Math and Architectures of Deep Learning

Krishnendu Chaudhury
with Ananya Ashok
Sujay Narumanchi
Devashish Shankar

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Math and Architectures of Deep Learning
Version 3

For more information on this and other Manning titles go to
manning.com

welcome

Dear Reader,

Welcome to Manning Early Access Program (MEAP) for "*Math and Architectures of Deep Learning*". This membership will give you access to the developing manuscript along with the resources which includes fully functional python/PyTorch code downloadable and executable via Jupyter-notebook.

Deep learning is a complex subject. On one hand, it is deeply theoretical with extensive mathematical backing. Indeed, without a good intuitive understanding of the mathematical underpinnings, one is doomed to merely running off the shelf pre-packaged models without understanding them fully. These models often do not lend themselves well to the exact problem one needs to solve and one is helpless if any change or re-architecting is necessary. On the other hand, deep learning is also intensely practical requiring significant Python programming skills on new platforms like Tensorflow and PyTorch. Failure to master those leaves one unable to solve any real problem.

This author feels that there is a dearth of books that addresses both of these aspects of the subject in a connected fashion. That is what has led to the genesis of this book.

The author will feel justified in his efforts if these pages help the reader to become a successful exponent in the art and science of deep learning.

Sincerely,

—Krishnendu Chaudhury

brief contents

Introduction:

Importance of mathematical principles underlying deep learning

Are you the type of person who wants to know *why* and *how* things work? Instead of feeling satisfied, even grateful, that a tool is solving the problem at hand - you try to understand *what the tool is really doing*, *why it is behaving in a certain way* and *whether it will work under changed circumstances*? If yes, you have my sympathies - life won't be peaceful for you. You also have my best wishes - these pages are dedicated to you.

The internet abounds with various pre-built deep learning models and training systems that hardly require one to understand the underlying principles. But practical problems often do not quite fit any of the publicly available models. These situations call for development of a custom model architecture. Developing these require one to understand the mathematical underpinnings of optimization and machine learning.

A relevant question can be asked here. These (deep learning and computer vision) are, after all, very practical subjects. Is the mathematics really necessary? Shouldn't one rather spend the time learning, say the python nuances of deep learning? Well, yes and no. Programming skills, in particular python, are mandatory. But without an intuitive understanding of the mathematics, the *how* and *why* and *can I re-purpose this model* will not be visible to the practitioner. In short, mathematics allows you to see the abstractions behind the implementation.

In many ways, the ability to form abstractions is the essence of higher intelligence. It is abstraction that enabled early humans to divine a digging and defending tool in what was merely a sharply pointed stone to other animals. The abstraction of the description of where something is with respect to some other thing fixed in the environment (aka coordinate systems and vectors) has done wonders to human civilization. Mathematics is the language for abstractions, the most precise, succinct and unambiguous known to humankind.

Having (hopefully) made a case for studying the underlying mathematical principles of deep learning and computer vision, we would hasten to add that mathematical rigor is *not* the goal here. Rather, the goal is to provide mathematical, in particular, geometrical, insights that make the subject more intuitive and indeed less of a black magic. At the same time, we will

provide *python coding exercises* and *visualization aids* throughout - thus these pages can be regarded as learning mathematical foundations of deep learning via geometrical examples and python exercises.

Fully functional detailed code backing the theory discussed in the book is provided via Jupyter-notebook at https://github.com/krishnonwork/mathematical-methods-in-deep-learning-ipython. Also, important snippets from the full code have been inserted in the body of the main book for illustration purposes - these are not fully functional.

Mastery over the material presented in this book will enable the reader to

- Understand state-of-the-art deep learning research papers. In fact, the book will attempt to provide in-depth, intuitive understandings of all the seminal papers of the day.
- Study and understand a deep learning code-base
- Directly lift off code snippets from the book to use in their tasks
- Ace an interview for ML engineer/scientist
- Determine whether a real life problem is amenable to machine/deep learning
- Troubleshoot neural network quality issues
- Identify the right neural network architecture to solve a real life problem
- Quickly implement a prototype architecture and train a deep learning model for some real life problem

A word of caution. We will often start at the basics but quickly move deeper. It is kind of important to read individual pieces end to end, even if one is familiar with the material presented at the beginning.

Finally, the ultimate justification of an intellectual endeavor is that one had fun pursuing it.

So, the author would consider himself successful if you enjoy reading these lines.

1

An overview of machine learning and deep learning

Deep learning has transformed computer vision, natural language and speech processing in particular and artificial intelligence in general. From a bag of semi-discordant tricks, none of which worked satisfactorily on a real life problem, artificial intelligence has become a formidable tool to solve real problems faced by industry, at scale. This is nothing short of a revolution going on under our very noses. If one wants to lead the curve of this revolution, it is imperative to understand the underlying principles and abstractions, rather than simply memorizing the "how to" steps of some hands on guide. This is where the mathematics comes in.

In this first chapter we will give an overview of deep learning. This will require us to use some concepts that have been explained in subsequent chapters. The reader should not worry if there are some open questions at the end of this chapter. This chapter is aimed at orienting one's mind towards this difficult subject. As individual concepts get clearer in subsequent chapters, the reader should consider coming back and giving this chapter a re-read.

1.1 A first look at machine/deep learning - a paradigm shift in computation

Making decisions and/or predictions is a central requirement of life. This essentially involves taking in a set of sensory or knowledge inputs and generating decisions or estimates by processing them.

For instance, a cat's brain is often trying to choose between the following options: *run away* from the object in front vs *ignore* the object in front vs *approach* the object in front and purr. It makes that decision by processing sensory inputs, like perceived *hardness* of the object in front, perceived *sharpness* of the object in front, etc. This is an instance of *classification* problem where the output is one out of a set of possible classes.

Some other examples of classification problem in life:

- buy *vs* hold *vs* sell *a certain stock, from inputs like* price history of this stock*,* change in price of this stock in recent times
- *object recognition (from an image), e.g.,:*
    - is this a car or a giraffe
    - is this a human or a non-human
    - is this an inanimate object or a living object
    - face recognition - is this Tom or Dick or Mary or Einstein or Messi
- action recognition from video, e.g.,:
    - is this person running or not running
    - is this person picking something up or not
    - is this person doing something violent or not
- Natural Language Processing aka NLP from digital documents, e.g.,:
    - does this news article belong to the realm of politics or sports
    - does this query phrase match a particular article in the archive

    etc.

Sometimes life requires a *quantitative* estimation as opposed to classification. *A* lion brain needs to estimate what should be the length of a jump so as to land on the top of its prey, by processing inputs like speed of prey, distance to prey etc. Another instance of quantitative estimation is to estimate house price, based on inputs like current income, crime statistics for the neighborhood etc.

Some other examples of quantitative estimations required by life

- object localization from an image: identifying the rectangle bounding the location of an object
- stock price prediction from historical stock prices and other world events
- similarity score between a pair of documents

Sometimes, a classification output can be generated from a quantitative estimate. For instance, the cat brain described above, can combine the inputs (hardness, sharpness etc) to generate a quantitative threat score. If that threat score is high, the cat runs away. If the threat score is near zero, the cat ignores the object in front. If threat score is negative, the cat approaches the object in front and purrs.

Many of these examples are pictorially depicted in Fig 1.1.

In each of these instances, there is a machine - viz., brain - that transforms sensory or knowledge inputs to decisions or quantitative estimates. The goal of machine learning is to emulate that machine.

One must note that machine learning has a long way to go before it can catch up with the human brain. The human brain can single handedly deal with thousands, if not millions, of such problems. On the other hand, at its present state of development, machine learning can hardly create a single general purpose machine that makes a wide variety of decisions and estimates. We are mostly trying to make separate machines to solve individual tasks (stock picker, car recognizer etc).

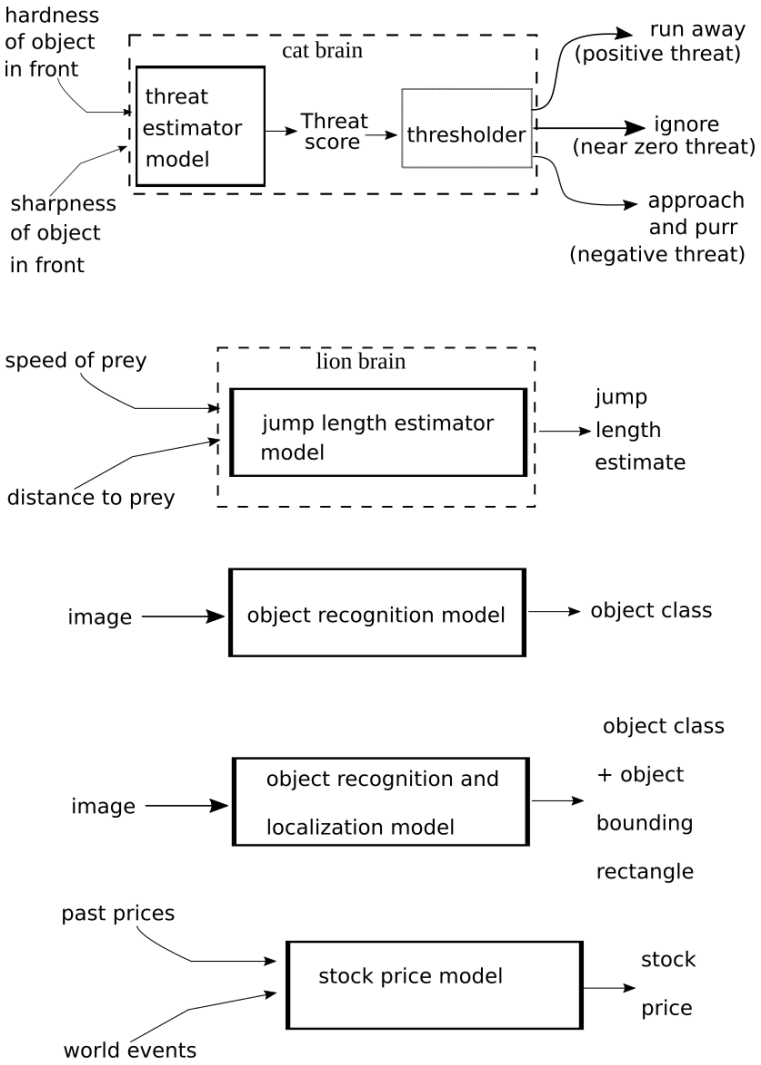


Figure 1.1: Examples of Decision Making and Quantitative Estimations in Life

At this point, one might ask, wait, converting inputs to outputs - isn't that exactly what computers have been doing for last thirty or more years? What is this paradigm shift I am hearing about? The answer: it *is* a paradigm shift because we do not provide a step by step instruction set - viz., a program - to the machine to convert the input to output. Instead, we develop a mathematical model for the problem.

Let us illustrate the idea with an example. For the sake of simplicity and concreteness, we will consider a hypothetical cat brain which needs to make only one decision in life - whether to *run away from the object in front* or *ignore it* or *approach and purr*. This decision, then, is

the output of the model we will discuss. And, in this toy example, the decision is made based on only two quantitative inputs (aka features), perceived hardness of the object in front and its perceived sharpness. (as depicted in Fig 1.1). We do *not* provide any step by step instruction such as *"if sharpness greater than some threshold then run away"* etc. Instead, we try to identify a *parameterized* function that takes the input and converts it to the desired decision or estimate. The simplest such function is a *weighted sum of inputs*:

$$y\,(hardness, sharpness) = w_0 \times hardness + w_1 \times sharpness + b$$

The weights w_0,w_1 and the bias *b* are the parameters of the function. The output y can be interpreted as a threat score. If the threat score exceeds a threshold the cat runs away. If it is close to 0 the cat ignores. If the threat score is negative the cat approaches and purrs. For more complex tasks, we will use more sophisticated functions.

Note that the weights are not known at first, we need to estimate them. This is done through a process called *model training*.

Overall, solving a problem via machine learning has following stages:

- We first design a parameterized model function (e.g., weighted sum) with unknown parameters (weights). This constitutes the *model architecture*. Choosing the right model architecture is where the expertise of the machine learning engineer comes into play.
- Then we estimate the weights via model training.
- Once the weights are estimated, we have a complete *model*. This model can take arbitrary inputs not necessarily seen before and generate outputs. The process, where a trained model processes an arbitrary real life input and emits an output is called *inferencing*.

In the most popular variety of machine learning, called *supervised learning*, we prepare the training data before we commence training. Training data comprises *example input items, each with its corresponding desired output*.[1] Training data is often created manually, i.e., a human goes over every single input item and produces the desired output (aka target output). It is usually the most arduous part of doing machine learning.

For instance, in our hypothetical cat brain example, some possible training data items are

$$\text{input: } (hardness = 0.01, sharpness = 0.02) \rightarrow \text{threat} = -0.90 \rightarrow decision\text{: "approach and purr"}$$

$$\text{input: } (hardness = 0.50, sharpness = 0.60) \rightarrow \text{threat} = 0.01 \rightarrow decision\text{: "ignore"}$$

$$\text{input: } (hardness = 0.99, sharpness = 0.97) \rightarrow \text{threat} = 0.90 \rightarrow decision\text{: "run away"}$$

where the input values of hardness and sharpness are assumed to lie between 0 and 1.

[1] If you have some experience with machine learning, you will realize that I am talking about "supervised" learning here. There are also machines that do not need known outputs to learn - the so called "unsupervised" machines – we will talk about them later.

What exactly happens during training? Answer: we iteratively process the input training data items. For each input item, we know the desired (aka target) output. On each iteration, we adjust the model weight values in a way that the output of the model function on that specific input item gets at least a little bit closer to the corresponding target output. For instance, suppose at a given iteration, the weight values are $w_0 = 20$ and $w_1 = 10$ and $b = 50$. On the input ($hardness = 0.01$, $sharpness = 0.02$), we get an output threat score $y = 50.3$ which is quite different from the desired $y = -0.9$. We will adjust the weights, for instance reduce the bias - so $w_0 = 20$ and $w_1 = 10$ and $b = 40$. The corresponding threat score $y = 40.3$ is still nowhere near the desired value, but it has moved closer. After doing this on many training data items, the weights would start approaching their ideal values. Note that how to identify the adjustments to the weight values is not discussed here. It needs somewhat deeper math and will be discussed later.

As stated above, this *process of iteratively tuning weights is called training or learning*. At the beginning of learning, the weights have random values, so the machine outputs often do not match desired outputs. But with time, more training iterations happen and the machine "learns" to generate the correct output. That is when the model is ready for deployment in real world. Given arbitrary input, the model will (hopefully) emit something close to the desired output during inferencing.

Come to think of it, that is probably how living brains work. They contain equivalents of mathematical models for various tasks. Here, the weights are the strengths of the connections (aka synapses) between the different neurons in the brain. In the beginning, the parameters are untuned, the brain repeatedly makes mistakes. E.g., a baby's brain often makes mistake in identifying edible objects - anybody who has had a child will know what we are talking about. But each example tunes the parameters (eating green and white rectangular things with $ sign invites much scolding - should not eat them in future etc). Eventually this machine tunes its parameters to yield better results.

One subtle point should be noted here. During training, the machine is tuning its parameters so that it produces the desired outcome - *on the training data input only*. Of course, it sees only a small fraction of all possible inputs during training - we are *not* building a lookup table from known inputs to known outputs here. Hence, when this machine gets released in the world, it mostly runs on input data it has never seen before. What guarantee do we have that it will generate the right outcome on never before seen data? Frankly, there is no guarantee. Only, in most real life problems, the inputs are not really random. They have a pattern. Hopefully, the machine will see enough during training to capture that pattern. Then, its output on unseen input will be close to desired value. The closer the distribution of the the training data is to real life, likelier that becomes.

1.2 A Function Approximation View of Machine Learning: Models and their Training

As stated in section 1.1, to create a brain-like machine that makes classifications or estimations, we have to find a mathematical function (model) that transforms inputs into corresponding desired outputs. Sadly however, in typical real life situations, we do not know that transformation function. For instance, we do not know the function that takes in past prices, world events etc and estimates the future price of a stock - something that stops us

from building a stock price estimator and getting rich. All we have is the training data - a set of inputs on which the output is known. How do we proceed then? Answer, we will try to model the unknown function. This means, we will create a function that will be a proxy or surrogate to the unknown function. Viewed in this way, machine learning is nothing but function approximation - we are simply trying to approximate the unknown classification or estimation function.

Let us briefly recapitulate the main ideas from the previous section. In machine learning, we try to solve problems that can be abstractly viewed as transforming a set of inputs to an output. The output is either a class or an estimated value. Since we do not know the true transformation function, we try to come up with a model function. We start by designing – using our physical understanding of the problem - a model function with tunable parameter values that could serve as a proxy for the true function. This is the *model architecture* and the tunable 7 parameters are also known as weights. The simplest model architecture is one where the output is a weighted sum of the input values. Determining the model architecture does not fully determine the model - we still need to determine the actual parameter values (weights). That is where *training* comes in. During training, we find an optimal set of weights that would transform the training inputs to outputs that match the corresponding training outputs as closely as possible.

Then we deploy this machine in the world - now its weights are estimated and the function is fully determined - on any input, it simply applies the function and generates an output. This is called *inferencing*. Of course, training inputs are only a fraction of all possible inputs, so there is no guarantee that inferencing will yield a desired result on all real inputs. The success of the model depends on the appropriateness of the chosen model architecture and the quality and quantity of training data.

In this context, the author would like to note that after mastering machine learning, the biggest struggle faced by a practitioner turns out to be procurement of training data. It is common practice, when one can afford it, to use humans to hand generate the outputs corresponding to the training data inputs (these target outputs are sometimes referred to as ground truth). This process, known as human labeling or human curation, involves an army of human beings looking at a substantial number of training data inputs and producing the corresponding ground truth output. For some well researched problems, one maybe lucky enough to get training data on the internet, else it becomes a daunting challenge. More on this later.

Now, let us study the process of model building with a concrete example, the cat brain machine shown in Fig 1.1.

1.3 A simple machine learning model - the cat brain

[2] For the sake of simplicity and concreteness, we will deal with a hypothetical cat which needs to make only one decision in life - whether to run away from the object in front or ignore it or

[2] This chapter is a lightweight overview of machine/deep learning. As such, it mildly relies upon mathematical concepts that we will introduce later. The reader is encouraged to read this chapter now, nonetheless, and perhaps re-read after the chapters on vectors and matrices have been digested.

approach and purr. And it makes this decision based on only two quantitative inputs pertaining to the object in front of the cat (shown in Fig 1.1).

INPUT FEATURES

i) x_0 signifying **Hardness**

ii) x_1 signifying **Sharpness** .

Without loss of generality, we can *normalize* the inputs. This is a pretty popular trick, whereby the input values ranging between a minimum possible value v_{min} and a maximum possible value v_{max} are transformed to values between 0 and 1. To transform an arbitrary input value v to a normalized value v_{norm} we use the formula

$$v_{norm} = \frac{(v - v_{min})}{(v_{max} - v_{min})} \tag{1.1}$$

In mathematical parlance, transformation via equation 1.1, $v \in [v_{min}, v_{max}] \rightarrow v_{norm} \in [0, 1]$ maps the values v from the input domain $[v_{min}, v_{max}]$ to the output values v_{norm} in the range [0, 1].

A 2 element vector $\vec{x} = \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} \in [0.1]^2$ represents a single input instance succinctly.

OUTPUT DECISIONS

The final output is multi-class, which can take one of three possible values

i) **0**: Implying run away from the object in front
ii) **1**: Implying ignore the object in front
iii) **2**: Implying approach and purr.

It is possible in machine learning to compute the class directly. However, in this example we will have our model estimate a **threat score**. It is interpreted as follows:

i) threat high positive – run away
ii) threat near zero - ignore
iii) threat high negative - approach and purr (negative threat is attractive) .

We can make a final multi-class run/ignore/approach decision based on threat score by comparing the threat score y against a threshold δ as follows

$$y \begin{cases} > \delta \rightarrow \text{high threat, run away} \\ >= -\delta \text{ and } <= \delta \rightarrow \text{threat close to zero, ignore} \\ < -\delta \rightarrow \text{negative threat, approach and purr} \end{cases} \tag{1.2}$$

MODEL ESTIMATION

Now for the all important step. We need to estimate the function which transforms the input vector to the output. With slight abuse of terms, we will denote this function as well as the output by y. In mathematical notation, we want to estimate $y(\vec{x})$.

Of course, we do not know the ideal function. We will try to estimate this unknown function from the training data. This is accomplished in two steps:

1. **Model Architecture Selection**: Designing a parameterized function that we expect is a good proxy or surrogate for the unknown ideal function.
2. **Training**: Estimating the parameters of that chosen function such that the outputs on training inputs match correspond outputs as closely as possible.

MODEL ARCHITECTURE SELECTION

This is the step where various machine learning approaches differ from one another. In this toy cat brain example, we will use the simplest possible model. Our model has 3 parameters, w_0, w_1, b - they can be represented compactly with a single 2 element vector $\vec{w} = \begin{bmatrix} w_0 \\ w_1 \end{bmatrix} \in \mathbb{R}^2$ and and a constant bias $b \in \mathbb{R}$ (here $\mathbb{R}$ denotes the set of all real numbers, $\mathbb{R}^2$ denotes the set of 2D vectors with both elements real, etc). It emits the threat score, y, which is computed as

$$y(x_0, x_1) = w_0 x_0 + w_1 x_1 + b = \begin{bmatrix} w_0 & w_1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} + b = \vec{w}^T \vec{x} + b \tag{1.3}$$

Note that b is a slightly special parameter. It is a constant, that does not get multiplied with any of the inputs. It is common practice in machine learning to refer to it as *bias* while the other parameters that get multiplied with inputs as weights.

MODEL TRAINING

Once the model architecture is chosen, we know the exact parametric function we are going to use to model the unknown function $y(\vec{x})$ that transforms inputs to outputs. We still need to estimate the function's parameters. Thus, we have a function with unknown parameters and the parameters are to be estimated from a set of inputs with known outputs (training data). We will choose the parameters so that the outputs on the training data inputs match the corresponding outputs as closely as possible.

It should be noted that this problem, has been studied by mathematicians and known as a *function fitting* problem in mathematics. What changed with the advent of machine learning however is the sheer scale. In machine learning, we deal with training data comprising millions and millions of items. This changed the philosophy of the solution. Mathematicians used a "closed-form solution", where the parameters are estimated by directly solving equations involving *all* the training data items together. In machine learning, one goes for iterative solutions, where one deals with a *few, perhaps a single,* training data item at a time. In the iterative solution, there is no need to hold the entire training data in the computer's memory. One simply loads small portions of it at a time and deals with only that portion. We will exemplify this with our cat brain example.

Concretely, the goal of the training process is to estimate the parameters w_0, w_1, b or equivalently the vector $\vec{w}$ along with constant b from equation 1.3 in such a way that the output y (x_0, x_1) on training data input (x_0, x_1) matches the corresponding known training data outputs (aka ground truth or GT) as much as possible.

Let the training data comprise N + 1 inputs $\vec{x}^{(0)}$, $\vec{x}^{(1)}$, …, $\vec{x}^{(N)}$. Here each $\vec{x}^{(i)}$ is a 2 × 1 vector denoting a single training data input instance. The corresponding desired threat values (outputs) are $y_{gt}^{(0)}$, $y_{gt}^{(1)}$, $\cdots y_{gt}^{(N)}$, say (here the subscript gt denotes ground truth). Equivalently, we can say training data comprises N + 1 (input, output) pairs:

$$\left(\vec{x}^{(0)}, y_{gt}^{(0)}\right), \left(\vec{x}^{(1)}, y_{gt}^{(1)}\right) \cdots \left(\vec{x}^{(N)}, y_{gt}^{(N)}\right)$$

Supposing $\vec{w}$ denotes the (as yet unknown) optimal parameters for the model. Then, given an arbitrary input $\vec{x}$ the machine will estimate a threat value of $y_{predicted} = \vec{w}^T\vec{x} + b$. On the i^{th} training data pair, ($\vec{x}^{(i)}$, $y_{gt}^{(i)}$) the machine will estimate

$$y_{predicted}^{(i)} = \vec{w}^T\vec{x}^{(i)} + b$$

while the desired output is $y_{gt}^{(i)}$. Thus the squared error (aka loss) made by the machine on the i^{th} training data instance is[3]

$$e_i^2 = \left(y_{predicted}^{(i)} - y_{gt}^{(i)}\right)^2$$

The overall loss on the entire training data set is obtained by adding the loss from each individual training data instance

$$E^2 = \sum_{i=0}^{i=N} e_i^2 = \sum_{i=0}^{i=N} \left(y_{predicted}^{(i)} - y_{gt}^{(i)}\right)^2 = \sum_{i=0}^{i=N} \left(\vec{w}^T\vec{x}_i + b - y_{gt}^{(i)}\right)^2$$

The goal of training is to find the set of model parameters (aka weights), $\vec{w}$, that minimizes the total error E. Exactly how we do this will be described later. In most cases, it is not possible to come up with a closed-form solution for the optimal $\vec{w}$, b.

Instead, we take an iterative approach depicted in Algorithm 1. In algorithm 1, we start with random parameter values and keep tuning parameters so that the total error goes down at least a little bit. Keep doing this until the error becomes sufficiently small.

Algorithm 1 Training a supervised model

Initialize parameters $\vec{w}$, b with random values

▷iterate while error not small enough

While $\left(E^2 = \sum_{i=0}^{i=N} \left(\vec{w}^T\vec{x}_i + b - y_{gt}^{(i)}\right)^2 > threshold\right)$ **do**

▷ iterate over all training data instances

for $\forall i \in [0, N]$ **do**

[3] In this context, it should be noted that it is a common practice to square the error/loss to make it sign independent. If we desired an output of, say 10, we are equally happy/unhappy if the output is 9.5 or 10.5. Thus, error of +5 or – 5 is effectively the same, hence we make the error sign independent.

▷ details provided in section 3.3 after gradients are introduced
Adjust $\vec{w}$, b so that E^2 is reduced
end for
end while
▷ remember the final parameter values as optimal
$\vec{w}_* \leftarrow \vec{w}\ b_* \leftarrow b$

In a purely mathematical sense, one continues the iterations until the error is minimal. But in practice, one often stops when the results are accurate enough for the problem being solved. It is worth re-emphasizing that error here refers only to error on training data.

INFERENCING

Finally, a trained machine (with optimal parameters $\vec{w}_*$, b_* is deployed in the world. It will receive new inputs $\vec{x}$ and will infer $y_{predicted}\ (\vec{x}) = \vec{w}_*{}^T \vec{x} + b_*$. Classification will happen by thresholding $y_{predicted}$ as shown in equation 1.2.

1.4 Geometrical View of Machine Learning

Each input to the cat's brain model is an array of 2 numbers: x_0 (signifying hardness of the object), x_1 (signifying sharpness of the object) or equivalently a 2×1 vector $\vec{x}$. A good mental picture here is to think of the input as a point in a high dimensional space. The input space is often called the feature space - a space where all the characteristic features to be examined by the model are represented. The feature space dimension is two here but in real life problems it will be in hundreds or thousands or more. The exact dimensionality of the input changes from problem to problem, but the intuition that it is a point remains.

The output y should also be viewed as a point in another high dimensional space. In this toy problem the dimensionality of the output space is 1, but in real problems it will be higher. Typically, however, number of output dimensions is much smaller than the number of input dimensions.

Geometrically speaking, a machine learning model essentially maps a point in the feature space to a point in the output space. It is expected that the classification or estimation job to be performed by the model is easier in the output space than the feature space. In particular, *for a classification job, input points belonging to separate classes are expected to map to separate clusters in output space*.

Let us continue with our example cat's brain model to illustrate the idea. As stated earlier, our feature space is $2\mathrm{D}$, with two coordinate axes X_0 signifying hardness and X_1 signifying sharpness[4]. Individual points in this $2\mathrm{D}$ space will be denoted by coordinate values (x_0, x_1), in

[4] We use X_0, X_1 as coordinate symbols instead of the more familiar X, Y so as not to run out of symbols when going to higher dimensional spaces.

lower case. This is depicted in Fig 1.2. As shown in the diagram, a good way to model the threat score is to measure distance from line $x_0 + x_1 = 1$.

From coordinate geometry, in a 2D space with coordinate axes X_0 and X_1, the signed distance of a point (a, b) from the line $x_0 + x_1 = 1$ is $y = \frac{a+b-1}{\sqrt{2}}$. Examining the sign of *y* we can determine which side of the separator line the input point belongs to.

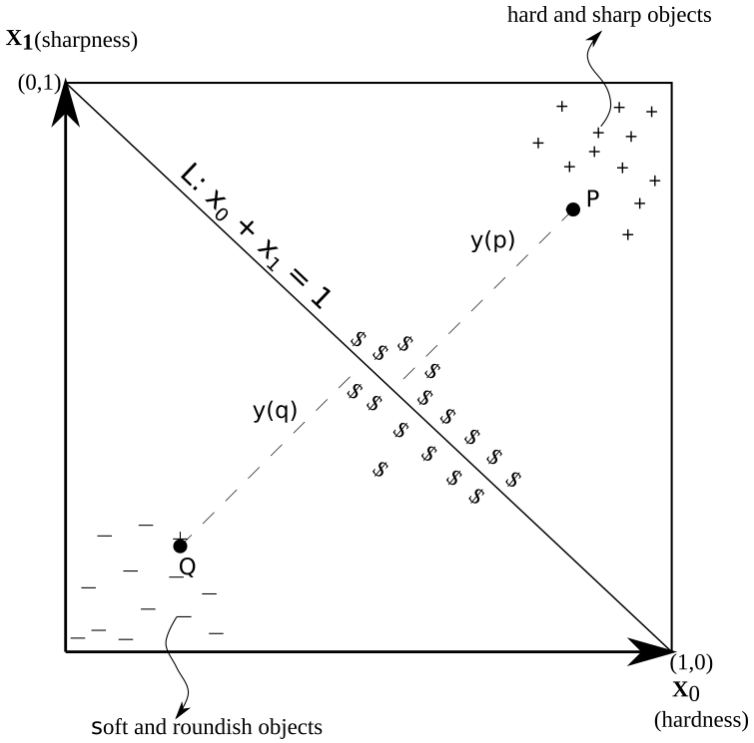


Figure 1.2: Geometrical View of Machine Learning: 2D input point space for cat brain model. The bottom left corner shows low hardness and low sharpness objects ('-' signs) while top right corner shows high hardness and high sharpness objects ('+' signs). The intermediate values are near the diagonal ('$' signs). In this simple situation, mere observation tells us that the threat score can be proxied by the signed distance, *y*, from the diagonal line $x_0 + x_1 - 1 = 0$. One can make the run/ignore/approach decision by thresholding *y*. Values close to zero imply ignore, positive values imply run away and negative values imply approach and purr. From high school geometry, the distance of an arbitrary input point $(x_0 = a, x_1 = b)$ from line $x_0 + x_1 - 1 = 0$ is $\frac{a+b-1}{\sqrt{2}}$. Thus, the function $y(x_0, x_1) = \frac{x_0+x_1-1}{\sqrt{2}}$ is a possible model for the cat brain threat estimator function. Training should converge to $w_0 = \frac{1}{\sqrt{2}}$, $w_1 = \frac{1}{\sqrt{2}}$ and $b = -\frac{1}{\sqrt{2}}$.

Thus, our simplified cat brain threat score model is

$$y(x_0, x_1) = \frac{1}{\sqrt{2}}x_0 + \frac{1}{\sqrt{2}}x_1 - \frac{1}{\sqrt{2}} \tag{1.4}$$

It maps the 2*D* input points, signifying hardness and sharpness of the object in front, to a 1*D* value corresponding to the signed distance from a separator line. This distance, physically interpretable as threat score, makes it possible to separate the classes (negative threat, neutral, positive threat) via thresholding as shown in equation 1.2. The separate classes form distinct clusters in the output space, depicted by $+$, $-$ and $\$$ signs in the output space. Low values of inputs produce negative threats (the cat will approach and purr), e.g., $y\ (0,\ 0) = -1/\sqrt{2}$.. High values of inputs produce high threat (cat will run away), e.g., $y\ (1,\ 1) = -1/\sqrt{2}$. Medium values of input produce near zero threat (cat will ignore), e.g., $y\ (0.5,\ 0.5) = 0$. Of course, because the problem is so simple, here we could come up with the model parameters via simple observation. In real life situations, this will need training.

The geometric view holds in higher dimensions too. In general, a n-dimensional input vector $\vec{x}$ is mapped to a m-dimensional output vector (usually $m < n$) in such a way that the problem becomes much simpler in the output space. An example with 3D feature space is shown in Figure 1.3.

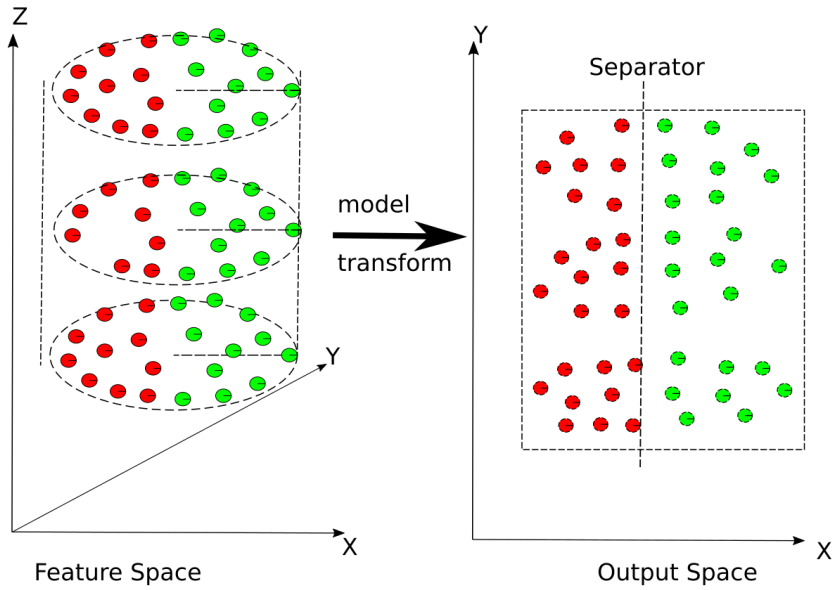


Figure 1.3: Geometrical View of Machine Learning: A model maps the points from input (feature) space to an output space where it is easier to separate the classes. For instance, in this figure, input feature points belonging to two classes, red and green, are distributed over the volume of a cylinder in a 3D feature space. The model unfurls the cylinder into a rectangle. The feature points get mapped onto a 2D planar output space where the two classes can be discriminated with a simple linear separator.

1.5 Regression vs Classification in Machine Learning

As briefly outlined in section 1.1, there are two types of machine learning models: *regressors* and *classifiers*.

In a *regressor*, the model tries to emit a desired value given a specific input. For instance, the first stage (threat score estimator) of the cat brain model in section 1.3 is a regressor model.

Classifiers on the other hand have a set of pre-specified classes. Given a specific input, they try to emit the *class* to which the input belongs. For instance, the full cat brain model has 3 classes: (i) run away (ii) ignore (iii) approach and purr. Thus, it takes an input (hardness and sharpness values) and emits an output decision (aka class).

In this example, we convert a regressor into a classifier by thresholding the output of the regressor (see equation 1.2. It is also possible to create models that directly output the class without having an intervening regressor.

1.6 Linear vs Nonlinear Models

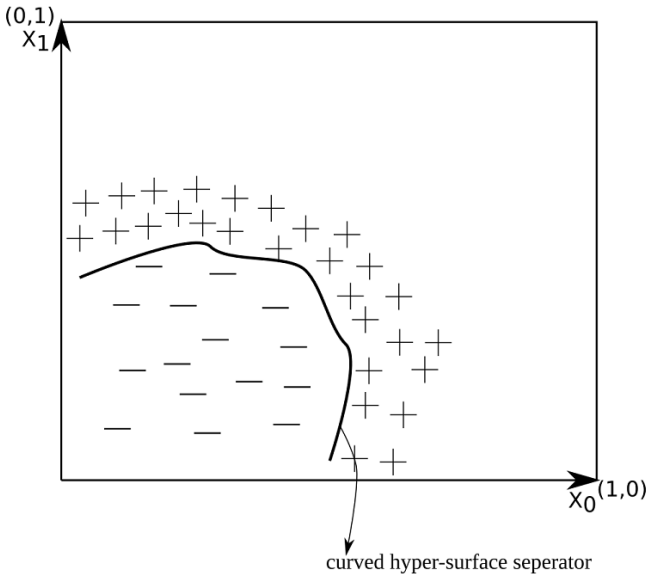


Figure 1.4: The two classes (indicated by '+' and '-') can not be separated by a line. Curved separator needed. In 3D, this is equivalent to saying no plane can separate the surfaces, a curved surface is necessary. In still higher dimensional spaces, this is equivalent to saying no hyper-plane can separate the classes. A curved hyper-surface is needed.

In Fig. 1.2 we faced a rather simple situation where the classes could be separated by a line (hyper-plane in higher dimensional surfaces). This often does not happen in real life. What if the points belonging to different classes are as shown in Fig. 1.4? In such cases, our model architecture should no longer be a simple weighted combination. It will be a non-linear

function. For instance, check the curved separator in Fig. 1.4. Another example is shown in Figure 1.5 - classifying the points in the 2D plane into the two classes indicated in blue and red requires non-linear models.

Non-linear models make sense from the function approximation point of view as well. Ultimately, our goal is to approximate very complex and highly non-linear functions that model the classification or estimation processes demanded by life. Intuitively, it seems better to use *non-linear functions* to model them.

A very popular non-linear function in machine learning is the *sigmoid* function, so named because it looks like the letter 'S' in the alphabet. The sigmoid function is typically symbolized by the Greek letter *s*. It is defined as

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{1.5}$$

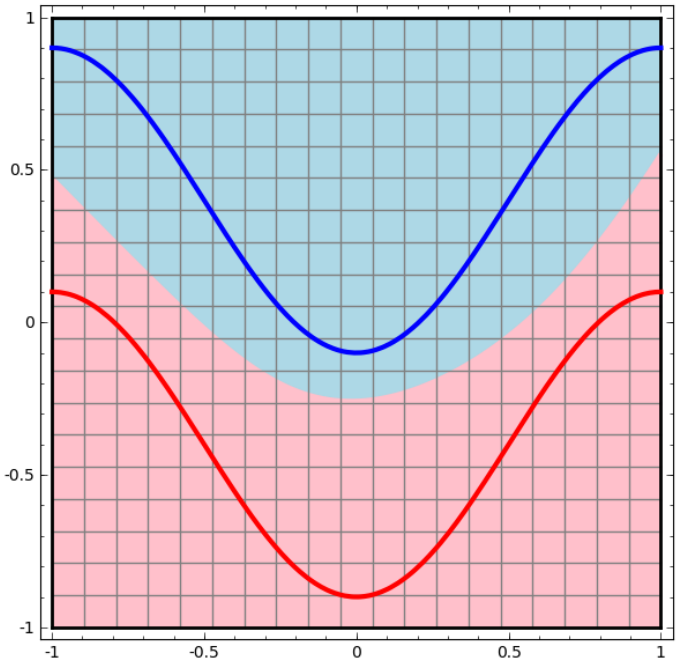


Figure 1.5: The two classes (indicated by blue and red colors respectively) can not be separated by a line. Non-linear (curved) separator needed.

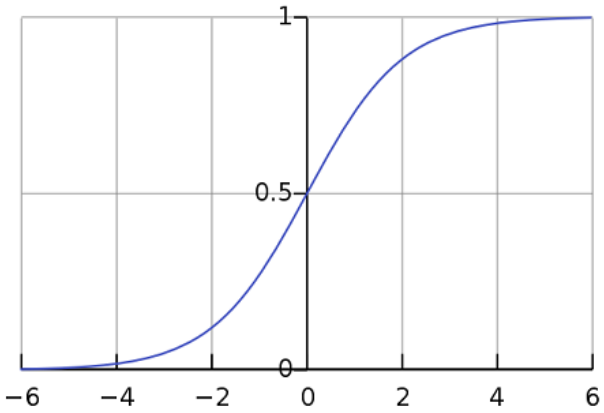


Figure 1.6: The sigmoid graph

The graph of the sigmoid function is shown in Fig. 1.6. Thus we can use the following model architecture

$$y = \sigma\,(\vec{w}^T \vec{x} + b) \tag{1.6}$$

Thus, a popular model architecture (still kind of simple) is that we take sigmoid (without parameters) of the weighted sum of the inputs. The sigmoid imparts the non-linearity. This architecture will be able to handle relatively more complex classification tasks than the weighted sum alone. In fact, equation 1.6 depicts the basic building block of a neural networks.

1.7 Higher Expressive Power through multiple non-linear layers: Deep Neural Networks

In section 1.6 we stated that adding non-linearity to the basic weighted sum yielded a model architecture that is able to handle more complex tasks. In machine learning parlance, the nonlinear model has more "expressive power".

Now consider a real life problem, say building a dog recognizer. The input space comprises pixel locations and pixel colors (x, y, r, g, b where r, g, b denotes red, green, blue components of a pixel color). The input dimensionality is large (proportional to the number of pixels in the image). Table 1.1 gives a small glimpse into possible variations in background and foreground that a typical deep learning system, say, a dog image recognizer has to deal with.

Table 1.1: A glimpse into background and foreground variations a typical deep learning system (here a dog image recognizer) has to deal with

We need a machine with really high expressive power here. How do we create such a machine in a principled way?

Instead of generating the output from input in a single step, how about taking a cascaded approach? We will generate a set of intermediate or hidden outputs from the inputs, where each hidden output is essentially a single logistic regression unit. Then we add another layer which takes the output of the previous layer as input. And so on. Finally, we will combine the outermost hidden layer outputs into the grand output.

We describe the system in the following equations. It should be noted that we have added a superscript to the weights to identify layer (layer 0 is closest to the input, layer *L* is the last layer furthest from input). We also have made the subscripts two dimensional (so that the weights for a given layer becomes a matrix). The first subscript identifies the destination node and the second subscript identifies the source node (see Fig 1.7).

The astute reader might notice that the following equations do *not* have an explicit bias term. That is because, for simplicity of notation, we have rolled it into the set of weights and assumed that one of the inputs, say $x_0 = 1$ and the corresponding weight, e.g., w_0 is the bias.

Layer 0: generates n_0 hidden outputs from $n + 1$ inputs

$$\begin{aligned} h_0^{(0)} &= \sigma\left(w_{00}^{(0)}x_0 + w_{01}^{(0)}x_1 + \cdots w_{0n}^{(0)}x_n\right) \\ h_1^{(0)} &= \sigma\left(w_{10}^{(0)}x_0 + w_{11}^{(0)}x_1 + \cdots w_{1n}^{(0)}x_n\right) \\ &\vdots \\ h_{n_0}^{(0)} &= \sigma\left(w_{n_0 0}^{(0)}x_0 + w_{n_0 1}^{(0)}x_1 + \cdots w_{n_0 n}^{(0)}x_n\right) \end{aligned} \tag{1.7}$$

Layer 1: generates n_1 hidden outputs from n_0 hidden outputs from layer 0

$$\begin{aligned} h_0^{(1)} &= \sigma\left(w_{00}^{(1)}h_0^{(0)} + w_{01}^{(1)}h_1^{(0)} + \cdots w_{0n_0}^{(1)}h_{n_0}^{(0)}\right) \\ h_1^{(1)} &= \sigma\left(w_{10}^{(1)}h_0^{(0)} + w_{11}^{(1)}h_1^{(0)} + \cdots w_{1n_0}^{(1)}h_{n_0}^{(0)}\right) \\ &\vdots \\ h_{n_1}^{(1)} &= \sigma\left(w_{n_1 0}^{(1)}h_0^{(0)} + w_{n_1 1}^{(1)}h_1^{(0)} + \cdots w_{n_1 n_0}^{(1)}h_{n_0}^{(0)}\right) \\ &\cdots \end{aligned} \tag{1.8}$$

Final Layer (L): generates m + 1 visible outputs from $n_{L\text{-}1}$ previous layer hidden outputs

$$\begin{aligned} h_0^{(L)} &= \sigma\left(w_{00}^{(L)}h_0^{(L-1)} + w_{01}^{(L)}h_1^{(L-1)} + \cdots w_{0n_{L-1}}^{(L)}h_{n_{L-1}}^{(L-1)}\right) \\ h_1^{(L)} &= \sigma\left(w_{10}^{(L)}h_0^{(L-1)} + w_{11}^{(L)}h_1^{(L-1)} + \cdots w_{1n_{L-1}}^{(L)}h_{n_{L-1}}^{(L-1)}\right) \\ &\vdots \\ h_m^{(L)} &= \sigma\left(w_{m0}^{(L)}h_0^{(L-1)} + w_{m1}^{(L)}h_1^{(L-1)} + \cdots w_{mn_{L-1}}^{(L)}h_{n_{L-1}}^{(L-1)}\right) \end{aligned} \tag{1.9}$$

The above equations can be pictorially depicted in Fig 1.7.

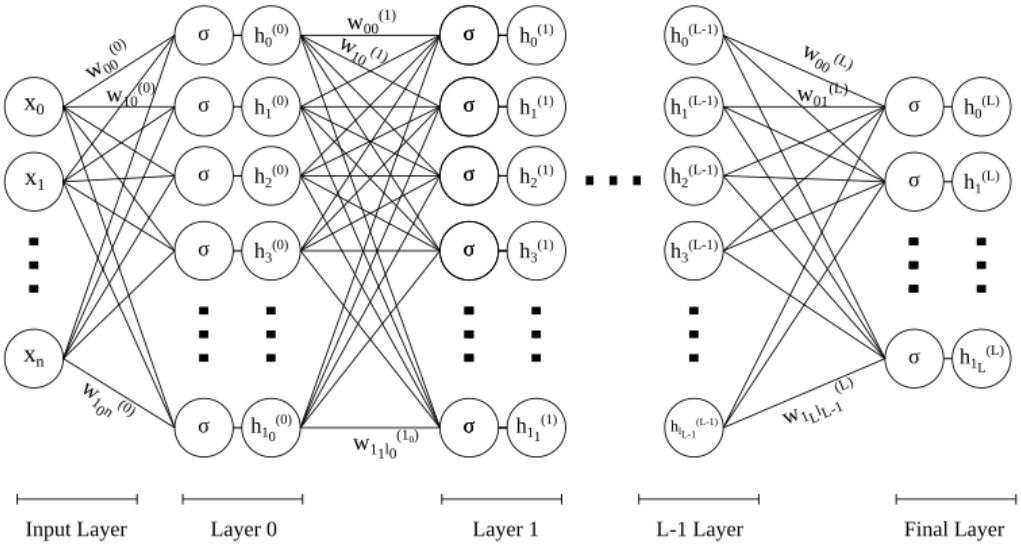


Figure 1.7: Multi Layered Neural Network

This machine, depicted in Fig 1.7, can be incredibly powerful, with huge expressive power. We can adjust its expressive power systematically to fit the problem at hand. This then is a neural network. We will devote the rest of the book to studying this.

