12]])

print(a)
print("a[:-1,:-1]")
print(a[:-1,:-1])

```

From the previous code, we have a 3x4 matrix

```

[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]

```

After slicing it with the following indexing scheme

```
a[:-1,:-1]
```

we get a 2x3 matrix by not including the last column and last row

```

[[1 2 3]
 [5 6 7]]

```

Another example of slicing with the -1 index is

```

a = np.array([[1, 2, 3, 4],
              [5, 6, 7, 8],
              [9, 10, 11, 12]])

print(a)
print("a[:-1,:-2]")
print(a[:-1,:-2])

```

From a matrix like this

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

We slice with

```
a[:-1,:-2]
```

and get a 2x2 matrix

```
[[1 2]
 [5 6]]
```

Sometimes, especially with Transformers in language translation, we want to shift a sentence or line of data to the left or right to better align it or disalign it with another sentence or line. This can be done as follows

```
## shifting

x = np.arange(15)
print(x)
print("shift right ", x[:-1] )
print("shift left  ", x[1:] )
```

Notice that here we just combine slicing with the -1 (last) index to create shifting.

The result is as follows

Given an input

```
input      [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]
```

after shifting we get

```
shift right [ 0 1 2 3 4 5 6 7 8 9 10 11 12 13]
shift left  [ 1 2 3 4 5 6 7 8 9 10 11 12 13 14]
```

Slicing is very useful to extract column vectors, for example.

```
a = np.array([[1, 2, 3, 4],
              [5, 6, 7, 8],
              [9, 10, 11, 12]])

print(a)
print("column 1 ")
print(a[:, 1])
```

With the previous code, we can slice the second column (column 1)

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

And get

```
[ 2  6 10]
```

We can also extract row vectors from a matrix

```
a = np.array([[1, 2, 3, 4],
              [5, 6, 7, 8],
              [9, 10, 11, 12]])

print(a)
print("row 1 ")
print(a[1, :])
```

For a 3x4 matrix

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

We can extract row 1

```
[5 6 7 8]
```

By slicing with

```
a[1, :]
```

Slicing in numpy does not copy to a new array but instead it still modifies the original array. To copy and create a new matrix you may want to use **.copy()** like so

```
a = np.array([[1, 2, 3, 4],
              [5, 6, 7, 8],
              [9, 10, 11, 12]])

new_a = a[:2, :2]
print(a)
print(new_a)
new_a[0,0] = 42
print(a)
new_a2 = a[:2, :2].copy()
new_a2[0,0] = 17
print(a)
```



After slicing the matrix of size 3x4 with `a[:2, :2]`

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

We get

```
[[1 2]
 [5 6]]
```

We then assign a new value to this new sliced matrix with the following expression

```
new_a[0,0] = 42
```

```
[[42  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

Now if we print the original matrix “a”, notice that the the first value has also been changed to 42.

```
[[42  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

Once you start getting into complex deep learning algorithms like CNNs, you will start using reshape operations. Here is an example of how to use reshape.

```
a = np.arange(1, 10)
b = a.reshape( (3,3) )
print(a)
print(b)
```

Notice here how we reshape from a vector of size [1, 10] to a matrix of size 3x3 as can be seen below

```
[1 2 3 4 5 6 7 8 9]
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

Besides using the reshaping operation, we can sometimes use **np.newaxis**. The **np.newaxis** function is critical when we wish to make a row vector into a column vector. The **np.newaxis** function is used extensively in numpy and tensorflow for broadcasting operation. An example of creating a new axis can be seen in the code below

```
v = np.array( [1, 2, 3, 4, 5] )
v1 = np.array( [5, 5, 5, 5, 5] )

m = np.array([[1, 2, 3, 4],
              [5, 6, 7, 8],
              [9, 10, 11, 12]])

print("reshape as row vector with reshape ", v.reshape( (1,5) ))
print("reshape as row vector with newaxis ", v1[np.newaxis, :])
print("reshape as column vector with newaxis ")
print( v1[:, np.newaxis] )
print(" reshape matrix m[:, np.newaxis, np.newaxis, :] with newaxis ")
print( m[:, np.newaxis, np.newaxis, :] )
```

The previous code example results in the following output

```
[
  [
    [ 1 2 3 4]
  ]
  [
    [ 5 6 7 8]
  ]
  [
    [ 9 10 11 12]
  ]
]
```

Notice here the double set of square brackets

```
reshape as row vector with reshape [[1 2 3 4 5]]
reshape as row vector with newaxis [[5 5 5 5 5]]
```

To make a column vector we use `[:, np.newaxis]`

```
reshape as column vector with newaxis
[[5]
 [5]
 [5]
 [5]
 [5]]
```

In this operation

```
m[:, np.newaxis, np.newaxis, :]
```

we reshape a matrix of size 3x4 into a matrix of size [3, 1, 1, 4]

```
reshape matrix m[:, np.newaxis, np.newaxis, :] with newaxis  
[[[ [ 1  2  3  4]]]  
 [ [ 5  6  7  8]]]  
 [ [ 9 10 11 12]]]]
```

Another important numpy array or tensor technique is concatenation. Natural Language Processing (NLP) approaches use concatenation extensively on autoregressive models, for example, for language translation.

In the following code example we see how we can use concatenation with the numpy function `np.concatenate`

```
a = np.array([1,2,3,4])  
b = np.array([5,6,7,8])  
c = np.array([9,10,11])  
  
ab = np.concatenate( [a, b] )  
abc = np.concatenate( [a, b, c] )  
  
print(a)  
print(b)  
print(c)  
  
print("concatenate a with b ", ab)  
print("concatenate a with b with c", abc)
```

Here we can concatenate the following 3 numpy arrays

```
a = [1 2 3 4]
b = [5 6 7 8]
c = [9 10 11]
```

and obtain

```
concatenate a with b      [1 2 3 4 5 6 7 8]
concatenate a with b with c [1 2 3 4 5 6 7 8 9 10 11]
```

An example of concatenation with matrices can be seen below. Notice the use of axis 0 to indicate on what dimension to concatenate

```
m1 = np.array([ [1,2,3],
                [4,5,6],
                [7,8,9] ])

m2 = np.array([ [10,11,12],
                [13,14,14],
                [16,17,18] ])

print(m1)
print(m2)
m1_m2_concat = np.concatenate([m1, m2], axis=0)
print("m1 m2 concat axis 0 ")
print(m1_m2_concat)
```

We take the following 2 matrices

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

```
[[10 11 12]
 [13 14 14]
 [16 17 18]]
```

And concatenate them on axis 0

m1 m2 concat axis 0

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]
 [13 14 14]
 [16 17 18]]
```

You can concatenate on axis=1 like so

```
m1 = np.array([ [1,2,3],
                [4,5,6],
                [7,8,9] ])

m2 = np.array([ [10,11,12],
                [13,14,14],
                [16,17,18] ])

print(m1)
print(m2)
m1_m2_concat = np.concatenate([m1, m2], axis=1)
print("m1 m2 concat axis 1 ")
print(m1_m2_concat)
```

Concatenating the matrices m1 and m2

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]

[[10 11 12]
 [13 14 14]
 [16 17 18]]
```

On axis 1 gives us a new matrix of size 3x6

```
m1 m2 concat axis 1
[[ 1  2  3 10 11 12]
 [ 4  5  6 13 14 14]
 [ 7  8  9 16 17 18]]
```

Another approach to concatenation is to use `vstack` and `hstack`.

```
## np.vstack -->> vertical stack
## np.hstack -->> horizontal stack

m1 = np.array([ [1,2,3],
                [4,5,6],
                [7,8,9] ])

m2 = np.array([ [10,11,12],
                [13,14,14],
                [16,17,18] ])

print(m1)
print(m2)
m1_m2_concat = np.vstack( [m1, m2] )
print(" vstack ")
print(m1_m2_concat)

m1_m2_concat2 = np.hstack( [m1, m2] )
print(" hstack ")
print(m1_m2_concat2)
```

Using `vstack` and `hstack` on the following 2 matrices

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

```
[[10 11 12]
 [13 14 14]
 [16 17 18]]
```

Gives you the following for vstack

```
vstack
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]
 [13 14 14]
 [16 17 18]]
```

And the following for hstack

```
hstack
[[ 1  2  3 10 11 12]
 [ 4  5  6 13 14 14]
 [ 7  8  9 16 17 18]]
```

Obviously, having data means that you want to perform many kinds of math operations on this data. The following are some examples of some of the most common operations.

```
## math operations in numpy

x = np.array([1,2,3,4])
print(x)
print("x + 10 ", x+10)
print("x - 10 ", x-10)
print("x * 10 ", x*10)
print("x / 2 ", x/2)
print(" -x ", -x)
print("x ** 3", x ** 3)
print("4^x", np.power(4, x) )
print(" np.log(x) ", np.log(x))
print(" np.log2(x)", np.log2(x))
print("np.log10(x)", np.log10(x) )
```



The results of the previous math operations are as follows

```
[1 2 3 4]
x + 10 [11 12 13 14]
x - 10 [-9 -8 -7 -6]
x * 10 [10 20 30 40]
x / 2 [0.5 1. 1.5 2. ]
-x [-1 -2 -3 -4]
x ** 3 [ 1 8 27 64]
4^x [ 4 16 64 256]
np.log(x) [0.      0.69314718 1.09861229 1.38629436]
np.log2(x) [0.      1.      1.5849625 2.      ]
np.log10(x) [0.      0.30103  0.47712125 0.60205999]
```

Trigonometric functions like sines and cosines are also available in numpy as can be seen here:

```
## min max n_samples
angles = np.linspace(0, 4, 10)
print(angles)
print("np.sin")
print(np.sin(angles))
```

Here we generate a numpy array of size 10

```
[0.      0.44444444 0.88888889 1.33333333 1.77777778 2.22222222
 2.66666667 3.11111111 3.55555556 4.      ]
```

As an example, we can then use np.sin to calculate our corresponding sine values

```
np.sin
[ 0.      0.42995636 0.77637192 0.9719379  0.9786557  0.79522006
 0.45727263 0.03047682 -0.40224065 -0.7568025 ]
```

Aggregates in numpy are ways in which you can perform an operation and reduce the result. Much like **tf.reduce\_sum()** in tensorflow. A numpy example of aggregate operations can be seen below

```
x = np.array( [1 , 2, 3, 4, 5] )
print(x)
print(" np.add.reduce(x) = ", np.add.reduce(x) )
print(" np.multiply.reduce(x) = ", np.multiply.reduce(x) )
print(" np.sum(x) = ", np.sum(x) )
print(" np.min(x) = ", np.min(x) )
print(" np.max(x) = ", np.max(x) )
```

The result of applying numpy aggregate functions gives us the following:

```
[1 2 3 4 5]
np.add.reduce(x) = 15
np.multiply.reduce(x) = 120
np.sum(x) = 15
np.min(x) = 1
np.max(x) = 5
```

Sometimes it is useful to extract minimums and maximums of values across dimensions. We can do that in numpy with np.min, np.sum, etc. For example:

```
m = np.array( [[1, 2, 3, 4],
               [5, 6, 7, 8],
               [9, 10, 11, 12],
               [13, 14, 15, 16] ])

print(m)

print(" m.sum() = ", m.sum() )
print(" np.min(m, axis=0) = ", np.min(m, axis=0) )
print(" np.min(m, axis=1) = ", np.min(m, axis=1) )
print(" np.min(m, axis=-1) (-1 is last item) = ", np.min(m, axis=-1) )
```

Here for the following matrix

```
[[ 1 2 3 4]
 [ 5 6 7 8]
 [ 9 10 11 12]
 [13 14 15 16]]
```

We can get the sum of the matrix and min values across different dimensions.

```
m.sum() = 136
np.min(m, axis=0) = [1 2 3 4]
np.min(m, axis=1) = [ 1 5 9 13]
np.min(m, axis=-1) (-1 is last item) = [ 1 5 9 13]
```

## Broadcasting

One extremely useful concept in numpy and tensorflow is “Broadcasting”. Here are some examples. Broadcasting allows you to perform an operation element wise between matrices and vectors when all we want is the smaller numpy array to be repeated.

```
## broadcasting
a = np.array( [0, 1, 2] )
m = np.ones( (3,3) )
ma = m + a
print(" m ")
print(m)
print(" a ")
print(a)
print("m + a ")
print(ma)
a = a[:, np.newaxis]
print("a[:, np.newaxis]")
print(a)
print("m + a ")
ma = m + a
print(ma)
```

The results of our previous code can be seen here. Broadcasting is best understood with examples. Deep learning algorithms such as Transformers rely heavily on broadcasting. Learn it well. In this example, we have a matrix “m”

```
m
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
```

And a vector “a”

```
a
[0 1 2]
```

We can broadcast add “a” to “m” and get.

```
m + a
[[1. 2. 3.]
 [1. 2. 3.]
 [1. 2. 3.]]
```

If we make “a” into a column vector.

```
a[:, np.newaxis]
[[0]
 [1]
 [2]]
```

We can broadcast add the new column vector “a” to matrix “m” like so

```
m + a
[[1. 1. 1.]
 [2. 2. 2.]
 [3. 3. 3.]]
```

We can also broadcast 2 vectors and have them be multiplied to get a matrix like so

```
v1 = np.array( [1, 1, 1] )
v2 = np.array( [0, 1, 2] )[:, np.newaxis]

print(v1)
print(v2)

vvplus = v1 + v2
v1v2 = v1 * v2

print("v1 + v2")
print(v1 + v2)
print("v1 * v2")
print(v1 * v2)
```

Here we have row vector v1

```
[1 1 1]
```

And column vector v2

```
[[0]
 [1]
 [2]]
```

We can then perform a broadcast add between v1 and v2

```
v1 + v2
[[1 1 1]
 [2 2 2]
 [3 3 3]]
```

And a broadcast matrix multiply. This multiply is element-wise.

$v1 * v2$

$\begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 2 & 2 & 2 \end{bmatrix}$

The previous multiplication with broadcasting element wise would look like this.

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} * \begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 2 & 2 & 2 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 2 & 2 & 2 \end{bmatrix}$$

Another example

```
## broadcasting
## 2 vectors multiplied to get a matrix

v1 = np.array( [1, 1, 1] )[:, np.newaxis]
v2 = np.array( [0, 1, 2] )

print(v1)
print(v2)

vvplus = v1 + v2
v1v2 = v1 * v2

print("v1 + v2")
print(v1 + v2)
print("v1 * v2")
print(v1 * v2)
```

Given v1

$$\begin{bmatrix} [1] \\ [1] \\ [1] \end{bmatrix}$$

Given v2

$$[0 \ 1 \ 2]$$

We can perform an element wise sum

$$\begin{aligned} &v1 + v2 \\ &\begin{bmatrix} [1 \ 2 \ 3] \\ [1 \ 2 \ 3] \\ [1 \ 2 \ 3] \end{bmatrix} \end{aligned}$$

This sum is equivalent to

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} + \begin{bmatrix} 0 & 1 & 2 \\ 0 & 1 & 2 \\ 0 & 1 & 2 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix}$$

The element wise multiplication results in

$$\begin{aligned} &v1 * v2 \\ &\begin{bmatrix} [0 \ 1 \ 2] \\ [0 \ 1 \ 2] \\ [0 \ 1 \ 2] \end{bmatrix} \end{aligned}$$

Which would look like this

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} * \begin{bmatrix} 0 & 1 & 2 \\ 0 & 1 & 2 \\ 0 & 1 & 2 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 2 \\ 0 & 1 & 2 \\ 0 & 1 & 2 \end{bmatrix}$$

## Masks

Masks are a very important technique in NLP. We apply Boolean logic to vectors and matrices to identify certain values like all the values that are zero. Here we can see an example of how to create masks using Boolean logic.

```
## masks
## boolean

m = np.array( [[1, 2, 3, 4],
               [5, 6, 7, 8],
               [9, 10, 11, 12],
               [13, 14, 15, 16] ])

print(m)

print("m < 5")
print(m < 5)

print("m > 11")
print(m > 11)

print("m == 14")
print(m == 14)

print("np.equal(m, 13)")
print(np.equal(m, 13))

print("np.equal(m, 13)")
print(np.equal(m, 13))

print("np.sum(m < 12, axis=1)")
print("how many values less than 12 in each row?")
print(np.sum(m < 12, axis=1)[: , np.newaxis])

print("m[m < 8]")
print(m[m < 8])
```



Given matrix “m”

```
[[ 1 2 3 4]
 [ 5 6 7 8]
 [ 9 10 11 12]
 [13 14 15 16]]
```

We can apply the following Boolean expressions

```
m < 5
[[ True True True True]
 [False False False False]
 [False False False False]
 [False False False False]]
```

```
m > 11
[[False False False False]
 [False False False False]
 [False False False True]
 [ True True True True]]
```

```
m == 14
[[False False False False]
 [False False False False]
 [False False False False]
 [False True False False]]
```

```
np.equal(m, 13)
[[False False False False]
 [False False False False]
 [False False False False]
 [ True False False False]]
```

```
np.equal(m, 13)
[[False False False False]
 [False False False False]
 [False False False False]
 [ True False False False]]
```

```
np.sum(m < 12, axis=1)
how many values less than 12 in each row?
[[4]
 [4]
 [3]
 [0]]
```

```
m[m < 8]
[1 2 3 4 5 6 7]
```

A more detailed form of indexing is called fancy indexing. These are some examples of fancy indexing or slicing

```
## masks
## boolean

m = np.array( [[1, 2, 3, 4],
               [5, 6, 7, 8],
               [9, 10, 11, 12],
               [13, 14, 15, 16] ])

print(m)
row = np.array( [0, 1, 2] )
col = np.array( [0, 1, 2] )
print("row = np.array( [0, 1, 2] )")
print("col = np.array( [0, 1, 2] )")
print(" m[row,col] ")
print( m[row,col] )
print("m[2:, [0, 2]] ")
print( m[2:, [0, 2]] )
```

Now let us look at some of the results. For the matrix m

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]
 [13 14 15 16]]
```

We can define a set of indices and perform the following slicing operation

```
row = np.array( [0, 1, 2] )
col = np.array( [0, 1, 2] )

m[row,col]
[ 1  6 11]
```

We can also do the following

```
m[2:,[0,2]]  
[[ 9 11]  
 [13 15]]
```

Here, a slightly more complex broadcasting example is presented

```
m = np.array( [[1, 2],  
               [3, 4],  
               [5, 6],  
               [7, 8],  
               [9, 10],  
               [11, 12],  
               [13, 14],  
               [15, 16],  
               [17, 18],  
               [19, 20] ] )  
  
print(m)  
m1 = m[:, np.newaxis, :] ## for broadcasting  
print(m1.shape)  
print(m1)  
  
m2 = m[np.newaxis, :, :] ## for broadcasting  
print(m2.shape)  
print(m2)  
  
diff = (m1 - m2)**2  
print(diff.shape)  
print(" diff = (m1 - m2)**2 ")  
print(diff)  
print("np.sum(diff, axis=-1)")  
print(np.sum(diff, axis=-1))
```

Given the following matrix

```
[[ 1 2]
 [ 3 4]
 [ 5 6]
 [ 7 8]
 [ 9 10]
 [11 12]
 [13 14]
 [15 16]
 [17 18]
 [19 20]]
```

We can reshape it with

```
m1 = m[:, np.newaxis, :] ## for broadcasting
```

the shape is now

```
(10, 1, 2)
```

And it looks like this

```
[[[ 1 2]]
 [[ 3 4]]
 [[ 5 6]]
 [[ 7 8]]
 [[ 9 10]]
 [[11 12]]
 [[13 14]]
 [[15 16]]
 [[17 18]]
 [[19 20]]]
```

We can create a new matrix m2 from the original m with

```
m2 = m[np.newaxis, :, :] ## for broadcasting
```

this gives us a new shape

```
print(m2.shape)
```

```
(1, 10, 2)
```

And m2 looks like this

```
[[[ 1 2]
   [ 3 4]
   [ 5 6]
   [ 7 8]
   [ 9 10]
   [11 12]
   [13 14]
   [15 16]
   [17 18]
   [19 20]]]
```

Now we can perform the following operations with

```
diff = (m1 - m2)**2
print(diff.shape)
print(" diff = (m1 - m2)**2 ")
print(diff)
print("np.sum(diff, axis=-1)")
print(np.sum(diff, axis=-1))
```

and get the following results

```
(10, 10, 2)
```

```

diff = (m1 - m2)**2
[[[ 0  0]
  [ 4  4]
  [16 16]
  [36 36]
  [64 64]
  [100 100]
  [144 144]
  [196 196]
  [256 256]
  [324 324]]
 ...

 [[324 324]
  [256 256]
  [196 196]
  [144 144]
  [100 100]
  [ 64  64]
  [ 36  36]
  [ 16  16]
  [  4  4]
  [  0  0]]]

np.sum(diff, axis=-1)
[[ 0  8 32 72 128 200 288 392 512 648]
 [ 8  0 32 72 128 200 288 392 512]
 [32  8  0 32 72 128 200 288 392]
 [72 32  8  0 32 72 128 200 288]
 [128 72 32  8  0 32 72 128 200]
 [200 128 72 32  8  0 32 72 128]
 [288 200 128 72 32  8  0 32 72]
 [392 288 200 128 72 32  8  0 32]
 [512 392 288 200 128 72 32  8  0]
 [648 512 392 288 200 128 72 32  8  0]]

```

That concludes a quick introduction to numpy arrays. Now we are ready to move on to tensorflow operations on tensors.

### 1.3.2 Tensor Operations with Tensorflow

In this section, we will now look at some special tensor operations and how they can be done with the tensorflow framework.

First we call the Tensorflow library.

```
import tensorflow as tf
import numpy as np

import matplotlib.pyplot as plt
```

In the tensorflow low level API which uses a static graph, you need a session to see the results of tensor operations. I usually just start the session, define my computational graph segments and then execute them when I want to see the results. In the next code segment, the session is initialized as follows:

```
sess=tf.Session() #start a session
```

Let us begin our discussion of tensor operations with some simple examples defining tensors in tensorflow.

```

#define tensors
a=tf.constant([[10,20],[30,40]]) #Dimension 2X2
b=tf.constant([5])
c=tf.constant([2,2])
d=tf.constant([[3],[3]])

#Run tensors to generate arrays
mat,scalar,one_d,two_d = sess.run([a,b,c,d])

print("mat")
print(mat)

print("scalar")
print(scalar)

print("one_d")
print(one_d)

print("two_d")
print(two_d)

```

After running the session, our tensors look like this

The tensor “mat” is a 2x2 tensor

```

mat
[[10 20]
 [30 40]]

```

The tensor “scalar” is a value of one called a scalar

```

scalar
[5]

```

The tensor one\_d is a 1x2 tensor

```

one_d
[2 2]

```



The tensor “two\_d” is a 2x1 tensor

```
two_d  
[[3]  
 [3]]
```

We can multiply a matrix with a scalar like so. The tensorflow function **tf.multiply** performs an element wise multiplication.

```
#broadcast multiplication with scalar  
print( " sess.run(tf.multiply(mat,scalar)) " )  
print( sess.run(tf.multiply(mat,scalar)) )
```

The previous operation gives us

```
sess.run(tf.multiply(mat,scalar))  
  
[[ 50 100]  
 [150 200]]
```

This element wise multiplication is equivalent to

$$\begin{bmatrix} 10 & 20 \\ 30 & 40 \end{bmatrix} * \begin{bmatrix} 5 & 5 \\ 5 & 5 \end{bmatrix} = \begin{bmatrix} 50 & 100 \\ 150 & 200 \end{bmatrix}$$

Now we perform a broadcast multiplication with a 1x2 array between the matrix “mat” and one\_d

```
#broadcast multiplication with 1_d array (Dimension 1X2)
print( " sess.run(tf.multiply(mat,one_d)) " )
print( sess.run(tf.multiply(mat,one_d)) )
```

This operation results in

```
sess.run(tf.multiply(mat,one_d))
[[20 40]
 [60 80]]
```

This element wise multiplication is equivalent to

$$\begin{bmatrix} 10 & 20 \\ 30 & 40 \end{bmatrix} * \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix} = \begin{bmatrix} 20 & 40 \\ 60 & 80 \end{bmatrix}$$

Finally, we perform a broadcast multiply (element wise) between the matrix “mat” and a column vector of size 2x1

```
#broadcast multiply 2_d array (Dimension 2X1)
print( " sess.run(tf.multiply(mat,two_d)) " )
print( sess.run(tf.multiply(mat,two_d)) )
```

Which results is

```
sess.run(tf.multiply(mat,two_d))
[[ 30 60]
 [ 90 120]]
```

This element wise multiplication is equivalent to

$$\begin{bmatrix} 10 & 20 \\ 30 & 40 \end{bmatrix} * \begin{bmatrix} 3 & 3 \\ 3 & 3 \end{bmatrix} = \begin{bmatrix} 30 & 60 \\ 90 & 120 \end{bmatrix}$$

Tensorflow can take a numpy array and convert it into a tensorflow tensor like so

```
arr = np.array([1, 5.5, 32, 11, 20])  
  
tensor1 = tf.convert_to_tensor(arr,tf.float64)  
  
print(tensor1)
```

When you print it, it will look like this

```
Tensor("Const_4:0", shape=(5,), dtype=float64)
```

Another example

```
arr = np.array([1, 5.5, 3, 11, 30])  
  
tensor = tf.convert_to_tensor(arr,tf.float64)  
  
print(sess.run(tensor))  
  
print(sess.run(tensor[1]))
```

Which gives us

```
[ 1.  5.5  3. 11. 30.]  
  
5.5
```

We can also convert numpy matrices into tensors in tensorflow like so

```
d2arr = np.array([ [1, 5.5, 3, 15, 20],
                   [10, 22, 30, 4, 50],
                   [60, 70, 83, 90, 101] ])

tensor = tf.convert_to_tensor(d2arr)

print( sess.run(tensor) )
```

The resulting tensor looks like this

```
[[ 1.  5.5  3. 15. 20.]
 [10. 22. 30.  4. 50.]
 [60. 70. 83. 90. 101.]]
```

In tensorflow **tf.multiply(a,b)** is identical to  $a*b$ . The function **tf.multiply(X, Y)** does element-wise multiplication so that

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 1 & 3 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 6 \\ 6 & 4 \end{bmatrix}$$

whereas **tf.matmul** does a matrix multiplication so that

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 3 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 3 \\ 2 & 1 \end{bmatrix}$$

Using **tf.matmul(X, X, transpose\_b=True)** means that you are calculating  $X \cdot X^T$  where  $^T$  indicates the transposition of the matrix and  $\cdot$  is the matrix multiplication.

The following are some examples of matrix element wise multiplication with **tf.multiply** vs. matrix multiplication with **matmul()**

```
a = tf.constant([[1, 2],
                 [3, 4]])

b = tf.constant([[1, 1],
                 [1, 1]])

print(sess.run( tf.add(a, b) ))

print(sess.run( tf.multiply(a, b) ))

print(sess.run( tf.matmul(a, b) ))
```

Given 2 matrices a and b

```
a = tf.constant([[1, 2],
                 [3, 4]])

b = tf.constant([[1, 1],
                 [1, 1]])
```

Performing **tf.add** gives us

```
[[2 3]
 [4 5]]
```

This is equivalent to

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 3 \\ 4 & 5 \end{bmatrix}$$

Now, when we perform an element wise multiplication with **tf.multiply** as follows

```
print(sess.run( tf.multiply(a, b) ))
```

we get

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

This is **tf.multiply** operation is equivalent to the following

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} * \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

Finally, the **tf.matmul** operation performs a matrix multiplication

```
print(sess.run( tf.matmul(a, b) ))
```

which results in

$$\begin{bmatrix} 3 & 3 \\ 7 & 7 \end{bmatrix}$$

This is **tf.matmul** operation is equivalent to the following

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 3 & 3 \\ 7 & 7 \end{bmatrix}$$

In numpy, you use **.dot()** or **np.matmul()** for matrix multiplication.

Other important functions in Tensorflow that you will use a lot for deep learning can be seen below. Let us look at an example.

Given a matrix **c**

```
c = tf.constant([[4.0, 5.0], [10.0, 1.0]])  
print( sess.run( c ) )
```

Which looks like this

```
[[ 4.  5.]  
 [10.  1.]]
```

You can find the largest value in the matrix using **tf.reduce\_max()** like so. This is a similar concept to aggregates in numpy

```
# Find the largest value  
print(" sess.run( tf.reduce_max(c) ) ")  
print( sess.run( tf.reduce_max(c) ) )
```

The result of **tf.reduce\_max** is

```
sess.run( tf.reduce_max(c) )  
10.0
```

Instead, if we want the index of the largest value we can use the **tf.argmax()**.

```
# Find the index of the largest value
print( " sess.run( tf.argmax(c) )" )
print( sess.run( tf.argmax(c) ) )
```

The argmax function gives us the following indices

```
sess.run( tf.argmax(c) )
```

```
[1 0]
```

For row 1 and column 0.

Finally, the softmax function is very important and is widely used in deep learning.

```
# Compute the softmax
print( " sess.run( tf.nn.softmax(c) )" )
print( sess.run( tf.nn.softmax(c) ) )
```

The results are as follows

```
sess.run( tf.nn.softmax(c) )
[[2.6894143e-01 7.3105860e-01]
 [9.9987662e-01 1.2339458e-04]]
```



A few other commonly used utility functions in Tensorflow, with examples, are shown next. The following Tensorflow functions are commonly used in deep learning and throughout this.

### **tf.one\_hot()**

Tensorflow provide a function for one-hot encoding which is:

`tf.one_hot()`

This function takes a **y** vector and converts it to the one-hot encoded version. For example:

```
indices = [0, 1, 2]
depth = 3
print tf.one_hot(indices, depth)
```

The output is

```
[[ 1, 0, 0],
 [ 0, 1, 0],
 [ 0, 0, 1] ]
```

Another example of one hot encoding can be see here

```
y = tf.one_hot(indices, depth)
depth = 4
indices = [0, 3]
print(sess.run(y))
```

### **tf.reduce\_mean()**

The function **tf.reduce\_mean()** is a built-in Tensorflow function that takes as input a tensor and computes the mean of the elements across the dimensions of a given tensor. So, it returns a reduced tensor. For example, given:

```
x = tf.constant( [[1, 1] , [2, 2]] )
```

we can get the following:

```
all = tf.reduce_mean(x)    =>    # 1.5
```

```
all = tf.reduce_mean(x, 0) =>    # [1.5, 1.5]
```

```
all = tf.reduce_mean(x, 1) =>    # [1, 2]
```

### **tf.reduce\_sum()**

The function **tf.reduce\_sum()** computes the sum of the elements across dimensions of a tensor. For example:

```
x = tf.constant( [[1, 1, 1], [1, 1, 1]])  
y = tf.reduce_sum(x)  #6  
y = tf.reduce_sum(x, 0)  #[2, 2, 2]  
y = tf.reduce_sum(x, 1)  #[3, 3]
```

### **tf.argmax()**

The function **tf.argmax()** is a Tensorflow function that returns the index of the largest value across the axis of a tensor.

For example,

```
answer = tf.argmax([35, 4, 72, 2])    =>    2
```

you can also define an axis like so

```
answer = tf.argmax( [ [23, 32, 49],  
                      [45,  1, 12] ] )
```

The previous function returns => [2, 0]



### **tf.equal()**

The `tf.equal()` function returns a vector of boolean values that compares the values in two tensors of equal dimensions.

For example, given:

```
x = [1, 2, 3]
```

```
y = [0, 1, 3]
```

```
tf.equal(x, y)    =>    [False, False, True]  or  [0, 0, 1]
```



## **tf.reshape()**

The function `tf.reshape` is a very important function in tensorflow that is widely used.

```
tensor 't' is [1, 2, 3, 4, 5, 6, 7, 8, 9]
tensor 't' has shape [9]
tf.reshape(t, [3, 3]) ==> [[1, 2, 3],
                           [4, 5, 6],
                           [7, 8, 9]]
```

Tensorflow has many other functions to calculate dimensions, shapes, and ranks. These are crucial when building deep neural networks from scratch. Some examples can be seen below.

```
rank_4_tensor = tf.zeros([3, 2, 4, 5])

print("Type of every element:", rank_4_tensor.dtype)
print("Shape of tensor:", rank_4_tensor.shape)
print("Elements along axis 0 of tensor:", rank_4_tensor.shape[0])
print("Elements along the last axis of tensor:", rank_4_tensor.shape[-1])
```

The output of these functions is as follows

```
Type of every element: <dtype: 'float32'>
Shape of tensor: (3, 2, 4, 5)
Elements along axis 0 of tensor: 3
Elements along the last axis of tensor: 5
```

Tensorflow also uses Broadcasting extensively. Understand this well by practicing a lot of examples possibly beyond this book. Broadcasting applies to elementwise multiplications with **tf.multiply** or **\***. Given that broadcasting can be best understood through examples, I will provide a few examples to better illustrate its use.

```
## Broadcasting

x = tf.constant([1, 2, 3])
y = tf.constant(2)
z = tf.constant([2, 2, 2])

print(sess.run(x))
print(sess.run(y))
print(sess.run(z))

print("sess.run(tf.multiply(x, 2))")
print(sess.run(tf.multiply(x, 2)))

print("sess.run(x * y)")
print(sess.run(x * y))

print("sess.run(x * z)")
print(sess.run(x * z))
```

Given the following tensors

```
x = [1 2 3]
y = 2
z = [2 2 2]
```

we can perform the following element wise multiplication

```
sess.run(tf.multiply(x, 2))
```

and get

```
[2 4 6]
```

An element wise multiplication between x and y

```
sess.run(x * y)
```

gives us

```
[2 4 6]
```

And finally for this example, a matrix multiplication between x and z

```
sess.run(x * z)
```

results in

```
[2 4 6]
```

Did you notice that they all result in the same output. Why? Hint: the answer has to do with broadcasting.

Here is yet another example of broadcasting. Hopefully, you are starting to get the idea that broadcasting is really important. In particular, it is widely used in deep learning algorithms that require the use of masks like in Transformers.

See if you can work this out by hand.

```

x = tf.constant([1, 2, 3])
y = tf.constant(2)
z = tf.constant([2, 2, 2])

print(sess.run(x))
print(sess.run(y))
print(sess.run(z))

x = tf.reshape(x,[3,1])
y = tf.range(1, 5)

print(x, "\n")
print(sess.run(x))

print(y, "\n")
print(sess.run(y))

print(tf.multiply(x, y))
print(sess.run(tf.multiply(x, y)))

```

The result looks like this

```

[1 2 3]

2

[2 2 2]

Tensor("Reshape:0", shape=(3, 1), dtype=int32)
[[1]
 [2]
 [3]]

Tensor("range:0", shape=(4,), dtype=int32)
[1 2 3 4]

Tensor("Mul_7:0", shape=(3, 4), dtype=int32)
[[ 1  2  3  4]
 [ 2  4  6  8]
 [ 3  6  9 12]]

```

This concludes a quick introduction to tensor operations with tensorflow. You will certainly encounter a lot of this and much more in your journey through machine learning and deep learning. Don't forget to close the session.

```
sess.close()
```

As I recommended earlier, for a better treatment of linear algebra and optimization I recommend the book “Linear Algebra and Optimization for Machine Learning: A Textbook” by Charu C. Aggarwal.

## 1.4 Summary

In this chapter (chapter 1), I have discussed the basic outline of the book and the environment setup to get started with Tensorflow and deep learning. I have also provided a brief introduction to numpy arrays, tensors, linear algebra operations commonly used in deep learning. The next chapter (chapter 2) will provide a quick introduction to machine learning using the Sklearn tool kit. Whenever prudent, the traditional machine learning techniques will be discussed and compared or contrasted to deep learning approaches.



## CHAPTER 2: TRADITIONAL MACHINE LEARNING

In this chapter, I will address some of the general machine learning topics. I will use python and the Sklearn library to provide example code on how to use the different classifiers and other tools. I will quickly move through logistic regression to arrive at other algorithms. Much of what we learn from using the SKlearn kit can be integrated with Tensorflow to make your deep learning code more powerful and modular. That way, you will be able to re-use your code for many different tasks. I will not focus on the theory of machine learning algorithms. There are many books on the theory of machine learning algorithms so instead I will concentrate on the practical aspects of using them in Python.

Machine Learning (ML) is essential for automated systems to make decisions and to infer new knowledge about the world. This section describes some of the most important methodologies currently in use in the field of machine learning (see the table below). Machine learning approaches can be divided into supervised learning (such as Support Vector Machines) and unsupervised learning (such as K-means clustering). Within supervised approaches, the learning methodologies can be divided based on whether they predict a class or a magnitude into classifiers and regression models, respectively. An additional categorization for these methods depends on whether they use sequential or non-sequential data. A comparison of different machine learning approaches is provided in the table below.

**Table: Machine learning techniques**

<b>Technique</b>	<b>Definition</b>	<b>Pros</b>	<b>Cons</b>
Support Vector Machines	Supervised learning approach that optimizes the margin that separates data.	SLT Confidence characteristic (expected risk)	class imbalance issues
Decision Trees	This method performs classification by constructing trees where branches are separated by decision points.	Easy to understand	Not flexible
Neural Networks	Model represents the structure of the human brain with neurons and links to the neurons.	Versatile	Can obscure the underlying structure of the model
K-means clustering	Unsupervised method that forms k-means clusters to minimize distance between centroids and members of cluster.	Unsupervised – so no training needed	Needs clearly defined separations in the data in order to be effective
Linear Discriminant Analysis (LDA)	Creates linear function of features to classify data	Simple yet robust classification method	Normality assumptions of the classes

Naïve Bayes	Probabilistic Learning to calculate the probability of seeing a certain condition in the world by selecting the most probable class given the feature vector	Fast, easy to understand the model	Bayes assumptions of independence
Maximum Likelihood Estimation (MLE)	Calculates the likelihood that an object will be seen based on its proportion in the sample data	Simple	Too simplistic for some applications
Hidden Markov Models (HMM)	A Markov Chain is a weighted automaton consisting of nodes and arcs where the nodes represent states and the arcs represent the probability of going from one state to another.	Probabilistic. Good for sequence mining	Combinatorial complexity/ needs prior knowledge

Classifiers are machine learning approaches that produce as an output a specific class given some input features. Important classifiers include Support Vector Machines (Burges 1998) commonly implemented using LibSVM (Chang and Lin, 2001), Naïve Bayes, artificial neural networks, deep learning based neural networks, decision trees, random forests, and the k-nearest neighbor classifier (Witten and Frank, 2005).

In their simplest form, deep learning based methods are simply neural nets with more layers. Deep learning methods have made a big impact in the field of machine learning in recent years. In particular, they are very important because, given enough computational power, they can automatically learn the optimal features to be used in a classification problem. In the past, learning what features to use

required using humans to engineer the features. This issue has now been alleviated somewhat by deep learning.

Additionally, artificial neural networks are classifiers that can handle non-linearly separable data. In theory, this capability allows them to model data that may be more difficult to classify.

## **2.1 Code Issues**

Before we begin, I want to address some issues about the code. First, I have tried to be consistent with other programmers of machine learning in Python. We will be using many libraries to implement the classifiers. In particular, I selected to use the SKlearn library since it is very powerful and widely use.

In general, I only introduce enough SKlearn code for us to know how to integrate it with Tensorflow in our deep learning endeavors. For a more in-depth discussion of SKlearn, I highly recommend the book “Python Machine Learning” by Sebastian Raschka. I have tried to be consistent with the conventions used in that book so that both books can be used together.

So now, getting back to the code, here is an example of some of the most important libraries that we can use for our machine learning code.

```
import numpy as np
from sklearn import datasets
from sklearn.cross_validation import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score
from matplotlib.colors import ListedColormap
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score, f1_score
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn import decomposition
```

As you may imagine the **SKlearn** library is the main library which contains most of the traditional machine learning tools we will discuss in this chapter. The **numpy** library is essential for efficient matrix and linear algebra operations. For those with experience with MatLab, I can say that numpy is a way of performing linear algebra operations in python similar to how they are done in MatLab. This makes the code more efficient in its implementation and faster as well. The **datasets** library seen above helps to obtain standard corpora. You can use it to obtain annotated data like Fisher's iris data set, for instance.

From **sklearn.cross\_validation** we can import **train\_test\_split** which is used to create splits in a data matrix such as 70% for training purposes and 30% for testing purposes. From **sklearn.preprocessing** we can import the **StandardScaler** module which helps to scale feature data. We will use functions such as these to scale our data for the Tensorflow based classifiers. Deep learning algorithms can

improve significantly when data is properly scaled. So, it is recommended to do this.

We will use the **sklearn.metrics** module for performance evaluation of the classifiers. I will show that this module can be used with SKlearn classifiers and with Tensorflow classifiers. Again, this will help to more easily understand deep learning since we don't have to use the more complex and very verbose Tensorflow functions. The main metrics used are:

- **accuracy\_score**
- **recall\_score**
- **f1\_score**
- **precision\_score**
- **confusion\_matrix**

Two more very important libraries are **matplotlib.pyplot** and **pandas**. The **matplotlib.pyplot** library is very useful for visualization of data and results and the **pandas** library is very useful for pre-processing. The **pandas** library can be very useful to pre-process large data sets in very fast and very efficient ways.

There are some parameters that are sometimes useful to set in your code. The code sample below shows the use of **np.set\_printoptions**. The function is used to print all values in a numpy array. This can be useful when trying to visualize the contents of a large data set.

```
## set parameters
np.set_printoptions(threshold=np.inf) ## print all values in numpy
array
```

Let us assume that our data is stored in the matrix **X**. The code segment below uses the function **train\_test\_split**. This function is used to split a data set (in this case

**X**) into 4 sets which are **X\_train**, **X\_test**, **y\_train**, **y\_test**. These are the 4 sets that will be used by the traditional classifiers or the deep learning classifiers. The sets that start with **X** hold the data (feature vectors) and the sets that start with **y** hold the labels per sample (e.g.  $y_1$  for the first feature vector,  $y_2$  for the second feature vector, and so on).

The values **test\_size=0.01** and **random\_state=42** in the function are parameters that define the split. The value 0.01 makes a train set that has 99% of all samples while the test set has 1% of all samples. In contrast **test\_size=0.20** would mean that there is a 80% and 20% split. The **random\_state=42** allows you to always get the same random data since the seed is defined as 42.

```
#X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.30, random_state=48)
## k-folds cross validation all goes in train sets (hence 0.01)
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.01, random_state=42)
```

To call all the functions or classifiers you can employ the following approach. Here we have defined 5 common classifiers. Notice that each one gets the 4 data sets obtained from the percentage split. Notice also that the data files have a **\_normalized** added to their name.

This is a good standard approach used by programmers to indicate that this data has been scaled. The next chapter addresses scaling. Here you run **X\_train** through a scaler function to obtain **X\_train\_normalized**. The labels (**y**) are not scaled.

```
#logistic_regression_rc(X_train_normalized, y_train,  
X_test_normalized, y_test)  
#svm_rc(X_train_normalized, y_train, X_test_normalized, y_test)  
#random_forest_rc(X_train, y_train, X_test, y_test)  
#knn_rc(X_train_normalized, y_train, X_test_normalized, y_test)  
multilayer_perceptron_rc(X_train_normalized, y_train,  
X_test_normalized, y_test)
```

Before we talk about the classifiers, let us address performance evaluation. Performance evaluation will help you to determine how good your classifier is given an annotated test data set.

## 2.2 Performance Evaluation

Evaluation of the performance of your classifiers is extremely important. The SKlearn kit provides very good modules to address this issue. In particular, evaluation often involves measuring accuracy, precision, recall, and f-measure. The best way to understand these metrics is to think of a confusion matrix. Confusion matrices show how many elements from a class are correctly and incorrectly classified.

For example, take the following:



	Predicted 1	Predicted 0
True 1	a	b
True 0	c	d

Given this table, we can calculate accuracy as

$$\text{accuracy} = (a+d) / (a + b + c + d)$$

precision is

$$\text{precision} = a / (a + c)$$

the recall metric can be computed as follows

$$\text{recall} = a / (a + b)$$

Finally, the f measure which is a harmonic average of recall and precision can be computed as follows

$$F = (2 * (\text{precision} * \text{recall})) / (\text{precision} + \text{recall})$$

The sample code to obtain these metrics is provided below.

```
def print_stats_percentage_train_test(algorithm_name, y_test,
                                     y_pred):

    print "algorithm is: ", algorithm_name
    print('Accuracy: %.2f' % accuracy_score(y_test, y_pred) )
    confmat = confusion_matrix(y_true=y_test, y_pred=y_pred)
    print "confusion matrix"
    print(confmat)
    print pd.crosstab(y_test, y_pred, rownames=['True'],
                      colnames=['Predicted'], margins=True)
    print('Precision: %.3f' % precision_score(y_true=y_test,
                                              y_pred=y_pred))
    print('Recall: %.3f' % recall_score(y_true=y_test,
                                        y_pred=y_pred))
    print('F1-measure: %.3f' % f1_score(y_true=y_test,
                                        y_pred=y_pred))
```

The code above shows a function to print the performance metric statistics. Notice that two sets are provided which are **y\_pred** and **y\_test**. The **y\_test** data set contains the original annotated labels. The **y\_pred** data set contains the labels predicted by your classifier. Each of the metric functions such as **f1\_score** and **recall\_score** uses these 2 data sets to calculate the respective metric.

## 2.3 Optimization

Before we begin to discuss some of the machine learning algorithms, I should say something about optimization. Optimization is a key process in machine learning. Basically, any supervised learning algorithm needs to learn a prediction equation given a set of annotated data. This prediction function usually has a set of parameters that must be learned. However, the question is “how do you learn these parameters?”

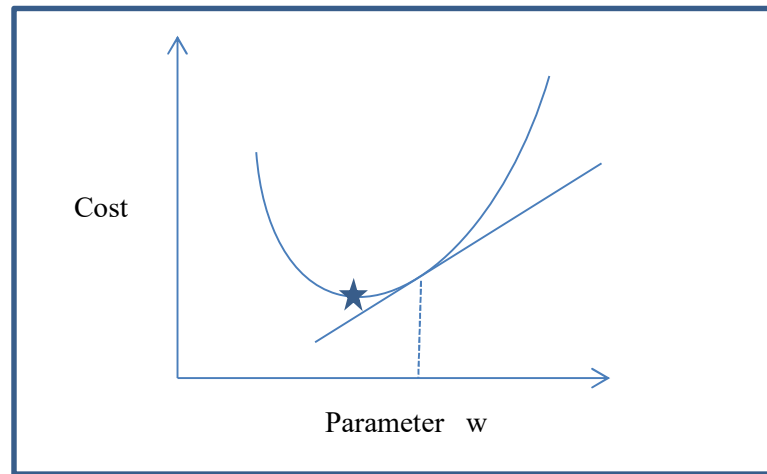
The answer is that you do so through:

### **optimization**

In its simplest form, optimization consists of trying a set of parameters with your model and seeing what result they give you. If the result is not good, the optimization algorithm needs to decide if you should decrease the values of the parameters or increase the values of the parameters.

In general, you do this in a loop (increasing and decreasing) until you find an optimal set of parameters. But one of the questions to answer here is: do the values go up or down? Well, as it turns out, there are methodologies based on calculus that help you to make this decision.

Let us try to picture this with a graph (below).



**Figure. Optimization graph**

The above graph represents an optimization problem. The y axis represents the cost (or penalty) of using a given parameter. The x axis represents the value of the parameter ( $w$ ) being used at the given iteration. The curve represents the behavior that the function being used to minimize the cost will follow for every value of parameter  $w$ .

As shown in the graph, the optimal value for the curve is found where the star is located (i.e where the value of cost is at a minimum). So, somehow the optimization algorithm needs to travel through the function and arrive at the position indicated by the star.

At that point, the value of “w” reduces the cost and finds the best solution. Instead of trying all values of “w” at random, the algorithm can make educated guesses about which direction to follow (up or down). To do this, we can use calculus to calculate the derivative of the function at a given point. This will allow us to determine the slope at that point. In the case of the graph, this represents the tangent line to the curve if we calculate the derivative at point w. If we calculate the slope at the position of the star symbol, then the slope is zero because the tangent at that point is parallel to the x axis. The slope at the point “w” will be positive. Based on this result, we can tell the direction we want to take for parameter w (decrease or increase). This type of optimization is called gradient descent and is very important in machine learning and deep learning. There are several approaches to implement gradient descent and this is just the simplest explanation for conceptual purposes. We can write the algorithm for this technique as follows:

```
old_x = 0
new_x = 4
step_size = 0.01
precision = 0.00001

def function_derivative(x):
    return 3*x ** 2 - 6*x

while absolute_value(new_x - old_x) > precision:
    old_x = new_x
    new_x = old_x - step_size * function_derivative(old_x)

print “result is: ”, new_x
```

In the previous code example we assume a function of

$$f() = x^3 - 3x^2 + 7$$

that needs to be optimized for parameter  $x$ . We will need the value of the derivative for each point  $x$ .

The derivative for  $f()$  is:

$$f'() = 3x^2 - 6x$$

So, the parameter  $x$  can be calculated in a loop using the derivative function which will determine the direction to follow when increasing or decreasing the parameter  $x$ .

## 2.4 Logistic Regression

Logistic regression is a simple algorithm that is often used by practitioners of machine learning because it can obtain good results. Logistic regression is a linear function much like linear regression which predicts the probability of a sample belonging to a given class. Logistic regression uses another optimization function instead of the standard least squares cost function used in linear regression.

The predicted values from a standard regression approach are now passed through a sigmoid function that basically maps the output to a probability range scale between 0 and 1. The code below provides an example of how to use the logistic regression function with SKlearn. Later, we will implement this logistic regression function again with Tensorflow.

```
def logistic_regression_rc(X_train_normalized, y_train,
                           X_test_normalized, y_test):
    from sklearn.linear_model import LogisticRegression
    lr = LogisticRegression(C=1000.0, random_state=0)
    lr.fit(X_train_normalized, y_train)
    y_pred = lr.predict(X_test_normalized)
    print_stats_percentage_train_test( "log re", y_test, y_pred)
```

In the previous function, the train and test sets are provided for the model to be trained and tested. In SKlearn most steps are abstracted. In contrast, Tensorflow will allow us to define more steps such as the cost function, optimization, and inference equation and other aspects. In the function `logistic_regression_rc`, first you initialized a logistic regression object (**lr**) and then you train and test it with the functions **lr.fit** and **lr.predict**. The final step is to measure performance using the previously described function `print_stats_percentage_train_test`. Most classifiers are implemented in the same way with SKlearn. In the next section, I will demonstrate how this is done for a neural network in Sklearn.

## 2.5 Neural Networks

Neural networks are very complex systems that take a long time to train. Therefore, the use of them in SKlearn may not be recommended except for the smallest of data sets. The code is shown here for contrast purposes with later implementations of neural networks in Tensorflow. In the next chapters, we will focus on how to do this in Tensorflow and how to create networks of multiple layers.

In the code below we can see that everything is very similar to the previous logistic regression implementation. The new changes appear in the definition of the **clf** multilayer object. Here, the parameter **hidden\_layer\_sizes=(100,100)** means that the architecture of the network consists of 2 hidden layers with 100 neurons each. A parameter such as (200, ) would mean that the network has 1 hidden layer with 200 neurons.

```
def multilayer_perceptron_rc(X_train_normalized, y_train,
                             X_test_normalized, y_test):
    from sklearn.neural_network import MLPClassifier
    clf = MLPClassifier(solver='lbfgs', alpha=1e-5,
                        hidden_layer_sizes=(100,100), random_state=1)
    clf.fit(X_train_normalized, y_train)
    y_pred = clf.predict(X_test_normalized)
    print_stats_percentage_train_test("multilayer perceptron",
                                      y_test, y_pred)
```

## 2.6 KNN

The k-nearest neighbor (KNN) classifier is a popular algorithm that I always like to use. It requires very little parameter tuning and can be easily implemented. Here, the Sklearn based code is similar to all previous approaches. The new aspects relate to the KNN parameters such as the value k.

The **n\_neighbors** parameter is the k. In this case, the five closest samples are selected.



```
def knn_rc(X_train_normalized, y_train, X_test_normalized, y_test):  
    from sklearn.neighbors import KNeighborsClassifier  
    knn = KNeighborsClassifier(n_neighbors=5, p=2,  
                              metric='minkowski')  
    knn.fit(X_train_normalized, y_train)  
    y_pred = knn.predict(X_test_normalized)  
    print_stats_percentage_train_test("knn", y_test, y_pred)
```

## 2.7 Support Vector Machines (SVM)

Before Deep Neural Networks, SVM was one of the big boys in the block. SVM is an example of a theoretically well founded machine learning algorithm. And for this reason it has always been very well respected by machine learning practitioners. This is in contrast to neural networks which haven't always been considered as very strong on their theoretical framework. As an example, in the rest of this chapter I will discuss SVM's framework and then provide example code to implement an SVM with SKlearn.

Support Vector Machines is a binary classification method based on statistical learning theory which maximizes the margin that separates samples from two classes (Burges 1998; Cortes 1995). This supervised learning machine provides the option of evaluating the data under different spaces through Kernels that range from simple linear to Radial Basis Functions [RBF] (Chang and Lin 2001; Burges 1998; Cortes 1995). Additionally, its wide use in the field of machine learning research and ability to handle large feature spaces makes it an attractive tool for many applications.

Statistical Learning Theory (SLT) methods assume prediction models that can be ascribed a confidence characteristic. They are based on the fact that both structural and empirical risks are minimized. The expected risk can be calculated based on the empirical risk that is present in the data with the associated upper and lower bounds. This generalization error can be expressed as follows:

$$R(\alpha) \leq R_{\text{emp}}(\alpha) + \sqrt{\left( \frac{h \left( \log \left( \frac{2l}{h} \right) + 1 \right) - \log \left( \frac{n}{4} \right)}{l} \right)}$$

In SVM, the maximization of the margin is based on the training samples that are closest to the optimal line (also known as support vectors). Because the method tries to maximize the margin between the samples of two classes under a set of constraints, it ultimately becomes an optimization problem to find the maximum separation band. The function that represents the margin is quadratic and can be solved using quadratic programming techniques with Lagrange operators. The “objective function” and constraints can be represented as follows where  $W$  is the weight vector:

$$\frac{1}{2} \|W\|^2$$

$$Y_i(W \cdot X_i + b) \geq 1$$

Non-linearly separable cases can be solved by mapping the initial set of features to a higher feature space by way of a Kernel Trick. This will provide higher freedom in separating the data in higher dimensional space. The Kernel trick takes advantage of the fact that SVMs do not need to know the mapping function because this is expressed as the dot product of the input data.

$$X_i \cdot X_j \rightarrow \phi(X_i) \cdot \phi(X_j)$$

Since most real world data includes outliers and noise, a soft margin approach can be introduced in the model to allow for some errors to occur. This softening of the margin is achieved by introducing an error term where the cost represents the penalty for each error

$$C \sum E_i$$

Under the soft margin approach, the objective function with constraints can be written as follows:

$$\frac{1}{2} \|W\|^2 + C \sum E_i$$

$$Y_i(W \cdot X_i + b) \geq 1 - E_i$$

where  $i$  represents each training sample,  $W$  is the weight vector (normal vector) that defines the maximum margin,  $b$  is the bias,  $Y_i$  is the class for each training sample  $i$ , and  $X_i$  is the feature vector for each training sample  $i$ . The  $E_i$  term in the model represents the slack error for each sample. The cost parameter ( $C$ ) represents the penalty for each error.

Although the SVM is used for binary classification, multiclass implementations can be achieved by creating different classifiers for each pair of class comparisons.

Multiple kernels can be used with support vector machines to map the feature vectors from input space to higher dimensional feature space. However, research has found that best results are usually obtained with the Radial Basis Function (RBF) kernel. The RBF kernel is defined in the Equation below.

$$K(x_i, x_j) = \exp\left(-\frac{\|x_i - x_j\|^2}{2\sigma^2}\right)$$

Unlike this beautifully well-defined algorithm, neural networks and deep neural networks are not always considered to be so well founded in a theory. Instead, over the years they have been considered as a type of technique with a black box framework. Because of this, many practitioners have at times disregarded neural networks and deep neural networks. However, since the early 2000s, deep neural networks have started to defeat all other techniques on challenge after challenge across the machine learning landscape. As such, both academia and industry have noticed and great interest has developed for these techniques.

Similarly to the previous algorithms, SVM has the same structure as others in SKlearn. It includes a parameter kernel which can be used to set kernels such as linear or rbf. The rbf kernel requires the parameters of cost and gamma. These values are usually data dependent and require parameter tuning.

