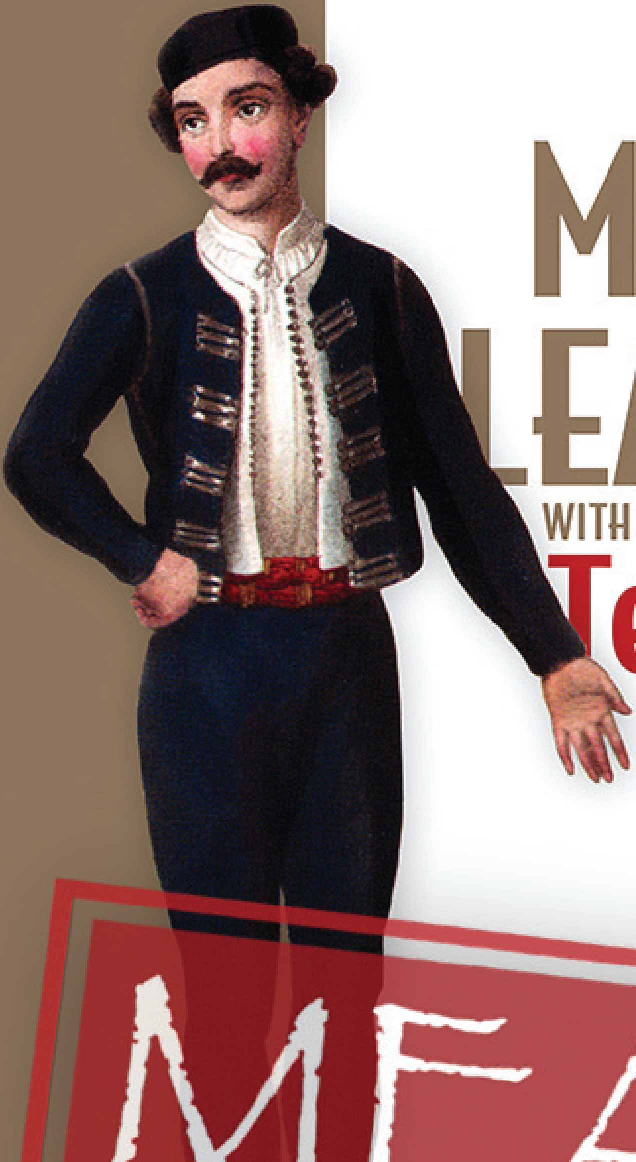
MACHINE LEARNING WITH TensorFlow

Nishant Shukla



MANNING

MEAP Edition
Manning Early Access Program

Machine Learning with TensorFlow

Version 8

welcome

Dear fellow early adopters, curious readers, and puzzled newcomers,

Thank you all for every bit of communication with me, whether it be through the official book forums, through email, on GitHub, or even on Reddit. I've listened carefully to your questions, suggestions, and concerns, regardless of whether or not I've replied to you (and I do apologize for not replying to you).

In the latest edition, I am proud to announce a beautiful makeover of every chapter. The text is greatly improved and slowed down to better cover complex matters, especially the areas where you requested more explanation. Most figures and mathematical equations have been updated to look crisp and professional. The code is now updated to TensorFlow v1.0, and it is also available on GitHub at https://github.com/BinRoot/TensorFlow-Book/. Also, the chapters are rearranged to better deliver the right skills at the right time, if the book were read in order.

Thank you for investing in the MEAP edition of *Machine Learning with TensorFlow*. You're one of the first to dive into this introductory book about cutting-edge machine learning techniques using the hottest technology (spoiler alert: I'm talking about TensorFlow). You're a brave one, dear reader. And for that, I reward you generously with the following.

You're about to learn machine learning from scratch, both the theory and how to easily implement it. As long as you roughly understand object-oriented programming and know how to use Python, this book will teach you everything you need to know to start solving your own big-data problems, whether it be for work or research.

TensorFlow was released just over a year ago by some company that specializes in search engine technology. Okay, I'm being a little facetious; well-known researchers at Google engineered this library. But with such prowess comes intimidating documentation and assumed knowledge. Fortunately for you, this book is down-to-earth and greets you with open arms.

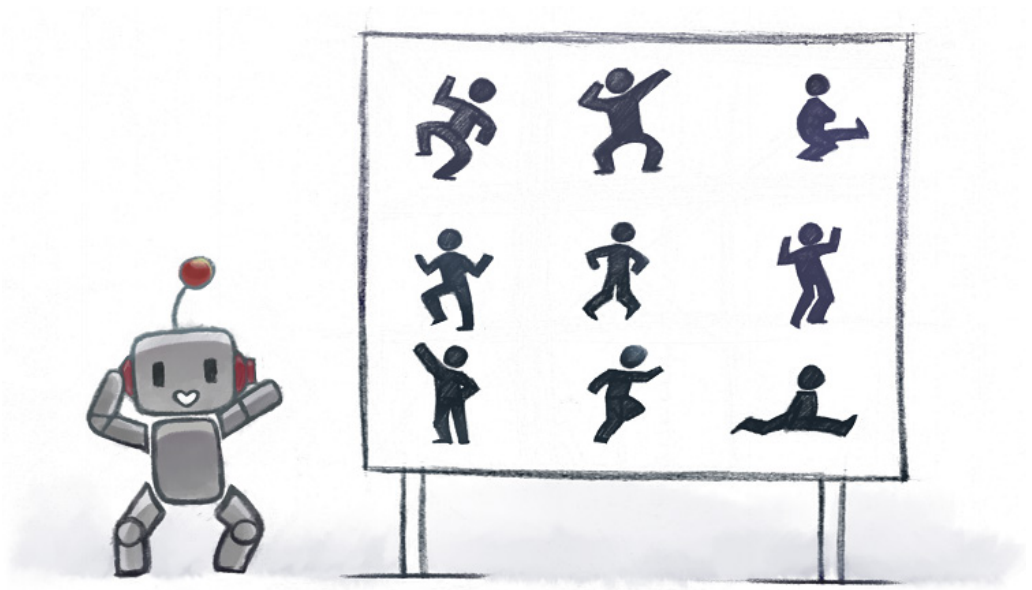
Each chapter zooms into a prominent example of machine learning, such as classification, regression, anomaly detection, clustering, and many modern neural networks. Cover them all to master the basics, or cater it to your needs by skipping around.

Keep me updated on typos, mistakes, and improvements because this book is undergoing heavy development. It's like living in a house that's still actively under construction; at least you won't have to pay rent. But on a serious note, your feedback along the way will be appreciated.

With gratitude,
—Nishant Shukla

brief contents

1
A machine-learning odyssey



This chapter covers

- **Machine learning fundamentals**
- **Data representation, features, and vector norms**
- **Existing machine learning tools**
- **Why TensorFlow**

Have you ever wondered if there are limits to what computer programs can solve? Nowadays, computers appear to do a lot more than simply unravel mathematical equations. In the last half-century, programming has become the ultimate tool to automate tasks and save time, but how much can we automate, and how do we go about doing so?