1.8 Summary

In this chapter we gave an overview of machine learning leading all the way up to deep learning. The ideas were illustrated with a toy cat brain example. Some mathematical notions (e.g., vectors) were used in this chapter without proper introduction. The reader is encouraged to revisit this chapter after vectors and matrices have been introduced.

The author would like to leave the reader with the following mental pictures from this chapter

- Machine learning as a fundamentally different paradigm of computing. In traditional computing, one provides a step by step instruction sequence to the computer, telling it what to do. In machine learning, one builds a mathematical model that tries to approximate the unknown function that generates a classification or estimation from inputs.
- The mathematical nature of the model function is stipulated from the physical nature and complexity of the classification or estimation task. Models have parameters. Parameter values are estimated from training data - inputs with known outputs. The parameter values are optimized so that the model output is as close as possible to training outputs on training inputs.
- An alternative geometric view of a machine is a transformation that maps points in the multi-dimension input space to a point in the output space.
- More complex the classification/estimation task, the more complex the approximating

function. In machine learning parlance, complex tasks need machines with higher expressive power. Higher expressive power comes from non-linearity (e.g., the sigmoid function, see 1.5) and layered combination of simpler machines. This takes us to deep learning, which is nothing but a multi-layered non-linear machine.

- Complex model functions are often built by combining simpler basis functions.

Tighten your seat belts, the fun is about to get more intense.

2

Introduction to Vectors, Matrices and Tensors from Machine Learning and Data Science point of view

At its core, machine learning, indeed all computer software, is about number crunching. One inputs a set of numbers to the machine and gets back a different set of numbers as output. However, this cannot be done randomly. It is important to organize these numbers appropriately, group them into meaningful objects that go in and come out of the machine. This is where vectors and matrices come in. These are concepts that mathematicians have been using for centuries – we are simply reusing them in machine learning. In this chapter, we will study vectors and matrices, primarily from a machine learning point of view. Starting from the basics, we will quickly graduate to advanced concepts, restricting ourselves to topics that have relevance to machine learning.

We provide Jupyter notebook based python implementations for most of the concepts discussed in this and other chapters. Complete fully functional code that can be downloaded and executed (after installing python and Jupyter notebook) can be found at https://github.com/krishnonwork/mathematical-methods-in-deep-learning-ipython. The code relevant to this chapter can be found at

https://nbviewer.jupyter.org/github/krishnonwork/mathematical-methods-in-deep-learning-ipython/tree/master/python/ch2/.

2.1 Vectors and their role in Machine Learning and Data Science

Let us revisit the machine learning model for cat brain that was introduced in 1.3 . It takes two numbers as input: representing the hardness and sharpness of the object in front of the cat. Cat brain processes the input and generates an output threat score which leads to *run away* or *ignore* or *approach and purr* decision. Now the two input numbers usually appear

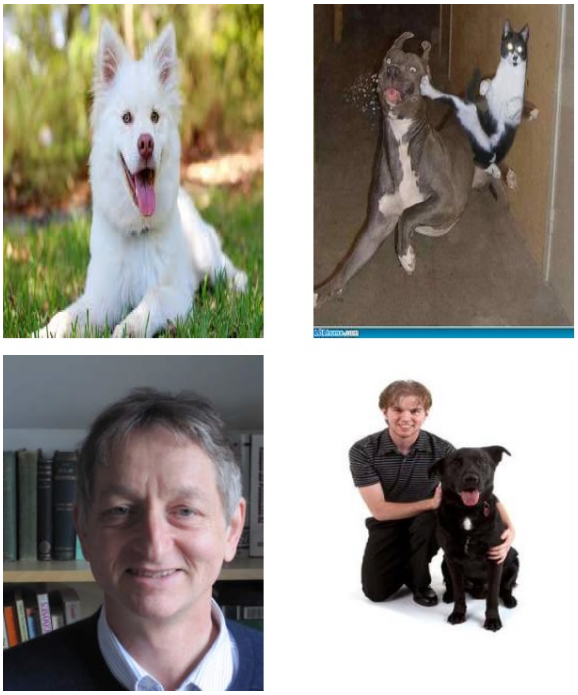
together and it will be handy to group them together into a single object. This object will be an ordered sequence of two numbers, the first one representing hardness and the second one representing sharpness. Such an object is a perfect example of a vector.

Thus, a *vector* can be thought of as an ordered sequence of two or more numbers, also known as an *array* of numbers[5]. Vectors constitute a compact way of denoting a set of numbers that together represent some entity. In this book, vectors will be represented by lower case letters with an overhead arrow and arrays by square brackets. For instance, the input to the cat brain model in 1.3 was a vector $\vec{x} = \begin{bmatrix} x_0 \\ x_1 \end{bmatrix}$ where x_0 represented hardness and x_1 represents sharpness.

Outputs to machine learning models too are often represented as vectors. For instance, consider an object recognition model that takes an image as input and emits a set of numbers indicating the probabilities that the image contains a dog, human or cat respectively. The output of such a model will be a 3 element vector $\vec{y} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix}$ where the number y_0 denotes the probability that the image contains a dog, y_1 denotes the probability that the image contains a human and y_2 denotes the probability that the image contains a cat. Table 2.1 shows some possible input images and corresponding output vectors.

[5] In mathematics, vectors can have an infinite number of elements. Such vectors cannot be expressed as arrays – but we will mostly ignore them in this book.

Table 2.1: Input images and corresponding output vectors denoting probabilities that the image contains a dog and/or human and/or cat respectively: possible output vector for top left image - [0.9 0.01 0.1], possible output vector for top right image- [0.9 0.01. 0.9], possible output vector for bottom left image - [0.01 0.99 0.01] possible output vector for bottom right image - [0.88 0.9. 0.001]



In multi-layered machines like neural networks, the input and output to each layer will be vectors. We also typically represent the parameters of the model function (see 1.3) as vectors. This is illustrated below in 2.3.

One particularly significant notion in machine learning and data science is the idea of a *feature vector*. This is essentially a vector that describes various properties of the object being dealt with in a particular machine learning problem. We will illustrate the idea with an example from the world of Natural Language Processing (NLP). Suppose we have a set of documents. We want to create a document retrieval system where, given a new document, we have to retrieve "similar" documents in the system. This essentially boils down to estimating similarity between documents in a quantitative fashion. We will study this problem in detail later, but for now we want to note that the most natural way to approach this is to create feature vectors for each document that quantitatively describe the document. Later, in section 2.5.6 we will see how to measure the similarity between these vectors, for now let us focus on simply creating descriptor vectors for the documents. A popular way to do this is to choose a set of interesting words – we typically exclude words like "and", "if", "to" which are present in all documents from this list - count the number of occurrence of interesting words in each document and make a vector of these. Table 2.2 shows a toy example with 6 documents and

corresponding feature vectors. For simplicity, we have considered only two ("gun" and "violence" in plural or singular, upper or lower case) of the possible set of words.

Table 2.2: Example Toy Documents and corresponding Feature Vectors describing them. Words eligible for the Feature Vector are colored in red. The first element of the feature vector indicates the number of occurrences of the word "gun", the second "violence".

docid	Document	Feature Vector
d_0	Roses are lovely. Nobody hates roses.	[0 0]
d_1	Gun violence has reached an epidemic proportion in America.	[1 1]
d_2	The issue of gun violence is really over-hyped. One can find many instances of violence where no guns were involved.	[2 2]
d_3	Guns are for violence prone people. Violence begets guns. Guns beget violence.	[3 3]
d_4	I like guns but I hate violence. I have never been involved in violence. But I own many guns. Gun violence is incomprehensible to me. I do believe gun owners are the most anti violence people on the planet. He who never uses a gun will be prone to senseless violence.	[5 5]
d_5	Guns were used in a armed robbery in San Francisco last night.	[1 0]
d_6	Acts of violence usually involves a weapon.	[0 1]

The sequence of pixels in the image can also be viewed as a feature vector. Neural networks in computer vision tasks usually expect this feature vector.

2.1.1 Geometric View of Vectors and its significance in Machine Learning and Data Science

Vectors can also be viewed geometrically. The simplest example is a 2-element vector $\vec{x} = \begin{bmatrix} x_0 \\ x_1 \end{bmatrix}$. Its 2 elements can be taken to be x and y, Cartesian coordinates in a 2-dimensional space. Then the vector will correspond to a point in that space. *Vectors with n elements will represent points in an n-dimensional space*. The ability to see inputs and outputs of machine learning models as points allows us to view the model itself as a geometric transformation that maps input points to output points in some high dimensional space. We have already seen this once in section 1.4. It is an enormously powerful concept that we will keep utilizing throughout the book.

We will briefly touch upon a subtle issue here. A vector represents the position of a point *with respect to another*. Furthermore, an array of coordinate values, like $\begin{bmatrix} x \\ y \end{bmatrix}$ describes the position of one point, *in a given coordinate system*. See Figure 2.1 to get a intuitive understanding of this. For instance, consider the plane of a page of this book. Suppose we want to reach the top right corner point of the page from the bottom left corner. Let us call the bottom left corner *O* and the top right corner *P*. We can travel the width (8.5 inches) rightwards to reach the bottom left corner and then travel the height (11 inches) upwards to

reach the top right corner. Thus, if we choose a coordinate system with the bottom left corner as origin and the *X* axis along the width and the *Y* axis along the height, point P corresponds to the array representation $\begin{bmatrix} 8.5 \\ 11 \end{bmatrix}$. But we could also have traveled along the diagonal from bottom left to top right corner to reach *P* from *O*. Either way, we end up at the same point *P*. Thus we have a conundrum. The vector $\vec{OP}$ represents the abstract geometric notion, position of *P* with respect to *O* independent of our choice of coordinate axes. On the other hand, the array representation depends on the choice of coordinate system. E.g., the array $\begin{bmatrix} 8.5 \\ 11 \end{bmatrix}$ represents the the top right corner point *P* only under a specific choice of coordinate axes (parallel to the sides of the page) and a reference point (bottom left corner). Ideally, we should specify the coordinate system along with the array representation to be unambiguous. How come then we never do so in machine learning? The answer: in machine learning, it does not matter what exactly the coordinate system is, as long as we stick to any fixed coordinate system.

There are explicit rules (which we will study below) that state how the vector transforms when the coordinate system changes. We will invoke them when necessary. All vectors used in a machine learning computation must consistently use the same coordinate system or must be transformed appropriately.

One other point. Planar spaces, e.g., the plane of the paper on which this book is written, are 2-dimensional (abbreviated 2D). The mechanical world we live in is 3-dimensional 3D). Human imagination usually fails to see higher dimensions. In machine learning and data science, we often will talk of spaces with thousands of dimensions. You may not see those spaces in your mind. But that is not a crippling limitation. You will use 3 dimensional analogues in your head. They work in a surprisingly large variety of cases. However, it is important to bear in mind that this is not always true. Some examples where the lower dimensional intuitions fail at higher dimensions will be shown later.

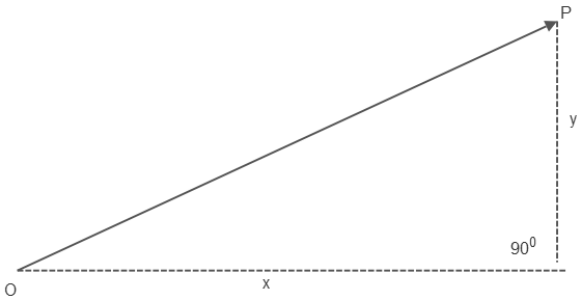


Figure 2.1: A vector describing the position of point P with respect to point O. The basic mental picture to have is an arrowed line. This agrees with the definition of vector we learnt in high school: vector has a magnitude (length of the arrowed line) and direction (indicated by the arrow). On a plane, this is equivalent to the ordered pair of numbers *x*, *y*, where the geometric interpretations of *x* and *y* are as shown in Figure. In this context, it is worthwhile to note that only the relative positions of the points O and P matter. If both the points are moved, keeping their relationship intact, the vector does not change.

2.2 Python code to create and access vectors and sub-vectors, slice and dice vectors, via Numpy and PyTorch parallel code

In this book, we will try to familiarize the the reader with numpy, PyTorch and similar programming paradigms alongside the relevant mathematics. Knowledge of python basics will be assumed. The reader is strongly encouraged to try out all code snippets in this book – after installing appropriate packages like numpy, PyTorch etc.

All the python code in this book is produced via jupyter-notebook. A summarized recapitulation of the theoretical material presented in code is provided right above the code snippet. The fully functional code demonstrating how to create vectors and access its elements, in Python Numpy as well as PyTorch can be found at https://nbviewer.jupyter.org/github/krishnonwork/mathematical-methods-in-deep-learning-ipython/blob/master/python/ch2/2.2-vector-numpy-pytorch-intro.ipynb.

2.2.1 Python Numpy code for introduction to Vectors

Numpy stands for Numerical Python. It is an inalienable part of practical machine learning.

Listing 2.1: Introduction to vectors via numpy.

```
1
2 v = np.array([0.11, 0.01, 0.98, 0.12, 0.98, 0.85, 0.03,
3               0.55, 0.49, 0.99, 0.02, 0.31, 0.55, 0.87, 0.63])
4
5 first_element = v[0]
6 third_element = v[2]
7
8 last_element = v[-1]
9 second_last_element = v[-2]
10
11 second_to_fifth_elements = v[1:4]
12
13 first_to_third_elements = v[:2]
14 last_two_elements = v[-2:]
15
16 num_elements_in_v = len(v)
```

vector in Numpy is a 1-dimensional array of numbers

input hardness vector of our cat-brain model

square bracket operator lets us access individual vector elements

negative indices count from end of array e.g., -1 denotes last element

-2 denotes second to last element

colon operator slices off a range of elements from the vector

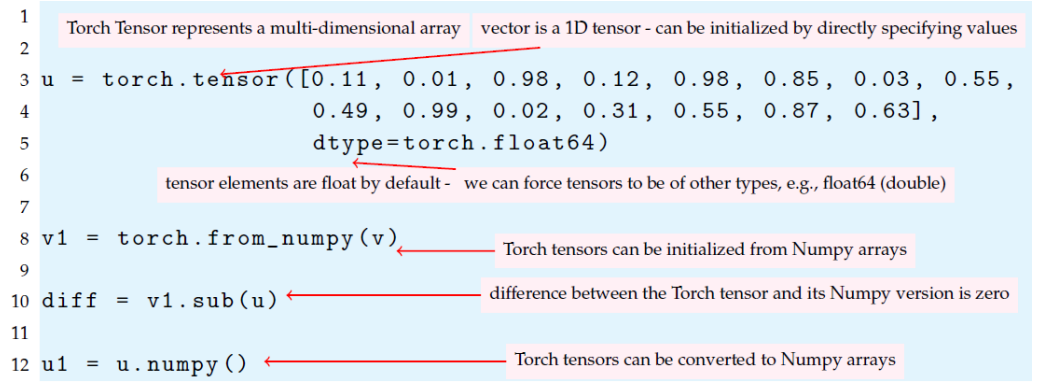
nothing before colon denotes beginning of array

nothing after colon denotes end of array

2.2.2 PyTorch code for introduction to Vectors

Pytorch is an open-source machine learning library developed by Facebook's artificial intelligence group. It is one of the most elegant practical tools for developing deep learning applications at present.

Listing 2.2: Introduction to vectors via PyTorch



2.3 Matrices and their role in Machine Learning and Data Science

Sometimes, it is not sufficient to group a set of numbers into a vector. We have to collect several vectors into another group. For instance, consider the input to training a machine learning model. Here we have several input instances, each comprising of a sequence of numbers. As seen in section 2.1, the sequence numbers belonging to a single input instance can be grouped into a vector. How do we represent the entire collection of input instances? This is where the concept of matrices, from the world of mathematics, come in handy. A *matrix* can be viewed as a rectangular array of numbers, arranged in a fixed count of rows and columns. Each row of a matrix is a vector, so is each column. Thus a matrix can be thought of as a collection of row vectors. It can also be viewed as a collection of column vectors. We can represent the entire set of numbers that constitute the training input to a machine learning model as a matrix, with each row vector corresponding to a single training instance.

Consider our familiar cat-brain problem again. As stated earlier, single input instance to the machine is a vector $\vec{x} = \begin{bmatrix} x_0 \\ x_1 \end{bmatrix}$ where x_0 describes hardness of the object in front of the cat . Now consider a training dataset with many such input instances, each with a known output threat score. You might recall from section 1.1 that the goal in machine learning is to create a function that maps these inputs to their respective outputs with as little overall error as possible. Our training data may look as shown in Table 2.3 below (it should be noted that in real life problems the training dataset is usually large in size, often runs into millions of input-output pairs, however in this toy problem we will have 15 training data instances). From Table 2.3, we can collect the columns corresponding to hardness and sharpness into a matrix as shown in equation 2.1 - this is a compact representation of the training dataset for this problem.[6]

[6] we will usually use upper case letters to symbolize matrices

Table 2.3: Example Training Dataset for our Toy Machine Learning Based Cat Brain

	input value: hardness	input value: sharpness	output: threat score
0	0.11	0.09	-0.8
1	0.01	0.02	-0.97
2	0.98	0.91	0.89
3	0.12	0.21	-0.68
4	0.98	0.99	0.95
5	0.85	0.87	0.74
6	0.03	0.14	-0.88
7	0.55	0.45	0.00
8	0.49	0.51	0.01
9	0.99	0.01	0.009
10	0.02	0.89	-0.07
11	0.31	0.47	-0.23
12	0.55	0.29	-0.14
13	0.87	0.76	0.65
14	0.63	0.74	0.36

$$\text{Example cat-brain dataset matrix } X = \begin{bmatrix} 0.11 & 0.09 \\ 0.01 & 0.02 \\ 0.98 & 0.91 \\ 0.12 & 0.21 \\ 0.98 & 0.99 \\ 0.85 & 0.87 \\ 0.03 & 0.14 \\ 0.55 & 0.45 \\ 0.49 & 0.51 \\ 0.99 & 0.01 \\ 0.02 & 0.89 \\ 0.31 & 0.47 \\ 0.55 & 0.29 \\ 0.87 & 0.76 \\ 0.63 & 0.74 \end{bmatrix} \tag{2.1}$$

Each row of matrix X is a particular input instance. Different rows represent different input instances. Thus, moving along a row, one encounters successive elements of a single input vectors. Moving along a column, one encounters elements of different input instances. Notice

that an individual element is now indexed by 2 numbers, as opposed to 1 in a vector. Thus the 0^{th} row is the vector $[x_{00}\, x_{01}]$ representing the 0^{th} input instance.

MATRIX REPRESENTATION OF DIGITAL IMAGES

Digital images too are often represented as matrices. Here, each element represents the brightness at a specific pixel position (x, y coordinate) of the image. Typically, the brightness value is normalized to an integer in the range 0 to 255. 0 is black and 255 is white and 128 is gray etc[7]. Following is an example of a tiny image, 9 pixel in width and 4 pixel in height.

$$I_{4,9} = \begin{bmatrix} 0 & 8 & 16 & 24 & 32 & 40 & 48 & 56 & 64 \\ 64 & 72 & 80 & 88 & 96 & 104 & 112 & 120 & 128 \\ 128 & 136 & 144 & 152 & 160 & 168 & 176 & 184 & 192 \\ 192 & 200 & 208 & 216 & 224 & 232 & 240 & 248 & 255 \end{bmatrix} \tag{2.2}$$

The brightness increases gradually from left to right and also top to bottom. I_{00} represents the top left pixel which is black. $I_{3,8}$ represents the bottom right pixel which is white. The intermediate pixels are various shades of gray in between black and white. The actual image looks as shown in Figure 2.2.

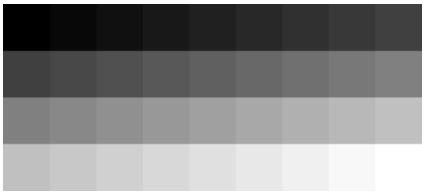


Figure 2.2: Image corresponding to matrix I4,9 in equation 2.2

2.4 Python Code: Introduction to Matrices, Tensors and Images via Numpy and PyTorch parallel code

For programming purposes, one can think of tensors as multi-dimensional arrays. Scalars are 0-dimensional tensors. Vectors are 1-dimensional tensors. Matrices are 2-dimensional tensors. RGB images are 3-dimensional tensors ($colorchannels \times height \times width$). A batch of 64 images is a 4-dimensional tensor ($64 \times colorchannels \times height \times width$).

7 in digital computers, numbers in the range 0..255 can be represented with a single byte of storage, hence this choice

2.4.1 Python Numpy code for introduction to Tensors, Matrices and Images

Listing 2.3: Introduction to matrices via numpy.

```
1
2   Matrix is a 2D array of numbers, i.e., 2D tensor. Training data input for a machine-learning model can be viewed as a matrix
3   Each input instance is one row.  Row count ≡ number of training examples, column count ≡ training instance size
4 X = np.array([[0.11, 0.09], [0.01, 0.02], [0.98, 0.91], [0.12, 0.21],
5               [0.98, 0.99], [0.85, 0.87], [0.03, 0.14], [0.55, 0.45],
6               [0.49, 0.51], [0.99, 0.01], [0.02, 0.89], [0.31, 0.47],
7               [0.55, 0.29], [0.87, 0.76], [0.63, 0.24]])
8   cat-brain training data input, 15 examples, each with 2 values - hardness, sharpness.
9   15 × 2 numpy array, created by directly specifying values
10 print("Shape of the matrix is: {}".format(X.shape))
11
12 first_element = X[0, 0]
13
14 row_0 = X[0, :]
15 row_1 = X[1, 0:2]
16
17 column_0 = X[:, 0]
18 column_1 = X[:, 1]
```

shape of a tensor is a list. For a matrix, first list element is num rows, second list element is num columns

Square brackets extract individual matrix elements

standalone colon operator denotes all possible indices

colon operator denotes range of indices

0^{th} column

1^{th} column

Listing 2.4: Slicing and dicing matrices.

Ranges of rows and columns can be specified via colon operator to slice off (extract) sub-matrices

```
first_3_training_examples = X[:3, ]
print("Sharness of 5-7 training examples is: {} ".format(X[5:8, 1]))
```

extract first 3 training examples (rows)

extract sharpness feature for 5^{th} to 7^{th} training examples

Listing 2.5: Tensors and Images in numpy

Numpy n-dimensional arrays represent tensors. A vector is a 1-tensor, a matrix is 2-tensor, a scalar is 0-tensor

```
tensor = np.random.random((5, 5, 3))
```

Creating a random tensor of specified dimensions

All images are tensors

RGB image of height H, width W is 3-tensor of shape [3, H, W]

```
I49 = np.array([[0,   8,  16, 24, 32, 40,  48,  56,  64],
                [64,  72,  80,  88,  96, 104, 112, 120, 128],
                [128, 136, 144, 152, 160, 168, 176, 184, 192],
                [192, 200, 208, 216, 224, 232, 240, 248, 255]],
               dtype=np.uint8)
```

4×9 single channel image shown in in Fig 2.2

```
img = cv2.imread('../../Figures/dog3.jpg')
img_b = img[:, :, 0]
img_g = img[:, :, 1]
img_r = img[:, :, 2]
img_cropped = img[0:100, 0:100, :]
```

Read $199 \times 256 \times 3$ image from disk

usual slicing dicing operators work

extract red, green, blue channels of image, shown in Fig. 2.3

crop out 100×100 sub-image, shown in Fig. 2.4

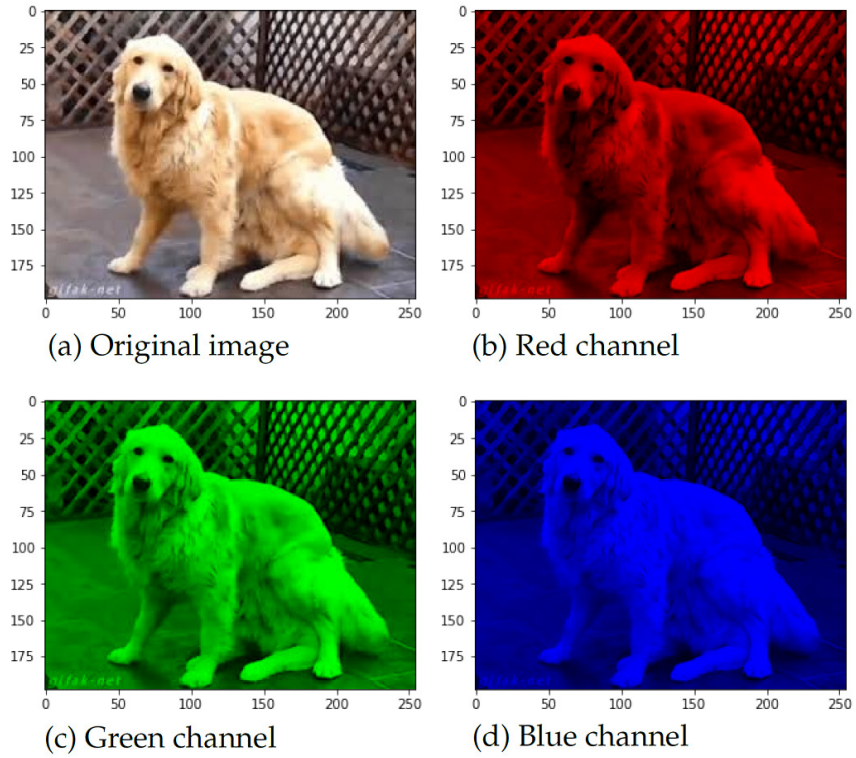


Figure 2.3: Tensors and images in numpy

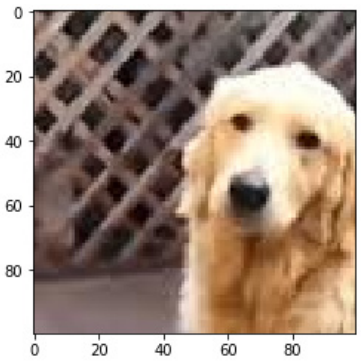


Figure 2.4: Cropped image of dog

2.4.2 PyTorch code for introduction to Tensors and Matrices

Listing 2.6: Tensors in PyTorch.

```
# cat-brain training data input, 15 examples, each with 2 values - hardness, sharpness
# 15 × 2 torch tensor, each element a 64 bit float, created by directly specifying values
Y = torch.tensor([[0.11, 0.09], [0.01, 0.02], [0.98, 0.91],
                  [0.12, 0.21], [0.98, 0.99], [0.85, 0.87],
                  [0.03, 0.14], [0.55, 0.45], [0.49, 0.51],
                  [0.99, 0.01], [0.02, 0.89], [0.31, 0.47],
                  [0.55, 0.29], [0.87, 0.76], [0.63, 0.24]],
                 dtype=torch.float64)
# Torch tensors can be converted to Numpy arrays, the two arrays are equivalent
np.allclose(X, Y.numpy(), rtol=1e-7)
# Torch tensors can be initialized from Numpy arrays, the two arrays are equivalent
np.allclose(torch.from_numpy(X), Y, 1e-7)
# Slicing operations of numpy arrays work on torch tensors too
print(Y[3, :])
print(Y[3:5, 1:2])
# Torch tensors can be added/ subtracted like Numpy arrays
print(torch.from_numpy(X) - Y)
# Torch tensors support automatic backpropagation
# Numpy arrays don't.
# Backpropagation crucial for training machine learning models
```

2.5 Basic Vector and Matrix operations in Machine Learning and Data Science

In this section we will introduce several basic vector and matrix operations along with examples to demonstrate their significance in image processing, computer vision and machine learning. It is meant to be an application-centric introduction to linear algebra. But it is not meant to be a comprehensive review of matrix and vector operations, for which the reader is referred to a textbook on linear algebra.

2.5.1 Matrix and Vector Transpose

In equation 2.2 we encountered the matrix $I_{4,9}$ depicting a tiny image. Suppose we want to rotate the image by $90°$, so that it looks like Figure 2.5.

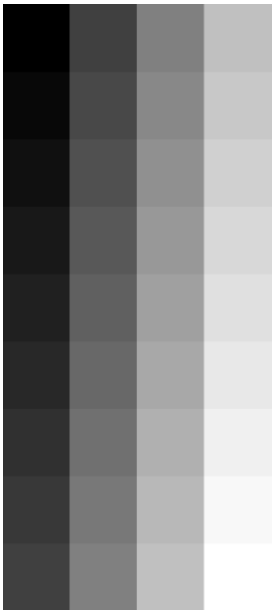


Figure 2.5: Image corresponding to transpose of matrix I4,9,shown in equation 2.3. This is equivalent to rotating the image by 90° angle

The original matrix $I_{4,9}$ and its transpose $I^T_{4,9} = I_{9,4}$ are shown below

$$I_{4,9} = \begin{bmatrix} 0 & 8 & 16 & 24 & 32 & 40 & 48 & 56 & 64 \\ 64 & 72 & 80 & 88 & 96 & 104 & 112 & 120 & 128 \\ 128 & 136 & 144 & 152 & 160 & 168 & 176 & 184 & 192 \\ 192 & 200 & 208 & 216 & 224 & 232 & 240 & 248 & 255 \end{bmatrix}$$

$$I^T_{4,9} = I_{9,4} = \begin{bmatrix} 0 & 64 & 128 & 192 \\ 8 & 72 & 136 & 200 \\ 16 & 80 & 144 & 208 \\ 24 & 88 & 152 & 216 \\ 32 & 96 & 160 & 224 \\ 40 & 104 & 168 & 232 \\ 48 & 112 & 176 & 240 \\ 56 & 120 & 184 & 248 \\ 64 & 128 & 192 & 255 \end{bmatrix} \tag{2.3}$$

By comparing equation 2.2 and equation 2.3 one can easily see that one can be obtained from the other by interchanging the row and column indices. This operation is generally known as matrix transposition.

Formally, the transpose of a matrix $A_{m,n}$ with *m* rows and *n* columns is another matrix with *n* rows and *m* columns. This transposed matrix, denoted $A^T_{n,m}$ is such that $A^T_{ij} = A_{ji}$. Like the

value at row 0 column 6 in matrix $I_{4,9}$ is 48. In the transposed matrix the same value will appear in row 6 and column 0. In matrix parlance $I_{4,9}[0, 6] = I_{9,4}{}^{T}[6, 0] = 48$.

Vector transposition is really a special case of matrix transposition (since all vectors are matrices - a column vector with *n* elements is a $n \times 1$ matrix). For instance, an arbitrary vector and its transpose are shown in equation

$$\vec{v} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \tag{2.4}$$

$$\vec{v}^{T} = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \tag{2.5}$$

2.5.2 Dot Product of two vectors and its role in Machine Learning and Data Science

In section 1.3 we saw the simplest of machine learning models where the output is generated by taking a weighted sum of the inputs (and then adding a constant bias value). This model/machine is characterized by the weights w_0, w_1 and bias b. Take the rows of Table 2.3. E.g., for row 0, input values are: hardness of approaching object $= 0.11$ and softness $= 0.09$. The corresponding model output will be $y = w_0 \times 0.11 + w_1 \times 0.09 + b$. In fact, goal of training is to choose w_0, w_1 and b such that model outputs are as close as possible to the known outputs: i.e., $y = w_0 \times 0.11 + w_1 \times 0.09 + b$ should be as close to -0.8 as possible, $y = w_0 \times 0.01 + w_1 \times 0.02 + b$ should be as close to -0.97 as possible etc. In general, given an input instance $\vec{x} = \begin{bmatrix} x_0 \\ x_1 \end{bmatrix}$ the model output is $y = x_0 w_0 + x_1 w_1 + b$.

We will keep returning to the above model throughout the chapter. But in this subsection, let us consider a different question. In this toy example we have only 2 values per input instance. That implies we have only 3 model parameters: 2 weights, w_0, w_1 and 1 bias b. Hence it is not very messy to write the model output flat out as $y = x_0 w_0 + x_1 w_1 + b$. Is there a compact way to represent the model output on a specific input instance, irrespective of the size of the input?

Turns out the answer is yes - we can use an operation called dot product from the world of mathematics. We have already seen in section 2.1 that an individual instance of model input can be compactly represented by a vector, say $\vec{x}$ (it can have any number of input values). We can also represent the set of weights as vector $\vec{w}$ - it will have the same number of items as input vector. The model output is obtained via the dot product operation of vectors. Dot product is simply the point wise multiplication of the two vectors $\vec{x}$ and $\vec{w}$ as shown below.

Formally, given two vectors $\vec{x} = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix}$ and $\vec{w} = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_n \end{bmatrix}$ dot product of the two vectors is defined as

$$\vec{x} \cdot \vec{w} = x_0 w_0 + x_1 w_1 + \cdots x_n w_n \tag{2.6}$$

In other words, sum of the products of corresponding elements of the two vectors is called *dot product* of the two vectors, denoted $\vec{a} \cdot \vec{b}$.

Note that the dot product notation can compactly represent the model output as $y = \vec{w} \cdot \vec{x} + b$.

The representation does not increase in size even when the number of inputs and weights are large.

Consider our (by now familiar) cat brain example again. Suppose the weight vector is $\vec{w} = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$ and the bias value $b = 5$. Then the model output for the 0^{th} input instance from Table 2.3 will be $\begin{bmatrix} 0.11 \\ 0.09 \end{bmatrix} \cdot \begin{bmatrix} 3 \\ 2 \end{bmatrix} = 0.11 \times 3 + 0.09 \times 2 + 5 = 5.51$.

It is another matter that these are bad choices for weight and bias parameters, since the model output 5.51 is a far cry from the desired output -0.89. We will soon see how to obtain better parameter values. For now, we just need to note that the dot product offers a neat way to represent the simple weighted sum model output.

The dot product is defined only if the vectors have the same dimensions.

Sometimes the dot product is also referred to as *inner product*, denoted $\langle \vec{a}, \vec{b} \rangle$. Strictly speaking, the phrase inner product is a bit more general, it applies to infinite dimensional vectors as well. In this book, we will often use the terms interchangeably, sacrificing mathematical rigor for enhanced understanding.

2.5.3 Matrix Multiplication and Machine Learning, Data Science

Matrix-Vector Multiplication: In section 2.5.2 we saw that given a weight vector, say $\vec{w} = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$ and the bias value $b = 5$, the weighted sum model output upon a single input instance, say $\begin{bmatrix} 0.11 \\ 0.09 \end{bmatrix}$ can be represented using a vector-vector dot product $\vec{w} \cdot \vec{x} + b = \begin{bmatrix} 0.11 \\ 0.09 \end{bmatrix} \cdot \begin{bmatrix} 3 \\ 2 \end{bmatrix} + 5$. Now, as depicted in equation 2.1, during training we are dealing with many training data instances at the same time. In fact, in real life, we typically deal with hundreds of thousands of input instances, each having hundreds of values. Is there a way to represent this compactly, such that it is independent of the count of input instances and their sizes?

Again turns out the answer is yes. We can use the idea of matrix-vector multiplication from the world of mathematics. The product of a matrix X and column vector $\vec{w}$ is another vector, denoted $X\vec{w}$. Its elements are the dot products between the row vectors of X and the column vector $\vec{w}$. E.g.,given the model weight vector $\vec{w} = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$ and the bias value $b = 5$, the outputs on the toy training dataset of our familiar cat-brain model (equation 2.1) can be obtained via the following steps

$$\begin{bmatrix} 0.11 & 0.09 \\ 0.01 & 0.02 \\ 0.98 & 0.91 \\ 0.12 & 0.21 \\ 0.98 & 0.99 \\ 0.85 & 0.87 \\ 0.03 & 0.14 \\ 0.55 & 0.45 \\ 0.49 & 0.51 \\ 0.99 & 0.01 \\ 0.02 & 0.89 \\ 0.31 & 0.47 \\ 0.55 & 0.29 \\ 0.87 & 0.76 \\ 0.63 & 0.74 \end{bmatrix} \begin{bmatrix} 3 \\ 2 \end{bmatrix} = \begin{bmatrix} 0.11 \times 3 + 0.09 \times 2 = 0.51 \\ 0.01 \times 3 + 0.02 \times 2 = 0.07 \\ 0.98 \times 3 + 0.91 \times 2 = 4.76 \\ 0.12 \times 3 + 0.21 \times 2 = 0.78 \\ 0.98 \times 3 + 0.99 \times 2 = 4.92 \\ 0.85 \times 3 + 0.87 \times 2 = 4.29 \\ 0.03 \times 3 + 0.14 \times 2 = 0.37 \\ 0.55 \times 3 + 0.45 \times 2 = 2.55 \\ 0.49 \times 3 + 0.51 \times 2 = 2.49 \\ 0.99 \times 3 + 0.01 \times 2 = 2.99 \\ 0.02 \times 3 + 0.89 \times 2 = 1.84 \\ 0.31 \times 3 + 0.47 \times 2 = 1.87 \\ 0.55 \times 3 + 0.29 \times 2 = 2.23 \\ 0.87 \times 3 + 0.76 \times 2 = 4.13 \\ 0.63 \times 3 + 0.74 \times 2 = 3.37 \end{bmatrix} \tag{2.7}$$

Adding the bias value of 5, the model output on the toy training dataset is

$$\begin{bmatrix} 5 + 0.51 = 5.51 \\ 5 + 0.07 = 5.07 \\ 5 + 4.76 = 9.76 \\ 5 + 0.78 = 5.78 \\ 5 + 4.92 = 9.92 \\ 5 + 4.29 = 9.29 \\ 5 + 0.37 = 5.37 \\ 5 + 2.55 = 7.55 \\ 5 + 2.49 = 7.49 \\ 5 + 2.99 = 7.99 \\ 5 + 1.84 = 6.84 \\ 5 + 1.87 = 6.87 \\ 5 + 2.23 = 7.23 \\ 5 + 4.13 = 9.13 \\ 5 + 3.37 = 8.37 \end{bmatrix} \tag{2.8}$$

In general, matrix column-vector multiplication works as follows.

$$A\vec{b} = \vec{c} \text{ or}$$

$$\begin{bmatrix} a_{11} & a_{12} & \cdots a_{1n} \\ a_{21} & a_{22} & \cdots a_{2n} \\ \vdots & & \\ a_{m1} & a_{m2} & \cdots a_{mn} \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} = \begin{bmatrix} c_1 = a_{11}b_1 + a_{12}b_2 + \cdots a_{1n}b_n \\ c_2 = a_{21}b_1 + a_{22}b_2 + \cdots a_{2n}b_n \\ \vdots \\ c_m = a_{m1}b_1 + a_{m2}b_2 + \cdots a_{mn}b_n \end{bmatrix} \tag{2.9}$$

Matrix-Matrix Multiplication: Generalizing the notion of matrix times vector, we can define a matrix times matrix. A matrix with m rows and p columns, say, $A_{m,p}$ can be multiplied with another matrix with p rows and n columns, say $B_{p,n}$ to generate a matrix with m rows and p columns, say $C_{m,n}$, e.g., $C_{m,n} = A_{m,p} B_{p,n}$. Note that the number of columns of the left matrix must match the number of rows in the right matrix. Element i, j of the result matrix - $C_{i,j}$ is obtained by point-wise multiplication of the elements of the i^{th} row vector of A and j^{th} column vector of B. The following example illustrates the idea

$$A_{3,2} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix}$$

$$B_{2,2} = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

$$C_{3,2} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

$$= \begin{bmatrix} c_{11} = a_{11}b_{11} + a_{12}b_{21} & c_{12} = a_{11}b_{12} + a_{12}b_{22} \\ c_{21} = a_{21}b_{11} + a_{22}b_{21} & c_{22} = a_{21}b_{12} + a_{22}b_{22} \\ c_{31} = a_{31}b_{11} + a_{32}b_{21} & c_{32} = a_{31}b_{12} + a_{32}b_{22} \end{bmatrix}$$

The computation for $C_{2,1}$ is shown via highlights by way of example. It is worthwhile to note that matrix multiplication is not commutative, in general, $AB \neq BA$.

At this point, the astute reader may already have noted that the dot product is a special case of matrix multiplication. For instance, the dot product between two vectors $\vec{w} = \begin{bmatrix} w_0 \\ w_1 \end{bmatrix}$ And $\vec{x} = \begin{bmatrix} x_0 \\ x_1 \end{bmatrix}$ is equivalent to transposing either of the two vectors and then doing a matrix multiplication with the other. In other words,

$$\vec{w} \cdot \vec{x} = \vec{w}^T \vec{x} = \begin{bmatrix} w_0 \\ w_1 \end{bmatrix}^T \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = \begin{bmatrix} w_0 & w_1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = \vec{x}^T \vec{w} = w_0 x_0 + w_1 x_1$$

The idea works in higher dimensions too. In general, given two vectors $\vec{x} = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix}$ and $\vec{w} = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_n \end{bmatrix}$, dot product of the two vectors is defined as

$$\begin{aligned} & \vec{x} \cdot \vec{w} \\ & = \vec{w}^T \vec{x} = \begin{bmatrix} w_0 & w_1 & \cdots & w_n \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} \\ & = \vec{x}^T \vec{w} = \begin{bmatrix} x_0 & x_1 & \cdots & x_n \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_n \end{bmatrix} \\ & = x_0 w_0 + x_1 w_1 + \cdots x_n w_n \end{aligned} \tag{2.10}$$

Another special case of matrix matrix multiplication is row-vector matrix multiplication, e.g.,

$$\begin{aligned} \vec{b}^T A &= \vec{c} \quad \text{or} \\ \begin{bmatrix} b_1 & b_2 & b_3 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix} &= \begin{bmatrix} c_1 = a_{11} b_1 + a_{21} b_2 + a_{31} b_3 & c_2 = a_{12} b_1 + a_{22} b_2 + a_{32} b_3 \end{bmatrix} \end{aligned}$$

Transpose of Matrix Products: Given two matrices A and B where the number of columns of A matches the number of rows of B (i.e., it is possible to multiply them) the transpose of the product is the product of the individual transposes, in reversed order. The rule also applies to matrix vector multiplication. The following equations capture this rule

$$
\begin{aligned}
(AB)^T &= B^T A^T \\
(A\vec{x})^T &= \vec{x}^T A^T \\
\left(\vec{x}^T A\right)^T &= A^T \vec{x}
\end{aligned}
\tag{2.11}
$$

2.5.4 Length of a Vector aka L2 norm and its role in Machine Learning

Suppose a machine learning model was supposed to output a target value $\vec{y}$ y but it outputted y instead. We are interested in the *error* made by the model. The error is the difference between the target and the actual outputs.

We would like to make one important note here. During computing errors, we are only interested in how far away from ideal the computed value is. We do not care whether the computed value is bigger or smaller than ideal. For instance, if the target (ideal) value is 2, the computed values 1.5 and 2.5 are equally in error - we are equally happy or unhappy with either of them. Hence, it is common practice to *square* error values. Thus for instance, if the the target value is 2 and the computed values 1.5 the error is $(1.5-2)^2 = 0.25$. If the target value is 2.5, the error is $(2.5-2)^2 = 0.25$. The squaring operation essentially eliminates the sign of the error value. We can then follow it up with a square root, but it is OK not to. One might ask, but wait, squaring alters the value of the quantity, don't care about the exact values of the error? Answer is, we usually don't, we only care about *relative* values of errors. If the target is 2, all we want that the error for an output value of, say, 2.1 is less than the error for output value of 2.5, the exact values of the errors do not matter.

Let us now continue with our discussion of machine learning model error. As seen earlier in section 2.5.3, given a model weight vector, say $\vec{w} = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$ and the bias value $b = 5$, the weighted sum model output upon a single input instance, say, $\begin{bmatrix} 0.11 \\ 0.09 \end{bmatrix}$ is $\begin{bmatrix} 0.11 \\ 0.09 \end{bmatrix} \cdot \begin{bmatrix} 3 \\ 2 \end{bmatrix} + 5 = 5.51.$

The corresponding target (ideal) output, from Table 2.3, is -0.8. The squared error $e^2 = (-0.8 - 5.51)^2 = 39.82$ gives us an idea of how good or bad the model parameters 3, 2, 5 are. For instance, if instead, we use a weight vector $\vec{w} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$ and bias value -1, we get model output $\vec{w} \cdot \vec{x} + b = \begin{bmatrix} 0.11 \\ 0.09 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \end{bmatrix} - 1 = -0.8.$ The output is exactly same as the target. The corresponding squared error $e^2 = (-0.8 - (-0.8))^2 = 0$. This (zero error) immediately tells us that $1, 1, -1$ is a much better choice of model parameters than $3, 2, 5$.

What happens when we have multiple inputs, as during training a model? In equation 2.8 we have seen that given the toy training dataset from Table 2.3, a simple weighted sum model with weights 3, 2 and bias 5 will generate the output vector

$$\vec{y} = \begin{bmatrix} 5.51 \\ 5.07 \\ 9.76 \\ 5.78 \\ 9.92 \\ 9.29 \\ 5.37 \\ 7.55 \\ 7.49 \\ 7.99 \\ 6.84 \\ 6.87 \\ 7.23 \\ 9.13 \\ 8.37 \end{bmatrix}$$

From Table 2.3 we also see that the target output vector is

$$\vec{\bar{y}} = \begin{bmatrix} -0.8 \\ -0.97 \\ 0.89 \\ -0.67 \\ 0.97 \\ 0.72 \\ -0.83 \\ 0.00 \\ 0.00 \\ 0.00 \\ -0.09 \\ -0.22 \\ -0.16 \\ 0.63 \\ 0.37 \end{bmatrix}$$

The differences between target and model output over the entire training set can be expressed as a vector

$$\vec{\bar{y}} - \vec{y} = \begin{bmatrix} 5.51 \\ 5.07 \\ 9.76 \\ 5.78 \\ 9.92 \\ 9.29 \\ 5.37 \\ 7.55 \\ 7.49 \\ 7.99 \\ 6.84 \\ 6.87 \\ 7.23 \\ 9.13 \\ 8.37 \end{bmatrix} - \begin{bmatrix} -0.8 \\ -0.97 \\ 0.89 \\ -0.67 \\ 0.97 \\ 0.72 \\ -0.83 \\ 0.00 \\ 0.00 \\ 0.00 \\ -0.09 \\ -0.22 \\ -0.16 \\ 0.63 \\ 0.37 \end{bmatrix} = \begin{bmatrix} 6.31 \\ 6.04 \\ 8.87 \\ 6.45 \\ 8.95 \\ 8.57 \\ 6.2 \\ 7.55 \\ 7.49 \\ 7.99 \\ -0.09 \\ -0.22 \\ -0.16 \\ 0.63 \\ 0.37 \end{bmatrix}$$

We can square the individual elements of the difference vector to obtain a squared error vector. However, to get a proper feel for the *overall* error during training, we would like to obtain a single number. What we would really like to do is to *square each term of the difference vector and then add those elements to yield a single number*. Recalling equation 2.10, this is exactly what would happen if we take the *dot product of the difference vector with itself*. That happens to be the definition of the squared magnitude or length or L2-Norm of a vector: dot product of the vector with itself. In the above example, the overall training (squared) error would be

$$E^2 = (\bar{\vec{y}} - \vec{y}) \cdot (\bar{\vec{y}} - \vec{y}) = (\bar{\vec{y}} - \vec{y})^T (\bar{\vec{y}} - \vec{y})$$

$$= \begin{bmatrix} 6.31 \\ 6.04 \\ 8.87 \\ 6.45 \\ 8.95 \\ 8.57 \\ 6.2 \\ 7.55 \\ 7.49 \\ 7.99 \\ -0.09 \\ -0.22 \\ -0.16 \\ 0.63 \\ 0.37 \end{bmatrix} \cdot \begin{bmatrix} 6.31 \\ 6.04 \\ 8.87 \\ 6.45 \\ 8.95 \\ 8.57 \\ 6.2 \\ 7.55 \\ 7.49 \\ 7.99 \\ -0.09 \\ -0.22 \\ -0.16 \\ 0.63 \\ 0.37 \end{bmatrix}$$

$$= (6.31)^2 + (6.04)^2 + (8.87)^2 + (6.45)^2 + (8.95)^2 + (8.57)^2 + (6.2)^2 + (7.55)^2 + (7.49)^2 + (7.99^2)$$
$$+ (-0.09)^2 + (-0.22)^2 + (-0.16)^2 + (0.63)^2 + (0.37)^2 = 566.12$$

Formally, the length of a vector $\vec{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$, denoted $\|\vec{v}\|$, is defined as $\|\vec{v}\| = \sqrt{\vec{v}^T \vec{v}} = \sqrt{v_1^2 + v_2^2 + \cdots v_n^2}$. This quantity is sometimes called L2 norm of the vector. In particular, given a machine learning model with output vector $\vec{y}$ and a target vector $\bar{\vec{y}}$, the error is same as the magnitude or L2-norm of the difference vector

$$e = \|\bar{\vec{y}} - \vec{y}\| = \sqrt{(\bar{\vec{y}} - \vec{y}) \cdot (\bar{\vec{y}} - \vec{y})} = \sqrt{(\bar{\vec{y}} - \vec{y})^T (\bar{\vec{y}} - \vec{y})}$$

2.5.5 Geometric intuitions for Vector Length - Model Error in Machine Learning

For a 2D vector $\vec{v} = \begin{bmatrix} x \\ y \end{bmatrix}$, as seen in Figure 2.1, the L2 norm $\|\vec{v}\| = \sqrt{x^2 + y^2}$ is nothing but the hypotenuse of the right angled triangle whose sides are elements of the vector.