```
def svm_rc(X_train_normalized, y_train, X_test_normalized, y_test):  
    from sklearn.svm import SVC  
    #svm = SVC(kernel='linear', C=1.0, random_state=0)  
    svm = SVC(kernel='rbf', random_state=0, gamma=0.0010, C=32)  
    svm.fit(X_train_normalized, y_train)  
    y_pred = svm.predict(X_test_normalized)  
    print_stats_percentage_train_test("svm (rbf)", y_test, y_pred)
```

## 2.8 The t-test for Machine Learning

Sometimes we want to compare between several machine learning algorithms using the previously discussed metrics (precision, recall, f-measure). The standard approach is to run 10-fold cross validation and look at the results. However, a more robust approach is to use a t-test. By definition, the t-test compares the mean or

averages of 2 populations to determine how different the populations are from each other.

## **2.9 Summary**

In this chapter, an overview of some of the main traditional topics of supervised machine learning was provided. In particular, the following machine learning algorithms were presented: logistic regression, KNN, Support Vector Machines, and neural networks. Code examples of their implementation using the SKlearn kit were presented and discussed. Additionally, issues related to classifier performance were also addressed. The next chapter will focus on issues related to data and data pre-processing that apply to both traditional machine learning as well as deep learning.

## CHAPTER 3: DATA LOADING AND PREPROCESSING

In this chapter, I will address the very important issue of dealing with the data. To me and most practitioners, data collection is the most important issue in machine learning. There are many aspects that must be addressed when dealing with data. These include:

- getting the data
- cleaning data
- pre-processing data
- building a corpus and annotating it
- performing inter-annotator agreement
- etc.

Here in this chapter I will also address general issues about data as well as data issues related to deep learning such as one-hot encoding.

### 3.1 Loading the Data

Data can be obtained from the web such as text from twitter or web pages. Specific data sets can also be obtained from the machine learning libraries. Sklearn has a

“dataset” module. This dataset module can be used to obtain certain data sets such as the iris dataset. An example of this code can be seen below.

```
iris = datasets.load_iris()
X = iris.data[:, [1,2,3]]
y = iris.target
```

Here, the first index in the data matrix represents the rows and the second index represents the columns. Data can also be obtained from text files. Many practitioners and academics will have their own data in text files. This data can be formatted in many different ways and loaded into the code. In most of the examples in this book, the data is assumed to be formatted in csv (comma separated) format. The code to load the data to both SKlearn based traditional machine learning algorithms and to Tensorflow code files is shown below. In the code we can see that we are using the **numpy** library (or namespace) to obtain the data. We assume the data is stored in the file [data/12559\\_Training\\_Dataset.csv](#) and is read by **loadtxt()** into the python variable **Matrix\_data**. Since we are using csv format, the data file can look like the following.

```
1.0, 6.1, 2.8, 4.7, 1.2
0.0, 5.7, 3.8, 1.7, 0.3
2.0, 7.7, 2.6, 6.9, 2.3
1.0, 6.0, 2.9, 4.5, 1.5
1.0, 6.8, 2.8, 4.8, 1.4
0.0, 5.4, 3.4, 1.5, 0.4
...
```

Once the data is in **Matrix\_data**, it can be processed as a numpy array matrix. This means that it is no longer just an array but instead it is more a vector or matrix as

in linear algebra. Many operations are now simplified like extracting certain columns or rows. This is usually referred to as slicing.

```
f_numpy = open("data/12559_Training_Dataset.csv", 'r')
Matrix_data = numpy.loadtxt(f_numpy, delimiter=",", skiprows=1)

A = len(Matrix_data[0,:])
print "num features,", A
#X=Matrix_data[:, [1,2,3,4,5,6]]
X = Matrix_data[:, :18] #[:, :149]
y = Matrix_data[:, 19]
```

In the previous code we can see that we can calculate the dimensions of the matrix as in `A = len(Matrix_data[0,:])` which gives you the number of columns or the number of features plus the class.

The following code example shows how data can be read with `np.loadtxt()` and sliced from the matrix **Test\_data** into 2 matrices **X\_test** and **y\_test**. Notice that **X\_test** has all rows from **Test\_data** and columns from 0 to 18. Whereas, the **y\_test** matrix has the same number of rows but only 1 column (19).

```
f_test = open("data/rc_3156_Test_19_features.csv", 'r')
Test_data = np.loadtxt(f_test, delimiter=",", skiprows=1)
X_test = Test_data[:, :18]
y_test = Test_data[:, 19]
```

## 3.2 Data and Feature Pre-Processing

Feature scaling is very important to achieving good results in classification tasks. For example, in Principal Component Analysis (PCA) which is a feature reduction



technique, feature scaling is very important. The purpose of PCA is to project data to a vector that captures the most variability in the data. If the features are not scaled properly, one feature could dominate over the others and therefore be considered as the most variable feature.

With feature scaling, features with real valued numbers from any range can be mapped to other ranges such as from -1.0 to 1.0. This is performed for all features so that no one feature will dominate in the model.

Other classifiers and Deep learning models are also susceptible to feature scaling. The code below shows how the data can be scaled from **X\_train** to **X\_train\_normalized**.

```
## feature scaling
sc = StandardScaler()
sc.fit(X_train)
X_train_normalized = sc.transform(X_train)
X_test_normalized = sc.transform(X_test)
```

For a lot of the discussion in this book, I will refer to the iris data set. You can usually read this data directly from the web and load it to your Tensorflow code. However, since your goal may be to analyze your own data, I thought it a good idea to assume that you will want to read your data from a csv file. Therefore, in the following code I am showing you a general way in which you can take a standard dataset such as iris or mnist and save it to a csv file.

This approach allows you to do the deep learning modeling by reading data from text files where the data is formatted in a very standard and well know format such as csv (comma separated format).

```
## create csv files from mnist or iris
def buildDataFromMnist(data_set):
    #iris = datasets.load_iris()
    X_train, X_test, y_train, y_test =
        train_test_split(data_set.data,
                        data_set.target, test_size=0.30, random_state=42)
    f=open('2.0_training_mnist.csv','w')
    for i,j in enumerate(X_train):
        k=np.append(np.array( y_train[i]), j )
        f.write(",".join([str(s) for s in k]) + '\n')
    f.close()
    f=open('2.0_testing_mnist.csv','w')
    for i,j in enumerate(X_test):
        k=np.append(np.array( y_test[i]), j )
        f.write(",".join([str(s) for s in k]) + '\n')
    f.close()
```

### 3.3 One Hot Encoding

Algorithm implementations in Tensorflow use one-hot encoding. Quite simply, one hot encoding means that you take labels in the following format.

0
1
1
2
0
1
1
2
1
...

and convert them to labels in the equivalent one-hot encoded format as shown below. So, we create new vectors where all values are zero except for the value of the position of the correct class in the vector.

1,0,0
0,1,0
0,1,0
0,0,1
0,0,1
1,0,0
0,1,0
0,1,0
0,1,0
0,0,1
0,1,0
...

One-hot encoding transforms the labels vector ( $y$ ) of size  $n$  into a matrix of size  $n$  by  $B$  where  $B$  represents the number of classes in the data set. For the case of the iris dataset, we have 3 classes and, therefore, for a sample with label 2 we would get an equivalent one-hot encoded vector equal to  $[0,0,1]$ .

The code to convert the data to one-hot encoded format is provided below. There are several ways of implementing one-hot encoding. I have used the approach below because I think it is the easiest to understand (while possibly not the most efficient or pythonic).

Notice how “a” is a 2-dimensional array initialized with all zeros. You use the value in the input vector **data** to determine the index **i** and **j** that are used to assign a 1 to the correct position in the matrix.

```
# Convert to one hot data
def convertOneHot_data2(data):
    y=np.array([int(i) for i in data])
    #print y[:20]
    rows = len(y)
    columns = y.max() + 1
    a = np.zeros(shape=(rows,columns))
    #print a[:20,: ]
    print rows
    print columns
    #rr = raw_input()
    #y_onehot=[0]*len(y)
    for i,j in enumerate(y):
        #y_onehot[i]=np.array([0]*(y.max() + 1) )
        #y_onehot[i][j]=1
        a[i][j]=1
    return (a)
```

### 3.4 Features

A supervised machine learning algorithm is only as good as the features that are provided to it. This statement used to be very true and many people made careers

of just developing features for problems in different domains. For instance, in NLP, many people would spend a lot of time developing parsers and other techniques to find and create features from a text based problem. Similarly, in image processing, researchers developed many techniques to filter data out of images to perform efficient image classification. Today, deep learning has somewhat changed this. It has managed to introduce approaches to decrease the amount of human involvement in the feature extraction process. Basically, deep learning methods, in some cases, have the ability to extract features from data using only un-supervised or semi-supervised techniques. In some way, you can say that deep learning algorithms can extract the features themselves without human involvement. This ability has had a very strong impact in the performance of the algorithms implemented in industry and in the work performed by machine learning specialists. These abilities for enhanced feature extraction are available in the main mediums of text processing and image processing.

#### **3.4.1 Features from Text**

Feature extraction from text usually involves the processing of text documents to extract tokens, words, chunks, or other phrases to be used to create the features. There are many, many types of features that can be extracted from text. Some of the methods (Jurafsky and Martin 2008) that can be used are:

- the bag of words approach
- frequency histograms of the words
- Part of speech tagging of the words
- Syntactic parsing of sentences
- Anaphora resolution

With the information provided by the previous methods, many features can be extracted. In general, text features can be binary or numeric. Binary features

include the presence or absence of words, part of speech tags, chunks, syntactic parses, semantic parses, etc. Numeric based text features can be derived from performing counts or calculating distance metrics between words or higher level semantic concepts.

One technique in particular that has had a lot of success to extract features for supervised machine learning is called the gramulator (McCarthy et al. 2012). The gramulator is a feature extraction technique used particularly in natural language processing. The main idea is that, for a 2 class problem, you want to extract features (e.g. words) that are very frequent in one class but infrequent in the other. This helps to better discriminate between the classes. The downside of this approach, however, is that it needs a lot of annotated or labeled data to extract the grams or words from each class that are infrequent in the opposite class. If the grams are representative of the entire population; then, it can be expected that a classifier will have good performance in the classification task.

The downside of most of these techniques is that in the past they have required annotated data. Deep learning has several new techniques that address this issue and obtain good performance without needing large amounts of annotated data. Some of the techniques will be addressed later in this book.

### **3.4.2 Features from Images**

Feature extraction from images is another area where deep learning is revolutionizing the way features are engineered and obtained. The input to a classifier when performing image processing and classification is an image. In general images are 2 dimensional arrays that contain the pixel intensities that define the image. For color images, you have 3 two-dimensional matrices where each stores the intensity for the R, G, and B colors of the image. In the past, these images were converted into feature vectors for use in machine learning using

image processing based feature extraction techniques. Again, here, researchers spend considerable amounts of time performing feature engineering. Some of the methods widely used in image processing (Gonzales and Woods) include:

- Image segmentation
- Region growing
- Image morphing
- Fourier transformations
- Etc.

Many features could be extracted from these methods. Generally speaking, many of these techniques are implemented as a filter that is applied to the images to convert them into another matrix of the same size or of another size. This new image will store new information about the image such as the pixel positions where an edge was detected. Here, once again, deep learning is changing how this process is done. Deep neural networks can be used to discover the features simply by providing the inputs and some annotation. This information alone allows the neural network architecture to connect the neurons in ways where it can discover features by itself. In fact, deep neural networks are not just used for classification but can also be used just for feature extraction. The extracted features can then be used with other classifiers such as SVM, logistic regression, etc. In these cases, the input layer and hidden layers are used for feature extraction without having to use the output layer for classification. Convolutional Neural Networks (CNNs) can provide this functionality in image processing and classification

### **3.5 Corpora**

A supervised machine learning algorithm is only as good as the data that is provided to it. Learning models learn how human annotators assign the labels to a sample and a system can only be expected to be as good as the human annotator.

Some tasks are more subjective than others for human annotators and this is usually reflected in the classifier performance. Therefore, many practitioners recommend that before measuring classifier performance, an analysis of the subjectivity of the human annotation should be performed. This is usually referred to as inter-annotator agreement.

When annotating a new resource, the quality of the annotation process must be measured in some way. Inter-annotator metrics refer to techniques used to measure the overall agreement between the annotations of two or more individuals. Important metrics used to evaluate inter-annotator agreement (Artstein and Poesio 2008) include:

- average observed agreement
- Pi
- Alpha
- S
- Kappa

These metrics differ in how they correct for expected chance agreement. Expected chance agreement is a probability that 2 annotators will agree on their annotation for an item by chance. This probability depends on the number of classes. Formally, this probability is calculated as follows:

$$A_e = \sum_{k \in K} P(k|c_1) \cdot P(k|c_2)$$



where  $c_i$  is the annotator  $i$ , and  $k$  is the assigned category. Inter-annotator agreement metrics are important because they help to set theoretical boundaries on the accuracy that a given machine learning methodology can achieve using the annotated corpora (Bird et al. 2009). A brief description of some of the techniques is provided in the following discussion.

- **Average Observed Agreement ( $A_o$ )**

Averaged observed agreement is the easiest metric to compute. It is the percentage of annotations that two annotators agreed upon. The metric is formulated as follows where the variable “samples” represents the total number of annotation samples and “agreed” is the amount of samples for which both annotators agreed.

$$A_o = \frac{1}{samples} \sum agreed$$

- **Chance-corrected Metrics ( $A_{corr}$ )**

Chance corrected metrics are those that take into account the expected chance agreement  $A_e$ . Once chance agreement is defined, the metric can be corrected. These types of metrics include: s, alpha, and kappa. Formally, the main concept in these metrics is defined as follows:

$$A_{corr} = \frac{A_o - A_e}{1 - A_e}$$

The following code shows how to perform this analysis using the well known nltk ([www.nltk.org](http://www.nltk.org)) framework.

```
#interannotator agreement

import nltk
toy_data = [
    ['1', 5723, 'ORG'],
    ['2', 5723, 'ORG'],
    ['1', 55829, 'LOC'],
    ['2', 55829, 'LOC'],
    ['1', 259742, 'PER'],
    ['2', 259742, 'LOC'],
    ['1', 269340, 'PER'],
    ['2', 269340, 'LOC']
]

task = nltk.metrics.agreement.AnnotationTask(data=toy_data)

print 'kappa', task.kappa()
print 'alpha', task.alpha()
print 'average Agreement', task.avg_Ao()
print 'pi', task.pi()
print 's', task.S()

print '#####'
print '#####'

toy1 = ['ORG', 'LOC', 'PER', 'PER']
toy2 = ['ORG', 'LOC', 'LOC', 'LOC']
cm = nltk.metrics.ConfusionMatrix(toy1, toy2)
print cm
```

More details about data collection, corpora development, and web scraping can be found from the following sources: Web Scraping with Python by Ryan Mitchell, Calix and Knapp (2011), and Natural Language Processing with Python by Bird, Klein, and Loper.

### **3.6 Summary**

In this chapter, a discussion about issues related to data and data pre-processing was provided. In particular, the following topics were addressed: data pre-processing, reading data from csv files, types of features, corpora processing, and one-hot encoding. In the next chapter, we will begin our discussion of how to program out first deep learning classifier.

## CHAPTER 4: DEEP LEARNING

In this chapter, I will introduce the topic of deep learning. My goal here is to help students and practitioners alike to get started with deep learning. In particular, I want them to be able to get data and pre-process it for deep learning, load it on their programs, perform classification tasks, and evaluate their results. I will discuss some aspects of the theory providing as much intuition as possible. For this task, I will use Linux, Python, Sklearn, and the Tensorflow framework. I will also discuss some aspects of using GPU based hardware by the end of the book.

For most of my sample code, I used Ubuntu Linux with conda. I installed the Tensorflow framework following the instructions in the main Tensorflow website (<https://www.tensorflow.org/>).

### 4.1 Things to know about the code

As in the previous section with sklearn, I have included the libraries first. Notice, that I use as many sklearn libraries as possible and only use the Tensorflow module when absolutely necessary. The goal, again, is to help the reader to best capture the Tensorflow concepts by simplification. Once your skills improve with deep learning frameworks, you will be able to more easily replace sklearn modules with alternative Tensorflow modules.

```
import tensorflow as tf
import numpy as np
from numpy import genfromtxt
from sklearn import datasets
from sklearn.datasets import fetch_mldata
from sklearn.cross_validation import train_test_split
import sklearn
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score, f1_score
import pandas as pd
import matplotlib.pyplot as plt
```

In the previous code section, it can be seen that for our deep learning code to work, we only need a few new libraries. The most important new library is **Tensorflow**. This of course is the main source of our new code. We will still rely on **numpy** to efficiently get the data although we can also get data with the statement **sklearn.datasets import fetch\_mldata**.

The next important issue in our code to do would be to set some parameters. Deep learning algorithms are iterative in the sense that they load samples in batches to avoid running out of memory. This is very important as it allows you to load millions of data samples in subsets called batches. Therefore, we can set up the parameters for batch processing. In the code below, we can see that we want to load batches of 100 samples each. For a training set of 1,000 samples, we would divide the set into 10 bins of 100 samples each and therefore run 10 batches.

Two additional parameters in this code are the **learning\_rate** and the **n\_epochs**. The **n\_epochs** parameter represents the number of times that we will run our main “for” loop. This is the loop that we will run to provide data to our algorithms for training and testing. At this point, an optimization begins to occur which helps the supervised learning algorithm to learn from the samples.

The **learning\_rate** value (in this case 0.01) is a very important parameter in the learning algorithm. Simply speaking, it represents the step that is taken in a gradient descent algorithm to find an optimal solution. Think of this like a giant walking over a mountainous region with many peaks and valleys. He is trying to find an optimal location; if the step is too big, the giant could go from peak to peak and skip a valley altogether. On the other hand, if the step is too small the giant may take too long to move in or out of a valley. In terms of Tensorflow, the learning rate can affect convergence.

Convergence is the term that means that the iterative algorithm is starting to tend to the optimal solution. The **learning\_rate** can also be assigned a function instead of just a value.

```
## a smarter learning rate for a gradient descent optimizer
learningRate = tf.train.exponential_decay(learning_rate=0.0008,
                                          global_step=1,
                                          decay_steps=trainX.shape[0],
                                          decay_rate=0.95,
                                          staircase=True)
```

This helps to adjust the learning rate in real time while the algorithm is running. Some researchers have suggested that this type of optimization can provide better results than just using a fixed value learning rate as in the example below.

```
#parameters
learning_rate = 0.01
n_epochs = 27000 #1000
batch_size = 100
```

Deep learning algorithms in Tensorflow consist of a basic structure. Simply put, they have:

- the main loop
- the initialization section
- the declaration of the main variables or placeholders
- and the definition of the classifier to use.

In the next section I will proceed to further elaborate on this.

The figure below presents the accuracy plot per epoch of a deep neural net as it is being trained. The **x** axis represents the epochs over time and the **y** axis represents the accuracy score. As can be seen, the accuracy is low at first but it starts to improve after a certain number of epochs when the classifier begins to learn and converge.

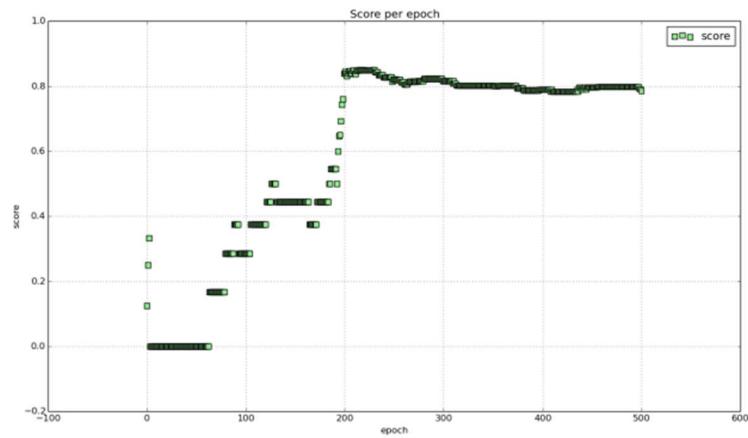


Figure. Accuracy per epoch

In this example on the figure, a deep net takes a few epochs to learn how to detect and classify the samples. After about 200 epochs, a deep net of say 2 hidden layers stabilizes and is able to maintain a consistent accuracy score.

## 4.2 Getting Started with Deep Learning

So what's it all about? Again, my focus here is not to discuss deep learning from the equations point of view or talk about the derivatives. Those are all very important concepts but can be overwhelming when starting. Instead, I want to help the reader write deep learning code with Tensorflow. Then, the reader will, no doubt, have many questions where the theory and fundamentals will help him or her to better understand and use the algorithms.



The figures below present some of the challenges that must be addressed when dealing with real data. Real data such as language data from Twitter is highly noisy and un-balanced. An imbalance in the data means, for example, that 80% of the samples belong to 1 class and the remaining 20% belong to the other class. Therefore, training a classifier to, say, perform emotion classification (where data is highly imbalanced) can be a difficult task. In the figure below, a sample of Twitter data is represented. Each sample in the dataset had multiple features but was compressed into a two dimensional vector for visualization purposes.

As can be seen, the data in the figures has a high overlap in the classes and the data is difficult to separate. A linear classifier (the line seen on the graph) can only separate a small portion of the samples and the majority are not easy to separate in this space. The graph below is available on-line in color.

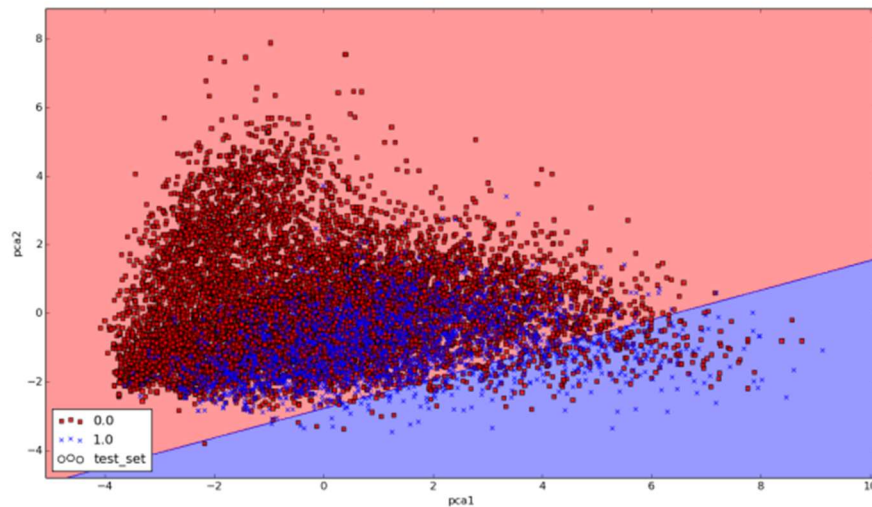


Figure. Logistic Regression Classifier

So how can we get more of the samples of one class without getting too many of the samples of the other class? Well, the answer is that we could use another type of line for the separation. For instance, instead of using a straight line we could use a curved line. This is why some algorithms are called linear and others are called non-linear. Non-linear algorithms can sometimes create better separations in the data and therefore obtain fewer errors. In the next Figure, an example of this is shown using Support Vector Machines (SVM). SVM methods can sometimes build better non-linear classifiers because of their ability to project data to higher dimensional spaces. Deep neural networks can be used to learn non-linear models as well. This property of non-linear models can help to obtain better classification accuracy scores. This figure is available on-line in color.

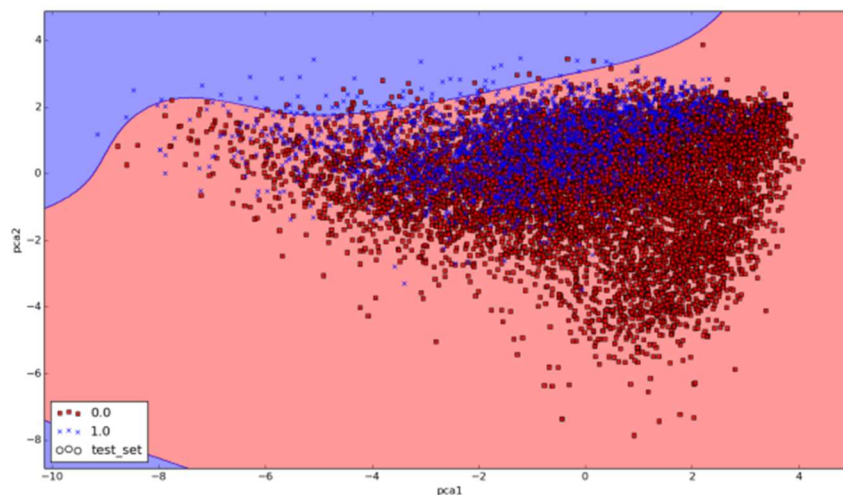


Figure. SVM Classifier

The figure above shows an SVM classifier building a non-linear separation line (the curved line in the graph) on the data. It could be expected that a deep neural

network could build even better separation boundaries and that different architectures could give different results. This takes us to the very important aspect of deep learning architecture.

Deep learning architecture is where we define the parameters such as layers and neurons that define a deep neural network. You could think of this as the way that you construct the line that you want to build and use. We will discuss this in the next sections but it is important to know that you can define many deep learning architectures.

### **4.3 Deep Learning Definition**

Deep learning systems are neural networks with many layers. As such, the more layers they have the deeper they are considered to be. In the next sections, I will begin to discuss neural networks. I will focus mostly on intuition when talking about neural networks. I will use linear regression and logistic regression constructs to define them. We can also think of deep neural nets as functions and this will become very useful as we start to program our code. Additional discussions of deep learning theory can be found in many other books such as “Fundamentals of Deep Learning: Designing Next-Generation Machine Intelligence” by Nikhil Buduma and Nicholas Locascio and Deep Learning by Goodfellow et al. (2016).

### **4.4 Tensorflow Basics**

The main idea with the implementation of deep neural networks in Tensorflow is that you need to define a graph and run your code through this graph on the CPU (s) or GPU (s).

Here are some of the main aspects of Tensorflow to be aware of:

- Tensorflow is object oriented
- Tensors are multidimensional arrays and Tensorflow variables are memory buffers that contain these tensors
- We call elements from Tensorflow by referencing the object **tf** as such:  
tf.placeholder
- In Tensorflow you need to define a graph and then run it. Nothing runs until you call the graph with an object called a session
- Tensorflow get its name from the fact that data is stored as tensors and flows through a graph

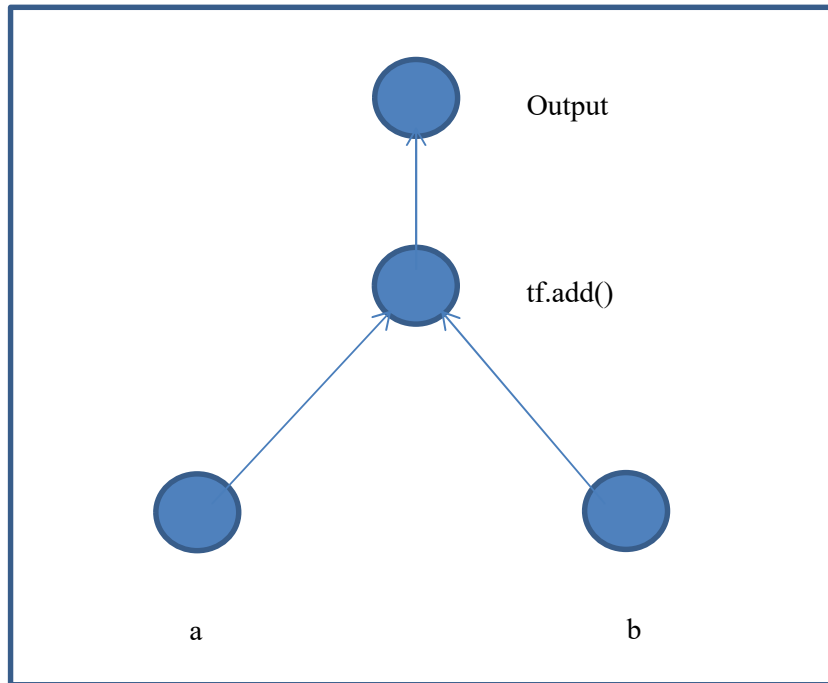
The first piece of Tensorflow code you are likely to see in many books and the web is the following. This piece of code has become the “defacto” hello world script. So, let us also use it to begin.

```
import tensorflow as tf
hello = tf.constant('Hello, TensorFlow!')
sess = tf.Session()
print(sess.run(hello))

a = tf.constant(10)
b = tf.constant(32)
print(sess.run(    tf.add(a, b)    ))
```

In this code we initialize 2 variables **a** and **b** and add them together. Additionally, we initialize the session object with **sess = tf.Session()**. What is important to note here is that whenever you reference an object in Tensorflow with **tf**, such as initializing the variable **a**, you are actually adding elements or nodes to a graph.

This graph does not get executed until the command `sess.run()` is called. The graph can be seen in the following figure.



**Figure. Tensorflow graph.**

In this case, the command adds the 2 numbers together. The answer for this after printing the results should be 42.

## 4.5 Loading your Data into your Tensorflow code

Loading the data can actually be somewhat complicated in Tensorflow. One of the issues is that you need to convert the data, say from something like the below

example, to another format using one-hot encoding for the labels. This can sometimes be complicated and can cause you problems when writing your algorithms.

```
class,f1,f2,f3,f4
1.0,6.1,2.8,4.7,1.2
0.0,5.7,3.8,1.7,0.3
2.0,7.7,2.6,6.9,2.3
1.0,6.0,2.9,4.5,1.5
1.0,6.8,2.8,4.8,1.4
0.0,5.4,3.4,1.5,0.4
1.0,5.6,2.9,3.6,1.3
2.0,6.9,3.1,5.1,2.3
1.0,6.2,2.2,4.5,1.5
1.0,5.8,2.7,3.9,1.2
```

Here, we are looking at the iris data set. The first column represents the annotated class. The IRIS dataset has 3 classes (0, 1, and 2). The other 4 columns represent the features per each sample. The code provided below will show how to read the data.

In the code below, we assume that the data is contained in the file **iris.csv** (available at the book github). We can use the **numpy.loadtxt()** function to obtain the data and to load it into a numpy matrix. Notice that the file has a header for the class and the 4 features. We can easily remove the header by using the parameter **skiprows=1** in the **numpy.loadtxt()** function.

```
f_numpy = open("data/iris.csv",'r')
Matrix_data = numpy.loadtxt(f_numpy, delimiter=",", skiprows=1)
#X=Matrix_data[:, [1,2,3,5,6]]
X = Matrix_data[:,1:4]
y = Matrix_data[:,0]
```

In the previous code we can slice the class column into the **y** variable and the rest of the columns into the **X** variable. Notice that we are following the SKlearn approach from the previous chapters. Tensorflow has its own approaches but I have used **SKlearn** here for the sake of simplicity. When slicing a 2-D matrix (matrix[**i**, **j**] ) we specify the number of rows with “**i**” and the number of columns with “**j**”. We can also specify ranges by doing the following: matrix[2:5, :]. The “:” on the “**j**” index indicates select all columns. If we wanted to select from a list of column indices we could do the following: `x=Matrix_data[:, [1,2,3,5,6]]`.

```
x_train, x_test, y_train, y_test = train_test_split(X, y
                                                    test_size=0.20, random_state=42)
```

Similarly to the previous chapters, we can now split the data from **X** and **y** to obtain the train and test sets: `x_train`, `x_test`, `y_train`, `y_test`. As can be seen in the previous code segment, these data sets are now ready for one-hot encoding. Only the labels need to be one-hot encoded. The code for performing the one hot-encoding step is below. The one-hot encoding function and process was defined in the previous chapter.

```
y_train_onehot = convertOneHot_data2(y_train)
y_test_onehot = convertOneHot_data2(y_test)
#print y_train_onehot[:20,:]
```

In the code above we take the labels in **y\_train** and **y\_test** and pass them through our one-hot encoding function to produce the new variables **y\_train\_onehot** and

**y\_test\_onehot**. You can visualize the new one-hot encoded vectors by running `print y_train_onehot[:20,:]`.

Tensorflow does provide a function for one-hot encoding which is:

**tf.one\_hot()**

This function takes a **y** vector and converts it to the one-hot encoded version. For example:

```
>>>indices = [0, 1, 2]
>>>depth = 3
>>>print tf.one_hot(indices, depth)

## the output is

[ [ 1, 0, 0],
  [ 0, 1, 0],
  [ 0, 0, 1] ]
```

Now, try this one and see what it gives you.

```
>>> y = tf.one_hot(indices, depth)
>>> depth = 4
>>>indices = [0, 3]
>>>print sess.run(y)
```

The next step in the process is to scale the **X** data. I suggest running your data with scaling and without to compare the performance of your classifier. Convergence of your deep learning algorithm and classification results can be better with scaled data. In this case we can use SKlearn's scaling module **StandardScaler()** to perform this task.



```
## feature scaling
sc = StandardScaler()
sc.fit(X_train)
X_train_normalized = sc.transform(X_train)
X_test_normalized = sc.transform(X_test)
```

As can be seen in the above code segment, we only need to scale the **X** data. We can fit a model with the train set and then use that model to scale both the **X\_train** and **X\_test** sets.

Now that our data is ready, we want to save some information about the dimensions of the data sets in a couple of variables **A** and **B**. This can be very helpful for code re-usability. We will store the number of features in **A** and the number of classes in **B**.

To obtain these values, we can use the **.shape[z]** function available to numpy arrays. In the code below we can see an example of how to get this value for **A**. In this case, “**z**” represents the matrix dimension (number 0 for rows and number 1 for columns).

```

# features (A) and classes (B)
# A number of features
# B = number of classes, 10 numbers for mnist
(0,1,2,3,4,5,6,7,8,9), 4 numbers for iris
A = X_train_normalized.shape[1]    #num features
B = y_train_onehot.shape[1]        #num classes
samples_in_train = X_train_normalized.shape[0]
samples_in_test = X_test_normalized.shape[0]
print "num features", A
print "num classes", B
print "num samples train", samples_in_train
print "num samples test", samples_in_test
print "press enter"
rr = raw_input()

```

At this point, we have completed most of the pre-processing steps and we are ready to define the learning algorithms. This is where we define neural network architecture, the cost functions to be used, and the functions to predict results for given test samples. In the next section, I will implement a linear regression algorithm in Tensorflow. After that, I will extend the linear regression framework to build a logistic regression model and later a deep architecture neural network model.

## 4.6 Linear Regression in Tensorflow

Linear regression refers to a model for predicting any kind of magnitude (such as housing prices, sales prices, ages, etc.) instead of just classes (e.g. happy vs. sad,

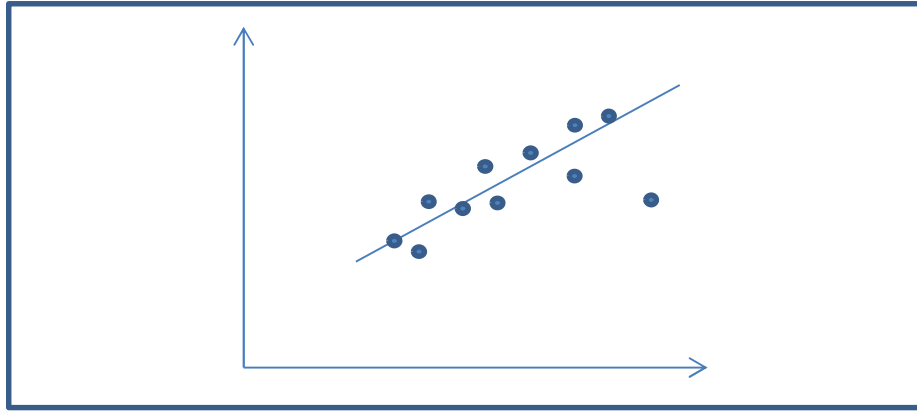
etc.). In linear regression, multiple variables and coefficients are combined to form an equation that can be used to fit a particular data set. The fitting process involves an optimization approach that minimizes the Least Squares Error.

The coefficients or weights for each variable are useful in determining the contribution of each variable to the fitted model of the data. The definition of the least squares error function can be seen in the function below

$$E = \sum_{i=1}^n (y_i - (wx_i + b))^2$$

Where **n** represents the number of samples.

In linear regression, a line is fitted to the dataset in feature space by minimizing the sum of errors. That is to say, the sum of the error distances between the values in the predicted line and the real values are estimated (see figure below).



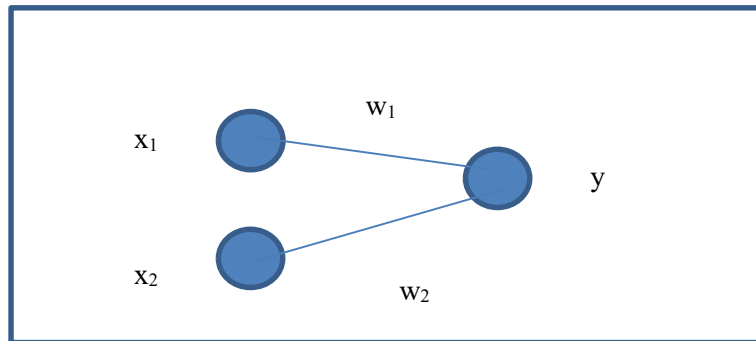
**Figure. Linear Regression Line.**

In the equation below, the term  $\langle \bullet, \bullet \rangle$  denotes the dot product in the input space between  $\mathbf{w}$  and  $\mathbf{x}$ . Here,  $\mathbf{x}$  is the input vector,  $\mathbf{w}$  is the weight vector, and  $\mathbf{b}$  is the bias.

$$f(x) = \langle w, x \rangle + b$$

This simple equation defines the linear regression and the parameters that must be learned given the training data. With this theoretical framework I will now proceed to describe how to code this with Tensorflow.

The linear regression model is implemented in the code below. The purpose of the code below is to be simple and to show how a linear regression algorithm can be implemented in python using Tensorflow.



**Figure. Linear Regression Architecture.**

For our example, we are going to assume a very simple regression problem for housing prices. We are going to have 1 input (1 feature  $x_1$ ) in our samples (the size of the house) and we will predict 1 output value  $y$  (the cost of the house). Therefore, we will have an equation as such:

$$y = w_1 * x_1 + b$$

We will need to solve this equation by learning the values of  $w_1$  and  $b$  (the parameters) given the training data in  $x_1$  and  $y$ . For an equation with 2 input features such as

$$y = w_1 * x_1 + w_2 * x_2 + b$$

we can show it in diagram form as can be seen in the figure.

As shown in the previous equation, a linear regression is an equation made up of 3 main elements:  $x$ ,  $w$ , and  $b$ . The  $x$  value holds the training data (features),  $w$  represents the weights per feature, and  $b$  is the offset value (bias) learned by the

model. These values are directly coded into our algorithm using Tensorflow. In the code below, it can be seen that after the library declarations, 4 variables have been declared.

They are:

- **x**
- **y\_**
- **W**
- **b**

The variables **x** and **y\_** are defined with Tensorflow placeholders. These are used to store our data whether training or testing. So, the data you read from your .csv or other source files goes here.

The syntax to declare a variable to hold the data is as follows:

```
x = tf.placeholder(tf.float32, [None, 1]).
```

Notice that you declare the data type (**tf.float32**), and you define the dimensions of the **x** variable. In this case, “None” and 1 are the dimensions of the feature data for our toy linear regression model. The number 1 means that this data has only 1 feature or column ( $x_1$ ). The “None” parameter indicates that the number of rows can vary. For this example, **x** is the feature data and **y\_** holds the labels for each sample in **x**.

The other 2 variables, **W** and **b**, are variables that define the prediction equation. They are the parameters to be learned. All examples in this book will use these variables whether we are defining logistic regression or deep neural nets. **W** and **b** are matrices (tensors) whose dimensions need to be defined. This is where we start to define the architecture of our models. In this case, the dimensions are [1,1] for

W and just [1] for b. Notice that we initialize these matrices with the Tensorflow construct `tf.zeros()`. In the case of W the dimension is [1, 1] because it is for a model with 1 feature ( $x_1$ ) and 1 output value (the predicted housing price y). Think of W as a matrix that connects one layer to the next. In this case, we are connecting the inputs to the outputs.

```
import numpy as np
import Tensorflow as tf

#####
x = tf.placeholder(tf.float32, [None, 1])
#####

init = tf.initialize_all_variables()
sess = tf.Session()
sess.run(init)
#####

steps = 100
for i in range(steps):
    xs = np.array([[i]]) #house size
    ys = np.array([[5*i]]) #house price
    feed = {x:xs, y_:ys}
    sess.run(train_step, feed_dict=feed)
    print("After %d iteration: " % i)
for i in range(100,200):
    xs_test = np.array([[i]]) #house size
    #print xs_test
    feed_test = {x:xs_test}
    result = sess.run(eval_op, feed_dict=feed_test)
    print "Run {},{}".format(i, result)
```

In the previous code, the statement

```
eval_op = tf.reduce_mean(float_val)
```

is used because there is only one value in the tensor. For example `[[996.23]]` and the function `tf.reduce_mean()` returns just the value 996.23 (i.e. without the square brackets).

The function `tf.reduce_mean()` is a built in Tensorflow function that takes as input a tensor and computes the mean of the elements across the dimensions of a given tensor. So, it returns a reduced tensor. For example, given:

```
x = tf.constant( [[1, 1] , [2, 2]] )
```

we can get the following:

```
all = tf.reduce_mean(x)    =>    # 1.5
```

```
all = tf.reduce_mean(x, 0) =>    # [1.5, 1.5]
```

```
all = tf.reduce_mean(x, 1) =>    # [1, 2]
```

To see the results, we can run:

```
print sess.run(all)
```

Okay, so now that we have defined the variables, the next step is to define the equation and define the cost function to learn the parameters (e.g. the weights). To do that, we use the following section of code. This is one of the most important parts of deep learning programming.

I also want to emphasize that the code in the segment below is the section that will vary the most for the different models. That is, this is where we define the specifics of the linear regression model, or the logistic regression model, or the neural network model, or the deep neural network model. In essence, this is the heart or core of it all or is it the brains?



```
y = tf.matmul(x, W) + b

cost = tf.reduce_sum(tf.pow((y_ - y),2))

train_step = tf.train.GradientDescentOptimizer(0.00001).minimize(cost)

correct_prediction = y
float_val = tf.cast(correct_prediction,tf.float32)
prediction_as_float = tf.reduce_mean(float_val)
eval_op = prediction_as_float
```

So, defining the specifics of the code means defining the regression equation and the cost function. Here we use the following statements to accomplish this. These 2 lines of code do all the heavy lifting.

- **y = tf.matmul(x, W) + b**
- **cost = tf.reduce\_sum(tf.pow((y\_ - y),2))**

The first statement **y = tf.matmul(x, W) + b** defines the regression equation. We use **tf.matmul(x, W)** to perform the matrix multiplication and we add **b** to the result. The dimensions of the **b** vector need to match the dimension of the **j** index of the weights vector (weights=[**i**, **j**]). As a side note, I will point out here that Tensorflow is designed to be used with GPUs, and GPUs are very good at matrix multiplications. The function **tf.matmul()** will perform these multiplications. This is the power of Tensorflow. Later in this book, I provide a code example to benchmark your system.

The line **cost = tf.reduce\_sum(tf.pow((y\_ - y),2))** defines the cost function. This is the standard least squares cost function in linear regression. Notice that the

**tf.pow()** function raises the difference between vector **y\_** and vector **y** to the power of 2. This result is passed through the **tf.reduce\_sum()** function and assigned to **cost**.

The function **tf.reduce\_sum()** computes the sum of the elements across dimensions of a tensor. For example:

```
>>>x = tf.constant( [[1, 1, 1], [1, 1, 1]])  
>>>y = tf.reduce_sum(x)  #6  
>>>y = tf.reduce_sum(x, 0)  #[2, 2, 2]  
>>>y = tf.reduce_sum(x, 1)  #[3, 3]
```

Whenever we want to reference the cost function or the regression equation, we can just do so by the variable names we defined for them which are **y** and **cost**, in this case.

Once the cost function is defined, we need to tell the model what optimization to use. In this case, the line:

```
train_step=tf.train.GradientDescentOptimizer(0.00001).minimize (cost)
```

indicates that we should use a gradient descent optimizer with a step size of 0.00001. This object, **train\_step**, is now the optimizer which references the cost function via **cost**. And cost itself is linked to the regression equation via **y**. The final part of this code segment just takes the predicted **y** value and performs typecasting and reducing operations (e.g. if you have a vector of values, average all of them and return 1 averaged value).

Once the inner workings of the model are defined, the next step is to move to the session definition and the main loop. In the code segment below, we can see

that this process involves defining the variables **init** and **sess**. The **init** variable will call:

**tf.initialize\_all\_variables()**

which initializes all the variables and placeholders (the tensors) in the graph once you run the session. Consider that at this point all we have done is define the graph structure without actually running anything in Tensorflow. To run things in Tensorflow we use **sess.run()** with the variable we want to run it with. In this case, we want to initialize the variables so we run **sess.run(init)**.

```
init = tf.initialize_all_variables()
sess = tf.Session()
sess.run(init)
```

Finally, the last part in our linear regression code is to run the main loop to actually train and test the model. In this case, I have broken up the main loop into 2 separate loops for better visualization. The first loop is for training and the second loop is for testing. The first training loop is very simple. The training data, in this case, is not read from a file but instead is created automatically. The feature vector is stored in **xs** and contains only one feature ( $x_1$ ) which is the “i” index. This feature represents the size of the house. The corresponding housing price for each **xs** sample is stored in **ys**.

For this scenario, each housing price value is 5 times its corresponding house size. Therefore, our model after training should be equivalent to:

- $y = 5 * x_1 + b$ .

In this case, the training phase would optimize the weight vector to be equal to 5 ( $W_1=5$ ). Notice that the process is repeated 100 times. Once **xs** and **ys** are defined, these values are assigned to the feed dictionary. This is an internal mechanism of Tensorflow that is very useful for passing the data from the python code to the Tensorflow graph. In this case, **xs** is assigned to **x** and **ys** is assigned to **y\_**. These variables are the same ones that we have already defined in the code.

The last part is to run the graph. Finally, this is where your Tensorflow code will run. Before now, the code had not executed but after **sess.run(train\_step)**, the code finally executes. In this case we run **train\_step** and everything that is linked to **train\_step**. So we run the gradient descent optimizer, the cost function for least squares estimation, and the linear regression equation.

```
steps = 100
for i in range(steps):
    xs = np.array([[i]]) #house size
    ys = np.array([[5*i]]) #house price
    feed = {x:xs, y_:ys}
    sess.run(train_step, feed_dict=feed)
    print("After %d iteration: " % i)
```

The second loop (below) is very similar to the first one except that it is focused on the testing phase. Here we use the model from the training phase to predict the housing prices given the house sizes for the range of values from 100 to 200. Remember that the data is created automatically, for this example. Notice that **sess.run()** now calls **eval\_op**. The variable **eval\_op** is linked to **y** which is basically the linear regression model.

```
for i in range(100,200):  
    xs_test = np.array([[i]])    #house size  
  
    feed_test = {x:xs_test}  
    result = sess.run(eval_op, feed_dict=feed_test)  
    print "Run {},{}".format(i, result)
```

So, here we used the trained model but with a new dataset **xs\_test**. The result is stored in the variable **result** and printed out on the screen. The result of the classification should be that the regression model predicts housing prices that are 5 times the input house size. This outcome can be seen in the output below.

```
Run 100, 500  
Run 101, 505  
...  
Run 200, 1000
```

The previous code for linear regression can be re-written in a more modular form using function calls. This approach will be very helpful later on as most implementations can be written in the same way and the only thing that changes is the internal definition of the functions. This new modular approach can be seen next.

```

import numpy as np
import Tensorflow as tf

#####
def inference(x):
    W = tf.Variable(tf.zeros([1,1]))
    b = tf.Variable(tf.zeros([1]))
    y = tf.matmul(x, W) + b
    return y

#####
def loss(y, y_):
    cost = tf.reduce_sum(tf.pow((y_ - y),2))
    return cost

#####
def training(cost):
    train_step =

tf.train.GradientDescentOptimizer(0.00001).minimize(cost)
    return train_step

#####
def evaluate(y, y_):
    correct_prediction = y
    float_val = tf.cast(correct_prediction,tf.float32)
    prediction_as_float = tf.reduce_mean(float_val)
    return prediction_as_float

#####
x = tf.placeholder(tf.float32, [None, 1])
y_ = tf.placeholder(tf.float32, [None, 1])
y = inference(x)
cost = loss(y, y_ )
train_step = training(cost)
eval_op = evaluate(y, y_)

```

```
#####
init = tf.initialize_all_variables()
sess = tf.Session()
sess.run(init)
#####

steps = 100
for i in range(steps):
    xs = np.array([[i]]) #house size
    ys = np.array([[5*i]]) #house price

    feed = {x:xs, y_:ys}
    sess.run(train_step, feed_dict=feed)

    print("After %d iteration: " % i)
for i in range(100,200):
    xs_test = np.array([[i]]) #house size
    ys_test = np.array([[2*i]]) #house price
    feed_test = {x:xs_test, y_:ys_test}
    result = sess.run(eval_op, feed_dict=feed_test)
    #print sess.run(y)
    print "Run {},{}".format(i, result)
    x_input = raw_input()
```

As can be seen from the previous code example, not much had to change in the code to make it more readable, modular, and easier to modify. The new aspects are that we now have 4 functions that can be used to define the model. They are:

- **inference()** for the equation
- **loss( )** for the cost function
- **training()** for the optimization

- **evaluate()** for the performance estimation

## 4.7 Linear Regression to Logistic Regression and Neural Nets

In this section, we are going to modify and extend our code from the linear regression model so that we can implement logistic regression, 1-layer neural nets, and deeper neural networks of multiple layers. In the following code sections I will make some modifications so that the new models can be implemented easily. We will use the variable names **x\_tf** and **y\_tf** for the feature vectors (**x**) and the labels (**y**).

Additionally, we will use the variables **A** and **B** that we previously described to define the number of neurons per layer. In this case **A** represents the number of features in our data set. For example, the iris data set has 4 features per sample ( $x_i = [f_1, f_2, f_3, f_4]$ ). The variable **x\_tf** is defined as a matrix (tensor) of **[None, A]** or **[None, 4]** for this iris dataset. Similarly, **y\_tf** is defined by **[None, B]** or **[None, 3]** where 3 stands for the number of classes which in this case, for the iris dataset, are setosa, versicolor, and virginica.

It is important to note, that the line `y_p_metrics = tf.argmax(output, 1)`, where 1 is the axis position, will be used for performance evaluation purposes. Its use will be shown in the main loop of the code. Here, the **y\_p\_metrics** variable is declared and assigned the index of the max value from **output** after running it through an `argmax` function to find the index of the max value (e.g. the predicted class).



The function `tf.argmax()` is a Tensorflow function that returns the index of the largest value across the axis of a tensor.

For example,

```
answer = tf.argmax([35, 4, 72, 2])    =>    2
```

you can also define an axis like so

```
answer = tf.argmax( [ [23, 32, 49],  
                     [45,  1, 12] ] )
```

The previous function returns => [2, 0]

```
x_tf = tf.placeholder("float", [None, A]) # Features  
y_tf = tf.placeholder("float", [None, B]) #correct label for x  
## for performance metrics  
y_p_metrics = tf.argmax(output, 1)
```

For the new models, we continue to use the **training()** function that we previously defined. Although very little changed, it is important to mention that the optimizer and learning rate can be changed and that there are many alternatives that could be used. A lot of research has been done on this topic and it is currently an on-going and open research field. Therefore, many optimization approaches exist and many of them are available in Tensorflow.

```
def training(cost):  
    optimizer = tf.train.GradientDescentOptimizer(learning_rate)  
    train_op = optimizer.minimize(cost)  
    return train_op
```

In linear regression we were training to predict any kind of real valued number. From this point on, we are going to try to focus mainly on classifiers and predicting classes. Therefore, in supervised learning, we will usually have the real labels and the predicted labels. Our models will predict the labels and we will need ways of comparing them to the real labels.

The following function is a way to evaluate the actual or real classes per sample (**y\_tf**) against the predicted classes (**output**) per sample in the test set. We compare the two sets of one-hot encoded labels with **tf.equal()** to measure the accuracy.

The **tf.equal()** function returns a vector of boolean values that compares the values in two tensors of equal dimensions.

For example, given:

$$x = [1, 2, 3]$$
$$y = [0, 1, 3]$$

`tf.equal(x, y) ==> [False, False, True] or [0, 0, 1]`

```
def evaluate(output, y_tf):  
    correct_prediction = tf.equal(tf.argmax(output,1),  
                                  tf.argmax(y_tf,1))  
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))  
    return accuracy
```

So, now we are ready to once again use the functions **inference()**, **loss()**, **training()**, and **evaluate()** to define our model. This is where the magic will happen and where the code will be specific to the algorithm and architecture being defined. Since this will vary based on classifier algorithm, each algorithm will be addressed in its own chapter section later. For now, I will simply define how we can call the functions. For example, in the next code section I have the function calls for logistic regression.

```
## for logistic regression  
output = inference(x_tf, A, B)  
cost = loss(output, y_tf)  
train_op = training(cost)  
eval_op = evaluate(output, y_tf)
```

If we were defining a deep neural network (with several layers), we can define the function calls as follows (below). The important aspect is to keep in mind that the

structure of the code is mostly maintained and that the only thing that really changes is the internal definition of the functions **inference()**, **loss()**, **training()**, and **evaluate()**. The only two functions we change to go from a logistic regression model to a deep neural network model are **inference()** and **loss()**.

```
## deep neural network
output = inference_deep(x_tf, A, B) ## for deep NN with 2 hidden
layers
cost = loss_deep(output, y_tf)
train_op = training(cost)
eval_op = evaluate(output, y_tf)
```

In the next code section, the session declaration and initialization are defined. This part is the same as what we defined for the linear regression model and is consistent for both logistic regression and neural networks.

```
# Initialize and run
init = tf.initialize_all_variables()
sess = tf.Session()
sess.run(init)
```

Earlier in this book, I mentioned that one of the advantages of deep neural networks and Tensorflow is that they were designed to process massive amounts of data.

Loading millions of records directly into RAM memory would not be suitable for most big data challenges. Instead, to be more efficient we can load data in batches.

That is, we can take our data set and divide it into bins of size **n** number of samples each.

In the following code section, I will show how this can be done. We need to define the size of each batch (in this case 100 samples) and the size of the train set file. To do that we use the function

```
num_samples_train_set = X_train_normalized.shape[0]
```

which gives us the number of rows in the data set. With these values we can calculate the number of batches with the following statement:

```
num_batches = int( num_samples_train_set/batch_size )
```

```
#batch size is 100
num_samples_train_set = X_train_normalized.shape[0]
num_batches = int(num_samples_train_set/batch_size)
```

We will then use **num\_batches** in the main for loop to read the data from the training set. Here is a snapshot of the entire main loop. It is included so that there is no confusion about the relation between the parts.

I will break this up further in the next page to describe its different aspects in more detail. Notice that we call `sess.run()` twice. Unlike the linear regression example we previously discussed, here we only have 1 main loop (2 were not necessary). We have 2 `sess.run()` calls because one is used for the training and one is used for the testing.

```
final_result = ""
for i in range(n_epochs):
    print "epoch %s out of %s" % (i, n_epochs)
    for batch_n in range(num_batches):
        sta = batch_n*batch_size
        end = sta + batch_size
        sess.run(train_op, feed_dict={x_tf:
                                       X_train_normalized[sta:end,:],
                                       y_tf:
                                       y_train_onehot[sta:end,:]})

    print "-----"
    print "Accuracy score"
    result, y_result_metrics = sess.run([eval_op,
                                         y_p_metrics], feed_dict={x_tf:
                                                                    X_test_normalized,
                                                                    y_tf: y_test_onehot})
    print "Run {},{}".format(i,result)
    y_true = np.argmax(y_test_onehot,1)
    print_stats_metrics(y_true, y_result_metrics)
    if i == 1000:
        plot_metric_per_epoch()
```

The next code section focuses on the training phase in the main loop. This section has 2 nested loops for training. The first one is to train the model for **n\_epochs**.

This allows us to set how many times we want to run the optimization of the model to learn the parameters (i.e. the weight vectors). Within this loop we define a second loop which is for the batch processing of the training data. With the previously calculated value **num\_batches**, we can proceed to read the data in smaller sets (the batches).

Just like we did with linear regression, our goal is to call **train\_op** with **sess.run()** and feed it the train set. We assign **X\_train\_normalized** to **x\_tf** and **y\_train\_onehot** to **y\_tf**. The only difference now is that we feed the data in a loop using our predefined batch dimensions. The dimensions are **sta** for the starting index and **end** for the ending index. In the case of **X\_train\_normalized**, in every iteration, we select the rows from **sta** to **end** (**sta:end**) and all the columns (:).

In **X\_train\_normalized**, the **\_normalized** part of the name indicates that the data has been scaled (see previous discussion). In **y\_train\_onehot**, the **\_onehot** part of the name indicates that the labels have been converted using one-hot encoding. One-hot encoding has been discussed in chapter 3.

