Table of Contents

Preface

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`

Shows commands or other text that should be typed literally by the user.

`Constant width italic`

Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/oreillymedia/title_title*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Book Title* by Some Author (O'Reilly). Copyright 2012 Some Copyright Holder, 978-0-596-xxxx-x."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Safari

Safari (formerly Safari Books Online) is a membership-based training and reference platform for enterprise, government, educators, and individuals.

Members have access to thousands of books, training videos, Learning Paths, interactive tutorials, and curated playlists from over 250 publishers, including O'Reilly Media, Harvard Business Review, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Adobe, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, and Course Technology, among others.

For more information, please visit *http://oreilly.com/safari*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://www.oreilly.com/catalog/<catalog page>*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

CHAPTER 1
Introduction to Deep Learning

Deep Learning has revolutionized the technology industry. Modern machine translation, search engines, and computer assistants are all powered by deep-learning. This trend will only continue as deep-learning expands its reach into robotics, pharmaceuticals, energy, and all other fields of contemporary technology. It is rapidly becoming essential for the modern software professional to develop a working knowledge of the principles of deep-learning.

This book will provide an introduction to the fundamentals of machine learning through Tensorflow. Tensorflow is Google's new software library for deep-learning. Tensorflow makes it straightforward for engineers to design and deploy sophisticated deep-learning architectures. Readers of "Deep Learning with Tensorflow" will learn how to use Tensorflow to build systems capable of detecting objects in images, understanding human speech, analyzing video and predicting the properties of potential medicines. Furthermore, readers will gain an intuitive understanding of Tensorflow's potential as a system for performing tensor calclus and will be able to learn how to use Tensorflow for tasks outside the traditional purview of machine learning.

Furthermore, "Deep Learning with Tensorflow" is one of the first deep-learning books written for practitioners. It teaches fundamental concepts through practical examples and builds understanding of machine-learning foundations from the ground up. The target audience for this book is practicing developers, who are comfortable with designing software systems, but not necessarily with creating learning systems. Readers should hold basic familiarity with basic linear algebra and calculus. We will review the necessary fundamentals, but readers may need to consult additional references to get details. We also anticipate that our book will prove useful for scientists and other professionals who are comfortable with scripting, but not necessarily with designing learning algorithms.

In the remainder of this chapter, we will introduce readers to the history of deep-learning, and to the broader impact deep learning has had on the research and commercial communities. We will next cover some of the most famous applications of deep-learning. This will include both prominent machine learning architectures and fundamental deep learning primitives. We will end by giving a brief persepctive of where deep learning is heading over the next few years before we dive into Tensorflow in the next few chapters.

Machine Learning eats Computer Science

Until recently, software engineers went to school to learn a number of basic algorithms (graph search, sorting, database queries and so on). After school, these engineers would go out into real world to apply these algorithms to systems. Most of today's digital economy is built on intricate chains of basic algorithms laboriously glued together by generations of engineers. Most of these systems are not capable of **adapting**. All configurations and reconfigurations have to be performed by highly trained engineers, rendering systems brittle.

Machine learning promises to change broadly the field of software development by enabling systems to adapt dynamically. Deployed machine learning systems are capable of learning desired behaviors from databases of examples. Furthermore, such systems can be regularly retrained as new data comes in. Very sophisticated software systems, powered by machine learning, are capable of dramatically changing their behavior without needed major changes to their code (just to their training data). This trend is only likely to accelerate as machine learning tools and deployment become easier and easier.

As the behavior of software engineered systems change, the roles of software engineers will change as well. In some ways, this transformation will be analogous to the transformation following the development of programming languages. The first computers were painstakingly programmed. Networks of wires were connected and interconnected. Then punchcards were set-up to enable the creation of new programs without hardware changes to computers. Following the punchcard era, the first assembly languages were created. Then higher level languages like Fortran or Lisp. Succeeding layers of development have created very high level languages like Python, with intricate ecosystems of pre-coded algorithms. Much modern computer science even relies on autogenerated code. Modern app developers use tools like Android Studio to autogenerate much of the code they'd like to make. Each successive wave of simplification has broadened the scope of computer science by lowering barriers to entry.

Machine learning promises to further and continue this wave of transformations. Systems built on spoken language and natural language understanding such as Alexa and Siri will allow non-programmers to perform complex computations. Furthermore, ML powered systems are likely to become more **robust** against errors. The capacity to retrain models will mean that codebases can shrink and that maintainability will increase. In short, machine learning is likely to completely upend the role of software engineers. Today's programmers will need to understand how machine learning systems learn, and will need to understand the classes of errors that arise in common machine learning systems. Furthermore, they will need to understand the design patterns that underly machine learning systems (very different in style and form from classical software design patterns). And, they will need to know enough tensor calculus to understand why a sophisticated deep architecture may be misbehaving during learning. It's not an understatement to say that understanding of machine learning (theory and practice) will become a fundamental skill that every computer scientist and software engineer will need to understand for the coming decade.

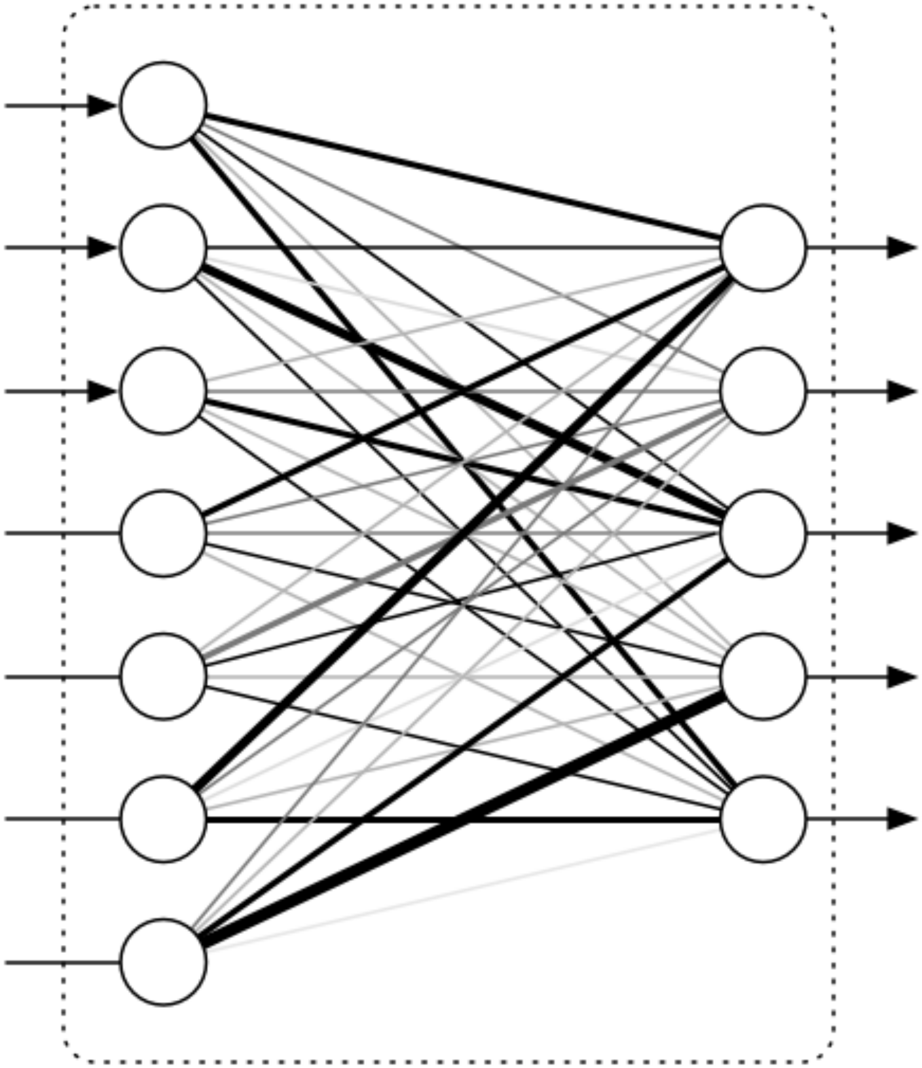
In the remainder of this chapter, we will provide a whirlwind tour of the basics of modern deep learning. The remainder of this book will go into much greater depth on all the topics we touch on today.

Deep Learning Primitives

Most deep architectures are built by combining and recombining a limited set of architectural primitives (neural network layers). In this section, we will provide a brief overview of the common modules which are found in many deep networks.

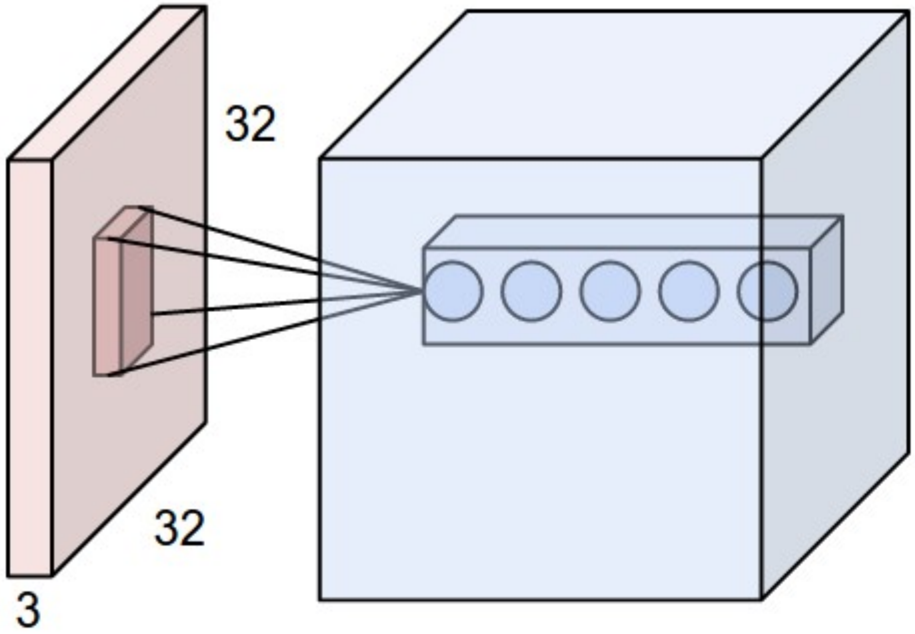
Fully Connected Layer

A fully connected network transforms a list of inputs into a list of outputs. The transformation is called fully connected since any input value can affect any output value. These layers will have many learnable parameters, even for relatively small inputs, but they have the large advantage that they assume no structure in the inputs.