Can a computer observe a photograph and say "ah ha, I see a lovely couple walking over a bridge under an umbrella in the rain?" Can software make medical decisions as accurately as that of trained professionals? Can software predictions about the stock market perform better than human reasoning? The achievements of the past decade hint that the answer to all these questions is a resounding "yes," and the implementations appear to share a common strategy.

Recent theoretic advances coupled with newly available technologies have enabled anyone with access to a computer to attempt their own approach at solving these incredibly hard problems. Okay, not just anyone, but that's why you're reading this book, right?

A programmer no longer needs to know the intricate details of a problem to solve it. Consider converting speech to text: a traditional approach may involve understanding the biological structure of human vocal chords to decipher utterances using many hand-designed, domain-specific, un-generalizable pieces of code. Nowadays, it's possible to write code that simply looks at many examples, and figures out how to solve the problem given enough time and examples.

The algorithm learns from data, similar to how humans learn from experiences. Humans learn by reading books, observing situations, studying in school, exchanging conversations, browsing websites, among other means. How can a machine possibly develop a brain capable of learning? There's no definitive answer, but world-class researchers have developed intelligent programs from different angles. Among the implementations, scholars have noticed recurring patterns in solving these kinds of problems that has led to a standardized field that we today label as *machine learning* (ML).

Trusting machine learning output

Detecting patterns is a trait that's no longer unique to humans. The explosive growth of computer clock-speed and memory has led us to an unusual situation: computers now can be used to make predictions, catch anomalies, rank items, and automatically label images. This new set of tools provides intelligent answers to ill-defined problems, but at the subtle cost of trust. Would you trust a computer algorithm to dispense vital medical advice such as whether to perform heart surgery?

There is no place for mediocre machine learning solutions. Human trust is too fragile, and our algorithms must be robust against doubt. Follow along closely and carefully in this chapter.

1.1 Machine learning fundamentals

Have you ever tried to explain to someone how to swim? Describing the rhythmic joint movements and fluid patterns is overwhelming in its complexity. Similarly, some software problems are too complicated for us to easily wrap our minds around. For this, machine learning may be just the tool to use.

Hand-crafting carefully tuned algorithms to get the job done was once the only way of building software. From a simplistic point of view, traditional programming assumes a deterministic output for each of its input. Machine learning, on the other hand, can solve a class of problems where the input-output correspondences are not well understood.

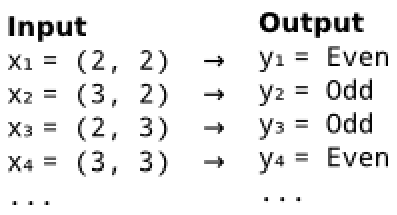
Full speed ahead!

Machine learning is a relatively young technology, so imagine you're a geometer in Euclid's era, paving the way to a newly discovered field. Or, treat yourself as a physicist during the time of Newton, possibly pondering something equivalent to general relativity for the field of machine learning.

Machine Learning is about software that learns from previous experiences. Such a computer program improves performance as more and more examples are available. The hope is that if you throw enough data at this machinery, it will learn patterns and produce intelligent results for newly fed input.

Another name for machine learning is *inductive learning*, because the code is trying to infer structure from data alone. It's like going on vacation in a foreign country, and reading a local fashion magazine to mimic how to dress up. You can develop an idea of the culture from images of people wearing local articles of clothing. You are learning *inductively*.

You might have never used such an approach when programming before because inductive learning is not always necessary. Consider the task of determining whether the sum of two arbitrary numbers is even or odd. Sure, you can imagine training a machine learning algorithm with millions of training examples (outlined in Figure 1.1), but you certainly know that's overkill. A more direct approach can easily do the trick.



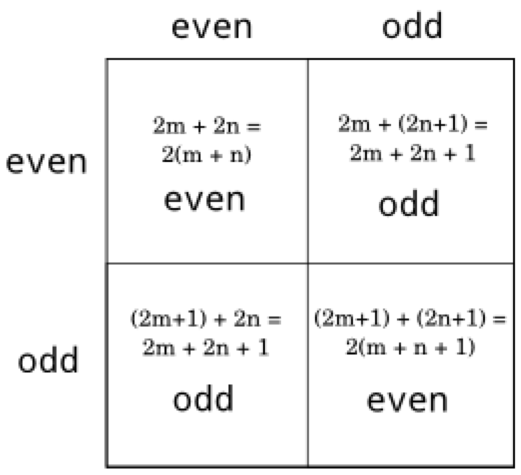
Input		Output
$x_1 = (2, 2)$	$\rightarrow$	$y_1 =$ Even
$x_2 = (3, 2)$	$\rightarrow$	$y_2 =$ Odd
$x_3 = (2, 3)$	$\rightarrow$	$y_3 =$ Odd
$x_4 = (3, 3)$	$\rightarrow$	$y_4 =$ Even
...		...

Figure 1.1 Each pair of integers, when summed together, results in an even or odd number. The input and output correspondences listed are called the ground-truth dataset.

For example, the sum of two odd numbers is always an even number. Convince yourself: take any two odd numbers, add them up, and check whether the sum is an even number. Here's how you can prove that fact directly:

> **For any integer *n*, the formula *2n+1* produces an odd number. Moreover, any odd number can be written as *2n+1* for some value *n*. So the number *3* can be written *2(1)* + *1*. And the number *5* can be written *2(2)* + *1*.**
>
> **So, let's say we have two different odd numbers *2n+1* and *2m+1*, where *n* and *m* are integers. Adding two odd numbers together yields *(2n+1)* + *(2m+1)* = *2n* + *2m* + *2* = *2(n+m+1)*. This is an even number because 2 times anything is even.**

Likewise, we see that the sum of two even numbers is also an even number: *2m* + *2n* = *2(m+n)*. And lastly, we also deduce that the sum of an even with an odd is an odd number: *2m* + *(2n+1)* = *2(m+n)* + *1*. Figure 1.2 visualizes this logic more clearly.



	even	odd
even	$2m + 2n = 2(m + n)$ even	$2m + (2n+1) = 2m + 2n + 1$ odd
odd	$(2m+1) + 2n = 2m + 2n + 1$ odd	$(2m+1) + (2n+1) = 2(m + n + 1)$ even

Figure 1.2 This table reveals the inner logic behind how the output response corresponds to the input pairs.

That's it! With absolutely no use of machine learning, you can solve this task on any pair of integers someone throws at you. Simply applying mathematical rules directly can solve this problem. However, in ML algorithms, we can treat the inner logic as a *black box*, meaning the logic happening inside might not be obvious to interpret.