```
for i in range(n_epochs):
    print "epoch %s out of %s" % (i, n_epochs)
    for batch_n in range(num_batches):
        sta = batch_n*batch_size
        end = sta + batch_size
        sess.run(train_op, feed_dict={x_tf:
                                        X_train_normalized[sta:end,:],
                                        y_tf:
                                        y_train_onehot[sta:end,:]})
```

The previous for loop will train the model and test it at the same time. In this way you will be able to see how the classification improves as more data is given to the model.

The final part of the loop (below) focuses on the testing and performance evaluation phase. Remember that we are now performing logistic regression based classification or classification based on deep neural networks. So the important aspect is that we will be predicting classes. This part of the code can be seen below.

```
result, y_result_metrics = sess.run( [eval_op, y_p_metrics],
                                     feed_dict={x_tf:
                                                X_test_normalized, y_tf: y_test_onehot})
print "Run {},{}".format(i,result)
y_true = np.argmax(y_test_onehot,1)
print_stats_metrics(y_true, y_result_metrics)
if i == 1000:
    plot_metric_per_epoch()
```

The previous code section provides the definition for the prediction and testing phase. Here, we call **sess.run()** for a pair of values **eval\_op** and **y\_p\_metrics**. These two previously defined values are the ones that can evaluate performance. The difference is that **eval\_op** only gives us the accuracy score stored in result whereas **y\_p\_metrics** can be used to obtain other metrics like precision, recall, and the f-measure.

We take the true test labels in one-hot format (**y\_test\_onehot**), extract the argmax value from the one-hot encoded form, and assign it to **y\_true**.



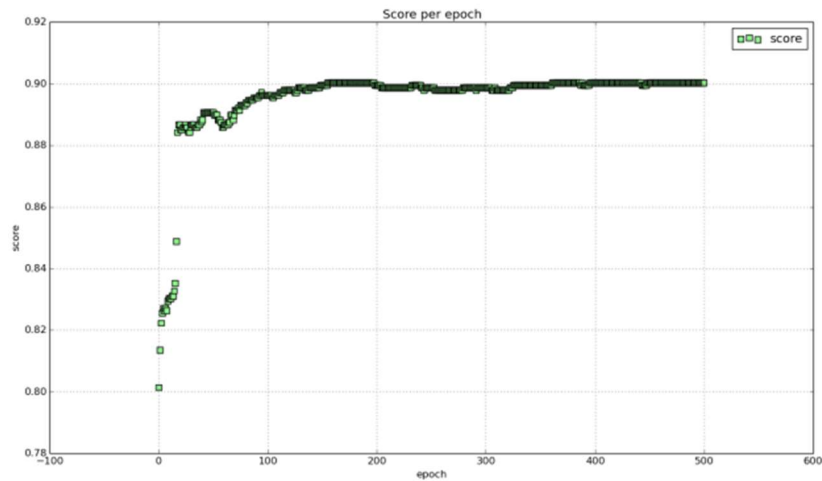
- `y_true = np.argmax(y_test_onehot,1)`

We then pass this value (`y_true`) and the `y_result_metrics` variable to our previously define function `print_stats_metrics()`. This function will print out the statistics just like we saw in chapter 2.

```
def plot_metric_per_epoch():
    x_epochs = []
    y_epochs = []
    for i, val in enumerate(precision_scores_list):
        x_epochs.append(i)
        y_epochs.append(val)

    plt.scatter(x_epochs, y_epochs, s=50, c='lightgreen', marker='s',
               label='score')
    plt.xlabel('epoch')
    plt.ylabel('score')
    plt.title('Score per epoch')
    plt.legend()
    plt.grid()
    plt.show()
```

The final part of the code `plot_metric_per_epoch()` is used to plot the metric value every 1000 epochs. For this to work, you store the metric values in a list such as `precision_scores_list` and then proceed to plot these values using the `matplotlib` library. The output should look like the following figure.



**Figure. Accuracy per epoch example**

Well, great. We have gotten to the end of the code. Now, all that remains is to define the inner workings of the logistic regression algorithms and the deep neural networks. We will do that in the next sections.

## 4.8 Logistic Regression in Tensorflow

Before we look at the deep neural nets, it is a good idea to look at logistic regression. Logistic regression is still a linear classifier but deep neural nets can build on the ideas of logistic regression. So, basically here we will use all the code we have developed so far.

The only difference is that now we will re-define the code for the functions:

- `inference()`
- `loss()`

As we can see in the next code segment, inference is mainly different from the linear regression approach because we now run our equation

```
tf.matmul(x_tf, W) + b
```

through a softmax function. Additionally, instead of predicting a real valued number with just one output neuron, we now have several output neurons which represent each possible class (3 in the case of IRIS, or 10 in the case of MNIST). Intuitively think of it this way. For the linear regression problem we were predicting housing prices given square footage as such:

$$\text{housing\_price} = w1 * \text{square\_footage} + b$$

or

$$y = w1 * x1 + b$$

This formula with values could look like this:

$$\$250,500 = 50 * 5000 + 500$$

The model learned the parameters 50 and 500, and with an input of 5000 square feet can predict a housing price. If you remember, this is a network of 1 input neuron and one output neuron. But the question is, how can we represent a model of several classes? Such as a model for the IRIS data set which has 3 classes?

Well, we know in linear regression we can learn to predict real valued numbers given inputs. From maths we also know that we can run real valued numbers through certain functions to obtain scaled versions of those numbers. In this case a function such as the sigmoid (or the softmax) can do the following:

$$\text{new\_y} = S(y)$$

where

$$y = w * x + b$$

This function will take any value and convert it into a value in the range from 0-1

For example:

$$\$250,500 = 50 * 5000 + 500$$

$$\text{new\_y} = S(\$250,500)$$

$$\text{new\_y} = 0.80$$

So, this allows us to learn functions that predict values between 0 and 1. This is good because these values can also be interpreted as probabilities or strengths of my prediction. As in, I have a 0.80 confidence that the house price is \$250,500. But what if I had 3 possible housing prices: \$150,700, \$250,500, and \$350,400. How can I represent this in the previous equation?

Well, if we built one equation for \$250,000. Then we can definitely learn equations for the other 2 prices as such:

$$\$150,700 = 30 * 5000 + 700$$

$$\$250,500 = 50 * 5000 + 500$$

$$\$350,400 = 70 * 5000 + 400$$

So now if you notice, we have created a system of 3 equations

$$y_1 = w_1 * x + b_1$$

$$y_2 = w_2 * x + b_2$$

$$y_3 = w_3 * x + b_3$$

If you look at this as a network, we can see that it has 3 output neurons and 1 input neuron and, in fact, the network will look like this:

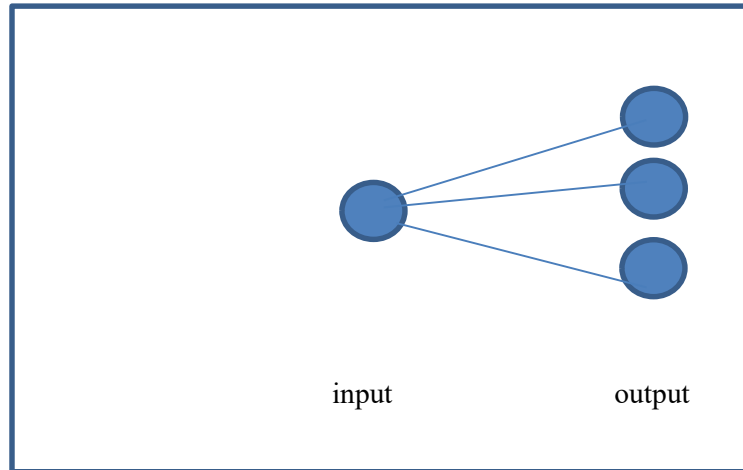


Figure. 3 housing price equations architecture.

If we apply our sigmoid function:

$$\text{new\_y} = S(y)$$

to each equation then we can get:

$$o1 = S(y_1) = w_1 * x + b_1$$

$$o2 = S(y_2) = w_2 * x + b_2$$

$$o3 = S(y_3) = w_3 * x + b_3$$

a new set of outputs which are now scaled to be from 0..1. So the formulas, that looked like this

$$\$150,700 = 30 * 5000 + 700$$

$$\$250,500 = 50 * 5000 + 500$$

$$\$350,400 = 70 * 5000 + 400$$

Can now look like this

$$0.20 = S(30 * 5000 + 700)$$

$$0.70 = S(50 * 5000 + 500)$$

$$0.10 = S(70 * 5000 + 400)$$

So, this intuition hopefully shows you how you can train models to predict confidence of a specific output neuron such as confidence that, given a square footage of 5000, the housing price is more likely to be \$250,000 (based on the 0.70 confidence).

Finally, to train this model you do not need to give it the housing prices but just the output neuron associated with the inputs. In this case our **X** matrix would consist of 1 sample (5000) associated with one output neuron ([0, 1, 0]).

Formally, a softmax function is a way of mapping a vector of real valued numbers in any range into a vector of real valued numbers in the range of zero to 1 (0-1.0) where all the values add up to 1.0. The result of the softmax therefore gives a probability distribution over several classes. The softmax function can be written as follows:

$$S(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$$

This function takes the **y** values as inputs (called logit scores) and produces a new **y** vector where all values add up to 1. For example, it can take

[2.0 , 1.0, 0.1]      and maps it to =>      [0.7, 0.2, 0.1]

In python we can test this as follows:

```
>>> logits = [2.0, 1.0, 0.1 ]
>>> import numpy as np
>>> exps = [ np.exp(i) for i in logits]
>>> sum_of_exps = sum(exps)
>>> softmax = [ j/sum_of_exps for j in exps ]
>>> print sum(softmax)
```

1

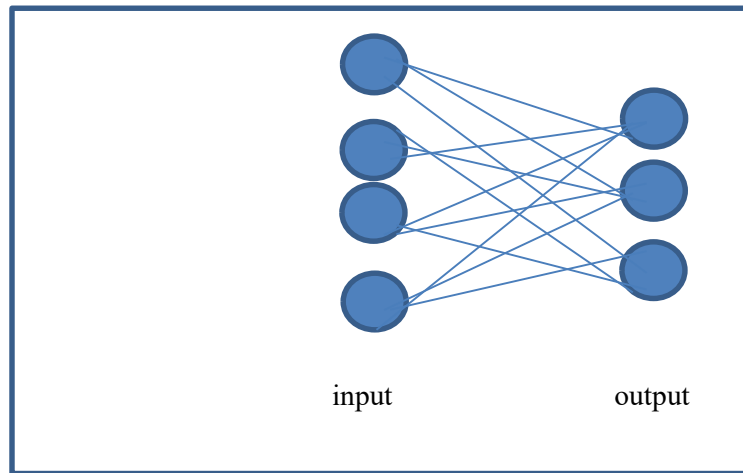
In the next code segment, the inference function for logistic regression is defined.

```
#defines the network architecture for simple logistic regression
def inference(x_tf, A, B):
    init = tf.constant_initializer(value=0)
    #W = tf.Variable(tf.zeros([A,B]))
    W = tf.get_variable("W", [A,B], initializer=init)
    #b = tf.Variable(tf.zeros([B]))
    b = tf.get_variable("b", [B], initializer=init)
    output = tf.nn.softmax(tf.matmul(x_tf, W) + b)
    return output
```

Notice that we can use **tf.get\_variable()** instead of **tf.Variable()** here to declare **W** and **b**. The values **A** and **B** represent, as previously defined, the number of



features and the number of classes. The difference between `tf.Variable()` and `tf.get_variable()` have to do with re-using the same variable. Basically, `tf.Variable()` is older and `tf.get_variable()` is newer and more efficient. The current Tensorflow world suggestion on this issue is to forget about `tf.Variable()` and always use `tf.get_variable()` instead.



**Figure. Logistic Regression Architecture for the Iris dataset.**

So, assuming we are working with the iris dataset, **A** would be equal to 4 and **B** would be equal to 3. The drawing below shows the architecture for a logistic regression model. If you look closely, you can conclude that a logistic regression model is like a neural network with no hidden layers. There are only 2 layers: the input layer and the output layer. A hidden layer would be a layer in between input and output that connects these 2 layers.

So for the previous code segment, in **W** we can see that we create a matrix that is [A,B] in size or for Iris [4,3]. The **b** vector (bias) has dimension **B** equal to 3 (for the 3 output neurons). That is, there is now a bias for every neuron in the output layer. Additionally, notice that the variable **init** in

```
init = tf.constant_initializer(value=0)
```

helps to initialize Tensorflow variables. There are several available options in Tensorflow to do this. Similarly, the loss function is also changed from the linear regression version.

In the linear regression version we used a Least Squares Error loss function. In this case we now use a logistic regression cost function as shown below. See Witten and Frank for details of this loss function. Logistic regression uses a cost function called cross entropy.

$$\text{cost}(g(w^T x), y) = \begin{cases} -\log(g(w^T x)) \rightarrow \text{if } y = 1 \\ -\log(1 - g(w^T x)) \rightarrow \text{if } y = 0 \end{cases}$$

In the previous cost function definition,  $g()$  represents the sigmoid function. We can also write these equations as follows:

$$\begin{array}{ll} -\log(h_0(x)) & \text{if } y=1 \\ -\log(1-h_0(x)) & \text{if } y=0 \end{array}$$

These 2 equations generate the following graphs:

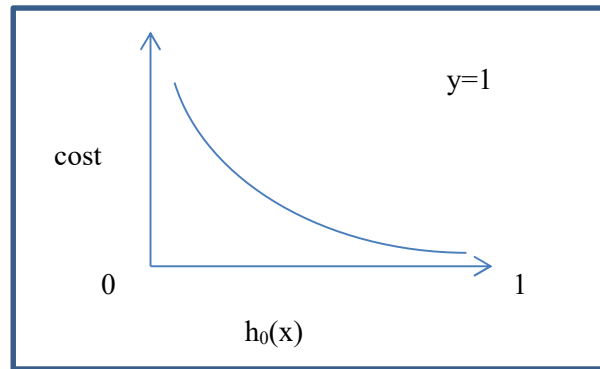


Figure. Logistic regression graphs

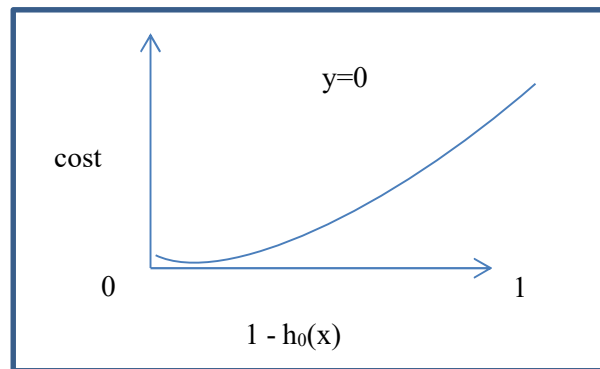


Figure. Logistic regression graphs

For convenience, the previous logistic regression functions can also be written as one function as follows:

$$J(\theta) = -\left(\frac{1}{m}\right) \sum_{i=1}^m [ y \log(h\theta(x)) + (1 - y) \log(1 - h\theta(x)) ]$$

The following code shows the implementation of the loss function for logistic regression as if we were writing out the equations.

One important thing to mention is that I have noticed that this model does not always converge. To correct this, I have added the following line of code

```
output2 = tf.clip_by_value (output, 1e-10, 1.0)
```

This line of code helps because some values in the process become NaN (Not a number) and the clipping in the function addresses the problem. This approach of writing out the cost function or equations is not always the most optimal but I have shown it here for contrast between linear regression and logistic regression. A better approach is to use Tensorflow built-in cost functions for cross entropy calculations. Future code for deep neural networks will abstract this by using the built-in functions.

```
#logistic regression
def loss(output, y_tf):
    output2 = tf.clip_by_value(output, 1e-10, 1.0)
    dot_product = y_tf * tf.log(output2)
    xentropy = -tf.reduce_sum(dot_product, reduction_indices=[1])
    loss = tf.reduce_mean(xentropy)
    return loss
```

There are actually 2 types of cross entropy. They are:

The binary form of cross entropy (for 2 classes):

$$J(\theta) = -\left(\frac{1}{m}\right) \sum_{i=1}^m [ y \log(h\theta(x)) + (1 - y) \log(1 - h\theta(x)) ]$$

And the multiclass form:

$$\text{dot\_product} = y\_tf * tf.log(\text{output2})$$

We actually used the multiclass cross entropy here:

```
def loss(output, y_tf):
    output2 = tf.clip_by_value(output, 1e-10, 1.0)
    dot_product = y_tf * tf.log(output2)
    xentropy = -tf.reduce_sum(dot_product, reduction_indices=[1])

    loss = tf.reduce_mean(xentropy)
    return loss
```

An easy way to understand this is with an example. Assume we try to classify images of vehicles and we have 5 categories for [plane, car, boat, bike, balloon]. Given a cross entropy model we have the following output vector for this multi-class problem as such:

$$Q1 = [0.3 \ 0.2 \ 0.05 \ 0.05 \ 0.40]$$

According to this model, the image is 30% plane, 20% car, 5% boat, 5% bike, and 40% balloon. The model is not very confident about what type of vehicle this is. In contrast, the label for each image would tell us with great certainty

$$P1 = [1 \ 0 \ 0 \ 0 \ 0]$$

that this is a plane. We can calculate the cross entropy to measure our model cost.

$$\text{CrossEntropy}(P1, Q1) = - \sum P1(i) \log Q1(i)$$

If we calculate the values we get:

$$\begin{aligned} \text{CrossEntropy} &= -(1 \log 0.3 + 0 \log 0.2 + 0 \log 0.05 + 0 \log 0.05 + 0 \log 0.4) \\ &= -\log 0.3 \\ &\sim 1.20 \end{aligned}$$

After training, the model is much better and it can predict the following

$$Q1 = [0.98 \quad 0.01 \quad 0.01 \quad 0 \quad 0]$$

As you can see below the cross entropy is now lower and optimized.

$$\begin{aligned} \text{CrossEntropy} &= -(1 \log 0.98 + 0 \log 0.01 + 0 \log 0.01 + 0 \log 0 + 0 \log 0) \\ &= -\log 0.98 \\ &\sim 0.0202 \end{aligned}$$

Therefore, with the cross entropy loss you compare predicted values to the labels.  
As the prediction improves, the cross entropy value goes down.

Well, that is it. Everything else is the same. In the next section, we will now look at neural nets of 1 layer! We are almost to our objective.

#### 4.8.1 A Simple Logistic Regression Example

In this section, I will show a simple logistic regression example. The first step is to read the libraries.

```
import tensorflow as tf
import numpy as np
from numpy import genfromtxt
import sklearn
from sklearn.preprocessing import StandardScaler
```

Then we set the parameters for the batch size, number of epochs, and learning rate.

```
number_epochs = 1000
learning_rate = 0.01
batch_size = 10
```

Here, we use `genfromtxt` to load the data. This is one of my favorite ways to read in data. It so straight forward and clear.

The data is available from the book `github`. Notice that we use a list comprehension to slice columns.

```
x_train = genfromtxt('cs-training.csv', delimiter=',', usecols=(i for i in range(1,5)) )
y_train = genfromtxt('cs-training.csv', delimiter=',', usecols=(0))

x_test = genfromtxt('cs-testing.csv', delimiter=',', usecols=(i for i in range(1,5)) )
y_test = genfromtxt('cs-testing.csv', delimiter=',', usecols=(0))
```

Next step is to normalize the data using the standard scaler from sklearn.

```
## normalizing

sc = StandardScaler()
sc.fit(x_train)

x_train_normalized = sc.transform(x_train)
x_test_normalized = sc.transform(x_test)
```

After obtaining the data, we can perform one hot encoding with tensorflow built in function. Depth of 3 indicates that there are 3 classes.

```
# one-hot encoding

depth = 3
y_train_onehot = tf.one_hot(y_train, depth)
y_test_onehot = tf.one_hot(y_test, depth)
```



The following code can get us the number of features and classes dynamically. Notice that `y_train_onehot` is a tensor in the computational graph. To read its data, we need to call the session.

```
# features (A)

A = len(x_train[0])
print A # number of features

#####
# classes (B)

B = 3 #len(sess.run(y_train_onehot[0]))
print "number of classes ", B
```

Now we are ready to declare the core functions. First we implement the inference function. We use softmax as described in previous sections.

```
def inference(x, A, B):
    W = tf.Variable( tf.zeros([A, B]) )
    b = tf.Variable(tf.zeros( [B] ) )
    output = tf.nn.softmax( tf.matmul(x, W) + b )
    return output
```

Here we use the loss function with the equations. Going forward we will use the built in tensorflow loss functions.

```
def loss(output, y):
    output = tf.clip_by_value(output, 1e-10, 1.0)
    dot_product = y * tf.log(output)
    xentropy = -tf.reduce_sum( dot_product )
    loss = tf.reduce_mean( xentropy )
    return loss
```

For training we use gradient descent as usual.

```
def training(cost):
    optimizer = tf.train.GradientDescentOptimizer(learning_rate)
    train_op = optimizer.minimize(cost)
    return train_op
```

The `evaluate` function calculates the accuracy of the model using several tensorflow utility functions. Check the appendix for examples of how to use these functions.

```
def evaluate(output, y):
    correct_prediction = tf.equal( tf.argmax(output,1) , tf.argmax(y,1) )
    accuracy = tf.reduce_mean( tf.cast(correct_prediction, "float") )
    return accuracy
```

Finally, we define the placeholders x for the data and y for the one hot encoded classes.

```
x = tf.placeholder("float", [None, A])  
y = tf.placeholder("float", [None, B])
```

Here we call the core functions.

```
## call the core functions  
  
output = inference(x, A, B)  
cost = loss(output, y)  
train_op = training(cost)  
eval_op = evaluate(output, y)
```

In this section we define and initialize the session.

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

Just before the loop we need to define the batch parameters.

```
## batch parameters

num_samples_train = len(y_train)
print num_samples_train
num_batches = int(num_samples_train/batch_size)
```

And once everything is complete we run the main loop to perform training and testing. Notice that there are 2 for loops. One for number of epochs and one for number of batches. If you think about it, in every epoch we are showing the model all the data. So on epoch 1, we show all the data to the model. Then, in epoch 2, we show the model all the data again.

### **Why is that?**

It took me a while to understand that but it turns out that that is a good thing. We want the model to experience things more than once. So we feed the data over and over again. This is something that you want to happen and is especially hard in reinforcement learning where the model may go in one direction, see something only once, and never see it again.

The best analogy I can think of is like going to work on the same road everyday. Eventually you may become really good at driving that road but you never try other roads. So that is why we give the data repeatedly to the model.

```

# MAIN_LOOP()

print "running..."
for i in range(number_epochs):
    for batch_n in range(num_batches):
        sta= batch_n*batch_size
        end= sta + batch_size
        y_temp = sess.run(y_train_onehot)
        sess.run( train_op , feed_dict={x: x_train_normalized[sta:end,:], y: y_temp[sta:end, :]})

        y_test_temp = sess.run(y_test_onehot)
        print "accuracy ..."
        accuracy_score = sess.run(eval_op, feed_dict={x: x_test_normalized, y: y_test_temp})
        print "run {}, {}".format(i, accuracy_score)

```

Now we are ready to add more layers. Next step is to add hidden layers.

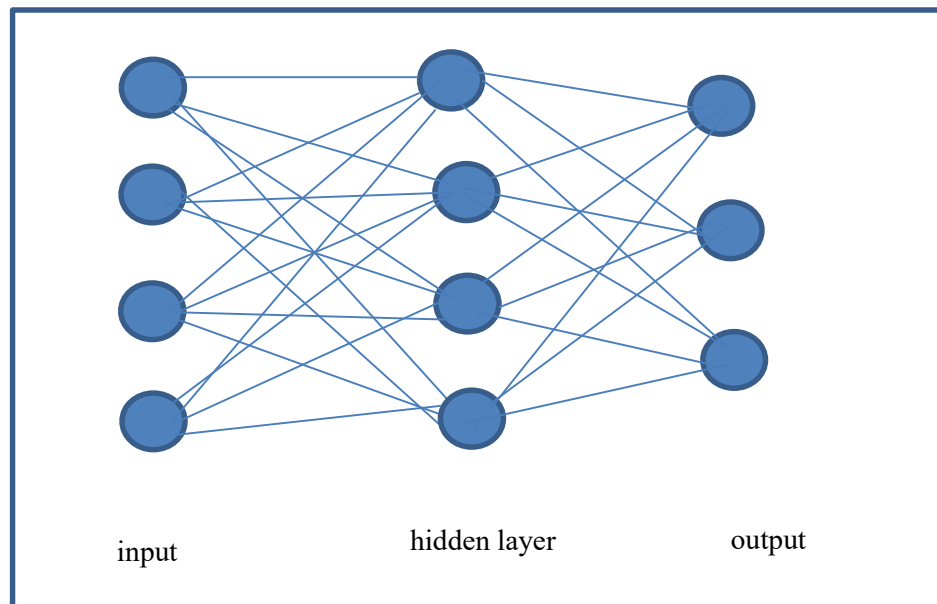
## 4.9 Layers of the Neural Network in Tensorflow

As we saw in the previous discussions, a logistic regression model is similar to a linear regression model. As you might imagine, a neural network is similar to the logistic regression model.

In particular, the neural network now has more layers and the more layers it has, the deeper it becomes.

Layers in between the input and output layers are called hidden layers and help to represent non-linearity and to better discover new features from the input features.

For example, in an image processing problem, a neuron in a hidden layer might take as inputs 2 features or input neurons that detect circles.

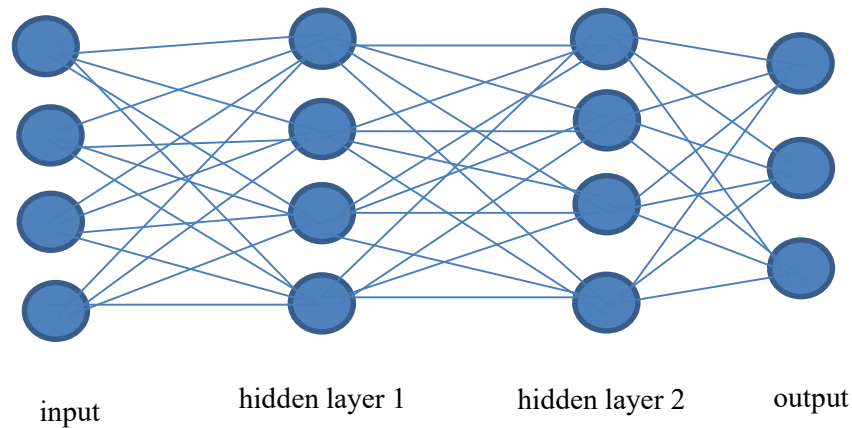


**Figure. A 1 layer neural network architecture.**

If these two circles are detected by the 2 input neurons, then the neuron in the next hidden layer which is connected to them might infer that a pair of eyes has been discovered. Therefore, this neuron in the hidden layer becomes a face detection or “pair of eyes” detection neuron.

Given enough data, the model may discover this intuition and many others. A 1-layer neural network can look like the previous figure and a 2-layer neural network can look like the network in figure below. As can be seen, we can continue to add layers and this is part of defining the architecture of a deep neural network. In theory, more layers can potentially give a model more power to learn from the data and be a better classifier.

Notice, however, that this also means that many more parameters need to be estimated (e.g. more weights). Therefore, until recently, this was very expensive to achieve. With the advent of high performance hardware and GPUs, this is now more feasible.



**Figure. A 2 layer neural network architecture.**

In our code, we need an efficient way of creating layers that is straightforward. For this, we will use a modular approach and simply define a function called **layer()**. If you look at the network figures, each layer has neurons connected to other layers. In fact, the lines that connect the neurons are the weights, so each neuron in say

hidden layer 1 is equal to the sum of the neurons in the previous layer times their respective weights which are learned by the model. Therefore, we can define the function **layer()** as can be seen below.

```
def layer(input, weight_shape, bias_shape):
    weight_stddev = (2.0/weight_shape[0])**0.5
    w_init =
        tf.random_normal_initializer(stddev=weight_stddev)
    bias_init = tf.constant_initializer(value=0)
    W = tf.get_variable("W", weight_shape,
        initializer=w_init)
    b = tf.get_variable("b", bias_shape,
        initializer=bias_init)
    return tf.nn.relu(tf.matmul(input, W) + b)
```

In the previous code, we continue to use the equation

```
tf.matmul(input, W) + b
```

and this time pass it through the neural network function

```
tf.nn.relu()
```

The relu (Hahnloser et al. 2000) function stands for rectified linear unit. It is an optimal neuron for neural networks and it serves as the activation function. Notice also, that in the code, a few **init** variables were defined to initialize the variables and improve the optimization performance.

Finally, the very important aspect here about this **layer()** function is that we now have a very efficient way of defining each layer in a deep neural network architecture. In the next section we will see how deep architectures can be defined using this **layer()** function.



## 4.10 Going Deep: An N layer Neural Network in Tensorflow

We have defined so much of the code already that there really isn't much left except to define the architecture and cost (or loss) function. For the cost function in a deep neural network, similarly to the least square error function in linear regression or cross entropy in logistic regression, we can define the equation or more conveniently use the built-in Tensorflow cost functions. In this case, we use `tf.nn.softmax_cross_entropy_with_logits()`. This is defined in the code segment below for the **loss\_deep()** function.

```
def loss_deep(output, y_tf):  
    xentropy = tf.nn.softmax_cross_entropy_with_logits(output,y_tf)  
    loss = tf.reduce_mean(xentropy)  
    return loss
```

Finally, the last part in our code is to define the architecture of the deep neural network. The following examples show how we could define the architectures using multiple layers. The important aspect to notice is that we are treating each layer as a function **layer()**. The output of one function becomes the input of the next function. So, for a network of 2 layers we can define the code as shown in the code segment below.

The input to the first layer in hidden scope 1 is **x\_tf** which is our data matrix that we defined in previous sections. The result of this equation would be passed to the variable **hidden\_1**. Then, for the next layer, **hidden\_1** becomes the input to the

layer in hidden scope 2, and finally **hidden\_2** becomes the input to the final layer in scope output which is the connection of hidden layer 2 to the output layer.

```
#defines network architecture
#deep neural network with 2 hidden layers
def inference_deep(x_tf, A, B):
    with tf.variable_scope("hidden_1"):
        hidden_1 = layer(x_tf, [A, 4],[4])
    with tf.variable_scope("hidden_2"):
        hidden_2 = layer(hidden_1, [4, 4],[4])
    with tf.variable_scope("output"):
        output = layer(hidden_2, [4, B], [B])
    return output
```

Using the iris data set, the following definition

```
hidden_1 = layer(x_tf, [A, 4],[4])
```

means that 1 input vector from the iris data set is connected to the first hidden layer of the model. Therefore, the neurons in the input layer are the 4 features per plant in the iris dataset. In contrast, for the hidden layer, the neurons represent abstract entities that are sums of the input features times their respective weights. The final part

```
output = layer(hidden_2, [4, B], [B])
```

in the function defines a mapping of 4 abstract neurons (from hidden 2) to the 3 neurons in the output layer which represent the 3 classes of setosa, virginica, and versicolor.

Here, hidden\_2 is defined as a mapping from hidden\_1 (which has 4 neurons) to hidden\_2 which also has 4 neurons. The number of neurons can be defined by the architect of the network.

Similarly, a deep neural network of 3 hidden layers can be defined as follows.

Notice that the following statement helps to define the third hidden layer

```
output = layer(hidden_3, [10, B], [B])
```

and that each hidden layer now consists of 10 neurons as can be seen in the code below.

```
#defines network architecture
#deep neural network with 3 hidden layers
def inference_deep_3layers(x_tf, A, B):
    with tf.variable_scope("hidden_1"):
        hidden_1 = layer(x_tf, [A, 10],[10])
    with tf.variable_scope("hidden_2"):
        hidden_2 = layer(hidden_1, [10, 10],[10])
    with tf.variable_scope("hidden_3"):
        hidden_3 = layer(hidden_2, [10, 10],[10])
    with tf.variable_scope("output"):
        output = layer(hidden_3, [10, B], [B])
    return output
```

Finally, as a last example, a 4 layer deep neural network can be defined with the following Tensorflow code. Here, the lines

```

hidden_4 = layer(hidden_3, [21, 21],[21])

output = layer(hidden_4, [21, B], [B])

```

have been added to the architecture and each hidden layer consists of 21 neurons. The number of neurons to define the layer is sometimes considered more of an art than a science so some experimentation will be required. However, some practitioners recommend that the number of neurons in the hidden layers should not exceed the number of neurons in the input layer (i.e. the features per sample).

A common approach by the Tensorflow community is to use `tf.variable_scope` to better define the scope of the variables in each layer. This provides efficiencies in the internal workings of the Tensorflow code.

```

#defines network architecture
#deep neural network with 4 hidden layers
def inference_deep_4layers(x_tf, A, B):
    with tf.variable_scope("hidden_1"):
        hidden_1 = layer(x_tf, [A, 21],[21])
    with tf.variable_scope("hidden_2"):
        hidden_2 = layer(hidden_1, [21, 21],[21])
    with tf.variable_scope("hidden_3"):
        hidden_3 = layer(hidden_2, [21, 21],[21])
    with tf.variable_scope("hidden_4"):
        hidden_4 = layer(hidden_3, [21, 21],[21])
    with tf.variable_scope("output"):
        output = layer(hidden_4, [21, B], [B])
    return output

```

## 4.11 Evaluating your Deep Neural Network in Tensorflow

Now that we have defined our entire deep neural network models, we can focus on some issues with performance evaluation. The previously defined `print_stats_metrics()` function can be used to estimate the performance of the deep neural network.

```
## print stats
precision_scores_list = []
accuracy_scores_list = []
def print_stats_metrics(y_test, y_pred):
    print('Accuracy: %.2f' % accuracy_score(y_test, y_pred) )
    accuracy_scores_list.append(accuracy_score(y_test, y_pred) )
    confmat = confusion_matrix(y_true=y_test, y_pred=y_pred)
    print "confusion matrix"
    print(confmat)
    print pd.crosstab(y_test, y_pred, rownames=['True'],
                      colnames=['Predicted'],
                      margins=True)
    precision_scores_list.append(precision_score(y_true=y_test,
                                                  y_pred=y_pred))
    print('Precision: %.3f' % precision_score(y_true=y_test,
                                              y_pred=y_pred))
    print('Recall: %.3f' % recall_score(y_true=y_test,
                                        y_pred=y_pred))
    print('F1-measure: %.3f' % f1_score(y_true=y_test,
                                        y_pred=y_pred))
```

The function takes in the predicted labels (**y\_pred**) and the annotated labels (**y\_test**) and compares performance. Really, the only issue is to make sure that both inputs to the function are in the correct format. The lists **precision\_scores\_list = []** and **accuracy\_scores\_list = []** are used for printing the performance plots with the plotting function described earlier. The code is shown below. The final part I want to discuss is how 10-fold cross validation works and can be implemented for our deep neural networks. In this case, I have provided a simplistic way of looking at 10-fold cross validation for the sake of ease of understanding. Quite simply, cross validation means dividing the data sets into bins (for instance 10) and then iterating through them while using some for training and some for testing. Usually, 9 are used for training and 1 is held out for testing. The following function `select_fold_to_use_rc(X, y, k)` defines the process. Using an input parameter **k** (the fold) we can select the indices that will be used for training and testing. Every fold, these indices will change. So, for each fold, the function `select_fold_to_use_rc(X, y, k)` returns the samples to use for training and the values to use for testing based on the selected indices.

```
## manual 10 fold crossvalidation get sets
def select_fold_to_use_rc(X, y, k):
    K = 10
    num_samples = len(X[:,0])
    training_indices = [i for i, x in enumerate(X) if i % K != k]
    testing_indices = [i for i, x in enumerate(X) if i % K == k]
    X_train = X[training_indices]
    y_train = y[training_indices]
    X_test = X[testing_indices]
    y_test = y[testing_indices]
    return X_train, X_test, y_train, y_test
```

Once we have the sets, we can simply perform the analysis with the selected sets. This of course can be done in a loop but here I have hard-coded each call to a different fold for illustration purposes. Tensorflow has its own built-in ways of performing cross validation.

```
## manual 10 fold cross validation
#select each fold
#X_train, X_test, y_train, y_test =
select_fold_to_use_rc(X_train_10fold, y_train_10fold, 1)
#X_train, X_test, y_train, y_test =
select_fold_to_use_rc(X_train_10fold, y_train_10fold, 2)
#X_train, X_test, y_train, y_test =
select_fold_to_use_rc(X_train_10fold, y_train_10fold, 3)
#X_train, X_test, y_train, y_test =
select_fold_to_use_rc(X_train_10fold, y_train_10fold, 4)
#X_train, X_test, y_train, y_test =
select_fold_to_use_rc(X_train_10fold, y_train_10fold, 5)
#X_train, X_test, y_train, y_test =
select_fold_to_use_rc(X_train_10fold, y_train_10fold, 6)
#X_train, X_test, y_train, y_test =
select_fold_to_use_rc(X_train_10fold, y_train_10fold, 7)
#X_train, X_test, y_train, y_test =
select_fold_to_use_rc(X_train_10fold, y_train_10fold, 8)
#X_train, X_test, y_train, y_test =
select_fold_to_use_rc(X_train_10fold, y_train_10fold, 9)
#X_train, X_test, y_train, y_test =
select_fold_to_use_rc(X_train_10fold, y_train_10fold, 0)
```

## 4.12 Summary

In this chapter, the main topic of deep learning was introduced. The chapter addressed the Tensorflow environment and code definitions as well as theoretical frameworks for deep learning. Issues about neural network architecture and performance evaluation were also presented. Finally, several coding examples using python, Sklearn, and Tensorflow were provided in an incremental fashion for the algorithms of linear regression, logistic regression, 1-layer neural networks, and n-layer neural networks. The next chapter will further look at other methods that can be implemented with Tensorflow and, in particular, their relationship with semantic vector spaces.



## CHAPTER 5: SEMANTIC VECTOR SPACES

In this chapter, I will discuss semantic vector spaces. I will discuss the vector space model and then address Latent Semantic Analysis (LSA), and the newer vector space models based on word embeddings such as Word2vec (Mikolov et al. 2013). Again, the focus will be on the intuition. Many companies today base their products on these very simple but powerful semantic vector spaces.

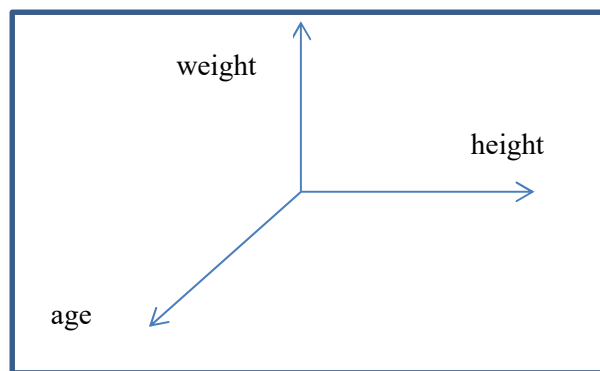
Annotating data is a very expensive and time consuming task and in many cases it is practically impossible to accomplish. In the past, unsupervised methods were not always the most successful in information extraction tasks and were sometimes relegated to data compression (i.e. PCA, SVD, auto-encoding, etc.), or to data clustering. However, in recent years there has been a renewed interest in unsupervised learning methods. Not taking advantage of the vast amounts of unsupervised data means that a great chunk of knowledge is being left on the table. In particular, deep learning has helped to make some unsupervised learning problems more tractable and/or approachable.

Current deep learning based methods are optimally designed to handle big data. That is, terabytes of data can now be processed and new patterns in the data discovered. Additionally, the properties of deep neural networks have allowed for the development of new architectures and methodologies to address unsupervised data using shallow or deep neural networks. Some of these new approaches based on deep learning include distributed word embeddings such word2vec (Mikolov et al. 2014), Autoencoders (Goodfellow et al. 2016), and Generative Adversarial Networks (Goodfellow et al. 2014). GANs will be discussed in another chapter.

Distributed word embeddings are techniques that learn dense vector representations for words based on the co-occurrence of words in very large data sets. Distributed word embeddings have been used extensively in many projects.

## 5.1 Vector Space Model

The vector space model refers to a space that can be created to represent samples. A set of axes is created from 0 to some range. Then, any samples can be represented in this space. For example, say that you want to represent 100 people in a room by their age, height and weight. Then, in this case we can create a vector space of 3 dimensions which are age, height, and weight. Now, every person in the room can be mapped to this space based on their values for these 3 parameters.



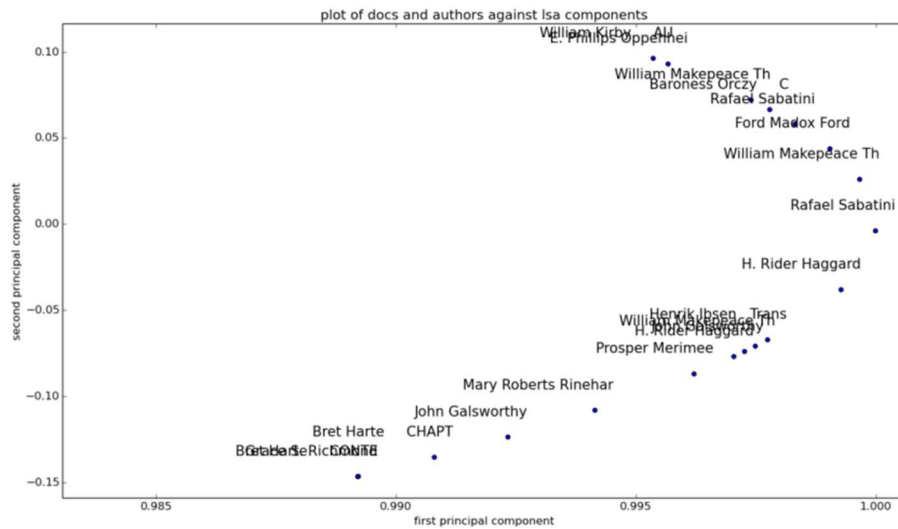
**Figure. Vector Space Model.**

## 5.2 Latent Semantic Analysis (LSA)

Latent Semantic Analysis (LSA) is a method for discovering relationships or concepts between a set of rows and columns in a dataset such as between documents and terms or between users and movies. These newly discovered concepts can be represented as vectors and they can become a new way of mapping the elements (e.g. the terms). This technique has been used extensively in natural language processing. The concept vectors are discovered using a technique called singular value decomposition (SVD). A very good discussion of SVD can be found in the book: *Data-Driven Modeling & Scientific Computation* by J. Nathan Kutz.

Once this new space has been created based on the discovered vectors, the elements or terms can be represented in this space. Within this space, distance function metrics can be used to measure similarity between the elements (such as the documents).

The following example shows how LSA can be used to plot and capture the relationship between authors of classical literary books. Since this LSA space has considered writing style, it is expected that books by the same author will appear close to each other.



**Figure. LSA Vector Space**

The code to perform the LSA analysis using the SKlearn kit can be seen in the following code segment. It is assumed that you receive a list of text documents or paragraphs in the python list

```
list_of_texts = []
```

The list of texts is vectorized and a tf-idf (term frequency-inverse document frequency) matrix is calculated. From this matrix, an LSA model is calculated with

```
model = lsa.fit(tfidfMatrix).
```

Then, the data is normalized and a similarity metric is calculated between all the elements in the matrix using

```
Similarity = np.asarray(np.asmatrix(dtm_lsa) *  
                        np.asmatrix(dtm_lsa_test_sample).T )
```

This similarity metric can help to determine which documents are close to each other. The last part of the code is to print the results using the **pandas** library.

```
vectorizer = CountVectorizer(min_df=1, stop_words='english')  
vec = vectorizer.fit(list_of_texts)  
dtm = vec.transform(list_of_texts)  
#####  
tfidf = TfidfTransformer()  
tfidfMatrix_metric = tfidf.fit(dtm)  
tfidfMatrix = tfidfMatrix_metric.transform(dtm)  
#####  
#lsa = TruncatedSVD(2, algorithm='arpack') ## 2 components  
lsa = TruncatedSVD(40, algorithm='arpack') ## 40 components  
model = lsa.fit(tfidfMatrix)  
dtm_lsa = model.transform(tfidfMatrix)  
#####  
Norm_obj = Normalizer(copy=False).fit(dtm_lsa)  
dtm_lsa = Norm_obj.transform(dtm_lsa)  
similarity = np.asarray(np.asmatrix(dtm_lsa) *  
                        np.asmatrix(dtm_lsa_test_sample).T )  
#print similarity  
print pd.DataFrame(similarity, index=list_of_authors,  
                    columns=authors_to_compare)
```

### 5.3 Word Embeddings and Neural Nets

Word embeddings are another approach like LSA that allow you to place data in a vector space to discover relationships. Specifically, word embeddings look at the co-occurrence of terms in a document. Word embeddings have existed for a while. However, the improvements in deep learning technologies have led to the development of newer techniques for calculating word embeddings. In particular, Mikolov et al. proposed new algorithms to calculate word embeddings. One of the methods he proposed is called word2vec. Although not a deep neural network (it only uses 1 hidden layer), this technique can be efficiently implemented in Tensorflow.

The python code for word2vec is available from the book GitHub at <https://github.com/rcalix1> and the book website at [www.galacticbackwater.com](http://www.galacticbackwater.com).

The word2vec code seldom needs to be modified, except for a few parameters. and so I will not discuss it here. Instead, I will focus the discussion on a few aspects of word2vec and utilization.

In particular, word2vec has been widely used for natural language processing. It takes a very large document set and produces a vector space where all the words can be mapped. This vector space is based on the co-occurrence of the words. The dimension  $m$  (number of orthogonal axes) in the space is determined by the number of neurons  $m$  in the hidden layer of the neural network used to perform the analysis. There are various implementations of word2vec. The most common one (the skip gram model) uses a 1-layer neural network to calculate the probability of co-occurrence between the words in a vocabulary defined from the input text set.

As can be seen in the figure below, the word2vec model has 3 layers:

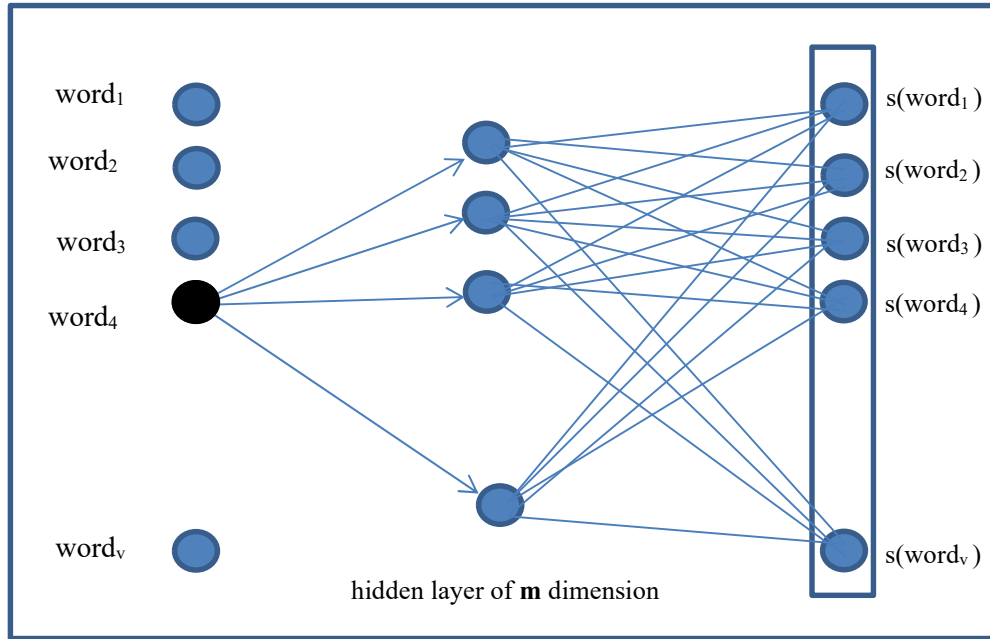
- input
- hidden 1
- output

The input layer consists of the words in the vocabulary of size “ $v$ ”. One-hot encoding once again is used here. For example, for the vector [dog, cat, blue] that might represent a vocabulary of a total of 3 words, the word cat would be encoded as follows:

$$[0, 1, 0]$$

The vocabulary defines the number of neurons ( $v$ ) in the input layer. These one-hot encoded vectors for the words in the vocabulary are the inputs to the input layer of the neural network. The outputs of the neural network are also the one-hot encoded word vectors using a softmax function (  $\mathbf{s}()$  ).

Similarly to logistic regression, the softmax is used to calculate a probability distribution over the words. In this case, the probability that word  $j$  in the vocabulary co-occurs with the other words in the vocabulary.



**Figure. word2vec model**

Finally, the hidden layer in the model learns the embeddings through the weight matrix ( $\mathbf{W}$ ) for each word. The number of neurons in the hidden layer is selected arbitrarily ( $\mathbf{m}$ ). In this way, each word will have an embedding represented by the  $\mathbf{m}$  neurons in the hidden layer. Assuming that the neurons in the hidden layer are called:

$$h_1, h_2, h_3, \dots, h_m$$

Then, an embedding for a word would be as follows:



$$\text{word}_j = w_1 * h_1 + w_2 * h_2 + w_3 * h_3 + \dots + w_m * h_m$$

and the weight matrix (**W**) would look like the following.

	$h_1$	$h_2$	$h_3$	$h_4$	...	$h_m$
$\text{word}_1$	-0.0214	-0.121	0.14	-0.126		-0.06
$\text{word}_2$	-0.13	0.109	-0.06	0.022		0.08
$\text{word}_3$	0.08	-0.11	-0.08	0.03		0.12
...						
$\text{word}_v$	-0.08	0.07	0.20	-0.04		0.006

In the next example we can see each word and its vector representation assuming a hidden layer of 6 neurons.

xanax,-0.0214805,-0.121305,0.145779,-0.126096,-0.114934,-0.0642776
sleep,-0.131741,-0.109269,-0.06438,0.0227307,-0.03691,0.0803119
like,0.0839722,-0.111817,-0.0829403,0.0319387,0.129373,0.132047
take,-0.00416363,0.200523,0.029628,0.0190524,-0.130761,-0.00113905
took,-0.0048833,0.0940326,-0.0491561,-0.0484801,-0.0244865,-0.0989419
adderall,-0.0865235,0.0788,0.203447,-0.0491842,-0.00105494,0.00675861,
feel,-0.00441388,0.0416924,-0.146706,0.000622775,-0.0609331,-0.0442516
got,-0.120518,-0.0410703,-0.0321787,0.0545667,-0.069366,0.101827

One of the important observations Mikolov et al. made about the word2vec space is that semantic relationships can be discovered by performing vector arithmetic. For instance, given enough data, the following vector arithmetic may be discovered:

$$\text{king} + \text{man} - \text{queen} = \text{woman}.$$

The following code segment shows a simple way of performing vector arithmetic, assuming you have a dictionary of vectors.

```
dictionaryWordsToVectors = {}  
#####  
f = open("data/ word2vec_vector_space.txt", "r")  
for line in f.readlines():  
    features = line.split(",")  
    n = len(features)  
    vector = features[1:n-1]  
    vector_float = [float(x) for x in vector]  
    key_word = features[0]  
    dictionaryWordsToVectors[key_word] = vector_float  
  
#####  
def getVectorFromWord(word):  
    vector = dictionaryWordsToVectors[word]  
    return vector  
    return top_5_closest  
#####  
def findClosestWordToThisVector_list(vector):  
    top_5_closest = []  
    temp_dict = {}  
    for key, value in dictionaryWordsToVectors.iteritems():  
        distance_result = distance.euclidean(vector, value)  
        temp_dict[key] = distance_result  
    sorted_x =  
        sorted(temp_dict.items(), key=operator.itemgetter(1))  
    for i in range(20):  
        top_5_closest.append(sorted_x[i][0])  
    return top_5_closest
```

```
#####  
a = np.array(getVectorFromWord("prozac"))  
c = np.array(getVectorFromWord("morphine"))  
result_vector = a + c  
result_word = findClosestWordToThisVector_list(a)
```

## 5.4 Word2vec code

In this section, I will explain some aspects of the word2vec code proposed by Mikolov et. al (2014). I do recommend that you use the available word2vec code from the Tensorflow foundation or the one posted on the github repository. That being said, I always wanted to understand word2vec and I am sure you are interested in understanding it as well. Therefore, here I will go over certain parts of the code and try to clarify aspects of it. Generally speaking, I would say that the word2vec code has 2 main components. One component is non deep learning related and the other is the deep learning architecture. Most algorithms we have discussed so far usually are straight forward when it comes to obtaining the data (**X**) and the labels (**y**). As such, the focus and challenge has been in the deep learning architecture itself. Given that we have mainly looked at supervised learning algorithms, our data was simple. Feature vectors of images, for instance, and their annotated one-hot encoded labels. Word2vec is different though. Word2vec is an unsupervised learning algorithm so there are no annotated labels. Instead, we need to learn mappings (distributed vector representations) from the co-occurrence of things such as terms in a text. This process of getting the initial train data is more difficult. Remember that we said in the previous section that the inputs and outputs of the model are the one-hot encoded words from a large vocabulary.

Okay, so here we go. The best way to understand this process of obtaining the **X** and **y** is with an example. The first thing we need is some libraries and parameters which can be seen in the below code section.

```
import collections
import math
import os
import random
import zipfile
import numpy as np
from six.moves import urllib
from six.moves import xrange
import tensorflow as tf

vocabulary_size = 55000
```

Next, let us assume that we have a very very large text like the following named “text8”:

```
... anarchism originated as a term of abuse first used ...
```

The first step is to convert this very large text into a list of words. To achieve this we can use a Tensorflow command.

```
# Read the data into a list of strings

f = open("text8")
words = tf.compat.as_str( f.read() ).split()
print('Data size', len(words))
```

Our variable **words** now contains the text in the form of a list.

```
## at this point words looks like this

words = ["anarchism", "originated", "as", "a", "term", "of", "abuse",
"first", "used", ...]
```

The next step is to perform a frequency count of the terms in the list **words**. The variable “**count**” is created and used for this purpose. After the frequency count is performed, count looks like the following:

```
count = [[unk,-1], (the,78), (of, 67), ...]
```

As can expected, the most frequent terms are “the”, “of”, etc. The variable UNK will be used to tally infrequent words. That is, not all words will be used as part of the vocabulary. For example, out of 100,000 words, the 55,000 most frequent words could be selected to form the vocabulary. As a result, the remaining 45,000 infrequent words would not be part of the vocabulary and would be counted as unknown words. The variable vocabulary size must be manually set. The code to perform the count can be seen below.

```
## this is a frequency count

count = [['UNK', -1]]
count.extend(collections.Counter(words).most_common(vocabulary_size - 1))
```

The next step is to build a dictionary of words and keys. Keys are created in order of most frequent using the current dictionary size. Most frequent terms have lower value keys. For example

```
dictionary = {'and': 3, 'of': 2, 'the': 1, 'UNK': 0}
```

The code to create the dictionary is provided below. Notice that it uses the previously created frequency counts to assign the keys in ascending order from most frequent to least frequent.

```
dictionary = dict()
for word, _ in count:
    dictionary[word] = len(dictionary)
```

The next step is to create the variable “**data**”. The code below uses the word, key pairs in the dictionary to create the variable “**data**”. The variable dictionary (which is a python dictionary data structure) has each word and its corresponding key. The resulting “**data**” object is a list that contains the keys for the words in the original list “**words**”. So now **data** has the same information as **words** except that instead of containing the actual word (such as cat) it contains its corresponding key (such as 3) from the dictionary. The objects **words** and **data** are 2 parallel lists; one list for the words and one list for the keys. So if **words** looked like this

```
words = ["hello", "Cat", "tree", ...]
```

then **data** now looks like this

```
data = [17, 3, 53, ...]
```

where in dictionary we have

```
dictionary["Cat"] = 3  
dictionary["hello"] = 17  
dictionary["tree"] = 53
```

The code to accomplish this is as follows:

```
data = list()  
unk_count = 0 ##count unknown words. Dictionary has 55k frequent  
terms  
  
for word in words:  
    if word in dictionary:  
        index = dictionary[word]  
    else:  
        index = 0 # dictionary['UNK']  
        unk_count = unk_count + 1  
    data.append(index)
```

The “if” statement in the code checks to see if the current word from **words** is in the dictionary (it is frequent) or if it is not (infrequent). If the word is not in the dictionary, the **unk\_count** is incremented. If the word is in the dictionary, then its corresponding key is extracted and appended to data. Unknown words have an index of zero.

Once the loop finishes counting the unknown words, the unknown parameter in the frequency count stored in **count** is updated with this value as can be seen in the next code section.

```
count[0][1] = unk_count
```

The object **count** is a frequency count of the words. The unknown word UNK, which is the first tuple in **count** initially had -1. After the unknown words count is complete, UNK will have total of all the unknown terms.

Now that we have created **words** and **data**, we can visualize them together with the following code. The loop shows the first 15 values for each object.