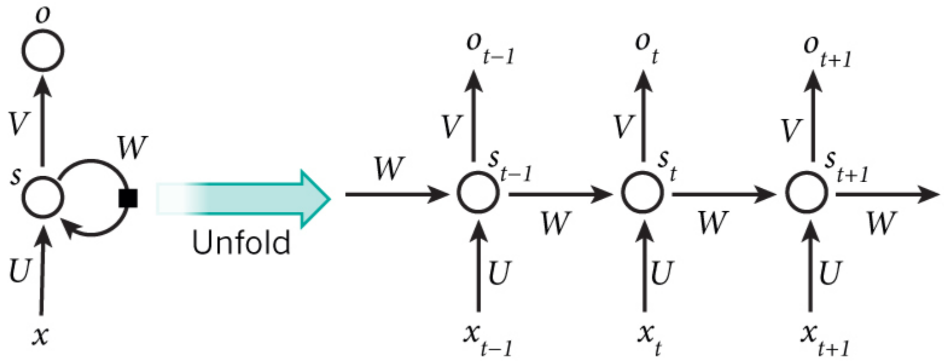
Convolutional Layer

A convolutional network assumes special spatial structure in its input. In particular, it assumes that inputs that are close to each other in the original input are semantically related. This assumption makes most sense for images, which is one reason convolutional layers have found wide use in deep architectures for image processing.



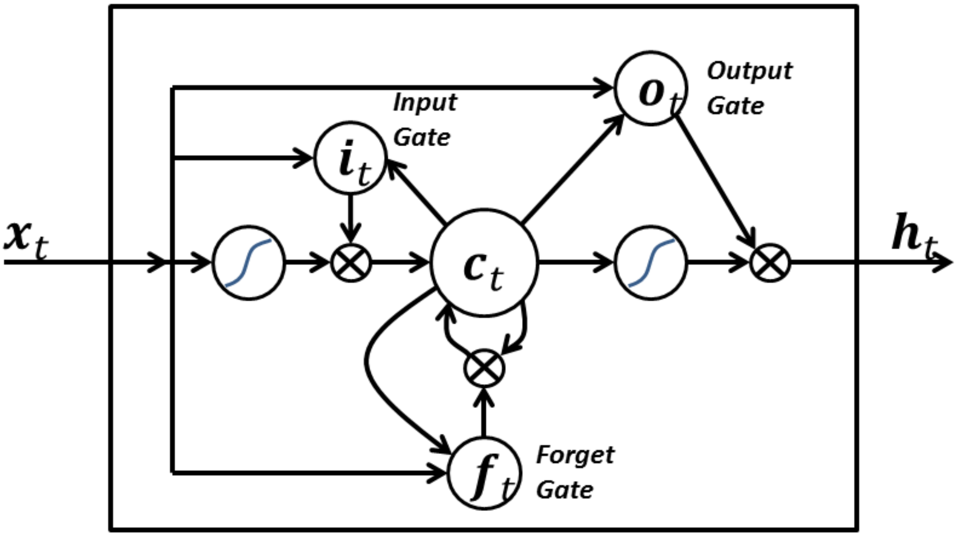
Recurrent Neural Network (RNN) Layers

Recurrent neural network layers are primitives which allow neural networks to learn from sequences of inputs. This layer assumes that the input evolves from sequence step to next sequence step following a defined update rule which can be learned from data. This update rule presents a prediction of the next state in the sequence given all the states which have come previously.



Long Short-Term Memory (LSTM) Cells

The RNNs presented in the previous section are capable of learning arbitrary sequence update rules in theory. In practice however, such models typically forget the past rapidly. So RNN layers are not adept at modeling longer term connections from the past, of the type that commonly arise in language modeling. The Long Short-Term Memory (LSTM) cell is a modification to the RNN layer that allows for signals from deeper in the past to make their way to the present.



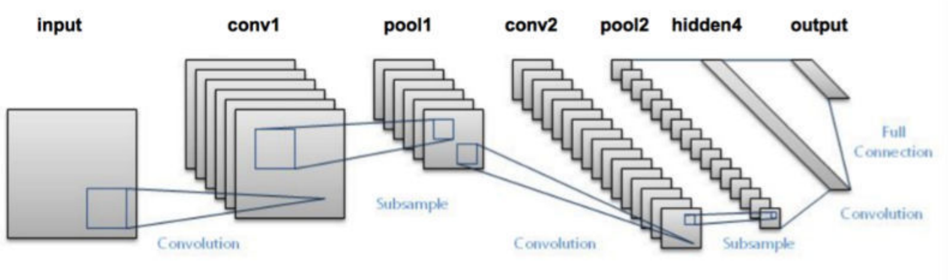
Deep Learning Zoo

There have been hundreds of different deep learning models that combine the deep learning primitives presented in the previous section. Some of these architectures have been historically important. Others were the first presentations of novel designs that influenced perceptions of what deep learning could do.

In this section, we present a "zoo" of different deep learning architectures that have proven influential for the research community. We want to emphasize that is an episodic history that makes no attempt to be exhaustive. The models presented here are simply those that caught the authors' fancy. There are certainly important models in the literature which have not been presented here.

LeNet

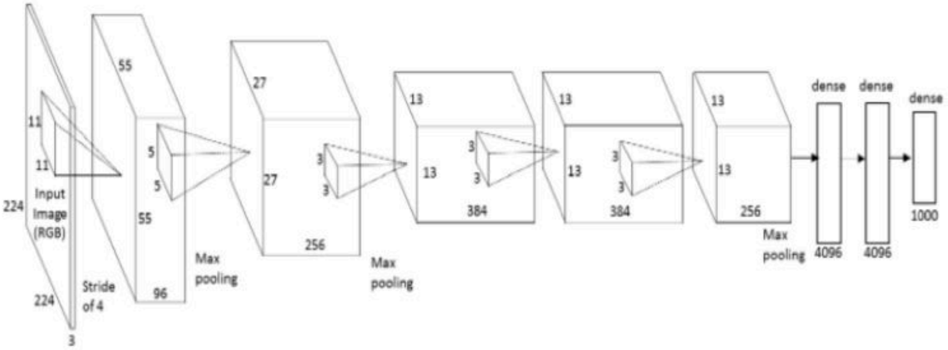
The LeNet architecture is arguably the first prominent "deep" convolutional architecture. Introduced in 1988, it was used to perform optical character recoginition (OCR) for documents. Although it performed its task admirably, the computational cost of the LeNet was extreme for the architectures available at the time, so the design languished in (relative) obscurity for a few decades after its creation.



AlexNet

The ImageNet Large Scale Visual Recognition Challenge (ILSVRC) was first organized in 2010 as a test of the progress made in visual recognition systems. The organizers made use of Amazon Mechanical Turk, an online platform to connect workers to requesters, to catalog a large collection of images with assocated lists of objects present in the image. The use of Mechanical Turk permitted the curation of a collection of data significantly larger than those gathered previously.

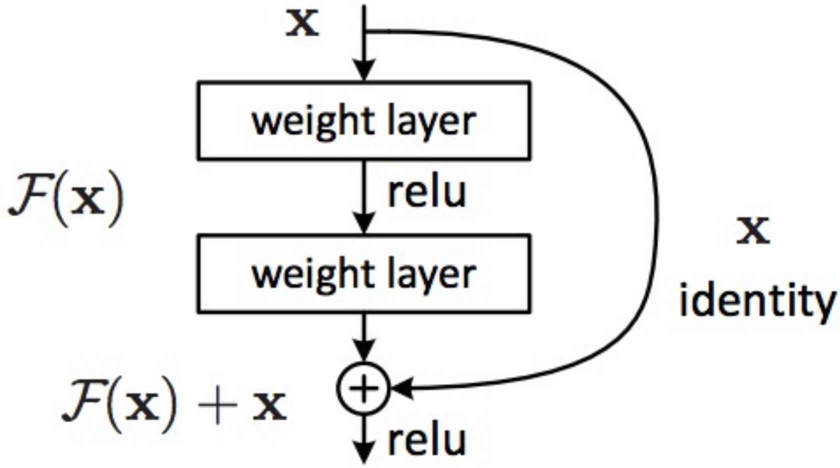
The first two years the challenge ran, more traditional machine-learned systems which relied on systems like HOG and SIFT features (hand-tuned visual feature extraction methods) triumphed. In 2012, the AlexNet architecture, based on a modification of LeNet run on powerful GPUs, entered and dominated the challenge with error rates one half of the nearest entrants. The strength of this victory dramatically galvanized the (already nascent) trend towards deep learning architectures in computer vision.



ResNet

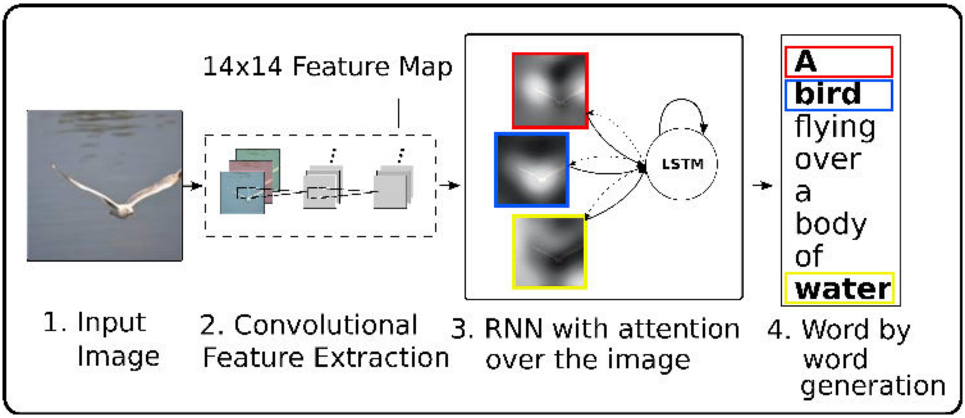
Since 2012, convolutional architectures have consistently won the ILSVRC challenge (along with many other computer vision challenges). The ResNet architecture, winner of the ILSVRC 2015 challenge, is particularly notable because it goes much deeper than previously convolutional architectures such as AlexNet. ResNet architecture trend up to 130 layers deep, in contrast to the 8-10 layer architectures that won previously.