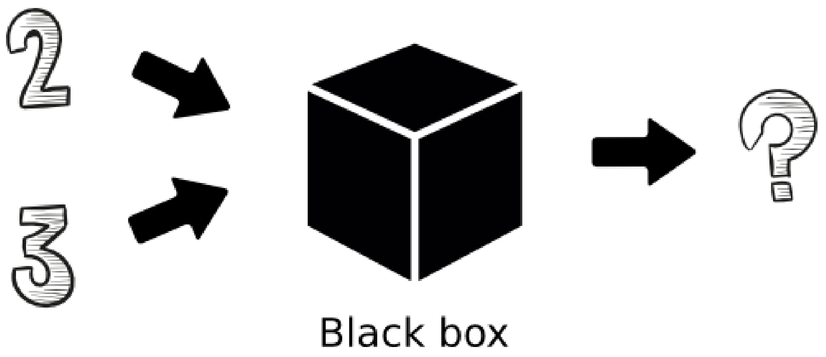


Figure 1.3 An ML approach to solving problems can be thought of as tuning the parameters of a black box until it produces satisfactory results.

PARAMETERS

Sometimes the best way to devise an algorithm that transforms an input to its corresponding output is too complicated. For example, if the input were a series of numbers representing a grayscale image, you can imagine the difficulty in writing an algorithm to label every object seen in the image. Machine learning comes in handy when the inner workings are not well understood. It provides us with a toolset to write software without adequately defining every detail of the algorithm. The programmer can leave some values undecided and let the machine learning system figure out the best values by itself.

The undecided values are called *parameters*, and the description is referred to as the *model*. Your job is to write an algorithm that observes existing examples to figure out how to best tune parameters to achieve the best model. Wow, that's a mouthful! Don't worry, this concept will be a reoccurring motif.

Machine learning might solve a problem without much insight

By mastering this art of inductive problem solving, we wield a double-edged sword. Although ML algorithms may appear to answer correctly to our tests, tracing the steps of deduction to reason why a result is produced may not be as immediate. An elaborate machine learning system learns thousands of parameters, but untangling the meaning behind each parameter is sometimes not the prime directive. With that in mind, I assure you there's a world of magic to unfold.

EXERCISE Suppose you've collected three months-worth of stock market prices. You would like to predict future trends to outsmart the system for monetary gains. Without using ML, how would you go about solving this problem? (As we'll see in chapter 8, this problem becomes approachable using ML techniques.)

LEARNING AND INFERENCE

Suppose you're trying to bake some desserts in the oven. If you're new to the kitchen, it can take days to come up with both the right combination and perfect ratio of ingredients to make something that tastes great. By recording recipes, you can remember how to quickly repeat the dessert if you happen to discover the ultimate tasting meal.

Similarly, machine learning shares this idea of recipes. Typically, we examine an algorithm in two stages: *learning* and *inference*. The objective of the learning stage is to describe the data, which is referred to as the *learned model*. This is our recipe.

Similar to how recipes can be shared and used by other people, the learned model is also reused by other software. The learning stage is the most time-consuming. Running an algorithm may take hours, if not days or weeks, to converge into a useful model. Figure 1.4 outlines the learning pipeline.

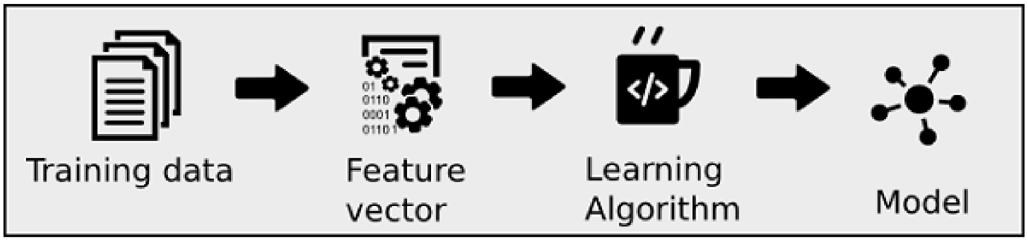


Figure 1.4 The general learning paradigm follows a structured recipe. First, the dataset needs to be transformed into a representation, most often a list of vectors, which can be used by the learning algorithm. The learning algorithm choses a model and efficiently searches for the model's parameters.

The inference stage uses the model to make intelligent remarks about never-before-seen data. It's like using a recipe you found online. The process typically takes orders of magnitude less time than learning, sometimes even being real-time. Inference is all about testing the model on new data, and observing performance in the process, as shown in figure 1.5.

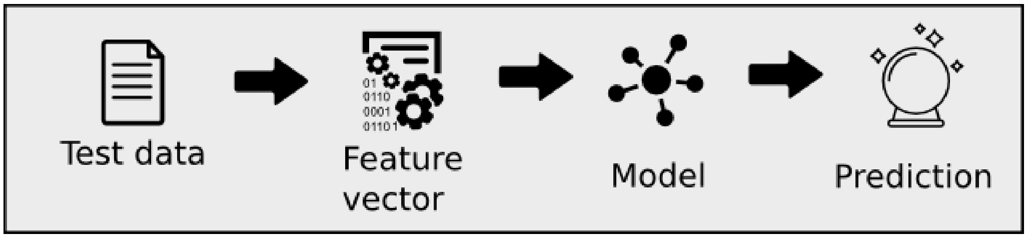


Figure 1.5 The general inference paradigm uses a model that has already been either learned or simply given. After converting data to a usable representation, such as a feature vector, it uses the model to produce intended output.

1.2 Data representation and features

Data is the first-class citizen of machine learning. Computers are nothing more than sophisticated calculators, and so the data we feed our machine learning systems must be mathematical objects, such as (1) vectors, (2) matrices, or (3) graphs.

1. *Vectors* have a flat and simple structure and are the typical embodiment of data in most real-world machine learning applications. They have two attributes: a natural number representing the *dimension* of the vector, and a *type* (such as real numbers, integers, and so on). Just as a refresher, some examples of 2-dimension vectors of integers are (1,2) or (-6,0). Some examples of 3-dimension vectors of real numbers are (1.1, 2.0, 3.9) or (π, π/2, π/3). You get the idea, just a collection of numbers of the same type.
2. Moreover, a vector of vectors is a *matrix*.
3. *Graphs*, on the other hand, are more expressive. A graph is a collection of objects (i.e. *nodes*) that can be linked together with *edges* to represent a network. A graphical structure enables representing relationships between objects, such as in a friendship network or a navigation route of a subway system. Consequently, they are tremendously harder to manage in machine learning applications. In this book, our input data will rarely involve a graphical structure.

Either way, a common theme in all forms of representation is the idea of *features*, which are observable properties of an object. The following scenario will help explain this further. When you're in the market for a new car, keeping tabs on every minor detail between different makes and models is essential. After all, if you're about to spend thousands of dollars, you may as well do so diligently. You would likely record a list of features about each car and compare them back and forth. An ordered list of features is often called a *feature vector*.

When buying cars, comparing mileage might be more lucrative than comparing something less relevant to your interest, such as weight. The number of features to track also must be just right, not too few and not too many. This tremendous effort to select both the number of measurements and which measurements to compare is called *feature engineering*. Depending on which features you examine, the performance of your system can fluctuate dramatically. Selecting the right features to track can make up for a weak learning algorithm.