```
for i in range(15):  
    print words[i], " -> ", data[i]  
    x = raw_input()
```

The result of the visualization is shown below. It shows word and key pairs in the format word -> key.

```
anarchism -> 5239  
originated -> 3084  
as -> 12  
a -> 6  
term -> 195  
of -> 2  
abuse -> 3137  
first -> 46  
used -> 59
```

The next step is to reverse the dictionary and its order.



That is, where as dictionary looked like this

```
dictionary = {'and': 3, 'of': 2, 'the': 1, 'UNK': 0}
```

Reverse\_dictionary now looks like this

```
Reverse_dictionary = {0: 'UNK', 1: 'the', 2: 'of', 3: 'and', 4: 'one', 5: 'in', 6: 'a', 7: 'to', 8: 'zero', 9: 'nine', 10: 'two', ...}
```

The code to achieve this is straight forward.

```
reverse_dictionary = dict(zip(dictionary.values(), dictionary.keys()))
```

For the variable **reverse\_dictionary** the keys are now in the place of words and words in place of the keys and the order is now ascending from 0, 1, 2, etc. This will help to look up words by key.

To print the most common words, the following code can be used

```
print('Most common words', count[:5])  
  
print('Sample data', data[:10], [reverse_dictionary[i] for i in data[:10]])
```

Okay, after all of this we are now ready to generate a batch of **x** data and a batch of labels **y**. To do this we will use a function called

`generate_batch()`

Since we are training a word2vec model we know that **x** is a list of one-hot encoded word vector and **y** is also a list of one-hot encoded vectors. Intuitively, each sample in **x** should look like this

`Cat = [0,0,1,0,...]`

And each sample in **y** should look like this

`Dog=[0,1,0,0,...]`

In fact, our dictionary already created the encoding for all words in the vocabulary. So, if our vocabulary contains 55,000 terms, then the dimensionality of our one hot encoded word vectors is 55,000. If a word such as “house” has key 513 in dictionary, then to encode it we create an all zeros vector of size 55,000 where only the position 513 has a value 1 and the rest are zeros.

Using the previously created objects (words, counts, data, dictionary, reverse\_dictionary) we can now proceed to generate batches.

The main idea is that we will create pairs of words (**w<sub>i</sub>**, **y<sub>i</sub>**). We will take a window of size **window\_size** (for instance 3) and we will create the vectors. We want to capture context in our training set (X,y). Therefore, we need a way to record the co-occurrence of words. In a loop, we iterate through the document

```
words = ["anarchism", "originated", "as", "a", "term", "of", "abuse",  
"first", "used", ...]
```

and pass a buffer or window (in this case of size 3) over each word so that we can have

```
From

words = ["anarchism", "originated", "as", "a", "term", "of", "abuse",
"first", "used", ...]

we want this

[anarchism, originated, as]
[Originated, as, a]
[as, a, term]
[a, term, of]
Etc.
```

So from each window we can obtain pairs of words which become the x and y samples. So from this buffer

```
[originated, as, a]
```

We get 2 word pairs

```
x -> y
as -> originated
as -> a
```

By doing this process we have learned that “as” co-occurs with “originated” and that it also co-occurs with the “a”. We can expand the size of the window to capture more context (i.e. more word pairs).

Now “as” is one-hot encoded

$$x_i: \text{as} = [0,0,0,0,0,0,0,0,1,0,0,0,0,\dots]$$

and “originated” is also one-hot encoded

$$y_i: \text{originated} = [0,0,0,0,0,0,0,0,0,0,0,0,1,\dots]$$

These 2 vectors can now be used to train the word2vec model as follows

$x_i$       -----      hidden      -----       $y_i$

or

as      -----      hidden      -----      originated

The previously described process can be performed with the following code contained in the function `generate_batch()`

```
data_index = 0

#####
def generate_batch(batch_size, num_skips, skip_window):
    global data_index
    assert batch_size % num_skips == 0
    assert num_skips <= 2 * skip_window
    batch = np.ndarray(shape=(batch_size), dtype=np.int32)
    labels = np.ndarray(shape=(batch_size, 1), dtype=np.int32)
    span = 2 * skip_window + 1 # [ skip_window target skip_window ]
    buffer = collections.deque(maxlen=span)
    for _ in range(span):
        buffer.append(data[data_index])
        data_index = (data_index + 1) % len(data)
    for i in range(batch_size // num_skips):
        target = skip_window # target label at the center of buffer
        targets_to_avoid = [skip_window]
        for j in range(num_skips):
            while target in targets_to_avoid:
                target = random.randint(0, span - 1)
            targets_to_avoid.append(target)
            batch[i * num_skips + j] = buffer[skip_window]
            labels[i * num_skips + j, 0] = buffer[target]
        buffer.append(data[data_index])
        data_index = (data_index + 1) % len(data)
    # Backtrack a bit to avoid skipping words in the end of a batch
    data_index = (data_index + len(data) - span) % len(data)
    return batch, labels
```

The variable **data\_index** is very important and is necessary to keep track of the current position in the initial input document. That is, if the buffer is centered on the word “term” for the below document

```
["anarchism", "originated", "as", "a", "term", "of", "abuse", ...]
```

then the window buffer contains [a, term, of] and `data_index = 4`. Again, we can use the following code to visualize the results of using `generate_batch()`.

```
print "generate batch does this"
print "given the data like this in data"

print('Sample data', data[:10], [reverse_dictionary[i] for i in data[:10]])

print "it generates pairs of words like this"

batch, labels = generate_batch(batch_size=8, num_skips=2, skip_window=1)
for i in range(8):
    print(batch[i], reverse_dictionary[batch[i]], '->', labels[i, 0],
          reverse_dictionary[labels[i, 0]])

print "this is the batch"
print batch
print "these are the labels"
print labels
```

The output of the previous code looks like this

```

generate_batch() does this
given the data like this in data

('Sample data', [5239, 3084, 12, 6, 195, 2, 3137, 46, 59, 156],
['anarchism', 'originated', 'as', 'a', 'term', 'of', 'abuse',
'first', 'used', 'against'])

it generates pairs of words like this:

(3084, 'originated', '->', 12, 'as')
(3084, 'originated', '->', 5239, 'anarchism')
(12, 'as', '->', 3084, 'originated')
(12, 'as', '->', 6, 'a')
(6, 'a', '->', 195, 'term')
(6, 'a', '->', 12, 'as')
(195, 'term', '->', 2, 'of')
(195, 'term', '->', 6, 'a')

Where this is the batch

[3084 3084  12  12   6   6 195 195]

And these are the labels

[[ 12]
 [5239]
 [3084]
 [  6]
 [195]
 [ 12]
 [  2]
 [  6]]

```

So we finally have our **xs** and **ws** which we can use in our neural network model.

So **xs** is

```
[3084 3084  12  12   6   6 195 195]
```

And **ys** is

```
[[ 12]
 [5239]
 [3084]
 [ 6]
 [ 195]
 [ 12]
 [ 2]
 [ 6]]
```

Notice that **xs** and **ys** are not currently one-hot encoded. This part you do not have to do and instead there are some built in Tensorflow functions that can do that for you if you just give it the desired indices to encode. The one-hot encoding part for the word vectors is done in the inference and objective functions of the network architecture.

Finally, keep in mind that the `generate_batch()` function is called from the main loop of your Tensorflow network. For example like this:

```
for step in xrange(num_steps):
    batch_inputs, batch_labels = generate_batch(batch_size, num_skips, window)
    feed_dict = {train_inputs: batch_inputs, train_labels: batch_labels}

    _, loss_val = sess.run([train_op, loss], feed_dict=feed_dict)
```

Wow! All of that just to get the data for **xs** and **ys**.



Now that we have our data **X (xs)** and **y (ys)**, we are ready for define the neural network architecture for word2vec. Word2vec (skip gram model) requires a set of variables to define the parameters of the algorithm. The variables can be seen in the below code section.

```
batch_size = 128
embedding_size = 128 # Dimension of the embedding vector
skip_window = 1      # How many words to consider left and right
num_skips = 2

valid_size = 16      # Random set of words to evaluate similarity on
valid_window = 100
valid_examples=np.random.choice(valid_window,valid_size,replace=False
)
num_sampled = 64     # Number of negative examples to sample
```

Once we have our parameter set and the data we can proceed to define the architecture for the neural network. First we define the inference function as follows. In this function we create an embedding matrix **embeddings** of size [vocabulary\_size, embedding\_size].

Embedding size is the number of dimensions of our embedding space (in this case 128). The function **embedding\_lookup** is a function that returns the vectors for the provided inputs in **train\_inputs**.

```

# train x - context data, Input data x
# train_inputs = [3084 3084 12 12 6 6 195 195]

#####

def inference(train_inputs, vocabulary_size, embedding_size):
    embeddings = tf.get_variable(
        tf.random_uniform([vocabulary_size, embedding_size], -1.0, 1.0) )
    embed_x = tf.nn.embedding_lookup(embeddings, train_inputs)
    #####
    nce_weights = tf.get_variable(tf.truncated_normal(
        [vocabulary_size, embedding_size],
        stddev=1.0 / math.sqrt(embedding_size)
    ))
    nce_biases = tf.get_variable(tf.zeros([vocabulary_size]))
    result = tf.matmul(embed_x, tf.transpose(nce_weights)) + nce_biases
    #####
    return embed_x, result, nce_weights, nce_biases

```

Then we define the loss function using NCE which is a method called noise contrastive estimation. The function **nce\_loss** performs the optimization. The object **nce\_weights** is of size [vocabulary\_size, 128], **nce\_biases** is of size [128], **train\_labels** is of size [batch size, 1], **embed\_x** is of size [batch size, 128].

```

def nce_loss( train_labels_y,
               embed_x, result, nce_weights, nce_biases, vocabulary_size, n_s):
    loss = tf.nn.nce_loss(
        weights=nce_weights,
        biases=nce_biases,
        labels=train_labels_y,
        inputs=embed_x,
        num_sampled=n_s,
        num_classes=vocabulary_size)
    return tf.reduce_mean(loss)

```

The training is done with gradient descent as follows.

```
def train(cost):  
    optimizer = tf.train.GradientDescentOptimizer(1.0).minimize(cost)  
    return optimizer
```

Now the validation function performs a similarity estimation as indicated below. It computes a cosine similarity estimation between batch examples and all embeddings. The matrix **embeddings** is normalized first. Then a look up the **normalized\_embeddings** is performed. Finally, the cosine matrix multiplication between **normalized\_embeddings** and **valid\_embeddings** is performed.

```
def validation(embeddings, valid_dataset):  
    norm = tf.sqrt(tf.reduce_sum(tf.square(embeddings), 1, keep_dims=True))  
    normalized_embeddings = embeddings / norm  
    valid_embeddings = tf.nn.embedding_lookup(  
        normalized_embeddings, valid_dataset)  
    similarity = tf.matmul(  
        valid_embeddings, normalized_embeddings, transpose_b=True)
```

Now that the core functions have been defined, we can proceed to define our **x** and **y** placeholders as follows:

```
# train x - context data, Input data x  
#[3084 3084 12 12 6 6 195 195]  
  
train_inputs_x = tf.placeholder(tf.int32, shape=[batch_size])
```

And

```
#train y - context data, #these are the labels
#[ [ 12]
# [5239]
# [3084]
# [ 6]
# [ 195]
# [ 12]
# [ 2]
# [ 6]]

train_labels_y = tf.placeholder(tf.int32, shape=[batch_size, 1])
```

We can now initialize the validation set

```
valid_dataset = tf.constant(valid_examples, dtype=tf.int32)
```

We are almost done. All that remains is to call the core functions, initialize the session, and call the main loop. The core functions are called as follows.

```
embed_x, result, nce_weights, nce_biases = inference(
    train_inputs_x, vocabulary_size, 128)

loss = nce_loss(train_labels_y,
    embed_x, result, nce_weights, nce_biases, n_s, vocabulary_size)

train_op = train(loss)

similarity = validation(embed_x, valid_dataset)
```

We now initialize the session and variables with

```
init = tf.global_variables_initializer()
num_steps = 100001
sess = tf.Session()
sess.run(init)
average_loss = 0
```

And at last we call the main loop as follows.

```
## MainLoop()

for step in xrange(num_steps):
    batch_inputs, batch_labels = generate_batch(
                                batch_size, num_skips,
                                skip_window)

    feed_dict = {train_inputs_x: batch_inputs, train_labels_y: batch_labels}

    train_op, loss_val = sess.run([train_op, loss], feed_dict=feed_dict)
```

That is it!! This concludes the word2vec algorithm discussion.

## 5.5 Knowledge Enhanced Vector Spaces

In this section, I will discuss how distributed vector representations can be expanded to include knowledge about the world such as common sense knowledge (Havasi et. al 2007). This can be referred to as knowledge enhanced vector spaces.

In the previous section, we described word2vec. However, as of this writing, there are several distributed vector representation techniques such as Word2vec (Mikolov et al 2013), FastText (Bojanowski et al. 2017), and WordRank (Ji et al. 2015). These distributed vectors can be enhanced with knowledge based vector space representations.

While the methods of distributed word embeddings have been extensively used, they still lack an ability to obtain true understanding from text. Some believe that prior knowledge of the words is required and not just their co-occurrence in large unlabeled data sets.

Knowledge vector spaces can be added to traditional co-occurrence vector spaces to better represent the text samples. Techniques for combining multiple distributed vector representations have been addressed in works such as Garten et al. (2015). Essentially, Garten et al. (2015) argues that different vector spaces can simply be concatenated to each other. That is, for 2 vector spaces for the word “w”, which are  $w_1 = [a, b, c]$  and  $w_2 = [d, e]$ , a new vector space can be created as  $w_3 = [a, b, c, d, e]$ . If the resulting vector space gets too big, dimensionality reduction techniques (such as PCA, SVD, autoencoders) can be used to further compress the data.

One particular problem with word embeddings and natural language processing in general (Ling et al. 2017) is that a lot of information is missing when performing the analysis whether supervised or unsupervised. We as humans, when reading a sentence, do not just rely on the co-occurrence between words to derive meaning from the text.

In fact, the meaning is derived from our previous knowledge of the individual words and their context and how they fit in the world. We know things about things and how they make us feel. Most modern word embeddings approaches lack this knowledge.

Ling et al. 2017 argues that prior knowledge is important in natural language processing tasks. Therefore, methodologies to enhance the way text is represented are needed. In particular, representations that aren't just based on co-occurrence

(e.g. word2vec), but that are also based on a knowledge vector space for each word. These knowledge based and co-occurrence based vector spaces will serve as the features used to represent the samples. These features can then be used for unsupervised pattern discovery.

In Ling et al. (2017) the authors propose the learning of word embeddings based on co-occurrence of words and based on the semantic similarity between words in a given knowledge base. They adjust their word embedding models by modifying the cost function of the model. In essence, the cost function now includes 2 optimization objectives instead of just one. The first optimization objective is the one related to the co-occurrence of words (such as in word2vec). The difference is that now the second objective includes the regularization objective related to the similarity of the words in the knowledge base.

Li et al. (2017) propose to create affective word embeddings to enhance words in a vector space. Their approach proposes a vector space for each word consisting of orthogonal axes for valence, arousal, potency, etc. Therefore, each word is represented as vectors of these axes. Their methodology learned the values for each axis using a linear regression model. First, they built a word embedding model based on co-occurrence metrics. Then, they annotated a seed set of words to indicate their given valence, intensity, or other affect level. Finally, with this annotated data, they trained a linear regression model that was used to predict the affect axis values for all new words. In this way they learned affective vector spaces.

In a knowledge vector space, each word is represented by a set of features that form the vector space. These features will represent well-known characteristics

about each word that can be used when processing text to obtain higher understanding of it. For example, the word “depressed” represents a state of a person. Therefore, this word in a knowledge based vector space could be represented by additional words such as: medical state, mental issue, physical issue, related to feelings, physical entity, etc. The word can then be represented as a feature vector using binary values or real values in a scale from 0-1. The following is a possible representation of the vector. Constructing the knowledge vector spaces is an important task that, in some cases, will be performed automatically and semi-manually but that will require validation by human annotators.

$$\text{Depressed} = [1, 1, 0, 1, 0, \text{etc.}]$$

The representation of this knowledge is very important and will be useful for feature extraction and representation. It will also be useful for pattern discovery via unsupervised learning methods.

A more advanced approach to add knowledge into word embeddings is to use additional optimization criteria in the cost functions of the word embedding models (Ling et al. 2017). For example, at a very high level, the criteria in a word embedding model may be to optimize the probability of predicting a word given a context of words such as:

$$J = \text{armax } P(w_t | w_{t-1}, w_{t+1})$$

To add knowledge, the objective function can be extended by adding a regularization parameter that accounts for knowledge such as the distance between



the word and other context words in a knowledge base. Based on this, the objective function could be re-written as:

$$J = \text{arimax } P(w_t | w_{t-1}, w_{t+1}) - u D_k(w_t, [w_{t-1}, w_{t+1}])$$

where  $D_k$  is some knowledge based distance function and  $u$  is a cost parameter. There are many different parameters that can be explored to adjust these models. The parameters include: different knowledge bases, different distance metrics ( $D_k$ ), different ways of representing distance in the knowledge bases (e.g. distance between nodes), different optimization functions and costs, and different word embeddings models (Word2vec, FastText, WordRank, etc.), etc.

## 5.6 Autoencoders

Autoencoders are a type of compression method where a neural network learns how to represent a vector of size “ $m$ ” into a vector of size “ $n$ ” where  $m \gg n$ . Here, the input and output vectors in the network are the original sample and the reproduced sample and the hidden layer of the network is the new compressed representation of the input vector. The objective function minimizes the difference/distance between the original input sample and the reproduced output sample.

In this section I will describe an auto-encoder. And in particular a denoising auto encoder. Usually the input and output vector are the same vector in an autoencoder. In the case of a denoising algorithm, think of the input vector  $x$  and the output vector  $y$  as the same one.

$$x = y$$

except that the  $x$  vector has noise. Therefore, this is more like

$$x + \text{random\_noise} \sim y$$

so they are not exactly alike. Think of television. You have 2 versions of the image the transmitted image via wireless ( $x + \text{noise}$ ) and the received image after denoising (with a perfect denoiser,  $y$  which is equal to  $x$  after the noise is removed). In reality the noise is not removed completely.

Also, remember that auto encoders are an unsupervised approach. Therefore, there are no labels. Instead the inputs and outputs are the same thing.

Okay, now for the code. In the next code segment we initialize the libraries. This code in full is available on the course github.

```
import tensorflow as tf
import numpy as np
import math

import scipy.spatial.distance as distance
```

We will create random data for ease of use. Notice the data is 5x4 in size. This could be an image.

```
## data

input = np.array([[ 2.0,  1.0,  1.0,  2.0],
                  [-2.0,  1.0, -1.0,  2.0],
                  [ 0.0,  1.0,  0.0,  2.0],
                  [ 0.0, -1.0,  0.0, -2.0],
                  [ 0.0, -1.0,  0.0, -2.0]])

print(input)
```

Now we add the noise to the original input. We make the input without noise equal to the output given that our goal is to remove noise. We want our inference function

to learn to remove noise so our output target should be data without noise. The data with noise will be the inputs to the network.

```
noisy_input = input + 0.2 * np.random.random_sample((input.shape)) - 0.1  
  
print noisy_input  
  
output = input
```

We normalize the data. We can scale it to [0..1]

```
# normalizing  
## scale to [0, 1]  
  
scaled_input_1 = np.divide((noisy_input - noisy_input.min()),  
                           (noisy_input.max()-noisy_input.min()))  
scaled_output_1 = np.divide( (output-output.min()), (output.max()-output.min()))  
  
print scaled_input_1  
print scaled_output_1
```

Or scale it to [-1..1]

```
## scale to [-1, 1]  
  
scaled_input_2 = (scaled_input_1*2)-1  
scaled_output_2 = (scaled_output_1*2)-1  
  
print scaled_input_2  
print scaled_output_2
```

Now we rename the scaled data.

```
## data set to use

input_data = scaled_input_2
output_data = scaled_output_2
```

Here we can print the dimensions.

```
print input_data.shape
print output_data.shape

n_sample, n_features = input_data.shape
```

Now we can create the place holders. Notice they both have the same dimensions.

```
x = tf.placeholder("float", [None, n_features]) ## (None, 4)
y_ = tf.placeholder("float", [None, n_features]) ## (None, 4)
```

Now we are ready to define the inference function. This network has a input vector of size 4, a hidden layer of size 3, and an output of size 4. We are using tanh as the activation function.

```

## nn with 1 hidden layer, neural net 4x3x4

def inference(x):
    Wh = tf.Variable( tf.random_uniform( (4,3),
                                         -1.0 / math.sqrt(n_features), 1.0 / math.sqrt(n_features) ),
                     dtype='float32' )
    bh = tf.Variable( tf.zeros([3]) , dtype='float32' )
    h1 = tf.nn.tanh( tf.matmul(x, Wh) + bh )

    #Wy = tf.transpose(Wh) #tied weights

    Wy = tf.Variable( tf.random_uniform( (3,4),
                                         -1.0 / math.sqrt(n_features), 1.0 / math.sqrt(n_features) ),
                     dtype='float32' )
    by = tf.Variable( tf.zeros([4]) , dtype='float32' )
    y = tf.nn.tanh( tf.matmul(h1, Wy) + by )

    return y, Wh, bh, h1

```

We define the loss function using Least Square Errors (LSE).

```

def loss_lse(y, y_):
    meansq = tf.reduce_mean( tf.square(y_ - y) ) ## lse
    return meansq

```

We could also use cross entropy.

```

def loss_cross_entropy(y, y_):
    xentropy = tf.nn.softmax_cross_entropy_with_logits(logits=y, labels=y_)
    loss = tf.reduce_mean(xentropy)
    return loss

```

Here we define the train function using gradient descent.

```
def train(cost):  
    train_step = tf.train.GradientDescentOptimizer(0.05).minimize(cost)  
    return train_step
```

Now we are ready to call the core functions.

```
y, Wh, bh, h1 = inference(x)  
  
cost_lse = loss_lse(y, y_)  
#cost_xentropy = loss_cross_entropy(y, y_) ## alternative  
  
train_op = train(cost_lse)
```

We are almost done, now we initialize variables.

```
init = tf.initialize_all_variables()  
sess = tf.Session()  
sess.run(init)
```

Here we set the number of epochs and the batch size.

```
n_epochs = 5000  
batch_size = min(50, n_sample) ## (50, 5)
```

Finally, we define the main loop for learning and print the losses.

```
for i in range(n_epochs):
    sample = np.random.randint(n_sample, size=batch_size)
    batch_xs = input_data[sample,:]
    batch_ys = output_data[sample,:]
    result = sess.run(train_op, feed_dict={x: batch_xs, y_:batch_ys})
    if i % 100 == 0:
        print i
        print sess.run(cost_xentropy, feed_dict={x:batch_xs,y_:batch_ys})
        print sess.run(cost_lse, feed_dict={x:batch_xs , y_:batch_ys})
```

After training, we predict denoised data.

```
print "output_data (no noise)"
print output_data

print "input_data (noisy data)"
print input_data

print "predicted_output_data (de-noised data)"
predicted_output_data = sess.run(y, feed_dict={x: input_data})
print predicted_output_data
```

The following code segment compares the distances between:

- a) the noisy input and the image without noise
- b) the noisy input and the de-noised image

The de-noised image should be closer to the noisy image because the denoising auto encoder tries to remove some noise.

range: noisy\_image.....de-noised\_image.....no\_noise\_image

The code is as follows:

```
## this code compares the distances between:
## a) the noisy input and the image without noise
## b) the noisy input and the de-noised image
## the de-noised image should be closer to the noisy image because the denoising AE
## tries to remove some noise
## range: noisy_image.....de-noised_image.....no_noise_image

dist = 0.0
print "distance between input_data (noisy) and output_data (no noise)"
for j in range(n_sample):
    res1 = distance.euclidean(input_data[j,:], output_data[j,:])
    dist = dist + res1
print dist

#####

dist_denoise = 0.0
print "distance between input_data (noisy) and predicted_output_data (after de-noising)"
for j in range(n_sample):
    res2 = distance.euclidean(input_data[j,:], predicted_output_data[j,:])
    dist_denoise = dist_denoise + res2
print dist_denoise
```

Finally, we can also print the weights, biases, and hidden layer of the model after training.

```
print "Wh (weights of encoder)"
print sess.run(Wh)

print "bh"
print sess.run(bh)

print "h1 (compressed version of x)"
print sess.run(h1, feed_dict={x: input_data})
```

That is it!



## 5.7 Deep Gramulator

The deep gramulator approach is a combination of the well proven principles of the gramulator (as previously defined) with the abilities of word embedding techniques such as word2vec.

As previously defined, the gramulator is a feature extraction technique used particularly in natural language processing. The main idea is that, for a 2 class problem, you want to extract features (e.g. words) that are very frequent in 1 class but infrequent in the other. This helps to better discriminate between the classes. The downside of this approach, however, is that it needs a lot of annotated or labeled data to extract the grams or words from each class that are infrequent in the opposite class. Assuming that the grams are representative of the entire population; then, it can be expected that a classifier will have good performance in the classification task. So the problem is how to get these optimal sets of grams when we do not have that much annotated data. Here, word embeddings can help. As previously mentioned, word embeddings can use the co-occurrence of words in a large data set of documents to create a vector space that can represent some aspects of this relationship.

Combining these two frameworks, the gramulator and word embeddings, we can naturally develop the deep gramulator which can be used to find grams from large amounts of un-labeled data by using a small set of annotated text. The algorithm creates the vector space with both text sets (labeled and un-labeled) and then proceeds to find the closest grams from un-labeled texts to the grams of labeled texts. In this case, from a set of originally labeled grams, an expanded set can be obtained. For example, given a 2 class problem, we can use a small set of grams

that are very frequent in class 1 but infrequent in class 2, to find more terms in the unlabeled data that might be related to class 1. As a result, you can obtain an expanded set of grams.

In the rest of this section, the deep gramulator algorithm will be discussed. The first part of the code for the deep gramulator shows the use of the Porter stemmer to shorten words. The porter stemmer can take several words such as happiness, happy, happiest, etc. and shorten them to their root or stem (e.g. happ). This is a very useful thing to do as it can result in more matches when the words are being compared.

```
from nltk.stem.porter import *  
  
stemmer = PorterStemmer()
```

The next part of the algorithm focuses on reading the word vectors from the vector space file “**word2vec\_vector\_space.txt**”. This assumes, of course, that the set of text documents has already been processed through word2vec and that the word vectors are stored in “**word2vec\_vector\_space.txt**”.

Notice here that each word is represented as a vector where the first column is the word and the columns 1 through 128 are the word2vec features that represent the word. Therefore, to extract the features we can use **Vectors = df.ix[:,1:128]** and to extract the words we can use **Words = df.ix[:,0]**. Finally, the feature vectors in **Vectors** are converted to a numpy matrix.

```

def get_list(path):
    the_list = []
    f = open(path, 'r')
    for line in f.readlines():
        word = line.replace("\n", "")
        if word not in the_list:
            the_list.append(word)
    return the_list

#####
## load problem data

path = '/home/user/DeepGramulator/ word2vec_vector_space.txt'
df = pd.read_csv(path, sep=',')

Vectors = df.ix[:, 1:128]
Words = df.ix[:, 0]

numpyMatrix = Vectors.as_matrix()

```

Once the vectors have been extracted, the next step is to perform the similarity calculation between all the vectors (i.e. all the words). Here, the most efficient approach is to perform a multiplication of all the word vectors at the same time using the numpy matrix. The following line of code

```
similarity = np.asarray(np.asmatrix(numpyMatrix) * np.asmatrix(numpyMatrix).T)
```

performs the computation. As can be seen, the key aspect is that the matrix **numpyMatrix** is multiplied by the transpose of itself. This operation can also be expressed as follows

$$s = M \cdot M^T$$

and can be further expressed for each pair of vectors as a cosine similarity metric given as

$$a \cdot b = \|a\| \|b\| \cos \theta$$

Once the similarity metric is calculated between all vectors, the result is stored in the variable **similarity**. The results are then converted into a pandas frame with

```
similarity_df = pd.DataFrame(similarity)
```

for ease of processing later.

```
## similarity

similarity = np.asarray(np.asmatrix(numpyMatrix) * np.asmatrix(numpyMatrix).T )
similarity_df = pd.DataFrame(similarity)
#print pd.DataFrame(similarity, index=Words, columns=Words).ix[1:10,1:10]
```

The next step in the process is to obtain the list of words (seed list) that have already been annotated as being either from one class or the other. The words are stored in the lists **auto\_yes\_train\_stemmed** and **auto\_no\_train\_stemmed** after being stemmed.

```
auto_yes_train = get_list('/home/user/DeepGramulator/train_auto_yes_gramulator_file.txt')
auto_no_train = get_list('/home/user/DeepGramulator/train_auto_no_gramulator_file.txt')

auto_yes_train_stemmed = [stemmer.stem(term) for term in auto_yes_train]
auto_no_train_stemmed = [stemmer.stem(term) for term in auto_no_train]
```

Once the words have been extracted, the next step in the process is to obtain the indices that match these words in the similarity matrix **similarity\_df**. To achieve this, the algorithm iterates through the words in the **Words** vector (previously defined) and checks to see if they are present in the lists of words per class (**auto\_yes\_train** and **auto\_no\_train**). This assumes that the words in the vector **Words** are aligned with the vectors in the matrix **similarity\_df**. As a result, 2 (or “n” depending on the number of classes) new lists are created where each contains the indices of the words in the vector space that correspond to the given class. For

this example, the 2 lists **indices\_auto\_yes\_train** and **indices\_auto\_no\_train** are obtained.

```
indices_auto_yes_train
    = [int(i) for i in range(len(auto_yes_train)) if stemmer.stem(Words[i].lower()) in auto_yes_train]
indices_auto_no_train
    = [int(i) for i in range(len(auto_no_train)) if stemmer.stem(Words[i].lower()) in auto_no_train]
```

The next step in the algorithm is to use the selected indices per class to slice the matrix **similarity\_df**. This will result in 2 matrices (1 per class assuming a 2 class problem) that contain the vectors of the words that are frequent in 1 class while infrequent in the other. The vectors are stored in **selected\_rows\_auto\_yes\_train** and **selected\_rows\_auto\_no\_train**.

```
selected_rows_auto_yes_train = similarity_df.ix[indices_auto_yes_train]
selected_rows_auto_no_train = similarity_df.ix[indices_auto_no_train]
```

With the 2 matrices from the previous step (**selected\_rows\_auto\_yes\_train** and **selected\_rows\_auto\_no\_train**) we can now obtain the words that are closest to these per class words using the code below.

```
result_test_auto_yes_similar = get_results_list(selected_rows_auto_yes_train)
result_test_auto_no_similar = get_results_list(selected_rows_auto_no_train)
```

In this case, the **get\_results\_list()** function can be defined as follows. Here, the code would take all the values in the sliced class similarity matrix and select the indices of the top 5 largest (i.e. most similar). Therefore, the function can obtain the 5 most similar words to a given annotated word per class.

```
top_n = 5
result_auto_train_yes = pd.DataFrame({
    n: selected_rows_auto_yes_train.T[col].nlargest(top_n).index.tolist()
    for n, col in enumerate(selected_rows_auto_yes_train.T)})
).T
```

Alternatively, we can define **get\_results\_list()** in a more detailed way to provide more intuition about the process. In this case, the function is passed the indices of the annotated words in **indices\_auto\_train**. Given these indices, the function will iterate through all rows in the similarity matrix that match that index. Once the row is obtained, all values in the row are saved to a list (**row\_list**) for further processing. The row list is then converted into a numpy array using the statement

**vals = np.array(row\_list)**

the next step is to sort the values and select the top 10 most similar. We can sort the values per row using the following statement:

**sort\_index = np.argsort(vals)**

once the values are sorted we can simply obtain the set using the statement

**top\_indeces = sort\_index[start\_index:end\_index]**

the final step is to use those indices to slice the words from the vector **Words**. Those words are now compared to the original lists to ensure that there are no duplicates (e.g. if word not in `auto_yes_train`). Stop words are also not used.

```
def get_results_list(indices_auto_train, top_n = 10):
    result_test_auto_similar = []
    for index in indices_auto_train:
        row = similarity_df.ix[index]
        row_list = row.values.tolist()
        vals = np.array(row_list)
        sort_index = np.argsort(vals)
        start_index = len(Words) - top_n
        end_index = len(Words)
        top_indeces = sort_index[start_index:end_index]
        Selected_words = Words[top_indeces]
        for word in Selected_words:
            if word not in auto_yes_train:
                if word not in auto_no_train:
                    if word not in nltk.corpus.stopwords.words('english'):
                        if word not in result_test_auto_similar:
                            result_test_auto_similar.append(word)

    return result_test_auto_similar
```

As a result, we now have the list of words that are closest to the annotated words per class. We can then write these out to a file or use them in the next step of our classifier.

## 5.8 Summary

In this chapter, the topic of semantic vector spaces was presented as well as its relationship to neural networks and Tensorflow. The vector space model was first introduced to provide the basic framework for other vector spaces based approaches. After discussing the vector space model, a traditional semantic vector

space technique called latent semantic analysis (LSA) was also presented. After this, a newer framework called word2vec was discussed. This framework although not a true deep neural network represents an example of the new generation of techniques that have been made available with the advances in deep learning theory and deep learning frameworks like Tensorflow. Finally, the chapter presented an example (using the deep gramulator) of how deep learning based techniques and tools can help to automatically obtain features from text for problems related to natural language processing.



## CHAPTER 6: CONVOLUTIONAL NEURAL NETWORKS

In this this chapter, I will focus on a more advanced topic in deep learning called Convolutional Neural Networks (CNNs). This type of technique has been used extensively for image processing. It has the capability to learn the features from the image data without human intervention. The explanations in this chapter assume that you have read previous chapters of this book. I will re-use a lot of the code we previously used to define our logistic regression and deep neural network algorithms. There will be a few new functions to define a convolutional network architecture and to perform some new operations but you will notice how much of the CNN code is similar do what we have done in previous chapters. For the discussion, I will use the Mnist data set. Finally, the code in this chapter can be downloaded from GitHub at <https://github.com/rcalix1>.

### 6.1 Loading the Mnist Data

The Mnist data set is a well known annotated dataset containing images of hand written digits in the range of 0, 1,...,9. The data set consists of a train set, a validation set, and a test set. It has around 70,000 images of dimension 28x28 in grey scale.

There are no new libraries for CNNs and instead we will rely mainly on the Tensorflow module.

```
import tensorflow as tf
```

The first step in our process is to obtain the data. In this section, for the sake of simplicity, I will not read the data from .csv files. Instead, we are going to obtain it directly from the **tensorflow.examples** module.

```
# load MNIST data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)
```

Notice that in this code the data and labels are stored in the **mnist** object and that the labels are already one-hot encoded.

## 6.2 Convolution and CNNs Defined

So, what is a convolution? Convolution is a mathematical operation between two functions **f** and **g** to produce a new modified function ( $f * g$ ). It is a special kind of operation that involves the multiplication of 2 input functions with some additional conditions. As an example, in image processing this could mean the convolution between function **g** (an image) with a function **f** (a filter) to produce a new modified version of the image.

Image processing uses filters to identify features in images such as for edge detection. Edge detecting filters, for instance, look for areas in an image of high variation to identify edges. That is, where the values of the pixels are all about the same may be considered a background but where the values are consistent and then start changing may mean that an edge is detected.

In the previous chapters, we have multiplied a data set matrix **X** with a weight matrix **W**. Applying a filter to an image is a similar process where you multiply

(using a convolution operation) an image matrix (equivalent to  $X$ ) with a filter matrix (similar to the weights). In fact, there are many types of filters that could be defined for image processing. In the past, these filters had to be defined by human feature engineers. The insight given by convolutional neural networks is that, given training data with labels, these filters (the convolution filters) can be learned by the model by learning the weights. And because the neural networks have multiple layers, convolutional filters learned from one layer can be used to transform inputs for the following layer.

### 6.3 Architecture for a Convolutional Neural Network

In this section, I will provide the main description of the architecture of the convolutional neural network and show some diagrams to better interpret the intuition of convolutional neural networks. The diagram below shows an overview of the model we are going to build to perform image classification of hand written digits.

The CNN we are going to use as our example consists of the following layers:

- input layer (the image)
- convolutional layer 1
- convolutional layer 2
- fully connected layer
- output layer

There are many details that could be described to define the architecture of a convolutional neural network. However, implementing one in Tensorflow is not that difficult. Defining the architecture of a CNN is similar to how we defined the

architecture of our previous deep learning classifiers. We need to define the number of layers and the size of each layer. In our previous layer definitions, the matrix multiplication was our main operation. When implementing a convolutional layer the main operation is a convolution. For convenience, here we will think of convolutional layers as black boxes of filters with inputs and outputs. Therefore, whenever we define a convolutional layer we need to define the following:

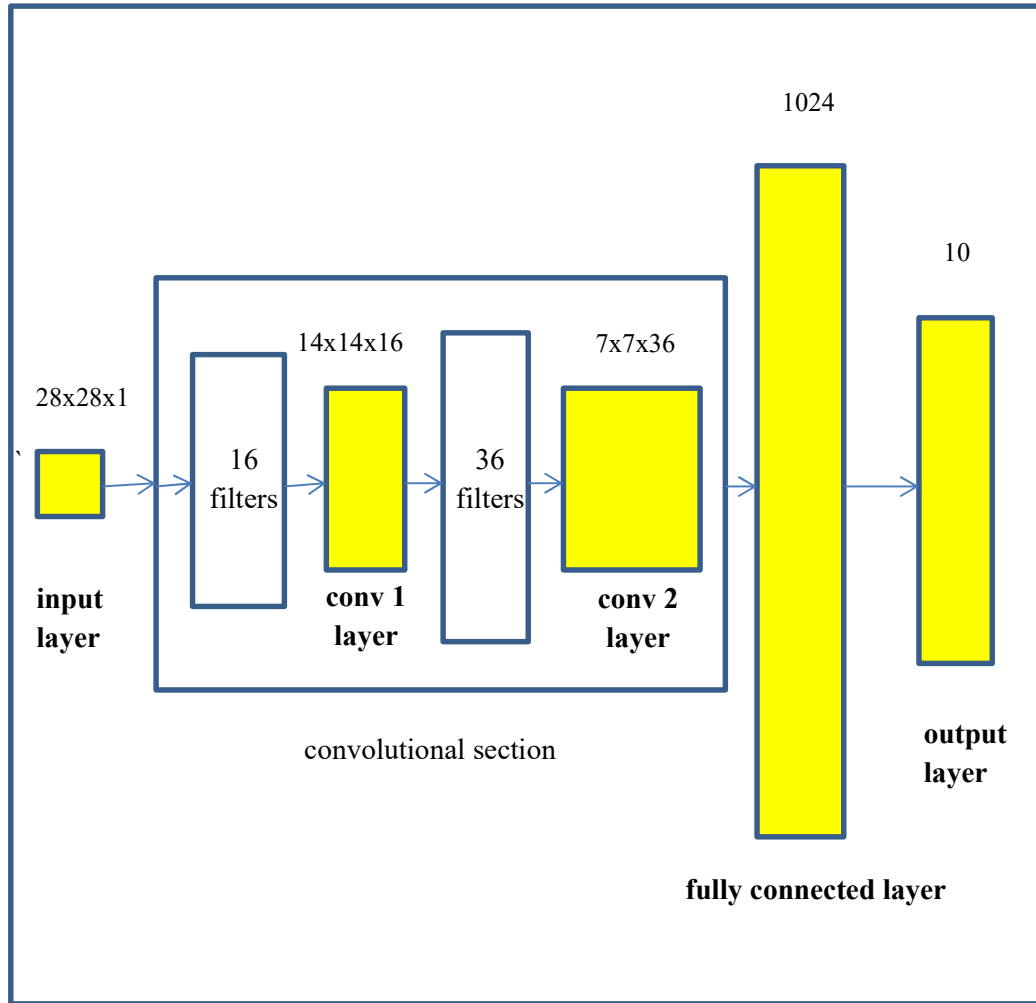
- The number of inputs and the size of each input. In this case, the size of each input refers to the size of the image. In the case of Mnist, the images are  $28 \times 28$  each. The number of inputs refers to the number of channels for the image. For instance, 1 channel for grey scale images and 3 channels for RGB or color images (color images are actually 3 matrices of size  $28 \times 28$ ).
- The number of filters. This is a value that is defined by the network architect. For example, 24 filters or 16 filters. These are the convolutional filters which will be applied to the images. In the case of 16 filters, it means that 16 different filters would be applied to 1 input image to produce 16 new processed images (these new 16 processed versions of the input images would be referred to as producing 16 output channels). The size of the filter is also defined (for instance a filter of  $5 \times 5$ ).
- The number of outputs. As indicated in the previous bullet, the outputs are referred to as output channels and consist of the processed images after convolution.

It is important to note that the convolution process is a bit more complicated when the number of inputs is more than 1; for example, when a convolutional layer has

16 input channels (16 versions of the input image) to the layer and 36 filters to be applied. In this case each input channel needs to be processed by all 36 filters. In the end, the layer will output 36 processed images. Additionally, it is important to note that the output images may not always retain their original size (e.g.  $28 \times 28$  for mnist). Instead, after each layer, the processed images may be down sampled. This is called maxpooling. In our example, we will down sample from  $28 \times 28$  to  $14 \times 14$  and then to  $7 \times 7$ .

So let us discuss the architecture of the CNN we are going to implement. The CNN will have the following characteristics.

- 1 input layer, 2 convolutional layers, 1 fully connected layer, and 1 output layer
- The input layer will consist of images of  $28 \times 28$  with 1 channel
- The first convolutional layer will have 16 filters. Each filter will be of size  $5 \times 5$ . Given that it has 16 filters, its output will be 16 processed images or 16 channels. The images will be down sampled to  $14 \times 14$ .
- The second convolutional layer will have 36 filters. The filters will be  $5 \times 5$ . Its output will be 36 processed images or 36 channels. The images will be down sampled to  $7 \times 7$ .
- The fully connected layer will have 1024 neurons. This is a normal layer that will connect the output of the second convolutional layer to the output layer.
- The output layer has 10 nodes which represent the 10 classes in the mnist dataset.



**Figure. CNN Architecture.**

Based on the previous characteristics (see figure above), we can define our network architecture dimensions as follows:

- Input layer: 28x28x1
- Conv layer 1 output: 14x14x16
- Conv layer 2 output: 7x7x36
- Fully connected layer: 1024
- Output layer: 10

The figure above presents a more visual representation of our example convolutional neural network.

## 6.4 Code for a Convolutional Neural Network

Once we have defined the architecture, we are ready to start coding our CNN. The code in this chapter is very similar to code from previous chapters. Therefore, I will only focus on new aspects of the code and will try not to repeat descriptions that have been provided in previous chapters.

The first part of the code that needs to be defined is the section used to set the algorithm parameters.

```
#parameters for the main loop
learning_rate = 0.001
n_epochs = 200000 ##27000
batch_size = 128
display_step = 10

# Parameters for the network
n_input = 784 # MNIST has 784 features, each image has shape of
28*28
n_classes = 10 # MNIST (0-9 digits)
dropout = 0.75 # Dropout, probability to keep units
```

Most of the parameters defined in this section are similar to parameters we have used in previous chapters. The only new parameter is the dropout. The dropout is a parameter for a technique first defined by Geoffrey Hinton. Dropout is a technique that helps to perform a better optimization during the weight search.

#### 6.4.1 Convolution and Maxpool Operations

At the heart of a CNN there are 2 main operations which are the convolution and the maxpool operation. The convolution code can be seen below. As can be seen, it takes **x** and **W** and performs a convolution operation. The strides parameter defines how the filter will slide across the image (e.g. every pixel, every 2 pixels, etc.)

```
## convolution operation
def conv2d(x, W, b, strides=1):
    # Conv2D function, with bias and relu activation
    x = tf.nn.conv2d(x, W,
                     strides=[1, strides, strides, 1], padding='SAME')
    x = tf.nn.bias_add(x, b)
    return tf.nn.relu(x)
```

The maxpool operation is used to down sample the images. For instance, in the case of the mnist images which have a size of 28x28, every time the images or their filtered equivalents pass through a maxpool function, the result is that the image is reduced in size. During the first convolution (convolution layer 1), the filtered images are down sampled from 28x28 to 14x14. This is implemented with the **max\_pool()** function.



```
def maxpool2d(x, k=2):  
    # MaxPool2D function  
    return tf.nn.max_pool(x, ksize=[1, k, k, 1],  
                           strides=[1, k, k, 1], padding='SAME')
```

### 6.4.2 Layer Definitions

As previously described, there are 4 main types of layers that will be defined for the CNN. They are:

- Input layer
- convolutional layers
- fully connected layer
- output layer

These layers are the key components that help to connect the outputs of one layer to the inputs of the next. We will create these layers using the functions

- layer()
- conv\_layer()
- fully\_connected\_layer()

```
def layer(input, weight_shape, bias_shape):
    W = tf.Variable(tf.random_normal(weight_shape))
    b = tf.Variable(tf.random_normal(bias_shape))
    mapping = tf.matmul(input, W)
    result = tf.add( mapping , b )
    return result
```

As can be seen above, the function **layer()** is the same one that we have used in previous chapters. This is a standard layer that performs a matrix multiplication between the weights and the input and adds a bias. We will use this function in the last part of the architecture to connect the fully connected layer to the output layer (i.e. the layer with the 10 classes). The `weight_shape` and `bias_shape` parameters will help define the dimensions of the layer.

To define the convolutional layers we will use the function **conv\_layer()**. This function takes an **input** parameter and a weight variable **W** and performs a convolution operation using our previously defined function **conv2d()**. Notice that the `max_pool` function is applied here to the variable **conv** to down sample the output images by a factor of 2.

```
def conv_layer(input, weight_shape, bias_shape):
    W = tf.Variable(tf.random_normal(weight_shape))
    b = tf.Variable(tf.random_normal(bias_shape))
    conv = conv2d(input, W, b)
    # Max Pooling (down-sampling)
    conv_max = maxpool2d(conv, k=2)
    return conv_max
```

The next section of code describes the fully connected layer. This layer re-shapes the convoluted images and maps them to a hidden layer of **n** neurons.

```
# fc_weight_shape = [7x7x36, 1024]
def
fully_connected_layer(conv_input,fc_weight_shape,fc_bias_shape,dropout):
    temp = tf.Variable(tf.random_normal(fc_weight_shape))
    new_shape = [ -1, temp.get_shape().as_list()[0] ] # [-1, 1764]
    fc = tf.reshape(conv_input, new_shape)
    mapping = tf.matmul(fc,tf.Variable(tf.random_normal( fc_weight_shape)))
    fc = tf.add( mapping, tf.Variable(tf.random_normal(fc_bias_shape)))
    fc = tf.nn.relu(fc)
    # Apply Dropout
    fc = tf.nn.dropout(fc, dropout)
    return fc
```

In the following line of code

```
new_shape = [ -1, temp.get_shape().as_list()[0] ] # [-1, 1764]
```

the -1 means auto expand. Therefore, the code should infer the value for that dimension. For this example, [-1, 1764] is equal to [1, 1764].

Notice that this layer uses standard functions we have seen before such as **tf.matmul()** and **tf.nn.relu()**. The dropout is also used here to improve the optimization.

### 6.4.3 Defining the CNN Architecture

This is the part of the code where the dimensions of the CNN must be defined. These include the input and output sizes per layer as well as the size of the convolution filters to be used in the convolution layers. For this example, all filters in all conv layers will be of size 5x5.

In the code segments below, I have provided two examples with somewhat different architectures to show how the architectures and dimensions of the CNN can be defined.

```
def inference_conv_net2(x, dropout):
    # Reshape input picture
    # shape=[-1, size_img_x, size_img_y, 1 channel (e.g. grey scale)]
    # here -1 infers the batch size
    x = tf.reshape(x, shape=[-1, 28, 28, 1])
    # Convolution Layer 1, filter 5x5 conv, 1 input, 16 outputs
    # max pool will reduce image from 28*28 to 14*14
    conv1 = conv_layer(x, [5, 5, 1, 16], [16] )

    # Convolution Layer 2, filter 5x5 conv, 16 inputs, 36 outputs
    # max pool will reduce image from 14*14 to 7*7
    conv2 = conv_layer(conv1, [5, 5, 16, 36], [36] )

    # Fully connected layer, 7*7*36 inputs, 1024 outputs
    # Reshape conv2 output to fit fully connected layer input
    fc1 = fully_connected_layer(conv2, [7*7*36, 1024], [1024] , dropout)

    # Output, 1024 inputs, 10 outputs (class prediction)
    output = layer(fc1 ,[1024, n_classes], [n_classes] )
    return output
```

The code above uses the function `inference_conv_net2()` to define the network. The first step is to reshape the input picture. The values in the shape parameter indicate that the image is 28x28 and has 1 channel (e.g. grey scale). The convolution layer 1 uses filters of 5x5 to process the images. It takes 1 input from `x` and outputs 16 processed images. The convolution layer 2 uses filters of 5x5 to process the images. It takes 16 input images or channels from `conv1` and outputs 36 processed images. The fully connected layer will take  $7*7*36$  inputs from `conv2` and connect them to a layer of 1024 neurons. The output layer will connect 1024 neurons to 10 neurons (the classes).

```
def inference_conv_net(x, dropout):
    # Reshape input picture
    # shape = [-1, size_image_x, size_image_y, 1 channel (e.g. grey scale)]
    # here -1 infers the batch size
    x = tf.reshape(x, shape=[-1, 28, 28, 1])
    # Convolution Layer 1, 5x5 conv, 1 input, 32 outputs
    conv1 = conv_layer(x, [5, 5, 1, 32], [32] )

    # Convolution Layer 2, 5x5 conv, 32 inputs, 64 outputs
    conv2 = conv_layer(conv1, [5, 5, 32, 64], [64] )

    # Fully connected layer, 7*7*64 inputs, 1024 outputs
    # Reshape conv2 output to fit fully connected layer input
    fc1 = fully_connected_layer(conv2, [7*7*64, 1024], [1024] , dropout)

    # Output, 1024 inputs, 10 outputs (class prediction)
    output = layer(fc1 ,[1024, n_classes], [n_classes] )
    return output
```

The code above uses the function `inference_conv_net()` to define the network. The first step is to reshape the input picture. The values in the shape parameter indicate that the image is 28x28 and has 1 channel (e.g. grey scale). The convolution layer 1 uses filters of 5x5 to process the images. It takes 1 input from `x` and outputs 32 processed images. The convolution layer 2 uses filters of 5x5 to process the images. It takes 32 input images or channels from `conv1` and outputs 64 processed images. The fully connected layer will take  $7*7*64$  inputs from `conv2` and connect them to a layer of 1024 neurons. The output layer will connect 1024 neurons to 10 neurons (the classes).

#### 6.4.4 Definition of the Loss, Optimization, and Evaluation Functions

In this section, we have the definitions of the loss, optimization, and evaluation functions. These definitions are very similar to what we have done in the previous chapters of this book.

```
def loss_deep_conv_net(output, y_tf):  
    xentropy = tf.nn.softmax_cross_entropy_with_logits(  
                                                output, y_tf)  
    loss = tf.reduce_mean(xentropy)  
    return loss
```

The training function uses an optimizer called the Adam Optimizer. This optimizer is considered to be better than the regular Gradient Descent optimizer.

```
def training(cost):
    optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
    #optimizer = tf.train.GradientDescentOptimizer(learning_rate)
    train_op = optimizer.minimize(cost)
    return train_op
```

The evaluate function has not changed from our previous implementations. As you can see, a lot of the code we learned in chapter 4 is being re-used in this chapter.

```
def evaluate(output, y_tf):
    correct_prediction = tf.equal(tf.argmax(output,1), tf.argmax(y_tf,1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
    return accuracy
```

Once the functions are defined, the next step in the process is to define the data and label placeholders and to call the functions. This is the core section of the CNN and is described in the next section.

#### 6.4.5 The Core of the Deep Neural Network

In this section, we define the place holders for the data **x\_tf** and the labels **y\_tf**. We also define the **keep\_prob** place holder which will be used for the dropout technique.

```

x_tf = tf.placeholder(tf.float32, [None, n_input])
y_tf = tf.placeholder(tf.float32, [None, n_classes])
keep_prob = tf.placeholder(tf.float32) #dropout
#####

output = inference_conv_net2(x_tf, keep_prob)
#output = inference_conv_net(x_tf, keep_prob)
cost = loss_deep_conv_net(output, y_tf)
train_op = training(cost)
eval_op = evaluate(output, y_tf)

```

The last part in the previous code section calls the functions to define the architecture, create the equation, and optimize the equation using the loss function definition. Much of this code is similar to what we have learned in previous chapters.

#### 6.4.6 Initializing Variables and the Session

Once everything in the graph is defined we can proceed to initialize the variables and create the session.

```

## for performance metrics
y_p_metrics = tf.argmax(output, 1)

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

```



Finally, the last part is to call the main loop to perform the training and testing of the model using the Mnist dataset.

#### 6.4.7 The Main Loop

This is the last part of the code that we need to define. If you look closely, you will see that it is exactly what we did in the previous chapters.

```
dropout2 = 1.0
step = 1
# Keep training until reach max iterations (n_epochs)
while step * batch_size < n_epochs:
    batch_x, batch_y = mnist.train.next_batch(batch_size)
    sess.run(train_op, feed_dict={x_tf: batch_x, y_tf: batch_y,
                                   keep_prob: dropout})

    loss, acc = sess.run([cost, eval_op], feed_dict={x_tf: batch_x,
                                                       y_tf: batch_y,
                                                       keep_prob: dropout2})

    result = sess.run(eval_op, feed_dict={x_tf: mnist.test.images[:256],
                                           y_tf: mnist.test.labels[:256],
                                           keep_prob: dropout2})

    result2, y_result_metrics = sess.run([eval_op, y_p_metrics],
                                         feed_dict={x_tf: mnist.test.images[:256],
                                                     y_tf: mnist.test.labels[:256],
                                                     keep_prob: dropout2})

    print "test1 {},{}".format(step,result)
    print "test2 {},{}".format(step,result2)
    y_true = np.argmax(mnist.test.labels[:256],1)
    print_stats_metrics(y_true, y_result_metrics)

    if step == 1000:
        plot_metric_per_epoch()

    step = step + 1
```

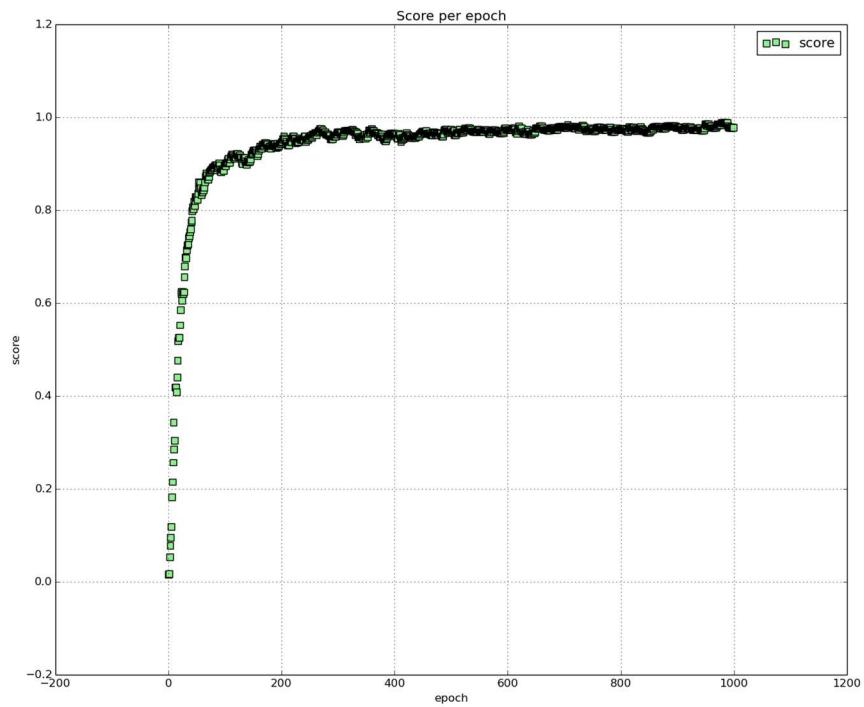
One modification is that we use Mnist's batching mechanism to simplify the process of reading in the data.

Once we run the code, the model should perform really well and we should obtain high classification accuracy scores for the Mnist dataset. I will discuss some of the results in the next section.

#### **6.4.8 Classification Results for the CNN**

The CNN model we defined in this chapter should perform really well. You should get accuracies as high as 99%. The graph below shows how the model performance improves as it learns from the training samples.

The model stabilizes around epoch 30.



**Figure. Performance per epoch of the CNN.**

As can be seen in the output below, the classifier does really well around the iteration 1562. All scores such as precision, accuracy, and f1-measure are high.