The same intuition holds in higher dimensions, e.g., if $\vec{v} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$, then $\|\vec{v}\| = \sqrt{x^2 + y^2}$ established from Pythagoras Theorem.

An *unit vector* is a vector whose length is 1, e.g.,[8] $\hat{v} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$ Unit vectors typically represent a direction.

In machine learning, the goal of training is often to minimize the length of the error vector (difference between model output vector and the target ground truth vector).

2.5.6 Geometric intuitions for the Dot Product - Feature Similarity in Machine Learning and Data Science

Consider the document retrieval problem depicted in Table 2.2 one more time. We have a set of documents, each described by its own feature vector. Given a pair of such documents, we have to find the "similarity" between them. This essentially boils down to estimating similarity between two feature vectors. In this section, we will see that the dot-product between a pair of vectors can be used as a measure of similarity between them.

For instance, consider the feature vectors corresponding to d_5 and d_6 in Table 2.2. They are $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$ The dot-product between them is $1 \times 0 + 0 \times 1 = 0$, low. Indeed there is no common word of interest between them and hence the documents are very dissimilar. On the other hand, the dotproduct between feature vectors of d_3 and d_4 is $\begin{bmatrix} 3 \\ 3 \end{bmatrix} \cdot \begin{bmatrix} 5 \\ 5 \end{bmatrix} = 3 \times 5 + 3 \times 5 = 30$, high. Indeed they have much commonality in words of interest and are similar documents. We will keep revisiting this problem and solve it with more and more finesse. Now we will study in greater detail how dot-products measure similarities between vectors in this sub-section. First we will show that the component of a vector along another is yielded by the dot product. Using this, we will show that the "similarity/agreement" between a pair of vectors is can be estimated using the dot product between them.

[8] unit vectors are conventionally depicted with the hat symbol as opposed to the little overhead arrow, as in $\hat{u}^T \hat{u} = 1$

DOT PRODUCT MEASURES COMPONENT OF ONE VECTOR ALONG ANOTHER

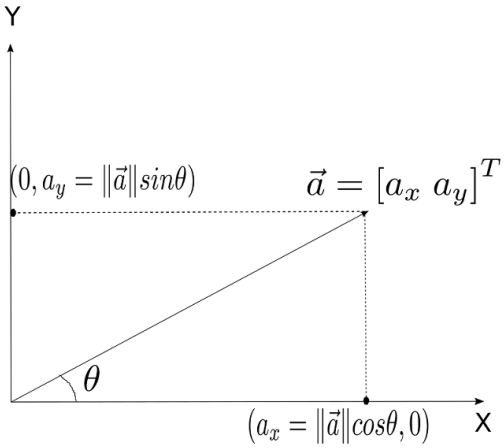


Figure 2.6: 2D Vector Components

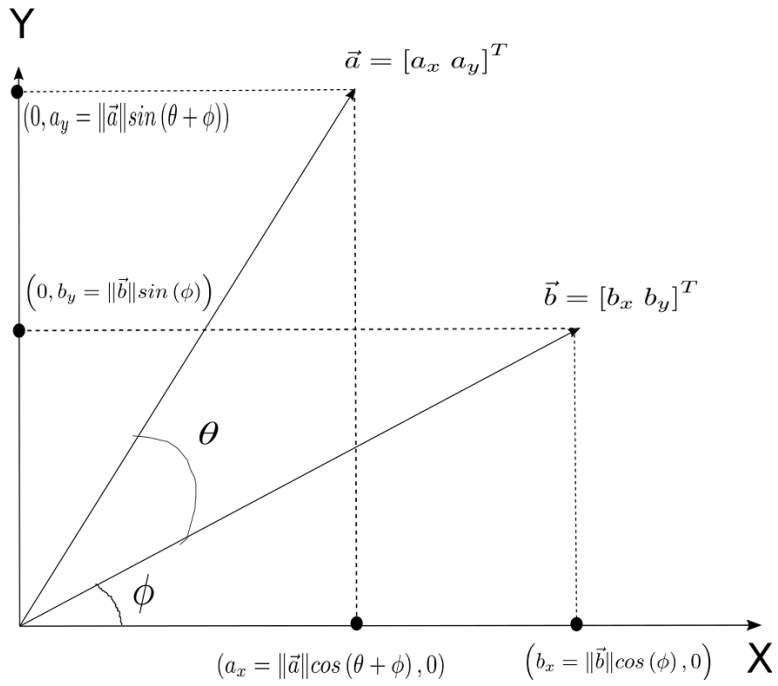


Figure 2.7: Dot Product as component of one vector along another. $\vec{a} \cdot \vec{b} = \vec{a}^T \vec{b} = a_x b_x + a_y b_y = \|\vec{a}\| \, \|\vec{b}\| cos(\theta)$

Let us examine a special case first - the component of a vector along a coordinate axis. The components of a vector along a coordinate axis can be obtained by multiplying the length of the vector with the cosine of the angle between the vector and the relevant coordinate axis. This is shown for 2D in Figure 2.6. As shown there, a vector $\vec{v}$ can be broken into two components along *X* and *Y* axes, as

$$\vec{v} = \begin{bmatrix} \|v\| \cos\theta \\ \|v\| \cos\left(90^\circ - \theta\right) \end{bmatrix} = \begin{bmatrix} \|v\| \cos\theta \\ \|v\| \sin\theta \end{bmatrix}$$

Note, how the length of the vector is preserved:

$$\left\| \begin{bmatrix} \|v\| \cos\theta \\ \|v\| \sin\theta \end{bmatrix} \right\| = \|v\| \left(\cos^2\theta + \sin^2\theta\right) = \|v\|$$

Now let us study the more general case of the component of one vector in the direction of another arbitrary vector. The component of a vector $\vec{a}$ along another vector $\vec{b}$, is $\vec{a} \cdot \vec{b} = \vec{a}^T \vec{b}$.

This is equivalent to $||\vec{a}||\ ||\vec{b}||cos(\theta)$ where θ is the angle between the vectors $\vec{a}$ and $\vec{b}$. This has been proved for the 2-dimension case in Appendix 5.1. The serious reader should read it for deeper intuitions.

DOT PRODUCT MEASURES AGREEMENT BETWEEN TWO VECTORS

The dot product can be expressed using the cosine of the angle between the vectors. Given two vectors $\vec{a}$ and $\vec{b}$, if θ is the angle between them we have (see Figure 2.7)

$$\begin{aligned} \vec{a} \cdot \vec{b} &= a_x b_x + a_y b_y \qquad \text{for two dimensions} \\ \vec{a} \cdot \vec{b} &= \vec{a}^T \vec{b} = ||\vec{a}||\ ||\vec{b}||cos(\theta) \text{ for all dimensions} \end{aligned} \tag{2.12}$$

Expressing the dot product using cosines makes it easier to see that it measures the *agreement* (aka *correlation*) between two vectors. If the vectors have the same direction, the angle between them is 0 and the cosine is 1 implying maximum agreement. The cosine progressively becomes smaller as the angle between the vectors increases until the two vectors become perpendicular to each other when the cosine becomes zero, implying no correlation - vectors are independent of each other. If the angle between them is 180° the cosine is -1 implying vectors are anti-correlated. Thus, the dot-product of two vectors is proportional to the directional agreement between them.

What role does the vector lengths play in all this? The dot product between two vectors is also proportional to the lengths of the vectors. This means agreement scores between bigger vectors will be higher (agreement between US president and German Chancellor counts more than agreement between you and me).

If one wants the agreement score to be neutral to the vector length, one can use a normalized dot product - between unit length vectors along same directions.

$$cosine\_similarity\left(\vec{a},\vec{b}\right) = \frac{\vec{a}^T\vec{b}}{\|\vec{a}\|\|\vec{b}\|} = cos\left(\theta\right)$$

The normalized do product (aka cosine similarity measure) indicates pure directional agreement. It is often used in document processing. Suppose we have some query text which we want to match against various archive documents. We want to retrieve archived documents rank ordered by their similarity to the query text. We will have descriptor vector corresponding to the query text as well as each archive document. We can use dot product between descriptor vectors as a measure of similarity, but we do *not* want longer documents to automatically score higher in similarity. Rather we want the similarity score to be independent of the length of the document. Cosine similarity is often used here. Document retrieval and cosine similarity are discussed in detail in section 4.5.1.

DOT PRODUCT AND DIFFERENCE BETWEEN TWO UNIT VECTORS

To obtain further insight into how the dot product indicates agreement or correlation between two directions, consider two unit vectors $\hat{u} = \begin{bmatrix} u_x \\ u_y \\ u_z \end{bmatrix}$, and $\hat{v} = \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}$ the difference between them is $\hat{u} - \hat{v} = \begin{bmatrix} u_x - v_x \\ u_y - v_y \\ u_z - v_z \end{bmatrix}$.

Note that, since they are unit vectors, $\|\hat{u}\| = \sqrt{u_x^2 + u_y^2 + u_z^2} = \|\hat{v}\| = \sqrt{v_x^2 + v_y^2 + v_z^2} = 1.$ The length of difference vector

$$\begin{aligned}\|\hat{u} - \hat{v}\| &= \sqrt{\left(u_x - v_x\right)^2 + \left(u_y - v_y\right)^2 + \left(u_z - v_z\right)^2} \\ &= \sqrt{u_x^2 + u_y^2 + u_z^2 + v_x^2 + v_y^2 + v_z^2 - 2\left(u_x v_x + u_y v_y + u_z v_z\right)} \\ &= \sqrt{2 - 2\hat{u}^T\hat{v}} = \sqrt{2\left(1 - \hat{u}\cdot\hat{v}\right)}\end{aligned}$$

From the last equality, it is evident that larger dot product implies a smaller difference, i.e., more agreement between the vectors.

2.6 Orthogonality of Vectors and its physical significance

Try moving an object at right angles to the direction in which you are pushing it. You will find it is impossible. In fact, the larger the angle, the less effective your force vector becomes (finally becoming totally ineffective at 90° angle). This is the reason why it is easy to walk on a horizontal surface (you are moving at right angles to the direction of gravitational pull, gravity vector is ineffective) but harder on an upwards incline (gravity vector is having some effect against you). These physical notions are captured mathematically in the notion of dot product. The dot product between two vectors $\vec{a}$ (say the push vector) and $\vec{b}$ (say the displacement of the pushed object vector) is $||\vec{a}||\ ||\vec{b}||cos(\theta)$ where θ is the angle between the two vectors. When θ is 0 - the two vectors are aligned - $cos\theta$ = 1, maximum possible value of

$cos\theta$ - push is maximally effective. As θ increases, $cos\theta$ decreases, push becomes less and less effective. Finally, at θ = 90°, $cos\theta$ = 0 and push becomes completely ineffective.

Two vectors are orthogonal if their dot product is zero. Geometrically, this means the vectors are perpendicular to each other. Physically, this means the two vectors are independent, one cannot influence the other. One can say, there is nothing in common between orthogonal vectors. For instance, the feature vector for d_5 is $\begin{bmatrix}1\\0\end{bmatrix}$ and that for d_6 is $\begin{bmatrix}0\\1\end{bmatrix}$ in Table 2.2. These are orthogonal (dot-product is zero) and one can easily see that none of the feature words ("gun", "violence") are common to both the documents.

2.7 Python code: Basic Vector and Matrix operations via Numpy

In this section we will demonstrate python numpy code to illustrate many of the concepts discussed above.

Fully functional code for this section, executable via Jupyter-notebook, can be found at https://nbviewer.jupyter.org/github/krishnonwork/mathematical-methods-in-deep-learning-ipython/blob/master/python/ch2/2.7-transpose-dot-matmul-numpy-pytorch.ipynb.

2.7.1 Python numpy code for Matrix Transpose

Listing 2.7: Transpose

```
Numpy arange function creates a vector whose elements go from start to stop in increments of step
 Here we create a 4 × 9 image corresponding to I4,9 in equation 2.2, shown in Fig. 2.2
I49 = np.array([np.arange(0, 72, 8), np.arange(64, 136, 8),
                np.arange(128, 200, 8), np.arange(192, 264, 8)])
Transpose operator interchanges row and columns. The 4 × 9 image becomes 9 × 4 image, shown in Fig. 2.5.
    Element at position (i, j) gets interchanged with element at position (j, i).
I49_t = np.transpose(I49)

                                          Assert that elements of original and transposed matrix
                                                have interchanged rows and columns
for i in range(0, I49.shape[0]):
    for j in range(0, I49.shape[1]):
        assert I49[i][j] == I49_t[j][i]

                                          The .T operator retrieves the transpose of an array
assert np.allclose(I49_t, I49.T, 1e-5)
```

2.7.2 Python numpy code for Dot product

The dot product of two vectors $\vec{a}$ and $\vec{b}$ represents the component of one vector along the other.

Consider two vectors $\vec{a}$= [a_1 a_2 a_3] and $\vec{b}$=[b_1 b_2 b_3]. Then $\vec{a}.\vec{b}$= a_1b_1 + a_2b_2 + a_3b_3.

Listing 2.8: Dot product

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

```
3 a_dot_b = np. dot(a, b)
4 print ("Dot product of these two vectors is: "
5                "{}". format ( a_dot_b ))
6
7 # Dot product of perpendicular vectors is zero
8 vx = np. array ([1 , 0]) # a vector along X- axis
9 vy = np. array ([0 , 1]) # a vector along Y- axis
10 print (" Example dot product of orthogonal vectors :"
11                " {}". format (np.dot(vx , vy)))
```

Output:

```
1 Dot product of these two vectors is: 32
2 Example dot product of orthogonal vectors : 0
```

2.7.3 Python numpy code for Matrix vector multiplication

Consider a matrix $A_{m,n}$ with m rows and n columns which is multiplied with a vector $\vec{b_n}$ with n elements.

Below we show an example with $m = 3$ and $n = 2$.

The resultant vector $\vec{c}_m$ is:

$$\begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

$$c_1 = a_{11}b_1 + a_{12}b_2$$

$$c_2 = a_{21}b_2 + a_{22}b_2$$

$$c_3 = a_{31}b_2 + a_{32}b_2$$

In general

$$c_i = a_{i1}b_1 + a_{i2}b_2 + \ldots + a_{in}b_n$$

Listing 2.9: Matrix vector multiplication

A linear model comprises a weight vector $\vec{w}$ and bias b. Given an input $\vec{x}$, the model outputs the dot product of input and weight vector plus bias, $y = \vec{x}^T\vec{w} + b$. For each training data instance, $\vec{x}_i$, we have output $y_i = \vec{x}_i^T\vec{w} + b$. For training data matrix X (whose rows are training data instances), we can write compactly $X\vec{w} + b = \vec{y}$.

cat-brain training data matrix, 15×2

```
X = np.array([[0.11, 0.09], [0.01, 0.02], [0.98, 0.91], [0.12, 0.21],
              [0.98, 0.99], [0.85, 0.87], [0.03, 0.14], [0.55, 0.45],
              [0.49, 0.51], [0.99, 0.01], [0.02, 0.89], [0.31, 0.47],
              [0.55, 0.29], [0.87, 0.76], [0.63, 0.24]])
w = np.random.random((2, 1))
b = 5.0

y = np.matmul(X, w) + b
```

random initialization of weight vector

Model training output: $\vec{y} = X\vec{w} + b$, compactly expressed as matrix vector multiplication. The scalar b is automatically replicated to create vector.

2.7.4 Python numpy code for Matrix Matrix Multiplication

Consider a matrix $A_{m,p}$ with m rows and p columns. Let us multiply it with another matrix $B_{p,n}$ with p rows and n columns. The resultant matrix $C_{m,n}$ will contain m rows and n columns. Note that number of columns in the left matrix A should match the number of rows in the right matrix B.

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \\ c_{31} & c_{32} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

$$c_{11} = a_{11}b_{11} + a_{12}b_{21}$$
$$c_{12} = a_{11}b_{12} + a_{12}b_{22}$$
$$c_{21} = a_{21}b_{11} + a_{22}b_{21}$$
$$c_{22} = a_{21}b_{12} + a_{22}b_{22}$$
$$c_{31} = a_{31}b_{11} + a_{32}b_{21}$$
$$c_{32} = a_{31}b_{12} + a_{32}b_{22}$$

In general

$$c_{ij} = \sum_{i=1}^{p} a_{ip} b_{pj}$$

Listing 2.10: Matrix matrix multiplication

```
C = AB ⟹ C[i,j] is dot product of i^th row of A and j^th column of B.
A = np.array([[1, 2], [3, 4], [5, 6]])
B = np.array([[7, 8], [9, 10]])
C = np.matmul(A, B)

w = np.array([1, 2, 3])
x = np.array([4, 5, 6])
assert np.dot(w, x) == np.matmul(w.T, x)
```

$$\begin{bmatrix}1 & 2\\ 3 & 4\\ 5 & 6\end{bmatrix}\begin{bmatrix}7 & 8\\ 9 & 10\end{bmatrix} = \begin{bmatrix}25 & 28\\ 57 & 64\\ 89 & 100\end{bmatrix}$$

Dot product can be viewed as a row matrix multiplied by a column matrix

2.7.5 Python numpy code for Transpose of Matrix Product

Given two matrices A and B where the number of columns of A matches the number of rows of B, the transpose of their product is the product of the individual transposes *in reversed order*. $(AB)^T = B^T A^T$

Listing 2.11: Transpose of Matrix product

```
(AB)^T                                          B^T A^T
assert np.all(np.matmul(A, B).T == np.matmul(B.T, A.T))
assert equality
assert np.all(np.matmul(A.T, x).T == np.matmul(x.T, A))
applies to matrix-vector multiplication too, (A^T x)^T = x^T A
```

2.7.6 Python numpy code for Matrix Inverse

Inverse of a matrix A is another matrix, denoted A^{-1} such that $AA^1 = A^{-1}A = \mathbb{I}$. This equation is the peer of the scalar world equation $aa^{-1} = 1$ for any constant *a*, e.g., $77^{-1} = 1$. $\mathbb{I}$ is the counterpart of the scalar quantity 1 and inverse of matrix A is the counterpart of the scalar 45 operation raising to power -1.

$\mathbb{I}$ is the so called *identity matrix*, defined as a matrix which has 1 in diagonal, 0 elsewhere. The 2 × 2 identity matrix is defined as $\mathbb{I} = \begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}$ The 3 × 3 identity matrix is defined as $\mathbb{I} = \begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix}$. The generic $n \times n$ identity matrix is $\mathbb{I} = \begin{bmatrix}1 & 0 & 0 & \cdots & 0\\ 0 & 1 & 0 & \cdots & 0\\ 0 & 0 & 1 & \cdots & 0\\ \vdots & \vdots & \vdots & \ddots & \vdots\\ 0 & 0 & 0 & \cdots & 1\end{bmatrix}$

Why do we need the inverse?

Let us say we want to solve a simultaneous equation with two variables x_1 and x_2, Such an equation can be written as

$$a_{11}x_1 + a_{12}x_2 = b_1$$
$$a_{21}x_1 + a_{22}x_2 = b_2$$

This can be written using matrices and vectors as

$$A\vec{x} = \vec{b}$$

where

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \vec{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \vec{b} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

Solution of $A\vec{x} = \vec{b}$ is

$$\vec{x}=A^{-1}\vec{b}$$

where A^{-1} is the matrix inverse, (assumed $det(A) \neq 0$). Compare this with the scalar equation $ax = b$ whose solution is $x = a^{-1}b$.

The determinant can be computed as

$$det(A) = a_{11}a_{22} - a_{12}a_{21}$$

The inverse is

$$A^{-1} = \frac{1}{det(A)} \begin{bmatrix} a_{22} & -a_{12} \\ -a_{21} & a_{11} \end{bmatrix}$$

Although the above example is shown with a small 2× system of simultaneous equations, the code below is general and works for arbitrary sized linear systems.

Listing 2.12: Matrix inverse for invertible matrix (non-zero determinant)

```
def determinant(A):
    return np.linalg.det(A)

def inverse(A):
    return np.linalg.inv(A)

A = np.array([[2, 3], [2, 2]], dtype=np.uint8)

A_inv = inverse(A)

I = np.eye(2)
assert np.all(np.matmul(A, A_inv) == I)

assert np.all(np.matmul(I, A) ==  A)\ and np.all(A == np.matmul(A,I))
assert np.all(np.matmul(A, A_inv) == I)\ and np.all(np.matmul(A_inv,
    A) == I)
```

$A = \begin{bmatrix} 2 & 3 \\ 3 & 2 \end{bmatrix}$

$A^{-1} = \begin{bmatrix} -1 & 1.5 \\ 1 & -1 \end{bmatrix}$

Numpy function $np.eye(n)$ generates identity matrix, I of size n

Verify $\begin{bmatrix} 2 & 3 \\ 3 & 2 \end{bmatrix} \begin{bmatrix} -1 & 1.5 \\ 1 & -1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

$\mathbb{I}$ is like 1. Verify $A\mathbb{I} = \mathbb{I}A = A$

Listing 2.13: Singular matrix

```
B = np.array([[1, 1], [2, 2]], dtype=np.uint8)
try:
    B_inv = inverse(B)
except np.linalg.LinAlgError as e:
    print("B cannot be inverted: {}\n".format(B, e))
```

$B = \begin{bmatrix} 1 & 1 \\ 2 & 2 \end{bmatrix}$, determinant $= 1 \times 2 - 2 \times 1 = 0$, singular.

Attempt to compute inverse causes python exception

2.8 Multidimensional Line and Plane Equations and their role in Machine Learning

Geometrically speaking, what does a machine learning classifier really do? We provided the outline of an answer in section 1.4. The reader is invited to review that and especially Figures 1.2, 1.3.

We will briefly recapitulate here.

Inputs to a classifier are feature vectors. These vectors can be viewed as points in some multidimensional feature space. The task of classification then boils down to separating the points belonging to different classes. The points maybe all jumbled up in the input space. It is the job of the model to transform them to a different (output) space where it is easier to separate the classes. A visual example of this was provided in Figure 1.3.

What is the geometrical nature of the separator? In a very simple situation, such as the one depicted in Figure 1.2 the separator is a line in 2D space. In real life situations, the separator is often a line or a plane in a high dimensional space. In more complicated situations, the separator is a curved surface, as depicted in Figure 1.4.

In this section, we will study the mathematics and geometry behind two types of separators, lines and planes in high dimensional spaces, aka hyper-lines and hyper-planes.

2.8.1 Multidimensional Line Equation

In high school geometry we learnt $y = mx + c$ as the equation of a line. But this does not lend itself readily to higher dimensions. Here we will study a better representation for a straight line that works equally well for any finite dimensional space.

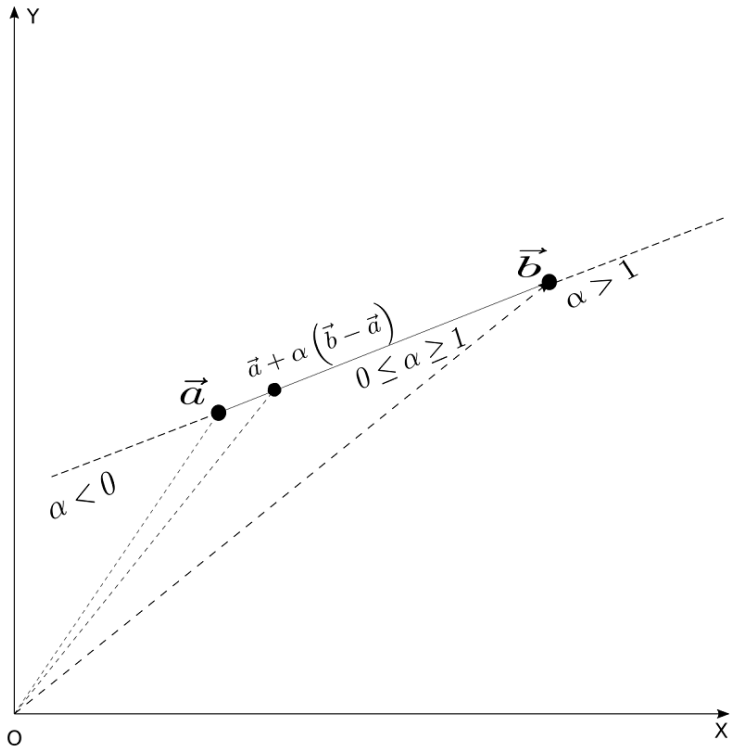


Figure 2.8: Any point $\vec{x}$ on the line joining two vectors $\vec{a}$, $\vec{b}$ is given by $\vec{x} = \vec{a} + \alpha(\vec{b} - \vec{a})$. By varying a, we can get any point on the line. $\alpha = 0$ yields $\vec{a}$. $\alpha = 1$ yields $\vec{b}$. Values of α in between 0 and 1 yields points in between $\vec{a}$ and $\vec{b}$. Negative values of α yields points to the left of $\vec{a}$. Greater than 1 values of α yields points to the right of $\vec{b}$. This equation of line works for any dimension, not just 2.

As shown in Figure 2.8, a line joining vectors $\vec{a}$ and $\vec{b}$ can be expressed as

$$\vec{x} = \vec{a} + \alpha(\vec{b} - \vec{a}) = (1 - \alpha)\vec{a} + \alpha\vec{b} \tag{2.13}$$

$$\Leftrightarrow (1 - \alpha)\vec{a} + \alpha\vec{b} - \vec{x} = 0 \tag{2.14}$$

What equation 2.14 is saying is this: any point on the line can be obtained by by varying *a*. Different ranges of a values yield different segments on the line. As shown in Fig 2.8, values of *a* in between 0 and 1 yields points in between $\vec{a}$ and $\vec{b}$. Negative values of *a* yields points to the left of $\vec{a}$. Greater than 1 values of *a* yields points to the right of $\vec{b}$. This equation of line works for any dimension, not just 2.

2.8.2 Multidimensional Planes and their role in Machine Learning

In section 1.5 we encountered classifiers. Lets take another look at them. Suppose we want to create a classifier that helps us to make *buy* or *no-buy* decisions on stocks based on only 3 input variables: (i) *Momentum* - rate at which the stock price is changing. Positive momentum means stock price is increasing and vice versa. (ii) *Dividend* paid last quarter (iii) *Volatility* - how much price fluctuations have been seen in last quarter. Let us plot all training points in the feature space with coordinate axes corresponding to the variables *momentum*, *dividend*, *volatility*. It can be seen that the classes can be separated by a plane in the 3 dimensional feature space (Figure 2.9).

Geometrically speaking, our model simply corresponds to this plane. Input points that are above the plane indicate buy decision (green circles) and input points below indicate no-buy decision (red circles). In general, one wants to buy high positive momentum stocks hence points at the higher end of the momentum axis are likelier to be *buy*. However, this is not the only indicator. For more volatile stocks, one demands higher *momentum* to switch from *no-buy* to *buy*. This is why the plane slopes upwards (higher *momentum*) as we move rightwards (higher *volatility*). Also, one demands less *momentum* for stocks with higher *dividends*. This is why the plane slopes downwards (lower *momentum*) as we go towards higher *dividends*.

Real problems will of course have more dimensions (since many more inputs will be involved in the decision) and the separator will become a hyper-plane. Also, in real life problems, often the points will be too intertwined in the input space for any separator to work. We will first have to apply a transformation that maps the point to an output space where it is easier to separate. Given their significance as class separators in machine learning, we will study hyper-planes in this subsection.

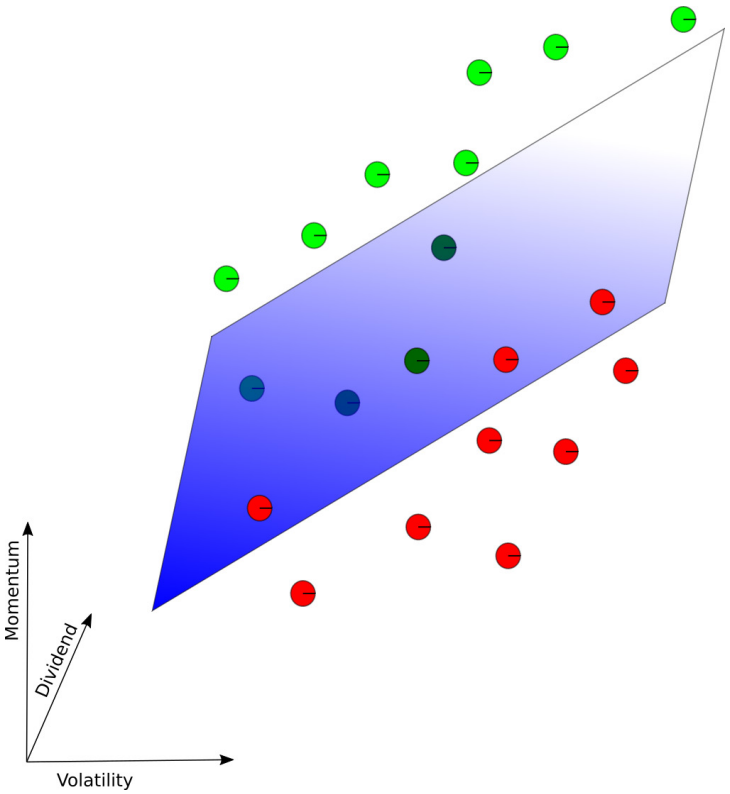


Figure 2.9: A toy machine learning classifier for stock buy vs no-buy decision making. Red indicates no-buy, green indicates buy. The decision is made from 3 input variables: (i) Momentum (ii) Dividend(iii) Volatility.

In high school 3D geometry we learnt $ax + by + cz + d = 0$ as the equation of a plane. Now we are going to study a version of it that works in higher dimensions. Geometrically speaking, given a plane (in any dimension), we will be able to find a direction, called *normal direction*, denoted $\hat{n}$, such that

- If we take any pair of points on the plane: say $\vec{x}_0$ and $\vec{x}$
- The line joining $\vec{x}$ and $\vec{x}_0$, i.e., the vector $\vec{x} - \vec{x}_0$, is orthogonal to $\hat{n}$

Thus, if we know a fixed point on the plane, say $\vec{x}_0$, then all points on the plane will satisfy

$\hat{n} \cdot (\vec{x} - \vec{x}_0) = 0$ or

$\hat{n}^T (\vec{x} - \vec{x}_0) = 0$

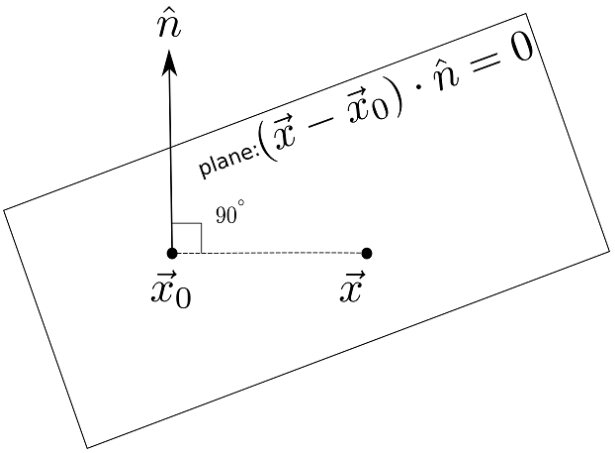


Figure 2.10: The normal to the plane is same at all points on the plane.This is the fundamental property of a plane. $\hat{n}$ depicts that normal direction. Let $\vec{x}_0$ be a point on the plane. All other points on the plane, depicted as $\vec{x}$ will satisfy the equation $(\vec{x} - \vec{x}_0)\ \hat{n} = 0$. This physically says that the line joining $\vec{x}_0$ and $\vec{x}$ is at right angles to $\hat{n}$. This formulation works for any dimension.

Since $\hat{n}$ and $\vec{x}_0$ are constants, $-\hat{n}^T \vec{x}_0$ is also a scalar constant, say b. Thus we can express the equation of a plane as

$$\hat{n}^T \vec{x} + b = 0 \tag{2.15}$$

In section 1.3, equation 1.3, we encountered the simplest machine learning model - just a weighted sum of inputs along with a bias. Denoting the inputs as $\vec{x}$, the weights as $\vec{w}$ and the bias as b, this model was depicted as $\vec{w}^T \vec{x} + b \quad \vec{w}^T \vec{x} + b$. Comparing with equation 2.15, we get the geomeric significance - the simple model of equation 1.3 is nothing but a planar separator with the weight vector $\vec{w}$ corresponding to the plane's normal. During training we are learning the weights - this is essentially learning the optimal plane that will separate the training inputs. In order to be consistent with the machine learning paradigm, henceforth we will write the equation of a hyper-plane as

$$\vec{w}^T \vec{x} + b = 0 \tag{2.16}$$

for some constant $\vec{w}$ and b. Note that $\vec{w}$ need not be an unit length vector. Since the right hand side is zero, if necessary, we can divide both sides by $||\vec{w}||$ to convert to a form like equation 2.15.

The sign of the expression $\vec{w}^T \vec{x} + b$ has special significance. All points $\vec{x}$ for which $\vec{w}^T \vec{x} + b < 0$ lie on the same side of the hyper-plane. All points $\vec{x}$ for which $\vec{w}^T \vec{x} + b > 0$ lie on the other

side of the hyper-plane. And of course, all points $\vec{x}$ for which $\vec{w}^T\vec{x} + b = 0$ lie on the hyper-plane.

It should be noted that the 3D equation $ax + by + cz + d = 0$ is a special case of equation 2.16 because

$ax + by + cz + d = 0$ can be rewritten as

$$\begin{bmatrix} a & b & c \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} + d = 0$$

which is same as $\vec{w}^T\vec{x} + b = 0$ with $\vec{w} = \begin{bmatrix} a \\ b \\ c \end{bmatrix}$ and $\vec{x} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$.

Incidentally, this tells us that in 3*D*, the normal to the plane $ax + by + cz + d = 0$ is

$$\hat{n} = \frac{1}{\sqrt{a^2+b^2+c^2}} \begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

2.9 Linear Combination, Linear Dependence, Vector Span and Basis Vectors, their Geometrical Significance, Collinearity Preservation

By now, it should be clear that machine learning and data science is all about points in high dimensional spaces. Consequently, it behooves us to have a decent understanding of these spaces. For instance, given a space, we may need to ask, would it be possible to express all points in the space in terms of a set of few vectors? What is the smallest set of vectors we really need for that purpose? This section is devoted to the study of these questions.

LINEAR DEPENDENCE

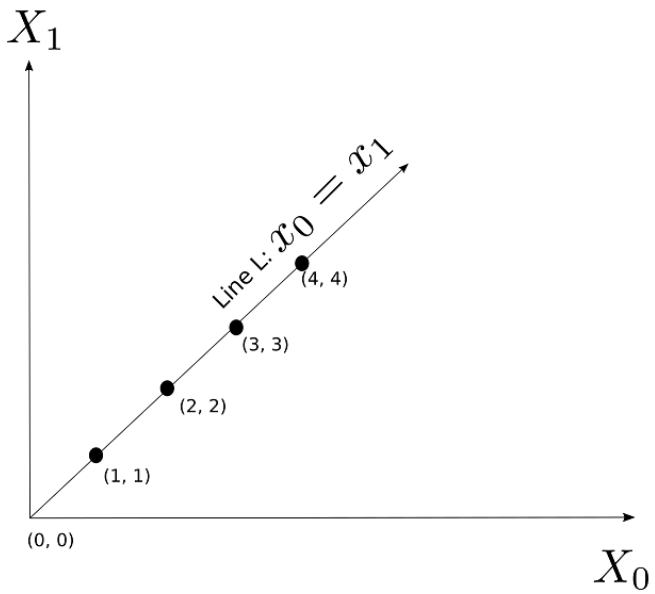


Figure 2.11: Linearly dependent points in a 2D plane.

Consider the vectors (points) shown in Figure 2.11. The corresponding vectors in 2D are

$$\vec{v}_0 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \qquad \vec{v}_1 = \begin{bmatrix} 2 \\ 2 \end{bmatrix}$$

$$\vec{v}_2 = \begin{bmatrix} 3 \\ 3 \end{bmatrix} \qquad \vec{v}_3 = \begin{bmatrix} 4 \\ 4 \end{bmatrix}$$

We can find find 4 scalars $\alpha_0 = 2$, $\alpha_1 = 2$, $\alpha_2 = 2$ and $\alpha_3 = -3$ such that

$$\alpha_0\vec{v}_0 + \alpha_1\vec{v}_1 + \alpha_2\vec{v}_2 + \alpha_3\vec{v}_3 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

If we can find such scalars, not all zero, we say the vectors $\vec{v}_0$, $\vec{v}_1$, $\vec{v}_2$, $\vec{v}_3$, are *linearly dependent*. The geometric picture to keep in mind is that points corresponding to linearly dependent vectors lie on a single straight line in the space containing them.

COLLINEARITY IMPLIES LINEAR DEPENDENCE

Proof: Let $\vec{a}$, $\vec{b}$ and $\vec{c}$ be three collinear vectors. From equation 2.14 there exists some $\alpha \in \mathbb{R}$ such that

$$\vec{c} = (1 - \alpha)\vec{a} + \alpha\vec{b}$$

The above equation can be rewritten as

$$\alpha_1\vec{a} + \alpha_2\vec{b} + \alpha_3\vec{c} = 0$$

where $\alpha_1 = (1 - \alpha)$, $\alpha_2 = \alpha$ and $\alpha_3 = -1$. Thus we have proved that 3 collinear vectors $\vec{a}$, $\vec{b}$ and $\vec{c}$ must also be linearly dependent.

LINEAR COMBINATION

Given a set of vectors $\vec{v}_1$, $\vec{v}_2$, ... $\vec{v}_n$ and a set of scalar weights α_1,α_2, ... α_n, the weighted sum $\vec{v}_1$, $\vec{v}_2$, ... $\vec{v}_n$ is called a *linear combination*.

GENERIC MULTI-DIMENSIONAL DEFINITION OF LINEAR DEPENDENCE

A set of vectors $\vec{v}_1$, $\vec{v}_2$, ... $\vec{v}_n$ are linearly dependent if there exists a set of weights α_1,α_2, ... α_n, not all zeros, such that $\alpha_1\vec{v}_1$,$\alpha_2\vec{v}_2$, ... $\alpha_n\vec{v}_n = 0$. E.g., the row vectors $[1\ 1]$ and $[2\ 2]$ are linearly dependent, since $-2[1\ 1] + [2\ 2] = 0$.

SPAN OF A SET OF VECTORS

Given a set of vectors $\vec{v}_1$, $\vec{v}_2$, ... $\vec{v}_n$, their *span* is defined as the set of all vectors that are linear combinations of the original set . This includes the original vectors.

E.g., Consider the 2 vectors $\vec{v}_{x\perp} = \begin{bmatrix}1\\0\end{bmatrix}$ and $\vec{v}_{y\perp} = \begin{bmatrix}0\\1\end{bmatrix}$.

The span of these 2 vectors is the entire plane containing these 2 vectors. For instance, the vector $\begin{bmatrix}18\\97\end{bmatrix}$ can be expressed as a weighted sum $18\vec{v}_{x\perp}+97\vec{v}_{y\perp}$ You can probably recognize that $\begin{bmatrix}1\\0\end{bmatrix}$ and $\begin{bmatrix}0\\1\end{bmatrix}$ are the familiar Cartesian coordinate axes (*X* axis and *Y* axis respectively) in 2D plane.

VECTOR SPACES, BASIS VECTORS AND CLOSURE

We have been talking informally about vector spaces. It is time to define them more precisely. **Vector Space**: A set of vectors (points) in *n* dimensions form a vector space if and only if the operations of *addition* and *scalar multiplication* are defined on the set. In particular, the above implies that it is possible to take linear combinations of members of a vector space. **Basis Vectors**: Given a vector space, a set of vectors that span the space is called a basis for the space.

For instance, for the space $\mathbb{R}^2$, the two vectors $\begin{bmatrix}1\\0\end{bmatrix}$ and $\begin{bmatrix}0\\1\end{bmatrix}$ are basis vectors. This essentially means that any vector in $\mathbb{R}^2$ can be expressed as a linear combination of these two.

The notion can be extended to higher dimensions. For $\mathbb{R}^n$, the vectors $\begin{bmatrix}1\\0\\\vdots\\0\end{bmatrix}, \begin{bmatrix}0\\1\\\vdots\\0\end{bmatrix}, \cdots, \begin{bmatrix}0\\0\\\vdots\\1\end{bmatrix}$ form a basis.

The alert reader has probably guessed by now that the basis vectors are related to coordinate axes. In fact, the basis vectors described above constitute the Cartesian coordinate axes. So far, we have only seen examples of basis vectors that are mutually orthogonal, e.g., the dot product of the two basis vectors in $\mathbb{R}^2$ shown above $\begin{bmatrix}0\\1\end{bmatrix} \cdot \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}0 & 1\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix} = 0$.

However, basis vectors do not have to be orthogonal. Any pair of linearly independent vectors forms a basis in $\mathbb{R}^2$. That said, orthogonal vectors are most convenient, as we shall see later. Basis vectors, then, are by no means unique.

Minimal and Complete Basis: Exactly n vectors are needed to span a space with dimensionality n. This means, the basis set for a space will have at least as many elements as the dimensionality of the space. That many basis vectors are also sufficient to form basis, that is we do not need any more to span the space. For instance, exactly n vectors are needed to form a basis in (i.e., span) $\mathbb{R}^n$.

A related fact is that, in $\mathbb{R}^n$ any set of m vectors, with $m > n$ will be linearly dependent. In other words, the largest size of a set of linearly independent vectors in a n-dimensional space is n. **Closure**: A set of vectors is said to be *closed* under linear combination if and only if the linear combination of any pair of vectors in the set also belongs to the same set. Consider the set of points $\mathbb{R}^2$. Recall, this is the set of vectors with 2 real elements. Take any pair of vectors, $\vec{a}$ and $\vec{b}$ in $\mathbb{R}^2$. For instance, say, $\vec{a} = \begin{bmatrix}11.2\\31.766\end{bmatrix}$ and $\vec{b} = \begin{bmatrix}177.01\\1031.99\end{bmatrix}$.

Any linear combination of these two vectors will also comprise two real numbers, i.e., will belong to $\mathbb{R}^2$. We say $\mathbb{R}^2$ is a *vector space* since it is *closed* under linear combination.

Consider the space $\mathbb{R}^2$. Geometrically speaking, this represents a 2D plane. Let us take two points on this plane, $\vec{a}$ and $\vec{b}$. A linear combination of them, geometrically corresponds to the line joining them. We know that if two points lie on a plane, all points on the line will also lie on the plane. This is the geometrical intuition behind the notion of closure on vector spaces. It can be extended to arbitrary dimensions.

On the other hand, the set of points on the surface of a sphere is not closed under linear combination, because, the line joining an arbitrary pair of of points on this set will not wholly lie on the surface of that sphere.

2.10 Linear Transforms - Geometric and Algebraic interpretations

Inputs to a machine learning or data science system are typically feature vectors (introduced in section 2.1) in high dimensional spaces. Each individual dimension of the feature vector corresponds to a particular property of the input. The feature vector, thus, is a descriptor for the particular input instance. It can be viewed as a point in the feature space. We usually transform the points to a friendlier space where it is easier to perform the analysis we are trying to do. For instance, if we are building a classifier, we try to transform the input to a space where the points belonging to different classes are more segregated (see section 1.3 in general and Fig 1.3 in particular for simple examples). Sometimes we transform to simplify the data, eliminating axes along which there is scant variation in the data. Given their significance in machine learning, in this section we will study the basics of transforms.

Consider the matrix

$$R = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$$

In section 2.14 we will see that this is a special kind of matrix called rotation matrix - for now simply consider it as an example of matrix. R can be thought of as a transformation operator that maps a point in 2D plane to another point in the same plane. In mathematical notation, $R : \mathbb{R}^2 \rightarrow \mathbb{R}^2$. In fact, as depicted in Figure 2.14, multiplication by this particular R rotates the position vector of a point in 2D plane by an angle $45°$.

The output and input points may belong to different spaces in a transform. For instance, consider the matrix

$$P = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

It is not hard to see that this matrix projects 3D points to the 2D X-Y plane.

$$P \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix}$$

We say the projection matrix $P : \mathbb{R}^3 \rightarrow \mathbb{R}^2$.

The transforms R and P share a common property - *they preserve collinearity*. This means, a set of vectors (points) $\vec{a}, \vec{b}, \vec{c} \cdots$ that originally lay on a straight line remain so after the transformation.