The general rule of thumb in ML is that more data produces better results. However, the same is not always true for having more features. Perhaps counterintuitive, if the number of features you're tracking is too high, then it may hurt performance. Scholars call this phenomenon the *curse of dimensionality*. Populating the space of all data with representative samples requires exponentially more data as the dimension of the feature vector increases. As a result, feature engineering is one of the most important problems in ML.

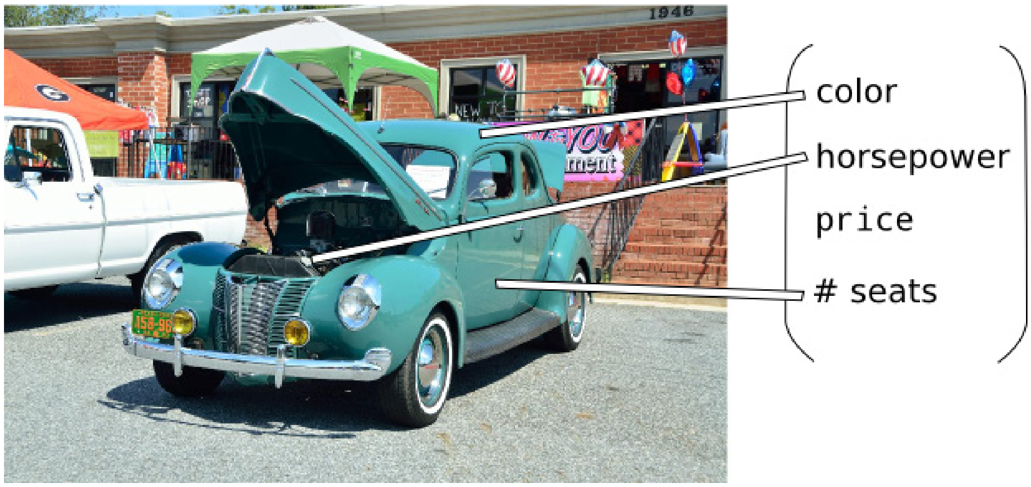


Figure 1.6 Feature engineering is the process of selecting relevant features for the task.

You may not appreciate it immediately, but something consequential happens when you decide which features are worth observing. For centuries, philosophers have pondered the meaning of *identity*; you may not immediately realize this, but you've come up with a definition of identity by your choice of specific features.

Imagine writing a machine learning system to detect faces in an image. Let's say one of the necessary features for something to be a face is the presence of two eyes. Implicitly, a face is now defined as something with eyes. Do you realize what types of trouble this can get us into? If a photo of a person shows him or her blinking, then our detector will not find a face because it couldn't find two eyes. The algorithm would fail to detect a face when a person in blinking. The definition of a face was inaccurate to begin with, and it's apparent from the poor detection results.

The identity of an object is decomposed into the features from which it's composed. For example, if the features you are tracking of one car exactly match the corresponding features of another car, they may as well be indistinguishable from your perspective. When hand-crafting features, we must take great care not to fall into this philosophical predicament of identity.

EXERCISE Let's say you're teaching a robot how to fold clothes. The perception system sees a shirt lying on a table (figure 1.7). You would like to represent the shirt as a vector of features so you can compare it with different clothes. Decide which features would be most useful to track.

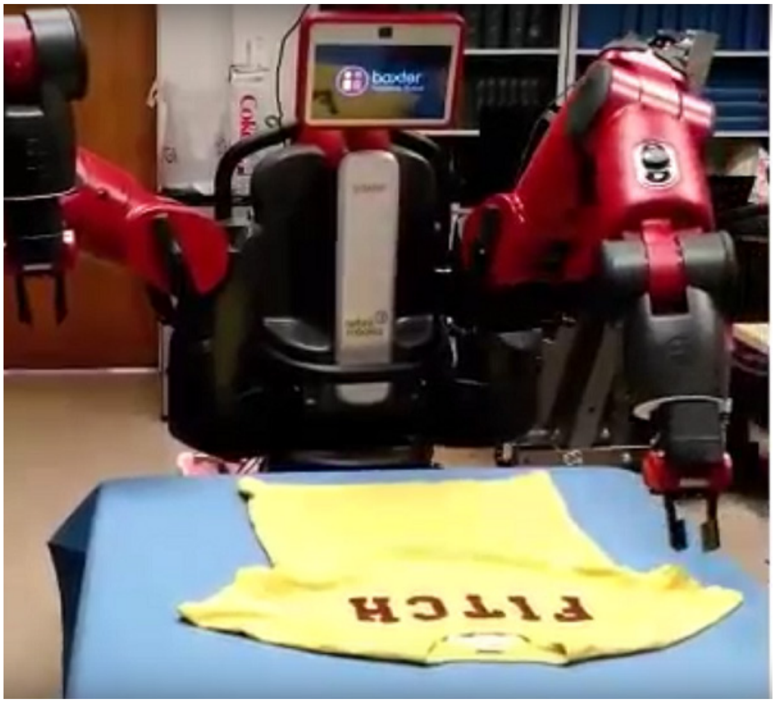


Figure 1.7 A robot is trying to fold a shirt. What are good features of the shirt to track?

EXERCISE Now, instead of detecting clothes, you ambitiously decide to detect arbitrary objects. What are some salient features that can easily differentiate objects?

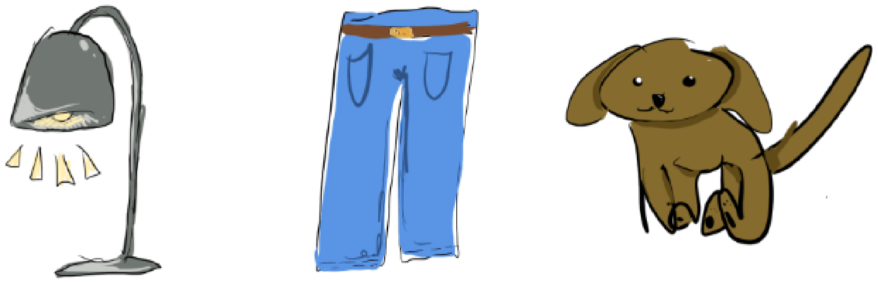


Figure 1.8 Here are images of three objects: a lamp, a pair of pants, and a dog. What are some good features that you should record to compare and differentiate objects?

Feature engineering is a refreshingly philosophical pursuit. For those who enjoy thought-provoking escapades into the meaning of self, I invite you to meditate on feature selection, as it is still an open problem. Fortunately for the rest of you, to alleviate extensive debates, recent advances have made it possible to automatically determine which features to track. You will be able to try it out for yourself in the chapter 8 about autoencoders.

Feature vectors are used in both learning and inference

The interplay between learning and inference is the complete picture of a machine learning system, as seen in figure 1.9. The first step is to represent real-world data into a feature vector. For example, we can represent images by a vector of numbers corresponding to pixel intensities (We'll explore how to represent images in greater detail in future chapters). We can show our learning algorithm the ground truth labels (such as "Bird" or "Dog") along with each feature vector. With enough data, the algorithm generates a learned model. We can use this model on other real-world data to uncover previously unknown labels.