```
Iteration 1562
Accuracy: 0.98
Precision: 0.981
Recall: 0.980
F1-measure: 0.980
```

The next section of output shows the confusion matrix for a test set of 256 samples. There are very few misclassification errors.

confusion matrix											
Predicted	0	1	2	3	4	5	6	7	8	9	All
True											
0	19	0	0	0	0	0	0	0	0	0	19
1	0	35	0	0	0	0	0	0	0	0	35
2	0	0	24	0	0	0	0	0	0	0	24
3	0	0	0	23	0	0	0	0	0	0	23
4	0	1	0	0	32	0	0	0	0	1	34
5	0	0	0	0	0	26	0	0	0	0	26
6	0	0	0	0	0	0	22	0	0	0	22
7	0	0	0	0	0	0	0	31	0	0	31
8	0	0	0	0	0	0	0	0	14	0	14
9	0	0	0	0	0	1	0	0	2	25	28
All	19	36	24	23	32	27	22	31	16	26	256

Misclassification examples include number 9 which was misclassified with 4, number 5 which was misclassified with 9, and the number 1 which was misclassified with 4.

## 6.5 CNNs for RGB Data

In the previous section we looked at CNNs for grey scaled data. In this section we apply CNNs to color images. The main change will relate to the fact that the inputs for grey scale are  $[n, m, 1]$  whereas inputs for color images are  $[n, m, 3]$ .

Next we describe the code. The data can be downloaded from the github. First we add the libraries as usual.

```
import sklearn
import tensorflow as tf
import numpy as np
from numpy import genfromtxt
from sklearn import datasets
#from sklearn.cross_validation import train_test_split
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score, f1_score
import pandas as pd
import matplotlib.pyplot as plt
```

Next we set parameters to remove warnings and to see all values when printing numpy arrays.

```
## set parameters

import warnings
warnings.filterwarnings("ignore")
np.set_printoptions(threshold=np.inf) #print all values in numpy array
```

Here we set the parameters for the loop. Notice that we train for 200,000 epochs. We read the data in batches of 128.

```
#parameters for the main loop  
  
learning_rate = 0.001  
n_epochs = 200000 ##27000  
batch_size = 128  
display_step = 10
```

Now it starts to get interesting. We need to set the size of the images. In MNIST our images were of size 28x28x1. In this example we use color images of size 100x100x3 (data available on the github). Given the size of each image, our number of features is 30,000 or 100x100x3. It is important that you are aware that data is fed as tensors into the computational graph. Therefore, the images or tensors may be reshaped from cubes to vectors and vice versa as needed. The number of classes for this data set is 4 given that we are trying to classify between 4 fruits. The last parameter is the dropout. Remember dropout forces the network to use fewer neurons.

```
# Parameters for the network  
#n_input = 784 # MNIST has 784 features because each image has shape of 28*28*1  
#n_input = 2352 # 28x28x3  
  
n_input = 30000 # 100x100x3  
n_classes = 4 # MNIST (0-3 fruits)  
dropout = 0.75 # Dropout, probability to keep units
```

Now we are ready to read in our data. We use PIL. Put your images in a folder named testA. We can use glob to gather the images from the folder and then PIL to read them in. The images are grouped by label into 4 different folders. Two

list for images and labels are used to store the data. I left the commented out lines so you can play with visualizing the images as you read them in.

```
## this will create your own data set (i.e. your_mnist)
## put your images in testA (I used pngs)

from PIL import Image

import glob
import numpy as np

train = []
labels = []

files = glob.glob ("testB/apple/*.jpg") # your apple images path

for myFile in files:
    my_im = Image.open(myFile).convert('RGB')  ## .convert('LA') ## is for greyscale

    #my_im.show()
    #resized_my_im = my_im.resize((28,28))  ## resize from 100x100x3 to 28x28x3
    #resized_my_im.show()
    #image = np.array(resized_my_im)

    image = np.array(my_im)
    print(image.shape)
    new_image = image.reshape(image.shape[0]*image.shape[1]*image.shape[2])
    print(new_image.shape)
    #input_string = input("???")
    train.append(new_image)
    labels.append(0)
```

The way I read the images uses 4 loops for each folder. This could have been done in many different ways. I simply thought this was the clearest approach.

The following reads in data for bananas.

```
files = glob.glob ("testB/banana/*.jpg") # your banana images path

for myFile in files:
    image = np.array(Image.open(myFile).convert('RGB'))
    print(image.shape)
    new_image = image.reshape(image.shape[0]*image.shape[1]*image.shape[2])
    print(new_image.shape)
    #input_string = input("???")
    train.append(new_image)
    labels.append(1)
```

Now we read in pepino jpgs.

```
files = glob.glob ("testB/pepino/*.jpg") # your pepino images path

for myFile in files:
    image = np.array(Image.open(myFile).convert('RGB'))
    print(image.shape)
    new_image = image.reshape(image.shape[0]*image.shape[1]*image.shape[2])
    print(new_image.shape)
    #input_string = input("???")
    train.append(new_image)
    labels.append(2)
```

Finally, we read in peppers.

```
files = glob.glob ("testB/pepper/*.jpg") # your pepper images path

for myFile in files:
    image = np.array(Image.open(myFile).convert('RGB'))
    print(image.shape)
    new_image = image.reshape(image.shape[0]*image.shape[1]*image.shape[2])
    print(new_image.shape)
    #input_string = input("???")
    train.append(new_image)
    labels.append(3)
```



After reading the images, we convert the lists to numpy arrays and visualize dimensions.

```
train = np.array(train,dtype='float32') #as mnist
labels = np.array(labels, dtype='float32')

print(train.shape)
print(labels.shape)
#print(train.shape[1])
```

You could reshape the images for various reasons. Here is an example of how to do that. Try it.

```
# convert (number of images x height x width x number of channels)
# to (number of images x (height * width * 3))
# for example (120 * 40 * 40 * 3)-> (120 * 4800)
#train = np.reshape(train,[train.shape[0],train.shape[1]*train.shape[2]*train.shape[3]])
#train = np.reshape(train,[train.shape[0],train.shape[1]*train.shape[2]])
```

After converting the images to numpy arrays, you can save them to file for later use.

```
# save numpy array as .npy formats

np.save('train', train)
```

For our purposes, I renamed the matrix as `your_mnist` so you know this is like mnist would be. Remember the data is now about fruits.

```
your_mnist = train
```

Remember to always normalize.

```
## normalization is very important

x=your_mnist
xmax, xmin = x.max(), x.min()
x = (x - xmin)/(xmax - xmin)

your_mnist = x
```

Now we perform train and test split as we previously learned.

```
x_all = your_mnist
labels_all = labels

x_train, x_test, y_train, y_test = train_test_split(x_all, labels_all, test_size=0.30,
                                                    random_state=42)
```

Here we define the performance metrics function for precision, recall, f-measure, and the confusion matrix.

```
## print stats
precision_scores_list = []
accuracy_scores_list = []

def print_stats_metrics(y_test, y_pred):
    print('Accuracy: %.2f' % accuracy_score(y_test, y_pred) )
    #Accuracy: 0.84
    accuracy_scores_list.append(accuracy_score(y_test, y_pred) )
    confmat = confusion_matrix(y_true=y_test, y_pred=y_pred)
    print "confusion matrix"
    print(confmat)
    print pd.crosstab(y_test, y_pred, rownames=['True'], colnames=['Predicted'], margins=True)
    precision_scores_list.append(precision_score(y_true=y_test, y_pred=y_pred,
                                                  average='weighted'))
    print('Precision: %.3f' % precision_score(y_true=y_test, y_pred=y_pred, average='weighted'))
    print('Recall: %.3f' % recall_score(y_true=y_test, y_pred=y_pred, average='weighted'))
    print('F1-measure: %.3f' % f1_score(y_true=y_test, y_pred=y_pred, average='weighted'))
```

The plotting function.

```
def plot_metric_per_epoch():
    x_epochs = []
    y_epochs = []
    for i, val in enumerate(accuracy_scores_list):
        x_epochs.append(i)
        y_epochs.append(val)

    plt.scatter(x_epochs, y_epochs, s=50, c='lightgreen', marker='s', label='score')
    plt.xlabel('epoch')
    plt.ylabel('score')
    plt.title('Score per epoch')
    plt.legend()
    plt.grid()
    plt.show()
```

Now we define the convolution function with strides equal to 1 and a relu function. Remember that relu removes negative values and looks like a hokey stick. Also, remember relu is not constrained to ranges such as [0..1] or [-1..1] but instead [0..infinity]. Therefore, it has more power to learn because the positive values range is much wider and has more power to learn.

```
def conv2d(x, W, b, strides=1):
    # Conv2D function, with bias and relu activation
    x = tf.nn.conv2d(x, W, strides=[1, strides, strides, 1], padding='SAME')
    x = tf.nn.bias_add(x, b)
    return tf.nn.relu(x)  ## relu removes negative values
```

The max pooling function. This problem of images that are 100x100x3 results in a max pooling situation where the images have dimensions that are uneven numbers. How do we resolve this? Easily. We use padding=SAME. So when the

image is image 25x25 maxpool would give an image that is  $\rightarrow 12.5 \times 12.5$ . Because of padding=same then it is rounded up to 13x13. Problem solved!

```
def maxpool2d(x, k=2):  
    # MaxPool2D function  
    # padding='SAME' is very useful for uneven images. If maxpooling  
    # image 25x25  $\rightarrow 12.5 \times 12.5$  then it is rounded up to 13x13  
    return tf.nn.max_pool(x, ksize=[1, k, k, 1], strides=[1, k, k, 1],  
                          padding='SAME')
```

Now we define the layer function as previously discussed which is used for the fully connected layer.

```
def layer(input, weight_shape, bias_shape):  
    W = tf.Variable(tf.random_normal(weight_shape))  
    b = tf.Variable(tf.random_normal(bias_shape))  
    mapping = tf.matmul(input, W)  
    result = tf.add(mapping, b)  
    return result
```

We define the convolutional layer function using our previous conv2d and max pool functions.

```
def conv_layer(input, weight_shape, bias_shape):  
    W = tf.Variable(tf.random_normal(weight_shape))  
    b = tf.Variable(tf.random_normal(bias_shape))  
    conv = conv2d(input, W, b)  
    # Max Pooling (down-sampling)  
    conv_max = maxpool2d(conv, k=2)  
    return conv_max
```

Next we define the fully connected layer. Notice that the -1 in the reshape allows to infer the size of that dimension.

```
def fully_connected_layer(conv_input, fc_weight_shape, fc_bias_shape, dropout):
    new_shape = [-1, tf.Variable(tf.random_normal(fc_weight_shape)).get_shape().as_list()[0]]
    fc = tf.reshape(conv_input, new_shape)
    w_fc = tf.Variable( tf.random_normal( fc_weight_shape ) )
    mapping = tf.matmul( fc , w_fc ) # y = w * x
    fc = tf.add( mapping, tf.Variable(tf.random_normal( fc_bias_shape )) )
    fc = tf.nn.relu(fc)
    # Apply Dropout
    fc = tf.nn.dropout(fc, dropout)
    return fc
```

Finally, we are ready to define the architecture. This is what you will usually need to figure out depending on your images. So we first reshape each input picture in the batch. Notice that the batch itself is a tensor of n images.

The shape is equal to

[-1, size\_image\_x, size\_image\_y, 3 channels (e.g. rgb)]

The resized image for rgb and batches of 128 would be [128, 30000] because there are 128 samples per batch and images are  $100 \times 100 \times 3 = 30000$ . This has to be reshaped because Convolutional layers only take 4 dimensional tensors as input.

The -1 infers the number of batches and then we make the 30,000 into  $100 \times 100 \times 3$ . In this first function we start with 16 filters.

```

## define the architecture here

def inference_conv_net2(x, dropout):
    # Reshape input picture
    # shape = [-1, size_image_x, size_image_y, 3 channels (e.g. rgb)]
    # the image for rgb and batches of 128 would be [128, 30000] because
    # there are 128 samples per batch and images are 100x100x3 = 30000
    # this has to be re-shaped because Convolutional layers only
    # take 4 dimensional tensors as input
    # the -1 infers the number of batches and then we make the 30,000 into 100x100x3
    x = tf.reshape(x, shape=[-1, 100, 100, 3])

    # Convolution Layer 1, filter 5x5 conv, 3 inputs or 3 channels, 16 outputs
    # max pool will reduce image from 100x100 to 50x50
    conv1 = conv_layer(x, [5, 5, 3, 16], [16] )

    # Convolution Layer 2, filter 5x5 conv, 16 inputs, 36 outputs
    # max pool will reduce image from 50x50 to 25x25
    conv2 = conv_layer(conv1, [5, 5, 16, 36], [36] )

    # Fully connected layer, 25x25*36 inputs, 128 outputs
    # Reshape conv2 output to fit fully connected layer input
    fc1 = fully_connected_layer(conv2, [25*25*36, 1024], [1024] , dropout)

    # Output, 128 inputs, 10 outputs (class prediction)
    output = layer(fc1 ,[1024, n_classes], [n_classes] )
    return output

```

The next code segment is another version of the previous function. Here we start with 32 filters instead of 16.

```
## define the architecture here

def inference_conv_net(x, dropout):
    # Reshape input picture
    # shape = [-1, size_image_x, size_image_y, 3 channels (e.g. rgb)]
    x = tf.reshape(x, shape=[-1, 100, 100, 3])

    # Convolution Layer 1, 5x5 conv, 3 inputs, 32 outputs
    conv1 = conv_layer(x, [5, 5, 3, 32], [32] )

    # Convolution Layer 2, 5x5 conv, 32 inputs, 64 outputs
    conv2 = conv_layer(conv1, [5, 5, 32, 64], [64] )

    # Fully connected layer, 25*25*64 inputs, 1024 outputs
    # Reshape conv2 output to fit fully connected layer input
    fc1 = fully_connected_layer(conv2, [25*25*64, 1024], [1024] , dropout)

    # Output, 1024 inputs, 10 outputs (class prediction)
    output = layer(fc1 ,[1024, n_classes], [n_classes] )
    return output
```

Finally, in this function we use 3 convolutions instead of 2 like in the previous functions.

```

def inference_conv_net_3_convolutions(x, dropout):
    # Reshape input picture
    # shape = [-1, size_image_x, size_image_y, 3 channels (e.g. rgb)]
    # the img for rgb and batches of 128 would be [128, 30000] because
    # there are 128 samples per batch and images are 100x100x3 = 30000
    # this has to be re-shaped because Convolutional layers only
    # take 4 dimensional tensors as input
    # the -1 infers the number of batches and then we make the 30,000 into 100x100x3

    x = tf.reshape(x, shape=[-1, 100, 100, 3])

    # Convolution Layer 1, filter 5x5 conv, 3 inputs or 3 channels, 16 outputs
    # max pool will reduce image from 100x100 to 50x50
    conv1 = conv_layer(x, [5, 5, 3, 16], [16] )

    # Convolution Layer 2, filter 5x5 conv, 16 inputs, 36 outputs
    # max pool will reduce image from 50x50 to 25x25
    conv2 = conv_layer(conv1, [5, 5, 16, 36], [36] )

    # Convolution Layer 2, filter 5x5 conv, 36 inputs, 64 outputs
    # max pool will reduce image from 25x25 to 13x13
    conv3 = conv_layer(conv2, [5, 5, 36, 64], [64] )

    # Fully connected layer, 13*13*64 inputs, 1024 outputs
    # Reshape conv2 output to fit fully connected layer input
    # maxpool function padding=same rounds up 12.5 to 13
    fc1 = fully_connected_layer(conv3, [13*13*64, 1024], [1024] , dropout)

    # Output, 128 inputs, 10 outputs (class prediction)
    output = layer(fc1 ,[1024, n_classes], [n_classes] )
    return output

```



So, that was the hard part. Now we are ready for the loss function. It is a simple cross entropy as previously described.

```
def loss_deep_conv_net(output, y_tf):  
    xentropy = tf.nn.softmax_cross_entropy_with_logits(logits=output, labels=y_tf)  
    loss = tf.reduce_mean(xentropy)  
    return loss
```

We train with the Adam optimizer.

```
def training(cost):  
    optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)  
    train_op = optimizer.minimize(cost)  
    return train_op
```

The standard accuracy function.

```
def evaluate(output, y_tf):  
    correct_prediction = tf.equal(tf.argmax(output,1), tf.argmax(y_tf,1))  
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))  
    return accuracy
```

Now we are ready for the place holders. Notice that the image cubes are flatten from 3 dimensions to just 1. There are alternatives to this.

```
x_tf = tf.placeholder(tf.float32, [None, n_input]) ## 100x100x3
y_tf = tf.placeholder(tf.float32, [None, n_classes])
keep_prob = tf.placeholder(tf.float32) #dropout (keep probability)
```

Now we are ready to call the core functions.

```
output = inference_conv_net_3_convolution(x_tf, keep_prob)
#output = inference_conv_net2(x_tf, keep_prob)
#output = inference_conv_net(x_tf, keep_prob)

cost = loss_deep_conv_net(output, y_tf)

train_op = training(cost)
eval_op = evaluate(output, y_tf)
```

Remember this converts from one hot encoding back to a 1 column class vector.

```
## for performance metrics

y_p_metrics = tf.argmax(output, 1)
```

This should be familiar by now. We initialize the session and variables.

```
# Initialize and run

#init = tf.global_variables_initializer()

init = tf.initialize_all_variables()
sess = tf.Session()
sess.run(init)
```

Here we one hot encode the labels. Remember that there are 4 fruits so we have 4 classes.

```
# one-hot encoding

depth = 4
y_train_onehot = sess.run(tf.one_hot(y_train, depth))
y_test_onehot = sess.run(tf.one_hot(y_test, depth))
```

Next we calculate the batch processing parameters.

```
## batch parameters

num_samples_train = len(y_train)
print num_samples_train
num_batches = int(num_samples_train/batch_size)
```

We can modify the dropout parameter here.

```
dropout2 = 1.0
```

Finally, we made it to the main loop.

```
for i in range(n_epochs):
    for batch_n in range(num_batches):
        sta= batch_n*batch_size
        end= sta + batch_size

        sess.run( train_op , feed_dict={x_tf: x_train[sta:end,:],
                                         y_tf: y_train_onehot[sta:end, :], keep_prob: dropout    })

        loss, acc = sess.run([cost, eval_op], feed_dict={x_tf: x_train[sta:end,:],
                                                         y_tf: y_train_onehot[sta:end, :], keep_prob: dropout2})

        result = sess.run(eval_op, feed_dict={x_tf: x_test, y_tf: y_test_onehot,
                                              keep_prob: dropout2})
        result2, y_pred = sess.run([eval_op, y_p_metrics], feed_dict={x_tf: x_test,
                                                                      y_tf: y_test_onehot, keep_prob: dropout2})

        print "test1 {}, {}".format(i,result)
        print "test2 {}, {}".format(i,result2)

        y_true = np.argmax(y_test_onehot, 1)
        print y_pred
        print y_true
        print_stats_metrics(y_true, y_pred)
        print "*****"
```

That is it.

## 6.6 Summary

In this chapter, an implementation of a CNN model for hand written digit identification was presented and discussed. The code was provided and results of the classification task were also presented and discussed. The CNN used the MNIST data set for inputs and focused on building deep neural networks with several convolution and maxpool layers. The architecture consisted of 2 convolutional layers followed by 1 fully connected layer of size 1024.

## CHAPTER 7: RECURRENT NEURAL NETWORKS

In this section of the book I will cover the topic of Recurrent Neural Networks (RNNs). This is an important and powerful technique in deep learning that deals with sequence data mining and classification. This technique can be applied to images and text. As you may imagine, we are going to re-use a lot of the code from previous sections and chapters. The only main differences will be in defining the RNN architecture and in arranging the data for sequence classification. For the example provided in this chapter, I will use the Mnist dataset. The code discussed here can be downloaded from the course website or the github repository.

### 7.1 Using a Recurrent Neural Network (RNN) on MNIST

In our first example, I will apply RNNs to the MNIST data set. The first part of the code is to define the libraries. We will reuse most of the previously used libraries. The only new steps will be to invoke the **rnn** module and the **rnn\_cell** module from Tensorflow. These modules will help us to define the architecture.

```
import Tensorflow as tf

#from tensorflow.contrib import rnn
from tensorflow.python.ops import rnn, rnn_cell
```

Once the library modules are imported, the next step will be to set the RNN parameters. This is discussed in the next two sections.

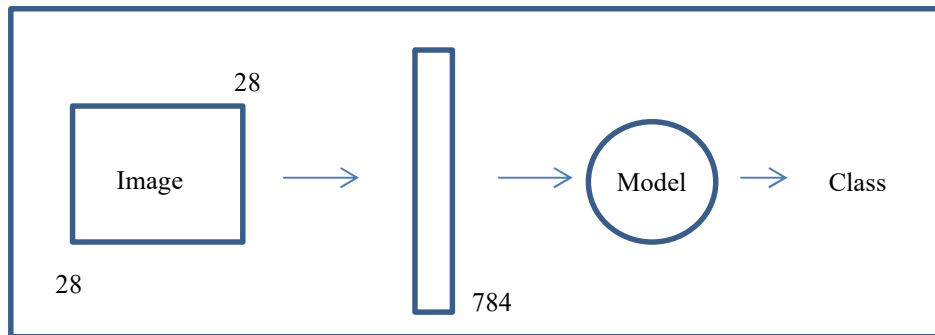
### 7.1.1 Processing Parameters

You can use this section to define any printing or output parameters. Here I use the warnings module to filter out any deprecation and other parameters.

```
## set parameters
import warnings
warnings.filterwarnings("ignore")
np.set_printoptions(threshold=np.inf)#print all values in numpy
array
```

### 7.1.2 Algorithm and Architecture Parameters

This section is very important as it helps to define the overall architecture of the Recurrent Neural Network (RNN). Here, we will use the Mnist dataset for our data. The figure below shows the non-sequential pipeline.

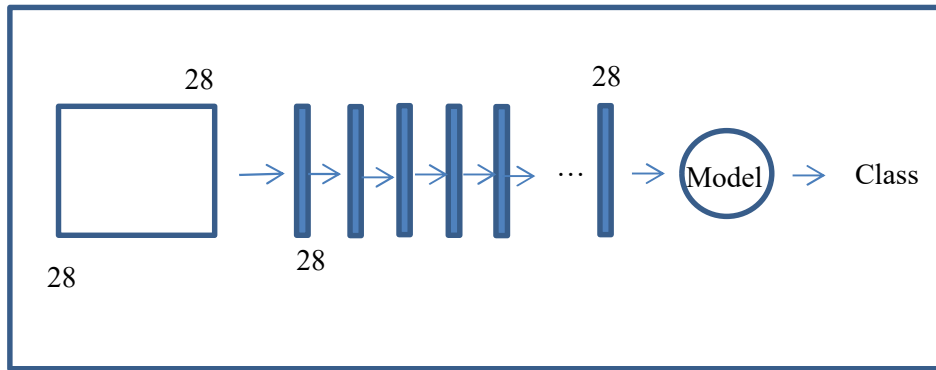


**Figure. Non-Sequential pipeline.**

In the past, we have treated each image as a sample of size  $28 \times 28$  that we want to classify as a digit. In the RNN, we still want to classify each image into one of the 10 digits. However, in the RNN we will not treat each image as a vector of 784 features ( $28 \times 28$ ). Instead, we will use a sequence modeling approach to classify the image. So, to summarize, let us compare the RNN's approach to previous approaches.

In previous approaches we looked at the whole image as an instance of 784 features and classified it that way. With the RNN we look at each image as a sequence of segments of the image or patches. We use this sequence from beginning to end to predict the final class for the image.

With RNNs the pipeline is as follows:



**Figure. Sequential pipeline.**

So, it can be seen in the previous image that the  $28 \times 28$  image is converted into 28 vectors of 28 features each (the pixels). These vectors are fed sequentially into the model. With this definition of how to represent the input data, we can proceed to define the parameters.



Some of the parameters used in the model are similar to the ones we have used in previous chapters. So, for the rest of this chapter I will only discuss new parameters that we have not seen before and that are related to the RNN.

As previously indicated, to predict the class per each 28x28 image we now think of the image as a sequence of rows. Therefore, you have 28 rows of 28 pixels each and we need to define this using some parameters. In this case, each row will be defined as a chunk and the size of each chunk will be defined as the chunk size. So we end up with a **chunk\_size** = 28, a number of chunks of **n\_chunks** = 28 (chunks per image). We still have the standard set of 10 classes. We define that as **n\_classes** = 10.

Finally, the architecture will require us to define the size of the RNN layer. We do that with **rnn\_size** = 128.

```
#parameters for the main loop
learning_rate = 0.001
n_epochs = 100000 ##27000
batch_size = 100
#to predict the class per 28x28 image, we now think of the image
#as a sequence of rows. Therefore, you have 28 rows of 28 pixels
#each
chunk_size = 28 # MNIST data input (img shape: 28*28)
n_chunks = 28 # chunks per image
rnn_size = 128 # size of rnn
n_classes = 10 # MNIST total classes (0-9 digits) ## B
```

After defining the parameters, the next step is to load the data. This is discussed in the next section.

### 7.1.3 Import the Data

As can be seen below, this step is exactly like previous steps. The Mnist dataset is loaded using `tensorflow.examples.tutorials.mnist`. Notice that the labels are still one-hot encoded.

```
# load MNIST data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)
```

In the next section we call the core functions that define the RNN deep learning model.

### 7.1.4 Calling the Core Functions

The next step in defining our RNN is to call the core functions that we have been using to build our Deep Learning models.

That is, we call:

- `Inference()`
- `Loss()`
- `Training()`
- `Evaluate()`

The main idea is still the same here and the only thing that changes is the architecture defined in the `inference()` function. In this case we will call the inference function by the name `RNN()`.

```
output = RNN(x_tf)
cost = loss_deep_rnn(output, y_tf)
train_op = training(cost)
eval_op = evaluate(output, y_tf)
```

Notice that the variables are the same ones that we have used in previous chapters such as `x_tf` for the data and `y_tf` for the labels. The 4 functions are still also referenced with the same set of variables **`output`**, **`cost`**, **`train_op`**, and **`eval_op`**.

The next 4 sections will be used to define each of the 4 core functions.

### 7.1.5 The Loss Function

Although I have changed the name of the function to **`loss_deep_rnn()`**, the loss function does not actually change from our previous implementations. It still uses the same inputs and the operation is based on

`softmax_cross_entropy_with_logits`

```
def loss_deep_rnn(output, y_tf):
    xentropy = tf.nn.softmax_cross_entropy_with_logits(output, y_tf)
    loss = tf.reduce_mean(xentropy)
    return loss
```

### 7.1.6 The Optimizer Function

In this example, the optimizer function is also similar to the one we used in previous chapters. Notice that we use the more efficient Adam optimizer.

```
def training(cost):  
    optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)  
    train_op = optimizer.minimize(cost)  
    return train_op
```

### 7.1.7 The Evaluation Function

The evaluation function is also consistent to our previous implementations. For this example, nothing really has changed.

```
def evaluate(output, y_tf):  
    correct_prediction = tf.equal(tf.argmax(output,1), tf.argmax(y_tf,1))  
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))  
    return accuracy
```

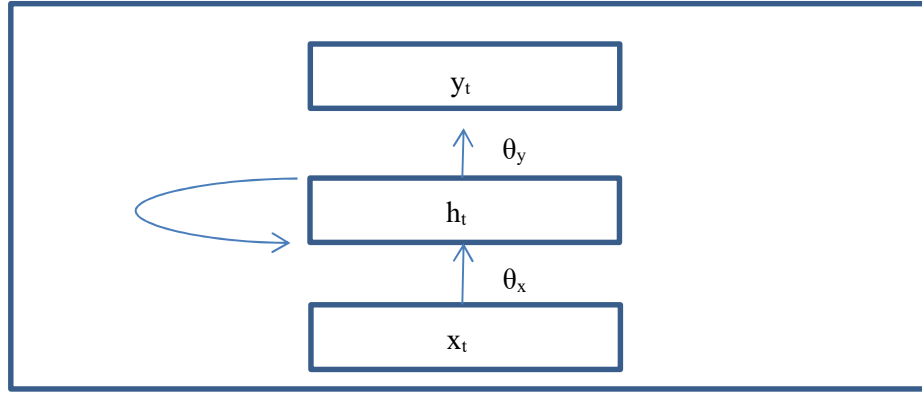
As you may have guessed by now, I saved the best for last. The next section will discuss the inference function for the RNN. This is a critical function where we define the architecture the Recurrent Neural Network (RNN).

### 7.1.8 The RNN Function and Architecture

The RNN function is the main function in which we can define the architecture of the Recurrent Neural Network. Generally speaking, an RNN can be thought of as

a regular neural network except that it now has the additional behavior of recurrence.

This behavior can be represented as follows:



**Figure. RNN model.**

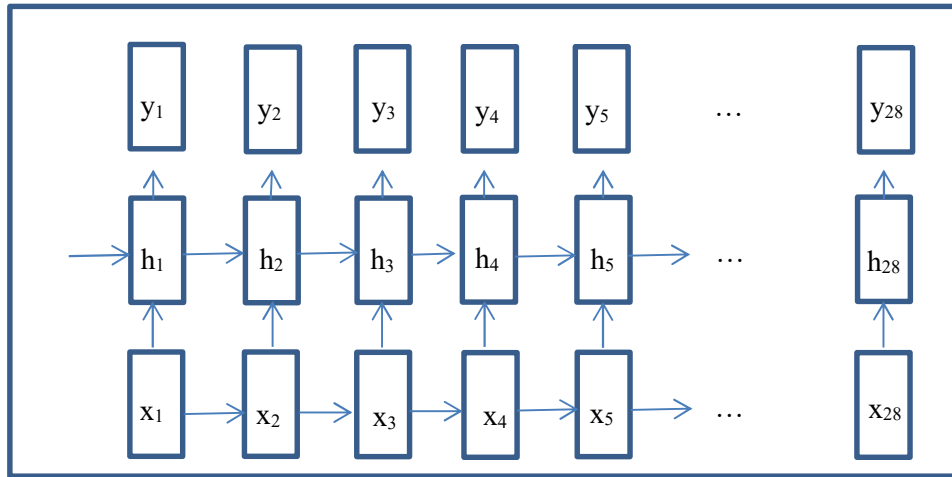
The architecture of an RNN can be expressed with the following equations.

$$y_t = \theta_y \phi(h_t)$$

where  $h_t$  can be defined as follows:

$$h_t = \theta_h \phi(h_{t-1}) + \theta_x x_t$$

In diagram form, the equations can be represented as follows



**Figure. RNN architecture.**

The following format can be used to load data into an RNN. Here, each row represents an image of 28 chunks with 28 features each.

```
X = [
    1          2          3          28
    [ [1,2,3,...,28], [1,2,3,...,28], [1,2,3,...,28], ..., [1,2,3,...,28] ]
    [ [1,2,3,...,28], [1,2,3,...,28], [1,2,3,...,28], ..., [1,2,3,...,28] ]
    [ [1,2,3,...,28], [1,2,3,...,28], [1,2,3,...,28], ..., [1,2,3,...,28] ]
    [ [1,2,3,...,28], [1,2,3,...,28], [1,2,3,...,28], ..., [1,2,3,...,28] ]
    ...
    [ [1,2,3,...,28], [1,2,3,...,28], [1,2,3,...,28], ..., [1,2,3,...,28] ]
]

X = np.array(X, dtype=np.float32)
```

The code used to implement the RNN architecture is shown below. Some aspects remain un-changed from implementations in previous chapters like the declarations of **W** and **b**. We also still perform a **matmul()** operation.

For our example, the RNN model is defined with

**rnn.rnn()**

This helps to create the RNN model which uses the standard LSTM cell or neuron (Long Short-Term Memory).

```
def RNN(x):
    W = tf.Variable( tf.random_normal( [rnn_size=128, n_classes=10] ))
    b = tf.Variable( tf.random_normal( [n_classes=10] ))

    x = tf.transpose(x, [1, 0, 2])
    x = tf.reshape(x, [-1, chunk_size=28] )
    x = tf.split(0, n_chunks=28, x)

    lstm_cell = rnn_cell.BasicLSTMCell(rnn_size=128)
    outputs, states = rnn.rnn(lstm_cell, x, dtype=tf.float32)

    rnn_output = tf.matmul(outputs[-1], W) + b
    return rnn_output
```

One of the main issues or requirements with the code in the RNN() function is that it needs to re-shape the data so that it is compatible with the requirements of the RNN architecture and model.

In the previous code segment, we can see that the data is being transformed via operations such as the transpose, re-shape, and split. Initially, the **x** variable has an initial shape of [100, 28, 28]. The transpose operation

**x = tf.transpose(x, [1, 0, 2])**

will transform the `x` shape into the following shape `[28, 100, 28]`. From here, the data `x` is reshaped using

```
x = tf.reshape(x, [-1, chunk_size=28] )
```

which results in a shape of `[2800, 28]`. Finally, the split function

```
x = tf.split(0, n_chunks=28, x)
```

will result in a shape of size `[28, 100, 28]`. This creates a list which is needed by the RNN code.

### 7.1.9 Creating the Labels and Data Tensors

In this section I will discuss our tensor placeholders. These are the same variable `x_tf` and `y_tf` that we have used to load our data. However, as you can see from the code below, the `x_tf` placeholder is a bit different. In the past we have defined `x_tf` with dimensions `(None, features)`. So, in the case of Mnist, we had `(None, 784)` because the vectors were of size 784 (`28x28`). However, as previously explained in the parameters section, we now break up each image (`28x28`) into a sequence of 28 chunks of size 28 per chunk. Therefore, we must capture this information efficiently with Tensorflow. Luckily, as its name indicates, Tensorflow can store this data into tensors of size `(None, n_chunks, chunk_size)`. This is how we will represent the data. We will need a re-shaping step to convert the images into this shape.

```
x_tf = tf.placeholder("float", [None, n_chunks, chunk_size] )  
y_tf = tf.placeholder("float", [None, n_classes])
```

All that remains for us to finish our code is to define the session, initialize the variables, and define the main loop. The next section will address the initialization of the session and the variables.



### 7.1.10 Initializing the Session and Variables

As can be seen below, defining the session and the variables involves the same steps as in previous code examples.

```
## for performance metrics
y_p_metrics = tf.argmax(output, 1)
#
# Initialize and run
init = tf.initialize_all_variables()
sess = tf.Session()
sess.run(init)
```

### 7.1.11 The Main Loop

Finally, we made it to the main loop. This is the last part of our code. Most of it is similar to our previous implementations although there are some differences. Here again, the data is read in batches. One of the aspects here is that the images of size 784 (28x28) must be re-shaped to accommodate for the RNN approach. Therefore, each **batch\_x** of vectors of 784 features each must be re-shaped into batches of sequences of 28 chunks with 28 elements each. We achieve this with the statement

```
batch_x = batch_x.reshape( (batch_size, n_chunks, chunk_size) )
```

Once the batch is re-shaped, it can be used by the training step just like in previous examples. In the testing phase of the main loop, 256 test samples are selected to evaluate the accuracy of the algorithm. The test data samples must also be re-shaped. We do this with the statement

```
test_data = mnist.test.images[:test_len].reshape((-1, n_chunks, chunk_size))
```

in this case, the re-shape statement

```
.reshape( (-1, n_chunks, chunk_size) )
```

converts the test samples into sequences of **n\_chunks** of size **chunk\_size** each.

```
step = 1
while step * batch_size < n_epochs:
    batch_x, batch_y = mnist.train.next_batch(batch_size)
    # batch_x has a vector of 784 which needs to be converted
    # to a sequence
    # Reshape data to get 28 chunks of 28 elements each
    batch_x = batch_x.reshape( (batch_size, n_chunks, chunk_size) )
    sess.run(train_op, feed_dict={x_tf: batch_x, y_tf: batch_y})
    loss, acc = sess.run([cost, eval_op], feed_dict={x_tf: batch_x,
                                                    y_tf: batch_y })

    test_len = 256
    test_data = mnist.test.images[:test_len].reshape((-1,
                                                    n_chunks, chunk_size))

    test_label = mnist.test.labels[:test_len]
    result_eval = sess.run( eval_op, feed_dict={x_tf: test_data,
                                                    y_tf: test_label} )

    print "result {},{}".format(step, result_eval)

    step = step + 1
```

The **.reshape()** function is used to reshape a tensor. In this case, the function takes 3 parameters to define the new dimensions of the tensor. For instance, the statement

```
batch_x = batch_x.reshape( (batch_size, n_chunks, chunk_size) )
```

is actually equivalent to the following statement.

```
batch_x = batch_x.reshape( (100, 28, 28) )
```

Before the **.reshape()** command, the tensor has a shape of [100, 784]. After reshaping, the tensor has dimensions of [100, 28, 28]. Similarly, the statement

```
test_data=mnist.test.images[:test_len].reshape((-1,n_chunks,chunk_size))
```

is equivalent to the statement

```
test_data=mnist.test.images[:test_len].reshape((-1,28,28))
```

Before the **.reshape()** command, the tensor has a shape of [256, 784]. After reshaping the tensor has dimensions of [256, 28, 28].

A good way to understand how the **.reshape()** function works is to print out the shape of the tensor before and after the reshape function. The following code segment can help to visualize these dimensions.

```
print "before"
the_shape = tf.shape(batch_x)
print sess.run(the_shape)
print "after"
x = raw_input()
```

### 7.1.12 Results

The previously described code provided an example of how to classify Mnist images using RNNs. The code only evaluated accuracy for this example although you can evaluate for the other metrics as well. I leave that as an exercise to the reader. For comparison, I ran this code as written here and obtained classification accuracies as high as or higher than 97%-99%.

than 97%-99%.

## 7.2 RNNs for NLP

In this section we will begin our discussion of RNNs and how they can be applied to natural language processing (NLP) problems. Before I discuss the code, I should say a few things about RNNs and related considerations. As discussed in the previous section, RNNs are sequential methods. That means that the model learns from the outputs from previous steps/layers. That also means that you may end up with very deep networks. Weights at time “t” are dependent from weights at times as far back as

$$t - n$$

where “n” could be really large. As weights are multiplied they could tend towards 0 or to really large numbers. Both scenarios are bad for deep neural networks. Weights that are 0 basically mean that the network learned nothing, for instance. This is known as the “vanishing gradients” problem. This problem has been addressed by using a special type of neuron called the LSTM or Long Short Term Memory (further discussed in the next section). Basically, LSTM neurons address the vanishing gradients problem by not multiplying the previous weights from the previous hidden layer. But instead just by passing them without modification to the next layer. In this code example we will use LSTM neurons.

Now, let us discuss our example. Here, we will train a model to learn from text data. The model will then attempt to create a story using the learned RNN model. The data is any text. The text is read as a single long string and then pre-processed.

First, we read the libraries as follows.

```
import tensorflow as tf
import numpy as np
from numpy import genfromtxt
from sklearn import datasets
#from sklearn.cross_validation import train_test_split
from sklearn.model_selection import train_test_split
import sklearn

from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score, f1_score
import pandas as pd
import matplotlib.pyplot as plt

import random
import collections
import time
```

Notice the important new library is called rnn. There are also other options.

```
from tensorflow.contrib import rnn

#from tensorflow.python.ops import rnn, rnn_cell
```

Next, we set processing parameters.

```
## set parameters

import warnings
warnings.filterwarnings("ignore")

np.set_printoptions(threshold=np.inf) #print all values in numpy array
```

We initialize the algorithm parameters.

```
learning_rate = 0.001
n_epochs = 50000 #100000 ##27000
#batch_size = 100
display_step = 1000
```

This section is very important. Here we need to define the vocabulary size which is the number of words that we will use. This is similar to what is done in Word2vec. The variable is initialized to zero but will be updated once the data is read. We also define the size of the hidden layer in the RNN cell. In this case we use 512.

```
vocab_size = 0 ## updated once data is read

n_input = 3
# number of units in RNN cell
n_hidden = 512
```

Next, we specify the name of the file.

```
training_file = 'input_file.txt'
```

The text looks like this:

“In the time of swords and periwigs and full-skirted coats with flowered lappets-- when gentlemen wore ruffles, and gold-laced waistcoats of paduasoy and taffeta-- there lived a tailor in Gloucester.”

Short extract from the THE TAILOR OF GLOUCESTER by Beatrix Potter.

Now we define the function to read our text from file. This returns a list of words.

```
def read_data(fname):
    with open(fname) as f:
        content = f.readlines()
    content = [x.strip() for x in content]
    content = [word for i in range(len(content)) for word in content[i].split()]
    content = np.array(content)
    return content
```

We create the list.

```
training_data = read_data(training_file)
```

Now we must build the data sets to use. This function calculates the frequency of the words. Then it creates a dictionary of words where the key is the word and frequency position is the value. The dictionary is also reversed so that the key is the frequency position and the value is the text word. This helps with lookups.

```
def build_dataset(words):
    count = collections.Counter(words).most_common()
    dictionary = dict()
    for word, _ in count:
        dictionary[word] = len(dictionary)
    reverse_dictionary = dict(zip(dictionary.values(), dictionary.keys()))
    return dictionary, reverse_dictionary
```

Now we call the function and it returns both versions of the dictionary. Here we can also calculate the vocabulary size.

```
dictionary, reverse_dictionary = build_dataset(training_data)
#print(reverse_dictionary)
vocab_size = len(dictionary)
```

Now we can define the metrics function.

```
precision_scores_list = []
accuracy_scores_list = []

def print_stats_metrics(y_test, y_pred):
    print('Accuracy: %.2f % accuracy_score(y_test, y_pred) )
    accuracy_scores_list.append(accuracy_score(y_test, y_pred) )
    confmat = confusion_matrix(y_true=y_test, y_pred=y_pred)
    print("confusion matrix")
    print(confmat)
    print(pd.crosstab(y_test, y_pred, rownames=['True'], colnames=['Predicted'], margins=True))
    precision_scores_list.append(precision_score(y_true=y_test,
                                                  y_pred=y_pred, average='weighted'))
    print('Precision: %.3f % precision_score(y_true=y_test, y_pred=y_pred, average='weighted'))
    print('Recall: %.3f % recall_score(y_true=y_test, y_pred=y_pred, average='weighted'))
    print('F1-measure: %.3f % f1_score(y_true=y_test, y_pred=y_pred, average='weighted'))
```

Now we are ready to define the inference function. A lot happens in this function. We define the weights matrix and the bias. We reshape the input to fit into the RNN. We define the LSTM cell. And we define the function for predicting outputs.

In our example we use

```
rnn_cell = rnn.BasicLSTMCell(n_hidden) ## 1 layer LSTM
```



but we could have used

```
#rnn_cell = rnn.MultiRNNCell([ rnn.BasicLSTMCell(n_hidden),  
                               rnn.BasicLSTMCell(n_hidden)]) ## 2 layer LSTM
```

```
def RNN_nlp(x):  
    W = tf.Variable( tf.random_normal( [n_hidden, vocab_size] ))  
    b = tf.Variable( tf.random_normal( [vocab_size] ))  
  
    # reshape to [1, n_input], only 1 per batch  
    x = tf.reshape(x, [-1, n_input])  
  
    # Generate a n_input-element sequence of inputs  
    # (eg. [had] [a] [general] -> [20] [6] [33])  
    x = tf.split(x, n_input, 1)  
  
    rnn_cell = rnn.BasicLSTMCell(n_hidden) ## 1 layer LSTM  
    #rnn_cell = rnn.MultiRNNCell([ rnn.BasicLSTMCell(n_hidden),  
                                   rnn.BasicLSTMCell(n_hidden)]) ## 2 layer LSTM  
  
    # generate prediction  
    outputs, states = rnn.static_rnn(rnn_cell, x, dtype=tf.float32)  
  
    # there are n_input outputs but  
    # we only want the last output  
  
    rnn_output = tf.matmul(outputs[-1], W) + b  
    return rnn_output
```

Now we can define the loss function which once again uses cross entropy.

```
def loss_deep_rnn(output, y_tf):  
    xentropy = tf.nn.softmax_cross_entropy_with_logits(logits=output, labels=y_tf)  
    loss = tf.reduce_mean(xentropy)  
    return loss
```

Now we use the RMSProp optimizer to train the model.

```
def training(cost):
    #optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
    optimizer = tf.train.RMSPropOptimizer(learning_rate=learning_rate)
    train_op = optimizer.minimize(cost)
    return train_op
```

This is our standard accuracy function.

```
def evaluate(output, y_tf):
    correct_prediction = tf.equal(tf.argmax(output,1), tf.argmax(y_tf,1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
    return accuracy
```

After defining our core functions we can define the placeholders. The value `n_input` is of size 3. We generate a `n_input` element sequence. This gives us:

(eg. [the] [cat] [is] -> [2] [16] [7])

```
x_tf = tf.placeholder("float", [None, n_input, 1]) ## the 1 could be vocab_size
y_tf = tf.placeholder("float", [None, vocab_size])
```

Now we can call the core functions

```
output = RNN_nlp(x_tf)
cost = loss_deep_rnn(output, y_tf)
train_op = training(cost)
eval_op = evaluate(output, y_tf)
```

The following line of code helps to convert from one hot back to non one hot encoding. The result is the predicted word id.

```
## predicted word id  
output_no_onehot = tf.argmax(output, 1)
```

Now we are ready to initialize the variables.

```
init = tf.initialize_all_variables()  
sess = tf.Session()  
sess.run(init)
```

In this section we define the step variable.

```
step = 0
```

This section of code allows us to specify the index tracking variable. We use this to track our position in the story. We use offset to slide a window accross the document.

Every iteration offset is incremented. So it is the index that tracks the current position in the document.

```
## use offset to slide the window across the document
## every iteration offset is incremented - it is the index that
## tracks the current position in the document

offset = random.randint(0, n_input+1) ## pick current index on story
end_offset = n_input + 1
```

Finally, we arrive at the main loop.

```
while step < n_epochs:
    if offset > ( len(training_data) - end_offset ): ## training_data is all the words in the story
        offset = random.randint(0, n_input + 1) ## reset to beginning of document

    symbols_in_keys = [ [dictionary[ str(training_data[i])]] for i in range(offset, offset+n_input) ]
    symbols_in_keys = np.reshape( np.array(symbols_in_keys), [-1, n_input, 1] )

    symbols_out_onehot = np.zeros( [vocab_size], dtype=float )
    symbols_out_onehot[dictionary[ str( training_data[ offset+n_input ] )]] = 1.0
    ## e.g. vector[37] = 1.0
    symbols_out_onehot = np.reshape( symbols_out_onehot, [1,-1] )

    _, acc, loss, pred_onehot, pred_word = sess.run([train_op, eval_op, cost,
                                                    output, output_no_onehot],
                                                    feed_dict={x_tf: symbols_in_keys, y_tf: symbols_out_onehot})

    test_sequence = [training_data[i] for i in range(offset, offset + n_input)]
    test_actual_after_sequence = training_data[offset + n_input]
    ## the actual word after a sequence of 3

    test_pred_after_sequence = reverse_dictionary[int(pred_word)]
    print("%s ... %s - [%s] vs [%s]" % (step, test_sequence, test_actual_after_sequence,
                                       test_pred_after_sequence) )

    step = step + 1
    offset = offset + (n_input+1)
```

After training, we can proceed to test our model by generating a story. This is after training.

This generates a short story using the trained **rnn**.

```
## this is after training - this generates a short story using
## the trained rnn

while True:
    try:
        prompt = "%s words: " % n_input
        print(prompt)
        sentence = raw_input()
        print(sentence)
        sentence = sentence.replace("\n", "")
        sentence = sentence.strip()
        words = sentence.split(' ')
        print(words)
        try:
            symbols_in_keys = [dictionary[str(words[i])] for i in range(len(words))]
            for i in range(64):
                keys = np.reshape(np.array(symbols_in_keys), [-1, n_input, 1])
                predicted_word_int = sess.run(output_no_onehot, feed_dict={x_tf: keys})
                sentence = "%s %s" % ( sentence, reverse_dictionary[ int(predicted_word_int) ] )
                symbols_in_keys = symbols_in_keys[1:]
                symbols_in_keys.append(predicted_word_int)
            print(sentence)
        except:
            print("Words not in dictionary")
    except:
        print("error with your input")
```

That is it. Now let us discuss LSTMs and Transformers.

## 7.3 LSTM

In this section I will discuss the LSTM (Long Short Term Memory) neuron. Basically, LSTM neurons address the vanishing gradients problem by not multiplying the previous weights from the previous hidden layer. But instead just by passing them without modification to the next layer.

## **7.4 Transformers**

In this section I will address Transformers. Transformers have become popular as of 2019. Currently considered to have certain benefits over LSTM neurons. One of the problems with LSTM neurons is that they do not parallelize very well and therefore result in slow and somewhat inefficient learning. Transformers address this issue by reading sequential data more efficiently and are better suited for parallel approaches. Therefore, they can be faster. They also have better architectures for processing the data. Transformers are so important that I have dedicated a whole chapter for them.

## **7.5 Summary**

In this chapter Recurrent Neural Networks (RNNs) were presented and discussed. An example using the Mnist hand written digits data set was used for the analysis. Issues related to data representation and RNN architecture were also discussed. Finally, the application of RNNs to NLP was presented.

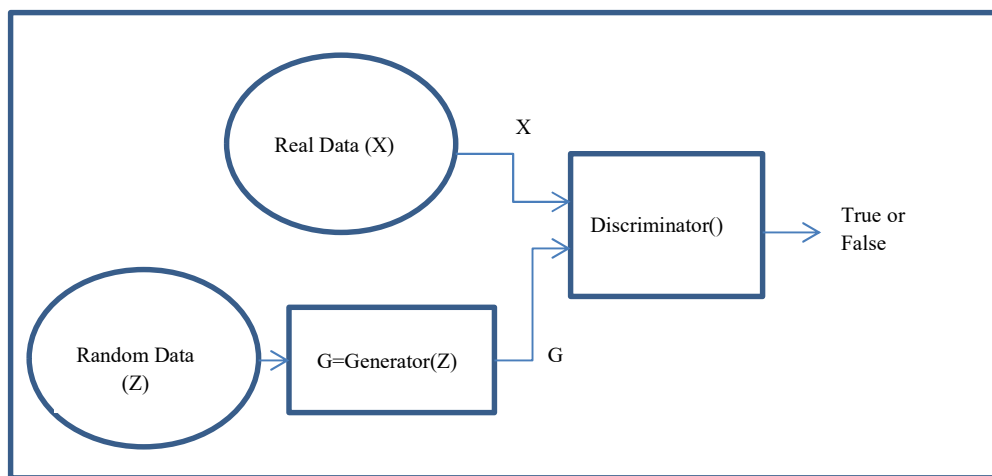
## CHAPTER 8: GENERATIVE ADVERSARIAL NETWORKS

In this section of the book I will cover Generative Adversarial Networks (GANs). Generative Adversarial Networks (GANs) (Goodfellow et al. 2014) are a first attempt at representing unsupervised problems in the context of games (e.g. GANs are modeled as a two player adversarial game). One of the biggest challenges we face with supervised learning is annotating the data. We cannot annotate automatically and without annotations we cannot train our learning models. But what if we could substitute the annotation of the data for something else? For instance, what if we could model the annotation task as a game or use other previous knowledge about the world as labels. These ideas are one of the main motivations for GANs.

GANs are deep neural networks that consist of a generator network connected to a discriminator network. The discriminator network has training data and the generator network only has random or noise data as input. GANs are essentially 2 player games where one player (the generator) creates synthetic data samples, while the second player (the discriminator) takes the generated sample and performs a classification.

This classification is performed to determine if the synthetic sample is similar to the distribution of the discriminator's training data. Since both networks are connected, the deep neural network (GAN) can learn to generate better synthetic samples with the help of the discriminator's output. Basically, the discriminator tells the generator how to adjust its weights to produce better synthetic samples.

Generative Adversarial Networks are methods that use 2 deep neural networks to interact with each other and generate data. Its formulation is consistent with 2 player adversarial game frameworks. One of the 2 algorithms (or networks) tries to learn a data distribution and produce new samples similar to the samples in the real data (the generator). The second algorithm (the adversary) is a classifier that tries to determine if the new samples generated by the generative algorithm are fake or real. These 2 algorithms work together to achieve an optimal outcome of producing better output samples.



**Figure. A GAN network using MNIST.**

The code to implement a GAN network is presented below.



## 8.1 GAN code

In this section, the GAN code will be described. First we list the libraries that we need.

```
import tensorflow as tf
import numpy as np
from numpy import genfromtxt
```

Next we initialize our variables

```
batch_size = 8
hidden_size = 4

num_steps = 5000
display_step = 10

seed = 42
tf.set_random_seed(seed)
```

We will need some helper functions such as a log estimation function.

```
def log(x):
    return tf.log(tf.maximum(x, 1e-5))
```

To keep things simple, instead of reading in data, we are going to generate it automatically. We will generate samples with a normal distribution of **mean**=4 and **sigma** = 0.5. **N** is the number of samples to generate.

These samples are for the discriminator and represent the real data.

```
class DataDistribution(object):
    def __init__(self):
        self.mu = 4
        self.sigma = 0.5

    def sample(self, N):
        samples = np.random.normal(self.mu, self.sigma, N)
        samples.sort()
        return samples
```

The data from the distribution class looks like this:

```
array([ 3.3777126 ,  3.46725909,  3.65951541,  3.81755036,  3.81998276,
        3.92007   ,  4.28720089,  4.51158124])
```

We also provide a set of data for the generator. Think of this as noise. We set a range between [-range, range] but the values are random. The generator uses noise as input.

```
class GeneratorDistribution(object):
    def __init__(self, range):
        self.range = range

    def sample(self, N):
        return np.linspace(
            -self.range, self.range, N) + np.random.random(N) * 0.01
```

The data from the generator class looks like this:

```
>>> generatorData  
array([-7.99054428, -6.2211434 , -4.43866942, -2.65729133, -0.88879437,  
       0.89649839, 2.67079517, 4.45113515, 6.22879001, 8.00656172])
```

First we define the layer function for the GAN.