Let us check this out for the rotation transformation, on the example from section 2.9. There we saw four vectors

$$\vec{o} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \qquad \vec{a} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$
$$\vec{b} = \begin{bmatrix} 2 \\ 2 \end{bmatrix} \qquad \vec{c} = \begin{bmatrix} 3 \\ 3 \end{bmatrix}$$

These vectors all lie on a straight $L : x = y$. The transformed versions of these vectors are

$$\vec{o}' = R\vec{o} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \qquad \vec{a}' = R\vec{a} = \begin{bmatrix} \sqrt{2} \\ 0 \end{bmatrix}$$
$$\vec{b}' = R\vec{b} = \begin{bmatrix} 2\sqrt{2} \\ 0 \end{bmatrix} \qquad \vec{c}' = R\vec{c} = \begin{bmatrix} 3\sqrt{2} \\ 0 \end{bmatrix}$$

It is trivial to see that the transformed vectors also lie on a (different) straight line. In fact, $\vec{o}'$, $\vec{a}'$ $\vec{b}'$ $\vec{c}'$ lie on the γ axis, which is the 45° rotated version of the original line $y = x$.

The projection transform represented by matrix P also preserves collinearity. Consider four collinear vectors in 3D

$$\vec{o} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \qquad \vec{a} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$
$$\vec{b} = \begin{bmatrix} 2 \\ 2 \\ 2 \end{bmatrix} \qquad \vec{c} = \begin{bmatrix} 3 \\ 3 \\ 3 \end{bmatrix}$$

The corresponding transformed vectors

$$\vec{o}' = P\vec{o} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \qquad \vec{a}' = P\vec{a} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$
$$\vec{b}' = P\vec{b} = \begin{bmatrix} 2 \\ 2 \end{bmatrix} \qquad \vec{c}' = P\vec{c} = \begin{bmatrix} 3 \\ 3 \end{bmatrix}$$

also lie on a straight line in 2D.

The class of transforms that preserve collinearity is known as linear transform. A more formal definition is provided below.

GENERIC MULTI-DIMENSION DEFINITION OF LINEAR TRANSFORM

A function ϕ is a linear transform if and only if it satisfies

$$\phi(\alpha\vec{a} + \beta\vec{b}) = \alpha\phi(\vec{a}) + \beta\phi(\vec{b}) \quad \forall\, \alpha, \beta \in \mathbb{R} \tag{2.17}$$

In other words, a transform is linear if and only if the transform of the linear combination of two vectors is same as the linear combination (with same weights) of the transforms of individual vectors.

Note: This can be remembered as "linear transform means transforms of linear combinations are same as linear combinations of transforms*".*

Multiplication with a rotation or projection matrix (shown above) are linear transforms.

ALL MATRIX VECTOR MULTIPLICATIONS ARE LINEAR TRANSFORMS

Let us verify here that matrix multiplication satisfies the definition of linear mapping (equation 2.17). Let $\vec{a}, \vec{b} \in \mathbb{R}^n$ be two arbitrary *n*-dimensional vectors and $A_{m,n}$ be an arbitrary matrix with *n* columns. Then, following standard rules of matrix vector multiplication

$$A(\alpha\vec{a} + \beta\vec{b}) = \alpha(A\vec{a}) + \beta\,\phi(A\vec{b})$$

which mimics equation 2.17 with ϕ replaced with matrix *A*. Thus we have proved that all matrix multiplications are linear transforms. The vice versa is not true. In particular, linear transforms that operate on infinite dimensional vectors are not matrices. But all linear transforms that operate on finite dimensional vectors are expressible as matrices. The proof is a bit more complicated and will be skipped.

Thus, in finite dimensions, multiplication with a matrix and linear transformation are one and the same thing. In section 2.3 we saw the array view of matrices. The corresponding geometric view, that all matrices represent linear transformation was presented in this section.

Let us finish this section by studying an example of a transform that is *not* linear. Consider the function

$$\phi(\vec{x}) = \|\vec{x}\|$$

for $\vec{x} \in \mathbb{R}$. This function ϕ maps *n*-dimensional vectors to a scalar which is the length of the vector, ϕ: $\mathbb{R}^n \rightarrow \mathbb{R}$. We will examine if it satisfies equation 2.17 with $\alpha_1 = \alpha_2 = 1$. For two specific vectors $\vec{a}, \vec{b} \in \mathbb{R}^n$

$$\phi(\vec{a}) = \phi\left(\begin{bmatrix} a_1 \\ a_2 \\ \cdots \\ a_n \end{bmatrix}\right) = \sqrt{a_1^2 + a_2^2 + \cdots a_n^2}$$

$$\phi\left(\vec{b}\right) = \phi\left(\begin{bmatrix} b_1 \\ b_2 \\ \cdots \\ b_n \end{bmatrix}\right) = \sqrt{b_1^2 + b_2^2 + \cdots b_n^2}$$

Now

$$\phi(\vec{a}) + \phi\left(\vec{b}\right) = \sqrt{a_1^2 + a_2^2 + \cdots a_n^2} + \sqrt{b_1^2 + b_2^2 + \cdots b_n^2}$$

and

$$\phi\left(\vec{a} + \vec{b}\right) = \phi\left(\begin{bmatrix} a_1 + b_1 \\ a_2 + b_2 \\ \vdots \\ a_n + b_n \end{bmatrix}\right) = \sqrt{(a_1 + b_1)^2 + (a_2 + b_2)^2 + \cdots (a_n + b_n)^2}$$

Clearly these two are not equal and hence we have violated equation 2.17 - ϕ is a non-linear mapping.

2.11 Multidimensional Arrays, Multi-linear Transforms and Tensors

One often hears the term *tensor* in connection to machine learning. Google's famous machine learning platform is called *Tensorflow*. In this section, we will introduce the reader to the concept of a tensor.

2.11.1 Array View: Multidimensional arrays of numbers

A tensor maybe viewed as a generalized *n*-dimensional array - although, strictly speaking, not all multi-dimensional arrays are tensors. We will learn more about the distinction between multi-dimensional arrays and tensors when we study multi-linear transforms. For now, we will not worry too much about the distinction. A vector can be viewed as a 1 tensor, a matrix is 2 tensor, a scalar is 0 tensor.

In section 2.3, we saw that digital images are represented as 2D arrays (matrices). A color image, where each pixel is represented by 3 colors, R, G, B (Red, Green, Blue), is an example of a multi-dimensional array or tensor. This is because it can be viewed as a combination of 3 images, the R, G and B images respectively.

The inputs and outputs to each layer in a neural network are also tensors.

2.12 Linear Systems and Matrix Inverse

Machine learning today is usually an iterative process. Given a set of training data, one wants to estimate a set of machine parameters that would yield target values (or close approximations to them) on training inputs. The number of training inputs and the size of the parameter set is often very large. This makes it impossible to have a closed-form solution - where one solves for the unknown parameters in a single step. Solutions are usually iterative. One starts with a guessed set of values for the parameters and iteratively improves the guess by processing training data. Having said that, one often encounters smaller problems in real life. One is better off using more traditional closed-form techniques here since they are much faster and more accurate. This section is devoted to studying these techniques.

Let us go back to our familiar cat brain problem and refer to its training data in Table 2.3. As before, we are still talking about a weighted sum model with 3 parameters - weights w_0, w_1 and bias *b*. Let us focus on the top three rows from the table, repeated here for convenience.

Table 2.4: Example Training Dataset for our Toy Machine Learning Based Cat Brain

	input value: hardness	input value: sharpness	output: threat score
0	0.11	0.09	-0.8
1	0.01	0.02	-0.97
2	0.98	0.91	0.89

The training data says, with hardness value 0.11 and sharpness value 0.09, we expect the output of the system to match (or closely approximate) the target value −0.8 etc. In other words, our estimated values for parameters w_0, w_1, b should ideally satisfy

$$0.11\ w_0 + 0.09\ w_1 + b = -0.8$$
$$0.01\ w_0 + 0.02\ w_1 + b = -0.97$$
$$0.98\ w_0 + 0.91\ w_1 + b = 0.89$$

We can express this via matrix multiplication as the following equation

$$\begin{bmatrix} 0.11 & 0.09 & 1 \\ 0.01 & 0.02 & 1 \\ 0.98 & 0.91 & 1 \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ b \end{bmatrix} = \begin{bmatrix} -0.08 \\ -0.97 \\ 0.89 \end{bmatrix}$$

There exists formal methods (discussed below) to directly solve such equations for w_0, w_1, b (in this very simple example one might just "see" that $w_0 = 1, w_1 = 1, b = -1$ solves the equation. In many situations, solving a simple linear system is all one needs to do. It is important to recognize simple problems and use simple tools that can solve them. This section is devoted to closed-form solutions for unknown parameters.

Let us say we want to solve a simultaneous equation with two variables, x_1 and x_2, we can easily write it as:

$$a_{11}x_1 + a_{12}x_2 = b_1$$
$$a_{21}x_1 + a_{22}x_2 = b_2$$

This can be written using matrices and vectors as $A\vec{x} = \vec{b}$ where

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$$
$$\vec{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \text{ and } \vec{b} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

The matrix formulation easily extends to multi-dimensions. An arbitrary linear system in n unknowns x_1, x_2, x_3, $\cdots$, x_n,

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1n}x_n = b_1$$
$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \cdots + a_{2n}x_n = b_2$$

.
.
.

$$a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + \cdots + a_{nn}x_n = b_n$$

can be expressed as

$$A\vec{x} = \vec{b}$$

Although equivalent, the matrix depiction is more compact and dimension independent. In machine learning, we usually have many variables (say thousands). Then, this compactness makes a serious difference. Also, $A\vec{x} = \vec{b}$ looks similar to the 1 variable equation we know so well: $ax = b$. In fact, many intuitions can be transferred from the 1D to higher dimensions.

What is the solution of the 1D equation? One learnt it in grade 5 perhaps: Solution of $ax = b$ is $x = a^{-1}b$ where $a^{-1} = 1/a$, $a \neq 0$.

We can use the exact same notation in all dimensions. Solution of $A\vec{x} = \vec{b}$ is $\vec{x} = A^{-1}\vec{b}$ where A^{-1} is the matrix inverse, (assumed $det(A) \neq 0$).

How does one compute the matrix inverse? There are completely precise but tedious rules for computing matrix inverses. Despite the importance of the concept, one rarely needs to actually compute a matrix inverse in life. We will not describe the general method here. Instead we will only breeze through inverse computations of 2×2 and 3×3 matrices.

DETERMINANT OF 2 × 2 AND 3 × 3 MATRICES

The determinant is always needed to compute the inverse of a matrix. For a 2×2 matrix

$$A_{2\times 2} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$$

the determinant is

$$det\,(A_{2\times 2}) = a_{11}a_{22} - a_{12}a_{21}$$

For instance, the determinant of the 2×2 matrix $\begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix}$ is $1 \times 1 - 2 \times 0 = 1$.

We will illustrate the 3×3 case with an example. Consider the matrix

$$A_{3\times 3} = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & 4 \\ 5 & 6 & 0 \end{bmatrix}$$

The steps are described below in Algorithm 2:

INVERSE OF 2 × 2 AND 3 × 3 MATRICES

For a 2×2 matrix

$$A_{2\times 2} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$$

the inverse is

$$A^{-1} = \frac{1}{det\,(A)} \begin{bmatrix} a_{22} & -a_{12} \\ -a_{21} & a_{11} \end{bmatrix}$$

For our example matrix $A_{3\times 3},\ A^{-1}_{3\times 3} = \frac{1}{1}\begin{bmatrix} -24 & 18 & 5 \\ 20 & -15 & -4 \\ -5 & 4 & 1 \end{bmatrix} = \begin{bmatrix} -24 & 18 & 5 \\ 20 & -15 & -4 \\ -5 & 4 & 1 \end{bmatrix}$ The algorithm is exemplified in Algorithm 3.

Algorithm 2 Computing determinant of the 3×3 matrix

Choose any one row or column as pivot. Let us say the top row.

Initialize: $S = 0$.

for each element of the pivot row or column **do**

- Block the row and column that intersects at that element. This uncovers a 2×2 sub-matrix.

  Compute its determinant. For instance, the uncovered sub-matrix for the top row middle column element, viz., 2 we get

  $$\begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & 4 \\ 5 & 6 & 0 \end{bmatrix}$$

  The uncovered sub-matrix is $\begin{bmatrix} 0 & 4 \\ 5 & 0 \end{bmatrix}$ and its determinant is $0 \times 0 - 5 \times 4 = -20$.
- Multiply the picked up element with the corresponding determinant, e.g., here -20×2.
- Add the row and column indices of the element (indices start from 1). If the sum is odd, multiply the above product with -1, else $+1$. For instance, the top left element has row index = 1 and column index = 1, so the sum is $1 + 1 = 2$, an even number. Hence the multiplier is +1. The top middle element has row index $= 1$ and column index = 2, their sum is 3, an odd number. Hence the multiplier is -1.
- Add the product of - the element, sub-matrix determinant and sign - to *S*. E.g., here $2 \times(-20) \times(-1) = 40$ gets added to S.

end for

.

return *S* as determinant. For our example matrix $A_{3\times 3}$, $det\ (A_{3\times 3}) = 1 \times(1 \times 0 - \times 4 \times 6) \times(+1) + 2 \times(0 \times 0 - \times 4 \times 5) \times(-1) + 3 \times(0 \times 6 - \times 1 \times 5) \times(+1) = -24 + 40 - \times 15 = 1$

Algorithm 3 Computing inverse of the 3 × 3 matrix

for each matrix element **do**

- Compute the determinant of the sub-matrix that is uncovered when one covers the row and column intersecting in that element - a la determinant computation.
- Multiply the the above with the sign corresponding to the element as in determinant computation.

end for

- Replace each matrix elements with the corresponding value from the above for loop. For our example matrix A, this yields $\begin{bmatrix} -24 & 20 & -5 \\ 18 & -15 & 4 \\ 5 & -4 & 1 \end{bmatrix}$
- Transpose the above matrix. This yields the adjoint matrix. For our example matrix $A_{3\times3}$, this yields $adj\,(A_{3\times3}) = \begin{bmatrix} -24 & 18 & 5 \\ 20 & -15 & -4 \\ -5 & 4 & 1 \end{bmatrix}$

return $A^{-1}_{3\times3} = \frac{1}{det(A_{3\times3})} adj\,(A_{3\times3})$ as inverse.

One can easily verify the correctness of the inverse computation by multiplying the matrix and its computed inverse:

$$A_{3\times3}A^{-1}_{3\times3} == \begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & 4 \\ 5 & 6 & 0 \end{bmatrix} \begin{bmatrix} -24 & 18 & 5 \\ 20 & -15 & -4 \\ -5 & 4 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Multiplying the matrix and its inverse yields a special matrix called the identity matrix. These are discussed in the next section in more detail.

IDENTITY MATRICES

For any matrix A, whose determinant $det(A)$ is not zero, we can compute another matrix A^{-1}, following the steps outlined in algorithm 3 such that

$$AA^{-1} = A^{-1}A = \mathbf{I}$$

For instance $A_{2\times2}A_{2\times2} = \mathbf{I}_{2\times2}$ where $\mathbf{I}_{2,2} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$.

And $A_{3\times3}A^{-1}{}_{3\times3} = \mathbf{I}_{3\times3}$ where $\mathbf{I}_{3,3} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$

In general, for any dimension n

$$\mathbf{I}_{n,n} = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & & & & \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix}$$

(the diagonal terms are 1, all other terms are 0). When there is no subscript, the dimensionality is to be inferred from context.

The identity matrix is the analog of 1 in higher dimensions. To get an intuition take note of the following facts

- The matrix inverse is defined such that $AA^{-1} = A^{-1}A = \mathbf{I}$ where A and $\mathbf{I}$ are square $n \times n$ matrices. Compare this to the scalar equation $a \times a^{-1} = a^{-1} \times a = 1$.
- For any matrix A, $\mathbf{I}A = A\mathbf{I} = A$. The reader is encouraged to verify this using standard matrix multiplication rules.
- For any vector $\vec{a}$, $\mathbf{I}\vec{a} = \vec{a}^T\mathbf{I} = \vec{a}$. The reader is encouraged to verify this using standard matrix multiplication rules.

Computing matrix inverse is not good programming practice however, because it is numerically unstable. Instead, one often uses Gaussian elimination or diagonalization (section 2.15.2) or singular value decomposition (section 4.4). Since Gaussian Elimination has little conceptual bearing upon machine learning we will not discuss it here.

2.12.1 Linear Systems with zero or near zero Determinants; Ill Conditioned Systems

We saw above that a linear system $A\vec{x} = \vec{b}$ has solution $\vec{x} = A^{-1}\vec{b}$ Now, for all dimensions, A^{-1} will have $\frac{1}{det(A)}$ as a factor. What if the determinant is zero?

The short answer: when the determinant is zero, the linear system cannot be exactly solved. We may still attempt to come up with an approximate answer (see subsection 2.12.2), but exact solution is not possible.

Let us examine the situation a bit more closely, with the aid of an example. Consider the following system of equations:

$$\begin{aligned} x_1 + x_2 &= 2 \\ 2x_1 + 2x_2 &= 4 \end{aligned}$$

It can be rewritten as a linear system with a square matrix, so:

$$\begin{bmatrix} 1 & 1 \\ 2 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \end{bmatrix}$$

But, one can quickly see that the system of equations cannot be solved. The second equation is really the same as the first. In fact, we can obtain the second by multiplying the first by a scalar,

2. Hence, we do not really have 2 equations, we have only 1 and hence the system cannot be solved. Now examine the row vectors of matrix A. They are $[1\ 1]$ and $[2\ 2]$. They are linearly dependent because $-2\ [1\ 1] + [2\ 2] = 0$. Now examine the determinant of matrix A. It is $2 \times 1 - 1 \times 2 = 0$.

The above is not a coincidence. Any one of them implies the other. In fact, the following statements about the linear system $A\vec{x} = \vec{b}$ (with a square matrix) are equivalent:

- Matrix A has a row/column that can be expressed as a weighted sum of others
- Matrix A has linearly dependent rows or columns.
- Matrix A has zero determinant (such matrices are called *singular* matrices)
- Inverse of matrix A, i.e., A^{-1}, does not exist. A is called *singular.*
- The linear system cannot be solved

The system is trying to tell you that you have fewer equations than you think you have and you cannot really solve the system of equations.

Sometimes, the determinant is not exactly zero, but close to zero. Such systems, although solvable is theory, are *numerically unstable*. Small changes in input causes the result to change drastically. For instance, consider the nearly singular matrix

$$A = \begin{bmatrix} 2 & 1 \\ 4 & 2.001 \end{bmatrix} \tag{2.18}$$

Its determinant is 0.002, close to zero. Let $\vec{b} = \begin{bmatrix} 3 \\ 6 \end{bmatrix}$ be a vector.

$$A^{-1} = \begin{bmatrix} 1000.5 & -500.0 \\ -2000. & 1000.0 \end{bmatrix} \tag{2.19}$$

(note how large the elements of A^{-1} are, this is due to division by an extremely small determinant and this in turn causes the instability illustrated below). The solution to the equation $A\vec{x} = \vec{b}$ is $\vec{x} = A^{-1}\vec{b} = \begin{bmatrix} 1.5 \\ 0 \end{bmatrix}$. But if we change $\vec{b}$ just a little and make $\vec{b} = \begin{bmatrix} 3 \\ 6.01 \end{bmatrix}$ the solution changes to a drastically different $\vec{x} = A^{-1}\vec{b} = \begin{bmatrix} -3.5 \\ 10.0 \end{bmatrix}$

This is inherently unstable and arising from the near singularity of the matrix *A*. Such linear systems are called ill-conditioned.

2.12.2 Over and Under Determined Linear Systems in Machine Learning and Data Science

What if the matrix A is *not* square? This implies the number of equations do not match number of unknowns. Does such a system even make sense? Surprisingly, it does. There are two possible cases, assuming that the matrix A is $m \times n$ (m rows and n columns).

- Case 1: $m > n$ (More equations than unknowns - Overdetermined System)

- Case 2: $m < n$ (Fewer equations than unknowns - Underdetermined System)

For instance, Table 2.3 leads to an over-determined linear system. Let us write down the system of equations.

$$
\begin{aligned}
0.11w_0 + 0.09w_1 + b &= -0.8 \\
0.01w_0 + 0.02w_1 + b &= -0.97 \\
0.98w_0 + 0.91w_1 + b &= 0.89 \\
0.12w_0 + 0.21w_1 + b &= -0.68 \\
0.98w_0 + 0.99w_1 + b &= 0.95 \\
0.85w_0 + 0.87w_1 + b &= 0.74 \\
0.03w_0 + 0.14w_1 + b &= -0.88 \\
0.55w_0 + 0.45w_1 + b &= 0.00 \\
0.49w_0 + 0.51w_1 + b &= 0.01 \\
0.99w_0 + 0.01w_1 + b &= 0.009 \\
0.02w_0 + 0.89w_1 + b &= -0.07 \\
0.31w_0 + 0.47w_1 + b &= -0.23 \\
0.55w_0 + 0.29w_1 + b &= -0.14 \\
0.87w_0 + 0.76w_1 + b &= 0.65 \\
0.63w_0 + 0.74w_1 + b &= 0.36
\end{aligned}
$$

yielding the overdetermined linear system

$$
\begin{bmatrix}
0.11 & 0.09 & 1 \\
0.01 & 0.02 & 1 \\
0.98 & 0.91 & 1 \\
0.12 & 0.21 & 1 \\
0.98 & 0.99 & 1 \\
0.85 & 0.87 & 1 \\
0.03 & 0.14 & 1 \\
0.55 & 0.45 & 1 \\
0.49 & 0.51 & 1 \\
0.99 & 0.01 & 1 \\
0.02 & 0.89 & 1 \\
0.31 & 0.47 & 1 \\
0.55 & 0.29 & 1 \\
0.87 & 0.76 & 1 \\
0.63 & 0.74 & 1
\end{bmatrix}
\begin{bmatrix} w_0 \\ w_1 \\ b \end{bmatrix}
=
\begin{bmatrix}
-0.8 \\ -0.97 \\ 0.89 \\ -0.68 \\ 0.95 \\ 0.74 \\ -0.88 \\ 0.00 \\ 0.01 \\ 0.009 \\ -0.07 \\ -0.23 \\ -0.14 \\ 0.65 \\ 0.36
\end{bmatrix}
\tag{2.20}
$$

This is a non-square 15 × 3 linear system. There are only 3 unknowns to solve for, w_0,w_1, b and there are 15 equations. Highly redundant, we needed only 3 equations and could have solved it via Gaussian Elimination (section 2.12. But the important thing to note is this: *the equations are not fully consistent*. There is no single set of values for the unknown that will satisfy all of them. In other words, the training data is not fully consistent - an almost

universal occurrence in all real life machine learning systems. Consequently, we have to find a solution that is optimal (causes as little error as possible) over all the equations.

We want to solve it such that the overall error, viz., $\|A\vec{x} = \vec{b}\|$ is minimized. In other words, we are looking for $\vec{x}$such that $A\vec{x}$ is as close to $\vec{b}$ as possible. This closed-form (i.e., non-iterative) method is an extremely important pre-cursor to machine learning and data science. We will revisit this multiple times, most notably in sections 2.12.3, 4.4.

2.12.3 Moore Penrose Pseudo-Inverse of a Matrix: solving Over or Under Determined Linear Systems

Suppose we have the overdetermined system with not-necessarily-square, $m \times n$ matrix A

$$A\vec{x} = \vec{b}$$

Since A is not guaranteed to be square, we can neither take determinant, nor inverse in general. So the usual $A^{-1}\ \vec{b}$ does not work. At this point, we observe that although inverse cannot be taken, transposing the matrix is always possible. Let us multiply both sides of the equation with A^T:

$$A\vec{x} = \vec{b} \Leftrightarrow A^T A\vec{x} = A^T\vec{b}$$

Notice that A^TA is a square matrix - its dimensions are $(m \times n) \times (n \times m) = m \times m$. Let us assume, without proof for the moment, that it is invertible. Then

$$A\vec{x} = \vec{b} \Leftrightarrow A^T A\vec{x} = A^T\vec{b} \Leftrightarrow \vec{x} = (A^TA)^{-1}\ A^T\vec{b}$$

Hmmm, not bad, we seem to be on to something. In fact, we just derived the *pseudo-inverse of matrix A, denoted* $A^+ = (A^TA)^{-1}\ A^T$. *Unlike the inverse, the pseudo-inverse does not need the matrix to be square with linearly independent rows.* Much like the regular linear system, we get the solution of the (possibly non-square) system of equations as $A\vec{x} = \vec{b} \Leftrightarrow \vec{x} = A^+\vec{b}$.

The pseudo-inverse based solution actually minimizes the error $\|A\vec{x} = \vec{b}\|$. We will provide an intuitive proof of that in the next section 2.12.4. Meanwhile, the reader is encouraged to write the python code to evaluate $(A^TA)^{-1}\ A^T\ \vec{b}$ and verify that it approximately yields the expected answer $\begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix}$ for equation 2.20.

2.12.4 Pseudo Inverse of a Matrix: A Beautiful Geometric Intuition

A matrix $A_{m\times n}$ can be rewritten in terms of its column vectors as $[\vec{a}_1, \vec{a}_2, \ldots \vec{a}_n]$, where $\vec{a}_1 \ldots \vec{a}_n$ are all m dimensional vectors. Then $\vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ \cdots \\ x_n \end{bmatrix}$ we get $A\vec{x} = x_1\vec{a}_1 + x_2\vec{a}_2 + \ldots x_n\vec{a}_n$ In other words,

$A\vec{x}$ is just a linear combination of the column vectors of A with the elements of $\vec{x}$ as the weights (the reader is encouraged to write out a small 3 × 3 system and verify this). The space of all vectors of the form $A\vec{x}$, i.e., the linear span of the column vectors of *A*, is known as the column space of *A*.

The solution to the linear system of equations, $A\vec{x} = \vec{b}$, can be viewed as the $\vec{x}$ that minimizes the difference of $A\vec{x}$ and $\vec{b}$, i.e., minimizes $\|A\vec{x} = \vec{b}\|$. This means, we are trying to find a vector in the column space of A that is closest to the point $\vec{b}$. Note that this interpretation does not assume a square matrix *A*. Nor does it assume non-zero determinant. In the friendly case where the matrix A is square and invertible, we will be able to find a vector $\vec{x}$ such that $A\vec{x}$ becomes exactly equal to $\vec{b}$ which makes $\|A\vec{x} = \vec{b}\| = 0$. If A is not square, we will try to find $\vec{x}$ such that $A\vec{x}$ is closer to $\vec{b}$ than any other vector in the column space of *A*. Mathematically speaking[9],

$$\|A\vec{x} - \vec{b}\| \leq \|A\vec{y} - \vec{b}\| \quad \forall \vec{y} \in \mathfrak{R}^n \tag{2.21}$$

From geometry, we intuitively know that the closest point to $\vec{b}$ in the column space of A is obtained

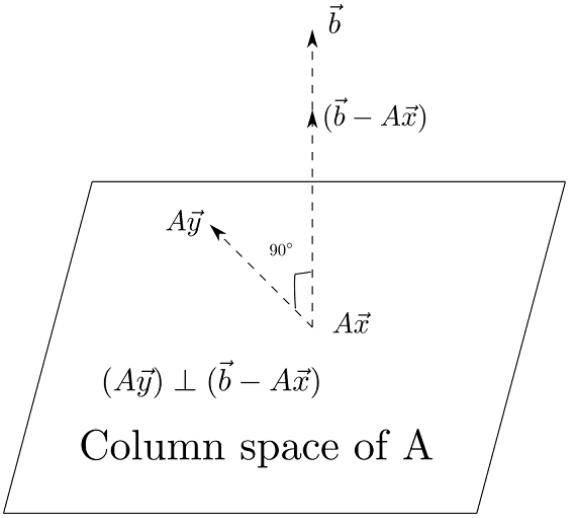


Figure 2.12: Solving a Linear System $A\vec{x} = \vec{b}$ is equivalent to finding the point on the column space of A that is closest to $\vec{b}$. The difference vector $\vec{b}-A\vec{x}$ will be perpendicular to all vectors $A\vec{y}$ in the column space of A.

[9] The mathematical symbol ∀ stands for "for all". Thus, $\forall \vec{y} \in \mathfrak{R}^n$ means "all vectors y in the n-dimensional space"

by dropping a perpendicular from $\vec{b}$ to the column space of A (see figure 2.12). The point where this perpendicular meets the column space is called the *projection* of $\vec{b}$ on column space of A. The solution vector $\vec{x}$ to equation 2.21 that we are looking for, should correspond to the projection of $\vec{b}$ on the column space of A. This in turn means $\vec{b}-A\vec{x}$ is orthogonal (perpendicular) to all vectors in column space of A (see Figure 2.12). We represent arbitrary vectors in the column space of A as $A\vec{y}$ for arbitrary $\vec{y}$. Hence,

$$\begin{aligned}\forall \vec{y} \in \mathfrak{R}^n \; (A\vec{y}) \perp (\vec{b}-A\vec{x}) &\Leftrightarrow (A\vec{y})^T \; (\vec{b}-A\vec{x}) = 0 \\ &\Leftrightarrow \vec{y}^T A^T \; (\vec{b}-A\vec{x}) = 0\end{aligned}$$

For the above equation to be true for all vectors $\vec{y}$, we must have $A^T(\vec{b}-A\vec{x}) = 0$.

$$\begin{aligned}& A^T \; (\vec{b}-A\vec{x}) = 0 \\ \Leftrightarrow\; & A^T A\vec{x} = A^T\vec{b} \\ \Leftrightarrow\; & \vec{x} = (A^T A)^{-1} A^T\vec{b}\end{aligned}$$

which is exactly the Moore-Penrose pseudo-inverse.

2.12.5 Python numpy code to solve over-determined systems

Fully functional code for this section, executable via Jupyter-notebook, can be found at https://nbviewer.jupyter.org/github/krishnonwork/mathematical-methods-in-deep-learning-ipython/blob/master/python/ch2/2.12.5-overdet-numpy.ipynb.

Consider the overdetermined system corresponding to cat-brain from Chapter 2. There are 15 training examples, each with input and desired outputs specified.

Our goal is to determine 3 unknowns w_0,w_1, b, such that for each training input $\vec{x}_i = \begin{bmatrix} x_{i,0} \\ x_{i,1} \end{bmatrix}$ and corresponding ground truth (known desired output) y_i, the model output

$$x_{i,0}w_0 + x_{i,1}w_1 + b = \begin{bmatrix} x_{i,0} & x_{i,1} \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \end{bmatrix} + b = \begin{bmatrix} x_{i,0} & x_{i,1} & 1 \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ b \end{bmatrix} \tag{2.22}$$

is as close to desired output y_i as possible.

Note the neat trick above - we added a 1 to the right of the input - this allows us to depict the whole system (including the bias) in a single compact matrix vector multiplication. We call this "augmentation" - we have augmented the input row vector with an extra 1 on the right. Collating all the training examples together, we get

$$\begin{bmatrix} x_{0,0} & x_{0,1} & 1 \\ x_{1,0} & x_{1,1} & 1 \\ \vdots & \vdots & \vdots \\ x_{N,0} & x_{N,1} & 1 \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ b \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \cdots \\ y_N \end{bmatrix} \tag{2.23}$$

Thus the overall system of linear equations is

$$X\vec{w} = \vec{y}$$

where X is the augmented input matrix - with a rightmost column of all 1s.

Note that *this is not a classic system of equations - it has more equations than unknowns*. We cannot solve this via matrix inversion. We *can* however, use the pseudo-inverse mechanism to solve this. The resulting solution yields the "best fit" or "best effort" solution - which minimizes the total error over all the training examples.

For our cat-brain problem, the exact system (repeated here for ease of reference) is

$$X = \begin{bmatrix} 0.11 & 0.09 & 1.00 \\ 0.01 & 0.02 & 1.00 \\ 0.98 & 0.91 & 1.00 \\ 0.12 & 0.21 & 1.00 \\ 0.98 & 0.99 & 1.00 \\ 0.85 & 0.87 & 1.00 \\ 0.03 & 0.14 & 1.00 \\ 0.55 & 0.45 & 1.00 \\ 0.49 & 0.51 & 1.00 \\ 0.99 & 0.01 & 1.00 \\ 0.02 & 0.89 & 1.00 \\ 0.31 & 0.47 & 1.00 \\ 0.55 & 0.29 & 1.00 \\ 0.87 & 0.76 & 1.00 \\ 0.63 & 0.24 & 1.00 \end{bmatrix} \quad \vec{y} = \begin{bmatrix} -0.8 \\ -0.97 \\ 0.89 \\ -0.67 \\ 0.97 \\ 0.72 \\ -0.83 \\ 0.00 \\ 0.00 \\ 0.00 \\ -0.09 \\ -0.22 \\ -0.16 \\ 0.63 \\ 0.37 \end{bmatrix} \quad \vec{w} = \begin{bmatrix} w_0 \\ w_1 \\ b \end{bmatrix} \tag{2.24}$$

We solve for $\vec{w}$ using the pseudo inverse formula $\vec{w} = (X^T X)^{-1} X^T y$

Listing 2.14: Solving an overdetermined system using pseudo-inverse

```
def pseudo_inverse(X):
    return np.matmul(np.linalg.inv(np.matmul(X.T, X)), X.T)

X = np.column_stack((X, np.ones(15)))

w = np.matmul(pseudo_inverse(X), y)

print("The solution is {}".format(w))
```

Numpy column stack operator adds a column to a matrix. Here the added column is all 1s.

X is the augmented data matrix from equation 2.24

Solution via pseudo-inverse. Expect the solution to be close to [1, 1, -1]

It is easy to verify that the solution to equation 2.24 is roughly $w_0 = 1, w_1 = 1, b = -1$.

But the equations are not consistent - no one solution perfectly fits all of them. Pseudo-inverse finds the "best fit" solution - minimizes total error for all the equations.

Output:

```
The solution is [ 1.07661761 0.89761672 -0.95816936]
Note that this is almost equal to [1.0 , 1.0 , -1.0])
```

2.13 Eigenvalues and Eigenvectors - swiss army knives in Machine Learning and Data Science

Machine Learning and Data Science is all about finding patterns in large volumes of high dimensional data. The inputs are typically feature vectors (introduced in section 2.1) in high dimensional spaces. Each individual dimension of the feature vector corresponds to a particular property of the input. The feature vector, thus, is a descriptor for the particular input instance. It can be viewed as a point in the feature space. We usually transform the points to a friendlier space where it is easier to perform the analysis we are trying to do (a simple example of such a transform was shown in section 1.3). For instance, if we are building a classifier, we try to transform the input to a space where the points belonging to different classes are more segregated. Sometimes we transform to simplify the data, eliminating axes along which there is scant variation in the data. Eigenvalues and eigenvectors are invaluable items in the tool set of a machine learning engineer or a data scientist - helping them understand how to transform and analyze large volumes of high dimensional data. In chapter 4 we will study how to use these tools to simplify and find broad patterns in large volume of multi-dimensional data.

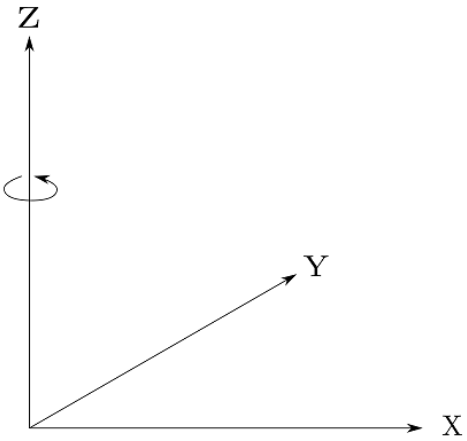


Figure 2.13: During Rotation, points on the axis of rotation do not change position

Transforms generally map vectors (points) in one space to different vectors (points) in the same or a different space. But a typical linear transform will leave a few points in the space (almost) unaffected. These points are called eigenvectors. They provide important insights into the transform. Let us look at a simple example. Suppose we are *rotating* points in 3D space about the Z axis (see Fig. 2.13). The points on the Z axis will stay where they were despite the rotation. In general, points on the axis of rotation (Z axis in this case) do not go anywhere after rotation. Thus, axis of rotation is the eigenvector of rotation transformation.

Extending this same idea, when *transforming* vectors $\vec{x}$ with a matrix A, are there vectors that do not change, at least, say, in direction? Turns out, the answer is yes. These are the so called *eigenvectors* - they do not change direction when undergoing linear transformation by a matrix *A*. To be precise, if $\vec{e}$ is an eigenvector of square matrix A[10], then

$$A\vec{e} = \lambda\vec{e}$$

Thus the linear transformation (i.e., multiplication by matrix *A*) has changed the length, but not the direction of $\vec{e}$ - because $\lambda\vec{e}$ is parallel to $\vec{e}$.

How to obtain λ and $\vec{e}$? Well

$$\begin{aligned} & A\vec{e} = \lambda\vec{e} \\ \Leftrightarrow\ & A\vec{e} - \lambda\vec{e} = \vec{0} \\ \Leftrightarrow\ & (A - \lambda\mathbf{I})\ \vec{e} = \vec{0} \end{aligned}$$

[10] one can compute eigenvectors and eigenvalues only of square matrices

where **I** denotes the Identity Matrix.

Of course, we are only interested in non-trivial solutions, where $\vec{e} \neq \vec{0}$. In that case, $A - \lambda\mathbf{I}$ cannot be invertible, because if it were, one could obtain the contradictory solution $\vec{e} = (A - \lambda\mathbf{I})^{-1}\vec{O} = \vec{O}$. Thus, $(A - \lambda\mathbf{I})$ is non-invertible, implying the determinant

$$det\,(A - \lambda\mathbf{I}) = 0$$

For an $n \times n$ matrix A, this yields an n^{th} degree polynomial equation with n solutions for the unknown λ. *Thus a $n \times n$ matrix has n eigenvalues, not necessarily all distinct*.

Lets compute eigenvalues and eigenvectors of a 3 × 3 matrix just for kicks. The matrix we choose is not random, as will be evident soon. But for now, think of it as an arbitrary matrix.

$$A = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.25}$$

We will compute eigenvalues and eigenvectors of *A*.

$$\begin{aligned}(A - \lambda\mathbf{I}) &= \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ 0 & 0 & 1 \end{bmatrix} - \begin{bmatrix} \lambda & 0 & 0 \\ 0 & \lambda & 0 \\ 0 & 0 & \lambda \end{bmatrix} \\ &= \begin{bmatrix} \frac{1}{\sqrt{2}} - \lambda & \frac{1}{\sqrt{2}} & 0 \\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} - \lambda & 0 \\ 0 & 0 & 1 - \lambda \end{bmatrix}\end{aligned}$$

Thus,

$$\begin{aligned} & det\,(A - \lambda\mathbf{I}) && = 0 \\ \Leftrightarrow & (1-\lambda)\left(\left(\frac{1}{\sqrt{2}} - \lambda\right)\left(\frac{1}{\sqrt{2}} - \lambda\right) + \frac{1}{\sqrt{2}}\frac{1}{\sqrt{2}}\right) && = 0 \\ \Leftrightarrow & (1-\lambda)\left(\lambda^2 - 2\frac{1}{\sqrt{2}}\lambda + \frac{1}{2} + \frac{1}{2}\right) && = 0 \\ \Leftrightarrow & (1-\lambda)\left(\lambda^2 - \sqrt{2}\lambda + 1\right) && = 0 \\ \Leftrightarrow & \lambda = 1 \text{ or } \lambda = \left(\frac{1}{\sqrt{2}} + i\frac{1}{\sqrt{2}}\right) \text{ or } \lambda = \left(\frac{1}{\sqrt{2}} - i\frac{1}{\sqrt{2}}\right) \\ \Leftrightarrow & \lambda = 1 \text{ or } \lambda = e^{i\frac{\pi}{4}} \text{ or } \lambda = e^{-i\frac{\pi}{4}} \text{ using De Moivre's rule} \end{aligned}$$

Here $i = \sqrt{-1}$. If necessary, the reader is encouraged to refresh imaginary and complex numbers from high school algebra.

Thus, we have found (as expected) 3 eigenvalues, 1, $e^{i\frac{\pi}{4}}$ and $e^{-i\frac{\pi}{4}}$. Each of them will yield one eigenvector. Lets compute the eigenvector corresponding to eigenvalue of 1 by way of example.

$$A\vec{e_1} = 1 \cdot \vec{e_1}$$

$$\Leftrightarrow \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ 0 & 0 & 1 \end{bmatrix} \vec{e_1} = \vec{e_1}$$

$$\Leftrightarrow \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} e_{11} \\ e_{12} \end{bmatrix} = \begin{bmatrix} e_{11} \\ e_{12} \end{bmatrix}$$

$$\Leftrightarrow e_{11} = e_{12} = 0 \Leftrightarrow \vec{e_1} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Thus, $\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$ is an eigenvector for the eigenvalue 1 for matrix A. So is $\begin{bmatrix} 0 \\ 0 \\ k \end{bmatrix}$ for any real *k*. In fact, if λ, $\vec{e}$ is an eigen value, eigenvector pair for matrix A, then $A\vec{e} = \lambda\vec{e} \Leftrightarrow A(k\vec{e}) = \lambda(k\vec{e})$, i.e. λ, $(k\vec{e})$ is also an eigenvalue, eigenvector pair of A. In other words, one can only determine the eigenvector upto a fixed scale factor. We take the eigenvector to be of unit length ($\vec{e}^T\vec{e} = 1$) without loss of generality.

The eigenvector for our example matrix turns out to be the Z axis. This is not an accident. Our matrix A was, in fact, a rotation about the Z axis. *A rotation matrix will always have* 1 *as an eigenvalue. The corresponding eigenvector will be the axis of rotation. In 3D, the other two eigenvalues will be complex numbers yielding the angle of rotation* This is detailed in section 2.14.

NON-ZERO EIGENVECTORS CORRESPONDING TO DIFFERENT EIGENVALUES ARE LINEARLY INDEPENDENT

Let us prove this to get some insights. Let λ_1, $\vec{e}_1$ and λ_2, $\vec{e}_2$ be eigenvalue, eigenvector pairs for a matrix A with $\lambda_1 \neq \lambda_2$. Then

$$A\vec{e}_1 = \lambda_1\vec{e}_1$$
$$A\vec{e}_2 = \lambda_2\vec{e}_2$$

If possible, let there be two constants α_1 and α_2 such that

$$\alpha_1 \vec{e}_1 = \alpha_2 \vec{e}_2 = 0 \tag{2.26}$$

In other words, suppose the two eigenvectors be linearly dependent. We will show that this assumption leads to an impossibility.

Multiplying equation 2.26 by A, we get

$$\begin{aligned} & \alpha_1 A\vec{e}_1 + \alpha_2 A\vec{e}_2 && = 0 \\ \Leftrightarrow & \alpha_1 \lambda_1 \vec{e}_1 + \alpha_2 \lambda_2 \vec{e}_2 && = 0 \end{aligned}$$

Also, we can multiply equation 2.26 by λ_2. Thus we get

$$\begin{aligned} & \alpha_1 \lambda_1 \vec{e}_1 + \alpha_2 \lambda_2 \vec{e}_2 && = 0 \\ & \alpha_1 \lambda_2 \vec{e}_1 + \alpha_2 \lambda_2 \vec{e}_2 && = 0 \end{aligned}$$

Subtracting, we get

$$\alpha_1 (\lambda_1 - \lambda_2)\vec{e}_1 \qquad = 0$$

By assumption, $\alpha \neq 0$, $\lambda_1 \neq \lambda_2$ and $\vec{e}_1$ is not all zeros. Thus it is impossible for their product to be zero.

FOR SYMMETRIC MATRICES EIGENVECTORS CORRESPONDING TO UNEQUAL EIGENVALUES ARE ORTHOGONAL TO EACH OTHER

Let us prove this to get some more insights. A matrix A is symmetric iff $A^T = A$. If λ_1, $\vec{e}_1$ and λ_2, $\vec{e}_2$ are eigenvalue, eigenvector pairs for a symmetric matrix A, then

$$A\vec{e}_1 = \lambda_1 \vec{e}_1 \tag{2.27}$$

$$A\vec{e}_2 = \lambda_2 \vec{e}_2 \tag{2.28}$$

Transposing equation 2.27

$$\vec{e}_1{}^T A^T = \lambda_1 \vec{e}_1{}^T$$

$$\vec{e}_1{}^T A^T = \lambda_1 \vec{e}_1{}^T$$

Right-multiplying by $\vec{e}_2$, we get

$$\begin{aligned} & \vec{e}_1{}^T A^T \vec{e}_2 = \lambda_1 \vec{e}_1{}^T \vec{e}_2 \\ \Leftrightarrow & \vec{e}_1{}^T A\vec{e}_2 = \lambda_1 \vec{e}_1{}^T \vec{e}_2 \end{aligned}$$

where the last equation follows from the matrix symmetry. Also, left-multiplying equation 2.28 by $\vec{e}_1{}^T$ we get

$$\vec{e}_1{}^T A\vec{e}_2 = \lambda_2 \vec{e}_1{}^T\vec{e}_2$$

Thus

$$\vec{e}_1{}^T A\vec{e}_2 = \lambda_1 \vec{e}_1{}^T\vec{e}_2$$
$$\vec{e}_1{}^T A\vec{e}_2 = \lambda_2 \vec{e}_1{}^T\vec{e}_2$$

Subtracting the equations, we get

$$0 = (\lambda_1 - \lambda_2)\ \vec{e}_1{}^T\vec{e}_2$$

Since $\lambda_1 \neq \lambda_2$, we must have $\vec{e}_1{}^T\vec{e}_2 = 0$, which means the two eigenvectors are orthogonal. Thus, if A is a $n \times n$ symmetric matrix, with eigenvectors $\vec{e}_1, \vec{e}_2, \cdots \vec{e}_n$ then $\vec{e}_1{}^T\vec{e}_j = 0$ for all i, j satisfying $\lambda_i \neq \lambda_j$.

2.13.1 Python numpy code to compute eigenvectors and eigenvalues

Fully functional Jupyter-notebook code for this section can be found at https://nbviewer.jupyter.org/github/krishnonwork/mathematical-methods-in-deep-learning-ipython/blob/master/python/ch2/2.13-eig-numpy.ipynb.

Listing 2.15: Eigen values and vectors:

```
1 from numpy import linalg as LA
2
3
4
5
6 A = np.array([[0.707, 0.707, 0], [-0.707, 0.707, 0], [0, 0, 1]])
7
8 l, e = LA.eig(A)
9
10 print("Eigen values are {}".format(l))
11 print("Eigen vectors are {}".format(e.T))
```

Axis of rotation is a line whose points do not change position after rotation
i.e., $R\vec{x} = \vec{x} \implies$ axis is eigen vector, corresponding to eigen value 1.

$$A = \begin{bmatrix} cos\,(45^\circ) & sin\,(45^\circ) & 0 \\ -sin\,(45^\circ) & cos\,(45^\circ) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

rotates points in 3D space around the Z-axis.
The axis of rotation is Z-axis, i.e., $[0 \quad 0 \quad 1]^T$

Function eig() in Numpy Linear Algebra package Linalg computes eigen values/vectors.