Very deep networks historically were challenging to learn; when deep networks go this far down, they start to run into the vanishing gradients problem. Put less technically, signals are attenuated as they progress through the network, leading to diminished learning. This attenuation can be explained mathematically, but the effect is that each additional layer multiplicatively reduces the strength of the signal, leading to caps on the effective depth of networks. The ResNet introduced an innovation which controlled this attenuation, the bypass connection. These connections allow signals to pass through the network undiminished and permit signals from dozens of layers deeper to communicate with the output.


x
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

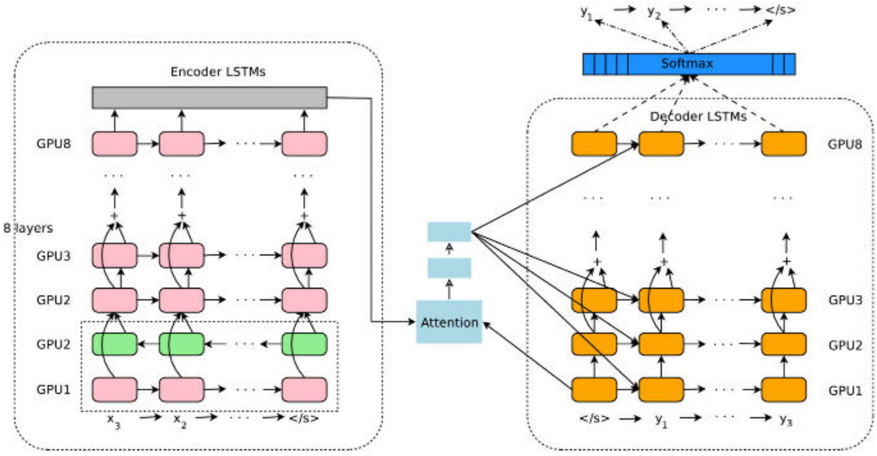
Neural Captioning Model

As practitioners became more comfortable with the use of deep learning primitives, they started to experiment with mixing and matching these primitive modules to create higher-order systems that could perform more complex tasks than basic object detection. Neural captioning systems attempt to automatically generate captions for the contents of images. They do so by combining a convolutional network, which extracts information from images, with an LSTM to generate a descriptive sentence for the image. The entire system is trained **end-to-end**. That is, the convolutional network and the LSTM network are trained together to achieve the desired goal of generating descriptive sentences for provided images. This end-to-end training is one of the key innovations powering modern deep learning systems.


14x14 Feature Map
LSTM
A
bird
flying
over
a
body
of
water
1. Input Image
2. Convolutional Feature Extraction
3. RNN with attention over the image
4. Word by word generation

Google Neural Machine Translation

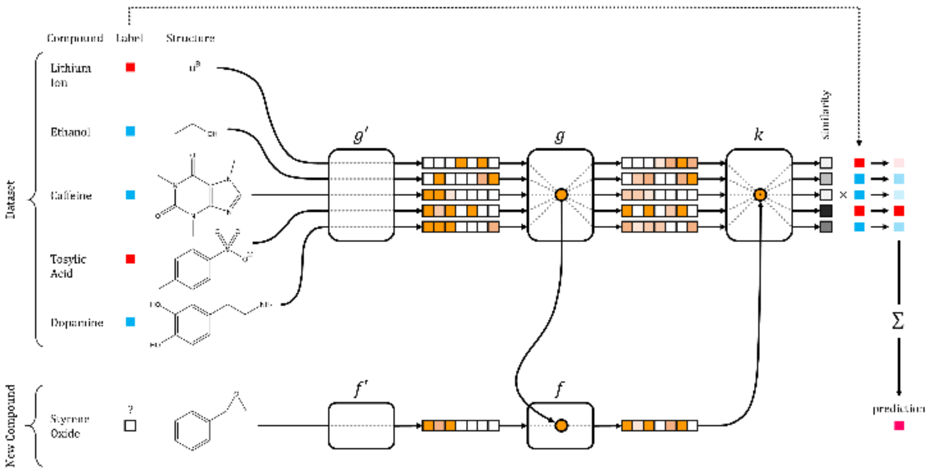
Google's neural machine translation (Google-NMT) system uses the paradigm of end-to-end training to build a production translation system, which takes sentences from the source language directly to the target language. The Google-NTM system depends on the fundamental building block of the LSTM, which it stacks over a dozen times and trains on an extremely large dataset of translated sentences. The final architecture provided for a breakthrough advance in machine-translation by cutting the gap between human and machine translations by up to 60%. The system is deployed widely now, and has already made a significant impression on the popular press.



One shot models

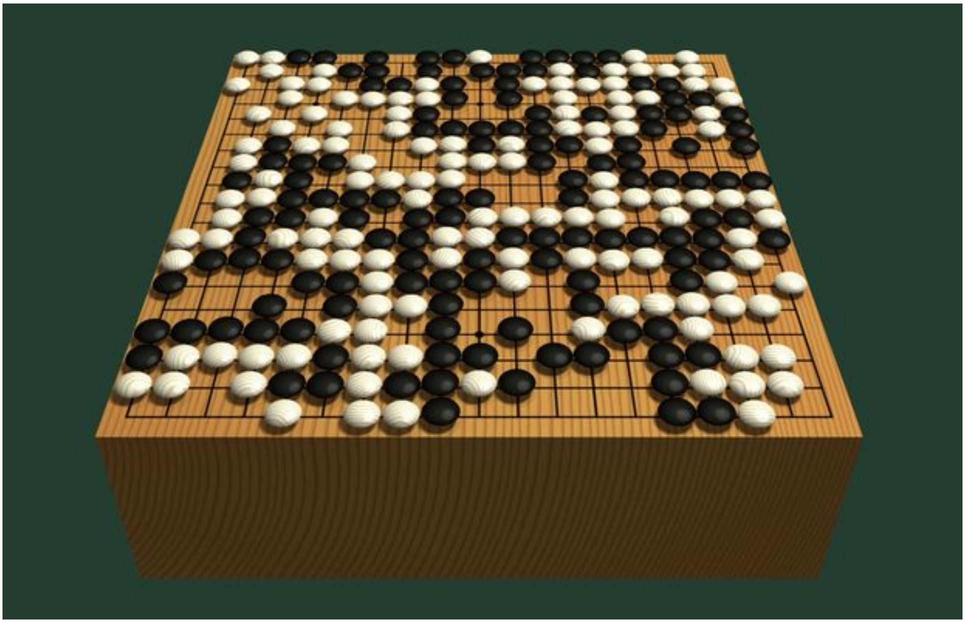
One shot learning is perhaps the most interesting new idea in machine/deep learning. Most deep learning techniques typically require very large amounts of data to learn meangingful behavior. The AlexNet architecture, for example, made use of the large ILSVRC dataset to learn a visual object detector. However, much work in cognitive science has indicated that humans need fewer examples to learn complex concepts. Take the example of baby learning about giraffes for the first time. A baby shown a single giraffe might be capable of learning to recognize a giraffe shown only one example of a giraffe.

Recent progress in deep-learning has started to invent architectures capable of similar learning feats. Given only a few examples of a concept (but given ample sources of side information), such systems can learn to make meaningful predictions with very few datapoints. One recent paper (by one of the authors of this book) used this idea to demonstrate that one-shot learning can function even in contexts babies can't (such as drug discovery for example).

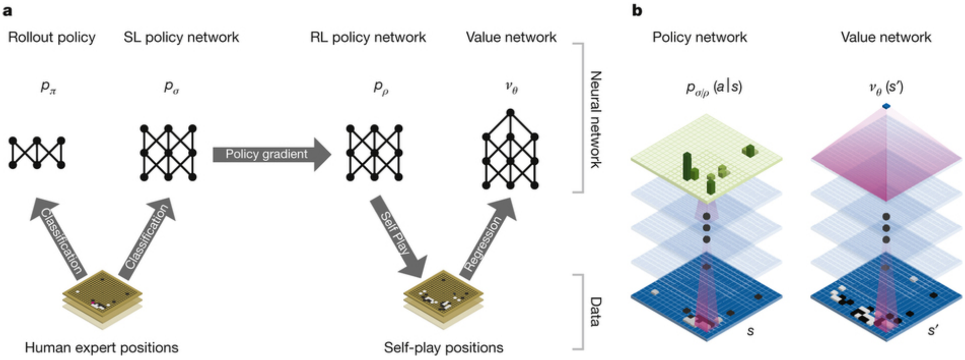


AlphaGo

Go is an ancient board game, widely influential in Asia. Computer Go was a major challenge for computer science since the techniques that enabled the computer chess system DeepBlue to beat Garry Kasparov do not scale to Go. Part of the issue is that Go has a much bigger board than Chess, resulting in far more moves possible per step. As a result, brute force search with contemporary computer hardware is insufficient to solve Go.

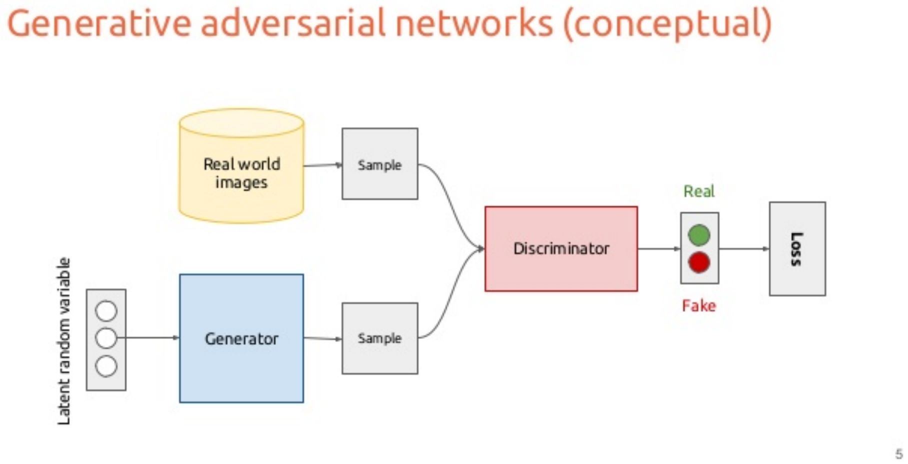


Computer Go was finally solved by AlphaGo from Google Deepmind. AlphaGo proved capable of defeating one of the world's strongest Go champions, Lee Sedol in a 5 game match. Some of the key ideas from AlphaGo are the use of deep value network and a deep policy network. The value network provides an estimate of the value of a board position. Unlike chess, in Go, it's very difficult to guess whether white or black is winning from the board state. The value network solves this problem by learning. The policy network on the other hand helps estimate the best move to take in a current board state. The combination of these two techniques with Monte Carlo Tree search (a more classical search method) helped overcome the large branching factor in Go games.