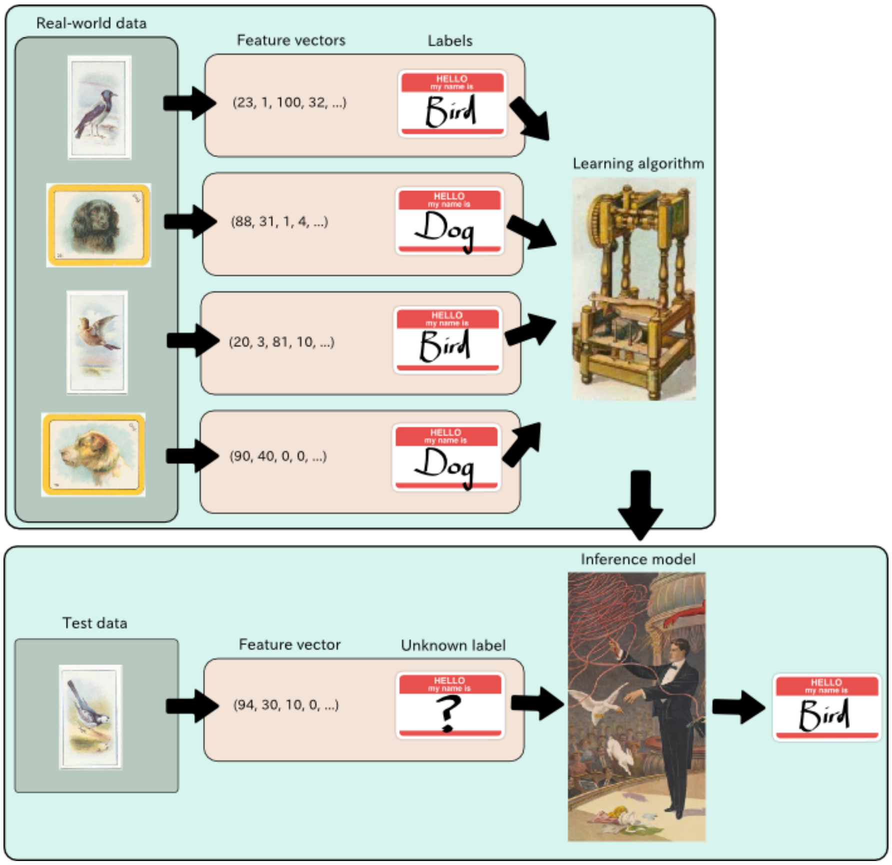


Figure 1.9 Feature vectors are a representation of real world data used by both the learning and inference components of machine learning. The input to the algorithm is not the real-world image directly, but instead its feature vector.

1.3 Distance Metrics

If you have feature vectors of potential cars you want to buy, you can figure out which two are most similar by defining a distance function on the feature vectors. Comparing similarities between objects is an essential component of machine learning. Feature vectors allow us to represent objects so that they we may compare them in a variety of ways. A standard approach is to use the *Euclidian distance*, which is the geometric interpretation you may find most intuitive when thinking about points in space.

Let's say we have two feature vectors, $x = (x_1, x_2, \ldots, x_n)$ and $y = (y_1, y_2, \ldots, y_n)$. The Euclidian distance $||x-y||$ is calculated by

$$(\Sigma(x_n - y_n)^2)^{1/2}$$

For example, the Euclidian distance between (0, 1) and (1, 0) is

```
  ||(0, 1) – (1, 0)||
= ||(-1, 1)||
= ((-1)^2 + 1^2)^1/2
= √2 ≈ 1.414.
```

Scholars call this the *L2 norm*. But that's actually just one of many possible distance functions. There also exists L0, L1, and L-infinity norms. All of these norms are a valid way to measure distance. Here they are in more detail:

- The *L0 norm* counts the total number of non-zero elements of a vector. For example, the distance between the origin (0, 0) and vector (0, 5) is 1, because there is only 1 non-zero element.
- The *L1 norm* is defined as $\Sigma|x_n|$. The distance between two vectors under the L1 norm is also referred to as the *Manhattan distance*. Imagine living in a downtown area like Manhattan, New York, where the streets form a grid. The shortest distance from one intersection to another is along the blocks. Similarity, the L1 distance between two vectors is along the orthogonal directions. So the distance between (0, 1) and (1, 0) under the L1 norm is 2.

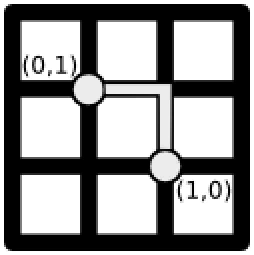


Figure 1.10 The L1 distance is also called the taxi-cab distance because it resembles the route of a car in a grid-like neighborhood such as Manhattan. If a car is travelling from point (0,1) to point (1,0), the shortest route requires a length of 2 units.

- The *L2 norm* is the Euclidian length of a vector, $(\Sigma(x_n)^2)^{1/2}$. It is the most direct route one can possibly take on a geometric plane to get from one point to another.

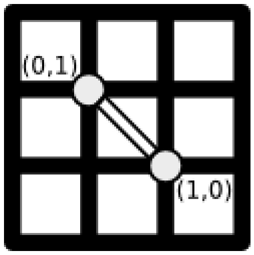


Figure 1.11 The L2 norm between points (0,1) and (1,0) is the length of a single straight line segment reaching both points.

- The *L-N norm* generalizes this pattern, resulting in $(\Sigma|x_n|^N)^{1/N}$. We rarely use finite norms above L2, but it's here for completeness.
- The *L-infinity norm* is $(\Sigma|x_n|^\infty)^{1/\infty}$. More naturally, it is the largest magnitude among each element. If the vector is (-1, -2, -3), t the L-infinity norm will be 3.

When do I use a metric other than the L2 norm in the real-world?

Let's say you're working for a new search-engine start-up trying to compete with established tech giants in the field. Your boss assigns you the task of using machine learning to personalize the search results for each user.

In addition, your boss wants a guarantee that users can only receive less than 5 erroneous search-results per month. Given a vector corresponding to the number of incorrect results shown per user in a month, you are trying to satisfy the boss' condition that the L-infinity norm must be below 5.

Suppose instead that your boss changes his/her mind and requires that less than 5 erroneous search-results can occur for all users collectively. In this case, you are trying to achieve a L1 norm below 5.

Actually, your boss changes his/her mind again. Now, the number of people with erroneous search-results should be less than 5. In that case, you are trying to achieve an L0 norm below 5.

Now that we can compare feature vectors, we have the tools necessary to use data for practical algorithms. Machine learning is often split into three perspectives: supervised learning, unsupervised learning, and reinforcement learning. Let's examine each, one by one.