Eigen values/vectors can contain complex numbers involving $j = \sqrt{-1}$

Output:

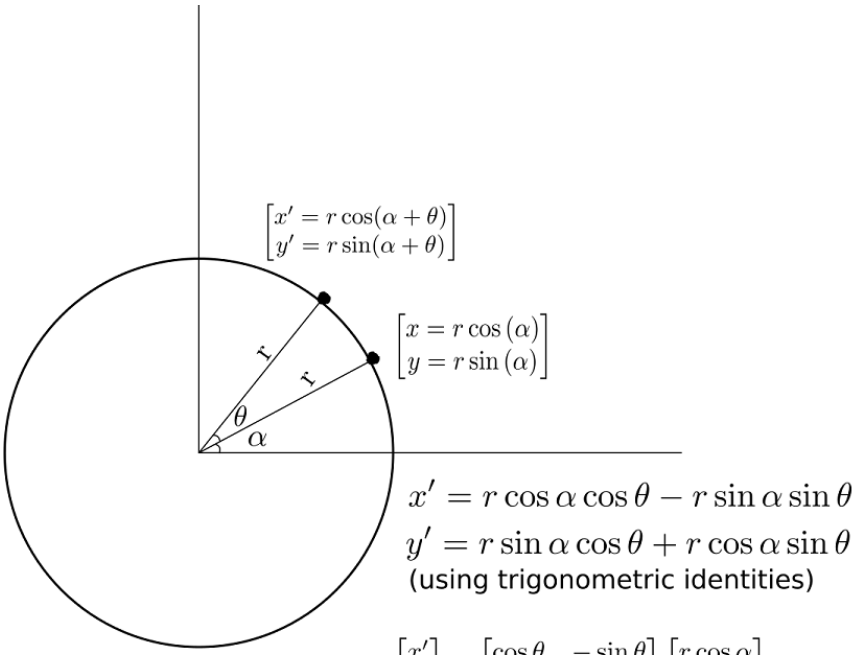
```
1 Eigen values are: [0.707+0.707 j 0.707 -0.707 j 1.0]
2
3 Eigen vectors are : [[0.707 0.707 j 0]
4 [0.707 -0.707 j 0]
5 [0 0 1]]
```

2.14 Orthogonal (Rotation) Matrices and their Eigenvalues and Eigenvectors

Of all transforms, rotation transforms have a special intuitive appeal because of their highly observable behavior in the mechanical world. Furthermore, they play a significant role in development and analysis of several machine learning tools. In this section we will do an overview of rotation (aka orthogonal) matrices.

Fully functional code for Jupyter-notebook for this section can be found at https://nbviewer.jupyter.org/github/krishnonwork/mathematical-methods-in-deep-learning-ipython/blob/master/python/ch2/2.14-rotation-numpy.ipynb.

ROTATION MATRICES



$$x' = r\cos\alpha\cos\theta - r\sin\alpha\sin\theta$$
$$y' = r\sin\alpha\cos\theta + r\cos\alpha\sin\theta$$

(using trigonometric identities)

$$\text{or} \quad \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} r\cos\alpha \\ r\sin\alpha \end{bmatrix}$$

$$= \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

$$\text{or} \quad \begin{bmatrix} x' \\ y' \end{bmatrix} = R \begin{bmatrix} x \\ y \end{bmatrix}$$

$$\text{or} \quad R^T R = RR^T = I \qquad \text{(easily verifiable)}$$

Figure 2.14: Rotation in plane about Origin. By definition rotation does not change the distance from the center

of rotation, that is what the circle indicates.

Figure 2.14 shows a point (x, y) rotated about the origin by an angle q. The original point's position vector made an angle a with the X axis. Post rotation, the point's new coordinates are (x', y'). Note that by definition rotation does not change the distance from the center of rotation, that is what the circle indicates. Some well known rotation matrices:

- **Planar Rotation by angle θ about Origin** (see Figure 2.14):

$$R_{2d} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \tag{2.29}$$

- **Rotation by angle θ in 3D space about Z axis:**

$$R_{3dz} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.30}$$

Note that the z coordinate remains unaffected by this rotation, viz.,

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} \cdot \\ \cdot \\ z \end{bmatrix}$$

- **Rotation by angle θ in 3D space about X axis:**

$$R_{3dx} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix} \tag{2.31}$$

Note that the X coordinate remains unaffected by this rotation, viz.,

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} x \\ \cdot \\ \cdot \end{bmatrix}$$

- **Rotation by angle θ in 3D space about Y axis:**

$$R_{3dy} = \begin{bmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{bmatrix} \tag{2.32}$$

Note that the γ coordinate remains unaffected by this rotation, viz.,

$$\begin{bmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} \cdot \\ y \\ \cdot \end{bmatrix}$$

Listing 2.16: Rotation matrices

```
Returns the matrix that performs in-plane 2D rotation by angle theta about origin.  Thus, multiplication with this matrix
moves a point to a new location. Angle between the position vectors of original and new point is theta (Fig. 2.14).

def rotation_matrix_2d(theta):
  return np.array([[cos(radians(theta)), -sin(radians(theta))],
                   [sin(radians(theta)), cos(radians(theta))]])
Returns the matrix that will rotate a point in 3D space about the chosen axis of rotation by angle theta degrees.
Axis of rotation, can be 0, 1 or 2 corresponding to x, y or z axis respectively.
def rotation_matrix_3d(theta, axis):
  if axis == 0:                                          R3dx from equation 2.31
    return np.array([[1, 0, 0],
                     [cos(radians(theta)),-sin(radians(theta)),0],
                     [sin(radians(theta)),cos(radians(theta)),0]])
  elif axis == 1:                                        R3dy from equation 2.32
    return np.array([[cos(radians(theta)),0,-sin(radians(theta))],
                     [0, 1, 0],
                     [sin(radians(theta)),0,cos(radians(theta))]])
  elif axis == 2:                                        R3dz from equation 2.30
    return np.array([[cos(radians(theta)),-sin(radians(theta)),0],
                     [sin(radians(theta)),cos(radians(theta)),0],
                     [0, 0, 1]])
```

Listing 2.17: Apply rotation matrices

```
u = np.array([1, 1, 1])                  create vector u⃗ (blue line in Fig 2.15)

R3dz = rotation_matrix_3d(45, 2)         R3dz from equation 2.30, 45° about Z-axis
v = np.matmul(R3dz, u_row)               v⃗ (green line in Fig 2.15) is u⃗ rotated by R3dz

R3dx = rotation_matrix_3d(45, 0)         R3dx from equation 2.30, 45° about X-axis
w = np.matmul(R3dx, u_row)               w⃗ (red line in Fig 2.15) is v⃗ rotated by R3dx
```

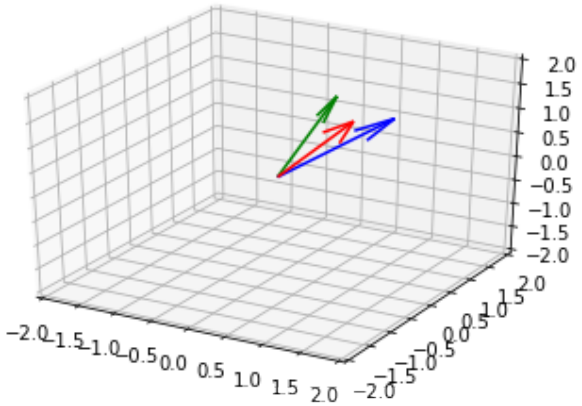


Figure 2.15: Rotation visualized: Here the original vector u(in blue) is first rotated by 45 degrees around the Z-Axis(in green) and then subsequently rotated again by 45 degrees around the X Axis

ORTHOGONALITY OF ROTATION MATRICES

A matrix R is orthogonal if and only if it its transpose is also its inverse, i.e., $R^TR = RR^T = \mathbf{I}$. All rotations matrices are orthogonal matrices. All orthogonal matrices represent some rotation. For instance:

$$\begin{bmatrix}\cos\theta & -\sin\theta \\ \sin\theta & \cos\theta\end{bmatrix}^T \begin{bmatrix}\cos\theta & -\sin\theta \\ \sin\theta & \cos\theta\end{bmatrix} = \begin{bmatrix}\cos\theta & \sin\theta \\ -\sin\theta & \cos\theta\end{bmatrix}^T \begin{bmatrix}\cos\theta & -\sin\theta \\ \sin\theta & \cos\theta\end{bmatrix}$$
$$= \begin{bmatrix}\cos^2\theta + \sin^2\theta & 0 \\ 0 & \cos^2\theta + \sin^2\theta\end{bmatrix} = \begin{bmatrix}1 & 0 \\ 0 & 1\end{bmatrix} = \mathbf{I}$$

The reader is encouraged to verify, likewise, that all the rotation matrices shown above are orthogonal.

ORTHOGONALITY IMPLIES ROTATION IS LENGTH PRESERVING

Given any vector $\vec{x}$ and rotation matrix R, let $\vec{x}' = R\vec{x}$ be the rotated vector. Lengths (magnitudes) of the 2 vectors $\vec{x}$, $\vec{x}'$ are equal,

(it is easy to see that $\|\vec{x}'\| = \vec{x}'^T\vec{x}' = (R\vec{x})^T(R\vec{x}) = \vec{x}^T R^T R\vec{x} = \vec{x}^T\mathbf{I}\vec{x} = \vec{x}^T\vec{x} = \|\vec{x}\|$)[11]

[11] from elementary matrix theory, we know that $(AB)^T = B^TA^T$

NEGATING THE ANGLE OF ROTATION IS EQUIVALENT TO INVERTING THE ROTATION MATRIX WHICH IS EQUIVALENT TO TRANSPOSING THE ROTATION MATRIX

For instance, consider in plane rotation. Say a point $\vec{x}$ is rotated about the origin to vector $\vec{x}$'via matrix $R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$. Thus, $\vec{x}$' = $R\vec{x}$. Now, we can go back from $\vec{x}$' to $\vec{x}$ by rotating by $-\theta$.

The corresponding rotation matrix is $\begin{bmatrix} \cos(-\theta) & -\sin(-\theta) \\ \sin(-\theta) & \cos(-\theta) \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} = R^T$. In other words, R^T inverts the rotation, i.e., rotates by the negative angle.

2.14.1 Python numpy code for orthogonality of rotation matrices

Listing 2.18: Orthogonality of rotation matrices

```
R_30 = rotation_matrix_2d(30)

assert np.allclose(np.linalg.inv(R_30), np.transpose(R_30))
assert np.allclose(np.matmul(R_30, R_30.T), np.eye(2))

u = np.array([[4],[0]])
v = np.matmul(R_30, u)

assert np.linalg.norm(u) == np.linalg.norm(v)

R_neg30 = rotation_matrix_2d(-30)
w = np.matmul(R_neg30, v)
assert np.all(w == u)

assert np.allclose(R_30, R_neg30.T)
assert np.allclose(np.matmul(R_30, R_neg30), np.eye(2))
```

a rotation matrix, R_{30}

Numpy function allclose() returns true if all elements are equal within specified tolerance

Inverse of a rotation matrix is same as its transpose

multiplication of a rotation matrix and its inverse yields the identity matrix

Numpy function eye() returns identity matrix

a vector $\vec{u}$ rotated by matrix R_{30} to vector $\vec{v}$, $R_{30}\vec{u} = \vec{v}$

Norm of a vector is same as its lengt Rotation preserves the length of a vector, $\|R\vec{u}\| = \|\vec{u}\|$

Rotation by an angle followed by rotation by negative angle takes the vector "back" to its original position

-equivalent to inverse rotation

Matrix that rotates by any angle is inverse of the matrix that rotates by negative of the same angle

EIGENVALUES AND EIGENVECTORS OF A ROTATION MATRIX: HOW TO FIND THE AXIS OF ROTATION

Let λ, $\vec{e}$ be an eigenvalue, eigenvector pair of a rotation matrix R. Then,

$$R\vec{e} = \lambda\vec{e}$$

Transposing both sides

$$\vec{e}^T R^T = \lambda \vec{e}^T$$

Multiplying left and right sides, respectively, with equivalent entities $R\vec{e}$ and $\lambda\vec{e}$, we get

$$\begin{aligned}
&\vec{e}^T R^T (R\vec{e}) = \lambda \vec{e}^T (\lambda \vec{e}) \\
\Leftrightarrow\ &\vec{e}^T (R^T R)\, \vec{e} = \lambda^2 \vec{e}^T \vec{e} \\
\Leftrightarrow\ &\vec{e}^T (\mathbf{I})\, \vec{e} = \lambda^2 \vec{e}^T \vec{e} \\
\Leftrightarrow\ &\vec{e}^T \vec{e} = \lambda^2 \vec{e}^T \vec{e} \\
\Leftrightarrow\ &\lambda^2 = 1 \\
\Leftrightarrow\ &\lambda = 1
\end{aligned}$$

(the negative solution $\lambda = -1$ corresponds to reflection). Thus, all rotation matrices will have 1 as one of its eigenvalues. The corresponding eigenvector $\vec{e}$ satisfies $R\vec{e} = \vec{e}$. This is the axis of rotation - the set of points that stay where they were post rotation.

PYTHON NUMPY CODE FOR EIGEN VALUES AND VECTORS OF ROTATION MATRICES - AXIS OF ROTATION

Listing 2.19: Axis of Rotation

```
R = np.array([[0.7071,  0.7071,  0],
              [-0.7071, 0.7071,  0],
              [0,       0,       1]])

l, e = LA.eig(R)

axis_of_rotation = e[:, np.where(l == 1.0)]
axis_of_rotation = np.squeeze(axis_of_rotation)

assert np.allclose(axis_of_rotation, np.array([0, 0, 1]))

p = np.random.randint(0, 10) * axis_of_rotation
assert np.allclose(np.matmul(R, p), p)
```

Matrix $R = \begin{bmatrix} cos\,(45^\circ) & sin\,(45^\circ) & 0 \\ -sin\,(45^\circ) & cos\,(45^\circ) & 0 \\ 0 & 0 & 1 \end{bmatrix}$ rotates 45° about Z-axis

Numpy function eig() computes eigen values/vectors

All rotation matrices will have an eigen value 1. The corresponding eigen vector is the axis of rotation (here Z-axis).

Obtain the eigenvector for eigen value 1

Numpy function where() returns the indices at which the specified condition is true

axis of rotation is the Z-axis

Take a random point on this axis - apply the rotation to this point - its position does not change

2.15 Matrix Diagonalization

In section 2.12 we studied linear systems and their importance in machine learning. We also remarked that the standard mathematical process of solving linear systems via matrix inversion is not very desirable from machine learning point of view. In this section, we will see one method of solving linear systems without matrix inversion. In addition, this section will help us to develop the insights necessary to understand quadratic forms and eventually PCA (Principal Component Analysis) - one of the most important tools in data science.

Consider a $n \times n$ matrix A with n linearly independent eigenvectors. Let S be a matrix with these eigenvectors as its columns. That is

$$A\vec{e_1} = \lambda_1\vec{e_1}$$
$$A\vec{e_2} = \lambda_2\vec{e_2}$$
$$\vdots \quad \vdots$$
$$A\vec{e_n} = \lambda_n\vec{e_n}$$

and $S = [\vec{e}_1\ \vec{e}_2 \cdots \vec{e}_n]$.

Then

$$AS = A\begin{bmatrix}\vec{e_1} & \vec{e_2} & \cdots \vec{e_n}\end{bmatrix} = \begin{bmatrix}A\vec{e_1} & A\vec{e_2} & \cdots A\vec{e_n}\end{bmatrix} = \begin{bmatrix}\lambda_1\vec{e_1} & \lambda_2\vec{e_2} & \cdots \lambda_n\vec{e_n}\end{bmatrix}$$

$$= \begin{bmatrix}\vec{e_1} & \vec{e_2} & \cdots \vec{e_n}\end{bmatrix}\begin{bmatrix}\lambda_1 & 0 & \cdots & 0\\ 0 & \lambda_2 & \cdots & 0\\ \vdots & \vdots & \vdots & \vdots\\ 0 & 0 & \cdots & \lambda_n\end{bmatrix} = S\Lambda$$

Where $\Lambda = \begin{bmatrix}\lambda_1 & 0 & \cdots & 0\\ 0 & \lambda_2 & \cdots & 0\\ \vdots & \vdots & \vdots & \vdots\\ 0 & 0 & \cdots & \lambda_n\end{bmatrix}$ is a diagonal matrix with the eigenvalues of A on the diagonal and 0 everywhere else.

Thus, we have

$$AS = S\Lambda$$

which leads to

$$A = S\Lambda S^{-1}$$

and

$$\Lambda = S^{-1}AS$$

If A is symmetric, then its eigenvectors are orthogonal. Then $S^TS = SS^T = \mathbf{I} \Leftrightarrow S^{-1} = S^T$ and we get the diagonalization of A

$$A = S\Lambda S^T$$

Note that diagonalization is not unique -a given matrix maybe diagonalized in multiple ways.

2.15.1 Python Numpy code for Matrix diagonalization

Fully functional code for this section, executable via Jupyter-notebook, can be found at https://nbviewer.jupyter.org/github/krishnonwork/mathematical-methods-in-deep-learning-ipython/blob/master/python/ch2/2.15-mat-diagonalization-numpy.ipynb.

Listing 2.20: Diagonalization of matrix

```
1  def diagonalise(matrix):
2      try:
3          l, e = np.linalg.eig(A)
4
5          sigma = np.diag(l)
6
7          return e, sigma, np.linalg.inv(e)
8      except np.linalg.LinAlgError:
9          print("Cannot diagonalise matrix!")
10
11 A = np.array([[0.7071, 0.7071, 0],
12                [-0.7071, 0.7071, 0],
13                [0, 0, 1]])
14
15 S, sigma, S_inv = diagonalise(A)
16
17 A1 = np.matmul(S, np.matmul(sigma, S_inv))
18
19 assert np.allclose(A, A1)
```

Diagonalization is factorizing a matrix $A = S\Sigma S^{-1}$.
S is a matrix with eigen vectors of A as columns.
Σ is a diagonal matrix with eigen values of A in diagonal.

Numpy function eig() returns eigen values and vectors.

Numpy function diag() creates diagonal matrix of given values

Return the three factors

Create a matrix A

Reconstruct A from its factors

Verify that the reconstructed matrix is same as original one.

2.15.2 Solving Linear Systems without Inverse via Diagonalization

Diagonalization has many practical applications. Let us study one now. In general, matrix inversion (i.e., computation of A^{-1} is a very complex process which is numerically unstable. Hence, solving $A\vec{x} = \vec{b}$ via $\vec{x} = A^{-1}\vec{b}$ is to be avoided when possible. In the particular case of a square symmetric matrix with *n* distinct eigenvalues, diagonalization can come to the rescue. We can solve in multiple steps: We first diagonalize *A*

$$A = S\Lambda S^T$$

Then

$$A\vec{x} = \vec{b}$$

can be written as: $S\Lambda S^T\vec{x} = \vec{b}$.

where S is the matrix with eigenvectors of A as its columns:

$$S = [\vec{e}_1\ \vec{e}_2 \cdots \vec{e}_n]$$

(Since A is symmetric, these eigenvectors are orthogonal. Hence $S^TS = SS^T = \mathbf{I}$) The solution can be obtained in a series of very simple steps as shown below

$$\underbrace{S\,\Lambda\,\underbrace{S^T\vec{x}}_{y_2}}_{y_1} = \vec{b}$$

First solve

$$S\vec{y}_1 = \vec{b}$$

As

$$\vec{y}_1 = S^T\vec{b}$$

Notice that both transpose and matrix vector multiplications are simple and numerically stable operations unlike matrix inversion. Then we get

$$\Lambda(S^T\vec{x}) = \vec{y}_1$$

Now solve

$$\Lambda\vec{y}_2 = \vec{y}_1$$

as

$$\vec{y}_2 = \Lambda^{-1}\vec{y}_1$$

Note that since Λ is a diagonal matrix, inverting it is trivial

$$\begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & \lambda_n \end{bmatrix}^{-1} = \begin{bmatrix} \frac{1}{\lambda_1} & 0 & \cdots & 0 \\ 0 & \frac{1}{\lambda_2} & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & \frac{1}{\lambda_n} \end{bmatrix} \tag{2.33}$$

As final step, solve

$$S^T\vec{x} = \vec{y}_2$$

As

$$\vec{x} = S\vec{y}_2$$

Thus we have obtained $\vec{x}$ without a single complex or unstable step.

2.15.3 Python Numpy code for Solving Linear Systems via diagonalization

Let us try solving the following set of equations:

$$\begin{aligned} x + y + z &= 8 \\ 2x + 2y + 3z &= 15 \\ x + 3y + 3z &= 16 \end{aligned}$$

This can be written using matrices and vectors as where

$$A\vec{x} = \vec{b}$$

$$A = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 2 & 3 \\ 1 & 3 & 3 \end{bmatrix} \quad \vec{x} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \quad \vec{b} = \begin{bmatrix} 8 \\ 15 \\ 16 \end{bmatrix}$$

Note that A is a symmetric matrix. It has orthogonal eigenvectors. The matrix with eigenvectors of A in columns is orthogonal. Its transpose and inverse are same.

Listing 2.21: Solving linear systems using diagonalization

```
A = np.array([[1, 2, 1], [2, 2, 3], [1, 3, 3]])
assert np.all(A == A.T)
b = np.array([8, 15, 16])

x_0 = np.matmul(np.linalg.inv(A), b)

w, S = np.linalg.eig(A)
y1 = np.matmul(S.T, b)
y2 = np.matmul(np.diag(1/ w), y1)
x_1 = np.matmul(S, y2)


assert np.allclose(x_0, x_1)
```

create a symmetric matrix A

assert that A maybe symmetric

create a vector $\vec{b}$

solve $A\vec{x} = \vec{b}$ using matrix inversion, $\vec{x} = A^{-1}\vec{b}$
Note: matrix inversion is numerically unstable

$S\Lambda \underbrace{S^T\vec{x}}_{\vec{y}_2} = \vec{b}$
$\underbrace{\quad\quad}_{\vec{y}_1}$

Solve $A\vec{x} = \vec{b}$ via diagonalization. $A = S\Sigma S^T$.

1. solve: $S\vec{y}_1 = \vec{b}$ as $\vec{y}_1 = S^T\vec{b}$ (no matrix inversion)
2. solve: $\Lambda\vec{y}_2 = \vec{y}_1$ as $\vec{y}_2 = \Lambda^{-1}\vec{y}_1$
(inverting a diagonal matrix is easy, see eq 2.33)
3. solve: $S^T\vec{x} = \vec{y}_2$ as $\vec{x} = S\vec{y}_2$ (no matrix inversion)

verify that the two solutions are same

2.15.4 Matrix powers using diagonalization

If matrix A can be diagonalized then

$$A = S\Lambda S^{-1}$$

$$A^2 = S\Lambda S^{-1}\, S\Lambda S^{-1} = S\Lambda \mathbf{I} \Lambda S^{-1} = S\Lambda^2 S^{-1}$$
$$A^n = \cdots = \cdots = S\Lambda^n S^{-1}$$

For a diagonal matrix $\Lambda = \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & \lambda_n \end{bmatrix}$, the n^{th} power is simply $\Lambda^n = \begin{bmatrix} \lambda_1^n & 0 & \cdots & 0 \\ 0 & \lambda_2^n & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & \lambda_n^n \end{bmatrix}$

If one needs to compute various powers of a $m \times m$ matrix A at various times, one should precompute the matrix S and compute any power with only $O(m)$ operations - compared to (nm^3)operations necessary for naive computations.

2.16 Spectral Decomposition of a Symmetric Matrix

We have seen subsection 2.15 that a square symmetric matrix with distinct eigenvalues can be decomposed as

$$A = S\Lambda S^T$$

where $S = [\vec{e}_1\ \vec{e}_2 \cdots \vec{e}_n]$. Thus,

$$A = \begin{bmatrix} \vec{e}_1 & \vec{e}_2 & \cdots \vec{e}_n \end{bmatrix} \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & \lambda_n \end{bmatrix} \begin{bmatrix} \vec{e}_1^{\,T} \\ \vec{e}_2^{\,T} \\ \vdots \\ \vec{e}_n^{\,T} \end{bmatrix}$$

The above equation can be rewritten as

$$A = \lambda_1 \vec{e}_1 \vec{e}_1^{\,T} + \lambda_2 \vec{e}_2 \vec{e}_2^{\,T} + \cdots + \lambda_n \vec{e}_n \vec{e}_n^{\,T} \tag{2.34}$$

Thus a square symmetric matrix can be written in terms of its eigenvalues and eigenvectors. This is the spectral resolution theorem.

2.16.1 Python numpy code for Spectral Decomposition of Matrix

Listing 2.22: Matrix powers using diagonalization

```
1  def spectral_decomposition(A):
2    assert len(A.shape) == 2\
3           and A.shape[0] == A.shape[1]
4           and np.all(A == A.T)
5
6    l, e = np.linalg.eig(A)
7
8    assert len(np.unique(l)) == A.shape[0],\
9           "Eigen values are not distinct!"
10
11   C = np.zeros((A.shape[0], A.shape[0], A.shape[0]))
12
13   for i, lambda_i in enumerate(l):
14       e_i = e[:, i]
15       e_i = e_i.reshape((3, 1))
16       C[i, :, :] = lambda_i * np.matmul(e_i, e_i.T)
17   return C
18
19 A = np.array([[1, 2, 1], [2, 2, 3], [1, 3, 3]])
20
21 C = spectral_decomposition(A)
22
23 A1 = C.sum(axis=0)
24
25 assert np.allclose(A, A1)
```

assert that A is a 2D tensor (i.e., matrix) and square symmetric

i.e., $A.shape[0]$ (num rows) $\triangleq$ $A.shape[1]$ (num columns)

and $A == A^T$

numpy function eig() returns eigen vectors/values

Define a 3D tensor C of shape $n \times n \times n$ to hold the n components from equation 2.34.

Each term $\lambda_i \vec{e}_i \vec{e}^T$ is a $n \times n$ matrix.

There are n such terms, all compactly held in tensor C.

$C[i] = \lambda_i \vec{e}_i \vec{e}^T$

Initialize C

Compute elements of C

Reconstruct A by adding its components stored in C

Numpy function sum() adds elements of a given tensor along a specified axis.

Adding elements of C along axis 0 means doing $\sum_i C[i]$

Verify that the matrix reconstructed from components matches the original

2.17 An application relevant to Machine Learning - finding the axes of a hyper-ellipse

The notion of an ellipse in high-dimensional space (aka hyper-ellipse) keeps coming back in various forms in machine learning. Here we will make a preliminary review of them. We will revisit these concepts later.

Recall the equation of ellipse from high school math lesson

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$$

This is a rather simple ellipse, 2 dimensional, centered at origin and its major and minor axes aligned with the coordinate axes. Denoting $\vec{x} = \begin{bmatrix} x \\ y \end{bmatrix}$ as the position vector, the same equation can be written as

$$\vec{x}^T \Lambda \vec{x} = 1$$

Where $\Lambda = \begin{bmatrix} \frac{1}{a^2} & 0 \\ 0 & \frac{1}{b^2} \end{bmatrix}$ is a diagonal matrix. Written in this form, the equation can be extended to beyond 2*D*, to a n-dimensional axis aligned ellipse centered at origin. Now, let us apply a rotation *R* to the axis. Then, every vector $\vec{x}$ transforms to $R\vec{x}$. The equation of the ellipse in the new (rotated) coordinate system is

$$\begin{aligned} & (R\vec{x})^T \Lambda (R\vec{x}) = 1 \\ \Leftrightarrow\ & \vec{x}^T (R^T \Lambda R) \vec{x} = 1 \end{aligned}$$

With $A = (R^T \Lambda R)$, the generalized equation of the ellipse is

$$\vec{x}^T A \vec{x} = 1$$

Note

- The ellipse is no longer axis aligned
- The matrix A is no longer diagonal
- *A* is symmetric. One can easily verify that $A^T = (R^T \Lambda R)^T = R^T \Lambda^T R = R^T \Lambda R$ (remember transpose of a diagonal matrix is itself)

If, in addition, we want to get rid of the "centered at origin" assumption, we get

$$(\vec{x} - \mu)^T A (\vec{x} - \mu) = 1$$

Now, let us flip the problem around. Suppose we have a generic n-dimensional ellipse as above. How do we compute its axes directions?

Clearly, if we could rotate the coordinate system so that the matrix in the middle is diagonal, we are done. Diagonalization (see section 2.15) is the answer. To be specific, we find the matrix S with eigenvectors of A in its columns. This is a rotation matrix (being orthogonal since A is symmetric). We transform (rotate) the coordinate system by applying this matrix. In this new coordinate system, the ellipse is axis aligned. Stated in another way, the new coordinate axes - these are the eigen vectors of A - yield the axes of the ellipse.

2.17.1 Python numpy code for Hyper Ellipses

Let us try finding the axes of the hyper ellipse described by the equation $5x^2 + 6xy + 5y^2 = 20$. Note: The actual ellipse we use as example is 2D (to facilitate visualization), but the code we develop will be general and extensible to multi-dimensions.

The ellipse equation can be written using matrices and vectors as $\vec{x}^T A \vec{x} = 1$ where

$$A = \begin{bmatrix} 5 & 3 \\ 3 & 5 \end{bmatrix} \vec{x} = \begin{bmatrix} x \\ y \end{bmatrix}.$$

To find the axes of the hyper ellipse, we need to transform the coordinate system so that the matrix in the middle becomes diagonal. Here is how this can be done: If we diagonalize A into $S\Sigma S^{-1}$, then the ellipse equation becomes $\vec{x}^T S\Sigma S^{-1}\vec{x}$ = 1 where Σ is a diagonal matrix. Since A is symmetric, its eigenvectors are orthogonal. Hence, the matrix containing these eigenvectors as columns is orthogonal, i.e., $S^{-1} = S^T$. In other words, S is a rotation matrix. So the ellipse equation becomes $\vec{x}^T S\Sigma S^T\vec{x}$ = 1 or $(\vec{x}^T S)\,\Sigma\,(S^T\vec{x})$ = 1 or $\vec{y}^T\Sigma\vec{y}$ = 1 where $\vec{y} = S^T\vec{x}$. This is of the desired form since Σ is a diagonal matrix. Remember, S is a rotation matrix. Thus, rotating the coordinate system by S aligns the coordinate axes with the ellipse axes.

Listing 2.23: Axes of hyper ellipse

```
ellipse_eq = sy.Eq(5*x**2 + 5*y**2 + 6*x*y, 20)


A = np.array([[5, 3], [3, 5]])

l, S = np.linalg.eig(np.array(A).astype(np.float64))

x_axis_vec = np.array([1, 0])

first_eigen_vec = S[:, 0]

dot_prod = np.dot(x_axis_vec, first_eigen_vec)

theta = math.acos(dot_prod)
theta = math.degrees(theta)
```

Equation of ellipse: $5x^2 + 6xy + 5y^2 = 20$ or $\vec{x}^T A\vec{x} = 20$ where $A = \begin{bmatrix} 5 & 3 \\ 3 & 5 \end{bmatrix}, \vec{x} = \begin{bmatrix} x \\ y \end{bmatrix}$

X-axis vector

Major Axis of the ellipse

dot product between two vectors is cosine of the angle between them

Angle between ellipse major axis and X-axis, 45° - see Fig 2.16

2.18 Summary

In this chapter, we studied vectors and matrices with the backdrop of machine learning and data science.

- We studied definitions of vectors, matrices and tensors along with their geometrical significances with the aid of machine learning centric examples.
- We studied various vector and matrix manipulation operations that are important for machine learning, data science.

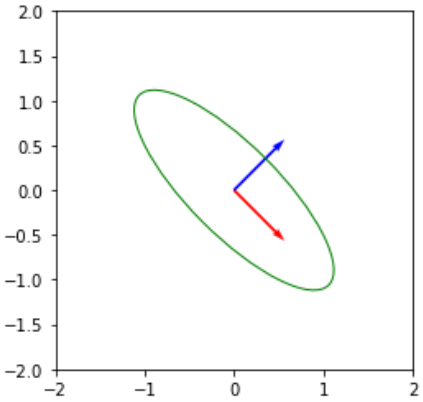


Figure 2.16: Note that the ellipse major axis is forming an angle of 45 degrees with X axis. Rotating coordinate system by this angle will align ellipse axes with coordinate axes. Subsequently, the first principal vector will also lie along this direction

- We studied the dot product and its significance in machine learning and data science to measure similarity between vectors.
- We studied the concept of orthogonality, which is fundamental to machine learning, data science.
- We studied linear systems of equations and various methods of solving them, including the Moore-Penrose inverse.
- We studied eigenvalues and eigenvectors and how to use them for diagonalizing a matrix, concepts that lead to various machine learning applications described in the next chapter.
- We studied python numpy (numerical python) codes to implement the various mathematical concepts we learnt. We also studied PyTorch tensors and how to use them in conjunction with numpy arrays.

3

Introduction to Vector Calculus from Machine Learning point of view

The core concept of machine learning is simple enough. We took a first look at it in section 1.3. Then in section 2.8.2 we studied classifiers as a special case. Let us revisit these with a different example this time. Also, in section 1.3 we skipped on the topic of error minimization. This time, armed with our knowledge of gradients, we will study the topic.

The python numpy/pytorch code for this section, in the form of fully functional and executable Jupyter-notebooks can be found at nbviewer.jupyter.org/github/../ch3/.

Suppose we want to create a classifier machine that classifies whether an image contain a car or a giraffe. Such classifiers, with only two classes, are known as *binary classifiers*. We identify a set of input signals which are collected together in an input vector denoted $\vec{x}$. In case of convolutional neural networks, aka CNNs, the inputs are the pixel values of the image. The image is usually scaled to a fixed size, say 224×224. Thus the image is representable as a matrix

$$X = \begin{bmatrix} X_{0,0} & X_{0,1} & \cdots & X_{0,223} \\ X_{1,0} & X_{1,1} & \cdots & X_{1,223} \\ \vdots & \vdots & \vdots & \vdots \\ X_{223,0} & X_{223,1} & \cdots & X_{223,223} \end{bmatrix}$$

Each element of the matrix, $X_{i,j}$ is a pixel color value in the range $[0, 255]$.

One brief aside is necessary at this point. In the previous chapters, we have always seen a *vector* to be the input to a machine learning system. In fact, the vector representation of the input allowed us to see it as a point in a high dimensional space. This lead to a lot of geometric insights about classification. But here, our input is an image which is akin to a *matrix* rather than a vector. Are those intuitions applicable when our input is a matrix as opposed to a vector? The answer is yes. A matrix can always be converted into a vector by a

process called *rasterization*. During rasterization, one iterates over the elements of the matrix left to right and top to bottom, storing successive encountered elements into a vector. The resulting vector is the rasterized vector. It has the same elements as the original matrix, only organized in a different fashion. The length of the rasterized vector will be equal to the product of row count and column count of the matrix. The rasterized vector for the above matrix X will have $224 \times 224 = 50176$ elements.

$$\vec{x} = \begin{bmatrix} x_0 = X_{0,0} \\ x_1 = X_{0,1} \\ \vdots \\ x_{223} = X_{0,223} \\ x_{224} = X_{1,0} \\ x_{225} = X_{1,1} \\ \vdots \\ x_{50175} = X_{223,223} \end{bmatrix}$$ where $x_i \in [0, 255]$ are values of the image pixels.

Thus, a 224×224 input image can be viewed as a vector (equivalently a point) in a 50176 dimensional space.

An example of such a space is geometrically depicted in Figures 3.1 in the context of giraffe and car classification in images. The points corresponding to giraffe are marked 'g' and those corresponding to car are marked '*c*'. Another example space is depicted in 3.2 in the context of horse and zebra classification in images. Here the points corresponding to horses are marked '*h*' and those corresponding to zebras are marked '*z*'. Usually, the points belonging to classes of interest will occupy a very small portion (sub-space) in the vast high-dimensional space of inputs.

This is because there is always inherent commonality in members of a class. For instance, all giraffes have predominantly yellow color with a bit of black. Cars have certain shapes. Because of this, points belonging to a given class will not be distributed haphazardly in the high-dimensional input space. Rather they would loosely form a *cluster*.

Geometrically speaking, the classifier is a hyper-surface that separates the 'c' cluster from the 'g' cluster. In the simple case depicted in Figure 3.1 this classifier surface is a hyper-plane (we often call surfaces as hyper-surfaces and planes as hyper-planes when in high dimensions). But in the more difficult case of classifying horse vs zebra depicted in Figure 3.2 we need a non-linear (curved) surface.

In either case, we do not know the exact surface. We do know that it takes the image as input and emits the classification as output

$$f(\vec{x}) = \begin{cases} 0 & \text{(giraffe)} \\ 1 & \text{(car)} \end{cases}$$

but we do not know the function $f(\vec{x})$. We do know the desired output value of the surface/-function for a specific set of input values - these are the training data – $(x^{(i)}, y^{(i)})$. We will have

to estimate the function $f(\vec{x})$ from the training data - an overall exercise that goes by the name modeling. This is the essence of machine learning.

As indicated in section 1.3, modeling the unknown function f has two steps.

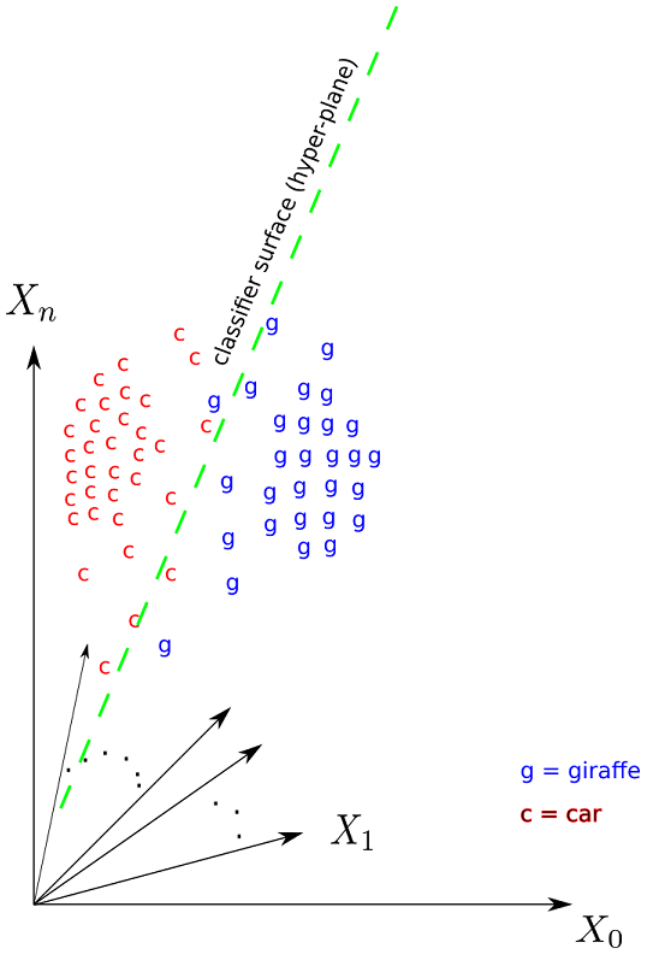


Figure 3.1: Geometric depiction of Classification problem. In the multidimensional input space, each instance of the classes to be separated correspond to a point. E.g., the points marked 'c' denote cars, 'g' denotes giraffe. In the friendly case, the points would form reasonably distinct clusters. Then the classification can be done with a relatively simple surface, e.g., a hyper-plane here. The model designer chooses the function class (e.g., hyper-plane) by examining the complexity of the problem. The exact parameters of the hyper-plane - orientation and position - are determined via training.

- **Select model architecture**: Choose the parametric function ϕ to mimic f .

Mathematically, we denote our model ϕ as ϕ ($\vec{x}$;$\vec{w}$, b). This basically means that the function ϕ takes a vector $\vec{x}$ as input and has a set of weight and bias parameters $\vec{w}$, b. The model designer will choose ϕ based on his/her understanding of the problem and experience.

- **Training**: Estimate the parameters $\vec{w}$, b such that ϕ emits the known correct output (as closely as possible) on the training data inputs. This is typically done via an iterative process. Training data comprises input vector instances $\vec{x}_i$ and corresponding known outputs y_i. For instance, the training data for the car vs giraffe classifier will comprise a number of images along with a *label* ('c' or 'g'). The labels for the training images are often created manually. In each iteration, we adjust the parameters $\vec{w}$, b such that the model makes a little bit less error. That is to say, the model output $y = \phi$ ($\vec{x}_i$; $\vec{w}$, b) gets a little bit closer to the target output y_i over all values of i. We iterate over all instances of the training data repeatedly. Each set of iterations over all training data instances is called an epoch. After many epochs, the model starts yielding correct output on arbitrary inputs, not just the training inputs. We say the model is trained.

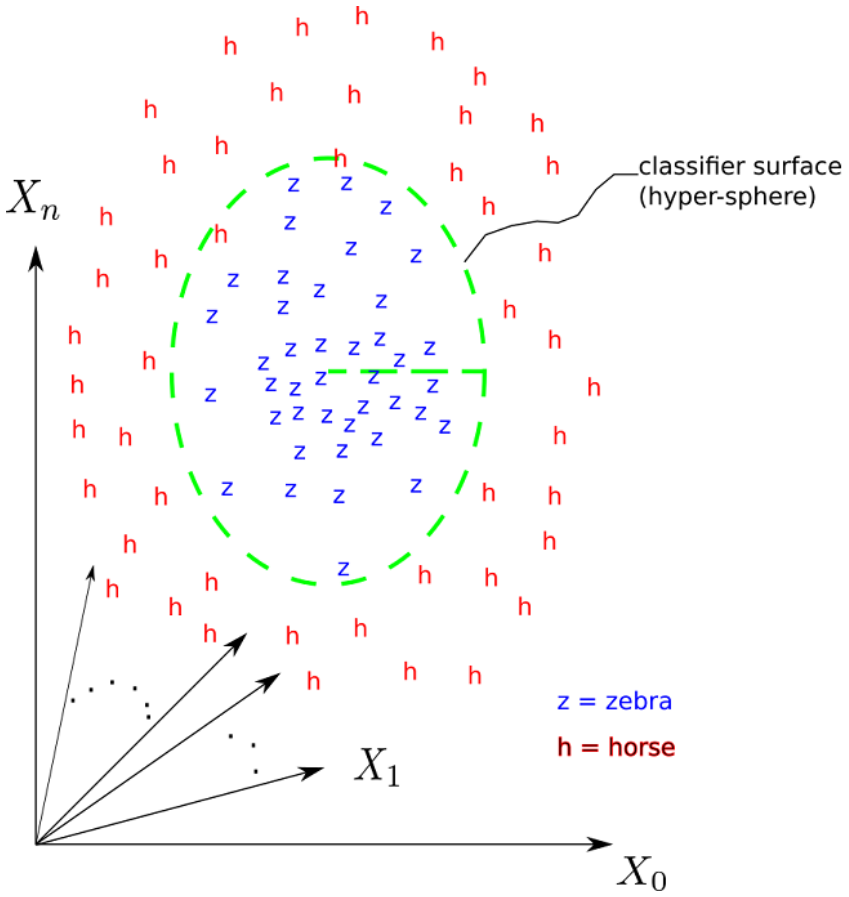


Figure 3.2: Geometric depiction of a slightly harder Classification problem. In the multidimensional input space, each instance of the classes to be separated correspond to a point. E.g., the points marked 'h' denote horses, 'z' denotes zebra. In this a bit more difficult case, the classification has be done with a curved (non-planar) surface, e.g., a hyper-sphere here. The model designer chooses the function class (e.g., hyper-sphere) by examining the complexity and physical nature of the problem. The parameters of the hypersphere - radius and center - are determined via training.

In the case of classifiers, the function $\phi\ (\vec{x};\vec{w},\ b)$ corresponds to a hyper-surface that separates the cluster of points belonging to individual classes. E.g., in the case of the binary classification problem depicted in 3.1, $\phi\ (\vec{x};\vec{w},\ b)$ may represent a plane (shown by the dashed line). Points on one side of the plane are classified as car while points on the other side are classified as giraffe. In Figure 3.2 a good planar separation does not exist - we need a non-linear separator - e.g., the spherical separator shown via dashed lines.

The separating surface is sometimes referred to as *decision boundary*. In the special case of binary classifiers, the *sign* of the expression $\phi(\vec{x};\vec{w}, b)$ representing the decision boundary has a special significance (see section 3.1).

For the simpler cases, where we think a linear model is sufficient (e.g., the one depicted in Figure 3.1), we can use the model architecture

$$\phi(\vec{x};\vec{w}, b) = \vec{w}^T\vec{x} + b$$

From equation 2.16 we know this is a planar separator.

For the more difficult cases, where we do not think a linear model will do (e.g., the one depicted in Figure 3.2), we can try the model architecture

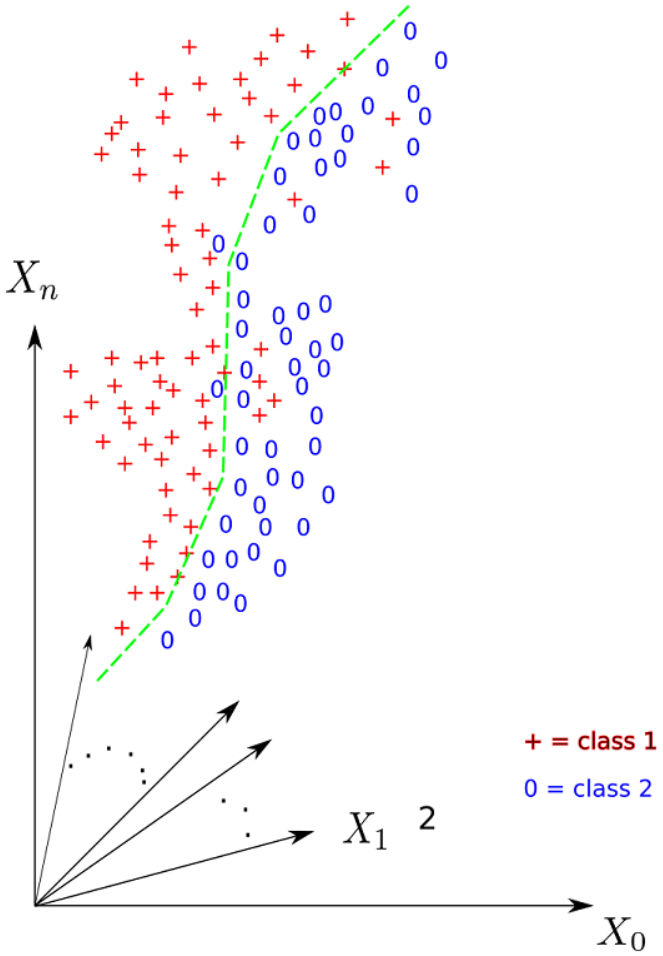


Figure 3.3: In real life problems, the separating surface is often not a well known surface like plane of sphere. And often the classification is not perfect - points fall on the wrong side of the separator.

$$\phi(\vec{x};\vec{w},b) = \vec{x}^T \begin{bmatrix} w_0 & 0 & \cdots & 0 \\ 0 & w_1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & w_n \end{bmatrix} \vec{x} + b = 0$$

The above equation represents a sphere in 3D ($n = 2$) - we will skip the proof of that here. It should be noted that in general, we do not even have a diagram showing the spatial distribution of the training data points. Indeed it is difficult to visualize high dimensional data on the 2D plane of a page.