Generative Adversarial Networks

Generative Adversarial Networks (GANs) are a new type of deep network that uses two dueling neural networks, the generator and the adversary which duel against one another. The generator tries to draw samples from a distribution (say tries to generate realistic looking images of birds). The discriminator then works on differentiating samples drawn from the generator from true data samples (is a particular bird a real image or generator-created). The power of GANs is that this "adversarial" training seems capable of generating image samples of considerably higher fidelity than other techniques.



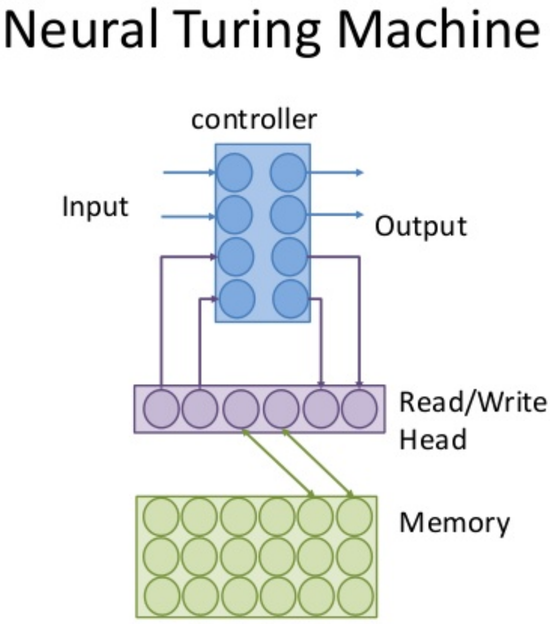
Generative adversarial networks (conceptual)

GANs have proven capable of generating very realistic images, and will likely power the next generation of computer graphics tools. Samples from such systems are now approaching photorealism (although, many theoretical and practical caveats still remain to be worked out with these systems).

A small yellow bird with a black crown and a short black pointed beak
Stage-I
Stage-II

Neural Turing Machines

Most of the deep-learning systems presented so far have only learned limited (even if complex) functions. For example, object detection in images, captioning, machine translation, or Go game-play. But, there's no fundamental reason a deep-learning architecture couldn't learn more sophisticated functions. For example, perhaps we could have deep architectures that learn general algorithms, concepts such as sorting, addition, or multiplication. The Neural Turing Machine (NTM) is a first attempt at making a deep-learning architecture capable of learning arbitrary algorithms. This architecture adds an external memory bank to an LSTM-like system, to allow the deep architecture to make use of scratch space to computer more sophisticated functions. At the moment, NTM-like architectures are still quite limited, and only capable of learning simple algorithms. However, as understanding of the design space improves, this limitation need not hold moving forward.



Deep Learning Frameworks

Researchers have been implementing software packages to facilitate the construction of neural network (deep learning) architectures for decades. Until the last few years, these systems were mostly special purpose and only used within an academic group. This lack of standardized, industrial strength software made it difficult for non-experts to make use of neural network packages.

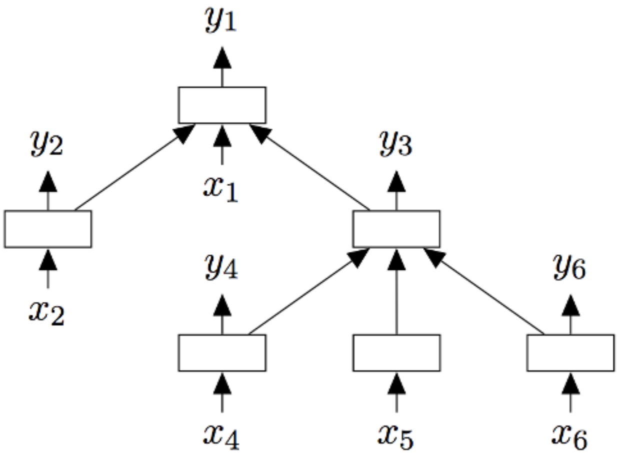
This situation has changed dramatically over the last few years. Google implemented the DistBelief system in 2012 and made use of it to construct and deploy many simpler deep learning architectures. The advent of DistBelief, and similar packages such as Caffe, Theano, Torch and Keras, MxNet and so on have widely spurred industry adoption.

Tensorflow draws upon this rich intellectual history, and builds upon some of these packages (Theano in particular) for design principles. Tensorflow (and Theano) in particular use the concept of tensors as the fundamental underlying primitive powering deep-learning systems. This focus on tensors distinguishes these packages from systems such as DistBelief or Caffe which don't allow the same flexibility for building sophisticated models.

While the rest of this book will focus on Tensorflow, understanding the underlying principles should allow readers to take the lessors learned and apply them with little difficulty to alternate deep learning frameworks. While the details certainly differ, most modern frameworks share the same basis as tensor calculus engines.

Dynamic Graphs

One of the major current weaknesses of TensorFlow is that constructing a new deep learning architecture is relatively slow (on the order of multiple seconds to initialize an architecture). As a result, it's not convenient to construct some sophisticated deep architectures which change their structure on the fly in TensorFlow. One such architecture is the TreeLSTM, which uses the syntactic parse tree of English sentences to perform natural language understanding. Sinch each sentence has a different parse tree, each sentence requires a slightly different architecture.



While such models can be implemented in Tensorflow, doing so requires significant ingenuity due to the limitations of the current Tensorflow API. New frameworks such as Chainer, DyNet, and PyTorch promise to remove these barriers by making the construction of new architectures light-weight enough so that models like the TreeLSTM can be constructed easily. It's likely that improving support for such models will be a major focus for TensorFlow developers moving forward.

One takeaway from this discussion is that progress in the deep learning framework space is rapid, and today's novel system can be tomorrow's old news. However, the fundamental principles of the underlying tensor calculus date back centuries, and will stand readers in good stead regardless of future changes in programming models. This book will emphasize using TensorFlow as a vehicle for developing an intuitive knowledge of the underlying tensor calculus.

Empirical Learning

Machine learning (and deep learning in particular), like much of computer science is a very empirical discipline. It's only really possible to understand deep learning through significant practical experience. For that reason, we've included a number of in-depth case-studies throughout the remainder of this book. We encourage readers to dive deeply into these examples and to get their hands dirty experimenting with their own ideas using Tensorflow. It's never enough to understand algorithms only theoretically.

CHAPTER 2
Introduction to Tensorflow Primitives

This chapter aims to introduce readers to fundamental aspects of Tensorflow. In particular, readers will learn how to perform basic computations in Tensorflow. A large part of this chapter will be spent introducing readers to the concept of tensors, and with how tensors are represented and manipulated within tensorflow. This discussion will necessitate a brief overview of some of the mathematical concepts that underly tensorial mathematics. In particular, we'll briefly review basic linear algebra and demonstrate how to perform basic linear algebraic operations with Tensorflow.

We'll follow this discussion of basic mathematics with a discussion of the differences between declarative and imperative programming styles. Unlike many programming languages, Tensorflow is largely declarative. Calling a tensorflow operation adds a description of a computation to Tensorflow's "computation graph". In particular, tensorflow code "describes" computations and doesn't actually perform them. In order to run Tensorflow code, users need to create `tf.Session` objects. We introduce the concept of sesions and describe how users can use them to perform computations in Tensorflow.

We end the chapter by discussing the notion of variables. Variables in tensorflow hold tensors and allow for stateful computation that modifies variables to occur. We demonstrate how to create variables and update their values via Tensorflow.

Introducing Tensors

Tensors are fundamental mathematical constructs in fields such as physics and engineering. Historically however, tensors have made fewer inroads in computer science, which has traditionally been more associated with discrete mathematics and logic. This state of affairs has started to change significantly with the advent of machine learning and its foundation on continuous, vectorial mathematics. Modern machine learning is founded upon the manipulation of tensors and the calculus of tensors.

Scalars, Vectors, and Matrices

To start, we will give some simple examples of tensors that readers will likely be familiar with. The simplest example of a tensor is a scalar, a single constant value drawn from the real numbers (Recall that the real numbers are decimal numbers of arbitrary precision, with both positive and negative numbers permitted). Mathematically, we denote the real numbers by the $\mathbb{R}$. More formally, we call a scalar a 0-tensor.

Aside on fields

Mathematically sophisticated readers will protest that it's entirely meaningful to define tensors based on the complex numbers, or with binary numbers. More generally, it's sufficient that the numbers come from a **field**: a mathematical collection of numbers where 0, 1, addition, multiplication, subtraction, and division are defined. Common fields include the real numbers $\mathbb{R}$, the rational numbers $\mathbb{Q}$, the complex numbers $\mathbb{C}$, and finite fields such as $\mathbb{Z}_2$. For simplicity, in much of the discussion, we will assume real valued tensors, but substituting in values from other fields is entirely reasonable.