1.4 Supervised Learning

By definition, a supervisor is someone higher up in the chain of command. When in doubt, he or she dictates what to do. Likewise, *supervised learning* is all about learning from examples laid out by a "supervisor" (such as a teacher).

A supervised machine learning system needs labeled data to develop a useful understanding, which we call its model. For example, given many photographs of people and their recorded corresponding ethnicity, we can train a model to classify the ethnicity of a never-before-seen individual in an arbitrary photograph. Simply put, a model is a function that assigns a label to some data. It does so by using previous examples, called a *training dataset*, as reference.

A convenient way to talk about models is through mathematical notation. Let x be an instance of data, such as a feature vector. The corresponding label associated with x is $f(x)$, often referred to as the *ground truth* of x. Usually, we use the variable $y = f(x)$ because it's quicker to write. In the example of classifying the ethnicity of a person through a photograph, x can be a hundred-dimensional vector of various relevant features, and y is one of a couple values to represent the various ethnicities. Since y is discrete with few values, the model is called a *classifier*. If y could result in many values, and the values have a natural ordering, then the model is called a *regressor*.

Let's denote a model's prediction of x as $g(x)$. Sometimes you can tweak a model to change its performance drastically. Models have some parameters that can be tuned either by a human or automatically. We use the vector θ to represent the parameters. Putting it all together, $g(x|\theta)$ more completely represents the model, read "g of x given θ."

ASIDE **Models may also have *hyper-parameters*, which are extra ad-hoc properties about a model. The word "hyper" in "hyper-parameter" is a bit strange at first. If it helps, a better name could be "meta-parameter," because the parameter is akin to metadata about the model.**

The success of a model's prediction $g(x|\theta)$ depends on how well it agrees with the ground truth y. We need a way to measure the distance between these two vectors. For example, the L2-norm may be used to measure how close two vectors lie. The distance between the ground truth and prediction is called the *cost*.

The essence of a supervised machine learning algorithm is to figure out the parameters of a model that results in the least cost. Mathematically put, we are trying to look for a θ^* that minimizes the cost among all data points $x \in X$.

$\theta^* = \arg\min \mathrm{Cost}(\theta \mid X)$
where $Cost(\theta \mid X) = \sum_x dist(\, g(x \mid \theta) - f(x)\,)$
and *dist* is a distance metric such as the L2-norm

Clearly, brute forcing every possible combination of θs, also known as a *parameter-space*, will eventually find the optimal solution, but at an unacceptable runtime. A major study in machine learning is about writing algorithms that efficiently search through this parameter-space. Some of the first algorithms include *gradient descent*, *simulated annealing*, and *genetic algorithms*. TensorFlow automatically takes care of the low-level implementation details of these algorithms, so we won't get into them in too much detail.

Once the parameters are learned one way or another, you can finally evaluate the model to figure out how well the system captured patterns from the data. A rule of thumb is not to evaluate your model on the same data you used to train it. Use the majority of the data for training, and the remaining for testing. For example, if you have 100 labeled data, randomly select 70 of them to train a model, and reserve the other 30 to test it.

Why split the data?

If the 70-30 split seems odd to you, think about it like this. Let's say your Physics teacher gives you a practice exam and tells you the real exam will be no different. You might as well memorize the answers and earn a perfect score without actually understanding the concepts. Similarly, if we test our model on the training dataset, we're not doing ourselves any favors. We risk a false sense of security since the model may merely be memorizing the results. Now, where's the intelligence in that?

Instead of the 70-30 split, machine learning practitioners typically divided their dataset 60-20-20. Training consumes 60% of the dataset, and testing uses 20%, leaving the other 20% for what is called "validation," which will be explained in the next chapter.

1.5 Unsupervised Learning

Unsupervised learning is about modeling data that comes without corresponding labels or responses. The fact that we can make any conclusions at all on just raw data feels like magic. With enough data, it may be possible to find patterns and structure. Two of the most powerful tools that machine learning practitioners use to learn from data alone are *clustering* and *dimensionality reduction*.

Clustering is the process of splitting the data into individual buckets of similar items. In a sense, clustering is like classification of data without knowing any corresponding labels. For instance, when reorganizing a bookshelf, you likely place similar genres together, or maybe you group them by author's last name. One of the most popular clustering algorithms is *K-means*, which is a specific instance of a more powerful technique called the *E-M algorithm*.

Dimensionality reduction is about manipulating the data to view it under a much simpler perspective. It is the ML equivalent of the phrase, “Keep it simple, stupid!” For example, by getting rid of redundant features, we can explain the same data in a lower-dimensional space and see which features really matter. This simplification also helps in data visualization or preprocessing for performance efficiency. One of the earliest algorithms is Principle Component Analysis (PCA), and some newer ones include autoencoders, which we’ll cover in chapter 7.

1.6 Reinforcement Learning

Supervised and unsupervised learning seem to suggest that the existence of a teacher is all or nothing. But, there is a well-studied branch of machine learning where the environment acts as a teacher, providing hints as opposed to definite answers. The learning system receives feedback on its actions, with no concrete promise that it's progressing in the right direction.

Exploration vs. Exploitation is the heart of reinforcement learning

Imagine playing a video-game that you've never seen before. You click buttons on a controller and discover that a particular combination of strokes gradually increases your score. Brilliant, now you repeatedly exploit this finding in hopes of beating the high-score. In the back of your mind, you think to yourself that maybe there's a better combination of button-clicks that you're missing out on. Should you exploit your current best strategy, or risk exploring new options?

Unlike supervised learning, where training data is conveniently labeled by a "teacher," *reinforcement learning* trains on information gathered by observing how the environment reacts to actions. In other words, reinforcement learning is a type of machine learning that interacts with the environment to learn which combination of actions yields the most favorable results. Since we're already anthropomorphizing our algorithm by using the words "environment" and "action," scholars typically refer to the system as an autonomous "agent." Therefore, this type of machine learning naturally manifests itself into the domain of robotics.

To reason about agents in the environment, we introduce two new concepts: states and actions. The status of the world frozen at some particular time is called a *state*. An agent may perform one of many *actions* to change the current state. To drive an agent to perform actions, each state yields a corresponding *reward*. An agent eventually discovers the expected total reward of each state, called the *value* of a state.

Like any other machine learning system, performance improves with more data. In this case, the data is a history of previous experiences. In reinforcement learning, we do not know the final cost or reward of a series of actions until it's executed. These situations render traditional supervised learning ineffective. The only information an agent knows for certain is the cost of a series of actions that it has already taken, which is incomplete. The agent's goal is to find a sequence of actions that maximizes rewards.

EXERCISE Would you use supervised, unsupervised, or reinforcement learning to solve the following problems? (a) Organize various fruits in 3 baskets based on no other information. (b) Predict the weather based off sensor data. (c) Learn to play chess well after many trial-and-error attempts.