3.1 Significance of the sign of the separating surface in binary classification

Consider a line in a 2D plane corresponding to the equation

$$y + 2x + 1 = 0$$

All points *on* the line will have x, y coordinate values satisfying the equation. The line divides the 2D plane into two half planes. All points on one half plane will have x, y values such that $y + 2x + 1$ is negative. All points in the other half plane will have x, y values such that $y + 2x + 1$ is positive. This is shown in Figure 3.4.

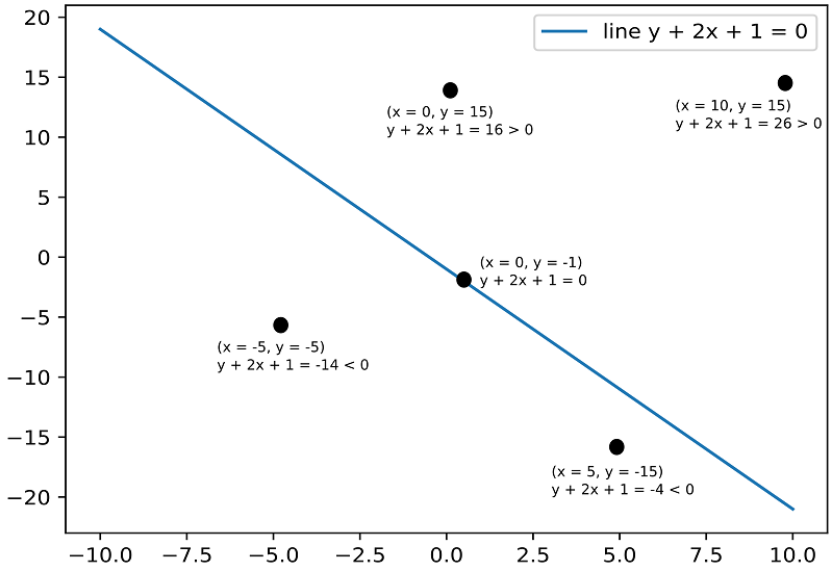


Figure 3.4: Given a point (x_0, y_0) and a separator $y + 2x + 1 = 0$, we can tell which side of the separator the point lies from the sign of $y_0 + 2x_0 + 1$

The idea can be extended to other surfaces and higher dimensions. Thus, binary classification can be viewed as estimating an optimal separating surface $\phi(\vec{x}; \vec{w}, b)$. The parameters $\vec{w}, b$ are estimated during training. Then, given any input vector $\vec{x}$, one can compute the sign of $\phi(\vec{x}; \vec{w}, b)$ to predict the class.

3.2 Estimating Model Parameters: Training

How do we estimate the parameters $\vec{w}, b$? As stated above, this is where training comes. We take a set of input vectors $\vec{x}^{(0)}$, $\vec{x}^{1}$, $\cdots\vec{x}^{N}$ for which the outputs are known. The known outputs are often created via human curation - a human being looks at the training input images and labels each image with the appropriate class, e.g., car vs giraffe or horse vs zebra. Each (*image, label*) pair constitutes a *training data instance*.

Thus overall training data comprises a set of labeled inputs (aka training data instances)

$$
\begin{array}{l}
(\vec{x}^{(0)}, y^{(0)}) \\
(\vec{x}^{(1)}, y^{(1)}) \\
\quad . \quad . \\
\quad . \quad . \\
\quad . \quad . \\
(\vec{x}^{(M)}, y^{(M)})
\end{array}
$$

Now we define a loss function. On a specific training data instance, this effectively measures the error made by the machine on that particular training data input-target pair ($x^{(i)}$, $y^{(i)}$). Although there are many sophisticated error functions more suitable for this problem, for now, let us use a squared error function for the sake of simplicity (this was introduced in section 2.5.4).

The squared error on the ith training data element is the squared difference between the output yielded by the model and the expected or target output

$$(e^{(i)})^2 = (\phi(\vec{x}^{(i)}; \vec{w}, b) - y^{(i)})^2 \tag{3.1}$$

The total loss (aka squared error) during training is

$$L(\vec{w}, b) = E^2(\vec{w}, b) = \sum_{i=0}^{i=N} \left(e^{(i)}\right)^2 \tag{3.2}$$

It should be noted that this total error is not a function of any specific training data instance. Rather, it is the *overall error over the entire training data set*. This is what we minimize by adjusting $\vec{w}$ and b. To be precise, we estimate $\vec{w}$ and b that will minimize $L(\vec{w}, b)$. In this context, it should also be noted that minimizing E^2 and E are equivalent.

3.3 Minimizing Error during Training a Machine Learning Model:

Gradient Vectors

The goal of training is to estimate the weights and bias parameter $\vec{w}, b$ that will minimize E. This is usually done by an iterative process. We start with random values of $\vec{w}, b$ and adjust these values so that the loss $L(\vec{w}, b) = E^2(\vec{w}, b)$ goes down at a high rate. Doing this many times is likely to take us close to the optimal values for $\vec{w}, b$. This is the essential idea behind the process of training a model in machine learning. It is important to note that we are minimizing the total error. This prevents us from over indexing on any particular training instance. If the training data is a well sampled set, the parameters $\vec{w}, b$ that minimizes loss over the training dataset will hold good during inferencing too.

How do we "adjust" $\vec{w}, b$ so that the value of loss $L = E^2$ goes down? This is where gradients come in. For any function $L(\vec{w}, b)$, the gradient with respect to $\vec{w}, b$, $\nabla_{\vec{w}, b} L(\vec{w}, b)$, indicates the direction along which maximum change in L occurs - gradient is the analog of derivative in 1-dimensional calculus. Intuitively, going down along the direction of the gradient of a function seems like the best strategy for minimizing the function value.

Geometrically speaking, if we start at an arbitrary point on the surface corresponding to $L(\vec{w}, b)$, and move along the direction of the gradient $\nabla_{\vec{w}, b} L(\vec{w}, b)$, then we will go towards the minimum with highest speed (this is discussed in detail through the rest of this section). In almost all machine learning approaches, we iteratively move towards the minimum by taking steps along $\nabla_{\vec{w}, b} L(\vec{w}, b)$. It should be noted that *the gradient is with respect to weights and not the input*. Thus the overall algorithm is shown in Algorithm 4.

Algorithm 4 Training a supervised model (overall idea)

Initialize $\vec{w}, b$ with random values
while $L(\vec{w}, b) >$ threshold **do**

$$\begin{bmatrix} \vec{w} \\ b \end{bmatrix} = \begin{bmatrix} \vec{w} \\ b \end{bmatrix} - \mu \nabla_{\vec{w},b} L(\vec{w}, b)$$

Recompute L on new $\vec{w}, b$
end while
$\vec{w}_* \leftarrow \vec{w}, b_* \leftarrow b$

The following points are to be noted.

- Mathematically, one should keep iterating until loss becomes minimal (i.e., gradient of the loss is zero). But in practice, one simply iterates until the accuracy is good enough for the purpose at hand.
- In each iteration, we are adjusting $\vec{w}, b$ along the gradient of the error function. We know
- from section 3.3 that this is the direction of maximum change for L. Thus, L is reduced at maximal rate.

- m is the learning rate - larger values imply longer steps and smaller values imply shorter steps.
- Longer steps are to be taken when far away from the minimum to progress quickly.
- Shorter steps are to be taken when near the minimum to avoid overshooting it.
- The simplistic approach outlined in algorithm 4 takes equal sized steps everywhere. In later chapters, we will study more sophisticated approaches where we try to sense how close to the minimum we are and vary step size accordingly.

3.3.1 Derivatives, Partial Derivatives, Change in function value and Tangents

We have seen that machine learning, training boils down to minimizing[12] a loss function $L(\vec{w})$ by adjusting the parameters $\vec{w}$ [13]. The minimization is done iteratively, starting at some random set of parameter values, and then continually adjusting them to lower the loss, until the loss is low enough. This raises the crucial question, given any value for the parameter set, how can we adjust the parameters so that the loss function gets reduced? Mathematically speaking, given current loss value $L(\vec{w})$ and parameter values $\vec{w}$, we want to determine the change in parameters $\delta\vec{w}$, such that the new (adjusted) loss value $L(\vec{w} + \delta\vec{w})$ is less than current loss value $L(\vec{w})$[14].

Equivalently, we want to determine $\delta\vec{w}$ that will make $\delta L = L(\vec{w}) - L(\vec{w} + \delta\vec{w})$ negative. If we have a consistent method of finding such $\delta\vec{w}$, we will continue adding that to the current value of parameters until the loss $L(\vec{w})$ is acceptably low.

The same question can be cast in a geometric fashion. Suppose we create plot of $L(\vec{w})$ versus $\vec{w}$ (Figure 3.6 shows a 2D version of this where the parameter set comprises a single value w). If we are at some arbitrary point on the curve L, say P, what is the direction we should move so as to get closer to the minimum (bottom of the bowl)?

Let us start with the simplest of cases, the parameter set comprises a single element w and the loss function $L(\vec{w})$ is a straight line $L(w) = mw + c$ (Figure 3.5 shows an example with $m = 2$ and $c = 1$). If we change w by a small amount, say dw, what is the corresponding change in $L(\vec{w})$?

In other words, we want to compute $\delta L = L(w + \delta w) - L(w)$. In this case, using the straight line equation, $\delta L = (m\ (w + \delta w) + c) - (m(w) + c) = m\ \delta w$. This is true for all dw. Hence, $\delta L/\delta w = m$. In particular, lim

[12] Most of what we say in these sections is equally applicable to the minimum and maximum. We choose to use the word minimum (plural minima) instead of the more mathematical word optimum (optima) for ease of reading. We can view machine learning as the business of minimizing loss. However, we can also view it as maximizing gain

[13] for the sake of brevity, here we will use the symbol w to denote all parameters - weight as well

[14] (note that if change in a quantity, say w, is infinitesimally small, we use the symbol dw to denote the change, whileif the change is small but not infinitesimally so, we use dw)

$$\lim_{w \to 0} \frac{\delta L}{\delta w} = m$$
$$\frac{dL}{dw} = m$$

Thus, for a straight line, the rate of change of *L* with respect to *w* is constant everywhere and equals the slope. Equivalently, the curve and its tangent is same everywhere.

We can rewrite this as

$$\delta L = \frac{dL}{dw} \delta w \tag{3.3}$$

The physical interpretation is this: to decrease *L*, the change in *w* must have opposite sign to the derivative. Geometrically speaking, we must follow the tangent towards the minimum.

Let us now consider a curve $L(w) = w^2$ (depicted in Figure 3.6). Now the rate of change of L depends on w, $\delta L / \delta w = 2w$. So the equation 3.3 does not hold in general. For instance, consider the point $P1$ in Figure 3.6. At this point, $w = 1$ and $L(w) = 1$ and $\delta L / \delta w = 2$. If we apply $\delta w = 2$, the new value of w is 3. The corresponding point on the curve is $P2$, with $L(w + \delta w) = 9$. Thus, $\delta L = 8$. This is *not* equal to $\delta L / \delta w * \delta w$ (which will be $2 * 2 = 4$ here).

But if δw is very very small (in mathematical parlance, infinitesimally small), we can approximate the curve locally by a straight line and then equation 3.3 holds. The derivative $\delta L / \delta w$ now corresponds to the slope of the tangent to the curve at the current point.

Thus in general, if we want to adjust the parameter value w so that some dependent loss $L(\vec{w})$ gets reduced, we must add a quantity dw to w whose sign is negative of the derivative $\delta L / \delta w$ evaluated at w. In the $(w, L(w))$ plot, this is equivalent to following the tangent downwards. This is what equation 3.3 tells us.

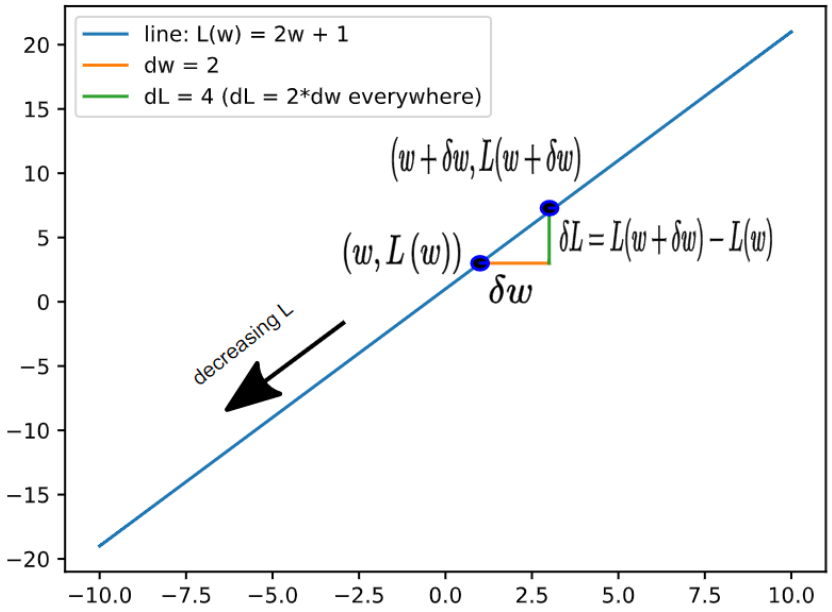


Figure 3.5: For a straight line $L(w) = mw + c$, $\delta L/\delta w = m$ everywhere (a constant). This implies $dL = mdw$. If δw denotes change we apply to w, $\delta L = \delta L / \delta w * \delta w$. To decrease L, $L + dL$ must be lesser than L, i.e., dL must be negative. δw must be of opposite sign to $\delta L/\delta w$ to achieve that. $\delta L/\delta w = m$ is in fact slope of the straight line. It can be viewed as tangent to the curve L. Following the tangent in the negative direction decreases the function.

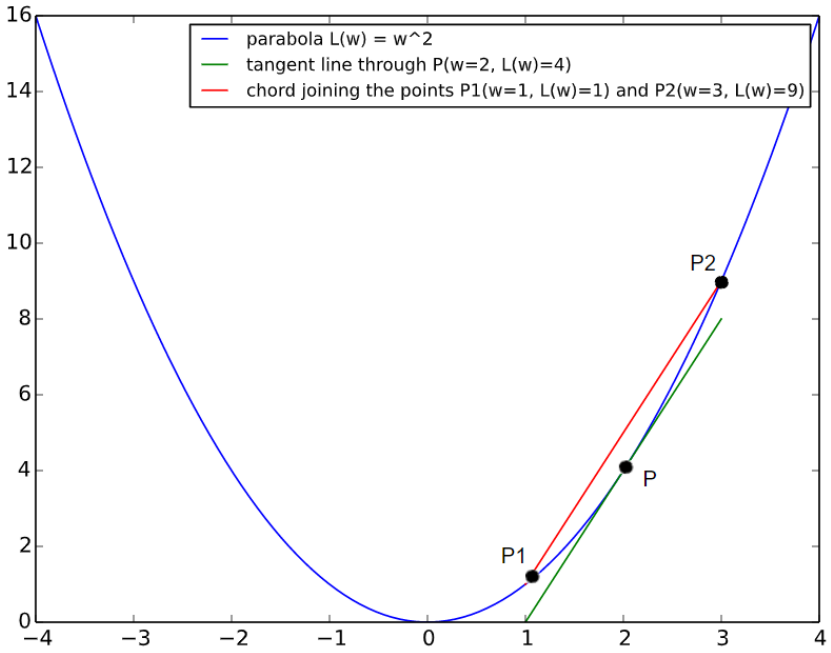


Figure 3.6: For curved lines $L(w) = w^2$, $\delta L / \delta w = 2w$. It is not a constant. Hence, $\delta L = \delta L/\delta w * \delta w$ is not generally true. But it holds for infinitesimally small δw. Then, $\delta L/\delta w$ becomes the slope of the tangent line to the curve. The intuition that δw must have opposite sign of $\delta L/\delta w$, i.e., to reduce L we must follow the tangent line in the negative direction, still holds. Later we will see that this is true for only a special class of functions called convex functions.

In higher dimensions, our loss function will be a function of many variables (tunable parameters). Hence, we will depict the input parameters with a *vector* instead of a scalar. In other words, $L(w)$ becomes $L(\vec{w})$. We can compute the change in $L(\vec{w})$ caused by a small vector displacement $\delta\vec{w}$.

This introduces a fundamental change in minimum computation. You see, now the parameter change is a vector, $\delta\vec{w}$ which not has a magnitude ($||\delta\vec{w}||$) but also a direction ($\hat{\delta\vec{w}}$). We can take the same sized step in the w space and the change in $L(\vec{w})$ will be different depending on the direction we step. The situation is illustrated in Figure 3.7.

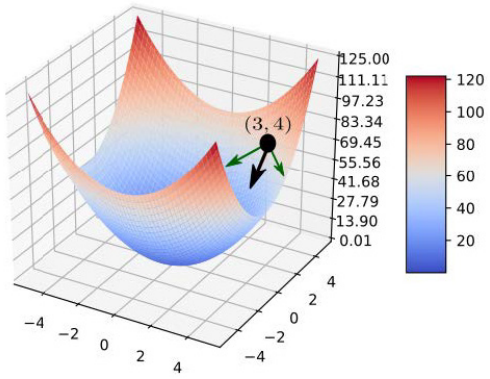


Figure 3.7: Plot for surface $L(\vec{w}) \equiv L\,(w_0,w_1) = 2w_0^2 + 3w_1^2$ against $\vec{w} \equiv (w_0,w_1)$. From an example point $P \equiv (w_0 = 3, w_1 = 4, L = 66)$ on the surface, one may travel in many directions to reduce L. Some of these are shown by arrows. The maximum reduction occurs when one travels along the thick black arrow - this happens when $\vec{w}$ is changed along $\delta\vec{w} = [-12, -24]^T$ which is negative of the gradient of the $L(\vec{w})$ at P

Figure 3.7 shows a function of two independent variables $L(\vec{w}) \equiv L(w_0,w_1) = 2w_0^2 + 3w_1^2$. Let usexamine this surface with a few concrete examples.

Suppose we are at $\vec{w} = \begin{bmatrix} w_0 = 3 \\ w_1 = 4 \end{bmatrix}$. The corresponding value of $L(\vec{w})$ is $2 * 32 + 3 * 4^2 = 66$.

Suppose, from this point, we suffer a small displacement $\delta\vec{w} = \begin{bmatrix} 0.0003 \\ 0.0004 \end{bmatrix}$, the new value is $L(\vec{w} + \delta\vec{w}) = L\,(3.0003, 4.0004) = 2 * 3.00032 + 3 * 4.00042 \approx 66.0132066$. Thus the displacement vector $\delta\vec{w} = \begin{bmatrix} 0.0003 \\ 0.0004 \end{bmatrix}$, causes a change $\delta L = 66.01320066 - 66 = 0.01320066$ in L.

Similarly, if the displacement vector is $\delta\vec{w} = \begin{bmatrix} 0.0004 \\ 0.0003 \end{bmatrix}$ we get $L(\vec{w} + \delta\vec{w}) = L(3.0004, 4.0003) = 2 * 3.00042 + 3 * 4.00032 \approx 66.0120006$. Thus this displacement vector causes a change $\delta L = 66.0120006 - 66 = 0.0120006$ in L.

As mentioned before, when our loss depends on a single scalar variable w, the displacement variable δw is also a scalar - it does not have a direction. But in higher dimensions, the displacement is a vector. It has both magnitude (length) and direction and the change in the function depends on both. Thus, in the above examples, the displacement vectors $\delta\vec{w} = \begin{bmatrix} 0.0003 \\ 0.0004 \end{bmatrix}$, and $\delta\vec{w} = \begin{bmatrix} 0.0004 \\ 0.0003 \end{bmatrix}$ both have the same length $\sqrt{0.0003^2 + 0.0004^2} = \sqrt{0.0004^2 + 0.0003^2} = 0.0005$ but the change caused by them is different.

What is the relationship between the displacement vector $\delta\vec{w}$ and the overall change in $L(\vec{w})$? We will look at this question in the general case with arbitrary dimensions. But for that, we need to know what a partial derivative is.

PARTIAL DERIVATIVES

Derivative $\delta L/\delta w$ of a function $L(w)$ indicates the rate of change of the function with respect to w. But if L is a function of many variables, how does it change if only one of those variables is changed? This question leads to the notion of partial derivatives. A partial derivative of a function of many variables is a derivative taken with respect to exactly one variable, treating all other variables as constants.

For instance, given $L(\vec{w}) \equiv L(w_0,x_1) = 2{w_0}^2 + 3{w_1}^2$, the partial derivative with respect to w_0 is

$$\frac{\partial L}{\partial w_0} = 4w_0$$
$$\frac{\partial L}{\partial w_1} = 3w_1$$

TOTAL CHANGE IN A MULTI-DIMENSIONAL FUNCTION: GRADIENTS

Partial derivatives estimate the change in a function if a single variable changes and the others stay constant. How do we estimate the change in a function's value if all the variables change together?

The total change can be estimated by taking a weighted combination of the partial derivatives. Let $\vec{w}$ and $\delta\vec{w}$ denote the point and displacement vector respectively.

$$\vec{w} = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_n \end{bmatrix}$$
$$\delta\vec{w} = \begin{bmatrix} \delta w_0 \\ \delta w_1 \\ \vdots \\ \delta w_n \end{bmatrix}$$

Then

$$\begin{aligned} \delta L(\vec{w}) &= L\left(\vec{w} + \delta\vec{w}\right) - L(\vec{w}) \\ &= \frac{\partial L}{\partial w_0}\delta w_0 + \frac{\partial L}{\partial w_1}\delta w_1 + \cdots + \frac{\partial L}{\partial w_n}\delta w_n \end{aligned} \tag{3.4}$$

Equation 3.4 is essentially saying that the total change in L is obtained by adding up the changes caused by displacements in individual variables. The rate of change of L with respect to change in w_i only is $\partial L/\partial w_i$. The displacement along the variable x_i is δw_i. Hence, the change caused by the i^{th} element of the displacement is $\partial L/\partial w_i * \delta w_i$ - this follows from equation 3.3. The total change then is obtained by adding the changes caused by individual elements of the displacement vector, i.e., summing over all i from 0 to n. This leads to equation 3.4.

Is there a way to represent equation 3.4 compactly? The answer is yes, by defining a quantity called gradient - the vector of all partial derivatives. Given an n-dimensional function $L(\vec{w})$,

its gradient is defined as

$$\nabla L\left(\vec{w}\right) = \begin{bmatrix} \frac{\partial L}{\partial w_0} \\ \frac{\partial L}{\partial w_1} \\ \vdots \\ \frac{\partial L}{\partial w_n} \end{bmatrix} \tag{3.5}$$

Using this, we can rewrite equation 3.4 as

$$\begin{aligned} \delta L\left(\vec{w}\right) &= L\left(\vec{w} + \delta\vec{w}\right) - L\left(\vec{w}\right) \\ &= \frac{\partial L}{\partial w_0}\delta w_0 + \frac{\partial L}{\partial w_1}\delta w_1 + \cdots + \frac{\partial L}{\partial w_n}\delta w_n \\ &= \left(\nabla L\left(\vec{w}\right)\right)^T \delta\vec{w} = \nabla L\left(\vec{w}\right) \cdot \delta\vec{w} \end{aligned} \tag{3.6}$$

Equation 3.6 tells us that the total change, ∂L in $L(\vec{w})$, caused by displacement $\delta\vec{w}$ from $\vec{w}$ in parameter space is the dot product between gradient vector $\triangledown L(\vec{w})$ and displacement vector $\delta\vec{w}$. This is the exact multi-dimensional analog of equation 3.3.

From section 2.5.6 we will recall that the dot product of two vectors (of fixed magnitude) attains maximum value when the vectors are aligned in direction. This yields a physical interpretation of the gradient vector - its direction is the direction in parameter space along which the multi-dimensional function is changing fastest. It is the multi-dimensional counterpart for derivative. This is why, in machine learning, when we want to minimize the loss function, we change the parameter values along the direction of the gradient vector of the loss function.

GRADIENT IS ZERO AT MINIMUM

Any optimum (i.e., maximum or minimum) of a function is a point of inflection. This means, the function will turn around at the point of optimum. In other words, the gradient direction on one side of the optimum will be opposite to that on the other side. If we try to travel smoothly from positive values to negative values, we must cross zero somewhere in between. Thus, the gradient is zero at the exact point of inflection (maximum or minimum). This is easiest to see in 2D and is depicted in Figure 3.8. However, the idea is general, it works in higher dimensions

too. The fact that gradient becomes zero at optimum is often used to algebraically compute the optimum. Following example illustrates this.

Take for instance, a simple example function $L(w_0, w_1) = \sqrt{w_0^2 + w_1^2}$. Its optimum will occur when its gradient is zero, i.e.,

$$\nabla_{\vec{w}} L = \begin{bmatrix} \frac{\partial L}{\partial w_0} \\ \frac{\partial L}{\partial w_1} \end{bmatrix} = \begin{bmatrix} 2w_0 \\ 2w_1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

The solution is

$$w_0 = 0, \quad w_1 = 0$$

the function attains its minimum value at the origin which agrees with our intuition.

From Figure 3.8, we can see that the function curves in one direction on one side of the minimum and curves in the opposite direction on the other side of the minimum. In other words, the minimum is always a point of inflection. The slope of the tangent is positive on one side negative on the other. At the exact minimum, the slope is zero. This agrees with our intuition that any smooth, continuous function must take a zero value in between positive and negative values.

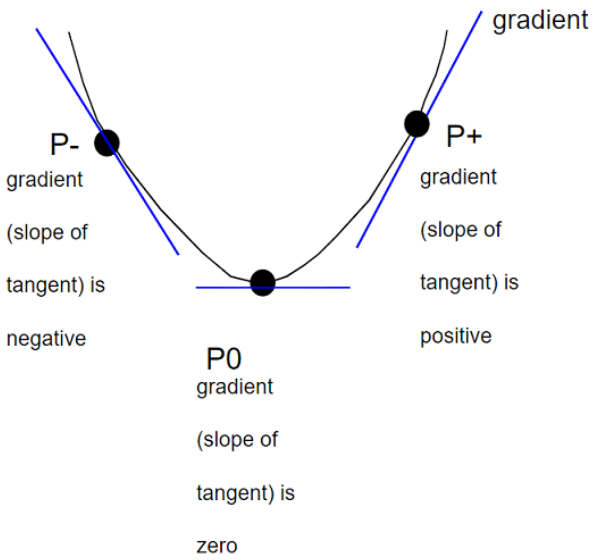


Figure 3.8: The minimum is always a point of inflection - meaning the function turns around at that point. If we consider any two points, e.g., P_- and P_+, on two sides of the minimum the gradient is positive on one side and negative on the other. Assuming the gradient changes smoothly, it must be zero in between at the minimum.

3.3.2 Level Surface representation and Loss Minimization

In Figure 3.7 we plotted the loss function $L(\vec{w})$ against the parameter values $\vec{w}$. In this section, we will study a different way of visualizing loss surfaces. This will lend further insight into gradients and minimization.

We will continue with our simple example function from last subsection to illustrate the idea.Consider a field $L(w_0, w_1) = \sqrt{w_0^2 + w_1^2}$ defined over the 2D W0,W1 plane. Its domain is the infinite 2D plane defined by the axes W0 and W1. Note that the function has constant values along concentric circles centered on the origin. For instance, at all points on the circumference of the circle $w_0^2 + w_1^2 = 1$ the function has the constant function value of 1. At all points on the circumference of the circle $w_0^2 + w_1^2 = 25$ the function has the constant function value of 5. Such constant function value curves on the domain are called level contours in 2D. This is shown as a heat-map in Fig 3.9. The idea of level contours can be generalized to higher dimensions where we have level surfaces or level hyper-surfaces. The reader should note that while the $\vec{w}$, $L(\vec{w})$ of representation of Figure 3.7 was on a $(n+1)$ dimensional space (where n is the dimensionality of $\vec{w}$), the level surface/contour representation is in n dimensional space.

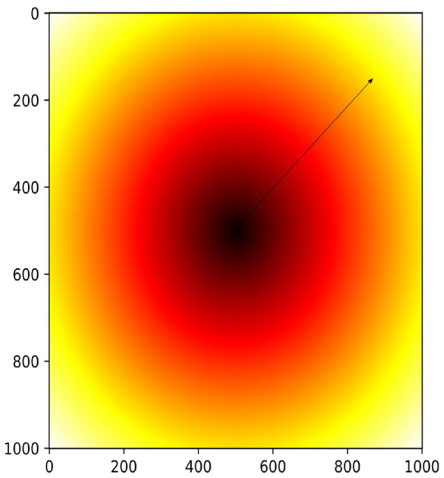


Figure 3.9: The domain of $L(w_0, w_1) = \sqrt{w_0^2 + w_1^2}$ shown as a heat-map of function values. Gradients point radially outward as shown by the arrowed line. The deepness of the red color changes fastest along the gradient (i.e., radii). This is the direction to follow to rapidly reach lower values of the function represented by the heatmap.

At any point on the domain, what is the direction along which the biggest *change* in function value occurs? The answer is *along the direction of the gradient*. The magnitude of the change corresponds to the magnitude of the gradient. In the current example, say we are at a point (w_0,w_1). There exists a level contour through this point - the circle with origin at center

passing through (w_0,w_1). If we move along the circumference of this circle, i.e., along the tangent to this circle, the function value does not change at all. In other words, at any point, the tangent to the level contour through that point is the direction of *minimal* change. On the other hand, *if we move perpendicular to the tangent, maximum change in the function value occurs*. The perpendicular to the tangent is known as *normal*. This is the direction of the gradient. *Gradient at any point on the domain is always normal to the level contour through that point, indicating the direction of maximum change in function value*. In Figure 3.9, the gradients are all parallel to the radii of the concentric circles. T Recall that during training a machine learning model, we essentially define a loss function in terms of a tunable set of parameters and try to minimize the loss by adjusting (tuning) the parameters. We start at a random point and iteratively progress towards the minimum.

Geometrically, this can be viewed as starting at an arbitrary point on the domain and continuing to move in a direction that minimizes the function value. Of course, we would like to progress to the minimum of the function value in as few iterations as possible. In Figure 3.9 the minimum i at the origin, which is also the center of all the concentric circles. Wherever we start, we will have to alway travel radially inwards to reach the minimum $(0, 0)$ of the function $\sqrt{w_0^2 + w_1^2}$.

In higher dimensions, level contours become level surfaces. Given any function $\vec{w}] \in \mathbb{R}^n$ we define level surfaces as $L(\vec{w})$ = *constant*. If we move along the level surface, the change in $L(\vec{w})$ is minimal (0). The gradient of a function at any point is normal to the level surface through that point. This is the direction along which the function value is changing fastest. Moving along the gradient, one passes from one level surface to another. This is shown in Figure 3.10. Here the function is 3D: $L(\vec{w})$ = $L(w_0,w_1,w_2)$ = ${w_2}^0$ + ${w_2}^1$ + ${w_2}^2$. The level surfaces ${w_2}^0$ + ${w_2}^1$ + ${w_2}^2$ = *constant* , for various values of the constant are concentric spheres, with origin as center. Gradient vector at any point is along the outward pointing radius of the sphere through that point. Another example is shown in Figure 3.11.

Here the function is 3D: $L(\vec{w})$ = f (w_0,w_1,w_2) = ${w_0}^2$ + ${w_1}^2$. The level surfaces ${w_0}^2$ + ${w_1}^2$ = *constant* for various values of the constant are coaxial cylinders, with W_3 as axis. Gradient vector at any point is along the outward pointing radius of the planar circle belonging to the cylinder through that point.

$$f(x_1, x_2, x_3) = f(\vec{x}) = x_1^2 + x_2^2 + x_3^2$$

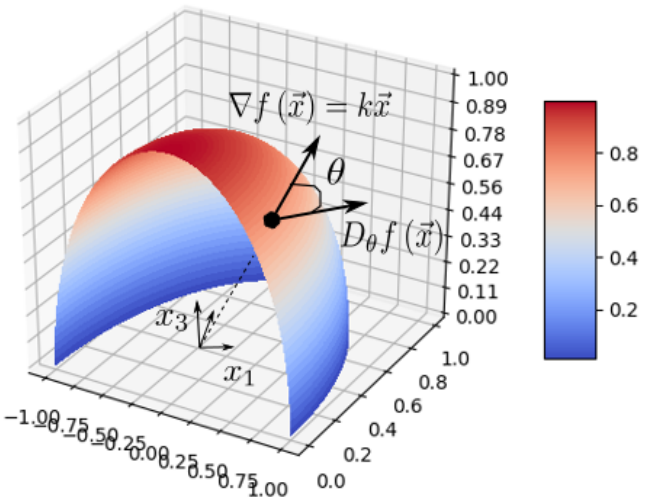


Figure 3.10: Gradient example in 3D: Function $L(w_0,w_1,w_2) = L(\vec{w}) = w_0^2 + w_1^2 + w_2^2$. Level surfaces $L(\vec{w}) =$ *constant*, are concentric spheres with origin as center. One such surface is partially shown in the diagram. $\nabla L(\vec{w}) = k\,[w_0\, w_1\, w_2]^T$. The gradient points radially outwards. Moving along the gradient, one goes from one level surface to another - corresponding to maximum change in $L(\vec{w})$. Moving along any direction orthogonal to the gradient, one stays on the same level surface (sphere) - corresponds to zero change in the function value. $Dq(\vec{w})$ denotes the directional derivative along the displacement direction making angle θ with the gradient. If we $\hat{l}$ denotes this displacement direction, $D_\theta(\vec{w}) = \nabla L(\vec{w}) \cdot \hat{l}$.

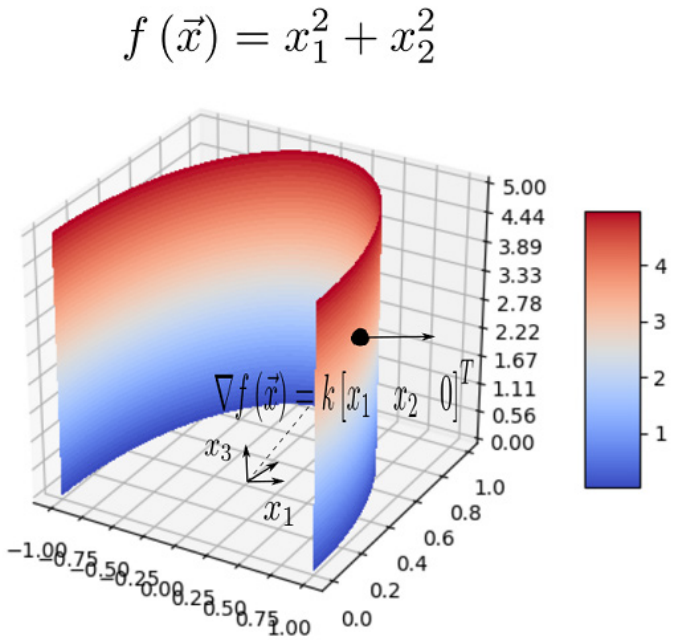


Figure 3.11: Gradient example in 3D: Function $L(w_0,w_1,w_2) = L(\vec{w}) = {w_0}^2 + {w_2}^2$. Level surfaces $f(\vec{w})$ = *constant*, are coaxial cylinders. One such surface is partially shown in the diagram. $\nabla L(\vec{w}) = k\,[w_0\, w_1\, 0]^T$. The gradient is normal to the curved surface of the cylinder, along the outward radius of the circle. Moving along the gradient, one goes from one level surface to another - corresponding to maximum change in $L(\vec{w})$. Moving along any direction orthogonal to the gradient, one stays on the same level surface (cylinder) - corresponds to zero change in function value.

3.4 Python numpy and PyTorch code for Gradient Descent, Error Minimization and Model Training

In this section, we will study implementations of model training. We will study numpy and a pytorch examples in which models are trained by minimizing errors via gradient descent. Before the code is presented, we will have a brief recap of the main ideas from a practical point of view.

Complete code for this section can be found at nbviewer.jupyter.org/github/../ch3/

3.4.1 Numpy and PyTorch code for Linear Models

If the true underlying function we are trying to predict is very simple, linear models suffice. Otherwise, we require non-linear models. Here we will study linear model. In machine learning, we identify the input and output variables pertaining to the problem at hand and cast the problem as generating outputs from input variables. All the inputs are represented together by the vector $\vec{x}$.

Sometimes there are multiple outputs, sometimes single output. Accordingly, we have an output vector $\vec{y}$ or output scalar *y*. Let us denote the function that generates the output from input vector as f, i.e., $y = f(\vec{x})$.

In real life problems, we do not know f. The crux of machine learning is to estimate *f* from a set of observed inputs $\vec{x}_i$ and their corresponding outputs y_i. Each observation can be depicted as a pair $\langle \vec{x}_i, y_i \rangle$. We model the unknown function f with a known function ϕ. ϕ is a parameterized function. Although the nature of ϕ is known, its parameter values are unknown. These parameter values are "learnt" via training. This means, we estimate the parameter values such that the overall error on the observations is minimized.

If $\vec{w}$, b denotes the current set of parameters (weights, bias), then the model will output $f(\vec{x}_i, \vec{w}, b)$ on the observed input $\vec{x}_i$. Thus the error on this i^{th} observation is $e_i^2 = (\phi(\vec{x}_i) - y_i)^2$. We can batch up several observations and add up the error into a batch error $L = \sum_{i=0}^{i=N} \left(e^{(i)}\right)^2$

The error is a function of the parameter set $\vec{w}$. The question is: how do we adjust $\vec{w}$ so that theerror e_i^2 decreases. We know a function's value changes most when we move along the direction of of the gradient of the parameters. Hence, we adjust the parameters $\vec{w}$, b as

$$\begin{bmatrix} \vec{w} \\ b \end{bmatrix} = \begin{bmatrix} \vec{w} \\ b \end{bmatrix} - \mu \nabla_{\vec{w},b} L(\vec{w}, b)$$

Each adjustment reduces the error. Starting from a random set of parameter values doing this "sufficiently" large number of times yields the desired model.

A simple and popular model ϕ is the linear function (predicted value is dot product between input and parameters plus bias): $\tilde{y}_i = \phi(\vec{x}_i, \vec{w}, b) = \vec{w}^T \vec{x} + b = \Sigma_j w_j x_j + b$. In the example below, this is the model architecture used.

Our initial implementation will simply mimic this formula. For more complicated models ϕ (with millions of parameters and non-linearities) we cannot obtain closed form gradients like this. The next example, based on PyTorch, relies on PyTorch's autograd (automatic gradient computation) which does not require the closed form gradient. //

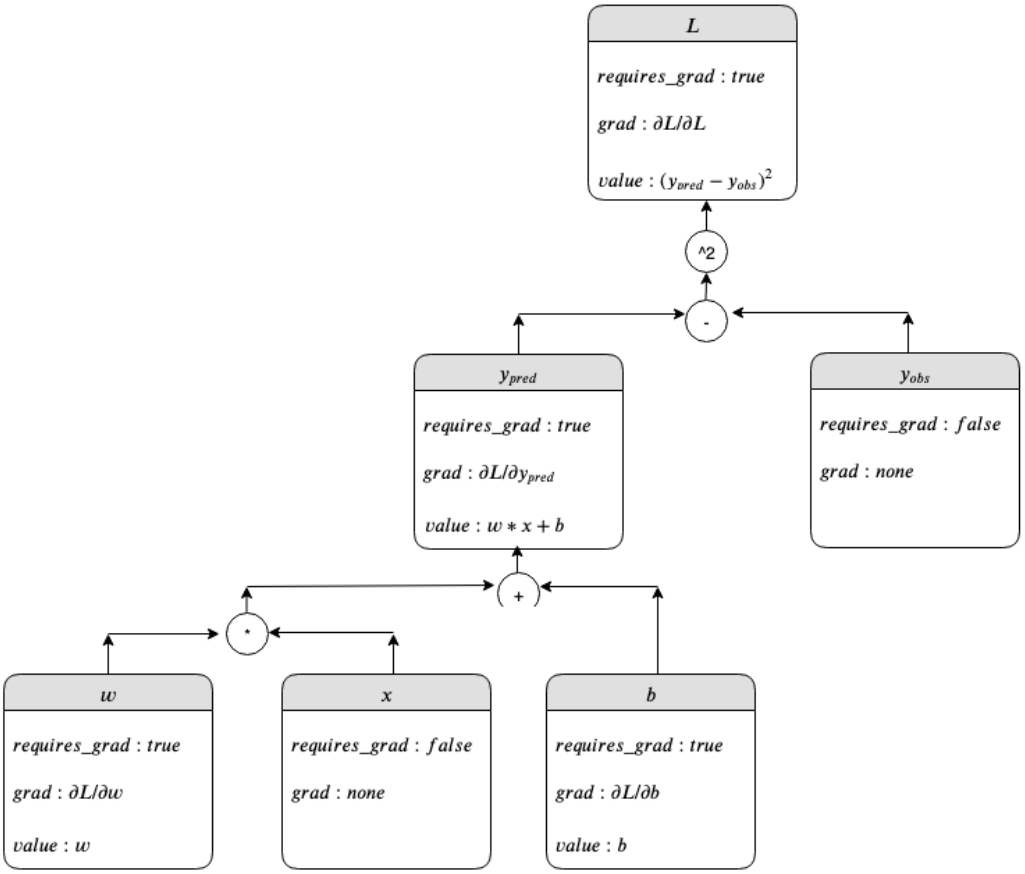


Figure 3.12: Auto grad Analysis

Listing 3.2: Linear modeling with PyTorch

```
def update_parameters(params, learning_rate):
  with torch.no_grad():
    for i, p in enumerate(params):
      params[i] = p - learning_rate * p.grad

    for i in range(len(params)):
      params[i].requires_grad = True


x = 10 * torch.rand(N, 1)
y = 1.5 * x + 2.73
y_obs = y + (0.5 * torch.rand(N, 1) - 0.25)

w = torch.rand(1, requires_grad=True)
b = torch.rand(1, requires_grad=True)
params = [b, w]

for step in range(num_steps):
  y_pred = params[0] + params[1] * x

  mean_squared_error = torch.mean((y_pred - y_obs) ** 2)
  mean_squared_error.backward()
  update_parameters(params, learning_rate)

print("True function: y = 1.5*x + 2.73")
print("Learnt function: y_pred = {}*x + {}"\
      .format(params[1].data.numpy()[0], params[0].data.numpy()[0]))
```

Update parameters, i.e., adjust weight, bias along gradient of error

Don't track gradients during parameter update

Restore gradient tracking

Generate random training input

Generate training output: apply a simple known function to input.

Then add noise.

Lets see if our learnt function matches the known underlying function

Our model - initialized with arbitrary parameter values

model error is (squared) difference between observed and predicted values

backpropagate: compute partial derivatives of the error with respect to each variable and store them in the "grad" field within the variable

update parameters using those partial derivatives

Output:

```
True function : y = 1.5* x + 2.73
Learnt function : y_pred = 1.50910639763* x + 2.66267037392
```

Listing 3.1: Numpy linear model (closed form formula for gradients needed).

```
x = 10 * np.random.randn(N)

y = 1.5 * x + 2.73
y_obs = y + (0.5 * np.random.randn(N))

for step in range(num_steps):
    y_pred = w*x + b
    mean_squared_error = np.mean((y_pred - y_obs) ** 2)

    w_grad = np.mean(2 * ((y_pred - y_obs)* x))
    b_grad = np.mean(2 * (y_pred - y_obs))
    w = w - learning_rate * w_grad
    b = b - learning_rate * b_grad

print("True function: y = 1.5*x + 2.73")
print("Learnt function: y_pred = {}*x + {}".format(w[0], b[0]))
```

generate random input values

generate output values by applying a simple known function. to input. Then add noise. Lets see if our learnt function matches the known underlying function

our model - initialized with arbitrary parameter values

model error is (squared) difference between observed and predicted values

Calculate gradient of error using calculus. Possible only with such simple models.

Adjust weight, bias along gradient of error

Output:

```
True function : y = 1.5*x + 2.73
Learnt function : y_pred = 1.50606334527* x + 2.71214524274
```

AUTOGRAD: PYTORCH AUTOMATIC GRADIENT COMPUTATION

In the numpy code above, we computed the gradient using calculus for this specific model architecture.

This approach does not scale to more complex models with millions of weights and perhaps non-linear complex functions. For scalability, we can use PyTorch, where gradients are computed via automatic differentiation. The user of the libraries need not worry about how to compute the gradients - they just construct the model function. Once the function is specified, PyTorch figures out how to compute its gradient through a technology called *autograd*.

Autograd Autograd is the technology in pytorch for automatic gradient computation. Here is how it is used. One explicitly tells pytorch to track gradients with respect to a variable by setting **requires_grad = True** when creating the variable. Pytorch remembers a computation graph which gets updated everytime we create an expression using tracked variables. Below is an example of computation graph

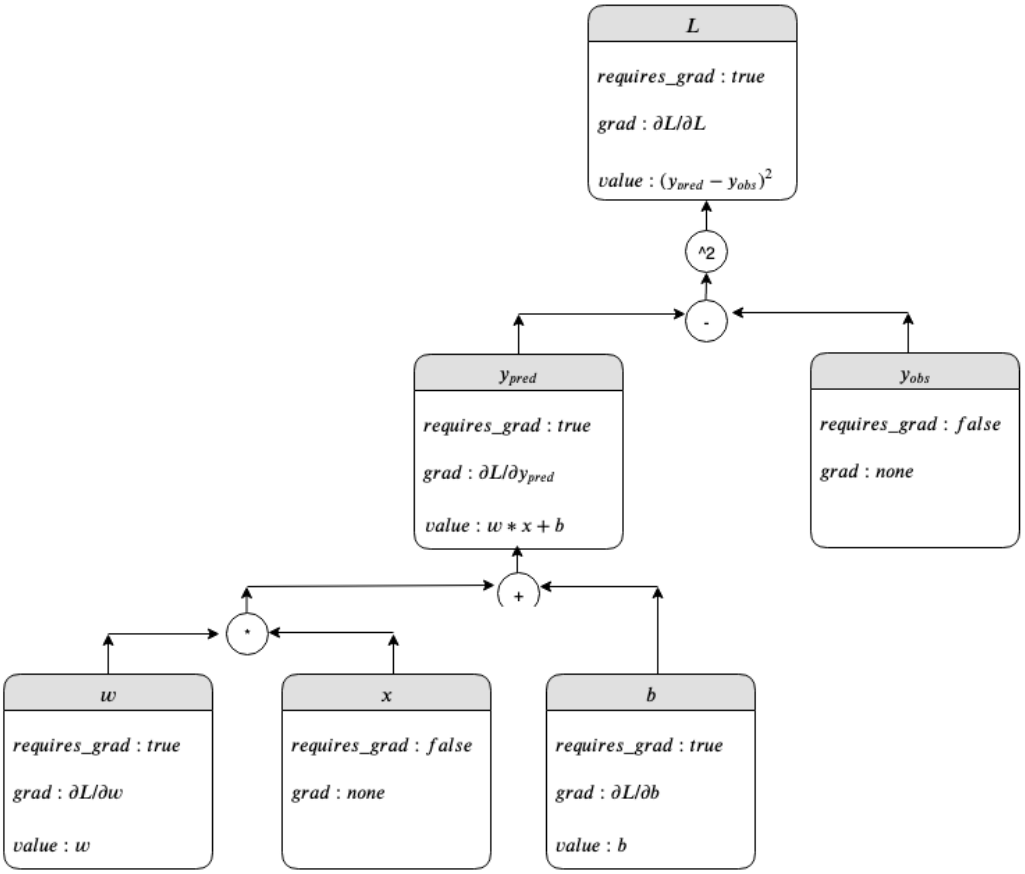


Figure 3.12: Auto grad Analysis

Listing 3.2: Linear modeling with PyTorch

```
def update_parameters(params, learning_rate):
  with torch.no_grad():
    for i, p in enumerate(params):
      params[i] = p - learning_rate * p.grad

    for i in range(len(params)):
      params[i].requires_grad = True


x = 10 * torch.rand(N, 1)
y = 1.5 * x + 2.73
y_obs = y + (0.5 * torch.rand(N, 1) - 0.25)

w = torch.rand(1, requires_grad=True)
b = torch.rand(1, requires_grad=True)
params = [b, w]

for step in range(num_steps):
  y_pred = params[0] + params[1] * x

  mean_squared_error = torch.mean((y_pred - y_obs) ** 2)
  mean_squared_error.backward()
  update_parameters(params, learning_rate)

print("True function: y = 1.5*x + 2.73")
print("Learnt function: y_pred = {}*x + {}"\
      .format(params[1].data.numpy()[0], params[0].data.numpy()[0]))
```

Update parameters, i.e., adjust weight, bias along gradient of error

Don't track gradients during parameter update

Restore gradient tracking

Generate random training input

Generate training output: apply a simple known function to input.