If scalars are 0-tensors, what constitutes a 1-tensor? Formally, speaking, a 1-tensor is a vector; a list of real numbers. Traditional, vectors are written as either column vectors

$$\begin{pmatrix} a \\ b \end{pmatrix}$$

or as row vectors

$$\begin{pmatrix} a & b \end{pmatrix}$$

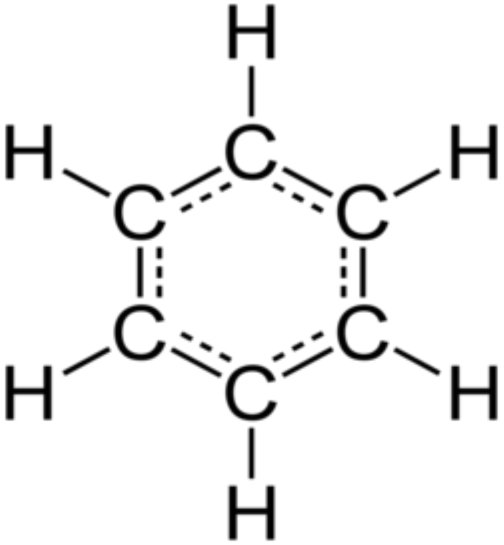
Notationally, the collection of all column vector of length 2 is denoted $\mathbb{R}^{2,1}$ while set of all row vectors of length 2 is $\mathbb{R}^{1,2}$. More computationally, we might saw that the shape of a column vector is $(2, 1)$, while the shape of a row vector is $(1, 2)$. This notion of tensor shape is quite important for understanding tensorflow computations, and we will return to it later on in this chapter. If we don't wish to specify whether a vector is a row vector or column vector, we can say it comes from the set $\mathbb{R}^2$ and has shape (2).

One of the simplest use of vectors is to represent coordinates in the real world. Suppose that we decide on an origin point (say the position where you're current standing). Then any position in the world can be represented by three displacement values from your current position (left-right displacement, front-back displacement, up-down displacement). Thus, the set of vectors (vector space) $\mathbb{R}^3$ can represent any position in the world. (Savvy readers might protest that the earth is spherical, so our representation doesn't match the natural human notion of direction, but it will suffice for a flat-world like a video game).

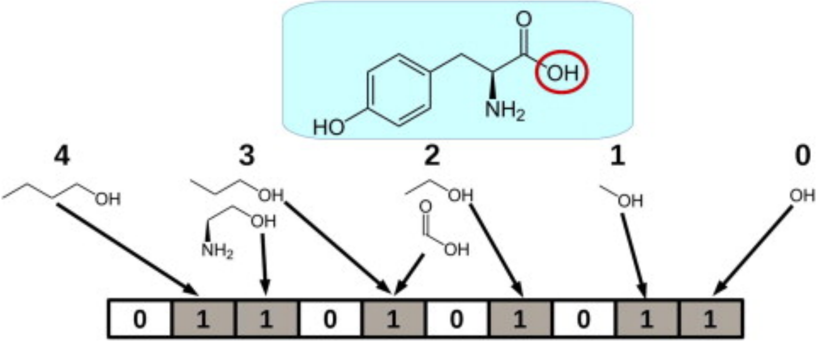
For a different example, let's suppose that a cat is described by its height, weight, and color. Then a video game cat can be represented a vector

$$\begin{pmatrix} \text{height} \\ \text{weight} \\ \text{color} \end{pmatrix}$$

in the space $\mathbb{R}^3$. This type of representation is often called a **featurization**. That is, a featurization is a representation of a real-world entity as a vector (or more generally as a tensor). Nearly all machine learning algorithms operate on vectors or tensors. Thus the process of featurization is a critical part of any machine learning pipeline. Often, the featurization system can be the most sophisticated part of a machine learning system. Suppose we have a benzene molecule



How can we transform this molecule into a vector suitable for a query to a machine learning system? There are a number of potential solutions to this problem, most of which exploit the idea of marking the presence of subfragments of the molecule. The presence or absence of specific subfragments is marked by setting indices in a binary vector (in $\{0, 1\}^n$) to 1/0 respectively.



Note that this process sounds (and is) fairly complex. In fact, one of the most challenging aspects of building a machine learning system is deciding how to transform the data in question into a tensorial format. For some types of data, this transformation is obvious. For others (such as molecules), the transformation required can be quite subtle. For the practitioner of machine learning, it isn't usually necessary to invent a new featurization method since the research literature is extensive, but it will often be necessary to read the scholarly literature to understand best practices for transforming a new data stream.

Now that we have established that 0-tensors are scalars ($\mathbb{R}$) and that 1-tensors are vectors ($\mathbb{R}^n$), what is a 2-tensor? Traditionally, a 2-tensor is referred to as a matrix

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

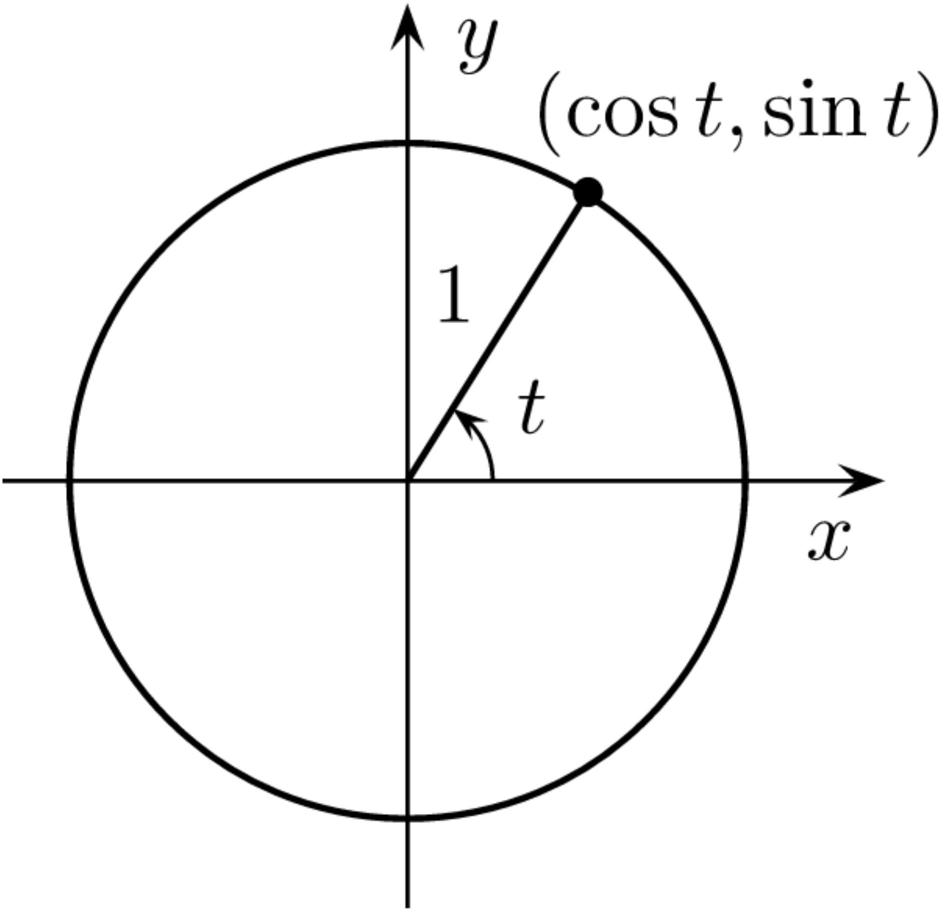
The matrix above has two rows and two columns. The set of all such matrices is referred to as $\mathbb{R}^{(2,2)}$. Returning to our notion of tensor shape above, the shape of this matrix is $(2, 2)$. Matrices are traditionally used to represent transformations of vectors. For example, the action of rotating a vector in the plane by angle α can performed by the matrix

$$R_\alpha = \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix}$$

To see this, note that the x unit vector $(1, 0)$ is transformed by matrix multplication into vector $(\cos(\alpha), \sin(\alpha))$. (We will cover the detailed definition of matrix multiplication later in the chapter, but will simply display the result for the moment).

$$\begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} \cos(\alpha) \\ \sin(\alpha) \end{pmatrix}$$

This transformation can be visualized graphically as well



Matrix Mathematics

There are a number of standard mathematical operations on matrices that machine learning programs use repeatedly. We will briefly review some of the most fundamental of these operations. The matrix transpose is a convenient operation that flips a matrix around its diagonal. Mathematically, suppose A is a matrix, then the transpose matrix A^T is defined by equation $A^T_{i,j} = A_{j,i}$. For example, the transpose of the rotation matrix R_α above is

$$R_\alpha^T = \begin{pmatrix} \cos(\alpha) & \sin(\alpha) \\ -\sin(\alpha) & \cos(\alpha) \end{pmatrix}$$

Addition of matrices is only defined for matrices of the same shape and is simply performed elementwise. For example

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

Similarly, matrices can be multiplied by scalars. In this case, each element of the matrix is simply multiplied elementwise by the scalar in question.

$$2 \cdot \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} = \begin{pmatrix} 2 & 4 \\ 6 & 8 \end{pmatrix}$$

Furthermore, it is sometimes possible to multiply two matrices directly. This notion of matrix multiplication is probably the most important mathematical concept associated with matrices. Note specifically that matrix multiplication is **not** the same notion as element-wise multiplication of matrices. Rather, suppose we have a matrix A of shape (m, n) with m rows and n columns. Then, A can be multiplied on the right by any matrix B of shape (n, k) (where k is any positive integer) to form matrix AB of shape (m, k). For the actual mathematical description, suppose A is a matrix of shape (m, n) and B is a matrix of shape (n, k). Then AB is defined by

$$(AB)_{i,j} = \sum_k A_{i,k} B_{k,j}$$

We displayed a matrix multiplication equation earlier in brief. Let's expand that example now that we have the formal definition.

$$\begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} \cos(\alpha) \cdot 1 - \sin(\alpha) \cdot 0 \\ \sin(\alpha) \cdot 1 - \cos(\alpha) \cdot 0 \end{pmatrix} = \begin{pmatrix} \cos(\alpha) \\ \sin(\alpha) \end{pmatrix}$$

The fundamental intuition is that rows of one matrix are multiplied against columns of the other matrix.

This definition hides a number of subtleties. Note first that matrix multiplication is not commutative. That is, $AB \neq BA$ in general. In fact, AB can exist when BA is not meaningful. Suppose for example A is a matrix of shape $(2, 3)$ and B is a matrix of shape $(3, 4)$. Then AB is a matrix of shape $(2, 4)$. However BA is not defined since $4 \neq 2$. As another subtlety, note that, as in the rotation example above, a matrix of shape (m, n) can be multiplied on the right by a matrix of shape $(n, 1)$. However, a matrix of shape $(n, 1)$ is simply a column vector. So, it is meaningful to multiply matrices by vectors. Matrix-vector multiplication is one of the fundamental building blocks of common machine learning systems.