```
def layer_GAN(input, weight_shape, bias_shape):  
    w_init = tf.random_normal_initializer(stddev=1.0)  
    bias_init = tf.constant_initializer(0.0)  
    W = tf.get_variable("w", weight_shape, initializer=w_init )  
    b = tf.get_variable("b", bias_shape, initializer=bias_init )  
    return tf.matmul(input, W) + b
```

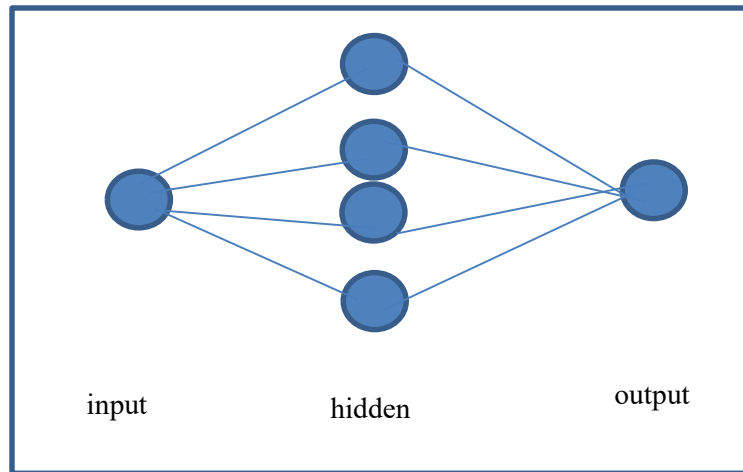
With GANs we have 2 inference functions; one for the generator and one for the discriminator. The inference function for the generator is:

```
def inference_generator(input):  
    h1=tf.nn.softplus(layer_GAN(input,[input.get_shape()[1], 4], [4]))  
    h2 = layer_GAN(h1, [ h1.get_shape()[1], 1] , [1])  
    return h2
```

The inference function for the generator with values is:

```
def inference_generator(input):  
    h1=tf.nn.softplus(layer_GAN(input,[1, 4], [4]))  
    h2 = layer_GAN(h1, [ 4, 1] , [1])  
    return h2
```

The neural network in `inference_generator()` looks like the following:



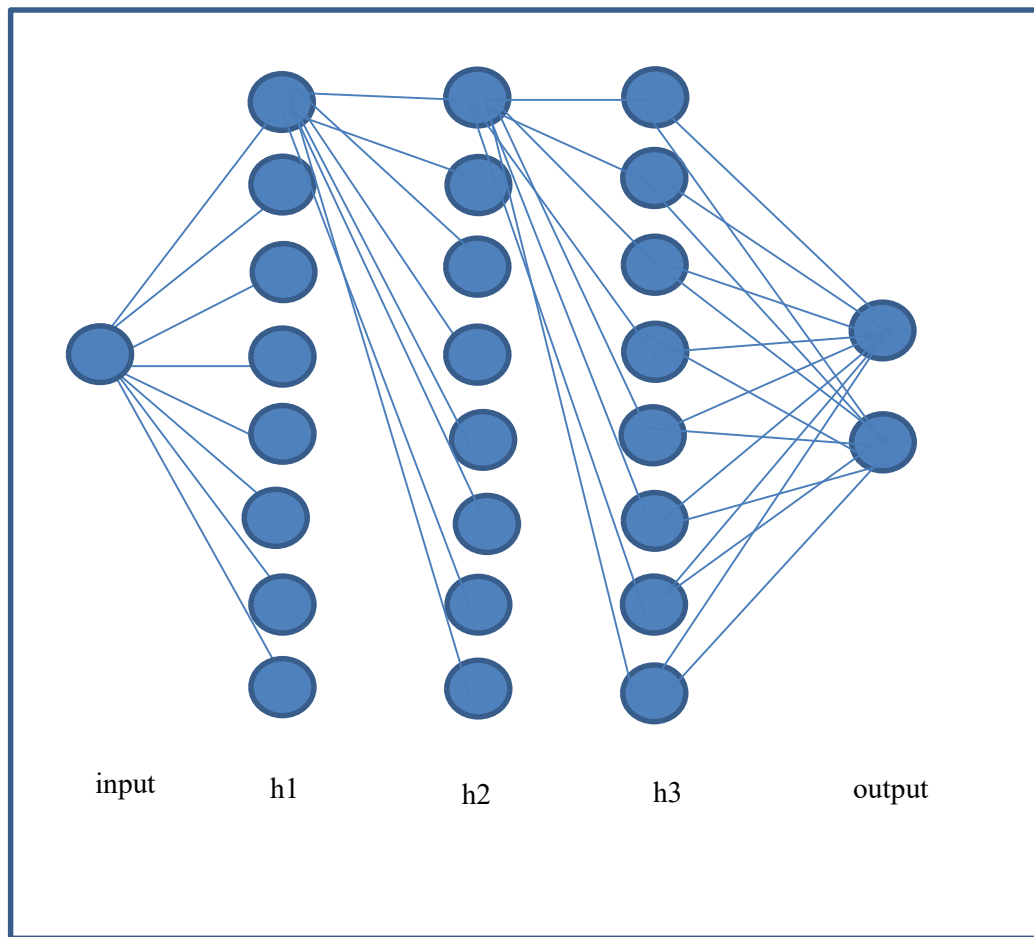
**Figure. Neural network in `inference_generator()`.**

And the inference function for the discriminator is:

```
def inference_discriminator(input):  
    h1 = tf.nn.relu(layer_GAN(input, [ 1, 8 ],[8] ))  
    h2 = tf.nn.relu(layer_GAN(h1, [8, 8] , [8]))  
    h3 = tf.nn.relu(layer_GAN(h2, [8, 8], [8]) )  
    h4 = tf.sigmoid(layer_GAN(h3, [ 8, 2 ] , [2]))  
    return h4
```

Notice that the discriminator has more capacity to learn.

The neural network in **inference\_discriminator()** looks like the following:



**Figure.** Deep neural network in **inference\_discriminator()**

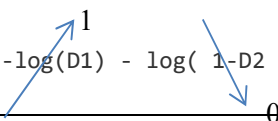
The GAN has 2 loss functions to calculate max entropy; one for the discriminator and one for the generator.

The loss for the discriminator is as follows:

```
def loss_d_GAN(D1, D2):  
    loss_d = tf.reduce_mean( -log(D1) - log( 1-D2 ) )  
    return loss_d
```

Think of the 2 parameters in loss\_d\_GAN as tending to 1 and 0 like so

```
def loss_d_GAN(D1, D2):  
    loss_d = tf.reduce_mean( -log(D1) - log( 1-D2 ) )  
    return loss_d
```



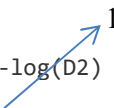
A diagram with two blue arrows. One arrow starts from the 'D1' parameter in the code and points diagonally upwards and to the right towards the number '1'. The other arrow starts from the '1-D2' expression in the code and points diagonally downwards and to the right towards the number '0'.

And the loss for the generator is:

```
def loss_g_GAN(D2):  
    loss_g = tf.reduce_mean( -log(D2) )  
    return loss_g
```

Think of parameter in loss\_g\_GAN as tending to 1 like so

```
def loss_g_GAN(D2):  
    loss_g = tf.reduce_mean( -log(D2) )  
    return loss_g
```



A diagram with a blue arrow starting from the 'D2' parameter in the code and pointing diagonally upwards and to the right towards the number '1'.

Next we can define the optimization function as follows:

```
def training_GAN(cost):  
    step = tf.Variable(0)  
    optimizer = tf.train.AdamOptimizer(0.001)  
    train_op=optimizer.minimize(cost)  
    return train_op
```

Once the core functions are defined, we can proceed to define our placeholders. We select a batch size of 8 so that **x** and **z** are data matrices of size [8, 1]. So they contain 8 samples with 1 feature each.

```
## batch size = 8
## so x and z are data matrices of size 8 x 1
## 8 samples with 1 feature each

x = tf.placeholder(tf.float32, shape=(batch_size , 1))
z = tf.placeholder(tf.float32, shape=(batch_size , 1))
```

Now we proceed to call the core functions as shown below:

```
with tf.variable_scope('G'):
    output_G = inference_generator(z)
with tf.variable_scope('D'):
    output_D1 = inference_discriminator(x)
with tf.variable_scope('D'):
    output_D2 = inference_discriminator(output_G )

#####

cost_d = loss_d_GAN(output_D1,output_D2)
cost_g = loss_g_GAN(output_D2)

#####

train_op_d = training_GAN(cost_d)
train_op_g = training_GAN(cost_g)
```

We are almost done. All that remains is to initialize the variables and create the session.

```
init = tf.initialize_all_variables()
sess = tf.Session()
sess.run(init)
```

Before we call the main loop we need to have some training data. In this case we call the data generating classes we previously defined and create some data.

```
data=DataDistribution()  
gen=GeneratorDistribution(range=8)
```

Finally, we can call the main loop as follows:

```
for step in range(num_steps):  
    x_data = data.sample(batch_size)  
    z_data = gen.sample(batch_size)  
  
    ## reshape from row to column vector  
    x_resaped = np.reshape(x_data, (batch_size, 1))  
    z_resaped = np.reshape(z_data, (batch_size, 1))  
    res_cost_d, res_train_d = sess.run(  
        [cost_d, train_op_d], feed_dict={x: x_resaped, z: z_resaped})  
  
    #update new data for generator  
    z_data = gen.sample(batch_size)  
    z_resaped = np.reshape( z_data, (batch_size, 1) )  
    res_cost_g, res_train_g = sess.run(  
        [cost_g, train_op_g], feed_dict={ z: z_resaped })  
  
    if step % display_step == 0:  
        print('{}: cost_d: {:.4f}\t cost_g: {:.4f}'.format(  
            step, res_cost_d, res_cost_g))  
        print('{}: train_d {} \t train_g: {}'.format(  
            step, res_train_d, res_train_g))
```

And that is it! We have completed our GAN model.

## 8.2 Generating MNIST digits with GANs

In this section, I will describe how to implement a GAN that can generate images. The algorithm will work with the tensorflow provided MNIST data set and with



your own image data which you can supply in a folder. As always, the code can be downloaded from my GitHub.

First we import the libraries.

```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
import os
from PIL import Image

import glob
```

The following function helps us to initialize our variable for the weight matrices.

```
def xavier_init(size):
    in_dim = size[0]
    xavier_stddev = 1. / tf.sqrt(in_dim / 2.)
    return tf.random_normal(shape=size, stddev=xavier_stddev)
```

In this code segment we generate the discriminator placeholders and variables. Notice that the architecture for this network is 784x128x1. Its input is an image vector (real or fake) and its output is one neuron with value 0 or 1.

```
X = tf.placeholder(tf.float32, shape=[None, 784])

D_W1 = tf.Variable(xavier_init([784, 128]))
D_b1 = tf.Variable(tf.zeros(shape=[128]))

D_W2 = tf.Variable(xavier_init([128, 1]))
D_b2 = tf.Variable(tf.zeros(shape=[1]))

theta_D = [D_W1, D_W2, D_b1, D_b2]
```

In the following code segment we define the placeholder and variables for the Generator. Notice that this will be a neural network of size 100x128x784 given that it wishes to generate MNIST images.

```
Z = tf.placeholder(tf.float32, shape=[None, 100])

G_W1 = tf.Variable(xavier_init([100, 128]))
G_b1 = tf.Variable(tf.zeros(shape=[128]))

G_W2 = tf.Variable(xavier_init([128, 784]))
G_b2 = tf.Variable(tf.zeros(shape=[784]))

theta_G = [G_W1, G_W2, G_b1, G_b2]
```

The following function can be used to generate seed vectors for the Generator.

```
def sample_Z(m, n):  
    return np.random.uniform(-1., 1., size=[m, n])
```

This is the function for the generator.

```
def generator(z):  
    G_h1 = tf.nn.relu(tf.matmul(z, G_W1) + G_b1)  
    G_log_prob = tf.matmul(G_h1, G_W2) + G_b2  
    G_prob = tf.nn.sigmoid(G_log_prob)  
  
    return G_prob
```

Here is the function with the architecture for the discriminator. As can be seen, it is a simple neural network with 1 hidden layer.

```
def discriminator(x):  
    D_h1 = tf.nn.relu(tf.matmul(x, D_W1) + D_b1)  
    D_logit = tf.matmul(D_h1, D_W2) + D_b2  
    D_prob = tf.nn.sigmoid(D_logit)  
  
    return D_prob, D_logit
```

We can use this handy function to print out the images generated by the Generator.

```

def plot(samples):
    fig = plt.figure(figsize=(4, 4))
    gs = gridspec.GridSpec(4, 4)
    gs.update(wspace=0.05, hspace=0.05)

    for i, sample in enumerate(samples):
        ax = plt.subplot(gs[i])
        plt.axis('off')
        ax.set_xticklabels([])
        ax.set_yticklabels([])
        ax.set_aspect('equal')
        plt.imshow(sample.reshape(28, 28), cmap='Greys_r')

    return fig

```

Here we use the Generator which produces a set of fake images ( $G\_sample$ ). Then we run the discriminator twice. Once on the real data ( $X$ ) and once on the fake data ( $G\_sample$ ).

```

G_sample = generator(Z)
D_real, D_logit_real = discriminator(X)
D_fake, D_logit_fake = discriminator(G_sample)

```

We can define the losses like this using the cross entropy function

```

# D_loss = -tf.reduce_mean(tf.log(D_real) + tf.log(1. - D_fake))
# G_loss = -tf.reduce_mean(tf.log(D_fake))

```

Or like this with the tensorflow built in cross entropy functions. The cross entropy functions are even more intuitive. **D\_logit\_real** is for real images so the discriminator wants to classify it as ones. For **D\_logit\_fake**, the discriminator goals is to classify it as zero.

```
D_loss_real=tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(
    logits=D_logit_real,
    labels=tf.ones_like(D_logit_real)  ))

D_loss_fake=tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(
    logits=D_logit_fake,
    labels=tf.zeros_like(D_logit_fake)  ))

D_loss = D_loss_real + D_loss_fake
```

The generator loss is similar

```
G_loss = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(
    logits=D_logit_fake,
    labels=tf.ones_like(D_logit_fake)  ))
```

If you notice, this approach is even more intuitive because you can see the goal of each loss. The goal of **G\_loss** is to fool the discriminator. As such, its goal is to make **D\_logit\_fake** equal to ones.

Here we define the optimization for our two losses **D\_loss** for the Discriminator and **G\_loss** for the Generator.

```
D_solver = tf.train.AdamOptimizer().minimize(D_loss, var_list=theta_D)
G_solver = tf.train.AdamOptimizer().minimize(G_loss, var_list=theta_G)
```

We specify a few parameters such as the batch size for training which is 128 and the size of the seed random vector that you feed to the generator. In this case 100.

```
mb_size = 128
Z_dim = 100
```

You can read the images from tensorflow examples like this

```
mnist = input_data.read_data_sets('../MNIST_data', one_hot=True)
```

Or you can read your own images like in the code below. Here we read the images from a folder testA using PIL and Image. Notice that I have used images that are 28x28 for convenience.

```

## this will create your own data set (i.e. your_mnist)
## put your images in testA (the example assumes png images)

train = []

files = glob.glob ("testA/*.png") # your image path

for myFile in files:
    image = np.array(Image.open(myFile).convert('LA'))
    ## size is (28,28,2) and now (28,28,1) - removes png transparency
    image = image[...,:1]
    new_image = image.reshape(image.shape[1]*image.shape[0])
    train.append(new_image)

```

The function `convert('LA')` changes color images to gray scale. The result image will be of size (28, 28, 2) because .png files have a transparency. You will need to convert it to (28, 28, 1) using **`image[...,:1]`**. Now make data set float and numpy array. You can print the dimensions to confirm your data is correct.

```

train = np.array(train,dtype='float32') #as mnist

print(train.shape)
print(train.shape[0])
print(train.shape[1])
#input("train size is")

```

Now you can save the numpy array

```
# save numpy array as .npy formats  
np.save('train',train)
```

I also renamed the data set

```
your_mnist = train
```

You can view image dimensions with the following functions:

```
#print(image.shape[1])  
#print(image.shape[0])  
#print(new_image.shape)  
#input("??")
```

In the case of color images you could convert using the following:

```
# convert (number of images x height x width x number of channels)  
# to (number of images x (height * width * 3))  
# for example (120 * 40 * 40 * 3)-> (120 * 4800)  
#train=np.reshape(train,[train.shape[0],train.shape[1]*train.shape[2]*train.shape[3]])  
#train = np.reshape(train,[train.shape[0],train.shape[1]*train.shape[2]])
```



It is recommended that you normalize your data to improve the learning process of the GAN.

```
## normalization
x=your_mnist
xmax, xmin = x.max(), x.min()
x = (x - xmin)/(xmax - xmin)

your_mnist = x
```

In the next code segment we initialize the session and create the output folder where we will save the new images generated by the Generator network.

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

if not os.path.exists('out/'):
    os.makedirs('out/')
```

The main loop here is implemented with the Tensorflow provided data

```

i = 0
for it in range(1000000):
    if it % 1000 == 0:
        samples = sess.run(G_sample, feed_dict={Z: sample_Z(16, Z_dim)})

        fig = plot(samples)
        plt.savefig('out/{}.png'.format(str(i).zfill(3)), bbox_inches='tight')
        i += 1
        plt.close(fig)

    X_mb, _ = mnist.train.next_batch(mb_size)

    _, D_loss_curr = sess.run([D_solver, D_loss],
                              feed_dict={X: X_mb, Z: sample_Z(mb_size, Z_dim)})
    _, G_loss_curr = sess.run([G_solver, G_loss],
                              feed_dict={Z: sample_Z(mb_size, Z_dim)})

    if it % 1000 == 0:
        print('Iter: {}'.format(it))
        print('D loss: {:.4}'.format(D_loss_curr))
        print('G_loss: {:.4}'.format(G_loss_curr))

    print()

```

Notice the high number of iterations.

Now the loop with your own data

```
i = 0
i2 = 0
for it in range(1000000):
    if it % 1000 == 0:
        samples = sess.run(G_sample, feed_dict={Z: sample_Z(16, Z_dim)})

        fig = plot(samples)
        plt.savefig('out/{}.png'.format(str(i).zfill(3)), bbox_inches='tight')
        i += 1
        plt.close(fig)

    index = i2*mb_size
    X_mb = your_mnist[index:index+mb_size,:]
    #print(X_mb.shape)
    i2 = i2 + 1
    if index >= your_mnist.shape[0]:
        i2 = 0

    _, D_loss_curr = sess.run([D_solver, D_loss],
                              feed_dict={X: X_mb, Z: sample_Z(mb_size, Z_dim)})
    _, G_loss_curr = sess.run([G_solver, G_loss],
                              feed_dict={Z: sample_Z(mb_size, Z_dim)})

    if it % 1000 == 0:
        print('Iter: {}'.format(it))
        print('D loss: {:.4}'.format(D_loss_curr))
        print('G_loss: {:.4}'.format(G_loss_curr))

        print()
```

In this code segment, every 1000 iterations we generate a set of fake images using the generator to see how it is doing.

```
if it % 1000 == 0:
    samples = sess.run(G_sample, feed_dict={Z: sample_Z(16, Z_dim)})

    fig = plot(samples)
    plt.savefig('out/{}.png'.format(str(i).zfill(3)), bbox_inches='tight')
    i += 1
plt.close(fig)
```

That's it!

### 8.3 Some Uses of GANs

Generative Adversarial Networks (GANs) are one of the latest and most exciting developments in machine learning during the last decade (Goodfellow 2014). At this point, the use of GANs has been focused on research for image processing and synthetic generation. However, several studies have looked at the application of GANs to cyber security problems.

Currently, GANs have been used to generate works of art in the styles of Picasso, for instance, or they can potentially generate text that is similar to the styles of Shakespeare or other great authors. The application of GANs to cyber security is more recent but there already exists a body of work to highlight possible applications. In particular, the common theme is that GANs can be used by attackers to masquerade their efforts. Recent works have used GANs for password generation (Hitaj 2017) and steganography (Shi 2017). It is easy to see how this idea could also be extended to polymorphic viruses and synthetically generated network attacks.

Understanding how attackers can use GANs to masquerade their efforts is critical to understanding how to develop better intrusion or malware detection systems.

## **8.4 Summary**

In this chapter, a description of Generative Adversarial Networks was provided. Some sample code was addressed as well as some applications of GANs to cyber security.

## CHAPTER 9: REINFORCEMENT LEARNING

In this section of the book I will cover the topic of Reinforcement Learning. This is an area of machine learning somewhere between supervised learning and unsupervised learning. It has been extensively applied to recommender systems and AI-based games. Recently, it was shown that a deep Q-network, using only pixels and game scores as inputs, could achieve a playing level comparable to that of professional human gamers across a set of 49 Atari games (Mnih et al. 2015). The main advantage of applying reinforcement learning to games is that games are governed by rules. You have game states (the inputs) and actions (output) that lead to new states and rewards (the objectives to maximize). Because of this, no annotation is needed and instead you rely on the rules of the game for feedback (e.g. instead of annotated labels).

There are several types of reinforcement learning techniques. In this chapter, I will focus on getting started with Q-learning since this is the technique used in the Mnih et al (2015) paper I referenced above. Here, I will try to provide a simple intuition based description of the technique. I should note that to achieve the level of Q-Learning presented in the Mnih et al (2015) paper, several additional optimizations need to be included. However, the discussion in this chapter should provide a simple way to get started with Q-Learning.

So what is Q-Learning? Q-Learning tries to learn the value of being in a given state (s), and taking a specific action from there.

As I indicated, Q-learn has been applied to games. The best way to understand the algorithm is to analyze it from the point of view of a game. Here we will use Python's OpenAI Gym module to play games. We will select the simple FrozenLake game environment.

FrozenLake is a game about crossing a frozen lake that has some cracks in the ice with holes and there is wind sometimes that pushes the person crossing it. The game is very simple and consists of a grid that is 4x4 like so.

hole	frozen	cheese	hole
frozen	frozen	hole	frozen
hole	frozen	frozen	hole
frozen	hole	frozen	start

So, the objective is to get to the cheese without falling into a hole or being pushed by the wind into a hole. There are 4 moves which are up, down, right, and left. There is only one reward and that is to get to the cheese. However, you only get that reward in the future by first taking several steps on frozen blocks without falling in a hole. Therefore, one challenge is that you have to state your objective in terms of several future moves. This is accomplished using something called the Bellman Equation.

The key to predicting these rewards is to know the associated reward given a current state and action to take. This is called a Q mapping

$$Q(\text{state}, \text{action}) = \text{reward}$$

For such a simple grid, we could just use a table. In this case our table would be 16x4 because there are 16 possible states (position in the grid of 4x4) and there are 4 actions (up, down, right, left). Since we know the rules of the game and the layout of the grid, we can populate the table and learn the Q rewards for each state/action pair.

An example of the table can be seen below.

	Up	Down	Left	right
State1	Q=0.6	Q=0.8	Q=0.1	Q=0.0
State2				
State3				
State4	0		0.1	
State5				
State6			0	
State7		0.6		0
State8			1	
State9	0			
State10		0		1
State11			1	
State12	0.02			
State13			0.4	
State14		0.6		
State15		0.3	0	
State16		1		0

Figure. Q-Learn Table



Now the main challenge is that we need to learn future rewards for future actions as we move through the grid. Here the bellman equation will help. Think of the bellman equation as a type of recursive equation that looks at the future state given a current state. The Bellman equation is as follows:

$$Q(\text{state}, \text{action}) = \text{reward} + \text{weight} * \max [ Q(\text{future\_state}, \text{future\_action} ) ]$$

These values can be looked up from the Table.

The code discussed here can be downloaded from the course website or the github repository. In the next section, the python Q-learning code will be discussed which only uses a table to determine the rewards and the path to follow. Section 9.2 will use the same algorithm but will replace the use of the table with a neural network so that we can see how deep neural networks can improve the approach.

## 9.1 Q-Learning using a Table

In this section we discuss the code to implement Q-Learning using a table. This code makes use of the OpenAI gym library. The libraries used can be seen in the next code segment.

```
import numpy as np
import gym
```

The frozenLake game can be initialized by creating the **env** object as can be seen below. This object represents the game and holds all the parameters related to states, actions, rewards, and current game state.

```
env = gym.make('FrozenLake-v0')
```

The next step is to initialize the table **Q** to all zeros and of size 16x4. Here `env.observation_space.n = 16` and `env.action_space.n = 4`.

```
Q = np.zeros([env.observation_space.n, env.action_space.n])
lr = .8
γ = .95
num_episodes = 2000
```

We take 2000 epochs (or episodes) and initialize some parameters **lr** and **γ**. Each episode represents a game played. We use **jList** and **rList** to collect the number of steps taken per episode and the total reward per episode, respectively. These are used to collect results of each game.

```
jList = []
rList = []
```

The following code segment goes over the main loop of the Q-learn algorithm. In the next code segment, the line

```
for i in range(num_episodes):
```

indicates that we are going to play `num_episodes=2000` games. During these 2000 tries we will learn the best path to take.

```

for i in range(num_episodes):
    s = env.reset()
    rAll = 0
    d = False
    j = 0
    while j < 99:
        j+=1
        zz = env.action_space.n
        a=np.argmax(Q[s,:]+np.random.randn(1,zz) *(1.0/(i+1)))
        s1,r,d,_ = env.step(a)
        Q[s,a] = Q[s,a] + lr*(r + y*np.max(Q[s1,:]) - Q[s,a])
        rAll += r
        s = s1
        if d == True:
            break
    #jList.append(j)
    rList.append(rAll)

```

The line

```
s = env.reset()
```

restarts the game for every episode so we can play it again and assign the initial state to **s**. The variable **rAll** adds up the accumulated rewards for this episode. The variables **d** and **j** are control variables to indicate if the game has ended and to count the number of steps taken.

The code in the while loop is what allows the algorithm to learn or update the values in the Q table designated by the variable **Q**. To take the first step we need to pick an action to follow. We do this with the following lines of code

```

zz = env.action_space.n
a=np.argmax( Q[s,:]+np.random.randn(1,zz) *(1.0/(i+1)) )

```

The variable **zz** is the size *n* of all actions in the game (up, down, left, right) which in this case is 4. The statement `Q[s, :]` selects the current *Q* values (rewards) associated with state **s**. The statement

```
np.random.randn(1, zz) * (1.0 / (i+1))
```

adds randomness to the 4 *Q* values for the current state. Basically, you randomly increment the *Q* values for the current state and then select the highest one with

```
np.argmax()
```

by selecting the highest *Q* value you determine what action (**a**) you take given the current state.

Once the action **a** is selected, we can proceed to evaluate it in the game to obtain our new state (position) and the reward (did we fall in a hole or advanced to a frozen block). We do this with

```
s1, r, d, _ = env.step(a)
```

here, **s1** is the new state (position) and **r** is the reward. The parameter **d** indicates end of the game. Given this new information about the result of our action, we can proceed to update the *Q*-table with our new results and new knowledge about the state of the game. This is done with the statement

```
Q[s, a] = Q[s, a] + lr * (r + gamma * np.max(Q[s1, :]) - Q[s, a])
```

In this statement, `Q[s, a]` contains the current *Q* value (reward) associated with the state **s** and the action **a**. This is the Bellman equation which can be viewed as

```

next_s_Q = lr*(r + y*np.max(Q[s1,:]) - Q[s,a])
Q[s,a] = Q[s,a] + next_s_Q

```

**next\_s\_Q** contains the current reward for state **s** plus the maximum reward for the next state **s1**. The parameters **lr** and **y** are weights to control the importance of the next state's reward when updating the current states reward (Q value).

We can think of this parameter

```

- Q[s,a] )

```

as a regularization parameter.

At this point we are almost done and we can proceed to accumulate our results. The statement

```

rAll += r

```

accumulates the total rewards. The statement

```

s = s1

```

assigns the current state **s1** to **s**. The line

```

if d == True:
    break

```

ends the game if **d** indicates end of game. The statement

```

jList.append(j)

```

accumulates the number of steps taken to reach end of game. The statement

```

rList.append(rAll)

```

appends rewards per game to a list so that they can be viewed later.

```
print "Score over time: " + str(sum(rList)/num_episodes)
```

That is it. We have finished our discussion of Q-learn with tables on the frozenLake game. Now we can proceed to replace the table with a neural network.

## 9.2 Q-Learning using a Neural Network

Now that we understand the frozenLake game with a table, we can proceed to replace the table with a neural network. It is important to note here that the weights matrix **W** in the neural network will now represent the Q table.

In this section of the chapter I will only discuss the parts that are different from the previous implementation.

First we include the libraries as can be seen below. Notice we now add Tensorflow.

```
import gym
import numpy as np
import random
import tensorflow as tf
import matplotlib.pyplot as plt
```

We create the game with the **env** object.

```
env = gym.make('FrozenLake-v0')
```

Next, we define our familiar neural network functions `inference()`, `loss()`, and `train()`. The function `inference()` creates **W** which is our new Q table. Notice the dimensions of **W** are 16x4 because we have 16 states in the game and 4 actions. **Qout** (our predicted **y** in previous chapters) is the result of a matmul operation between `inputs1` (our states) and **W** (the weights or Q values in this case).

With

```
predict = tf.argmax(Qout,1)
```

we select the action (**a**) to take. Here is the code for the inference function.

```
def inference(inputs1):  
    W = tf.Variable(tf.random_uniform([16,4],0,0.01))  
    Qout = tf.matmul(inputs1,W)  
    predict = tf.argmax(Qout,1)  
    return predict, Qout, W
```

As can be seen from the previous code, the network looks like the figure below. It is important to note that this is a basic architecture and that much more complex deep architectures with different activation functions could be used such as architectures with many hidden layers or convolutional neural networks, etc.

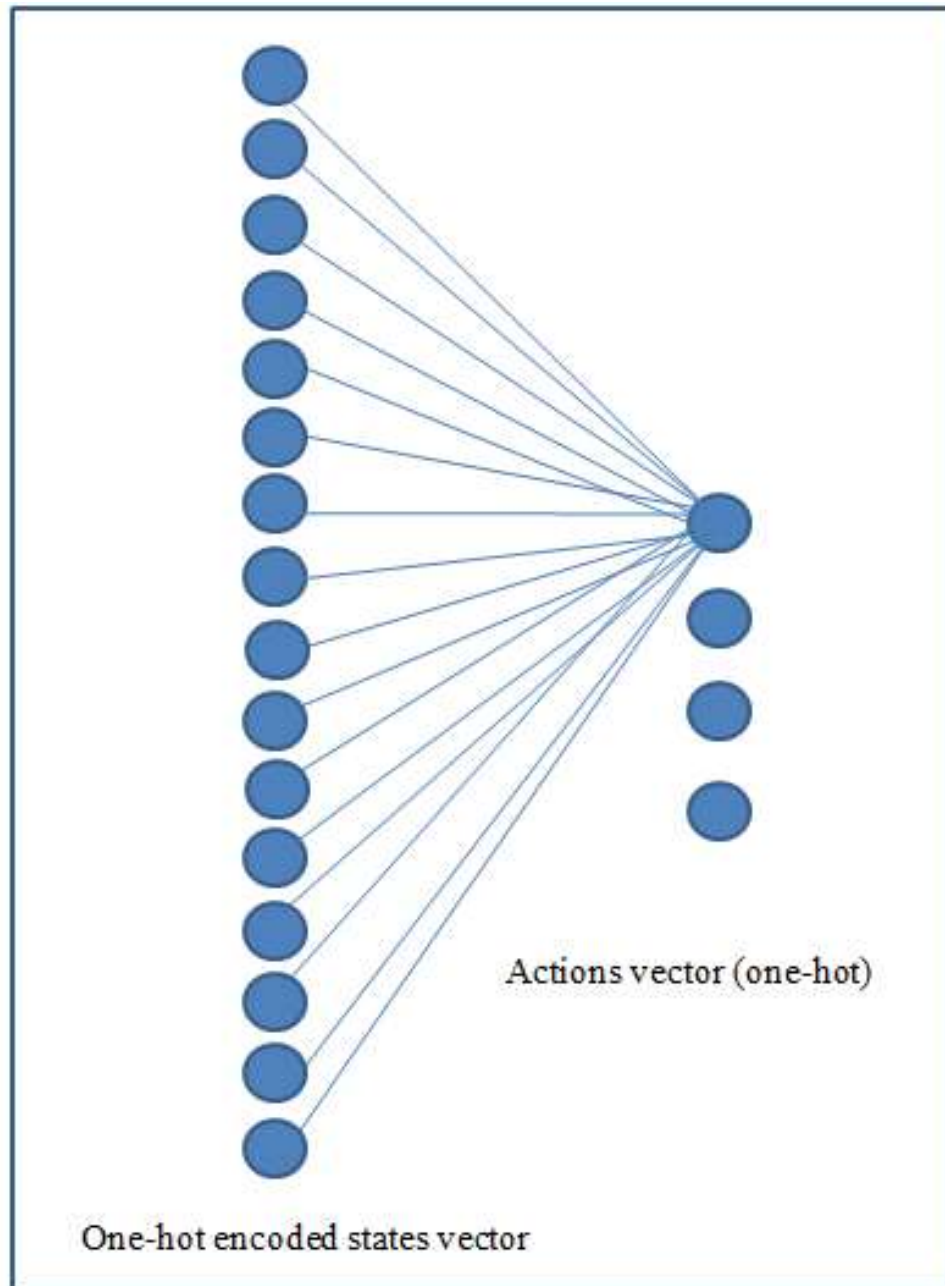


Figure. Q-Learning network.



The loss function is Least Squares Estimation which is the same as linear regression! Here, basically, we compare `Current_Q` to `estimated_Q` and try to minimize the error.

```
def loss(nextQ, Qout):  
    loss = tf.reduce_sum(tf.square(nextQ - Qout))  
    return loss
```

The optimization is nothing more than the very familiar Gradient Descent with a learning rate of 0.1.

```
def train(loss):  
    trainer = tf.train.GradientDescentOptimizer(learning_rate=0.1)  
    updateModel = trainer.minimize(loss)  
    return updateModel
```

I leave `evaluate()` for the reader to complete as an exercise.

```
def evaluate():  
    print "evaluate"
```

In the next statement we initialize the placeholder to hold the data. The placeholder **`inputs1`** holds the one hot encoded vector representing the state of the game.

The placeholder **`nextQ`** is used to store the one hot encoded vector of the 4 possible rewards for each action to take.

```
tf.reset_default_graph()

inputs1 = tf.placeholder(shape=[1,16],dtype=tf.float32)
nextQ = tf.placeholder(shape=[1,4],dtype=tf.float32)
```

Next, we call the core functions like so

```
predict, Qout, W = inference(inputs1)
cost = loss(nextQ, Qout)
trainOp = train(cost)
```

Now we are ready for the main loop. We initialize the variables in the graph and a few parameters.

```
init = tf.initialize_all_variables()
y = 0.99
e = 0.1
num_episodes = 2000
```

Then we create lists to contain total steps taken per episode (game) and total rewards per game.

```
jList = []
rList = []
```

Finally, we are ready for the main loop which is shown in the next code segment below.

```
with tf.Session() as sess:
    sess.run(init)
    for i in range(num_episodes):
        s = env.reset()
        rAll = 0
        d = False
        j = 0
        while j < 99:
            j=j+1
            a,allQ = sess.run( [predict,Qout], feed_dict=
                               {inputs1:np.identity(16)[s:s+1]})

            s1,r,d,_ = env.step(a[0])
            Q1 = sess.run(Qout, feed_dict=
                          {inputs1:np.identity(16)[s1:s1+1]})

            maxQ1 = np.max(Q1)
            targetQ = allQ
            targetQ[0,a[0]] = r + y*maxQ1
            _,W1 = sess.run( [trainOp, W], feed_dict=
                             {inputs1:np.identity(16)[s:s+1],nextQ:target
                              Q})

            rAll += r
            s = s1
            if d == True
                break
        jList.append(j)
        rList.append(rAll)
```

As an be seen in the code segment below, we run the main loop 2000 times (**num\_episodes**) which means that we play 2000 games. Each time we play a game, we reinitialize the board ( `s = env.reset()` ) and initialize the rewards

variable (**rAll**) to zero. The variable **j** is the counter for the current step and **d** is used to determine in the game is over (win or loss).

```
for i in range(num_episodes):
    s = env.reset()
    rAll = 0
    d = False
    j = 0
```

for every game iteration we run the following while loop. This while loop is the main code that helps us to learn that Q values and traverse the board (e.g. play the frozen lake game).

```
while j < 99:
    j=j+1
    a,allQ = sess.run( [predict,Qout], feed_dict=
        {inputs1:np.identity(16)[s:s+1]})

    s1,r,d,_ = env.step(a[0])
    Q1 = sess.run(Qout, feed_dict=
        {inputs1:np.identity(16)[s1:s1+1]})

    maxQ1 = np.max(Q1)
    targetQ = allQ
    targetQ[0,a[0]] = r + y*maxQ1
    _,W1 = sess.run( [trainOp, W], feed_dict=
        {inputs1:np.identity(16)[s:s+1],nextQ:targetQ})

    rAll += r
    s = s1
    if d == True
        break
```

We perform 99 steps since it should not take more than 99 steps to traverse the frozen lake. If it does, the game should end. The first line in the while loop is used to increment the steps

```
j=j+1
```

after incrementing the steps, we proceed to perform our first session run operation to train the Tensorflow graph. Here we call **predict** and **Qout** from the `inference()` function calls.

```
a,allQ = sess.run( [predict,Qout], feed_dict=
                    {inputs1:np.identity(16)[s:s+1]})
```

The statement

```
np.identity(16)[s:s+1]
```

takes the current state in the variable **s** and converts it into a one-hot encoded representation. For instance, if the current state is 4, then the one-hot encoded representation (of size 16) looks like this

```
[0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

The next step is to take the predicted action in “**a**” and run it through the game. We use **a[0]** instead of just **a** because **a** is a tensor. Assuming the action is 1 (down), printing **a** alone will result in

```
[1]
```

Whereas, printing `a[0]` will result in

```
1
```

So the below statement runs the action through `env.step()` and this function returns the new state **s1** which is the new position in the frozen lake grid, **r** is the reward associated with the step **s** (for instance `r=0.43`), and **d** indicates if the game is over (found the cheese or fell in the frozen lake).

```
s1,r,d,_ = env.step(a[0])
```

with the new state **s1**, we proceed to run the Tensorflow graph again with session run. Here we call **Qout** again using **s1**.

Recall that **Qout** is

```
Qout = tf.matmul(inputs1, W)
```

in the inference function. In this case, **inputs1** is the one-hot encoded vector of size 16 that represents state **s1**.

```
Q1 = sess.run(Qout, feed_dict= {inputs1:np.identity(16)[s1:s1+1]})
```

So **Q1** will now contain the 4 neuron vector with the Q values for all 4 actions given state **s1**. Currently, the vector **allQ** for state **s** looks like this with some values

```
allQ = [ 0.0
         0.3
         0.4
         0.02 ]
```

And **Q1** for state **s1** looks like this for some values

```
Q1 = [ 0.9
       0.1
       0.04
       0.7 ]
```

Therefore, we have predicted Q values for state “s” and predicted Q values for state “s1”.

Next, we proceed to select the highest value in **Q1**. In this case **maxQ1** gets assigned the value 0.9 from our previous example (**maxQ1**=0.9). The code is as follows

```
maxQ1 = np.max(Q1)
```

now we use a new variable **targetQ** which will be equal to the bellman equation. We assign to it **allQ**

```
targetQ = allQ
```

so that **targetQ** is now

```
targetQ = [ 0.0  
            0.3  
            0.4  
            0.02 ]
```

Recall that **a[0]** holds the index of the action taken (e.g. down or 1). Therefore, in the vector **targetQ** we select that position ( **targetQ[0, 1]** ) and add to it the reward value **r** and **maxQ1** times some **y** parameter. Recall that the Bellman equation looks like this:

$$Q(\text{state}, \text{action}) = \text{reward} + \text{weight} * \max [ Q(\text{future\_state}, \text{future\_action} ) ]$$

The code is as follows

```
targetQ[0,a[0]] = r + y*maxQ1
```

and with values this looks like the following

```
targetQ[0,1] = 0.43 + 0.99*0.9
```

after this update rule **targetQ** has been modified from

```
targetQ = [ 0.0
            0.3
            0.4
            0.02 ]
```

to

```
targetQ = [ 0.0
            1.321
            0.4
            0.02 ]
```

interestingly, only one of the 4 values in **targetQ** is updated using the bellman equation. The other values remain the same. Finally, we do a final update of the Tensorflow graph by calling **trainOp** with session run and state “s”. Additionally, the placeholder **nextQ** is assigned the result from the Bellman equation **targetQ**.

```
_,W1 = sess.run( [trainOp, W], feed_dict=
                  {inputs1:np.identity(16)[s:s+1],nextQ:targetQ})
```



This is important because **nextQ** will be used in the loss function with the next predicted **Qout** like so

```
predict, Qout, W = inference(inputs1)
cost = loss(nextQ, Qout)
trainOp = train(cost)
```

Finally, the last piece of code adds up the rewards, assigns the new state **s1** to **s**, and checks to see if the game is over.

```
rAll += r
s = s1
if d == True
    break
```

once you exit the while loop, the last part is to append the results of the current game to **jList** and **rList**.

```
jList.append(j)
rList.append(rAll)
```

Well, that is it with the algorithm discussion. Finally, we print our results and plot them.

```
print "Percent of succesful episodes: " +  
      str(sum(rList)/num_episodes) + "%"  
  
plt.plot(rList)  
plt.show()  
plt.plot(jList)  
plt.show()
```

That is it. We have completed implementing our Q learning algorithm with a neural network. In the next section we will add a simple improvement to the code that will improve performance.

### 9.3 Q-Learning using a Neural Network and Randomness

In the previous section we described the code to implement Q-learning with a neural network on the frozen lake game. That was the simplest implementation of it.

To improve the results, we can add a few lines of additional code which will allow the algorithm to better converge and learn better Q-values. The additions are simple and basically relate to adding randomness to the code. Notice in the code above that a few new statements have been added.

```

with tf.Session() as sess:
    sess.run(init)
    for i in range(num_episodes):
        s = env.reset()
        rAll = 0
        d = False
        j = 0
        while j < 99:
            j=j+1
            a,allQ = sess.run( [predict,Qout], feed_dict=
                               {inputs1:np.identity(16) [s:s+1]})

            if np.random.rand(1) < e:
                a[0] = env.action_space.sample()

            s1,r,d,_ = env.step(a[0])
            Q1 = sess.run(Qout, feed_dict=
                          {inputs1:np.identity(16) [s1:s1+1]})

            maxQ1 = np.max(Q1)
            targetQ = allQ
            targetQ[0,a[0]] = r + y*maxQ1
            _,W1 = sess.run( [trainOp, W], feed_dict=
                             {inputs1:np.identity(16) [s:s+1],nextQ:target
                              Q})

            rAll += r
            s = s1
            if d == True:
                e = 1./((i/50) + 10)
                break
        jList.append(j)
        rList.append(rAll)

```

These new lines add randomness to the selection of the next action to take. The idea is that add the beginning of the learning process, the action prediction function may not be very good. Therefore, picking an action randomly at the beginning may be better than picking actions with the inference() function.

This is reflected in the code segment below.

```
if np.random.rand(1) < e:  
    a[0] = env.action_space.sample()
```

a random number is obtained and compared to **e**. If less than **e**, the action **a[0]** is selected randomly

```
a[0] = env.action_space.sample()
```

as the algorithm improves and the Q values are better, the value of **e** can be adjusted so that action is more often selected with the inference function and not with the random function

```
a[0] = env.action_space.sample()
```

The code can be seen here.

```
if d == True:  
    e = 1./((i/50) + 10)  
    break
```

where

```
e = 1./((i/50) + 10)
```

adjusts the value of “**e**”.

## 9.4 Summary

In this chapter we have discussed the Q learning algorithm as part of the larger topic of Reinforcement Learning using tables and neural networks with randomization.

## CHAPTER 10: TRANSFORMERS

Transformers are the latest step in the evolution of deep learning. As is necessary with progress, newer deep learning algorithms are also much more complicated, with deeper and more resource intensive networks. The Transformer is the best example of this. Transformers are, for me, the first algorithm I was not able to run on a laptop. They truly require a machine learning “war machine”. Lots of GPU power and memory, etc. The algorithms are much more complicated and, as you will see, the networks are very deep.

Transformers are one of the new innovations in NLP since 2017. They were first made popular by the paper “Attention Is All You Need” by Vaswani et al. (2017). They are very interesting and seem to be very powerful. Many researchers suggests that they are better than RNNs for NLP because they parallelize better and because of the Attention mechanism.

So far, Transformers have been used to develop very impressive implementations such as BERT (Devlin et al. 2018), and GPT-2 (Radford et al. 2019), as of this writing, which seem to be very good at language understanding. Transformers have been applied to language translation, question answering, document summarization, automatic code generation, text generation, etc. Okay, let’s get started.

### 10.1 Encoder Decoder with Multi-Head Attention

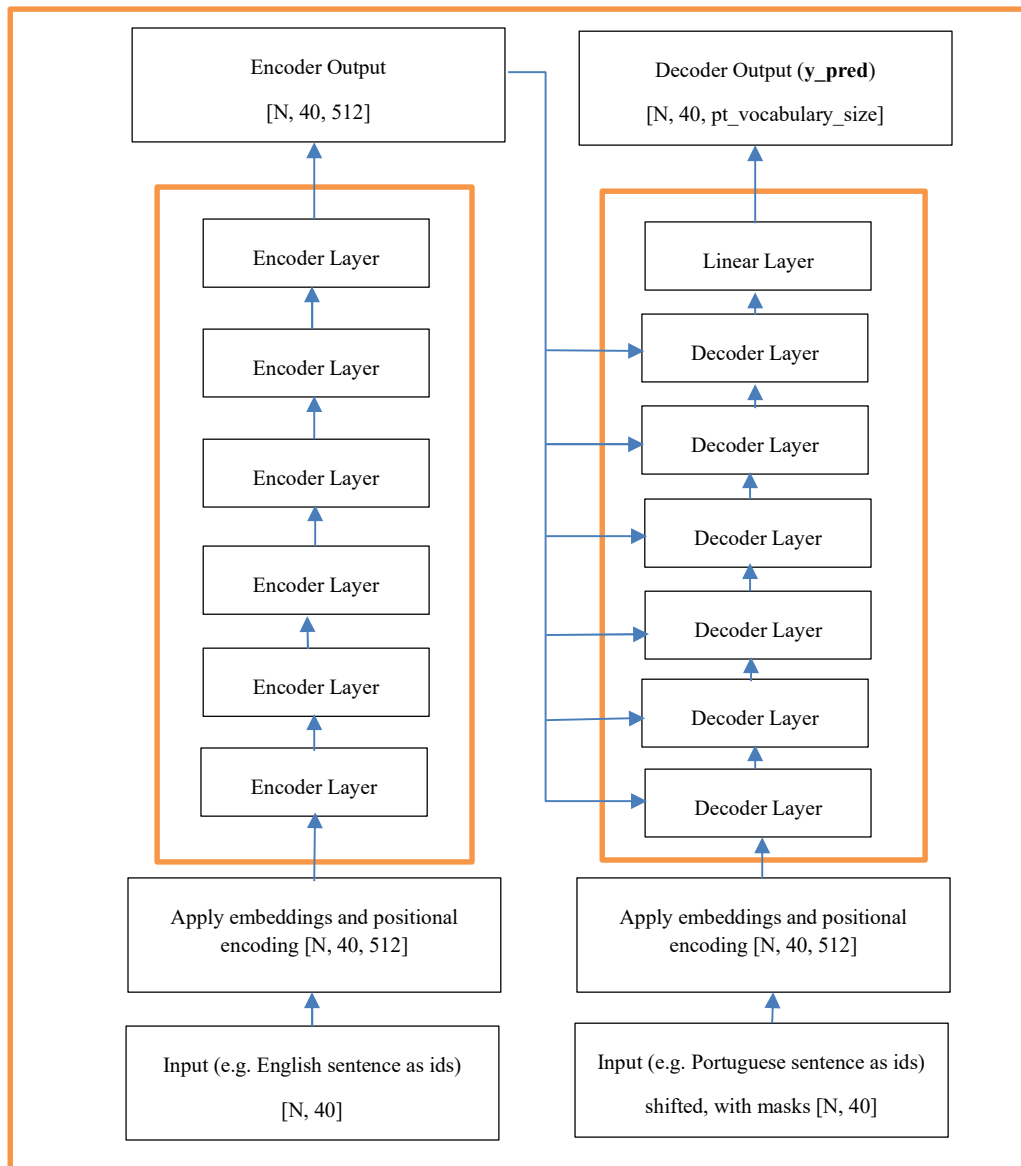
In this section, I will present the first version of the Transformer first made popular in the paper “Attention Is All You Need” by Vaswani et al. As of this writing, there are newer versions of Transformers called BERT, GPT-2, GPT-3, etc. I will simply call this first implementation the Encoder Decoder with Multi Head Attention

Transformer (that's a mouth full). The Encoder Decoder with Multi Head Attention Transformer is a very deep network. The architecture has an encoder followed by a decoder. The encoder has 6 sublayers called encoder layers.

Each encoding layer has a Multi-Head Attention layer followed by a standard fully connected feed forward layer. The input to the encoder goes through all this layers in the encoder and is converted into an encoder output. The input to the encoder and the output of the encoder have the same dimensions. For instance, here, the input to the encoder would be the English sentence (given a translation problem).

The decoder layer has 2 inputs. One input is the encoder output. The second input to the decoder varies based on whether you are training or predicting. If you are training, the input to the decoder is the sentence in the other language. For instance, the Spanish sentence. In the decoder, when training the Transformer, a mask is needed to prevent the model from seeing all the words it is trying to predict. This is called a look ahead mask.

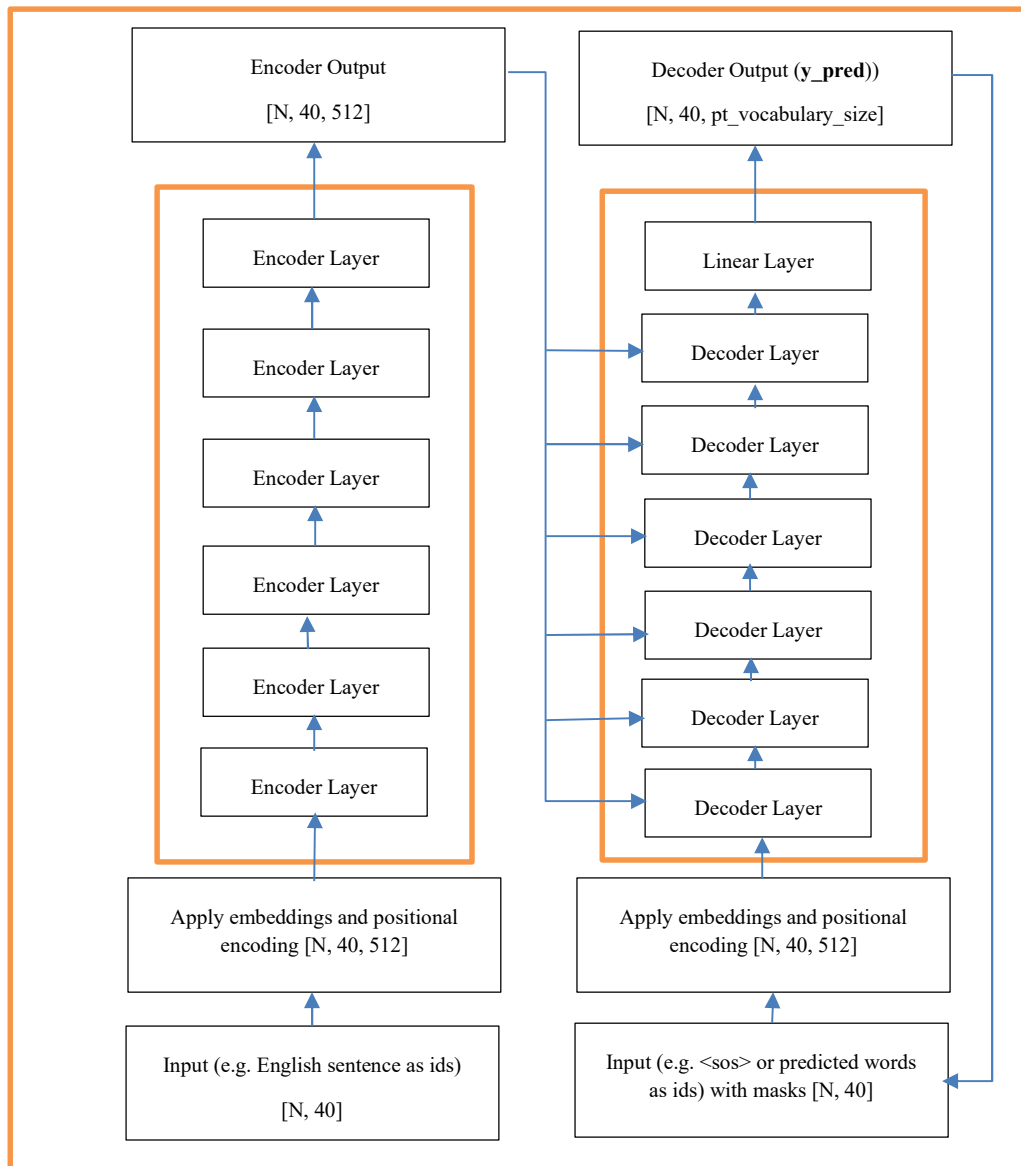
If you are testing, the input to the decoder is just the previous words before the word you are trying to predict. You start with a start of sentence token (e.g. < sos >) and predict iteratively. The predicted word is then added to the previous tokens and the process is repeated.



**Figure. Transformer (and inputs during training)**

The previous figure shows the Transformer model during training.





**Figure. Transformer (and inputs during testing)**

The previous figure shows the Transformer model and inputs during testing.

Now that we have looked at the big picture, we can proceed to discuss the main ideas of the Transformer model.

### **10.1.1 The Main Ideas of the Transformer**

So, where does one start with Transformers? The Transformer is complex and it involves several ideas to make them work correctly. In this section I will present the main ideas first with some relevant code. Understanding these concepts or steps really well before venturing to write the code for the whole Transformer is really important. It will save you time in the long run. So now, let us proceed to discuss these topics. In the next section I will start discussing the code for the the full transformer.

#### **Numpy arrays, tensors, and linear algebra**

Linear algebra, numpy arrays, and tensor operations are at the heart of understanding the Transformer architecture. Before you continue, I strongly recommend that you read and practice the topics in chapter 1, and in particular, the section on linear algebra, numpy arrays, and tensor operations.

#### **Inputs and outputs**

When dealing with deep neural networks I like to think of inputs and outputs first and treat the network as a black box. So, let us start there. Let's quickly remember our classic example of MNIST supervised classification. In MNIST standard feed forward classification, you have an input image which is 28x28 and a predicted vector of size 10 for the classes. So, what do the inputs and outputs look like for transformers? For language translation, they are lists of ids. Each id can represent a word in a sentence. This is best visualized with an example.

First, let us look at the classic use case for Transformers. As I said earlier, Transformers have been used extensively in NLP. And the simplest example is language translation where we have sentence pairs. Such as the following for English-Spanish translation:

"the cat is sleeping" --> which translates to --> "el gato esta durmiendo"

Therefore, first we need to understand how to encode this for the neural network and then to understand how exactly it is that the network will train and learn. So, again, before you look into the network's very deep and complex layers, I believe that one needs to focus on:

- Taking text sentences and converting them into sequences of ids
- Padding these sequences of ids

Consider that after encoding and padding, your sentences will look like this:

English

```
[12110 203 4 3947 29 2 168 2 4 27 68 4333 8 3622 2943
1012 1 12111 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0]
```

Spanish

```
[12110 13 4 3947 29 2 5 32 36 16 1145 4 58 34 7905 58
25 28 354 2482 3 17 27 28 4395 9 2886 7 12111 0 0 0
0 0 0 0 0 0 0]
```

## **Masks**

Masks serve several purposes. One is to help ignore the padded values during training. The other goal is to block the given word you want to predict (or future words). This brings up the important aspect of training with Transformers. Transformers predict the last word in a sequence. For example:

Given an input in english: "the cat is sleeping"

a Transformer is also given part of the output sentence. In this case: "el gato esta ?". The Transformer will predict the next word in the sequence which in this case would be "durmiendo" to complete the translation as "el gato esta durmiendo". All of this is achieved through the masks to ignore padded values and to only show the partial sentence. The type of training that will be used for Transformer training is called "Teacher Forcing". So definitely understand this concept.

## **Teacher forcing**

You may have already read somewhere (on-line) that the Transformer network predicts one word at a time and that that word is read back as an input in the next iteration. Also, the network predicts the last word in the sequence of words. But you may think, aren't those last words just padding? Eh? So, what is going on here? As it turns out, the mechanism of predicting one word at a time and feeding it back as an input in the next iteration is only done during the testing phase and it is not done during training. Instead, during training we use "Teacher Forcing".

Teacher forcing is a technique in auto regressive models where you do not use the predicted outputs of the decoder to feed back as input but instead you use the real data. This helps the model to learn the correct information instead of its own erroneous predictions (especially at the beginning of training).

## Attention

The Attention mechanism in Transformers is the heart of the whole algorithm. The attention matrix is nothing more than a dot product matrix multiplication between all the words in a sentence (e.g. the input English sentence). The idea is that, given the input and output, the model learns to correlate the words in the sentence to determine their importance. This is done multiple times and that is why it is called a multi head attention mechanism.

## Embeddings

Embedding converts the sequence of ids into a sequence of embeddings. You will go from a 2d tensor to a 3d tensor of size:

$$N * \text{seq\_length\_max} * \text{embedding\_dimension}$$

where **N** is the batch size, **seq\_length\_max** is 40, and **embedding\_dimension** is 512.

## Positional Encoding

This is the technique that allows you to encode sequence. Transformers are all about being parallel. Their direct competitor is Recurrent Neural Networks (RNNs). RNNs have had several problems in the past. One is that they do not scale well to GPUs and parallel approaches because of their recurrence and dependence on previous steps. The other problem is the famous "vanishing gradients" problem

which was addressed by LSTMs. Transformers did away with type of sequence modeling approach used in RNNs all together so they are very good for parallel approaches. But how do they address or encode the sequence? Obviously knowing that the word "cat" goes before the word "sleeping" is useful. This is where a technique called positional encoding comes into play. Basically, after embedding, you have a vector per word of, say, size 512.

Now, with positional encoding, a function that calculates sines and cosines, is used to create a new vector also of size 512 that represents position (i.e. sequence) of the words. The 2 vectors are added together (embedding + positional\_encoding) to get the new inputs to the network. Also, the position vector values are smaller than the embedding vector values so as to not let position dominate.

To illustrate this, we can look at the code to generate a positional encoding vector and visualize it with matplotlib. In the code segment below, first you create a vector of dimension 512 called “**i**”. Then, you create the position vector **pos**. After that, With the vector “**i**”, you calculate the angle rates (**angle\_rates**) with the formula

```
angle_rates = 1 / np.power(10000, (2 * (i // 2)) / 512.0 )
```

Given the vectors **pos** and **angle\_rates**, we perform a multiplication to get a matrix (**angle\_rads**) of size [40, 512]. Now we can calculate the sines and cosines of **angle\_rads** and plot it.