Then add noise.
Lets see if our learnt function matches the known underlying function

Our model - initialized with arbitrary parameter values

model error is (squared) difference between observed and predicted values

backpropagate: compute partial derivatives of the error with respect to each variable and store them in the "grad" field within the variable

update parameters using those partial derivatives

Output:

```
1 True function : y = 1.5* x + 2.73
2 Learnt function : y_pred = 1.50910639763* x + 2.66267037392
```

3.4.2 Non-linear Models in PyTorch

In code listing captioned "Numpy Linear Model" and "Linear modeling with PyTorch" we fit a linear model to a data distribution that we know to be linear. From outputs we can see that those models converged to a pretty good approximation of the underlying output function - in the Numpy as well as PyTorch case. We also see that graphically from Fig 3.13. But what happens if the underlying output function is non-linear?

For real world problems, we will not know the underlying true output function. But here, for the sake of gaining insight, we will continue to generate known output functions which we will perturb with noise to make it slightly realistic. We will first try to use a linear model on the non-linear data distribution (code listing "Linear Approximation of non linear data"). As expected (and demonstrated via the output as well as Fig 3.14) this model does not do well. This is because we are using an inadequate model architecture. Further training will not help.

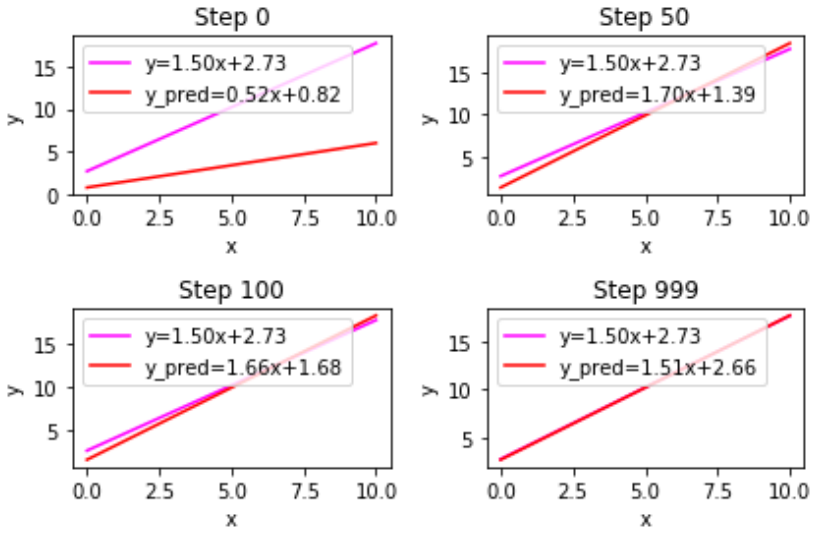


Figure 3.13: Linear approximation of linear data: We see that towards step 1000 the model has more or less converged to true underlying function

Then we will try a non-linear model (code listing "Non linear modeling with PyTorch"). As expected (and demonstrated via the output as well as Fig 3.15) the non linear model does well. In real life problems, we usually assume non-linearity and choose a model architecture accordingly.

Complete code for this section, fully executable via Jupyter-notebook, can be found at nbviewer.jupyter.org/../ch3/3.4.3-gradients-catbrain-numpy-pytorch.ipynb.

Listing 3.3: Linear approximation of non linear data.

```
x = 10 * torch.rand(N, 1)

y = x**2 - x + 2.0
y_obs = y + (0.5 * torch.rand(N, 1) - 0.25)

w = torch.rand(1, requires_grad=True)
b = torch.rand(1, requires_grad=True)
params = [b, w]
for step in range(num_steps):
  y_pred = params[0] + params[1] * x
  mean_squared_error = torch.mean((y_pred - y_obs) ** 2)
  mean_squared_error.backward()
  update_parameters(params, learning_rate)

print("True function: y = 1.5*x + 2.73")
print("Learnt function: y_pred = {}*x + {}"\
      .format(params[1].data.numpy()[0], params[0].data.numpy()[0]))
```

Generate random input training data

Generate training output: apply a known non-linear function to input. Then perturb with noise.

Train a linear model as in previous listing

Output:

```
True function : y=x^2 -x + 2
Learnt function : y_pred = 8.79633331299* x + -13.4027605057
```

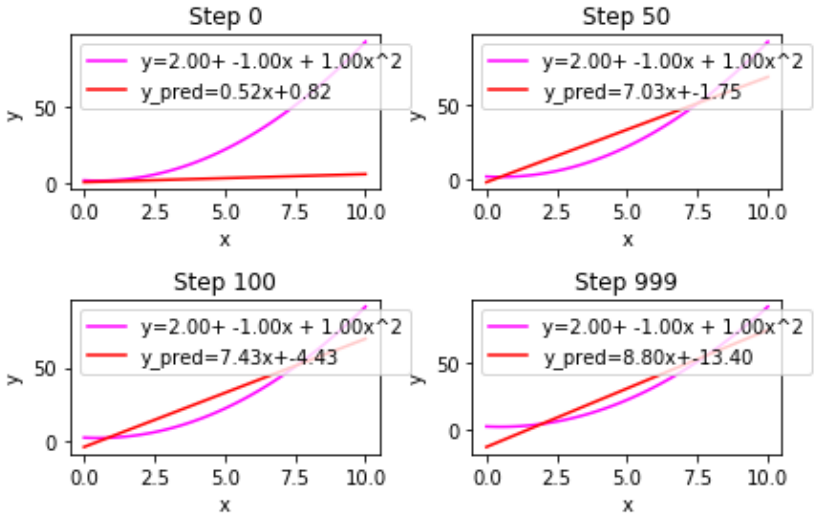


Figure 3.14: Linear approximation of non linear data: Clearly the model is not converging to anything close to the desired/true function. Our model architecture is inadequate

Listing 3.4: Non linear modeling with PyTorch

```
params = [w0, w1, w2]
for step in range(num_steps):
    y_pred = params[0] + params[1] * x + params[2] * (x**2)
    mean_squared_error = torch.mean((y_pred -y_obs) ** 2)
    mean_squared_error.backward()
    update_parameters(params, learning_rate)

print("True function: y= 2 - x + x^2")
print("Learnt function: y_pred = {} + {}*x + {}*x^2"\
      .format(params[0].data.numpy()[0],
              params[1].data.numpy()[0],
              params[2].data.numpy()[0]))
```

Output:

```
True function : y= 2 - x + x^2
Learnt function : y_pred = 1.87116754055 + -0.953767299652* x + 0.996278882027* x^2
```

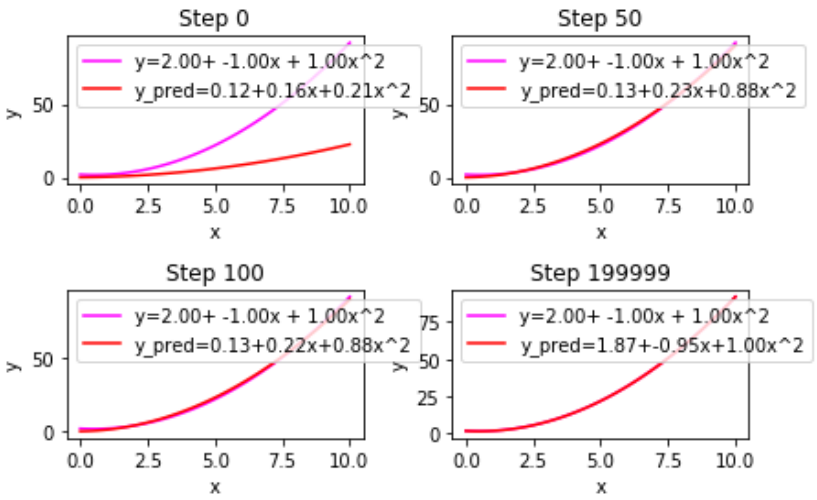


Figure 3.15: Non linear model: If we use a non linear model instead, we see that the model has more or less converged to true underlying function

3.4.3 A Linear Model for the cat-brain in PyTorch

In section 2.12.5 we solved the cat-brain problem directly via pseudo-inverse. Let us now train a PyTorch model over the same dataset. As expected, the model parameters will be converge to a solution close to that obtained by the pseudo-inverse technique (this being a

simple training dataset) - but in this course, we will demonstrate our first somewhat sophisticated PyTorch model.

Listing 3.5: Our first realistic PyTorch model (solves the cat-brain problem)

```
X = np.array([[0.11, 0.09], ... [0.63, 0.24]])

X = np.column_stack((X, np.ones(15)))

y = np.array([-0.8, ... 0.37])

X = torch.from_numpy(X)
y = torch.from_numpy(y)

class LinearModel(torch.nn.Module):
    def __init__(self, num_features):
        super(LinearModel, self).__init__()
        self.w = torch.nn.Parameter(
            torch.rand(num_features, 1).type('torch.DoubleTensor'))

    def forward(self, X):
        y_pred  = torch.mm(X, self.w)
        return y_pred

model =  LinearModel(num_features=num_unknowns)

loss_fn = torch.nn.MSELoss(reduction='sum')

optimizer = torch.optim.SGD(model.parameters(), lr=1e-2)

for step in range(num_steps):
    y_pred = model(X)
    loss = loss_fn(y_pred, y)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

solution_gd = np.squeeze(model.w.data.numpy())
print("The solution via gradient descent is {}".format(solution_gd))
```

Add a column of all 1s to augment data matrix X.

$X, \vec{y}$ created (see section 2.12.2) as per equation 2.24

It is easy to verify that the solution to equation 2.24 is roughly $w_0 = 1, w_1 = 1, b = -1$.

But the equations are not consistent - no one solution perfectly fits all of them.

We expect the learnt model to be close to $y = x_0 + x_1 - 1$.

Parameter is a type (sub-class) of Torch Tensor suitable for model parameters (weights+bias).

Linear Model: $\vec{y} = X\vec{w}$ (X is augmented and $\vec{w}$ includes bias)

A ready-made class for computing squared error loss

A ready-made class for updating weights using gradient of error

zero out all partial derivatives

Compute partial derivatives via AutoGrad

Update parameters from gradient computed in the backward() step

Output:

```
The solution via gradient descent is [ 1.0766 0.8976 -0.9581]
```

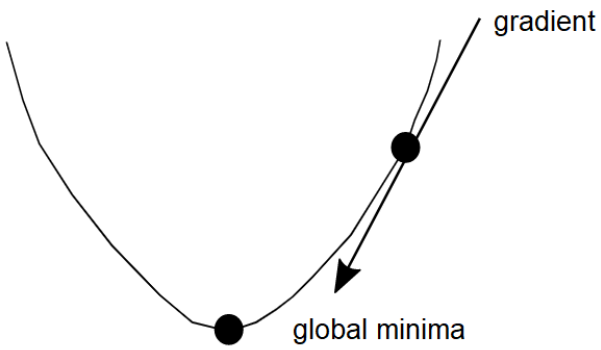


Figure 3.16: A convex function: Only global minima, no local minima. Descending along gradient is guaranteed to reach the global minimum. Friendly error functions are like this

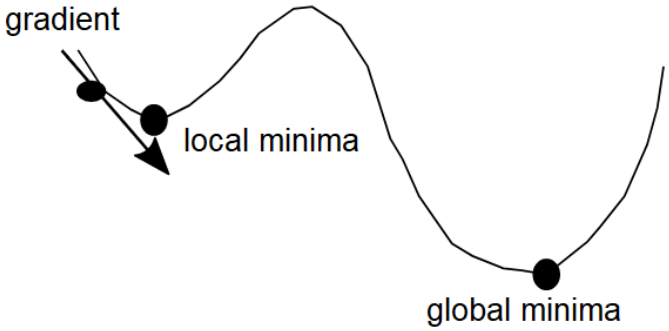


Figure 3.17: A nonconvex function: Local minima exists. Descending along gradient may reach a local minimum and never discover the global minimum. Unfriendly error functions are like this

3.5 Convex, Non-convex functions; Global and Local Minima

A convex surface (see Figure 3.16) has a single optimum (maximum/minimum) - the global one. Wherever we are on such a surface, if we keep taking downward steps, eventually we will reach the global minimum. On the other hand, a non-convex surface looks something like Figure 3.17. Here, we might get stuck in a local minimum. For instance, see Figure 3.17. If we start at the point marked with the arrowed line indicating gradient and move downward following the gradient, we will arrive at a local minimum. At the minimum, gradient is zero and we will never move out of that point.

There was a time when researchers spent a lot of effort trying to avoid local minima. Special techniques (such as simulated annealing) were developed to avoid them. However, neural networks typically do not do anything special to deal with local minima and non-convex functions. Quite often the local minimum is good enough. Or one retrains starting from a different random point luckily escapes the local minimum.

After training, we have an estimated output function $f(\vec{x})$ with a set of weights that minimizes the error over the training data set. Its time to launch the classifier in life. It takes an arbitrary input $\vec{x}$ (not necessarily seen before) and evaluates $f(\vec{x})$ and emits a decision. This latter process is called "*inferencing*" as opposed to "training".

It is important to realize that the *error has been minimized only on the training data set*. This will yield a good classifier in the real world if and only if the training data set was a good representation of the data one encounters during inferencing. It is absolutely imperative that the training data set is *sufficiently large and sufficiently varied* for the classifier to do well in practice.

3.6 Multi-dimensional Taylor series and Hessian Matrix

3.6.1 1D Taylor Series recap

Suppose we are trying to describe the curve $f(x)$ in the neighborhood of a particular point *x*. If we stay infinitesimally close to x then, as described in subsection 3.3, we can approximate the curve with a straight line and say

$$f(x+\delta x) = f(x) + \frac{df}{dx}\delta x$$

But in the general case, if we are describing a continuous (smooth) function in the neighborhood of a specific point, we use Taylor series. Taylor series allows us to describe a function in the neighborhood of a specific point in terms of the value of the function and its derivatives at that point. It is relatively simple in **1D**.

$$f(x+\delta x) = f(x) + \frac{(\delta x)}{1!}\frac{df}{dx} + \frac{(\delta x)^2}{2!}\frac{d^2f}{dx^2} + \cdots \tag{3.7}$$

Note that the terms become progressively smaller (since they involve higher and higher powers of a small number δx). Hence although the series goes on till infinity, in practice, one entails negligible loss in accuracy by dropping higher order terms. One often uses the first order approximation or at most second order.

A handy example of Taylor series is the expansion of the exponential function e^x near $x = 0$.

$$e^t = e^{0+t} = 1 + t + \frac{t^2}{2!} + \frac{t^3}{3!}\cdots$$

where we have used the fact that $\frac{d^n}{dx^n}\left(e^x\right)\big|_{x=0} = e^x\big|_{x=0} = 1$ for all n.

3.6.2 Multi-dimensional Taylor series and Hessian Matrix

In equation 3.7 we expressed a function of one variable in a small neighborhood around a point in terms of the derivatives. Can we do a similar thing in higher dimensions. The answer is yes. We simply needs to replace the first derivative with gradient. We need to replace the

second derivative with its multi-dimensional counterpart - the Hessian matrix. The multi-dimensional Taylor series is as follows

$$f\left(\vec{x}+\vec{\delta x}\right)=f\left(\vec{x}\right)+\frac{1}{1!}\left(\vec{\delta x}\right)^{T}\nabla f\left(\vec{x}\right)+\frac{1}{2!}\left(\vec{\delta x}\right)^{T}H\left(\vec{x}\right)\left(\vec{\delta x}\right)+\cdots \tag{3.8}$$

where $H(\vec{x})$, called the *Hessian matrix* defined as

$$H\left(\vec{x}\right)=\begin{bmatrix}\frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n}\\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n}\\ \vdots & \vdots & \vdots & \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2}\end{bmatrix} \tag{3.9}$$

Note that the Hessian matrix is symmetric since $\frac{\partial^2 f}{\partial x_i \partial x_j}=\frac{\partial^2 f}{\partial x_j \partial x_i}$. It should be noted that Taylor expansion assumes that the function is continuous in the neighborhood.

3.7 Convex sets and functions

In section 3.5 we briefly encountered convex functions and how convexity tells us whether the function has local minima or not. In this section, we will study convex functions in more detail. In particular, we will learn how to tell whether a given function is convex. We will also study some important properties of convex functions which will come in handy later, for instance when we study Jensen's Inequality in probability and statistics, sec **??**.

We will mostly illustrate the ideas in 2D space, they can be easily extended to higher dimensions.

CONVEX SETS

Informally speaking, a set of points is said to be convex if and only if the straight line joining any pair of points in the set lies entirely within the set. Consider the green regions in Fig. 3.18. If we take any pair of of points in the green region and join them with a straight line, all points on that line will also be in the green region. This is illustrated by the points A and B in Fig. 3.18. The complete set of points in any such region together will constitute a convex set.

Conversely, a set of points is non-convex if it contains at least one pair of points whose joining line contains a point not belonging to the set. For instance, see the red region in Fig. 3.18. We show a pair of points A and B whose joining line passes through points not belonging to the red region.

The boundary of a convex set is always a convex curve.

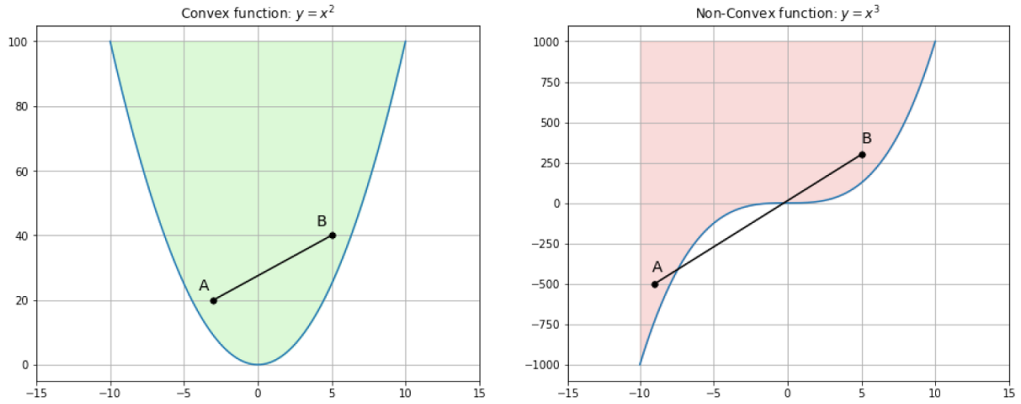


Figure 3.18: Convex and non convex sets. The points in the green region form a convex set. The line joining any pair of points in green region lies entirely in the green region, e.g., AB in left hand figure. The points in the red region form a non convex set. For instance, the line joining points AB in right hand figure passes through non red region even though both end points belong to red region.

CONVEX CURVES AND SURFACE

Consider a function $g(x)$. Let us pick any two points on the curve $y = g(x)$, $A \equiv (x_1, y_1 = g(x_1))$ and $B \equiv (x_2, y_2 = g(x_2))$. Now consider the line segment *L* joining *A* and *B*. From section 2.8.1, equation 2.14 and Fig 2.8 we know that all points *C* on *L* can be expressed as a weighted average ofthe coordinates of *A* and *B* with the sum of weights being 1. Thus, $C \equiv (\alpha_1 x_1 + \alpha_2 x_2, \alpha_1 y_1 + \alpha_2 y_2)$ where $\alpha_1 + \alpha_2 = 1$. Compare C with its corresponding point D on the curve, which has the same X coordinate. $D \equiv (\alpha_1 x_1 + \alpha_2 x_2, g(\alpha_1 x_1 + \alpha_2 x_2))$.

If and only if $g(x)$ is a convex function, C will always be above D or

$$\alpha_1 y_1 + \alpha_2 y_2 = \alpha_1 g(x_1) + \alpha_2 g(x_2) \geq g(\alpha_1 x_1 + \alpha_2 x_2)$$

Viewed in another way, if we drop a perpendicular to the X-axis from any point on a line joining a pair of points on the curve, that perpendicular will cut the curve at a lower point (i.e., smaller in Y-coordinate).

This is illustrated in Fig. 3.19 left hand side with the function$g(x) = x_2$ (known to be convex) and $A \equiv (-3, 9)$ and $B \equiv (5, 25)$, $\alpha_1 = 0.3$, $\alpha_2 = 0.7$. It can be seen that the weighted average point C on the line lies above the corresponding point on the curve D. The right hand side illustrates the non-convex function $g(x) = x_3$, with $A \equiv (-8, -512)$ and $B \equiv (5, 125)$, $\alpha_1 = 0.3$, $\alpha_2 = 0.7$. One weighted average point, viz., C, on the line joining points A and B on the curve, which lies below the point on the curve with same X coordinate, viz., D, is shown.

In fact, we need not restrict ourselves to two points only. We can take weighted average of an arbitrary number of points on the curve, with the weights summing to one. The point corresponding to the weighted average will lie above the curve (i.e., above the point on curve with same X coordinate).

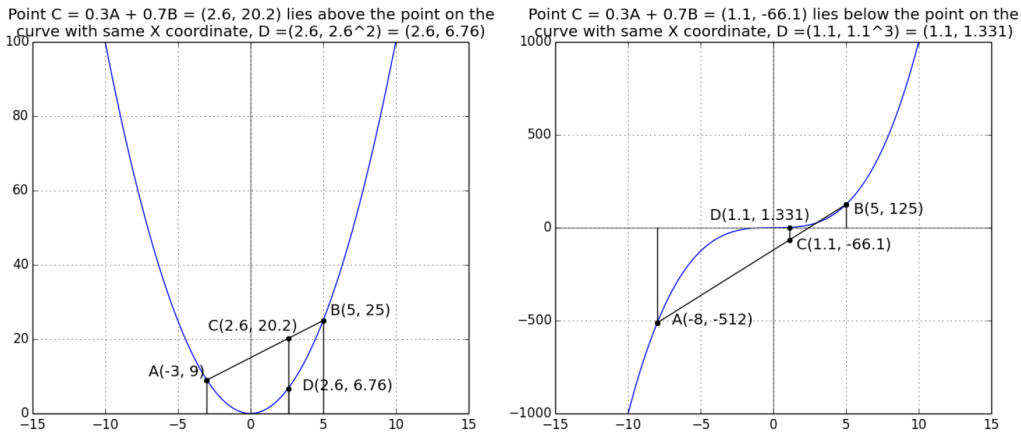


Figure 3.19: Convex and non convex curves. A and B are a pair of points on the curve. $C = 0.3A + 0.7B$ is a weighted average of the coordinates of A and B, with weights summing to 1. C lies on the line joining joining A and B. The left hand figure shows a convex curve. Here C lies above the corresponding curve point D. The right hand figure shows a non-convex curve. Here C lies below the corresponding curve point D.

The idea extends to higher dimensions too.

Definition 1

In general, a multi-dimensional function $g(\vec{x})$ is convex if and only if

- Given an arbitrary set of points on the function surface (curve if the function is **1D**), $(\vec{x}_1, g(\vec{x}_1)), (\vec{x}_2, g(\vec{x}_2)), \cdots, (\vec{x}_n, g(\vec{x}_n))$
- Given an arbitrary set of n weights $\alpha_1, \alpha_2, \cdots, \alpha_n$ that sum to 1, i.e., $S_i^n{}_{=1}\ \alpha_i = 1$
- *The weighted sum of function outputs exceeds or equals the function output on the weighted sums*

$$\Sigma_{i=1}^{n} \alpha_i g\left(\vec{x}_i\right) \geq g\left(\Sigma_{i=1}^{n} \alpha_i \vec{x}_i\right) \tag{3.10}$$

A little thought will reveal that definition 1 implies that convex curves always curls upwards and/or rightwards, everywhere. This leads to another equivalent definition of convexity.

Definition 2

- A 1D function $g(x)$ is convex if and only if its curvature is positive everywhere, i.e.,

$$\frac{d^2g}{dx^2} \geq 0 \quad \forall x \tag{3.11}$$

- A multidimensional function $g(\vec{x})$ is convex if and only if its Hessian matrix (see section 3.6.2, equation 3.9) is positive semi-definite (i.e., all the eigenvalues of the Hessian matrix are greater than or equal to zero). This is just the multi-dimension analog of equation 3.11.

One subtle point to note is that, if the second derivative is negative everywhere or the Hessian is negative semi-definite, the curve or surface is said to be *concave*. This is different from non-convex curves where the second derivative is positive in some places and negative in some other places. The negative of a concave function is a convex function. But the negative of a non-convex function is again non-convex. A function that curves upwards everywhere is always going to lie above its tangent. This leads to another equivalent definition of a convex function

Definition 3

- A function $g(x)$ is convex if and only if all the points on the curve $S \equiv (x, g(x))$ lie above the tangent line T at any point A on S, with S touching T only at A.
- A function $g(\vec{x})$ is convex if and only if all the points on the surface $S \equiv (\vec{x}, g(\vec{x}))$ lie above the tangent plane T at any point A on S, with S touching T only at A.

This is illustrated in Fig. 3.20.

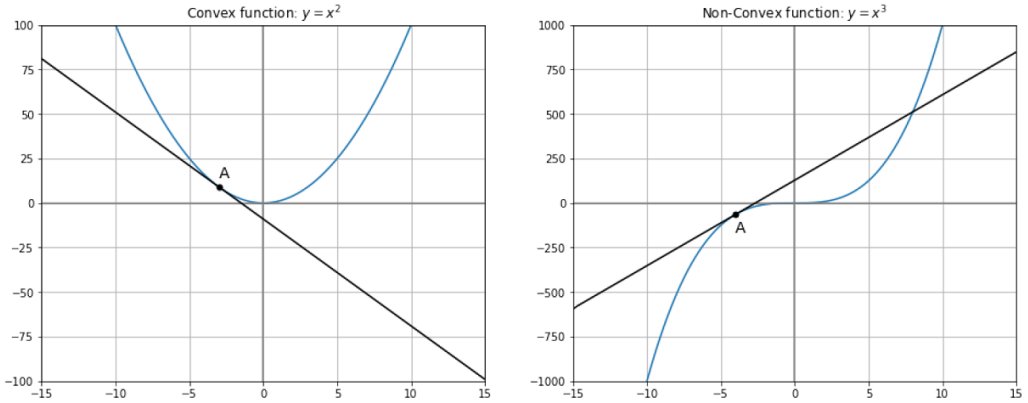


Figure 3.20: Convex and non convex curves. The left hand figure shows a convex curve. If we draw a tangent line at any point A on the curve, the entire curve will be *above* the tangent line, touching it only at A. The right hand

side shows a non-convex function where part of the curve lies above the tangent and another part below.

CONVEXITY AND TAYLOR SERIES

In section 3.6.1, equation 3.7 we saw the one dimensional Taylor expansion for a function in the neighborhood of a point x. If we retain terms in Taylor expansion only upto the first derivative and ignore all subsequent terms, that is equivalent to approximating the function at x with its tangent at x (see fig. 3.20). This is the linear approximation to the curve. If we retain one more term, i.e., upto second derivative, we get the quadratic approximation to the curve. Now, if the second derivative of the function is always positive (as in convex functions), the quadratic approximation to the function will always be greater than or equal to the linear approximation. In other words, locally, the curve will curve in a way that it lies above the tangent. This connects second derivative definition with the above tangent definition of convexity.

SOME EXAMPLES OF CONVEX FUNCTION

The function $g(x) = x^2$ is convex. The easiest way to verify this is to compute

$d^2g/dx^2 = d(^2x)/dx = 2$, which is positive always.

In fact, any even power of $x.$, $g(x) = x^{2n}$ for an integer n, like x^4, x^6 etc is convex.

$g(x) = e^x$ is also convex. It can be easily verified by taking its second derivative.

$g(x) = \log x$ is concave. Hence, $g(x) = -\log x$ is convex.

Multiplication by a positive scalar preserves convexity. Also sum of convex functions is itself a convex function.

3.8 Chapter Summary

In this chapter we studied machine learning training in more detail.

- We gained more insights into geometrical view of machine learning. In particular, we learnt how machine learning classifiers boil down to separating clusters of points in highdimensional spaces.
- We learnt about various separating surfaces in high-dimensions and the role played by the sign of the separating surface.
- We learnt about gradients of multi-dimensional functions and how they indicate the direction of maximal change in function value.
- We learnt how machine learning training quickly progresses towards the minimum loss by moving along gradient of the loss function.
- We learnt about *convex* and *non-convex* functions and *local* and *global minima*. In a convex function, gradient based descent is guaranteed to converge to the global minimum while in a non-convex function, such approaches may get stuck in local minima.
- We learnt about multi-dimensional Taylor series to create local approximations to a smooth function. We learnt about the Hessian matrix which is needed to make higher order approximations.

4

Linear Algbraic Tools in Machine Learning and Data Science

As mentioned earlier, finding patterns in large volumes of high dimensional data is the name of the game in machine learning and data science. The data often appears in the form of large matrices (a toy example of this is shown in section 2.3 and also in equation 2.1). The rows of the data matrix would represent feature vectors for individual input instances. The number of columns would match the size of the feature vector. The number of rows would match the number of observed input instances. The geometrical representation of such an input matrix would be a set of points (the number of points matches the number of rows in the matrix). The number of dimensions of the space would match the number of columns in the data matrix). The distribution of these points is usually not uniformly random - meaning these points are not spread all over the space. Rather they will occupy a rather small sub-region of that space. Such a skewed distribution of points is shown in Figure 4.2 as a toy instance. Instead of being distributed all over the 2D space, the points are lying within a long and very narrow elliptic shape.

For instance, consider the problem of determining similarity between documents. This is an important problem machine learning. Its use is in document search and retrieval where given a query document, the system needs to retrieve - in ranked order - documents from archive that match the query document. Google solves this problem for instance. Each document is represented by a document descriptor vector. It is a very very long vector - its length matches the size of the vocabulary of the documents in the system - with a fixed position for every word in the vocabulary, ignoring perhaps prepositions and conjunctions. For every document, the descriptor contains the frequency (number of occurrences) of every word in the vocabulary. If the word does not occur, we put a zero there. We will store one descriptor vector for every document in the archive. Such machine learning systems indeed exist. However, we do not explicitly store the descriptor vectors in their entirety. Instead, we store them in a *logically equivalent fashion*, meaning we store the frequency along with the actual word in the descriptor vector - *not explicitly storing the words that did not occur*.

Certain words often occur together ("Michael" and "Jackson", "driver-less" and "cars"). Consequently, the points corresponding to the descriptor vectors will have a very skewed shape, reminiscent of Figure 4.2.

If the sub-region of the space occupied by the data points has very little variation (spread) along certain directions, we can ignore those directions when measuring similarity. We ignore the small variance directions by projecting the data on the high variance dimensions only. This leads to a much simplified representation of the data and the process is called dimensionality reduction. Not only does this lead to a simplified representation, it often leads to elimination of noisy signals, because the small variations are often caused by noise.

The above ideas form the basis of the technique of Principal Component Analysis (PCA). It is one of the most important tools in the repertoire of a data scientist and machine learning practitioner. These ideas also underly the technique of Latent Semantic Analysis (LSA) for document retrieval - a fundamental approach for solving Natural Language Processing (NLP) problems in machine learning. This chapter is dedicated to studying a set of methods leading up to the PCA and LSA. We study a basic document retrieval system along with python code in the end.

This chapter deals in some intricate mathematics. The reader will be well advised to persevere through all of that, including theorem proofs. Getting an intuitive understanding the proofs will lead to significantly better insights.

Fully functional code for chapter 4, runnable via Jupyter-Notebook is available at our public github repository at https://nbviewer.jupyter.org/github/krishnonwork/mathematicalmethods-in-deep-learning-ipython/tree/master/python/ch4/.

4.1 Quadratic Forms and their Minimization

Given a square symmetric matrix *A*, the scalar quantity $Q = \vec{x}TA\vec{x}$ is called a quadratic form. It is seen in various situations in machine learning.

For instance, recall the equation of a circle we learnt in high school.

$$(x_0 - \alpha_0)^2 + (x_1 - \alpha_1)^2 = r^2$$

where the center of the circle is (α_0, α_1) and radius is *r*. This equation can be rewritten as

$$\begin{bmatrix}(x_0 - \alpha_0) & (x_1 - \alpha_1)\end{bmatrix} \begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix} \begin{bmatrix}(x_0 - \alpha_0)\\ (x_1 - \alpha_1)\end{bmatrix} = r^2$$

If we denote the position vector $\begin{bmatrix}x_0\\ x_1\end{bmatrix}$ as $\vec{x}$ and the center of the circle $\begin{bmatrix}\alpha_0\\ \alpha_1\end{bmatrix}$ as $\vec{\alpha}$ the above equation can be written compactly as

$$(\vec{x} - \vec{\alpha})^T \mathbf{I} (\vec{x} - \vec{\alpha}) = r^2$$

Stated in this matrix format, it is no longer restricted to 2D. It is in fact a hyper-sphere. The LHS is a quadratic form (transpose of a vector multiplied with a symmetric matrix multiplied with the same vector).

Consider, now, the equation of ellipse we learnt in high school.

$$\frac{(x_0 - \alpha_0)^2}{\beta_0^2} + \frac{(x_1 - \alpha_1)^2}{\beta_1^2} = 1$$

The reader can verify that this can be written compactly in matrix form as

$$\begin{bmatrix}(x_0 - \alpha_0) & (x_1 - \alpha_1)\end{bmatrix} \begin{bmatrix}\frac{1}{\beta_0^2} & 0 \\ 0 & \frac{1}{\beta_1^2}\end{bmatrix} \begin{bmatrix}(x_0 - \alpha_0) \\ (x_1 - \alpha_1)\end{bmatrix} = 1$$

or, equivalently

$$(\vec{x} - \vec{\alpha})^T A\, (\vec{x} - \vec{\alpha}) = 1 \tag{4.1}$$

where

$$A = \begin{bmatrix}\frac{1}{\beta_0^2} & 0 \\ 0 & \frac{1}{\beta_1^2}\end{bmatrix}.$$

Once again, the matrix representation is dimension independent. In other words, equation 4.1 represents a hyper-ellipse. It should be noted that if the ellipse axes are aligned with the coordinate axes, matrix A is diagonal. If we rotate the coordinate system, each position vector will be rotated by an orthogonal matrix R. Equation 4.1 gets transformed as follows (we have used the rules for transposing product of matrices from equation 2.11)

$$(R(\vec{x} - \vec{\alpha}))^T A\, (R(\vec{x} - \vec{\alpha})) = 1$$
$$((\vec{x} - \vec{\alpha})^T (R^T A R)\, (\vec{x} - \vec{\alpha}) = 1$$

Replacing *RTAR* with A, we get the same equation as equation 4.1, only A is no longer a diagonal matrix.

For a generic ellipsoid with arbitrary axes, A will have non-zero off diagonal terms, but it will still be symmetric. Thus, the multi-dimensional hyper-ellipse is represented by a quadratic form. The hyper-sphere is a special case of this.

The quadratic form is also found in the second term of the multi-dimensional Taylor expansion shown in equation 3.8, $\frac{1}{2!}\left(\vec{\delta x}\right)^T H\,(\vec{x})\left(\vec{\delta x}\right)$ is a quadratic form in the Hessian matrix.

Another huge application of quadratic forms is Principal Component Analysis which is so important that we have a whole section devoted to it (section 4.3).

An important question is what choice of $\vec{x}$ maximizes or minimizes the quadratic form. Because the quadratic form is part of the multi-dimension Taylor series, this question arises when we are trying to determine what direction to move in so as to approach the minimum with maximal speed during minimization of loss $L\,(\vec{x})$. It will also arise in context of Principal Component Analysis which we will study later.

If $\vec{x}$ is an arbitrary length vector, we can make *Q* arbitrarily big or small by simply changing the length of $\vec{x}$. Consequently, optimizing *Q* with arbitrary length $\vec{x}$ is not a very interesting problem. Rather, we want to know which *direction* of $\vec{x}$ optimizes *Q*. Hence, for the rest of this section, we will discuss quadratic forms with unit vectors $Q = \hat{x}^T A \hat{x}$ (recall that $\hat{x}$ denotes an unit length vector satisfying $\hat{x}^T \hat{x} = \| \hat{x} \|^2 = 1$). Equivalently, one can use a different flavor, $Q = \frac{\vec{x}^T A \vec{x}}{\vec{x}^T \vec{x}}$. We will use the former expression here. We are essentially searching over all possible directions $\hat{x}$, examining which direction optimizes $Q = \hat{x}^T A \hat{x}$.

Using matrix diagonalization (section 2.15)

$$Q = \hat{x}^T A \hat{x} = \hat{x}^T S \Lambda S^T \hat{x}$$

where $S = [\vec{e}_1 \ \vec{e}_2 \cdots \vec{e}_n]$ is the matrix with eigenvectors of A as its columns and Λ is a diagonal matrix with the eigenvalues of A on the diagonal and 0 everywhere else. Substituting

$$\hat{y} = S^T \hat{x}$$

we get

$$\begin{aligned} Q &= \hat{x}^T A \hat{x} = \hat{x}^T S \Lambda S^T \hat{x} \\ &= \hat{y}^T \Lambda y \end{aligned} \tag{4.2}$$

where $\sum_{i=1}^{n} y_i^2 = 1$.

It should be noted that since A is symmetric, its eigenvectors are orthogonal. This implies *S* is an orthogonal matrix. Consequently, $S^T S = S S^T = \mathbf{I}$. Recall from section 2.14 that for orthogonal matrix *S* the transformation $S^T \hat{x}$ is length preserving. Consequently, $\hat{y}$ is also of unit length. Expanding the right hand side of equation 4.2 we get

$$\begin{aligned} Q &= \begin{bmatrix} y_1 & y_2 & \cdots & y_n \end{bmatrix} \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & \lambda_n \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \\ &= \sum_{i=1}^{n} \lambda_i y_i^2 \end{aligned} \tag{4.3}$$

We can assume that the eigenvalues are sorted in decreasing order of magnitude (because if not we can always renumber them).

Lemma: The quantity $\sum_{i=1}^{n} \lambda_i y_i^2$ where $\sum_{i=1}^{n} y_i^2 = 1$ and $\lambda_1 \geq \lambda_2 \geq \cdots \lambda_n$, attains its maximum value when $y_1 = 1$, $y_2 = \cdots y_n = 0$.
Intuitive Proof:
If possible, let that the maximum value occur at some other value of $\hat{y}$. We are constrained by the fact that $\hat{y}$ is an unit vector, so we must maintain $\sum_{i=1}^{n} y_i^2 = 1$ None of the elements of $\hat{y}$ can exceed 1. Also, if we reduce y_1' from 1 to a smaller value, say $\sqrt{1-\epsilon}$ some other element(s) must go up by an equivalent amount to compensate. Keeping that in mind, let us suppose that the maximum value occurs at $\hat{y}$' which looks like

$$\begin{bmatrix} \sqrt{1-\epsilon} \\ \vdots \\ 0 \\ \vdots \\ \sqrt{\epsilon} \\ \vdots \end{bmatrix}$$

and all other elements are zero. In other words, there is an additional non-zero value, other than the first one, in $\hat{y}$', viz., the j^{th} element is $\sqrt{\epsilon}$. Consequently y'_1 has to decrease from 1 to $\sqrt{1-\epsilon}$, to ensure $\sum_{i=1}^{n} \left(y_i^{'}\right)^2 = 1$.
Let us compare the corresponding two values of the quadratic form

$$Q = \sum_{i=1}^{n} \lambda_i y_i^2$$

$$Q^{'} = \sum_{i=1}^{n} \lambda_i \left(y_i^{'}\right)^2$$

We see

$$Q - Q' = \lambda_1 \epsilon - \lambda_j \epsilon = (\lambda_1 - \lambda_j)\, \epsilon$$

But this quantity is greater than zero since $\lambda_1 > \lambda_j$ - we sorted the λs in decreasing order at the beginning. Hence, $Q - Q' > 0$ implying $Q > Q'$. This contradicts the assumption that $\hat{y}$' is a maximum. This means, to maximize the right hand side of equation 4.3, we must have 1 as the first element (corresponding to the largest eigenvalue) of the unit vector $\hat{y}$ and zeros everywhere else. Anything else violates the condition that the corresponding quadratic form $Q = \sum_{i=1}^{n} \lambda_i y_i^2$ is a maximum.

$$\hat{y} = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$. The corresponding $\hat{x} = S\hat{y} = \vec{e}_1$ - the eigenvector corresponding to the largest eigenvalue of A.

Thus, *the quadratic form $Q = \hat{x}^T A \hat{x}$ attains its maximum when $\hat{x}$ is along the eigenvector corresponding to the largest eigenvalue of A. The corresponding maximum Q is equal to the largest eigenvalue of A. Similarly, the minimum of the quadratic form occurs when $\hat{x}$ is along the eigenvector corresponding to the smallest eigenvalue.*

As stated above, many machine learning problems boil down to minimizing a quadratic form. We will study a few of them in later sections.

4.1.1 Symmetric Positive (Semi)definite Matrices

A square symmetric $n \times n$ matrix A is positive semi-definite if and only if

$$\vec{x}^T A \vec{x} \geq \forall \vec{x} \in \mathbb{R}^n$$

In other words, a positive semi-definite matrix yields a non-negative quadratic form with all $n \times 1$ vectors $\vec{x}$.

If we disallow the equality, we get symmetric positive definite matrices. Thus A square symmetric $n \times n$ matrix A is positive definite if and only if

$$\vec{x}^T A \vec{x} > \forall \vec{x} \in \mathbb{R}^n$$

From equation 4.2 and 4.3, if Q is positive or zero, all the λ_is are also positive or zero (since the y^2_is are non-negative). Hence, symmetric positive (semi)definiteness is equivalent to the condition that all eigenvalues of the matrix are greater than (or equal to) zero.

4.2 Spectral and Frobenius Norm of a Matrix

SPECTRAL NORM

In section 2.5.4 we saw that the length (aka magnitude) of a vector $\vec{x}$ is $\|\vec{x}\| = \vec{x}^T \vec{x}$. Is there an equivalent notion of magnitude for a matrix *A*?

Well, a matrix can be viewed as an amplifier of a vector. The matrix A amplifies the vector $\vec{x}$ to $\vec{b} = A\vec{x}$. So we can take the maximum possible value of $\|A\vec{x}\|$ over all possible $\vec{x}$. That is a measure for the magnitude of A. Of course, if we consider arbitrary length vectors, we can make $\vec{b}$ arbitrarily large by simply scaling $\vec{x}$ - for any A. That is uninteresting. Rather, we want to examine which direction of $\vec{x}$ gets amplified most and by how much.

We examine with unit vectors $\hat{x}$: what is the maximum (or minimum) value of $\|A\vec{x}\|$ and what direction $\hat{x}$ materializes it?

The quantity

$$\|A\|_2 = \max_{\hat{x}} \|A\hat{x}\|_2$$

is known as the spectral norm of the matrix A. Note that $A\vec{x}$ is a vector and its $\|A\vec{x}\|_2$ is its length. We will sometimes drop the subscript 2 and just denote the spectral norm as $\|A\|$.

Now, consider the vector $A\hat{x}$. Its magnitude

$$\|A\hat{x}\| = (A\hat{x})^T(A\hat{x}) = \hat{x}^T A^T A\hat{x}$$

This is a quadratic form. From section 4.1 we know it will maximize (minimize) when $\hat{x}$ s aligned with the largest (smallest) eigenvalue of A^TA.

Thus the spectral norm is given by the largest eigenvalue of A^TA.

$$\|A\|_2 = \max_{\hat{x}} \|A\hat{x}\| = \sigma_1 \tag{4.4}$$

where σ_1 is the largest eigenvalue of A^TA. It is also (square of) the largest singular value of A. We will see σ_1 again in section 4.4, when we study SVD.

FROBENIUS NORM

An alternative measure for the "magnitude" of a matrix is the Frobenius norm, defined

$$\|A\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}\|a_{ij}\|^2} \tag{4.5}$$

In other words, it is the root mean square of all the matrix elements.