ANSWER (a) unsupervised, (b) supervised, (c) reinforcement

1.7 Existing Tools

One of the easiest ways to get started with machine learning and data analysis is through the *Scikit-learn* Python library (http://scikit-learn.org/). Python is a great language to prototype ideas that eventually become industry standard implementations. Some of the most enduring and successful Python libraries such as NumPy, SciPy, and matplotlib form the backbone of Scikit-learn. The tools are simple, making them easy to use.

However, Scikit-learn feels like assembly language because the library is relatively low-level. As a result, performing sophisticated algorithms can easily result in buggy code. Instead of interacting directly with Scikit-learn, using higher-level libraries such as TensorFlow, Theano, or Caffe offers a more robust setup while sacrificing some flexibility.

Hadoop or Apache Spark are some higher-level industry standard frameworks to deal with big data where the emphasis is parallelism and distributed computing. Commonly, the lower-level library used to interact with these parallel architectures is Apache *Mahout*, providing a complete Java interface.

Apart from TensorFlow, some of the most common machine learning libraries include Theano, Caffe, Torch, and Computational Graph Toolkit as shown in Figure 1.12.

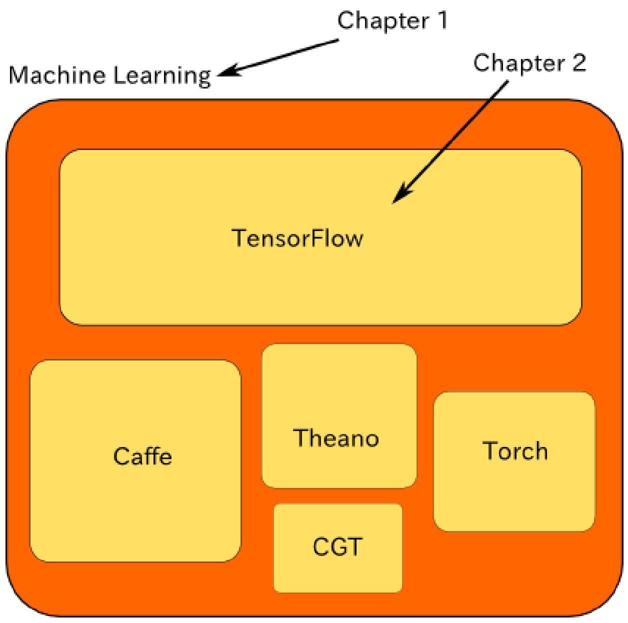


Figure 1.12 TensorFlow is one the most popular machine learning libraries. At the time of writing, it has more than twice as many favorites on GitHub than Caffe, the next most popular library. The size of the rectangles roughly corresponds to popularity on GitHub.

TensorFlow was released to the public in November 2015. Before that, the following were some of the most commonly used machine learning libraries:

1.7.1 Theano

Throughout the years, machine learning practitioners have already implemented many well-known neural networks in Theano. It uses a symbolic graph to decouple implementation from design. The programming environment is in Python, which makes it easy for a novice to jump in and give it a go regardless of platform. There is little support, however, for a low-level interface to make substantial customization. Moreover, there is a considerable overhead for rapid prototyping since it compiles code to binary every time, making a quick modification or debugging a burden.

1.7.2 Caffe

Caffe's primary interactions are easy to use because of its simple Python interface. For more complex algorithms or customized neural networks, one can also use the C++ interface. Since most platforms support C++, it is readily deployable. However, the purpose of Caffe is primarily for networks that deal with images. Text or time-series data will be unnatural to process though Caffe.

1.7.3 Torch

Lua is the programming language of choice for this framework. Since the Lua environment is foreign to most developers, there's a nontrivial risk associated with using Torch for machine learning projects. But on the bright side, Torch has strong support for optimization solvers, so you wouldn't need to reinvent the wheel.

1.7.4 Computational Graph Toolkit

A lab in University of California, Berkeley released Computational Graph Toolkit (CGT) for generalized graph operations, often used in machine learning. It supports some of the same features as Theano, but with an emphasis on parallelism. Unfortunately, documentation is relatively sparse compared to the other libraries, and the community is not as prevalent as that of Theano or Caffe.

1.8 TensorFlow

Google open-sourced their machine learning framework called TensorFlow in late 2015 under the Apache 2.0 license. Before that, it was used proprietarily by Google in its speech recognition, Search, Photos, and Gmail, among other applications.

A bit of history

A former scalable distributed training and learning system called DistBelief is the primary influence on TensorFlow's current implementation. Ever written a messy piece of code and wished you could start all over again? That's essentially the dynamic between DistBelief and TensorFlow.

The library is implemented in C++ and has a convenient Python API, as well a lesser appreciated C++ API. Because of the simpler dependencies, TensorFlow can be quickly deployed to various architectures.

Similar to Theano, computations are described as flowcharts, separating design from implementation. With little to no hassle, this dichotomy allows the same design to be implemented on not just large-scale training systems with thousands of GPUs, but also simply on mobile devices. The single system spans a broad range of platforms.

One of the fanciest properties of TensorFlow is its *automatic differentiation* capabilities. One can experiment with new networks without having to redefine many key calculations.

ASIDE Automatic differentiation makes it much easier to implement backpropagation, which is a computationally heavy calculation used in a branch of machine learning called neural networks. TensorFlow hides the nitty-gritty details of backpropagation so that you can focus on the bigger picture. Chapter 7 covers an introduction to neural networks with TensorFlow.

All the mathematics is abstracted away and unfolded under the hood. It's like using WolframAlpha for a Calculus problem set.

Another feature of this library is its interactive visualization environment called TensorBoard. This tool shows a flowchart of how data transforms, displays summary logs over time, as well as traces performance. Figure 1.13 shows an example of what TensorBoard looks like when in use. The next chapter will cover using it in greater detail.