```

## [64, 40, 512]
batch = np.random.normal(0, 1, (64, 40, 512))
print(batch)

## np.newaxis changes i from [512,] to [1, 512]
i = np.arange(512)[np.newaxis, :]

print("i")
print(i)

## np.newaxis changes pos from [40,] to [40, 1]
pos = np.arange(40)[:, np.newaxis]

print("pos")
print(pos)

angle_rates = 1 / np.power(10000, (2 * (i // 2)) / 512.0 )

print(" angle_rates = 1 / np.power(10000, (2 * (i // 2)) / 512 ) ")
print(angle_rates)

## multiply 2 vectors to get a matrix of size [40, 512]
angle_rads = pos * angle_rates

print("angle_rads = pos * angle_rates")
print(angle_rads)

print("i.shape ", i.shape)
print("pos.shape ", pos.shape)
print("batch.shape ", batch.shape)
print("angle_rates.shape ", angle_rates.shape)
print("angle_rads.shape ", angle_rads.shape)

angle_rads[:, 0::2] = np.sin(angle_rads[:, 0::2])    ## even index
angle_rads[:, 1::2] = np.cos(angle_rads[:, 1::2])    ## odd index

print("sin and cos to angle_rads ")
print(angle_rads)

```

```

## plot
plt.pcolormesh(angle_rads, cmap='RdBu')
plt.xlabel('Depth')
plt.xlim((0, 512))
plt.ylabel('Position')
plt.colorbar()
plt.show()

#####

angle_rads = angle_rads[np.newaxis, ...] ## (1, 40, 512) for broadcasting
print("angle_rads[np.newaxis, ...] ")
print("angle_rads.shape ", angle_rads.shape)

```

The previous code gives us the following dimensions of all the vectors we have created.

```

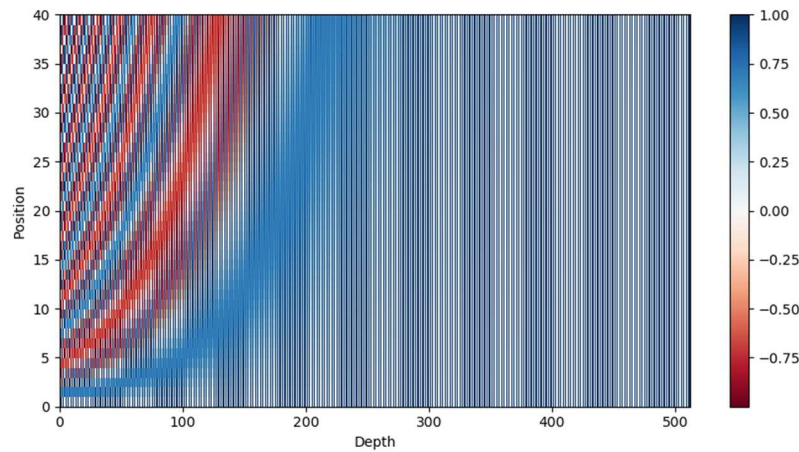
i.shape (1, 512)
pos.shape (40, 1)
batch.shape (64, 40, 512)
angle_rates.shape (1, 512)
angle_rads.shape (40, 512)
sin and cos to angle_rads

angle_rads[np.newaxis, ...]
angle_rads.shape (1, 40, 512)

```

And it generates this graph with is our positional encoding tensor.





**Figure. Positional Encoding Tensor**

Now that we have discussed some of the main ideas in the Transformer, we are now ready to start discussing the code.

### 10.1.2 Code

Now that we have discussed the main ideas of the Transformer, let us proceed to write the code for a full Encoder Decoder with Multi Head Attention Transformer. This is for the implementation of a Transformer-based Translator using the English to Portuguese dataset. The code and data set are available on the book GitHub. The implementation is done on the Tensorflow low level API with a static graph for more clarity and detail of the algorithm. I tested this code and implemented it on Tensorflow version 1.10 using a MacBook Pro and a GPU box with an Nvidia RTX 2080 Ti (11 VRAM), AMD processor with 12 cores, and with 128 GB of RAM. I used the Mac for writing and debugging of the code with reduced parameters such as batches of 16 and very few iterations. The Mac cannot run

efficiently using full batches and full epochs with an actual data set so you will need a GPU machine (even for the tiny English-to-Portuguese data set). Remember that transformers are very deep and very resource intensive. The data set used here is the Tensorflow English-to-Portuguese data set which has about 50,000 sentence pairs for training. On the GitHub, I also provided links to a larger English-to-Spanish dataset with about 2,000,000 sentence pairs. I wrote a paper about this larger data set based experiment and you can read about the results there.

Okay, with that out of the way, let's get started! First, let us introduce the libraries.

```
import sklearn
import tensorflow as tf
import numpy as np
import nltk
from nltk.tokenize import word_tokenize
from numpy import genfromtxt
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score, f1_score
import pandas as pd
import pickle
import collections
```

In the code segment below, I provide the usual settings to the code.

```
## some settings

import warnings
warnings.filterwarnings("ignore")
np.set_printoptions(threshold=np.inf) #print all values in numpy array
```

In this section we specify the parameters to the transformer. The transformer uses batches of size 64, an embedding layer of size 512, maximum sentence length of 40. These are the parameters used in the Vaswani et al. (2017) paper.

```
## parameters

batch_size = 64

d_model = 512  ## hidden dimension of encoder/decoder
d_ff = 2048    ## hidden dimension of feedforward layer

dropout_rate = 0.3
smoothing = 0.1  ## label smoothing rate
learning_rate = 0.0001  ## 0.0003

MAX_LENGTH = 40

START_TOKEN = "<sos>"
END_TOKEN   = "<eos>"
UNK_TOKEN   = "<unk>"

n_epochs = 2000

VOCAB_SIZE_EN = 0
VOCAB_SIZE_PT = 0
```

## Data Wrangling

Tensorflow 2.0 offers many new techniques for extracting and processing data sets. As I like building things from scratch, I will present my own approach to data wrangling. The approach is very standard and is similar to what you do in NLP for algorithms like word2vec, for instance.

I like working with python dictionaries so I extracted the data set and created python dictionaries for training and testing. I saved the dictionaries to a Python pickle files for ease of use. The following code shows how to load the dictionaries.

```
def load_dictionary(file_name):
    with open(file_name, 'rb') as handle:
        dict = pickle.loads( handle.read() )
    return dict
```

After loading the data sets from file, you have to create the dictionary and reverse dictionary. You create 2 dictionaries per language (e.g. two for English and two for portuguese). These are dictionaries of ids to tokens and vice-versa. Notice that I set the vocabulary size to 12,000. You can play with this value for optimal performance.

```
## Includes <eos> and <sos> tokens

def build_dataset(words):
    START_TOKEN = "<sos>"
    END_TOKEN   = "<eos>"
    UNK_TOKEN   = "<unk>"
    count = collections.Counter(words).most_common(12000)
    dictionary = dict()
    for word, _ in count:
        ## add + 1 so that 0 is not used as an index to avoid padding conflict
        dictionary[word] = len(dictionary) + 1      ## + 1
    size_vocab = len(dictionary)
    dictionary[START_TOKEN] = size_vocab
    dictionary[END_TOKEN]   = size_vocab + 1
    dictionary[UNK_TOKEN]   = size_vocab + 2
    reverse_dictionary = dict(zip(dictionary.values(), dictionary.keys()))
    return dictionary, reverse_dictionary
```

The following is an example function in case you want to process sentences with regular expressions or tokenize manually.

```
def preprocess_sentence(sentence):
    sentence = sentence.lower().strip()
    # creating a space between a word and the punctuation following it
    # eg: "it is a cat." => "it is a cat ."
    sentence = re.sub(r"([?!.!])", r"\1 ", sentence)
    sentence = re.sub(r"["+ " "]+',', " ", sentence)
    # replacing everything with space except (a-z, A-Z, ".", "?", "!", ", ")
    sentence = re.sub(r"[^a-zA-Z?!.!]+", " ", sentence)
    sentence = sentence.strip()
    return sentence
```

For tokenization, I used the NLTK tokenizer. In the paper “Attention Is All You Need”, the authors used byte pair encoding. Byte pair encoding does not use full words as tokens. Instead, you do something like this: "walk" and "ing" for the word “walking”. Therefore, words are broken into smaller elements. Byte-pair encoding is used to tokenize sentences in a language, which, like the WordPiece encoding, breaks words up into tokens that are slightly larger than single characters but less than entire words.

```
def get_tokens(sentence_list):
    tokens_list = []
    for sentence in sentence_list:
        tokens = word_tokenize(sentence)
        for word in tokens:
            tokens_list.append(word)
    tokens_list = np.array(tokens_list)
    return tokens_list
```

Once you have the dictionaries and the tokens, you can proceed to convert words into ids with the encode function.

```
def encode(sentence, dictionary):
    ids_list = []
    tokens = word_tokenize(sentence)
    for word in tokens:
        if word in dictionary.keys():
            ids_list.append( dictionary[word] )
    return ids_list
```

Decoding is just the process in reverse. Here you convert ids back to tokens using the reverse dictionary for convenience and speed up.

```
def decode(list_ids, reverse_dictionary):
    words_list = []
    for id in list_ids:
        if id in reverse_dictionary.keys():
            words_list.append( reverse_dictionary[id] )
    return words_list
```

The following function aligns the English and Portuguese sentence pairs and creates two lists.

```
## this returns 2 lists of english and portuguese sentences that are aligned by index

def get_en_and_pt_sentences(train_dict):
    en_list, pt_list = [], []
    for key, val in train_dict.items():
        print(key)
        print(val)
        en_list.append( val['en'] )
        pt_list.append( val['pt'] )
    return en_list, pt_list
```

The below line of code just loads the sentences data before processing from the pickle objects.

```
## Read in the data of english and portuguese sentences

train_dict = load_dictionary("data/en_pt_train_dictionary.txt")
validation = load_dictionary("data/en_pt_val_dictionary.txt")
```

The next function creates 2 lists of aligned English and Portuguese sentences.

```
english_sentence_list, portuguese_sentence_list = get_en_and_pt_sentences(train_dict)
```

The function **get\_tokens** converts each sentence into a list of tokens.

```
print("creating the dictionaries takes a while ... ")

en_tokens = get_tokens(english_sentence_list)
pt_tokens = get_tokens(portuguese_sentence_list)
```

After creating the dictionaries for each language, we calculate the vocabulary size for each language.

```
## when 2 languages, you have 2 separate tokenizers.

en_dictionary, en_reverse_dictionary = build_dataset(en_tokens)
pt_dictionary, pt_reverse_dictionary = build_dataset(pt_tokens)

VOCAB_SIZE_EN = len(en_dictionary)
VOCAB_SIZE_PT = len(pt_dictionary)

print("vocab size english ", VOCAB_SIZE_EN)
print("vocab size portuguese ", VOCAB_SIZE_PT)
```

The following “**for**” loop brings all the previous functions together. It results in 2 lists of sentence ids, one for each language (2 lists of numpy objects). Notice that to each sentence list of ids we add the start token id at the beginning and the end token id at the end.

The final “**if**” statement is used to only include sentences shorter than 40 tokens (the max length I used). Sentences shorter than 40 will be padded but all sentences will eventually be tensors of size 40 (n\_tokens + padding).

```
english_sentence_ids_list = []
portuguese_sentence_ids_list = []

for i in range( len(english_sentence_list) ):
    en_sentence = english_sentence_list[i]
    pt_sentence = portuguese_sentence_list[i]

    en_sentence_ids = encode(en_sentence, en_dictionary)
    pt_sentence_ids = encode(pt_sentence, pt_dictionary)

    en_sentence_ids = np.array(en_sentence_ids)
    pt_sentence_ids = np.array(pt_sentence_ids)

    en_START_TOKEN_id = en_dictionary['<sos>']
    en_END_TOKEN_id = en_dictionary['<eos>']

    pt_START_TOKEN_id = pt_dictionary['<sos>']
    pt_END_TOKEN_id = pt_dictionary['<eos>']

    en_sentence_ids = np.concatenate(
        [ [en_START_TOKEN_id], en_sentence_ids, [en_END_TOKEN_id] ] )
    pt_sentence_ids = np.concatenate(
        [ [pt_START_TOKEN_id], pt_sentence_ids, [pt_END_TOKEN_id] ] )

    if len(en_sentence_ids) <= MAX_LENGTH and len( pt_sentence_ids ) <= MAX_LENGTH:
        english_sentence_ids_list.append( en_sentence_ids)
        portuguese_sentence_ids_list.append( pt_sentence_ids)
```

While I never use keras, I decided to use keras for this part (to save me time). Here, Keras is not used to define architecture. Instead, the function



### `tf.keras.preprocessing.sequence.pad_sequences`

is Keras's standard padding function. I will change this in a later version of this book and will write the function from scratch. Notice that english sentences are inputs to the encoder and portuguese sentences are inputs to the decoder when training. During training, Teacher Forcing is used for the decoder with masking to prevent look aheads. This will be discussed later. Later, I will also discuss a step called shifting for the decoder inputs. Shifting removes one token from the inputs and aligns them. Given that I want everything to be the same size I have made the max length for the Portuguese sentences 41 instead of 40.

```
en_MAX_LENGTH = MAX_LENGTH
pt_MAX_LENGTH = MAX_LENGTH + 1

english_sentence_ids_list = tf.keras.preprocessing.sequence.pad_sequences(
    english_sentence_ids_list, maxlen=en_MAX_LENGTH, padding='post')

portuguese_sentence_ids_list = tf.keras.preprocessing.sequence.pad_sequences(
    portuguese_sentence_ids_list, maxlen=pt_MAX_LENGTH, padding='post')
```

If you would like to view the data, you can do so with the following code.

```
for i in range( len(english_sentence_ids_list) ):
    print("@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@")
    print(english_sentence_ids_list[i])
    print(portuguese_sentence_ids_list[i])
    ## input()
```

After padding, the data will look like this:

en

```
[12110 203 4 3947 29 2 168 2 4 27 68 4333
 8 3622 2943 1012 1 12111 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0]
```

pt

```
[12210 13 4 3947 29 2 5 32 36 16 1145 4
58 34 7905 58 25 28 354 2482 3 17 27 28
4395 9 2886 7 12211 0 0 0 0 0 0 0
0 0 0 0]
```

And without padding, the data will look like this:

en

```
[12110 13 4 3947 29 2 5 32 36 16 1145 4
58 34 7905 58 25 28 354 2482 3 17 27 28
4395 9 2886 7 12111 ]
```

pt

```
[12210 62 585 132 202 4395 11969 3 43 18 27 107
7042 15 10 814 11717 4 4053 89 2960 2 157 119
1 12211 ]
```

It has been shown by various researchers that Xavier initialization gives better results for this type of problem. The following is an implementation of Xavier initialization.

```
def xavier_init(size):
    in_dim = size[0]
    xavier_stddev = 1. / tf.sqrt(in_dim / 2.)
    return tf.random_normal(shape=size, stddev=xavier_stddev)
```

As described before, we must record the position of the words in the sentence. We do this with positional encoding. The input to the positional encoding function is the embedding tensor which is of size  $[N, 40, 512]$ .

$N$  is the batch size which in this case is 64, 40 is the number of tokens in a sentence plus padding, and 512 is the embedding size. That is, we embed each id from the one hot encoded vector of size “`vocabulary_size`” to a smaller vector of size 512 using:

```
embeddings_en = tf.Variable( tf.random_uniform( [VOCAB_SIZE_EN, 512], -1.0, 1.0) )
embed_en_enc_in = tf.nn.embedding_lookup(embeddings_en, x_ph_enc_in)
```

This is the standard Tensorflow embedding lookup approach.

```
def positional_encoding(embeddings, dropout):    ## embeddings is [N, 40, 512]

    ## tf.newaxis changes i from [512,] to [1, 512]
    i_vec = np.arange(512.0)[np.newaxis, :]

    ## tf.newaxis changes pos from [40,] to [40, 1]
    pos = np.arange(40.0)[:, np.newaxis]

    angle_rates = 1 / np.power(10000, (2 * (i_vec // 2)) / 512.0 )

    ## multiply 2 vectors to get a matrix of size [40, 512]
    angle_rads = pos * angle_rates

    ## this assignment operation cannot be done in tensorflow
    ## have to do this in numpy because tensors are not assignable

    angle_rads[:, 0::2] = np.sin(angle_rads[:, 0::2])    ## even index
    angle_rads[:, 1::2] = np.cos(angle_rads[:, 1::2])    ## odd index

    ## angle_rads [40, 512]

    angle_rads = angle_rads[np.newaxis, ...]    ## (1, 40, 512) for broadcasting

    pos_encoding = angle_rads

    ## embeddings      +      pos_encoding      ## broadcasting
    ## [N, 40, 512]    +      [1, 40, 512]    = [N, 40, 512]
    ## for broadcasting

    ## tensorflow allows you to add numpy array to tensor
    emb_pos = embeddings + pos_encoding

    ## dropout is applied to the combination of pos_encoding and embeddings
    emb_pos = tf.nn.dropout(emb_pos, dropout)

    return emb_pos    ## this is [N, 40, 512]
```

Notice that in the positional\_encoding function we create all vectors using numpy instead of Tensorflow. We do this for convenience because the assignment

operation of sines and cosines cannot be done in tensorflow. You have to do this in numpy because tensors are not assignable.

```
angle_rads[:, 0::2] = np.sin(angle_rads[:, 0::2])    ## even index
angle_rads[:, 1::2] = np.cos(angle_rads[:, 1::2])    ## odd index
```

As can be seen above, Transformers use a lot of broadcasting throughout. Here is an example of that. Notice that initially `angle_rads` is of size `[40, 512]`. However, because of broadcasting, you need to change it to `[1, 40, 512]`. We do this with `newaxis` function like so

```
angle_rads = angle_rads[np.newaxis, ...]    ## (1, 40, 512) for broadcasting
```

after that, we can perform the sum

```
embeddings      +      pos_encoding      ## broadcasting
[N, 40, 512]    +      [1, 40, 512]    = [N, 40, 512]
```

The output of the positional encoding function is a tensor of size `[N, 40, 512]`. It is recommended that you make the values in the embedding much larger than the values in the positional encoding so position does not dominate.

Another way to view positional encoding is to use “**for**” loops. While this approach is more intuitive, it is not recommended as it does not parallelize well.

```
## an intuitive version of positional encoding using for loops
## not used here but left in for reference

def pos_intuitive(x, d_model=512, max_seq_len = 40):
    # create constant 'pe' matrix with values dependant on pos and i

    pe = np.zeros(max_seq_len, d_model)
    for pos in range(max_seq_len):
        for i in range(0, 512, 2):
            pe[pos, i] = math.sin(pos / (10000 ** ((2 * i)/512)))
            pe[pos, i + 1] = math.cos(pos / (10000 ** ((2 * (i + 1))/512)))

    pe = pe.unsqueeze(0)

    # make embeddings relatively larger
    x = x * tf.sqrt(d_model)

    seq_len = x.size(1)
    x = x + Variable( pe[:, :seq_len] )
    return x
```

## The Encoder Architecture

The encoder has 6 sublayers called encoder layers. Each encoding layer has a multi-head attention layer followed by a standard fully connected feed forward layer. The attention layer consists of 8 parallel Attention sub layers that are later concatenated. The intuition is that each of these 8 layers can learn something new and different. So this gives more capacity to the network. The input to the encoder goes through all these layers in the encoder and is converted into an encoder output. The input to the encoder and the output of the encoder have the same dimensions. For instance, here, the input would be the English sentence.

The next code segment shows the standard encoder layer. Notice that 6 identical encoder layers are created where the outputs of one become the inputs of the next layer. The dimensions of all inputs and output at this stage are the same  $[N, 40, 512]$  where  $N$  is usually 64. That is 64 samples in each batch. The input is the **embed\_en\_pos\_enc\_in** tensor which is a batch of 64 sentences, with 40 tokens per each sentence, and where 512 is each id that has been embedded to a vector of 512. Remember that the embedding vectors of size 512 are learned by the model so initially they are random data.

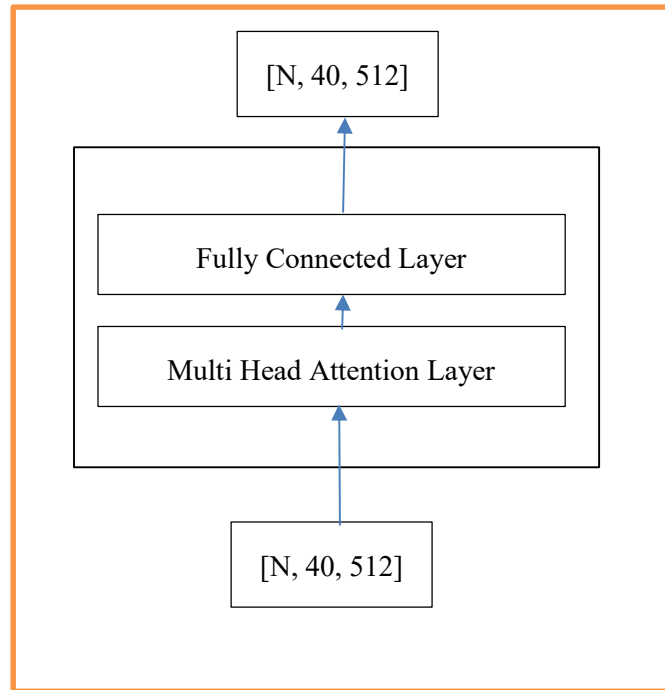
The **enc\_padding\_mask** tensor is the padding mask for the English sentence. Basically we want the Transformer to ignore tokens with 0 value (i.e. padding).

```
## embed_en_pos_enc_in is [N, 40, 512]

def encoder(embed_en_pos_enc_in, enc_padding_mask, dropout):

    with tf.variable_scope("Encoder_1"):
        h1 = encoder_layer(embed_en_pos_enc_in, enc_padding_mask, dropout)
    with tf.variable_scope("Encoder_2"):
        h2 = encoder_layer(h1, enc_padding_mask, dropout)
    with tf.variable_scope("Encoder_3"):
        h3 = encoder_layer(h2, enc_padding_mask, dropout)
    with tf.variable_scope("Encoder_4"):
        h4 = encoder_layer(h3, enc_padding_mask, dropout)
    with tf.variable_scope("Encoder_5"):
        h5 = encoder_layer(h4, enc_padding_mask, dropout)
    with tf.variable_scope("Encoder_6"):
        h6 = encoder_layer(h5, enc_padding_mask, dropout)
    return h6  ## [N, 40, 512]
```

Notice that the encoder is made up of 6 encoder layers.



**Figure.** `encoder_layer` (x6) in the encoder



Each encoder layer has an Attention layer and a fully connected layer.

```
def encoder_layer(x, enc_padding_mask, dropout):

    #####
    ## MultiHead Attention segment

    with tf.variable_scope("Enc_MultiHead_Attention_1"):
        z1 = Enc_MultiHeadAttention(x, enc_padding_mask, dropout)
    with tf.variable_scope("Enc_MultiHead_Attention_2"):
        z2 = Enc_MultiHeadAttention(x, enc_padding_mask, dropout)
    with tf.variable_scope("Enc_MultiHead_Attention_3"):
        z3 = Enc_MultiHeadAttention(x, enc_padding_mask, dropout)
    with tf.variable_scope("Enc_MultiHead_Attention_4"):
        z4 = Enc_MultiHeadAttention(x, enc_padding_mask, dropout)
    with tf.variable_scope("Enc_MultiHead_Attention_5"):
        z5 = Enc_MultiHeadAttention(x, enc_padding_mask, dropout)
    with tf.variable_scope("Enc_MultiHead_Attention_6"):
        z6 = Enc_MultiHeadAttention(x, enc_padding_mask, dropout)
    with tf.variable_scope("Enc_MultiHead_Attention_7"):
        z7 = Enc_MultiHeadAttention(x, enc_padding_mask, dropout)
    with tf.variable_scope("Enc_MultiHead_Attention_8"):
        z8 = Enc_MultiHeadAttention(x, enc_padding_mask, dropout)

    ## [N, 40, 64] after concat it is [N, 40, 64*8] = [N, 40, 512]
    z_concat = tf.concat([z1, z2, z3, z4, z5, z6, z7, z8], -1) ## [N, 40, 512]

    W0 = tf.Variable(      xavier_init( [batch_size, 8*64, 512] ) )
    b = tf.Variable(tf.random_normal([batch_size, 40, 512]))
    z = tf.add( tf.matmul(z_concat, W0), b ) ## [N, 40, 512]

    residual1 = layer_norm(x + z) ## this is [N, 40, 512]

    #####
    ## Feed Forward segment

    h1 = fully_connected_layer(residual1, dropout) ## [N, 40, 512]
    residual2 = layer_norm(residual1 + h1) ## [N, 40, 512]

    return residual2 # [N, 40, 512]
```

In the function **encoder\_layer** below, the input **x** is of size  $[N, 40, 512]$ . So imagine

$[1200 \ 45 \ 23 \ 1201 \ 0 \ 0 \ 0]$

is a sentence except that instead of ids like 45, 23, etc. you have vectors there of size 512. One for each id. The padding mask is simply a  $[N, 40]$  tensor with 0s and 1s like so

$[0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 1]$

The **encoder\_layer** consists of the multi-head attention segment followed by the fully connected layer segment. I used **tf.variable\_scope** to re-use the function: **Enc\_MultiHeadAttention()**

Notice also that **x** and the **padding\_mask** go into the function **Enc\_Multihead\_Attention** 8 times, in parallel, and that the results are concatenated.

The researchers in Vaswani et al. (2017) found that they could reduce the dimensionality of these 8 attention heads to improve performance. As you will see in the function

### **Enc\_MultiHeadAttention**

In the **Enc\_Multihead\_Attention** function, the tensors are reduced from size  $[N, 40, 512]$  to  $[N, 40, 64]$ .

After concatenation, you end up with a tensor of the original size which is

$$[N, 40, 64*8] = [N, 40, 512]$$

The concatenation step is as follows. The value of -1 just indicates that you concatenate on the last dimension of the tensors. So, for each tensor of size  $[N, 40, 64]$ , you only concatenate the last 64. That results in  $64 * 8 = 512$ .

```
z_concat = tf.concat([z1, z2, z3, z4, z5, z6, z7, z8], -1) ## [N, 40, 512]
```

In the following code segment, we map the result in **z\_concat** to a new tensor by doing this (a standard NN layer connection).

```
W0 = tf.Variable(      xavier_init( [batch_size, 8*64, 512] ) )
b = tf.Variable(tf.random_normal([batch_size, 40, 512]))
z = tf.add( tf.matmul(z_concat, W0) , b ) ## [N, 40, 512]
```

and with the following step, we take the current processed tensor “**z**” and add it to the original tensor “**x**”. Remember that **x** has not been processed. This operation is called a residual operation and is done to avoid vanishing gradients or situations where so much processing has happened that the weights are almost zero after a while. This technique of the “residuals” helps models to learn much better.

```
residual1 = layer_norm(x + z) ## this is [N, 40, 512]
```

The result of adding **x** and **z** is normalized and results in a tensor of size  $[N, 40, 512]$ . The **residual1** is the output of the 8 headed monster called Multi-Head Attention. This **residual1** becomes the input to the fully connected layer as can be seen in the following code segment

```

h1 = fully_connected_layer(residual1, dropout) ## [N, 40, 512]
residual2 = layer_norm(residual1 + h1)    ## [N, 40, 512]
return residual2    # [N, 40, 512]

```

the result of the fully connected layer is once again added to the original input and normalized. The output of the encoder is residual2 which is a tensor of size [N, 40, 512]. This will be the input to the decoder and is also called the **encoder\_output**. The intuition here is that you took English sentences and you encoded them with this scheme.

Now, let us explore the inner workings of the encoder multihead attention function. The input to the enc\_multihead attention function is a tensor **x** of size [N, 40, 512]. And an Encoder padding mask of size [N, 40]. Remember that masking does this:

```

if [1200 45 23 1201 0 0 0]
then [ 0 0 0 0 1 1 1]

```

let's break down the code in

### **Enc\_MultiHeadAttention**

The first segment is what is referred to as calculating the keys (K), values (V), and queries (Q). This is what is needed to calculate and use the Attention mechanism. Notice that these are nothing more than tensors that are the result of a matrix multiplication (matmul) between the input **X** ([N, 40, 512]) matrix multiplied by a

respective weight matrix. Notice that the dimension of the resulting Q, K, and V is now 64 and not 512 (remember:  $8 \times 64 = 512$ ). Effectively the 40 tokens per sentence in the batch of 64 are now embeddings of 64 instead of 512 as can be seen here.

```
Wq = tf.Variable(      xavier_init( [batch_size, 512, 64] ) )
bq = tf.Variable( tf.random_normal( [batch_size, 40, 64] ) )
Q = tf.matmul(x, Wq) + bq    ## Nx40x64

Wk = tf.Variable(      xavier_init( [batch_size, 512, 64] ) )
bk = tf.Variable( tf.random_normal( [batch_size, 40, 64] ) )
K = tf.matmul(x, Wk) + bk    ## Nx40x64

Wv = tf.Variable(      xavier_init( [batch_size, 512, 64] ) )
bv = tf.Variable( tf.random_normal( [ batch_size, 40, 64] ) )
V = tf.matmul(x, Wv) + bv    ## Nx40x64
```

The next step is where it gets interesting. In this step, you calculate a score of word\_i's importance to all other words in the input sentence by effectively doing a dot product. That is it! That is the magic of the attention mechanism. The intuition is that at the sentence level, every word in the sentence looks at every other word, for the given problem (e.g. translation), and during the learning process it assigns weights to words that are important given other words. That is why it is called Attention (as in paying attention). The queries (Q) tensor is matrix multiplied by the transpose of the keys (K) tensor. The result **scores\_matrix** is of size [N, 40, 40].

```
scores_matrix = tf.matmul( Q, K, transpose_b=True)
scores_matrix = scores_matrix/(tf.sqrt(64.0))
```

we don't really want the network to pay attention to the padding so we're going to mask it with the Encoder padding mask of size [N, 40]. This is another place where we use broadcasting as can be seen in the following code segment.

```

def Enc_MultiHeadAttention(x, enc_padding_mask, dropout):    ### [N, 40, 512]

    Wq = tf.Variable(      xavier_init( [batch_size, 512, 64] ) )
    bq = tf.Variable( tf.random_normal( [batch_size, 40, 64] ) )
    Q = tf.matmul(x, Wq) + bq    ## Nx40x64

    Wk = tf.Variable(      xavier_init( [batch_size, 512, 64] ) )
    bk = tf.Variable( tf.random_normal( [batch_size, 40, 64] ) )
    K = tf.matmul(x, Wk) + bk    ## Nx40x64

    Wv = tf.Variable(      xavier_init( [batch_size, 512, 64] ) )
    bv = tf.Variable( tf.random_normal( [batch_size, 40, 64] ) )
    V = tf.matmul(x, Wv) + bv    ## Nx40x64

    ## calc a score of word_i importance to all other words
    scores_matrix = tf.matmul( Q, K, transpose_b=True)    ### (N, 40, 40)
    scores_matrix = scores_matrix/(tf.sqrt(64.0))
    ## depth=64

    ## scores matrix is [N, 40, 40]

    #####
    ## we don't really want the network to pay attention to the padding
    ## so we're going to mask it

    ## Encoder padding mask [N, 40]

    ##      [N, 40, 40] + [N, 1, 40]    ## for broadcasting broadcast
    scores_matrix = scores_matrix + (enc_padding_mask[:, tf.newaxis, :] * -1e9)

    #####
    # softmax is normalized on the last axis (seq_len_k) so that the scores
    # add up to 1. ## axis -1 is for last dimension in this tensor
    a1 = tf.nn.softmax(scores_matrix, axis = -1) # (N, seq_len_q, seq_len_k)

    a1 = tf.nn.dropout(a1, dropout)

    a2 = tf.matmul(a1, V)    ## [N, 40, 40] * [N, 40, 64]

    return a2    ## [N, 40, 64]

```

We use **newaxis** to make the rank equal for both tensors and for broadcasting to be possible. We multiply the padding by **-1e9** which is basically negative infinity in python. What this does is that when we run the result tensor through the softmax function, the softmax function will make those infinities close to zero which will allow us to ignore the padding.

```

[N, 40, 40] + [N, 1, 40]    ## for broadcasting
scores_matrix = scores_matrix + (enc_padding_mask[:, tf.newaxis, :] * -1e9)

```

Finally, in the previous code segment the softmax is calculated on the last axis (seq\_len\_k) so that the scores add up to 1. The axis -1 is for the last dimension in this tensor. Notice that the last step is to matrix multiply the values (V) tensor with the scores matrix after running it through the softmax and a dropout layer as can be seen here.

```

a1 = tf.nn.softmax(scores_matrix, axis = -1) # (N, seq_len_q, seq_len_k)
a1 = tf.nn.dropout(a1, dropout)
a2 = tf.matmul(a1, V) ## [N, 40, 40] * [N, 40, 64]
return a2 ## [N, 40, 64]

```

the resulting tensor is of size [N, 40, 64]. Remember that this function is done 8 times. That is why the Attention mechanism is called the 8 headed monster.

That is it for attention! If you understand this, then you understand Attention for all other layers of the transformer.

Congratulations!

The last part of the encoder is the fully connected layer. That is just a simple deep learning layer as you first learned when you started with deep learning.

```
## input [N, 40, 512]

def fully_connected_layer(input, dropout):
    w_h1 = tf.Variable( xavier_init( [batch_size, 512, 2048] ) )
    b_h1 = tf.Variable( tf.random_normal( [batch_size, 40, 2048] ) )

    h1_mul = tf.matmul( input , w_h1 )
    h1 = tf.add( h1_mul, b_h1 )

    h1_relu = tf.nn.relu(h1)
    h1_drop = tf.nn.dropout(h1_relu, dropout)

    w_h2 = tf.Variable( xavier_init( [batch_size, 2048, 512] ) )
    b_h2 = tf.Variable( tf.random_normal( [batch_size, 40, 512] ) )

    ## h1_drop = [N, 40, 2048]
    h2_mul = tf.matmul( h1_drop , w_h2 ) ## [N, 40, 512]
    h2 = tf.add( h2_mul, b_h2 )

    return h2 ## [N, 40, 512]
```

As you can see in the code below, the Feed Forward layer provides for non-linearity using RELU and changes the representation of data to a higher dimension of 2048 before going back down to 512.

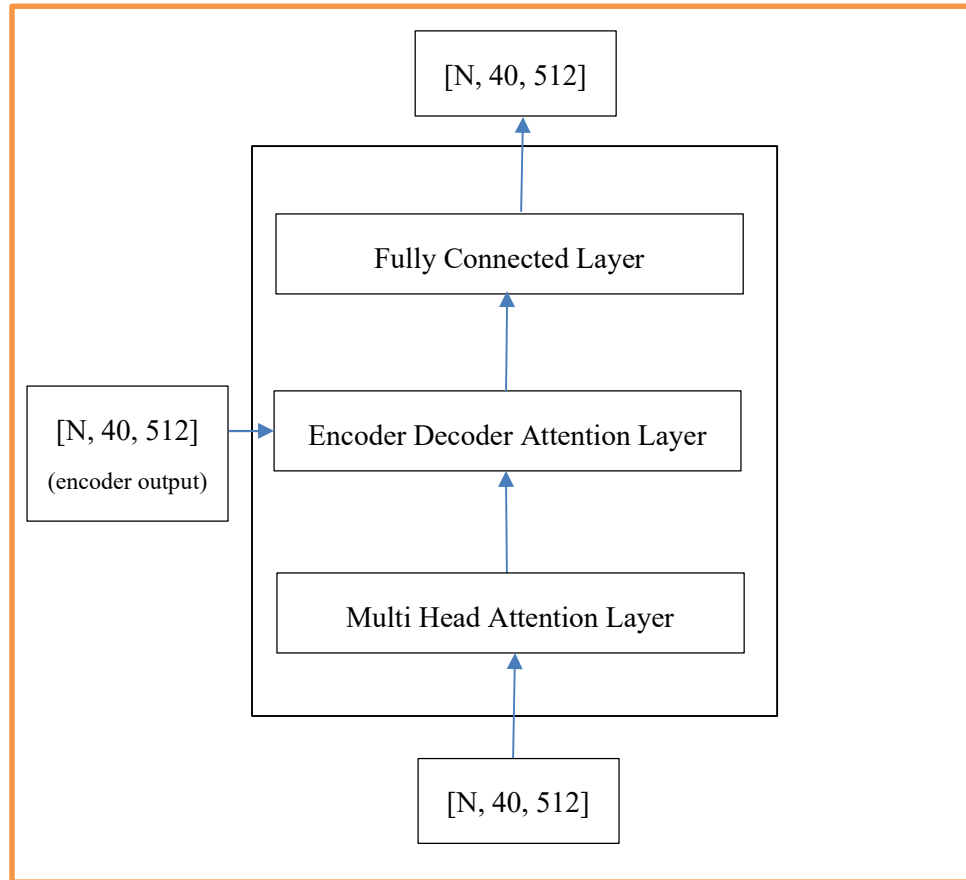
Now that we are done with the encoder, let us move to the decoder. The decoder is similar to the encoder.

### The Decoder Architecture

The decoder layer has 2 inputs. One input is the encoder output. The second input to the decoder varies based on whether you are training or predicting. If you are training, the input to the decoder is the sentence in the other language. For instance,



the Portuguese or Spanish sentence. When training, a mask is needed here to prevent the model from seeing all the words it is trying to predict. This is called a look ahead mask.



**Figure. decoder\_layer (x6) in the decoder**

If you are testing, the input is just the previous words before the word you are trying to predict. You start with a start of sentence token (e.g. < sos >) and predict. The predicted word is then added to the previous tokens and the process is repeated. The decoder consists of 6 decoder layers, followed by a linear layer. Each decoder layer has a decoder multi-head attention layer, followed by a decode-encoder attention layer, and a fully connected layer. The attention layers consist of 8 parallel attention sub layers that are later concatenated. The numbers 6 and 8 are a choice the architect makes.

The first code segment in this section describes the decoder's overall architecture. The decoder has more inputs than the encoder.

They include the following:

- **encoder\_output**: the encoder output is what you get from the encoder. It is the representation of the input English sentence. It is size [N, 40, 512]
- **embed\_pt\_pos\_dec\_in**: this varies depending on whether you are training or testing. If training, then you are using Teacher Forcing and this is the Portuguese sentence of size [N, 40, 512]. If predicting this is still a tensor of the same dimension but it contains the currently predicted sentence tokens since every time you predict, you add the predicted word to the input.
- **enc\_out\_padding\_mask**: this is the padding mask to ignore the padding tokens in the input to the encoder. Size is [N, 40] (e.g. for the English sentence).
- **dec\_look\_ahead\_comb\_mask**: this mask is a bit more complex. It is actually the sum of a padding mask with a look ahead mask. The size is

[N, 40, 40]. See the mask calculations code below for a more detailed explanation.

Notice that in the code below, I use `variable_scope` to re-use the function **`decoder_layer`**. Notice that **`decoder_layer`** takes the 2 masks, the decoder input, and the encoder output as inputs. All of these will be used in the attention mechanism of the decoder.

There will be now in the **`decoder_layer`** function 2 attention mechanisms. Basically, one Attention mechanism is for the decoder input which is the Portuguese sentence and corresponding padding and look ahead masks. The second attention mechanism is for the encoder output and the output from the first attention mechanism (plus corresponding masks). That is it really.

Just like with the encoder there are 6 decoder layers where the outputs of one layer become the inputs of the next layer. Unlike the encoder, the decoder has one last layer as can be seen here

```
## h6 = [N, 40, 512]
dec_out_one_hot = dec_final_linear_layer(h6)
return dec_out_one_hot      ## [N, 40, vocabulary_size]
```

this final layer maps a tensor of size [N, 40, 512] to a tensor of size [N, 40, `pt_vocab_size`] where **`pt_vocab_size`** is the size of the Portuguese vocabulary.

This is what allows us to select the predicted word.

```
## encoder_output = [N, 40, 512]
## embed_pt_pos_dec_in = [N, 40, 512]

def decoder(encoder_output, embed_pt_pos_dec_in,
            enc_out_padding_mask, dec_look_ahead_comb_mask, dropout):

    with tf.variable_scope("Decoder_layer_1"):
        h1 = decoder_layer(embed_pt_pos_dec_in, encoder_output,
                           enc_out_padding_mask, dec_look_ahead_comb_mask, dropout)

    with tf.variable_scope("Decoder_layer_2"):
        h2 = decoder_layer(h1, encoder_output,
                           enc_out_padding_mask, dec_look_ahead_comb_mask, dropout)

    with tf.variable_scope("Decoder_layer_3"):
        h3 = decoder_layer(h2, encoder_output,
                           enc_out_padding_mask, dec_look_ahead_comb_mask, dropout)

    with tf.variable_scope("Decoder_layer_4"):
        h4 = decoder_layer(h3, encoder_output,
                           enc_out_padding_mask, dec_look_ahead_comb_mask, dropout)

    with tf.variable_scope("Decoder_layer_5"):
        h5 = decoder_layer(h4, encoder_output,
                           enc_out_padding_mask, dec_look_ahead_comb_mask, dropout)

    with tf.variable_scope("Decoder_layer_6"):
        h6 = decoder_layer(h5, encoder_output,
                           enc_out_padding_mask, dec_look_ahead_comb_mask, dropout)

    #####
    ## h6 = [N, 40, 512]
    dec_out_one_hot = dec_final_linear_layer(h6)

    return dec_out_one_hot    ## [N, 40, vocabulary_size]
```

Okay, get ready. The **decoder\_layer** function is the busiest function of the Transformer. It is basically very similar to the **encoder\_layer** except that it has 2 attention mechanisms instead of just one. The Multi-Head Attention is the first

attention mechanism. The Portuguese sentence and corresponding padding mask are the only inputs to this sub layer.

The output of this attention mechanism plus the encoder output are the inputs to the second attention mechanism which is usually referred to as the Encoder-Decoder-Attention mechanism. The output of this second Attention mechanism is passed to a fully connected layer just like the one used in the encoder. The first Masked multi-head attention layer is done 8 times in parallel just like in the encoder and the results are concatenated. This concatenated result is added to the original after mapping it through one more layer to calculate the residual. Like so

```
W0 = tf.Variable(      xavier_init( [batch_size, 8*64, 512] ) )
b0 = tf.Variable( tf.random_normal( [batch_size, 40, 512] ) )
z1 = tf.matmul(z_concat, W0) + b0
residual1 = layer_norm(input_dec_layer + z1)
```

Remember that adding **z1** plus the original input is called a “residual” and it helps the model to learn better. The result is normalized to help the model be more stable. The **residual1** and the encoder output become the inputs to the second attention layer in the decoder called encoder-decoder-attention layer. Once again this is an 8 headed parallel operation which results in 8 tensors of size [N, 40, 64]. These are then concatenated, mapped to one more layer (**z2**), normalized, and added to the original value to create another residual. In this case, the tensor **residual2**. Finally, as can be seen here

```
## Feed Forward segment
h1 = fully_connected_layer(residual2, dropout)
residual3 = layer_norm(residual2 + h1)
return residual3      ## [N, 40, 512]
```

we take **residual2** and run it through a fully connected layer like we did on the encoder and we repeat the residual process again. Here is the **decoder\_layer** function in all its glory!

```

## encoder_output = [N, 40, 512]
## input_dec_layer = [N, 40, 512]
## enc_out_padding_mask [N, 40]
## dec_look_ahead_comb_mask [N, 40, 40]

def decoder_layer(input_dec_layer,
                  encoder_output, enc_out_padding_mask, dec_look_ahead_comb_mask, dropout):

    #####
    ## Masked multi-head attention

    with tf.variable_scope("Dec_MultiHead_Attention_1"):
        z1 = Dec_MultiHeadAttention(input_dec_layer, dec_look_ahead_comb_mask, dropout)
    with tf.variable_scope("Dec_MultiHead_Attention_2"):
        z2 = Dec_MultiHeadAttention(input_dec_layer, dec_look_ahead_comb_mask, dropout)
    with tf.variable_scope("Dec_MultiHead_Attention_3"):
        z3 = Dec_MultiHeadAttention(input_dec_layer, dec_look_ahead_comb_mask, dropout)
    with tf.variable_scope("Dec_MultiHead_Attention_4"):
        z4 = Dec_MultiHeadAttention(input_dec_layer, dec_look_ahead_comb_mask, dropout)
    with tf.variable_scope("Dec_MultiHead_Attention_5"):
        z5 = Dec_MultiHeadAttention(input_dec_layer, dec_look_ahead_comb_mask, dropout)
    with tf.variable_scope("Dec_MultiHead_Attention_6"):
        z6 = Dec_MultiHeadAttention(input_dec_layer, dec_look_ahead_comb_mask, dropout)
    with tf.variable_scope("Dec_MultiHead_Attention_7"):
        z7 = Dec_MultiHeadAttention(input_dec_layer, dec_look_ahead_comb_mask, dropout)
    with tf.variable_scope("Dec_MultiHead_Attention_8"):
        z8 = Dec_MultiHeadAttention(input_dec_layer, dec_look_ahead_comb_mask, dropout)

    z_concat = tf.concat([z1, z2, z3, z4, z5, z6, z7, z8], -1)    ## [N, 40, 512]

    W0 = tf.Variable(      xavier_init( [batch_size, 8*64, 512] ) )
    b0 = tf.Variable( tf.random_normal( [batch_size, 40, 512] ) )
    z1 = tf.matmul(z_concat, W0) + b0
    residual1 = layer_norm(input_dec_layer + z1)

    #####
    ## multi head attention with encoder output
    ## this is the decoder_encoder_attention segment

    with tf.variable_scope("En_De_Att_MultiHead_Attention_1"):
        dea1 = encoder_decoder_attention(residual1, encoder_output,
                                         enc_out_padding_mask, dropout)
    with tf.variable_scope("En_De_Att_MultiHead_Attention_2"):

```

```

        dea2 = encoder_decoder_attention(residual1, encoder_output,
                                         enc_out_padding_mask, dropout)
    with tf.variable_scope("En_De_Att_MultiHead_Attention_3"):
        dea3 = encoder_decoder_attention(residual1, encoder_output,
                                         enc_out_padding_mask, dropout)
    with tf.variable_scope("En_De_Att_MultiHead_Attention_4"):
        dea4 = encoder_decoder_attention(residual1, encoder_output,
                                         enc_out_padding_mask, dropout)
    with tf.variable_scope("En_De_Att_MultiHead_Attention_5"):
        dea5 = encoder_decoder_attention(residual1, encoder_output,
                                         enc_out_padding_mask, dropout)
    with tf.variable_scope("En_De_Att_MultiHead_Attention_6"):
        dea6 = encoder_decoder_attention(residual1, encoder_output,
                                         enc_out_padding_mask, dropout)
    with tf.variable_scope("En_De_Att_MultiHead_Attention_7"):
        dea7 = encoder_decoder_attention(residual1, encoder_output,
                                         enc_out_padding_mask, dropout)
    with tf.variable_scope("En_De_Att_MultiHead_Attention_8"):
        dea8 = encoder_decoder_attention(residual1, encoder_output,
                                         enc_out_padding_mask, dropout)

    dea_concat = tf.concat([dea1, dea2, dea3, dea4, dea5, dea6, dea7, dea8], -1)

    W0_dea = tf.Variable(    xavier_init( [batch_size, 8*64, 512] ) )
    b0_dea = tf.Variable( tf.random_normal( [batch_size, 40, 512] ) )
    z2 = tf.matmul(dea_concat, W0_dea) + b0_dea
    residual2 = layer_norm(residual1 + z2)      ## [N, 40, 512]

    #####
    ## Feed Forward segment

    h1 = fully_connected_layer(residual2, dropout)
    residual3 = layer_norm(residual2 + h1)

    return residual3

```

The first decoder attention mechanism is shown in the function

### **Dec\_MultiHeadAttention**

As you can see below, this is exactly like the attention mechanism for the encoder except that it needs to include a look-ahead mask.

```

## x [N, 40, 512]
## look_ahead_mask [N, 40, 40]

def Dec_MultiHeadAttention(x, look_ahead_mask, dropout):

    Wq = tf.Variable( xavier_init( [batch_size, 512, 64] ) )
    bq = tf.Variable( tf.random_normal( [batch_size, 40, 64] ) )
    Q = tf.matmul(x, Wq) + bq # Nx40x64

    Wk = tf.Variable( xavier_init( [batch_size, 512, 64] ) )
    bk = tf.Variable( tf.random_normal( [batch_size, 40, 64] ) )
    K = tf.matmul(x, Wk) + bk # Nx40x64

    Wv = tf.Variable( xavier_init( [batch_size, 512, 64] ) )
    bv = tf.Variable( tf.random_normal( [batch_size, 40, 64] ) )
    V = tf.matmul(x, Wv) + bv # Nx40x64

    ## calc a score of word_i importance to all other words
    scores_matrix = tf.matmul( Q, K, transpose_b=True) ### (N, 40, 40)
    scores_matrix = scores_matrix/(tf.sqrt(64.0)) ## [N, 40, 40]

    #####
    ## look_ahead_mask [N, 40, 40]
    ## [N, 40, 40] + [N, 40, 40]

    scores_matrix = scores_matrix + (look_ahead_mask * -1e9) ## [N, 40, 40]

    #####
    # softmax is normalized on the last axis (seq_len_k) so that the scores
    # add up to 1. ## axis -1 is for last dimension in this tensor

    a1 = tf.nn.softmax(scores_matrix, axis=-1) # (N, seq_len_q, seq_len_k)

    a1 = tf.nn.dropout(a1, dropout)

    a2 = tf.matmul(a1, V) ## [N, 40, 40] * [N, 40, 64]

    return a2 ## [N, 40, 64]

```

During Teacher Forcing training, in the training phase, we feed the Portuguese sentences but whenever we predict a word we are only allowed to look at the words



directly before the work we are trying to predict. This is accomplished with the look ahead mask.

This part of the code in the **Dec\_Multihead\_Attention function**

```
Wq = tf.Variable(      xavier_init( [batch_size, 512, 64] ) ) )
bq = tf.Variable( tf.random_normal( [batch_size, 40, 64] ) ) )
Q = tf.matmul(x, Wq) + bq    # Nx40x64

Wk = tf.Variable(      xavier_init( [batch_size, 512, 64] ) ) )
bk = tf.Variable( tf.random_normal( [batch_size, 40, 64] ) ) )
K = tf.matmul(x, Wk) + bk    # Nx40x64

Wv = tf.Variable(      xavier_init( [batch_size, 512, 64] ) ) )
bv = tf.Variable( tf.random_normal( [batch_size, 40, 64] ) ) )
V = tf.matmul(x, Wv) + bv    # Nx40x64
```

is exactly as we described for the encoder. You calculate the keys, queries, and values which are tensors that map the input **x** of size  $[N, 40, 512]$  to size  $[N, 40, 64]$ . We then calculate the scores matrix which is the Attention mechanism. This is a dot product. We matrix multiply **Q** with the transpose of **K**. This results in a matrix that is size  $[N, 40, 40]$ .

```
## calc a score of word_i importance to all other words
scores_matrix = tf.matmul( Q, K, transpose_b=True)  ### (N, 40, 40)
scores_matrix = scores_matrix/(tf.sqrt(64.0))      ## [N, 40, 40]
```

After calculating the score matrix, we need to mask the values so that we don't cheat by looking ahead. We apply the look ahead and padding masks. The mask for look ahead attention happens before the softmax calculation. Notice that the masking is done to the **dot\_product** scores matrix only. The mask is multiplied with **-1e9** (close to negative infinity).

```

## look_ahead_mask [N, 40, 40]
## [N, 40, 40] + [N, 40, 40]
scores_matrix = scores_matrix + (look_ahead_mask * -1e9)  ## [N, 40, 40]

```

This is done because the mask is summed with the scaled matrix multiplication of Q and K and is applied immediately before a softmax. The goal is to zero out padded cells, and large negative inputs to softmax are near zero in the output.

For example, softmax for “a”

```

>>> a = tf.constant([0.6, 0.2, 0.3, 0.4, 0, 0, 0, 0, 0, 0])
>>> tf.nn.softmax(a)

```

gives the following

```

<tf.Tensor: shape=(10,), dtype=float32, numpy=
array([0.15330984, 0.10276665, 0.11357471, 0.12551947, 0.08413821,
       0.08413821, 0.08413821, 0.08413821, 0.08413821, 0.08413821],
      dtype=float32)>

```

now, if some of the values are negative infinities

```

>>> b = tf.constant([0.6, 0.2, 0.3, 0.4, -1e9, -1e9, -1e9, -1e9, -1e9, -1e9])
>>> tf.nn.softmax(b)

```

then softmax gives us

```

<tf.Tensor: shape=(10,), dtype=float32, numpy=
array([ 0.3096101 , 0.20753784, 0.22936477, 0.25348732,  0.    ,  0.    ,  0.    ,  0.    ,  0.    ,  0.    ],
      dtype=float32)>

```

Notice the infinities are now zeros! At this point, just like with the **encoder\_multihead\_attention** function, the **decoder\_multihead\_attention** function takes the **scores\_matrix** after adding the mask and applies the softmax. The softmax is normalized on the last axis (**seq\_len\_k**) so that the scores add up to 1. The value of axis = -1 is for the last dimension in this tensor.