One of the nicest properties of standard multiplication is that it is a linear operation. More precisely, a function f is called linear if $f(x + y) = f(x) + f(y)$ and $f(cx) = cf(x)$ where c is a scalar. To demonstrate that scalar multiplication is linear, suppose that a, b, c, d are all real numbers. Then we have

$$a \cdot (b \cdot c) = b \cdot (ac)$$

$$a \cdot (c + d) = ac + ad$$

We make use of the commutative and distributive properties of scalar multiplication above. Now suppose that instead, A, C, D are now matrices where C, D are of the same size and it is meaningful to multiply AC or D (b remains a real number). Then matrix multiplication is a linear operator:

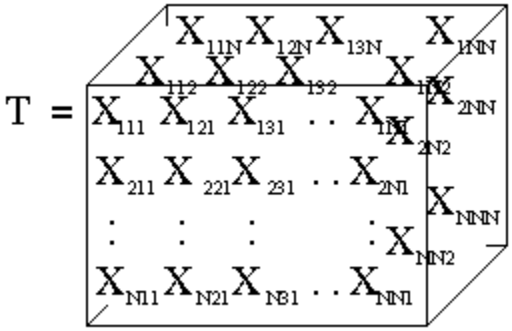
$$A(b \cdot C) = b \cdot (AC)$$

$$A(C + D) = AC + AD$$

Put another way, matrix multiplication is distributive and commutes with scalar multiplication. In fact, it can be shown that the notion of linearity completely defines matrix multiplication mathematically. For a computer science analogy, think of linearity as a property demanded by an abstract method in a superclass. Then standard multiplication and matrix multiplication are concrete implementations of that abstract method for different subclasses (respectively real numbers and matrices).

Tensors

In the previous sections, we introduced the notion of scalars as 0-tensors, vectors as 1-tensors, and matrices as 2-tensors. What then is a a 3-tensor? Before passing to a general definition, it can help to think about the commonalities between scalars, vectors, and matrices. Scalars are single numbers. Vectors are lists of numbers. To pick out any particular element of a vector requires knowing its index. Hence, we need 1 index element into the vector (thus a 1-tensor). Matrices are tables of numbers. To pick out any particular element of a matrix requires knowing its row and column. Hence, we need 2 index elements (thus a 2-tensor). It follows naturally that a 3-tensor is a set of numbers where there are 3 required indices. It can help to think of a 3-tensor as a rectangular prism of numbers



The 3-tensor T displayed above is of shape (N, N, N). An arbitrary element of the tensor would then be selected by specifying (i, j, k) as indices.

There is a linkage between tensors and shapes. A 1-tensor has a shape of length 1, a 2-tensor a shape of length 2, and a 3-tensor of length 3. The astute reader might protest that this contradicts our earlier discussion of row and column vectors. By our definition, a column vector has shape $(n, 1)$. Wouldn't that make a column vector a 2-tensor (or a matrix)? This is exactly what has happened. Recall that a vector which is not specified to be a row vector or column vector has shape (n). When we specify that a vector is a row vector or a column vector, we in fact specify a method of transforming the underlying vector into a matrix. This type of dimension expansion is a common trick in tensor manipulation, and as we will see later in the chapter, amply supported within tensorflow.

Note that another way of thinking about a 3-tensor is as a list of matrices all with the same shape. Suppose that W is a matrix with shape (n, n). Then the tensor $T_{ijk} = \left(W_1, \cdots, W_n\right)$ consists of n copies of the matrix W stacked back-to-back.

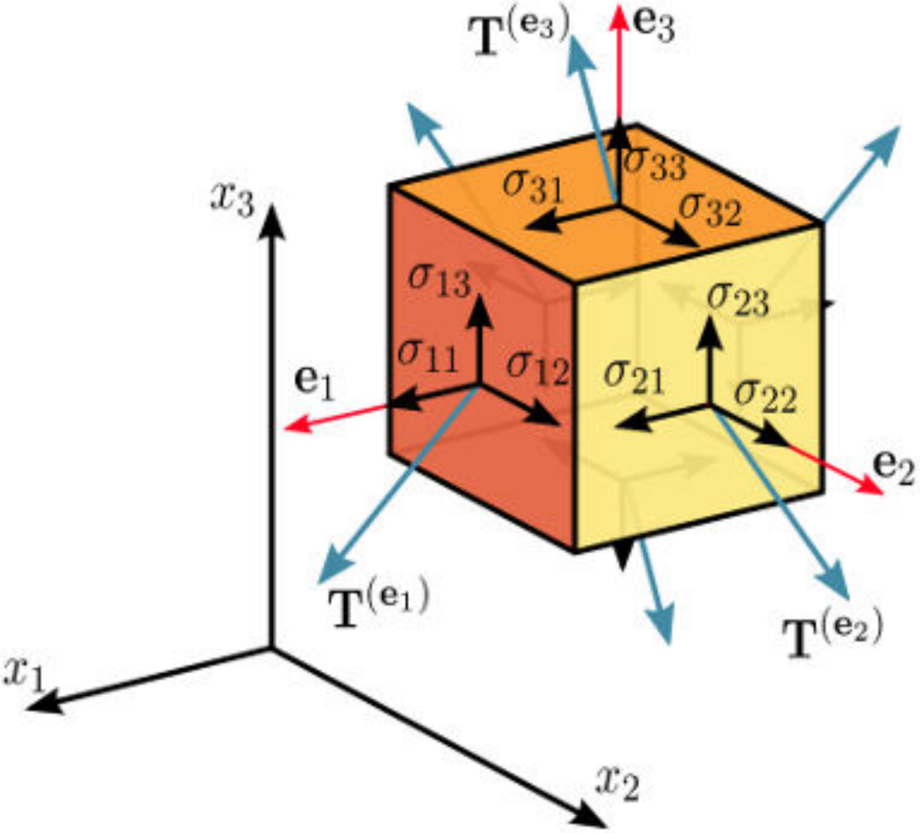
Note that a black and white image can be represented as a 2-tensor. Suppose we have a 224x224 pixel black and white image. Then, pixel (i, j) is 1/0 to encode a black/white pixel respectively. It follows that a black and white image can be represented as a matrix of shape $(224, 224)$. Now, consider a 224x224 color image. The color at a particular is typically represented by 3 separate RGB channels. That is, pixel (i, j) is represented as a tuple of numbers (r, g, b) which encode the amount of red, green, and blue at the pixel respectively. Each of r, g, b is typically an integer from 0 to 255. It follows now that the color image can be encoded as a 3-tensor of shape $(224, 224, 3)$. Continuing the analogy, consider a color video. Suppose that each frame of the video is a 224x224 color image. Then a minute of video (at 60 fps) would be a 4-tensor of shape $(224, 224, 3, 3600)$. Continuing even further, a collection of 10 such videos would then form a 5-tensor of shape $(10, 224, 224, 3, 3600)$. In general, tensors provide for a convenient representation of numeric data. In practice, it's not common to see tensors of higher order than 5-tensors, but it's good to design any tensor software to allow for arbitrary tensors since intelligent users will always come up with use cases designers don't consider.

Tensors in physics.

Tensors are used widely in physics to encode fundamental physical quantities. For example, the stress tensor is commonly used in material science to define the stress at a point within a material. Mathematically, the stress tensor is a 2-tensor of shape (3, 3)

$$\sigma = \begin{pmatrix} \sigma_{1,1} & \sigma_{1,2} & \sigma_{1,3} \\ \sigma_{2,1} & \sigma_{2,2} & \sigma_{2,3} \\ \sigma_{3,1} & \sigma_{3,2} & \sigma_{3,3} \end{pmatrix}$$

Then, suppose that n is a vector of shape (3) that encode a direction. The stress T^n in direction n is specified by the vector $T^n = T \cdot n$ (note the matrix-vector multiplication). This relationship is depicted pictorially below



As another physical example, Einstein's field equations of general relativity are commonly expressed in tensorial format

$$R_{\mu\nu} - \frac{1}{2} R g_{\mu\nu} + \Lambda g_{\mu\nu} = \frac{8\pi G}{c^4} T_{\mu\nu}$$

Here $R_{\mu\nu}$ is the Ricci curvature tensor, $g_{\mu\nu}$ is the metric tensor, $T_{\mu\nu}$ is the stress-energy tensor, and the remaining quantities are scalars. Note however, that there's an important subtletly distinguishing these tensors and the other tensors we've discussed previously. Quantities like the metric tensor provide a separate tensor (in the sense of an array of numbers) for each point in space-time (mathematically, the metric tensor is a tensor field). The same holds for the stress tensor discussed above, and for the other tensors in these equations. At a given point in space-time, each of these quantities becomes a symmetric 2-tensor of shape $(4, 4)$ using our notation.

Part of the power of modern tensor calculus systems such as Tensorflow is that some of the mathematical machinery long used for classical physics can now be adapted to solve applied problems in image processing, and language understanding. At the same time, today's Tensor calculus systems are still limited compared with the mathematical machinery of physicists. For example, there's no simple way to talk about a quantity such as the metric tensor using Tensorflow yet. We hope that as tensor calculus becomes more fundamental to computer science, the situation will change and that systems like Tensorflow will serve as a bridge between physical world and the computational world.

Mathematical Asides

The discussion so far in this chapter has introduced tensors informally via example and illustration. In our definition, a tensor is simply an array of numbers. It's often convenient to view a tensor as a function instead. The most common definition introduces a tensor as a multilinear function from a product of vector spaces to the real numbers.

$$T : V_1 \times V_2 \times \cdots V_n \to \mathbb{R}$$

This definition uses a number of terms we haven't introduced yet. A vector space is simply a collection of vectors. We've seen a few examples of vector spaces such as $\mathbb{R}^3$ or generally $\mathbb{R}^n$. We won't lose any generality by holding that $V_i = \mathbb{R}^{d_i}$. As we defined previously, a function f is linear if $f(x + y) = f(x) + f(y)$ and $f(cx) = cf(x)$. A multilinear function is simply a function which is linear in each argument. This function can be viewed as assigning individual entries of a multidimensional array, when provided indices into the array as arguments.