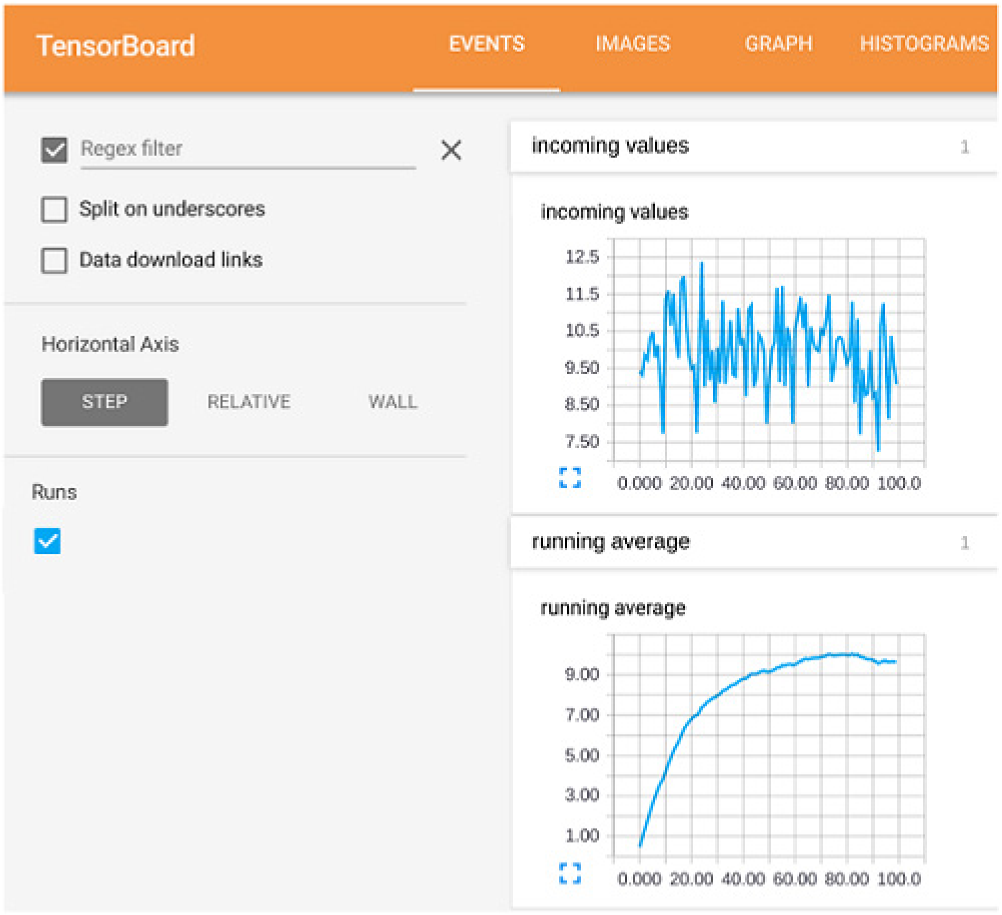


Figure 1.13 Example of TensorBoard in action

Unlike Theano, prototyping in TensorFlow is much faster (code initiates in a matter of seconds as opposed to minutes) because much of the operations come pre-compiled. It becomes easy to debug code due to subgraph execution. What that means is that an entire segment of computation can be reused without recalculation.

Because TensorFlow is not only about neural networks, it also has out-of-the-box matrix computation and manipulation tools. Most libraries such as Torch or Caffe are designed solely for deep neural networks, but TensorFlow is more flexible.

The library is well documented and is officially supported by Google. Machine learning is a sophisticated topic, so having an exceptionally well-reputed company behind TensorFlow is comforting.

1.9 Overview of future chapters

Chapter 2 demonstrates how to use various components of TensorFlow. Chapters 3-6 are about how to implement classic machine learning algorithms in TensorFlow, whereas chapters 7-11 cover algorithms based on neural networks (see figure 1.14). The algorithms solve a wide variety of problems such as prediction, classification, clustering, dimensionality reduction, and planning.

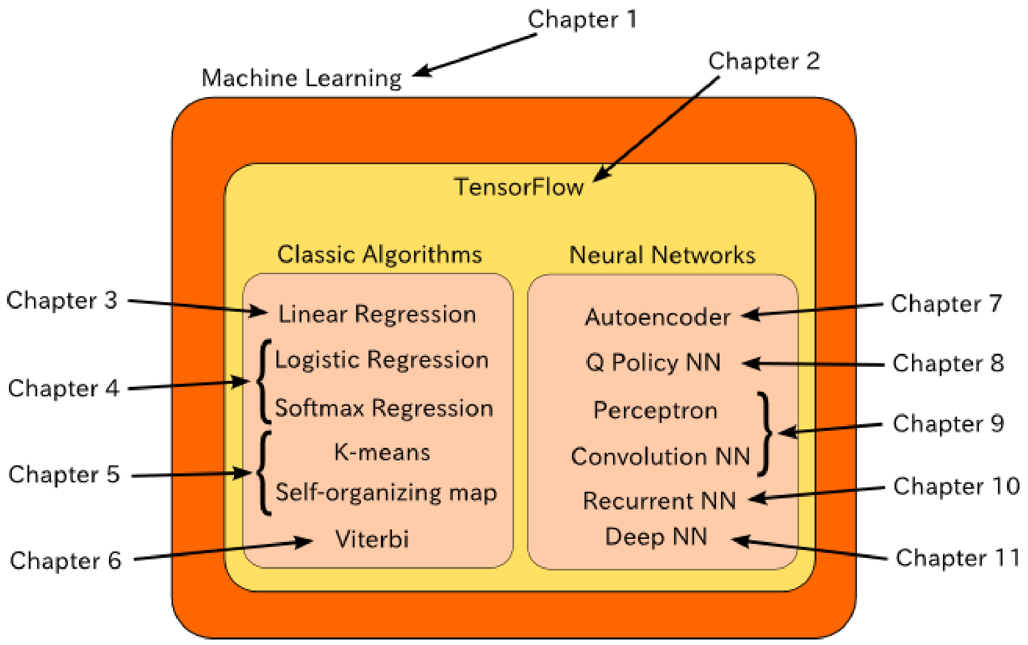


Figure 1.14 The technical chapters are divided into two categories of algorithms: (1) classic algorithms, and (2) neural networks.

There are many algorithms to solve the same real world problem, and many real-world problems that are solved by the same algorithm, but table 1.1 covers the ones laid out in this book.

Table 1.1 Many real-world problems can be solved using the corresponding algorithm found in their respective chapters.

Real world problem	Algorithm	Chapter
Predicting trends, fitting a curve to data points, describing relationships between variables	Linear regression	3
Classifying data into two categories, finding the best way to split a dataset	Logistic regression	4
Classifying data into multiple categories	Softmax regression	4
Revealing hidden causes of observations, finding the most likely hidden reason for a series of outcomes	Hidden Markov Model (Viterbi)	5
Clustering data into a fixed number of categories, automatically partitioning data points into separate classes	K-means	6
Clustering data into arbitrary categories, visualizing high-dimensional data into a lower-dimensional embedding	Self-organizing map	6
Reducing dimensionality of data, learning latent variables responsible for high-dimensional data	Autoencoder	7
Planning actions in an environment using neural networks (reinforcement learning)	Q Policy neural network	8
Classifying data using supervised neural networks	Perceptron	9
Classifying real-world images using supervised neural networks	Convolution neural network	9
Producing patterns that match observations using neural networks	Recurrent neural network	10

1.10 Summary

You learned quite a bit about machine learning in this chapter, including the following:

- Machine learning is about using examples to develop an expert system that can make useful statements about new inputs.
- A key property of ML is that performance tends to improve with more training data.
- Over the years, scholars have crafted three major archetypes that most problems fit: supervised learning, unsupervised learning, and reinforcement learning.
- After a real-world problem is formulated in a machine learning perspective, several algorithms become available. Out of the many software libraries and frameworks to accomplish an implementation, we chose TensorFlow as our silver bullet. Developed by

Google and supported by its flourishing community, TensorFlow gives us a way to easily implement industry standard code.

In the next chapter, we'll get our hands dirty with TensorFlow to get comfortable with using the library.