```
a1 = tf.nn.softmax(scores_matrix, axis=-1) # (N, seq_len_q, seq_len_k)
a1 = tf.nn.dropout(a1, dropout)
a2 = tf.matmul(a1, V) ## [N, 40, 40] * [N, 40, 64]
return a2 ## [N, 40, 64]
```

Finally, just like before, dropout is applied and the result is multiplied with the matrix V. The final tensor is of size [N, 40, 64].

Remember that the previous function

#### **Dec\_MultiHeadAttention**

is done 8 times in parallel with the same input and that the outputs of these 8 attention layers are concatenated together. The concatenated output becomes the input to the second Attention layer which is called **encoder\_decoder\_attention**.

Let us continue, the next step in the decoder layer is to take the outputs from the the **dec\_multihead\_attention** layers and process it through another Attention layer. This layer is called the **encoder\_decoder\_attention** layer. And as its name implies this layer has something to do with the encoder. If you remember, we have the **encoder\_output** which we haven't used yet. As it turns out this is where the **encoder\_output** is inserted to the decoder.

Therefore, the **encoder\_decoder\_attention** layer has 2 inputs which are:

- the output from the first **dec\_multihead\_attention** layer which is of size [N, 40, 512].
- The output from the encoder which we call **encoder\_output** and which is also of size [N, 40, 512]

The second decoder attention mechanism (**decoder\_encoder\_attention**), as you may imagine, will be very similar to all our previous discussions of attention mechanisms. The only difference is that we now have 2 inputs to an attention layer and we need to account for that.

Just like before we calculate the queries, keys, and values but notice the difference this time in the following code segment

```
Wq = tf.Variable(      xavier_init( [batch_size, 512, 64] ) )
bq = tf.Variable( tf.random_normal( [batch_size, 40, 64] ) )
Q = tf.matmul(input, Wq) + bq  # Nx40x64 ## from decoder_attention layer below

Wk = tf.Variable(      xavier_init( [batch_size, 512, 64] ) )
bk = tf.Variable( tf.random_normal( [batch_size, 40, 64] ) )
K = tf.matmul(encoder_output, Wk) + bk  ## from encoder output [N, 40, 64]

Wv = tf.Variable(      xavier_init( [batch_size, 512, 64] ) )
bv = tf.Variable( tf.random_normal( [batch_size, 40, 64] ) )
V = tf.matmul(encoder_output, Wv) + bv  ## from encoder output [N, 40, 64]
```

The full code is below

```

## input      [N, 40, 512]
## encoder_output [N, 40, 512]
## mask      [N, 40]

def encoder_decoder_attention(input, encoder_output, mask, dropout):

    Wq = tf.Variable(      xavier_init( [batch_size, 512, 64] ) )
    bq = tf.Variable( tf.random_normal( [batch_size, 40, 64] ) )
    Q = tf.matmul(input, Wq) + bq  # Nx40x64  ## from decoder_attention layer below

    Wk = tf.Variable(      xavier_init( [batch_size, 512, 64] ) )
    bk = tf.Variable( tf.random_normal( [batch_size, 40, 64] ) )
    K = tf.matmul(encoder_output, Wk) + bk  ## from encoder output [N, 40, 64]

    Wv = tf.Variable(      xavier_init( [batch_size, 512, 64] ) )
    bv = tf.Variable( tf.random_normal( [batch_size, 40, 64] ) )
    V = tf.matmul(encoder_output, Wv) + bv  ## from encoder output [N, 40, 64]

    ## calc a score of word_i importance to all other words
    ## Q = [N, 40, 64], K=[N, 40, 64], transpose(K) = [N, 64, 40]
    ## Q * transpose(K) = [N, 40, 64] * [N, 64, 40] = [N, 40, 40]

    scores_matrix = tf.matmul(Q, K, transpose_b=True)  ### [N, 40, 40]
    scores_matrix = scores_matrix/( tf.cast( tf.sqrt(64.0), tf.float32 ) )

    ## mask look ahead attention happens before the softmax
    ## masking done to the dot_product matrix only
    ## The mask is multiplied with -1e9 (close to negative infinity). This is done because
    ## the mask is summed with the scaled matrix multiplication of Q and K and is applied
    ## immediately before a softmax. The goal is to zero out these cells, and large negative
    ## inputs to softmax are near zero in the output.
    ## mask = mask.unsqueeze(1) -- remove dimensions with value 1
    ## mask [N, 40]
    ## [N, 40, 40] + [N, 1, 40]  ## for broadcast
    scores_matrix = scores_matrix + (mask[:, tf.newaxis, :] * -1e9)

    # softmax is normalized on the last axis (seq_len_k) so that the scores
    # add up to 1. ## axis -1 is for last dimension in this tensor
    a1 = tf.nn.softmax(scores_matrix, axis=-1) # (N, seq_len_q, seq_len_k)

    a1 = tf.nn.dropout(a1, dropout)
    # (N, seq_len_q, depth_v) ## scores_matrix * V
    a2 = tf.matmul(a1, V)  ## [N, 40, 40] * [N, 40, 64] = [N, 40, 64]

    return a2  ## [N, 40, 64]

```

The calculation is the same. The only thing that changes now is the inputs. Notice that the Q tensor is matrix multiplied with the output from the previous multi-head attention layer or the actual input in the case of the first decoder layer.

Whereas, the K and V tensors are multiplied by the encoder output.

Then, as before, we calculate the scores matrix

$$Q * \text{transpose}(K) = [N, 40, 64] * [N, 64, 40] = [N, 40, 40]$$

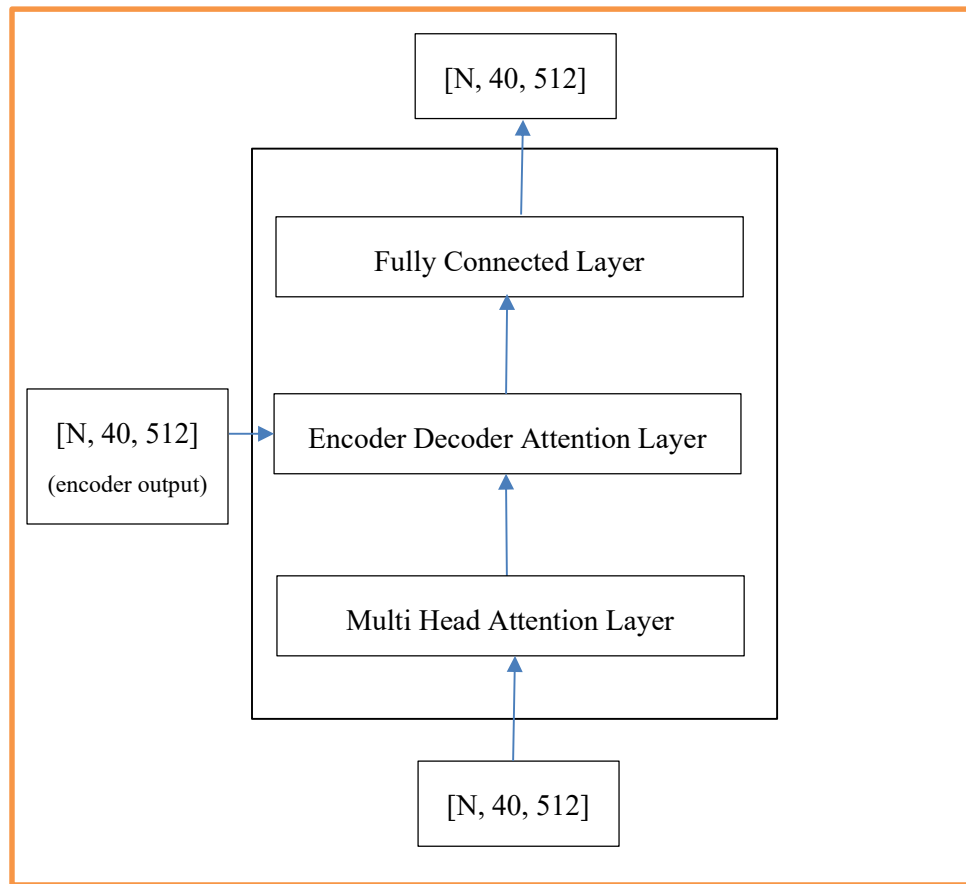
Which gives us a tensor of size [N, 40, 40]. From here the process is exactly the same as all other attention layers.

So just to recap, each **decoder\_layer** function has 3 sub layers which are

- Decoder multihead attention
- Encoder-Decoder attention layer
- And the fully connected layer

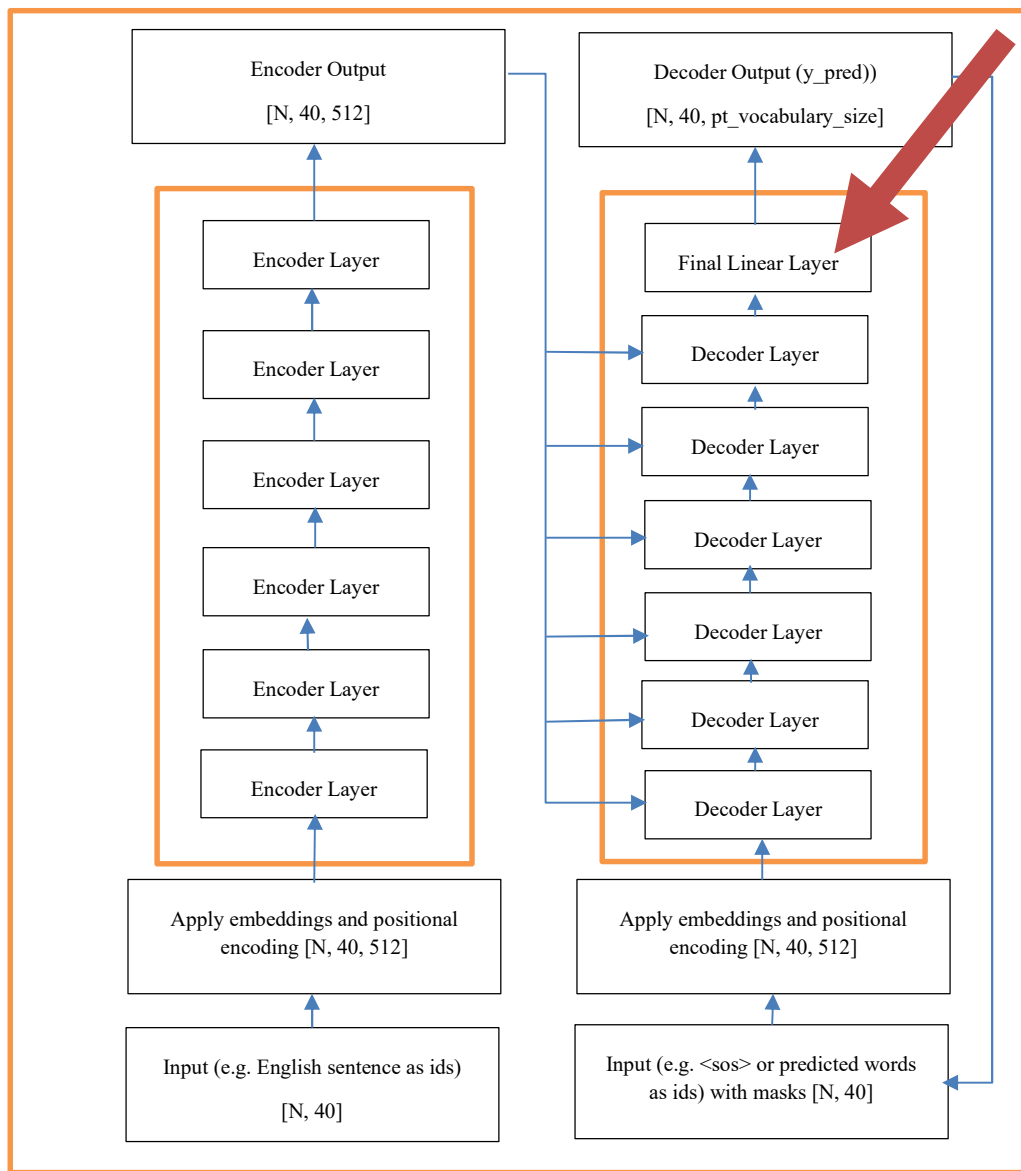
After the 2 attention layers, each decoder layer has a fully connected layer with RELU activation. This is exactly as we described for the encoder layer.

As can be seen in the figure.



**Figure. decoder\_layer (x6) in the decoder**

And that completes the description of the attention layers in the **decoder\_layer** functions of the decoder. If you remember, (see figure below), there is one more layer in the decoder that we have not discussed



**Figure. Transformer (and inputs during testing)**



The decoder has a final linear layer after the 6 decoder\_layer functions. We proceed to discuss it in the next section.

### Decoder Final Linear Layer

The final layer in the decoder is the **decoder\_final\_layer**. This is a linear layer with no non-linearities and a softmax that maps the tensor [N, 40, 512] to a tensor of size [N, 40, pt\_vocab\_size] as can be seen in the next code segment.

```
## input = [N, 40, 512]

def dec_final_linear_layer(input):
    w_h1 = tf.Variable(      xavier_init(      [batch_size, 512, VOCAB_SIZE_PT] ) )
    b_h1 = tf.Variable(      tf.random_normal( [batch_size, 40, VOCAB_SIZE_PT] ) )

    h1_mul = tf.matmul( input , w_h1 )
    h1 = tf.add( h1_mul, b_h1 )

    softmax_h1 = tf.nn.softmax( h1 , axis=-1 ) ## [N, 40, vocabulary_size]
    dec_out_one_hot = softmax_h1      ## [N, 40, vocabulary_size]

    #####
    ## if you wanted the ids, you could do this

    dec_out_ids = tf.argmax( softmax_h1 , axis=-1)
    dec_out_ids = tf.cast(dec_out_ids, tf.int32)

    #####

    ## you could return
    ## dec_out_ids      or      dec_out_one_hot
    ##      [N, 40]          [N, 40, vocabulary_size]
    ## because of the loss function used (sparse_cross_entropy)
    ## dec_out_one_hot seems to be the correct one

    return dec_out_one_hot      ## [N, 40, vocabulary_size]
```

And that concludes the encoder and decoder layers. Now we are ready to discuss some of the additional utility functions and the process of training, losses, and the prediction function.

The following function is an example of how to normalize your data. Data normalization in deep learning is essential to have more stable models.

```
## x [N, 40, 512]

def layer_norm(x):
    eps = 1e-6
    size = 512
    # create two learnable parameters to calibrate normalization
    alpha = np.ones(size)
    bias = np.zeros(size)
    mean, var = tf.nn.moments(x, axes=[0, 1, 2])
    standard_dev = tf.sqrt(var)
    norm = alpha * (x - mean) / (standard_dev + eps) + bias
    return norm    ## [N, 40, 512]
```

Throughout this chapter, masks have been mentioned several times. The masks used in the Transformer were one of the most difficult things for me to understand. It took me considerable time to finally understand how they work. In the next section, we will discuss the code for masks.

## Masks

There are only 2 types of masks that you need here. They are:

- The padding mask
- The look ahead mask

Masks, as it turns out, are a simple mechanism; you add them to your scores matrix to mask away padding or terms you should not be looking at when predicting words. Let's start with the look ahead mask. The look-ahead mask is used to mask the future tokens in a sequence. In other words, the mask indicates which entries should not be used. This means that to predict the third word, only the first and second words will be used. Similarly to predict the fourth word, only the first, second, and the third word will be used and so on. Remember that unlike RNNs, Transformers do not rely on sequence in the network, per se. Instead, sequence is encoded as a feature with positional encoding. However, when predicting a term, the Transformer can see the encoder input all at once (i.e. the English sentence), and during training, could potentially see the entire decoder input (i.e. the Portuguese sentence). Obviously, if we show it the whole portuguese sentence, then the transformer will just copy the decoder input as its output. Instead, we want to predict the next Portuguese **word** given all previous Portuguese words and the entire English sentence. During training, to block out some of the future terms or the current term we want to predict, we use the look ahead mask. Programatically, we just create a square matrix with size equal to the max length of the sentence including padding. For example

```
x = tf.random.uniform( (1, 3) )    ## [the cat is]
look_ahead_mask = create_look_ahead_mask(x.shape[1])  ## 3 (seq_len)
```

The mask will look like this for a sequence length of 3

```
<tf.Tensor: shape=(3, 3), dtype=float32, numpy=
array([[0., 1., 1.],
       [0., 0., 1.],
       [0., 0., 0.]], dtype=float32)>
```

This simple square matrix makes sure that we cannot look ahead . The code for the look ahead mask is as follows:

```
def create_look_ahead_mask(seq_len):
    ones_tensor = np.ones( (seq_len, seq_len) )    ## 40x40
    mask = np.triu(ones_tensor, k=1)  ## k=1 means above diagonal
    mask = mask.astype('uint8')
    return mask    # (seq_len, seq_len)  ## [40,40]
```

Now that we have discussed the look ahead mask, let's move on to the padding mask. Let us consider an example. For an input sequence like this one

[1200 45 23 1201 0 0 0]

Your padding mask looks like this

[ 0 0 0 0 1 1 1]

You simply place 1s on the 0 padded positions. The following code shows how to do this.

```
## source_seq    [N, 40]

def create_padding_mask(source_seq):

    ## put 1 where padded, elementwise
    padding_mask = tf.cast(tf.equal(source_seq, 0), dtype=tf.float32)

    return padding_mask    ##    [N, 40]
```

And now we can create one single function that will create all the masks we will need for the Transformer. The inputs to this function are `enc_in [N, 40]` which is the English sentence and `dec_in [N, 40]` which is the portuguese sentence. This function calculates 3 padding masks and 1 look ahead mask. The last padding mask and look ahead mask are combined for convenience. Therefore, this function returns 3 masks.

First we calculate the encoder padding mask. Again, the idea is to have something like this.

Encoder\_in padding mask for one english sentence

```
if [1200 45 23 1201 0 0 0]
then [ 0 0 0 0 1 1 1]
```

We then calculate the padding mask for the encoder output. Since we will use this in the decoder, we will need to mask it for padding to avoid paying attention to zero values. The encoder output and this mask will be used in the **decoder\_encoder\_attention** block in the decoder. This padding mask is used to mask the **encoder\_outputs**. The `enc_in` tensor can be used here for convenience because the **enc\_out** has the same dimensions and padding as **enc\_in**.

Finally, the third mask is a combination of a padding mask and a **look\_ahead\_mask**. The decoder is the only place where we need a look ahead. The input to the very first layer of the decoder is the Portuguese sentence. As we are predicting words in this sentence during training via Teacher Forcing, we need to mask future terms so that a word being predicted can only use the terms that

have appeared before it. The look ahead mask is a simple square matrix of size sequence length (e.g. 40) where the upper triangle above the diagonal is all 1s. So, to recap, we need to calculate the padding mask for the decoder in (the Portuguese sentence), a look ahead mask, and combined them. The look ahead mask is used to mask future tokens in the dec input received by the decoder.

Notice that attention **dot\_product** matrices in the attention layers are 40x40 for out current set of parameters and the look ahead mask is also [40, 40]. So, the dimensions match. To combine the 2 masks we use **tf.maximum()** which is a function that returns the maximum values elementwise between 2 matrices. The function **tf.maximum** supports broadcasting and so we can use **newdimension** to make sure the tensors have the same rank. The mask **dec\_in\_padding\_mask** of size [N, 40] is converted to size [N, 1, 40]. The **look\_ahead\_mask** [40, 40] is changed to size [1, 40, 40]. We need to broadcast so we can add these new dimensions as can be seen below.

```
dec_combined_mask = tf.maximum(
    dec_in_padding_mask[:, tf.newaxis, :], look_ahead_mask[tf.newaxis, ...])
                                [N, 1, 40]                [1, 40, 40]
```

```

## x_ph_enc_in      [N, 40]
## y_ph_dec_in      [N, 40]

def create_masks(enc_in, dec_in):

    enc_in_padding_mask = create_padding_mask(enc_in)

    dec_enc_out_padding_mask = create_padding_mask(enc_in)

    ## decoder_in padding mask - portuguese sentence
    ## padding mask for decoder_in
    dec_in_padding_mask = create_padding_mask(dec_in)

    look_ahead_mask = create_look_ahead_mask( dec_in.shape[1] ) ## [40, 40]

    ## returns the maximum elementwise
    ## tf.maximum supports broadcast
    ## dec_in_padding_mask      [N, 40]
    ## look_ahead_mask          [40, 40]
    ## need to broadcast so add new dimensions (see book for another example of how this works)
    ##                                [N, 1, 40]                                [1, 40, 40]

    dec_combined_mask = tf.maximum(
        dec_in_padding_mask[:, tf.newaxis, :], look_ahead_mask[tf.newaxis, ...])

    ##      [N, 40]      [N, 40, 40]      [N, 40]
    return enc_in_padding_mask, dec_combined_mask, dec_enc_out_padding_mask

```

Finally, the **create\_masks** function returns the masks:

- enc\_in\_padding\_mask
- dec\_combined\_mask
- dec\_enc\_out\_padding\_mask

Now we are ready to proceed to define core functions, place holders and the main loop. As usual, I will define the inference function, the loss, and the training function together. The inference function will be called **inference\_transformer** and it is the place where you bring all the Transformer parts together. So, in essence, it is the transformer itself. We define the Transformer architecture here. The inputs are **x\_ph\_enc\_in** which is the English sentence of size [N, 40] and **y\_ph\_dec\_in** which is the Portuguese sentence of size [N, 40]. The first step is to create the masks as previously described. Then we proceed to perform the embeddings. As previously mentioned, the inputs

$$\begin{array}{ll} \mathbf{x\_ph\_enc\_in} & [N, 40] \\ \mathbf{y\_ph\_dec\_in} & [N, 40] \end{array}$$

currently have dimension [N, 40]. We will change that to make each id into a vector of dimensionality 512 by using the Tensorflow lookup function. This embedding is done in the following code segment:

```
embeddings_en = tf.Variable( tf.random_uniform( [VOCAB_SIZE_EN, 512], -1.0, 1.0) )
embed_en_enc_in = tf.nn.embedding_lookup(embeddings_en, x_ph_enc_in)
## token embeddings are multiplied by a scaling factor which is square root of depth size
embed_en_enc_in = embed_en_enc_in * tf.sqrt( tf.cast(512, tf.float32) )

embeddings_pt = tf.Variable( tf.random_uniform( [VOCAB_SIZE_PT, 512], -1.0, 1.0) )
embed_pt_dec_in = tf.nn.embedding_lookup(embeddings_pt, y_ph_dec_in)
## token embeddings are multiplied by a scaling factor which is square root of depth size
embed_pt_dec_in = embed_pt_dec_in * tf.sqrt( tf.cast(512, tf.float32) )
```



This embedding layer maps from ids which have a size equal to vocabulary size to 512. Embedding vectors of size 512 are initially random and will be learned via backpropagation. The token embeddings are multiplied by a scaling factor which is the square root of the depth size like so

```
embed_en_enc_in = embed_en_enc_in * tf.sqrt( tf.cast(512, tf.float32) )
```

These embeddings are later added to another tensor called positional encoding which holds the positional information. The scaling factor allows the embeddings data to not be dominated by the positional data after the sum.

After mapping the data to the embeddings, we can proceed to add the positional data. We do this with the following code

```
embed_en_pos_enc_in = positional_encoding( embed_en_enc_in , dropout ) ## [N, 40, 512]  
embed_pt_pos_dec_in = positional_encoding( embed_pt_dec_in , dropout ) ## [N, 40, 512]
```

the **positional\_encoding** was previously defined, but if you remember, in that function both tensors have the same size of [N, 40 ,512] and so it is an easy sum.

After doing all these transformations (hence the name?), we are ready to enter the Transformer network. As we can see in this code segment

```

                                [N, 40, 512]    [N, 40]
encoder_output = encoder(embed_en_pos_enc_in, enc_padding_mask, dropout)

                                [N, 40, 512]    [N, 40, 512]
y = decoder(encoder_output, embed_pt_pos_dec_in,
                                enc_out_padding_mask, dec_look_ahead_comb_mask, dropout)
                                [N, 40]          [N, 40, 40]

return y    ## [N, 40, vocabulary_size]

```

the flow is fairly simple, inputs go into the encoder. The encoder converts those inputs into an encoder output. This encoder output plus some decoder inputs go into the decoder. And the decoder will produce a decoder output of size

[N, 40, pt\_vocab\_size]

```

## define the Transformer architecture here
## x_ph_enc_in    [N, 40]
## y_ph_dec_in    [N, 40]

def inference_transformer(x_ph_enc_in, y_ph_dec_in, dropout):

    #####
    ## masks
    ## [N, 40]          [N, 40, 40]          [N, 40]
    enc_padding_mask, dec_look_ahead_comb_mask, enc_out_padding_mask
    = create_masks(x_ph_enc_in, y_ph_dec_in)

    #####
    ## embedding layer from vocab size to 512. Embeddings are initially random
    ## and are learned by backpropagation

    embeddings_en = tf.Variable( tf.random_uniform( [VOCAB_SIZE_EN, 512], -1.0, 1.0) )
    embed_en_enc_in = tf.nn.embedding_lookup(embeddings_en, x_ph_enc_in)
    ## token embeddings are multiplied by a scaling factor which is square root of depth size
    embed_en_enc_in = embed_en_enc_in * tf.sqrt( tf.cast(512, tf.float32) )

    embeddings_pt = tf.Variable( tf.random_uniform( [VOCAB_SIZE_PT, 512], -1.0, 1.0) )
    embed_pt_dec_in = tf.nn.embedding_lookup(embeddings_pt, y_ph_dec_in)
    ## token embeddings are multiplied by a scaling factor which is square root of depth size
    embed_pt_dec_in = embed_pt_dec_in * tf.sqrt( tf.cast(512, tf.float32) )

    #####
    ## positional encoding

    embed_en_pos_enc_in = positional_encoding( embed_en_enc_in , dropout ) ## [N, 40, 512]
    embed_pt_pos_dec_in = positional_encoding( embed_pt_dec_in , dropout ) ## [N, 40, 512]

    #####
    ## now enter the transformer network
    ## [N, 40, 512]          [N, 40]
    encoder_output = encoder(embed_en_pos_enc_in, enc_padding_mask, dropout)

    ## [N, 40, 512]          [N, 40, 512]
    y = decoder(encoder_output, embed_pt_pos_dec_in,
                enc_out_padding_mask, dec_look_ahead_comb_mask, dropout)
    ## [N, 40]          [N, 40 , 40]

    return y    ## [N, 40, vocabulary_size]

```

After defining the inference function, we can proceed to define the loss function. The loss is similar to other cross entropy based loss functions except that we want to use a mask and that the **y\_pred** and **y\_real** tensors have different dimensions. We mask the loss incurred by the padded tokens to 0 so that they do not contribute to the mean loss. Therefore, we want to ignore padding when calculating the loss function. For this loss, **y\_ph\_dec\_real** is of size [N, 40] and **y\_pred** is one hot encoded of size [N, 40, vocab\_size], which is big. Conveniently, Tensorflow has the loss function

**tf.nn.sparse\_softmax\_cross\_entropy\_with\_logits**

which by definition takes tensors with different dimensions as our inputs.

For example,

```
tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y_ph_dec_real, logits=y_)
```

Here the loss takes **y\_ph\_dec\_real** as [N, 40] and **y\_pred** as [N, 40, vocab\_size]. We also mask the **y\_real** data to avoid the padding. This is done through this code segment

```
## y_ph_dec_real [N, 40]
## tf.equal(y_ph_dec_real, 0) -->> ([0 0 0 1 1 1])
## then reverse it with tf.math.logical_not -->> ## [1 1 1 0 0 0]

mask = tf.math.logical_not( tf.equal(y_ph_dec_real, 0) )
mask = tf.cast(mask, tf.float32)    ## [N, 40]
```

and that is it! The full loss can be seen below.

```
def loss(y_pred, y_ph_dec_real):

    y_ = label_smoothing( y_pred )

    ## y_ph_dec_real [N, 40]
    ## tf.equal(y_ph_dec_real, 0) -->> ([0 0 0 1 1 1])
    ## then reverse it with tf.math.logical_not -->> ## [1 1 1 0 0 0]
    mask = tf.math.logical_not( tf.equal(y_ph_dec_real, 0) )
    mask = tf.cast(mask, tf.float32)    ## [N, 40]

    ## labels [N, 40]
    ## tf.nn.sparse_softmax_cross_entropy_with_logits returns:
    ## Returns: A Tensor of the same shape as labels
    ## and of the same type as logits with the softmax cross entropy loss
    loss_ = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y_ph_dec_real, logits=y_)

    ## loss_ ## [N, 40]
    ## mask  ## [N, 40]

    loss_ = loss_ * mask          ## element wise
    return tf.reduce_sum(loss_)/tf.reduce_sum(mask)
```

Once the loss is defined, we can specify the training function which uses the Adam optimizer. In their paper, Vaswani et al. (2017) recommended to use a certain set of parameters for Adam as indicated in the function below.

```
def training(cost):
    optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate,
                                       beta1=0.9, beta2=0.98, epsilon=1e-9)
    train_op = optimizer.minimize(cost)
    return train_op
```

Wow! At this point we are almost done with the transformer. Now we just need to define the inputs and how to feed them into the Transformer through the main loop.

Let's keep going.

In the next code segment we proceed to define the placeholders. Initially for me this was difficult to understand. I did not quite get these placeholders. In deep learning, you are used to 2 placeholders for the input and output. But if you look at the code below, I have 3! So what is going on? The best way to understand this is to think of the translation problem. In translation, we have, for example, an English sentence and a Portuguese sentence. The English sentence is simple. That is the input to the encoder (**x\_ph\_enc\_in**). What about the Portuguese sentence? The Portuguese sentence is what you want to predict. Therefore, it should be the data (**y\_ph\_dec\_real**) we feed to the loss function to be compared with **y\_pred** (**y\_ph\_dec\_real**). So far everything is normal. But what about **y\_ph\_dec\_in**? What is it for? The placeholder **y\_ph\_dec\_in** is the decoder input and it is a bit more complex. The simplest explanation is that the decoder input is also the Portuguese sentence. So **y\_ph\_dec\_in** and **y\_ph\_dec\_real** are both the Portuguese sentence but on one sentence the words are all shifted to the left.

This was confusing to me at first. The Portuguese sentence or the batch (of 64 Portuguese sentences), call it **batch\_pt**, is divided into **batch\_pt\_inp** and **batch\_pt\_real** for the decoder. The tensor **batch\_pt\_inp** is passed as an input to the decoder. The **batch\_pt\_real** is that same input shifted by 1 and only used by the loss function. It is compared to **y\_pred**. At each location "i" in each sentence in **batch\_pt\_inp**, the tensor **batch\_pt\_real** contains the next token that should be predicted.

For example, for a sentence which has already been padded

```
batch_pt = "SOS A lion in the jungle is sleeping EOS 0 0"
```

we can split it into the two tensors **batch\_pt\_inp** and **batch\_pt\_real** as seen below. Notice that the sentences are the same except that one (**batch\_pt\_real**) is shifted to the left. The shifting allows you to align words so that you can predict the next word in the sequence.

```
batch_pt_inp = "SOS A lion in the jungle is sleeping EOS 0"
batch_pt_real = "A lion in the jungle is sleeping EOS 0 0"
```

In this case, given this English sentence we would feed the model the SOS token and the model should predict A, given SOS A, then the model should predict lion, and so on. This can be better visualized like this:

Given		Predict
SOS	-- >>	A
SOS A	-- >>	lion
SOS A lion	-- >>	in
SOS A lion in	-- >>	the

And so on.

During the testing phase, the predicted value is concatenated to the previously predicted values. And the first token given is <sos>.

During training we use Teacher Forcing which means we do not use predicted values but we actually use the real values. Using predicted values during training, especially when the model has not learned much, would mean that you would be teaching the model a lot of errors. The look ahead mask is critical here because it prevents looking ahead.

Finally, notice that the data type of all these placeholders is integer because these placeholder hold the ids or padding.

```
## sentences + padding = 40
x_ph_enc_in  = tf.placeholder( tf.int32, [None, 40] )  ## english enc in
y_ph_dec_in  = tf.placeholder( tf.int32, [None, 40] )  ## portuguese dec in
y_ph_dec_real = tf.placeholder( tf.int32, [None, 40] )  ## portuguese dec out
```

One more placeholder can be declared for dropout.

```
dropout_ph = tf.placeholder( tf.float32 )      ## dropout
```

After defining the placeholders, we can proceed to call the core functions. We call the transformer which predicts **y\_pred** (**y**). The value of **y** contains the one hot encoded id after the softmax. The predicted **y** value is fed to the loss function along with **y\_real** which is the actual portuguese sentence. The **y** tensor is size  $[N, 40, \text{pt\_vocab\_size}]$  and the **y\_ph\_dec\_real** is of size  $[N, 40]$ . This is not a problem because, as previously described, the loss function we are using is designed for this exact scenario. After we define the loss, we can call the Adam optimizer through the **training()** function.



```
y = inference_transformer(x_ph_enc_in, y_ph_dec_in, dropout_ph)
cost = loss(y, y_ph_dec_real)
train_op = training(cost)
```

At this point we are done with all architecture in the computational graph and we can move to call the session and train the model. So let us proceed.

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

After initializing the data and session let us grab our data and store it in **X\_en** [N, 40] and **X\_pt** [N, 41] for convenience.

```
## changing to shorter more common notation
## these are now numpy arrays with padding

X_en = english_sentence_ids_list      ## [number_of_samples, 40]
X_pt = portuguese_sentence_ids_list   ## [number_of_samples, 41]

print("X_en.shape ", X_en.shape)
print("X_pt.shape ", X_pt.shape)
```

Here we calculate the number of batches to load into our GPU.

```
#batch size is 64

num_samples = X_en.shape[0]
print(num_samples)
num_batches = int(num_samples/batch_size)
```

And finally we have arrived at the Main Loop for training. In the main loop, after calculating the batch offsets, we can slice the data by doing the following

```
batch_en = X_en[sta:end, :]      ## [N, 40]
batch_pt = X_pt[sta:end, :]      ## [N, 41]
```

This will result in batches with rows like these of sequences of ids for the corresponding words in the sentence.

```
batch_en  -> the cat is 0 0 -> [12110 12 34 ... 56 12111 0 0 0]
batch_pt  -> el gato es 0 0 -> [12210 6 54 ... 23 23 12211 0 0 0]
```

After getting the batches, we can perform shifting

```
## batch_pt [N, 41]
batch_pt_inp = batch_pt[:, :-1]      ## [N, 40]
batch_pt_real = batch_pt[:, 1:]      ## [N, 40]
```

This shifting results in the following alignment. Notice that this is only done for the Portuguese sentences (i.e. the sentences fed into the decoder). The target (**batch\_pt**) is divided into **batch\_pt\_inp** and **batch\_pt\_real** for the decoder. The **batch\_pt\_inp** is passed as an input to the decoder. The **batch\_pt\_real** is that same

input shifted by 1 and only used by the loss function. It is compared to **y\_pred**. At each location “i” in each sentence in **batch\_pt\_inp**, **batch\_pt\_real** contains the next token that should be predicted. For example,

```
batch_pt  = "SOS  A  lion  in   the  jungle  is   sleeping  EOS  0  0"
```

after the shifting becomes

```
batch_pt_inp = "SOS  A  lion  in   the  jungle  is   sleeping  EOS  0"
batch_pt_real = " A  lion  in   the  jungle  is   sleeping  EOS   0  0"
```

what matters is that given "jungle", the transformer should predict "is", given "is" the transformer should predict "sleeping", and so on. Finally, we call the session and pass the data through **feed\_dict** as can be seen here

```
train_step = sess.run( train_op , feed_dict={ x_ph_enc_in: batch_en,
                                              y_ph_dec_in: batch_pt_inp,
                                              y_ph_dec_real: batch_pt_real,
                                              dropout_ph: dropout_rate
                                              })
```

And that is it for training. The entire code segment can be seen below

```

for i in range(n_epochs):
    for batch_n in range(num_batches):

        sta = batch_n * batch_size
        end = sta + batch_size

        print("current epoch is ", i)
        print("num batches ", num_batches)
        print("batch n ", batch_n)

        ## batches with rows like this of sequence ids for words in the sentence
        ## batch_en -> the cat is 0 0 -> [12110 12 34 ... 56 12111 0 0 0]
        ## batch_pt -> el gato es 0 0 -> [12210 6 54 ... 23 23 12211 0 0 0]

        batch_en = X_en[sta:end, :]          ## [N, 40]
        batch_pt = X_pt[sta:end, :]          ## [N, 41]

        #####
        ## shift - shifting happens after padding
        ## shift batch_pt_real_labels to the left
        ## decoder_in -> el gato
        ## decoder_out_labels -> gato es

        ## batch_pt [N, 41]
        batch_pt_inp = batch_pt[:, :-1]      ## [N, 40]
        batch_pt_real = batch_pt[:, 1:]      ## [N, 40]

        #####
        ## pass the data and train

        train_step = sess.run( train_op , feed_dict={ x_ph_enc_in: batch_en,
                                                         y_ph_dec_in: batch_pt_inp,
                                                         y_ph_dec_real: batch_pt_real,
                                                         dropout_ph: dropout_rate
                                                         })

```

After training the model, we are ready to begin testing it. For the evaluation we need to encode the input sentence (english) using the english tokenizer (**tokenizer\_en**). We also add the start and end tokens so the input is equivalent to what the model is trained with. This is the encoder input.

The **decoder\_input** is the start token in the portuguese tokenizer. The decoder then outputs the predictions by looking at the **encoder\_output** and its own self-attention of all previously predicted words before the one being predicted. After predicting a word, the model concatenates it to the decoder input to predict the next word in the sequence. In this approach, the decoder predicts the next word based on the previous words it has predicted. The process should look like this

```
sent_en  -> <sos> the cat is <eos> -> [12110 12 34 ... 56 12111 ]
sent_pt  -> <sos>                        -> [12210 ]
```

To keep track of the currently predicted word, in the evaluation function, we can use an index. As we are predicting each decoder output tensor, we can use the index to select the word we need given the current iteration. For example

```
iteration 0 (index=0)
enc_in [sos the cat eos 0 ]
dec_in [sos 0 0 0 0 ]
y_pred [el * * * * ]
index -> 0

iteration 1 (index=1)
enc_in [sos the cat eos 0 ]
dec_in [sos el 0 0 0 ]
y_pred [el gato * * * ]
index -> 1

iteration 2 (index=2)
enc_in [sos the cat eos 0 ]
dec_in [sos el gato 0 0 ]
y_pred [el gato eos * * ]
index -> 2
```

In the evaluation “for” loop, we need to take the data and pad it to size 40. After padding, the data will look like this

```
enc_in_pad [batch_size, 40] ##axis=1 ->> [12110, 7, 15, 47, 12111, 0, 0, ..., 0]
dec_in_pad [batch_size, 40] ##axis=1 ->> [12220, 0, 0, 0,      0, 0, 0, ..., 0]
```

After padding, we can feed the data to the computational graph to predict the next word with the following code

```
y_pred = sess.run( y , feed_dict={x_ph_enc_in: enc_in_pad,
                                   y_ph_dec_in: dec_in_pad,
                                   dropout_ph: dropout_rate  })
```

the predicted tensor is of size

```
y_pred = [batch_size, 40, pt_vocab_size]
```

our batch of 64 can contain 64 translations or just 1. So we can select the one we want by doing something like this

```
prediction = y_pred[0, index, :]
```

this tensor has size [1, 1, pt\_vocab\_size], at this point we can also increment the index for the next iteration.

```
index = index + 1
```

now we are ready to get the id of the currently predicted word by using `agmax`

```
predicted_id_tf = tf.cast(tf.argmax(prediction, axis=-1), tf.int32)
```

To evaluate and get an integer we can call the session

```
predicted_id = sess.run(predicted_id_tf)
```

Once we have the id for the current iteration we can proceed to check if this is the end token. If it is, we stop with

```
if predicted_id == pt_END_TOKEN_id:  
    break
```

if we don't stop, then we concatenate the currently predicted id to our decoder input and repeat the process like so

```
## concatenate predicted_id to output then give to decoder as input  
pt_sentence_ids = np.concatenate( [ pt_sentence_ids, [predicted_id] ] )
```

once we hit the end of sentence token, we return **pt\_sentence\_ids** which will have the translated sentence. The full **evaluate()** function is shown below.

```

def evaluate(sentence):

    en_sentence_ids = encode(sentence, en_dictionary)
    en_sentence_ids = np.array(en_sentence_ids)

    en_START_TOKEN_id = en_dictionary['<sos>']
    en_END_TOKEN_id = en_dictionary['<eos>']

    pt_START_TOKEN_id = pt_dictionary['<sos>']
    pt_END_TOKEN_id = pt_dictionary['<eos>']

    en_sentence_ids = np.concatenate(
        [[en_START_TOKEN_id], en_sentence_ids, [en_END_TOKEN_id]] )

    pt_sentence_ids = np.concatenate( [ [pt_START_TOKEN_id] ] )

    ## currently the data looks like this
    ## [12110, 7, 15, 47, 12111]
    ## [12220]

    #####

    index = 0  ## keep track of the predicted id we want

    for i in range(MAX_LENGTH):

        en_sentence_ids_list = []
        pt_sentence_ids_list = []

        if len(en_sentence_ids) <= MAX_LENGTH and
           len( pt_sentence_ids ) <= MAX_LENGTH:
            ## this "for" loop is a cheat to have tensor of size batch with same input
            for nn in range(batch_size):
                en_sentence_ids_list.append( en_sentence_ids )
                pt_sentence_ids_list.append( pt_sentence_ids )

            ## [batch_size, 40]
            enc_in_pad = tf.keras.preprocessing.sequence.pad_sequences(
                en_sentence_ids_list, maxlen=MAX_LENGTH, padding='post')

            ## [batch_size, 40]
            dec_in_pad = tf.keras.preprocessing.sequence.pad_sequences(
                pt_sentence_ids_list, maxlen=MAX_LENGTH, padding='post')

            ## enc in pad [batch size, 40]  ##axis=1 ->> [12110, 7, 15, 47, 12111, 0, 0, ..., 0]

```



```

## dec_in_pad [batch_size, 40] ##axis=1 ->> [12220, 0, 0, 0, 0, 0, 0, ..., 0]

#####

## shifting not needed here in test/prediction

#####
## pass the data and evaluate

y_pred = sess.run( y , feed_dict={x_ph_enc_in: enc_in_pad,
                                   y_ph_dec_in: dec_in_pad,
                                   dropout_ph: dropout_rate })

## y_pred = [batch_size, 40, pt_vocab_size]

## all in batch are the same sentence so I just need the first one
prediction = y_pred[0, index, :] # [1, 1, pt_vocab_size]

index = index + 1

## get id of current predicted word
predicted_id_tf = tf.cast(tf.argmax(prediction, axis=-1), tf.int32)

predicted_id = sess.run(predicted_id_tf)

print("predicted_id ", predicted_id)

## return the result if the predicted_id is equal to the end token
if predicted_id == pt_END_TOKEN_id:
    break

## concatenate predicted_id to output then give to decoder as input
## dec_in = tf.concat( [dec_in, predicted_id] )

pt_sentence_ids = np.concatenate( [ pt_sentence_ids, [predicted_id] ] )

print("sentence being predited")
print(pt_sentence_ids)

return pt_sentence_ids

```

The last piece of the puzzle is to call the evaluate function. Give it the input data and then obtain the predictions list. The predictions list contains a list of ids so this must be converted back to the text. We do this with the **decode** function.

```
def predict(sentence):  
    prediction = evaluate(sentence)  
    print(prediction.shape)  
    prediction_list = prediction.tolist()  
    pt_sentence_words = decode(prediction_list, pt_reverse_dictionary)  
    print(pt_sentence_words)
```

I know I say this a lot but know I mean it. That is it!

```
sentence = "the cat is sleeping"  
predict(sentence)
```

This completed our discussion on the Encoder Decoder with Multi-Head Attention proposed by Vaswani et al. (2017). While I used fixed values for certain parameters, you can change these and experiment with your own model and data.

## 10.2 Summary

In this chapter, I have introduced the topic of Transformers. I discussed the main ideas and code for the Encoder Decoder with Multi-Head Attention Transformer first introduced by Vaswani et al. (2017).

## CHAPTER 11: CONCLUSIONS AND FINAL THOUGHTS

In this book I have only begun to scratch the surface on deep learning algorithms. I hope that these examples and discussions helped you to improve your deep learning coding skills and furthered your interest in machine learning in general. There are more deep learning methodologies that you may want to pursue as well. The Tensorflow website at [www.tensorflow.org](http://www.tensorflow.org) may be a good starting point to continue your studies.

In this final chapter, I want to address a few loose ends relevant to Tensorflow and I will present a few closing thoughts.

### 11.1 Benchmarking Tensorflow

In this section I want to address benchmarking. Tensorflow was made to be used with GPUs and CPUs. If you want to test the performance of your processors, one way to do it is with the following code. In the following code you are performing a matrix multiplication using

```
tf.matmul(a, b)
```

The important aspect is that you can select the device to use. For instance, the following

```
with tf.device('/cpu:0')
```

tells Tensorflow to use the CPU. In contrast, using

```
with tf.device('/gpu:0')
```

would tell Tensorflow to perform the computations using the GPU.

```

import Tensorflow as tf
with tf.device('/cpu:0'):
    a = tf.zeros(shape=[10000,1000], dtype=tf.float32)
    b = tf.zeros(shape=[1000,1000], dtype=tf.float32)
    c = tf.matmul(a, b)
# Creates session with log_device_placement set to True
sess = tf.Session(config=tf.ConfigProto(log_device_placement=True))

# Runs the op.
for i in range(200):
    print i
    print sess.run(c)

```

## 11.2 Conclusions

Since 2007, computational power has certainly improved and today machine learning can take advantage of these more powerful processors to process large amounts of data. In the following table we can see that there are many types of processors. Some are old and traditional and some are new and still experimental. The most widely used processor before 2007 was the CPU. Now, GPUs are the most exciting and promising because they allow deep neural networks to learn the model parameters in very short periods of time.

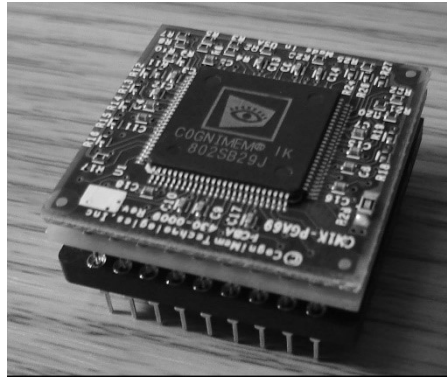
The future may bring even more types of processors which will further improve machine learning and deep neural networks. Currently, several companies are starting to develop their own neural processors. Google, for instance, has developed the Tensor Processing Unit (TPU). This processing unit accelerates deep learning calculations on their servers.

Other neural or cognitive processors have been around since the 80's. One example of such processors is the CM1K seen in the image below.

**Table. Processor Architectures**

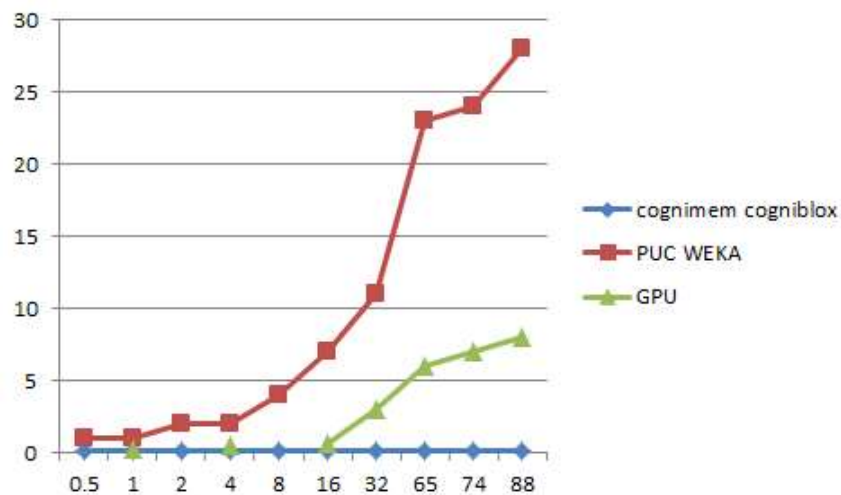
Run	Speed	Cores	Neurons	Power Consumption
CM1K	27 MHz		1024	0.5 Watts
CM2K Elegan	75 MHz		2048	1.6 Watts
Spikey [26]			384	6 Watts
IBM True North			1000000	0.07 Watts
Qualcomm Zeroth				N/A
Intel Chip [27]				N/A
NVidia GPU Titan X	1000 MHz	3072		250 Watts
GeForce GPU GTX 980	1126 MHz	2048		165 Watts
Intel CPU i7-4720HQ	2.6 GHz	4		47 Watts
AMD FX8350 CPU	4 GHz	8		125 Watts
Samsung Exynos 5433	1900 MHz	8		4 Watts
Qualcomm SD805	2700 MHz	4		5 Watts

The CM1K is a neural processor developed by IBM in the 1980's. Its architecture consists of 1024 neurons that can be used to implement simple algorithms like the k-nearest neighbor (KNN) algorithm or the Restricted Coulomb Energy Algorithm. They are very energy efficient but have some limitations in the number of samples they can support.



**Figure. CM1K-PGA69 machine learning chip.**

Finally, the important aspect, especially with Tensorflow is that the coding is and should be independent of the processor. So, the same code written for a CPU in Tensorflow should also work for a GPU and the only change required might be for a parameter that indicates if the code should be run on the GPU or the CPU.



**Figure. GPU, CPU, and cognitive processor speed**

As shown in the graph above, dedicated neural network processors could eventually be much faster than CPUs and GPUs. In the graph above, the trend line for the cognimem cogniblox was obtained using a cognitive processor (the CM1K), the PUC weka line was obtained using a CPU, and the GPU trend line was obtained with a GeForce GPU. The x axis represents the number of test samples used and the y axis represents the amount of time (processing in seconds) it took to process the samples while performing machine learning classification. As can be seen in the graph, the cognitive processor does best, followed by the GPU and then the CPU.

### **11.3 Summary**

This chapter addressed some loose ends related to Tensorflow or deep learning in general. It also presented some concluding remarks about computer processors and their importance for the future of machine learning and deep learning.

## REFERENCES

- Artstein, R., Poesio, M. (2008). Inter-Coder Agreement for Computational Linguistics. *Computational Linguistics*, Volume 34, Issue 4, pp. 555-596.
- Bird, S., Klein, E., Loper, E. (2009). *Natural Language Processing with Python*. 1st ed., O'Reilly Media.
- Buduma, N., Locascio, N. (2016). *Fundamentals of Deep Learning: Designing Next-Generation Machine Intelligence*. O'Reilly
- Burges, C. J. (1998). A Tutorial on Support Vector Machines for Pattern Recognition. *Data Mining and Knowledge Discovery*, vol. 2, pp. 121-167.
- Calix, R., Knapp, G. (2011). Affect Corpus 2.0: An Extension of a Corpus for Actor Level Emotion Magnitude Detection. In *Proceedings of the 2nd ACM Multimedia Systems (MMSys) conference*, San Jose, California, U.S.A, pp. 129-132.
- Chang, C.-C., Lin, C. (2001). LIBSVM: a library for support vector machines. Retrieved from <http://www.csie.ntu.edu.tw/~cjlin/libsvm>.
- Cortes, C., Vapnik, V. (1995). Support-Vector Networks. *Machine Learning*, vol. 20, pp. 273-297.
- Gonzales, R., Woods, R. (2007). *Digital Image Processing*. Pearson Publishing



- Hahnloser, R., Sarpeshkar, R., Mahowald, M., Douglas, R., Seung, H. (2000). Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. *Nature*. 405. pp. 947–951.
- Jurafsky, D., Martin, J. (2008). *Speech and Language Processing*, 2nd ed., New Jersey: Prentice Hall.
- Kutz, J. Nathan. (2013). *Data-Driven Modeling & Scientific Computation: Methods for Complex Systems & Big Data*. Oxford Publishing
- McCarthy, P., Watanabe, S., Lamkin, T. (2012). *The Gramulator: A Tool to Identify Differential Linguistic Features of Correlative Text Types*, IGI-Global
- Mitchell, Ryan. (2015). *Web Scraping with Python*. O'Reilly Publishing
- Mikolov, T. , Sutskever, I., Chen, K., Corrado, G., Dean, J. (2013). Distributed representations of words and phrases and their compositionality. *Advances in Neural Information Processing Systems*.
- Raschka, Sebastian. (2015). *Python Machine Learning*. PACKT Publishing.
- Witten, I., Frank, E. (2005). *Data Mining: Practical Machine Learning Tools and Techniques*. 2d. edition, Morgan Kaufmann Publishers Inc., San Francisco.
- Hitaj, Briland & Gasti, Paolo & Ateniese, Giuseppe & Perez-Cruz, Fernando. (2017). *PassGAN: A Deep Learning Approach for Password Guessing*. arXiv:1709.00440

- Haichao Shi, Jing Dong, Wei Wang, Yinlong Qian, Xiaoyu Zhang, SSGAN: Secure Steganography Based on Generative Adversarial Networks, 2017, arXiv:1707.01613
- Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, Yoshua Bengio, Generative Adversarial Networks, 2014, arXiv:1406.2661
- Ling, Y., An, Y., Liu, M., Hasan, S., Fan, Y., and Hu, X. (2017). Integrating extra knowledge into word embedding models for biomedical NLP tasks, 2017 International Joint Conference on Neural Networks (IJCNN), Anchorage, AK, 2017, pp. 968-975.
- Bojanowski, P., Grave, E., Joulin, A., Mikolov, T. (2017). Enriching Word Vectors with Subword Information. TACL, Association for Computational Linguistics (ACL 2017)
- Garten, J., Sagae, K., Ustun, V., Dehghani, M. (2015). Combining Distributed Vector Representations for Words, In Proceedings of NAACL-HLT 2015, Association for Computational Linguistics, 2015.
- Goodfellow, I., Bengio, Y., Courville, A. (2016). Deep Learning. MIT Press. 2016
- Havasi, C., Speer, R., Alonso, J. (2007). ConceptNet 3: a flexible, multilingual semantic network for common sense knowledge. 22nd Conference on Artificial Intelligence, 2007.
- Ji, S., Yun, H., Yanardag, P., Matsushima, S., and Vishwanathan, S. (2015). Wordrank: Learning word embeddings via robust ranking. arXiv preprint arXiv:1506.02761.

- Li, M., Lu, Q., Long, Y., Gui, L. (2017). Inferring Affective Meanings of Words from Word Embedding, *Journal of IEEE Transactions on Affective Computing*, 2017
- Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, Illia Polosukhin. (2017). Attention is all you need. *NIPS'17: Proceedings of the 31st International Conference on Neural Information Processing Systems* December 2017 Pages 6000–6010
- Devlin, J., Chang, M.-W., Lee, K. & Toutanova, K. (2018). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. [arxiv:1810.04805](https://arxiv.org/abs/1810.04805)
- Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. (2019). Language Models are Unsupervised Multitask Learners.
- Aggarwal, Charu C. (2020). *Linear Algebra and Optimization for Machine Learning: A Textbook*. Springer. 1st ed. 2020
- Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dhharshan Kumaran, Daan Wierstra, Shane Legg & Demis Hassabis, Human-level control through deep reinforcement learning, *Nature* volume 518, pages 529–533 (26 February 2015)

## APPENDIX A: FULL DEEP LEARNING CODE

All code examples used in this book can be downloaded from:

<https://github.com/rcalix1/Deep-learning-ML-and-tensorflow>

## APPENDIX B: USEFUL TENSORFLOW FUNCTIONS

This appendix will provide examples of useful Tensorflow functions that are used throughout the book and that you may need to write your own deep learning algorithms.

---

---

### **tf.one\_hot()**

Tensorflow does provide a function for one-hot encoding which is:

`tf.one_hot()`

This function takes a **y** vector and converts it to the one-hot encoded version.

For example:

```
>>>indices = [0, 1, 2]
>>>depth = 3
>>>print tf.one_hot(indices, depth)

## the output is

[ [ 1, 0, 0],
  [ 0, 1, 0],
  [ 0, 0, 1] ]
```

Now, try this one and see what it gives you.

```
>>> y = tf.one_hot(indices, depth)
>>> depth = 4
>>>indices = [0, 3]
>>>print sess.run(y)
```

---

---

### **tf.reduce\_mean()**

The function `tf.reduce_mean()` is a built in Tensorflow function that takes as input a tensor and computes the mean of the elements across the dimensions of a given tensor. So, it returns a reduced tensor. For example, given:

```
x = tf.constant( [[1, 1] , [2, 2]] )
```

we can get the following:

```
all = tf.reduce_mean(x)    =>    # 1.5
```

```
all = tf.reduce_mean(x, 0) =>    # [1.5, 1.5]
```

```
all = tf.reduce_mean(x, 1) =>    # [1, 2]
```

To see the results, we can run:

```
print sess.run(all)
```

---

---

### **tf.reduce\_sum()**

The function `tf.reduce_sum()` computes the sum of the elements across dimensions of a tensor. For example:

```
>>>x = tf.constant( [[1, 1, 1], [1, 1, 1]])
>>>y = tf.reduce_sum(x)  #6
>>>y = tf.reduce_sum(x, 0)  #[2, 2, 2]
>>>y = tf.reduce_sum(x, 1)  #[3, 3]
```

---

---

### **tf.argmax()**

The function `tf.argmax()` is a Tensorflow function that returns the index of the largest value across the axis of a tensor.

For example,

```
answer = tf.argmax([35, 4, 72, 2])    =>    2
```

you can also define an axis like so

```
answer = tf.argmax( [ [23, 32, 49],  
                      [45,  1, 12] ] )
```

The previous function returns => [2, 0]

---

---

### **tf.equal()**

The `tf.equal()` function returns a vector of boolean values that compares the values in two tensors of equal dimensions.

For example, given:

```
x = [1, 2, 3]
```

```
y = [0, 1, 3]
```

```
tf.equal(x, y) ==> [False, False, True] or [0, 0, 1]
```

---

---

### **tf.reshape()**

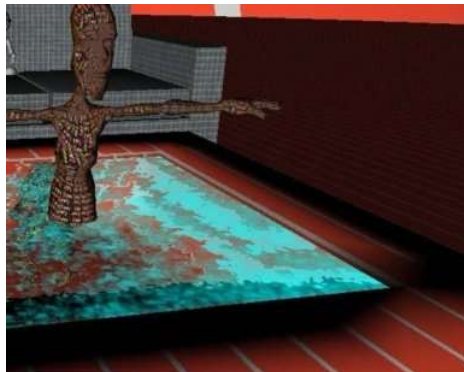
```
# tensor 't' is [1, 2, 3, 4, 5, 6, 7, 8, 9]
# tensor 't' has shape [9]
tf.reshape(t, [3, 3]) ==> [[1, 2, 3],
                             [4, 5, 6],
                             [7, 8, 9]]
```



## **VITA**

Ricardo A. Calix was born in La Ceiba, Honduras, on June 11<sup>th</sup>, 1978. He received the B.S. degree in industrial and systems engineering from Universidad Tecnologica Centroamericana, Tegucigalpa, Honduras, in 2001. He received an M.B.A and M.S. in engineering science from Louisiana State University (LSU), Baton Rouge, in 2006 and 2010, respectively. He was a graduate assistant in the department of industrial engineering at LSU from 2007 to 2011, where he conducted research in natural language processing. He obtained his Ph.D. degree in engineering science with a research focus on machine learning from Louisiana State University, Baton Rouge, in 2011. His research interests include natural language processing, deep learning, and reinforcement learning. He is currently an associate professor of computer information technology at Purdue University Northwest.

## Other books at galacticbackwater.com



Automated Semantic Understanding of  
Human Emotions in Writing and Speech

**Ricardo Calix**

### **Automated Semantic Understanding of Human Emotions in Writing and Speech**

Affective Human Computer Interaction (A-HCI) will be critical for the success of new technologies that will be prevalent in the 21st century. If cell phones and the internet are any indication, there will be continued rapid development of automated assistive systems that help humans to live better, more productive lives. These will not be just passive systems such as cell phones, but active assistive systems like robot aides in use in hospitals, homes, entertainment room, office, and other work environments. Such systems will need to be able to properly deduce human emotional state before they determine how to best interact with people. This work explores and extends the body of knowledge related to Affective HCI. New semantic methodologies are studied for reliable and accurate detection of human emotional states and magnitudes in written and spoken speech; and for mapping emotional states and magnitudes to 3-D facial expression outputs. This is a dissertation on affective human computer interaction.

**Available from Amazon.com**



**Galactic Backwater Media ©**  
**[www.galacticbackwater.com](http://www.galacticbackwater.com)**