We won't use this more mathematical definition much in this book, but it serves as a useful bridge to connect the deep learning concepts we will learn about with the centuries of mathematical research that have been undertaken on tensors by the physics and mathematics communities.

Covariance and Contravariance.

Our definition here has swept many details under the rug which would need to be carefully attended to for a formal treatment. For example, we don't touch upon the notion of covariant and contravariant indices here. What we call a n-tensor is better described as a (p, q)-tensor where $n = p + q$ and p is the number of contravariant indices, and q the number of covariant indices. Matrices are $(1, 1)$-tensors for example. As a subtlety, there are 2-tensors which are not matrices! We won't dig into these topics carefully here since they don't crop up much in machine learning, but we encourage discerning readers to understand how covariance and contravariance affect the machine learning systems they construct.

Basic Computations in Tensorflow

We've spent the last sections covering the mathematical definitions of various tensors. It's now time to cover how to create an manipulate tensors using Tensorflow. For this section, we recommend readers follow along using an interactive python session (with IPython). Many of the basic Tensorflow concepts are easiest to understand after experimenting with them directly.

When experimenting with Tensorflow interactively, it's convenient to use `tf.InteractiveSession()`. Invoking this statement within IPython will make Tensorflow behave almost imperatively, allowing beginners to play with tensors much more easily. We will enter into an in-depth discussion of imperative vs. declarative style and of sessions later in this chapter.

```
>>> import tensorflow as tf
>>> tf.InteractiveSession()
<tensorflow.python.client.session.InteractiveSession>
```

The rest of the code in this section will assume that an interactive session has been loaded.

Initializing Constant Tensors.

Until now, we've discussed tensors as abstract mathematical entities. However, a system like Tensorflow must run on a real computer, so any tensors must live on computer memory in order to be useful to computer programmers. Tensorflow provides a number of functions which instantiate basic tensors in memory. The simplets of these are `tf.zeros()` and `tf.ones()`. `tf.zeros()` takes a tensor shape (represented as a python tuple) and returns a tensor of that shape filled with zeros. Let's try invoking this command in the shell.

```
>>> tf.zeros(2)
<tf.Tensor 'zeros:0' shape=(2,) dtype=float32>
```

It looks like Tensorflow returns a reference to the desired tensor rather than the value of the tensor itself. To force the value of the tensor to be returned, we will use the method `tf.Tensor.eval()` of tensor objects. Since we have initialized `tf.InteractiveSession()`, this method will return the value of the zeros tensor to us.

```
>>> a = tf.zeros(2)
>>> a.eval()
array([ 0.,  0.], dtype=float32)
```

Note that the evaluated value of the tensorflow tensor is itself a python object. In particular, `a.eval()` is a `numpy.ndarray` object. Numpy is a sophisticated numerical system for python. We won't attempt an in-depth discussion of Numpy here beyond noting that Tensorflow is designed to be compatible with Numpy conventions to a large degree.

We can call `tf.zeros()` and `tf.ones()` to create and display tensors of various sizes.

```
>>> a = tf.zeros((2, 3))
>>> a.eval()
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]], dtype=float32)
>>> b = tf.ones((2,2,2))
>>> b.eval()
array([[[ 1.,  1.],
        [ 1.,  1.]],

       [[ 1.,  1.],
        [ 1.,  1.]]], dtype=float32)
```

To provide a crude analogy, `tf.zeros()` and `tf.ones()` are like the C function `malloc()` which allocates memory for programs to work in. This analogy doesn't stretch far however, since Tensorflow doesn't often compute on CPU memory. (Most heavy-duty Tensorflow systems perform computations on GPU memory. We won't get into the details of how Tensorflow manages various computing devices here).

What if we'd like a tensor filled with some quantity besides 0/1? The `tf.fill()` method provides a nice shortcut for doing so.

```
>>> b = tf.fill((2, 2), value=5.)
>>> b.eval()
array([[ 5.,  5.],
       [ 5.,  5.]], dtype=float32)
```

`tf.constant` is another function, similar to `tf.fill` which allows for construction of Tensors which shouldn't change during the program execution.

```
>>> a = tf.constant(3)
>>> a.eval()
3
```

Sampling Random Tensors

Although working with constant tensors is convenient for testing ideas, it's much more common to initialize tensors with random values. The most common way to do this is to sample each entry in tensor from a random distribution. `tf.random_normal` allows for each entry in a tensor of specified shape to be sampled from a Normal distribution of specified mean and standard deviation.

Symmetry Breaking

Many machine learning algorithms learn by performing updates to a set of tensors that hold weights. These update equations usually satisfy the property that weights initialized at the same value will continue to evolve together. Thus, if the initial set of tensors is initialized to a constant value, the model won't be capable of learning much. Fixing this situation requires *symmetry breaking*. The easiest way of breaking symmetry is to sample each entry in a tensor randomly.

```
>>> a = tf.random_normal((2, 2), mean=0, stddev=1)
>>> a.eval()
array([[-0.73437649, -0.77678096],
       [ 0.51697761,  1.15063596]], dtype=float32)
```

One thing to note is that machine learning systems often make use of very large tensors which often have tens of millions of parameters. At these scales, it becomes common to sample random values from Normal distributions which are far from the mean. Such large samples can lead to numerical instability, so it's common to sample using `tf.truncated_normal()` instead of `tf.random_normal()`. This function behaves the same as `tf.random_normal()` in terms of API, but drops and resamples all values more than 2 standard deviations from the mean.

`tf.random_uniform()` behaves like `tf.random_normal()` except for the fact that random values are sampled from the Uniform distribution over a specified range.

```
>>> a = tf.random_uniform((2, 2), minval=-2, maxval=2)
>>> a.eval()
array([[-1.90391684,  1.4179163 ],
       [ 0.67762709,  1.07282352]], dtype=float32)
```

Tensor Addition and Scaling

Tensorflow makes use of python's operator overloading to make basic tensor arithmetic straightforward with standard python operators.

```
>>> c = tf.ones((2, 2))
>>> d = tf.ones((2, 2))
>>> e = c + d
>>> e.eval()
array([[ 2.,  2.],
       [ 2.,  2.]], dtype=float32)
>>> f = 2 * e
>>> f.eval()
array([[ 4.,  4.],
       [ 4.,  4.]], dtype=float32)
```

Tensors can also be multiplied this way. Note however that this is element wise multiplication and **not** matrix multiplication.

```
>>> c = tf.fill((2,2), 2.)
>>> d = tf.fill((2,2), 7.)
>>> e = c * d
>>> e.eval()
array([[ 14.,  14.],
       [ 14.,  14.]], dtype=float32)
```

Matrix Operations

Tensorflow provides a variety of amenities for working with matrices. (Matrices by far are the most common type of tensor used in practice). In particular, Tensorflow provides shortcuts to make certain types of commonly used matrices. The most widely used of these is likely the identity matrix. Identity matrices are square matrices which are 0 everywhere except on the diagonal, where they are 1. `tf.eye()` allows for fast construction of identity matrices of desired size.

```
>>> a = tf.eye(4)
>>> a.eval()
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.]], dtype=float32)
```

Diagonal matrices are another common type of matrix. Like identity matrices, diagonal matrices are only nonzero along the diagonal. Unlike the diagonal matrices, they may take arbitrary values along the diagonal. Let's construct a diagonal matrix with ascending values along the diagonal. To start, we'll need a method to construct a vector of ascending values in Tensorflow. The easiest way for doing is invoking `tf.range(start, limit, delta)`. The resulting vector can then be fed to `tf.diag(diagonal)` which will construct a matrix with the specified diagonal.

```
>>> r = tf.range(1, 5, 1)
>>> r.eval()
array([1, 2, 3, 4], dtype=int32)
>>> d = tf.diag(r)
>>> d.eval()
array([[1, 0, 0, 0],
       [0, 2, 0, 0],
       [0, 0, 3, 0],
       [0, 0, 0, 4]], dtype=int32)
```

Now suppose that we have a specified matrix in Tensorflow. How do we compute the matrix transpose? `tf.matrix_transpose()` will do the trick nicely.

```
>>> a = tf.ones((2, 3))
>>> a.eval()
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]], dtype=float32)
>>> at = tf.matrix_transpose(a)
>>> at.eval()
array([[ 1.,  1.],
       [ 1.,  1.],
       [ 1.,  1.]], dtype=float32)
```

Now, let's suppose we have a pair of matrices we'd like to multiply using matrix multiplication. The easiest way to do so is by invoking `tf.matmul()`.

```
>>> a = tf.ones((2, 3))
>>> a.eval()
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]], dtype=float32)
>>> b = tf.ones((3, 4))
>>> b.eval()
array([[ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.]], dtype=float32)
>>> c = tf.matmul(a, b)
>>> c.eval()
array([[ 3.,  3.,  3.,  3.],
       [ 3.,  3.,  3.,  3.]], dtype=float32)
```

Conscientious readers can check that this answer matches the mathematical definition of matrix multiplication we provided above.

Tensor types

Readers may have noticed the `dtype` notation in the examples above. Tensors in tensorflow come in a variety of types such as `tf.float32`, `tf.float64`, `tf.int32`, `tf.int64`. It's possible to to create tensors of specified types by setting `dtype` in tensor construction functions. Furthermore, given a tensor, it's possible to change its type using casting functions such as `tf.to_double()`, `tf.to_float()`, `tf.to_int32()`, `tf.to_int64()` and others.

```
>>> a = tf.ones((2,2), dtype=tf.int32)
>>> a.eval()
array([[0, 0],
       [0, 0]], dtype=int32)
>>> b = tf.to_float(a)
>>> b.eval()
```

```
array([[ 0.,  0.],
       [ 0.,  0.]], dtype=float32)
```

Tensor Shape Manipulations

Within Tensorflow, tensors are just collections of numbers written in memory. The different shapes are views into the underlying set of numbers that provide different ways of interacting with the set of numbers. At different times, it can be useful to view the same set of numbers as forming tensors with different shapes. tf.reshape() allows tensors to be converted into tensors with different shapes.

```
>>> a = tf.ones(8)
>>> a.eval()
array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.], dtype=float32)
>>> b = tf.reshape(a, (4, 2))
>>> b.eval()
array([[ 1.,  1.],
       [ 1.,  1.],
       [ 1.,  1.],
       [ 1.,  1.]], dtype=float32)
>>> c = tf.reshape(a, (2, 2, 2))
>>> c.eval()
array([[[ 1.,  1.],
        [ 1.,  1.]],

       [[ 1.,  1.],
        [ 1.,  1.]]], dtype=float32)
```

Notice how we can turn the original 1-tensor into a 2-tensor and then into a 3-tensor with tf.reshape. While all necessary shape manipulations can be performed with tf.reshape(), sometimes it can be convenient to perform simpler shape manipulations using functions such as tf.expand_dims or tf.squeeze. tf.expand_dims adds an extra dimension to a tensor of size 1. It's useful for increasing the rank of a tensor by one (for example, when converting a vector into a row vector or column vector). tf.squeeze on the other hand removes all dimensions of size 1 from a tensor. It's a useful way to convert a row or column vector into a flat vector.

This is also a convenient opportunity to introduce the tf.Tensor.get_shape() method. This method lets users query the shape of a tensor.

```
>>> a = tf.ones(2)
>>> a.get_shape()
TensorShape([Dimension(2)])
>>> a.eval()
array([ 1.,  1.], dtype=float32)
>>> b = tf.expand_dims(a, 0)
>>> b.get_shape()
TensorShape([Dimension(1), Dimension(2)])
>>> b.eval()
array([[ 1.,  1.]], dtype=float32)
```

```
>>> c = tf.expand_dims(a, 1)
>>> c.get_shape()
TensorShape([Dimension(2), Dimension(1)])
>>> c.eval()
array([[ 1.],
       [ 1.]], dtype=float32)
>>> d = tf.squeeze(b)
>>> d.get_shape()
TensorShape([Dimension(2)])
>>> d.eval()
array([ 1.,  1.], dtype=float32)
```

Introduction to Broadcasting

Broadcasting is a term (introduced by Numpy) for when a tensor systems matrices and vectors of different sizes can be added together. These rules allow for conveniences like adding a vector to every row of a matrix. Broadcasting rules can be quite complex, so we will not dive into a formal discussion of the rules. It's often easier to experiment and see how the broadcasting works.

```
>>> a = tf.ones((2, 2))
>>> a.eval()
array([[ 1.,  1.],
       [ 1.,  1.]], dtype=float32)
>>> b = tf.range(0, 2, 1, dtype=tf.float32)
>>> b.eval()
array([ 0.,  1.], dtype=float32)
>>> c = a + b
>>> c.eval()
array([[ 1.,  2.],
       [ 1.,  2.]], dtype=float32)
```

Notice that the vector `b` is added to every row of matrix `a` above. Notice another subtlety that we explicitly set the `dtype` for `b` above. If the dtype isn't set, Tensorflow will report a type error. Let's see what would have happened if we hadn't set the `dtype` above.

```
>>> b = tf.range(0, 2, 1)
>>> b.eval()
array([0, 1], dtype=int32)
>>> c = a + b
ValueError: Tensor conversion requested dtype float32 for Tensor with dtype int32:
'Tensor("range_2:0", shape=(2,), dtype=int32)
```

Unlike languages like C, Tensorflow doesn't perform implicit type casting under the hood. It's often necessary to perform explicit type casts when doing arithmetic operations.

Imperative and Declarative Programming

Most situations in computer science involve imperative programming. Consider a simple python program

```
>>> a = 3
>>> b = 4
>>> c = a + b
>>> c
7
```

This program, when translated into machine code, instructs the machine to perform a primitive addition operation on two registers, one which contains 3, and the other which contains 4. The result is then 7. This style of programming is called **imperative** since programming tells the computer explicitly which actions to perform.

An alternate style of programming is **declarative**. In a declarative system, a computer program is a high level description of the computation which is to be performed. It does not instruct the computer exactly how to perform the computation. Consider the Tensorflow equivalent of the program above.

```
>>> a = tf.constant(3)
>>> b = tf.constant(4)
>>> c = a + b
>>> c
<tf.Tensor 'add_1:0' shape=() dtype=int32>
>>> c.eval()
7
```

Note that the value of `c` isn't `7`! Rather, it's a Tensor. The code above specifies the computation of adding two values together to create a new Tensor. The actual computation isn't executed until we call `c.eval()`. In the sections before, we have been using the `eval()` method to simulate imperative style in Tensorflow since it can be challenging to understand declarative programming at first.

Note that declarative programming is by no means an unknown concept to software engineeering. Relational databases and SQL provide an example of a widely used declarative programming system. Commands like SELECT and JOIN may be implemented in an arbitrary fashion under the hood so long as their basic semantics are preserved. Tensorflow code is best thought of as analogous to a SQL program; the Tensorflow code specificies a computation to be performed, with details left up to Tensorflow. The Tensorflow developers exploit this lack of detail under the hood to tailor the execution style to the underlying hardward, be it CPU, GPU, or mobile device.

However, it's important to note that the grand weakness of declarative programming is that the abstraction is quite leaky. For example, without detailed understanding of the underlying implementation of the relational database, long SQL programs can become unbearably inefficient. Similarly, large Tensorflow programs implemented without understanding of the underlying learning algorithms are unlikely to work well. In the rest of this section, we will start paring back the abstraction, a process we will continue through the rest of the book.

Tensorflow Graphs

Any computation in Tensorflow represented as an instance of a `tf.Graph` object. Such as graph consists of a set of instances of `tf.Tensor` objects and `tf.Operation` objects. We have covered `tf.Tensor` in some detail, but what are `tf.Operation` objects? We have already seen them over the course of this chapter. A call to an operation like `tf.matmul` creates a `tf.Operation` instance to mark the need to perform the matrix multiplication operation.

When a `tf.Graph` is not explicitly specified, Tensorflow adds tensors and operations to a hidden global `tf.Graph` instance. This instance can be fetched by `tf.get_default_graph()`.

```
>>> tf.get_default_graph()
<tensorflow.python.framework.ops.Graph>
```

It is possible to specify that Tensorflow operations be performed in graphs other than the default. We will demonstrate examples of this in future chapters.

Tensorflow Sessions

In Tensorflow, a `tf.Session()` object stores the context under which a computation is performed. At the beginning of this chapter, we used `tf.InteractiveSession()` to set up an environment for all Tensorflow computations. This call created a hidden global context for all computations performed. We then used `tf.Tensor.eval()` to execute our declaratively specified computations. Underneath the hood, this call is evaluated in context of this hidden global `tf.Session`. It can be convenient (and often necessary) to use an explicit context for a computation instead of a hidden context.

```
>>> sess = tf.Session()
>>> a = tf.ones((3, 3))
>>> b = tf.matmul(a, a)
>>> b.eval(session=sess)
array([[ 2.,  2.],
       [ 2.,  2.]], dtype=float32)
```

This code evaluates `b` in the context of `sess` instead of the hidden global session. In fact, we can make this more explicit with an alternate notation

```
>>> sess.run(b)
array([[ 2.,  2.],
       [ 2.,  2.]], dtype=float32)
```

In fact, calling `b.eval(session=sess)` is just syntactic sugar for calling `sess.run(b)`.

This entire discussion may smack a bit of sophistry. What does it matter which session is in play given that all the different methods seem to return the same answer? Explicit sessions don't really show their value until we start to perform computations which have state, a topic we will discuss in the following section.

Tensorflow Variables

All the code that we've dealt with in this section has dealt in constant tensors. While we could combine and recombine these tensors in any way we chose, we could never change the value of tensors themselves (only create new tensors with new values). The style of programming so far has been **functional** and not **stateful**. While functional computations are very useful, machine learning often depends heavily on stateful computations. Learning algorithms are essentially rules for updating stored tensors to explain provided data. If it's not possible to update these stored tensors, it would be hard to learn.

The `tf.Variable()` class provides a wrapper around tensors which allows for stateful computations. The variable objects serve as holders for tensors. Creating a variable is easy enough.

```
>>> a = tf.Variable(tf.ones((2, 2)))
>>> a
<tensorflow.python.ops.variables.Variable>
```

What happens when we try to evaluate the variable `a` as thought it were a tensor?

```
>>> a.eval()
FailedPreconditionError: Attempting to use uninitialized value Variable
```

The evaluation fails since variables have to be excplicitly initialized. The easiest way to initialize all variables is to invoke `tf.global_variables_initializer`. Running this operation within a session will initialize all variables in the program.

```
>>> sess = tf.Session()
>>> sess.run(tf.global_variables_initializer())
>>> a.eval(session=sess)
array([[ 1.,  1.],
       [ 1.,  1.]], dtype=float32)
```

After initialization, we are able to fetch the value stored within the variable as though it were a plain tensor. So far, there's not much more to variables than plain tensors. Variables only become interesting once we can assign to them. `tf.assign()` lets us do this. Using `tf.assign()` we can update the value of an existing variable.

```
>>> sess.run(a.assign(tf.zeros((2,2))
array([[ 0.,  0.],
       [ 0.,  0.]], dtype=float32)
>>> sess.run(a)
```

```
array([[ 0.,  0.],
       [ 0.,  0.]], dtype=float32)
```

What would happen if we tried to assign to variable `a` a value not of shape `(2,2)`? Let's find out.

```
>>> sess.run(a.assign(tf.zeros((3,3))))
ValueError: Dimension 0 in both shapes must be equal, but are 2 and 3 for 'Assign_3'
(op: 'Assign') with input shapes: [2,2], [3,3].
```

We see that Tensorflow complains. The shape of the variable is fixed upon initialization and must be preserved with updates. As another interesting note, `tf.assign` is itself a part of the underlying global `tf.Graph` instance. This allows Tensorflow programs to update their internal state every time they are run. We shall make heavy use of this feature in the chapters to come.

Review

In this chapter, we've introduced the mathematical concept of tensors, and briefly reviewed a number of mathematical concepts associated with tensors. We then demonstrated how to create tensors in Tensorflow and perform these same mathematical operations within Tensorflow. We also briefly introduced some underlying tensorflow structures like the computational graph, sessions, and variables. If you haven't completely grasped the concepts discussed in this chapter, don't worry much about it. We will repeatedly use these same concepts over the remainder of the book, so there will be plenty of chances to let the ideas sink in.