It can be proved that the Frobenius norm is equal to the root mean square of all the singular values (eigenvalues of A^TA) of the matrix

$$\|A\|_F = \sqrt{\sum_{i=1}^{min(m,n)}\sigma_i^2} \tag{4.6}$$

4.3 Principal Component Analysis

Suppose we have a set of numbers, $X = \{x^{(0)}, x^{(1)}, \cdots, x^{(n)}\}$ We want to get a sense of how tightly packed these points are. In other words, we want to measure the *spread* of these numbers. Figure 4.1 show such a distribution. Note that the points need not be uniformly distributed. In

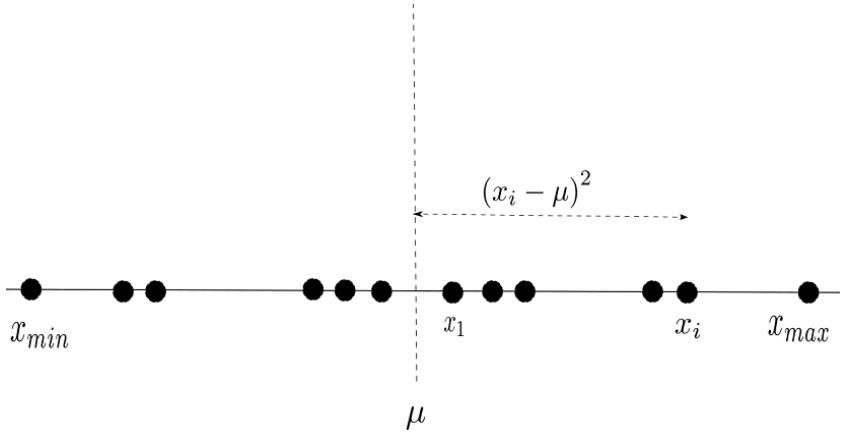


Figure 4.1: A 1*D* distribution of points. Distance between extreme points is *not* a fair representation of the spread of points; the distribution is not uniform and the extreme points are far away from others. Most points are within a more tightly packed region.

particular, the extreme points (x_{max}, x_{min}) maybe far away from most other points (as in Figure 4.1). Thus, x_{max}–x_{min} / n+1 is not a fair representation of the average spread of points here. Most points are within a more tightly packed region. The statistically sensible way to obtain the spread is to first obtain the mean

$$\mu = \frac{1}{n}\sum_{i=0}^{n} x^{(i)}$$

Then obtain the average distance of the numbers from the mean

$$\sigma^2 = \frac{1}{n}\sum_{i=0}^{n} \left(x^{(i)} - \mu\right)^2$$

(one can, if one wishes, take square root and use σ but it is often not necessary to incur that extra computational burden). This scalar quantity, σ, is a good measure of the the mean packing density or spread of points in 1D. The astute reader will recognize that above is nothing but the famous variance formula from statistics. Can we extend the notion to higher dimensional data?

Lets first try 2*D*. As usual, we will name our coordinate axes X_0, X_1 etc, instead of X, Y - to facilitate the extension to multi-dimensions. Individual 2D data point is denoted $\vec{x}^{(i)} = \begin{bmatrix} x_0 \\ x_1 \end{bmatrix}$.

The dataset is $\{\vec{x}^{(0)}, \vec{x}^{(1)}, \cdots \vec{x}^{(n)}\}$.

The mean is straightforward, instead of one we will have 2 means

$$\mu_0 = \frac{1}{n}\sum_{i=0}^{n} x_0^{(i)}$$

$$\mu_1 = \frac{1}{n}\sum_{i=0}^{n} x_1^{(i)}$$

Thus, we now have a mean vector

$$\vec{\mu} = \begin{bmatrix}\mu_0 \\ \mu_1\end{bmatrix} = \frac{1}{n}\sum_{i=0}^{n} \vec{x}^{(i)}$$

Now let us do the variance. The immediate problem we face, there are infinite number of possible directions in the 2D plane. We can measure variance along any of them. It will be different for each choice. We can, of course, find the variance along the X_0 and X_1 axes

$$\sigma_{00}^2 = \frac{1}{n}\sum_{i=0}^{n} \left(x_0^{(i)} - \mu_0\right)^2$$

$$\sigma_{11}^2 = \frac{1}{n}\sum_{i=0}^{n} \left(x_1^{(i)} - \mu_1\right)^2$$

σ_{00} and σ_{11} tells us the variance when the data varies along *only one* of the axes X_0 and X_1 respectively. But, in general, there will be joint variation along both axes. In order to deal with joint variation, let us introduce a cross term

$$\sigma_{01}^2 = \sigma_{10}^2 = \frac{1}{n}\sum_{i=0}^{n} \left(x_0^{(i)} - \mu_0\right)\left(x_1^{(i)} - \mu_1\right) = \sigma_{10}^2$$

The above equations can be written compactly in matrix vector notation.

$$\vec{\mu} = \frac{1}{n}\sum_{i=0}^{n} \vec{x}^{(i)}$$

$$C = \begin{bmatrix}\sigma_{00} & \sigma_{01} \\ \sigma_{10} & \sigma_{11}\end{bmatrix} = \frac{1}{n}\sum_{i=0}^{n} \left(\vec{x}^{(i)} - \vec{\mu}\right)\left(\vec{x}^{(i)} - \vec{\mu}\right)^T$$

(Note: In the expression for *C*, we are *not* taking the dot product of the vectors $(\vec{x}^{(i)} - \vec{\mu})$ and $(\vec{x}^{(i)} - \vec{\mu})$. The dot product would have been $(\vec{x}^{(i)} - \vec{\mu})^T(\vec{x}^{(i)} - \vec{\mu})$ here the second element of the product is transposed as opposed to the first. Consequently, the result is a matrix. Dot product would have yielded a scalar.)

In fact, the last equations are general, meaning they can be extended to any dimension. To be precise, given a set of *n* multi-dimensional data points $X\{\vec{x}^{(0)}, \vec{x}^{(1)}, \vec{x}^{(2)},\cdots,\vec{x}^{(n)}\}$, we can define

$$\vec{\mu} = \frac{1}{n}\sum_{i=0}^{n} \vec{x}^{(i)} \tag{4.7}$$

$$C = \frac{1}{n}\sum_{i=0}^{n}\left(\vec{x}^{(i)} - \vec{\mu}\right)\left(\vec{x}^{(i)} - \vec{\mu}\right)^{T} \tag{4.8}$$

Note how the mean has become a vector (it was a scalar for 1*D* data) and the scalar variance of 1*D*, σ, has now become a matrix *C*. This matrix is called the *covariance matrix*. The $(n+1)$-dimensional mean and covariance matrix can also be defined as

$$\vec{\mu} = \begin{bmatrix} \mu_0 \\ \mu_1 \\ \cdots \\ \mu_n \end{bmatrix}$$
$$C = \begin{bmatrix} \sigma_{00} & \sigma_{01} & \cdots & \sigma_{0n} \\ \sigma_{10} & \sigma_{11} & \cdots & \sigma_{1n} \\ \vdots & \vdots & \vdots & \vdots \\ \sigma_{n0} & \sigma_{n1} & \cdots & \sigma_{nn} \end{bmatrix} \tag{4.9}$$

Where

$$\sigma_{ij} = \sum_{k=0}^{n}\left(x_i^{(k)} - \mu_i\right)\left(x_j^{(k)} - \mu_j\right) \tag{4.10}$$

For $i = j$, σ_{ii} is essentially the variance of the data along the i^{th} dimension. Thus the diagonal elements of matrix *C* contain the variance along the coordinate axes. Off-diagonal elements correspond to cross-covariances.

Equations 4.8 and 4.9 are equivalent to each other.

What is the direction of maximum spread/variance? Let us first consider an arbitrary direction specified by the unit vector $\hat{l}$. Recalling that the component of any vector along a direction is yielded by the dot product of the vector with the unit direction vector, the components of the data points along $\hat{l}$ are given by

$$X' = \left\{ \hat{l}^T \vec{x}^{(0)}, \hat{l}^T \vec{x}^{(1)}, \cdots, \hat{l}^T \vec{x}^{(n)} \right\}$$

(remember Figure 2.7 - the component of one vector along another is given by the dot product between them? Note that $\hat{l}^T \vec{x}^{(i)}$ are dot products and hence scalar values).

The spread along direction $\hat{l}$ is given by the variance of the scalar values in X'. The mean of the values in X' is given by

$$\begin{aligned}\mu' &= \frac{1}{n}\sum_{i=0}^{n} \hat{l}^T \vec{x}^{(i)} \\ &= \hat{l}^T \left(\frac{1}{n}\sum_{i=0}^{n} \vec{x}^{(i)} \right) \\ &= \hat{l}^T \vec{\mu}\end{aligned}$$

and the variance

$$\begin{aligned}C' &= \frac{1}{n}\sum_{i=0}^{n} \left(\hat{l}^T \vec{x}^{(i)} - \hat{l}^T \vec{\mu} \right) \left(\hat{l}^T \vec{x}^{(i)} - \hat{l}^T \vec{\mu} \right)^T \\ &= \frac{1}{n}\sum_{i=0}^{n} \hat{l}^T \left(\vec{x}^{(i)} - \vec{\mu} \right) \left(\hat{l}^T \left(\vec{x}^{(i)} - \vec{\mu} \right) \right)^T \\ &= \frac{1}{n}\sum_{i=0}^{n} \hat{l}^T \left(\vec{x}^{(i)} - \vec{\mu} \right) \left(\vec{x}^{(i)} - \vec{\mu} \right)^T \hat{l} \\ &= \hat{l}^T \left(\frac{1}{n}\sum_{i=0}^{n} \left(\vec{x}^{(i)} - \vec{\mu} \right) \left(\vec{x}^{(i)} - \vec{\mu} \right)^T \right) \hat{l} \\ &= \hat{l}^T C \hat{l}\end{aligned}$$

The above can be optimized using Quadratic Form optimization techniques we learnt in 4.1. Overall, we have the following results

- Variance is maximal when $\hat{l}$ is along the the eigenvector corresponding to the largest eigenvalue of the covariance matrix *C*. This direction is called the *first principal axis* of the multidimensional data.
- The components of the data vectors along the principal axis are known as *first principal components*.
- The value of the variance along the first principal axis, given by the corresponding eigenvalue of the covariance matrix, is called the *first principal value*.
- The second principal axis is the eigenvector of the covariance matrix corresponding to the second largest eigenvalue of the covariance matrix. Second principal components and values are defined likewise.
- Note that the principal axes are orthogonal to each other, being eigenvectors of the symmetric covariance matrix.

What is the practical significance of PCA? Why would one like to know the direction along which the spread is maximum for a point distribution? The next two sections, sections 4.3.1, 4.3.4 are devoted to answering this question.

4.3.1 Application of PCA in Data Science: Dimensionality Reduction

Dimensionality Reduction is one of the most important tools in the machine learning practitioner's toolbox. In typical real life datasets, larger variations correspond to the underlying structural pattern of the data - which we want to learn. The smaller variations happen due to noise (e.g., faulty collection method) which we want to ignore.

For instance, consider the 2D distribution in Figure 4.2. Let us say that the underlying structure is 1D (shown by the two-arrowed thick line in the figure). The variation along the direction perpendicular to the Principal Axis is noise caused by measurement error.

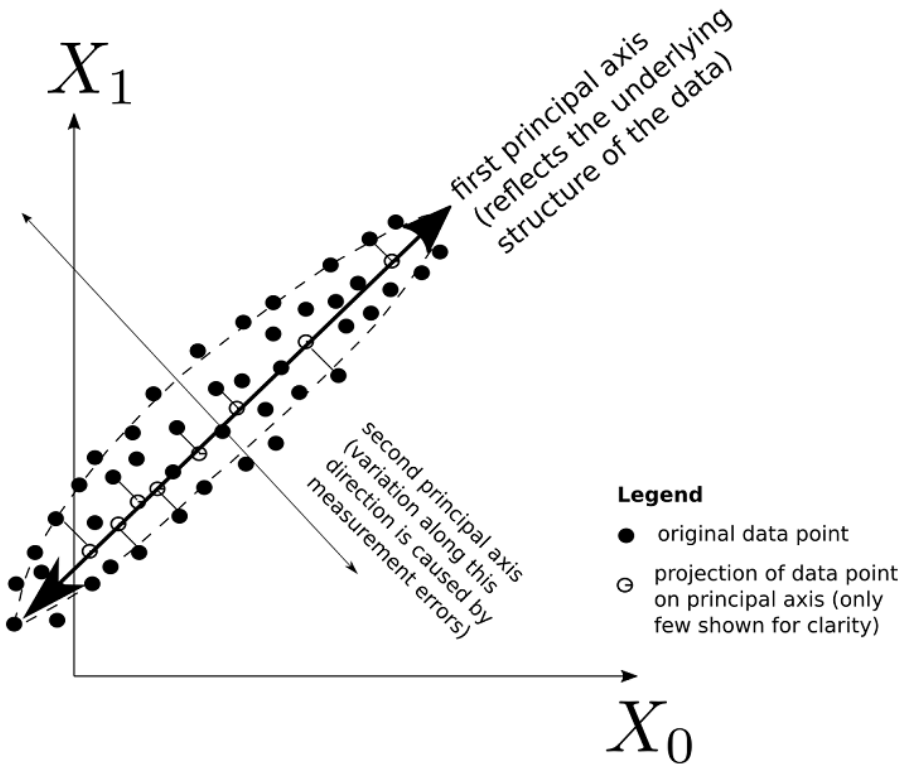


Figure 4.2: *A 2D* data distribution. The true underlying pattern is indicated by the two arrowed straight line. The variation along the direction perpendicular to the true pattern is caused by measurement errors. PCA finds the underlying straight line pattern as first principal axis. Rotating the coordinate system so that this becomes the X axis and then replacing every point by its projection on the new X axis converts the data from 2D to 1D. The projected data is less noisy. And it captures the true distribution.

We could obtain a simpler and at the same time less noisy version of the data by replacing each data point with its projection on the first principal axis. This will convert the 2D dataset into a 1D dataset, thereby reducing its dimensionality. This also brings out the true underlying pattern in the data.

In general, dimensionality reduction throws away the noise or unimportant variations, thereby enabling the machine learning system to discover the true pattern in the data from a more basic and simpler representation. We obtain a lower dimensional representation of the data by getting rid of the principal components corresponding to relatively small principal values.

4.3.2 Python Numpy code: PCA and dimensionality reduction

In this section we provide implementations of PCA in python numpy and demonstrate various properties via simple examples. The first example computes PCA on synthetic correlated data.

The complete code fully functional via Jupyter-Notebook can be found at https://nbviewer.jupyter.org/github/krishnonwork/mathematical-methods-in-deep-learning-ipython/tree/master/python/ch4/.

Listing 4.1: Principal Component Analysis computation

```
def pca(X):                                   # return principal values and vectors
    covariance_matrix = np.cov(X, rowvar=False)
    l, e = np.linalg.eig(covariance_matrix)
    return l, e
```

Fully functional code for PCA computation is available at https://nbviewer.jupyter.org/github/krishnonwork/mathematical-methods-in-deep-learning-ipython/blob/master/python/ch4/4.3.2-common.ipynb.

Listing 4.2: Numpy code demonstrating PCA on synthetic correlated data

```
x_0 = np.random.normal(0, 100, N)             # random feature vector

x_1 = 2 * x_0 +\                              # correlated feature vector
      np.random.normal(0, 20, N)              # + small noise

X = np.column_stack((x_0, x_1))               # data matrix- spread mostly along y = 2x

                                              # one principal value large, one small
                                              # first principal vec along y = 2x
principal_values, principal_vectors = pca(X)

                                              # dimensionality reduction
X_proj = np.dot(X, first_princpal_vec)        # by projecting on first principal vec
```

Output:

```
Principal values are: [54594.39 96.57]
First Principal Vector is: [ -0.44 , -0.89]
```

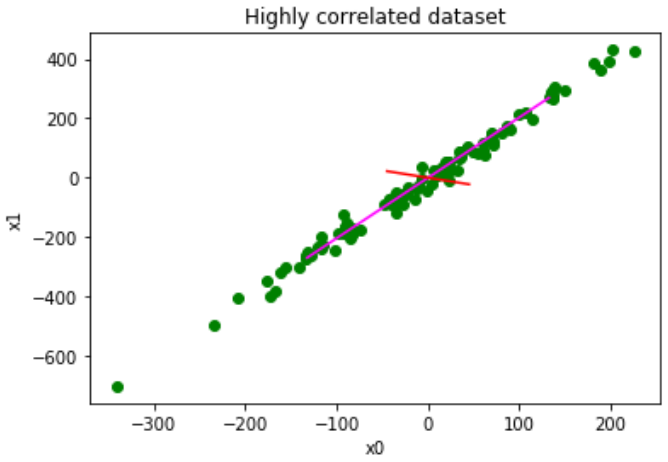


Figure 4.3: One principal value will be much larger than the other - indicating data points are distributed more or less along a straight line, $y = 2x$. The principal axis captures that straight line. If we convert the 2D data to 1D by projecting all data points on the principal axis, little error will occur in positions of individual points - little information loss. This is dimensionality reduction.

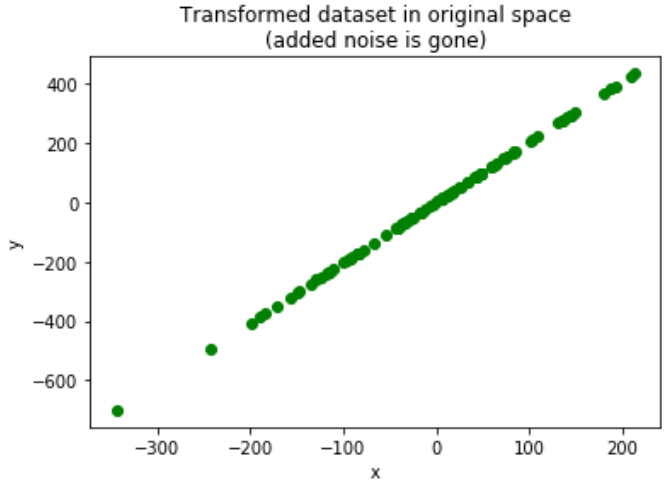


Figure 4.4: Reduced dimensionality representation of same data

Fully functional code for example PCA computation on synthetic correlated data is available at https://nbviewer.jupyter.org/github/krishnonwork/mathematical-methods-in-deep-learning-ipython/blob/master/python/ch4/4.3.2-pca-numpy.ipynb.

Listing 4.3: Numpy code demonstrating PCA on synthetic uncorrelated data

```
x_0 = np.random.normal(0, 100, N)
x_1 = np.random.normal(0, 100, N)
X = np.column_stack((x_0, x_1))


principal_values, principal_vectors = pca(X)
```

Random uncorrelated feature vector pair

principal values close to each other - spread of data points comparable in both directions

Output:

```
Principal values are [8348.79940518 13162.68537655]
```

Fully functional code for PCA computation on synthetic un-correlated data is available at https://nbviewer.jupyter.org/github/krishnonwork/mathematical-methods-in-deep-learning-ipython/blob/master/python/ch4/4.3.2-pca-uncorrelated-numpy.ipynb.

4.3.3 Drawback of PCA from Data Science viewpoint

PCA assumes that the underlying pattern is linear in nature. Where this is not true, PCA will not capture the correct underlying pattern. For instance, see figure 4.6.

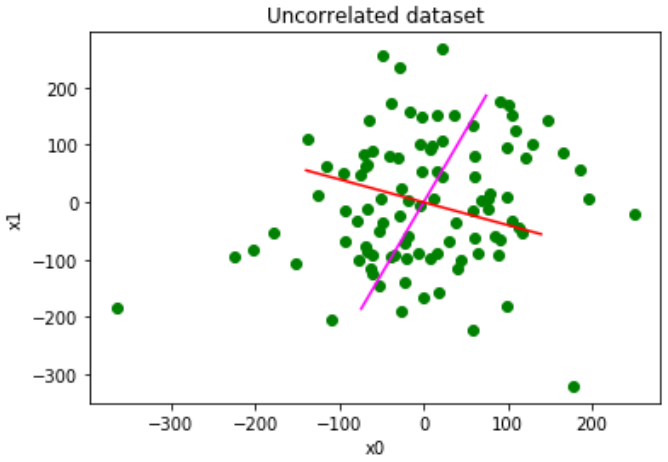


Figure 4.5: PCA on uncorrelated data - projecting on first principal vector will result in large error in positions for most points. Dimensionality reduction will lead to high error.

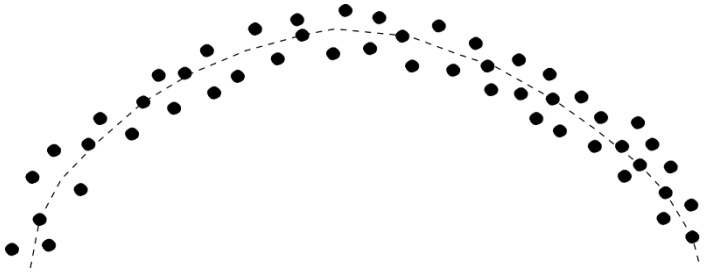


Figure 4.6: *A 2D data distribution with curved underlying pattern. Impossible to find a strait line or vector such that all points are near it. PCA will not do well.*

Listing 4.4: Numpy code demonstrating PCA on synthetic non-linearly correlated data

```
x_0 = np.random.normal(0, 100, N)
x_1 = 2 * (x_0 ** 2) + np.random.normal(0, 5, N)
X = np.column_stack((x_0, x_1))

principal_values, principal_vectors = pca(X)
```

Random non-linearly correlated feature vector pair + noise

principal vectors fail to capture the underlying distribution

4.3.4 Application of PCA in Data Science: Data Compression

If we want to represent a large multi-dimensional dataset within a fixed byte budget, what is the information we can get rid of with least loss of accuracy? Clearly, these are the principal components along the smaller principal values - getting rid of them actually helps, as described in 4.3.1. So, in order to compress data, one often performs PCA and then replaces the data points with their projections on first few principal axes. This reduces the number of data components to store. This is the underlying principle in JPEG 98 image compression techniques.

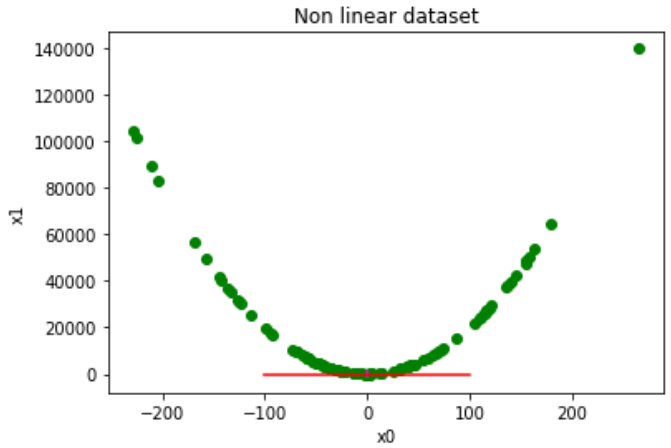


Figure 4.7: PCA on synthetic non linear data. There is correlation, the points (green dots) are distributed along a curve, single dimensional but not linear. The principal axis (red line) is not capturing underlying pattern of the non-linear data. Projecting data on this axis results in large error in data positions - loss of information.

4.4 Singular Value Decomposition

Singular Value Decomposition (abbreviated SVD) maybe the most important linear algebraic tool in machine learning. Among other things, PCA and LSA implementations are built based upon SVD.We will illustrate the basic idea below.[15]

The SVD theorem states that any matrix A, singular or non-singular, rectangular or square, can be decomposed as the product of 3 matrices.

$$A = U\Sigma V^T \tag{4.11}$$

where (assuming that the matrix A is $m \times n$)

- Σ is a $m \times n$ diagonal matrix. Its diagonal elements contain the square roots of the eigenvalues of A^TA. These are also known as the singular values of A. The singular values appear in a decreasing order in the diagonal of Σ.
- V is a $n \times n$ orthogonal matrix containing eigenvectors of A^TA in its columns.
- U is a $m \times m$ orthogonal matrix contains eigenvectors of AA^T in its columns.

We will provide an informal proof of the SVD theorem through a series of Lemmas (small proofs).

[15] There are several mildly different forms of SVD, we have chosen the one that seems intuitively the simplest

Lemma 1: A^TA is symmetric positive semi-definite. Its eigenvalues are non-negative and eigenvectors are orthogonal.

Proof: Let us say A has m rows and n columns. Then A^TA is a $n \times n$ square matrix.

$$(A^TA)^T = A^T(A^T)^T = A^TA$$

which proves that A^TA is symmetric. Also, for any $\vec{x}$,

$$\vec{x}^TA^TA\vec{x} = (A\vec{x})^T(A\vec{x}) = \|A\vec{x}\|^2 > 0$$

which as per section 4.1.1 proves that the matrix A^TA is symmetric and positive semi-definite. Hence, its eigenvalues are all positive or zero.
We have proved in section 2.13 that symmetric matrices have orthogonal eigenvectors. That completes the proof.
Accordingly, let (λ_i, $\hat{v}_i$), for $i \in [1, n]$ be the set of eigenvalue, eigenvector pairs of A^TA. Note that without loss of generality we can assume $\lambda_1 \geq \lambda_2 \geq \cdots \lambda_n$ (because if not we can always renumber the eigenvalues and eigenvectors). Now, by definition

$$A^TA\,\hat{v}_i = \lambda_i\,\hat{v}_i \qquad \forall\, i \in [1, n]$$

From Lemma 1

$$\hat{v}_i^T\hat{v}_j = \begin{cases} 0 & i \neq j \\ 1 & i = j \end{cases} \tag{4.12}$$

Note that $\hat{v}_i$s are unit vectors (that is why we are using the hat sign as opposed to the overhead arrow). As described in section 2.13, eigenvectors remain eigenvectors if we change their length. We are free to choose any length as long as we choose it consistently. We are choosing the unit length eigenvectors here.

Lemma 2: AA^T is symmetric positive semi-definite. Its eigenvalues are non-negative and eigenvectors are orthogonal.

Proof:

$$(AA^T)^T = (A^T)^T\,A^T = AA^T$$

Also,

$$\vec{x}^TAA^T\vec{x} = (A^T\vec{x})^T(A^T\vec{x}) = \|(A^T\vec{x})\| \geq 0$$

Lemma 3: $1/\sqrt{\lambda_i} \times A\hat{v}_i$, $\forall i \in [1, n]$ is a set of orthogonal unit vectors.

Let us take the dot product of a pair of these vectors

$$\left(\frac{1}{\sqrt{\lambda_i}}A\hat{v}_i\right)^T\left(\frac{1}{\sqrt{\lambda_j}}A\hat{v}_j\right) = \frac{1}{\sqrt{\lambda_i\lambda_j}}\hat{v}_i^T A^T A\hat{v}_j$$
$$= \frac{1}{\sqrt{\lambda_i\lambda_j}}\hat{v}_i^T\left(A^T A\hat{v}_j\right)$$

Since λ_j, $\hat{v}_j$ are eigenvalue, eigenvector pairs of A^TA, the above can be rewritten as

$$\frac{1}{\sqrt{\lambda_i\lambda_j}}\hat{v}_i^T\lambda_j\hat{v}_j$$

which, using equation 4.12, can be rewritten as

$$\sqrt{\frac{\lambda_j}{\lambda_i}}\hat{v}_i^T\hat{v}_j = \begin{cases}0 & i \neq j\\ 1 & i = j\end{cases}$$

Lemma 4: If (λ_j, $\hat{v}_j$) is an eigenvalue, eigenvector pair of A^TA, then (λ_i, $\hat{u}_i = 1/\sqrt{\lambda_i}\, A\hat{v}_i$) is an eigenvalue, eigenvector pair of AA^T.

Proof:
By given

$$A^TA\hat{v}_i = \lambda_i\,\hat{v}_i$$

Left multiplying both sides of the equation by *A*, we get

$$AA^TA\hat{v}_i = \lambda_i\,A\hat{v}_i$$
$$AA^T(A\hat{v}_i) = \lambda_i\,(A\hat{v}_i)$$

Substituting $\vec{f}_i = A\hat{v}_i$ in the last equation, we get

$$AA^T\vec{f}_i = \lambda_i\,A\vec{f}_i$$

which proves $\vec{f}_i = A\hat{v}_i$ is an eigenvector of AA^T with λ_i as corresponding eigenvalue. Multiplying by $1/\sqrt{\lambda_i}$ converts it into an unit vector as per Lemma 3. This completes the proof of the lemma.
Now we are ready to go into the proof of the SVD theorem.

Case 1: **More rows that columns in A**

If m, the number of rows in A is greater than or equal to n, the number of columns in A, we define

$$U = \begin{bmatrix} \hat{u}_1 & \hat{u}_2 & \cdots & \hat{u}_n & \hat{u}_{n+1} & \cdots \hat{u}_m \end{bmatrix}$$

$$\Sigma = \begin{bmatrix} \sqrt{\lambda_1} & 0 & \cdots & 0 \\ 0 & \sqrt{\lambda_2} & \cdots & 0 \\ & \vdots & & \\ 0 & 0 & \cdots & \sqrt{\lambda_n} \\ 0 & 0 & \cdots & 0 \\ & \vdots & & \\ 0 & 0 & \cdots & 0 \end{bmatrix}$$

$$V = \begin{bmatrix} \hat{v}_1 & \hat{v}_2 & \cdots & \hat{v}_n \end{bmatrix}$$

Note:

- From Lemma 1, we know that the eigenvalues of A^TA are positive. This makes the square roots, $\sqrt{\lambda_i}s$, real.
- U is a $m \times m$ orthogonal matrix whose columns are the eigenvectors of AA^T. Since, AA^T is $m \times m$, it has m eigenvalues and eigenvectors. The first n of them are $\hat{u}_1 = 1/\sqrt{\lambda} \cdot A\hat{v}_1$, $\hat{u}_2 = 1/\sqrt{\lambda} \cdot A\hat{v}_2, \cdots$, $\hat{u}_n = 1/\sqrt{\lambda_i} \cdot A\hat{v}_n$ (from Lemma 4 we know these are eigenvectors of AA^T). In this case, by our initial assumption, $n < m$. Thus AA^T has $(m \times n)$ more eigenvectors, $\hat{u}_{n+1}, \cdots \hat{u}_m$.

Consider the matrix product $U\Sigma$. From basic matrix multiplication rules (section 2.5, one can see that

$$U\Sigma = \begin{bmatrix} \hat{u}_1 & \hat{u}_2 & \cdots & \hat{u}_n & \hat{u}_{n+1} & \cdots \hat{u}_m \end{bmatrix} \begin{bmatrix} \sqrt{\lambda_1} & 0 & \cdots & 0 \\ 0 & \sqrt{\lambda_2} & \cdots & 0 \\ & \vdots & & \\ 0 & 0 & \cdots & \sqrt{\lambda_n} \\ 0 & 0 & \cdots & 0 \\ & \vdots & & \\ 0 & 0 & \cdots & 0 \end{bmatrix}$$

$$= \begin{bmatrix} \sqrt{\lambda_1}\hat{u}_1 & \sqrt{\lambda_2}\hat{u}_2 & \cdots & \sqrt{\lambda_n}\hat{u}_n \end{bmatrix}$$

Note that the last columns of U, $\hat{u}_{n+1}, \cdots \hat{u}_m$ are getting multiplied by all zeros in Σ and vanishing. Thus,

$$
\begin{aligned}
U\Sigma &= \begin{bmatrix} \sqrt{\lambda_1}\hat{u}_1 & \sqrt{\lambda_2}\hat{u}_2 & \cdots & \sqrt{\lambda_n}\hat{u}_n \end{bmatrix} \\
&= \begin{bmatrix} A\hat{v}_1 & A\hat{v}_2 & \cdots & A\hat{v}_n \end{bmatrix} \\
&= A \begin{bmatrix} \hat{v}_1 & \hat{v}_2 & \cdots & \hat{v}_n \end{bmatrix} \\
&= AV
\end{aligned}
$$

Here the later columns of U, the ones named with u's fail to survive as they get multiplied with the zeros at the bottom of Σ. Thus we have proved that

$$AV = U\Sigma$$

Then

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

Since V is orthogonal, VVT = **I**. Hence

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

which completes the proof of the Singular Value Theorem.

Case 2: **Less rows than columns in A**

If m, the number of rows in A is lesser than or equal to n, the number of columns in A, we will have

$$
\begin{aligned}
U &= \begin{bmatrix} \hat{u}_1 & \hat{u}_2 & \cdots & \cdots \hat{u}_m \end{bmatrix} \\
\Sigma &= \begin{bmatrix} \sqrt{\lambda_1} & 0 & \cdots & 0 & \cdots & 0 \\ 0 & \sqrt{\lambda_2} & \cdots & 0 & \cdots & 0 \\ & \vdots & & & & \\ 0 & 0 & \cdots & \sqrt{\lambda_n} & \cdots & 0 \end{bmatrix} \\
V &= \begin{bmatrix} \hat{v}_1 & \hat{v}_2 & \cdots & \hat{v}_n \end{bmatrix}
\end{aligned}
$$

The proof follows along similar lines.

4.4.1 Application of SVD: PCA computation

We will illustrate the idea first with a toy dataset. Consider a 3D dataset with 5 points. We will use a superscript to denote the index of the data instance and subscript to denote the component. Thus the i^{th} data instance vector is denoted as $[x_0{}^{(i)}\ x_1{}^{(i)}\ x^{(i)}{}_2]$. We denote the entire data set with a matrix in which each instance appears as a row vector. The data matrix is

$$X = \begin{bmatrix} x_0^{(0)} & x_1^{(0)} & x_2^{(0)} \\ x_0^{(1)} & x_1^{(1)} & x_2^{(1)} \\ x_0^{(2)} & x_1^{(2)} & x_2^{(2)} \\ x_0^{(3)} & x_1^{(3)} & x_2^{(3)} \\ x_0^{(4)} & x_1^{(4)} & x_2^{(4)} \end{bmatrix}$$

We will assume that the data is already mean-subtracted. Now examine the matrix product X^TX. Using ordinary rules of matrix multiplication

$$X^TX = \begin{bmatrix} \sum_{i=0}^{4} \left(x_0^{(i)}\right)^2 & \sum_{i=0}^{4} x_0^{(i)} x_1^{(i)} & \sum_{i=0}^{4} x_0^{(i)} x_2^{(i)} \\ \sum_{i=0}^{4} x_1^{(i)} x_0^{(i)} & \sum_{i=0}^{4} \left(x_1^{(i)}\right)^2 & \sum_{i=0}^{4} x_1^{(i)} x_2^{(i)} \\ \sum_{i=0}^{4} x_2^{(i)} x_0^{(i)} & \sum_{i=0}^{4} x_2^{(i)} x_1^{(i)} & \sum_{i=0}^{4} \left(x_2^{(i)}\right)^2 \end{bmatrix}$$

From equations 4.10 and 4.9

$$X^TX = \begin{bmatrix} \sigma_{00} & \sigma_{01} & \sigma_{02} \\ \sigma_{10} & \sigma_{11} & \sigma_{12} \\ \sigma_{20} & \sigma_{21} & \sigma_{22} \end{bmatrix} = C$$

Thus X^TX is the covariance matrix of the dataset X. This in fact holds for arbitrary dimensions and arbitrary instance counts.

Thus, if we create a data matrix X with each data instance forming a row, X^TX yields the covariance matrix of the dataset. Now, from sec 4.3 we know its eigenvectors of the covariance matrix are the principal axes and corresponding eigenvalues are the principal values of the dataset. Also, from section 4.4 we know that performing SVD on X would yield

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

where the columns of V are the eigenvectors and the squares of the singular values are the eigenvalues of X^TX Hence we see, performing SVD on a data matrix yields PCA of the data (assuming prior mean subtraction).

4.4.2 Application of SVD: Solving arbitrary Linear System

A linear system is a system of simultaneous linear equations

$$A\vec{x} = \vec{b}$$

We first encountered a linear system in section 2.12. It is possible to use matrix inversion to solve such a system, as

$$\vec{x} = A^{-1}\,\vec{b}$$

However, this is numerically unstable. This is because the matrix inverse contains the determinant of the matrix in its denominator. The determinant can be zero - then this method of solution is not feasible. Ideally, in this case, we would have liked to obtain a "best effort" solution. If the determinant is near zero, the inverse will contain very large numbers. Minor noise in $\vec{b}$ will get multiplied by these large numbers and cause large errors in computed solution $\vec{x}$. Solving a linear system via SVD, $A = U\Sigma V^T$ addresses all these issues.

The steps are as follows

- $A\vec{x} = \vec{b} \Rightarrow U(\Sigma V^T\vec{x}) = \vec{b}$

  Solve $U\vec{y}_1 = \vec{b}$ using orthogonality of U, as $\vec{y}_1 = U^T\vec{b}$
- Now we have $\Sigma(V^T\vec{x}) = \vec{y}_1$

  Solve $\Sigma\vec{y}_2 = \vec{y}_1$

  For any diagonal matrix

$$\Sigma = \begin{bmatrix} d_1 & 0 & \cdots & 0 \\ 0 & d_2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & \cdots & \cdots & d_n \end{bmatrix}$$

  we can easily compute

$$\Sigma^{-1} = \begin{bmatrix} \frac{1}{d_1} & 0 & \cdots & 0 \\ 0 & \frac{1}{d_2} & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & \cdots & \cdots & \frac{1}{d_n} \end{bmatrix}$$

  Hence, $\vec{y}_2 = \Sigma^{-1}\vec{y}_1$
- Now we have $V^T\vec{x} = \vec{y}_2$. Solve this using orthogonality of V as $\vec{x} = V\vec{y}_2$

Thus we have solved for $\vec{x}$ without inverting the matrix A.

- For invertible square matrices A this method will yield the same solution as the matrix inverse based method.
- For non square matrices, this boils down to the Moore Penrose inverse and yields the best effort solution.

4.4.3 Rank of a Matrix

In section 2.12 we studied linear system of equations. Such a system can be represented in matrixvector form

$$A\vec{x} = \vec{b}$$

Each row of A and $\vec{b}$ contributes one equation. If we have as many independent equations as unknown, the system is solvable. This is the simplest case. In this case, the matrix A is square and invertible. $det(A)$ is non-zero and A^{-1} exists.

Sometimes, the situation maybe misleading. Consider the following system.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 5 \\ 7 \\ 12 \end{bmatrix}$$

Although there are 3 rows and apparently 3 equations, the equations are not independent. For instance, the third equation can be obtained by adding the first two. We really have only 2, not 3, equations. We say the linear system is "degenerate". All the following statements will be true for such a system $A\vec{x} = \vec{b}$

- The linear system is degenerate
- $det(A) = 0$
- A^{-1} cannot be computed, A is not invertible.
- Rows of A are linearly dependent. There exists a linear combination of the rows that sum to zero. E.g., in the above example $\vec{r}_0 + \vec{r}_1 = 0$
- At least one of the singular values of A, i.e., eigenvalues of A^TA is zero. In fact, the number of linearly independent rows is equal to the number of non-zero eigenvalues.

The number of linearly independent rows in a matrix is called its rank. It can be proved that a matrix has as many non-zero singular values as its rank. It can also be proved that the number of linearly independent columns in a matrix matches the number of linearly independent columns. Hence, rank can also be defined as the number of linearly independent columns in a matrix.

A non-square rectangular matrix with m rows and n columns will have a rank $r = min(m, n)$. Such matrices are never invertible. One usually resorts to SVD to solve them.

A square matrix with n rows and n columns will be invertible (non-zero determinant) if and only if it has rank n. Such a matrix is said to have full rank. Full rank matrices are invertible. They can be solved via matrix inverse computation. However, inverse computation is not numerically stable always. SVD can be applied here as well with good numerical properties. Non full rank matrices are degenerate. Rank then, is a measure of non-degeneracy of the matrix.

4.4.4 Python numpy code for linear system solving via SVD

This section shows numpy based implementation of Singular Value Decomposition (SVD) and demonstrates an application - solving a linear system via SVD.

Listing 4.5: Numpy code to solve invertible linear system via matrix inversion as well as SVD

```
A = np.array([[1, 2, 1], [2, 2, 3], [1, 3, 3]])
b = np.array([8, 15, 16])

x_0 = np.matmul(np.linalg.inv(A), b)

U, S, V_t = np.linalg.svd(A)

y1 = np.matmul(U.T, b)

S_inv = np.diag(1 / S)

y2 = np.matmul(S_inv, y1)

x_1 = np.matmul(V_t.T, y2)

assert np.allclose(x_0, x_1)
```

← Simple test linear system of equations

matrix inversion is numerically unstable, SVD better

$A = USV^T \implies A\vec{x} = \vec{b} \triangleq USV^T\vec{x} = \vec{b}$

Solve $U\vec{y}_1 = \vec{b}$, remember $U^{-1} = U^T$ as U is orthogonal

Solve $S\vec{y}_2 = \vec{y}_1$, remember S^{-1} is easy as S is diagonal

Solve $V^T\vec{x} = \vec{y}_2$, remember $V^{-T} = V$ as V is orthogonal

two solutions are same

Output:

```
Solution via inverse : [ 1. 2. 3.]
Solution via SVD : [ 1. 2. 3.]
```

Listing 4.6: Numpy code to solve overdetermined linear system via Pseudo-Inverse and SVD

```
A = np.array([[0.11, 0.09], [0.01, 0.02],
              [0.98, 0.91], [0.12, 0.21],
              [0.98, 0.99], [0.85, 0.87],
              [0.03, 0.14], [0.55, 0.45],
              [0.49, 0.51], [0.99, 0.01],
              [0.02, 0.89], [0.31, 0.47],   <- cat-brain dataset - non-square matrix
              [0.55, 0.29], [0.87, 0.76],
              [0.63, 0.24]])
A = np.column_stack((A, np.ones(15)))
b = np.array([-0.8, -0.97, 0.89, -0.67,
              0.97, 0.72, -0.83, 0.00,
              0.00, 0.00, -0.09, -0.22,
              -0.16, 0.63, 0.37])

                  solution via pseudo-inverse


x_0 = np.matmul(np.linalg.pinv(A), b)

                        solution via SVD

U, S, V_t = np.linalg.svd(A, full_matrices=False)
y1 = np.matmul(U.T, b)
S_inv = np.diag(1 / S)
y2 = np.matmul(S_inv, y1)
x_1 = np.matmul(V_t.T, y2)


assert np.allclose(x_0, x_1)   <- two solutions are same
```

Output:

```
Solution via pseudo - inverse : [1.07661761 0.89761672 -0.95816936]
Solution via SVD : [1.07661761 0.89761672 -0.95816936]
```

Fully functional code for SVD based linear system solving can be found at https://nbviewer.jupyter.org/github/krishnonwork/mathematical-methods-in-deep-learning-ipython/blob/master/python/ch4/4.4.4-svd-linear-system-numpy.ipynb.

4.4.5 Python numpy code for PCA computation via SVD

This section demonstrates PCA computation via SVD.

Listing 4.7: Computing PCA directly and using SVD

```
1
2 principal_values, principal_vectors = pca(X)
3
4
5 X_mean = X - np.mean(X, axis=0)
6 U, S, V_t = np.linalg.svd(X_mean)
7
8
9 V = V_t.T
```


4.4.6 Application of SVD: Best low rank approximation of a matrix

Given a matrix A of some rank p, we sometimes want to approximate it with a matrix of lower rank r, where $r < p$. How do we obtain the best rank r approximation of A?

MOTIVATION

Why would one want to do this? Well, consider a data matrix X as shown in section 4.4.1. As explained in section 4.3.1, we often want to eliminate the small variances in the data (likely due to noise) and get the pattern underlying the large variations. Replacing the data matrix with a lower rank matrix is a tool that often achieves this. Although, we must bear in mind that this does not happen when the underlying pattern is non-linear (e.g., Figure 4.6).

APPROXIMATION ERROR

What exactly do we mean by "best approximation"? The Frobenius norm can be taken as the magnitude of the matrix. Accordingly, given a matrix A and its rank r approximation A_r, the approximation error is $e = ||A - A_r||_F$.

METHOD

To fix our ideas, let us consider a $m \times n$ matrix A. From sec 4.4 we know it will have $min(m, n)$ singular values. Let its rank be $p \leq min(m, n)$. We want to approximate this matrix with a rank $r(< p)$ matrix.

Let us rewrite the SVD expression. To make our ideas concrete, we will assume $m > n$. Also, as usual, we have the singular values sorted in decreasing order $\lambda_1 \geq \lambda_2 \geq \lambda_n$. We will partition U, Σ, V.

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

$$= \begin{bmatrix} \hat{u}_1 & \cdots & \hat{u}_r & \hat{u}_{r+1} & \cdots \hat{u}_m \end{bmatrix} \begin{bmatrix} \sqrt{\lambda_1} & 0 & \cdots & \cdots & 0 & \cdots & 0 \\ 0 & \sqrt{\lambda_2} & \cdots & \cdots & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & \sqrt{\lambda_r} & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 & \sqrt{\lambda_{r+1}} & \cdots & 0 \\ \vdots & \vdots & \cdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & 0 & 0 & \cdots & \sqrt{\lambda_n} \\ 0 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & 0 & 0 & \cdots & 0 \end{bmatrix} \begin{bmatrix} \hat{v}_1^T \\ \cdots \\ \hat{v}_r^T \\ \hat{v}_{r+1}^T \\ \cdots \\ \hat{v}_n^T \end{bmatrix}$$

$$= \begin{bmatrix} U_1 & U_2 \end{bmatrix} \begin{bmatrix} \Sigma_1 & 0 \\ 0 & \Sigma_2 \end{bmatrix} \begin{bmatrix} V_1^T \\ V_2^T \end{bmatrix}$$

$$= U_1 \Sigma_1 V_1^T + U_2 \Sigma_2 V_2^T$$

It can be proved that $U_1 \Sigma_1 V_1{}^T$ is a rank r matrix. Furthermore, it is best rank r approximation of A.

4.5 Machine Learning Application: Document Retrieval

We will now bring together several of the concepts we studied with a illustrative toy example, viz. the document retrieval problem we first encountered in section 2.1. Briefly recapitulating, we have a set of documents $\{d_0, \cdots, d_6\}$. Now given an incoming query phrase, we have to retrieve documents that match the query phrase. We will use *bag of words* model - i.e., our matching approach does not pay attention to *where* a word appears in a document, it simply pays attention to *how many times* a word appears in a document. Although not the most sophisticated, this is quite a popular technique owing to its conceptual simplicity.

Our documents are:

- d_0: Roses are lovely. Nobody hates roses.
- d_1: Gun violence has reached an epidemic proportion in America.
- d_2: The issue of gun violence is really over-hyped. One can find many instances of violence where no guns were involved.
- d_3: Guns are for violence prone people. Violence begets guns. Guns beget violence.
- d_4: I like guns but I hate violence. I have never been involved in violence. But I own many guns. Gun violence is incomprehensible to me. I do believe gun owners are the most anti violence people on the planet. He who never uses a gun will be prone to senseless violence.
- d_5: Guns were used in a armed robbery in San Francisco last night
- d_6:Acts of violence usually involve a weapon.

4.5.1 TF-IDF and Cosine Similarity in Machine Learning based Document Retrieval

Before studying PCA, let us study some more elementary techniques for document retrieval. These are based on TF-IDF and Cosine Similarity.

TERM FREQUENCY (TF)

Term Frequency is defined as the number of occurrences of a particular term in a document. In a slightly looser sense, any quantity proportional to the number of occurrences is also known as Term Frequency. E.g., TF of the word "gun" in d_0, d_6 is 0, in d_1 it is 1, in d_3 it is 3 etc. Note that we are being case independent.

INVERSE DOCUMENT FREQUENCY(IDF)

Certain terms, e.g., "the", appears in pretty much all documents. These should be ignored during document retrieval. How do we down-weight them?

IDF is obtained by inverting and then taking logarithm of the fraction of all documents in which the term occurs. For terms that occur in most documents, IDF weight will be very low. It will be high for relatively esoteric terms.

DOCUMENT FEATURE VECTORS

Each document is represented by a vector with as many elements as the size of our vocabulary (i.e., the number of distinct words over all the documents). For real life document retrieval systems like Google, this vector will be very very long indeed. But not to worry, this vector is notional, it never gets explicitly stored in the computer's memory. Anyway, this vector has a fixed position for every word in the vocabulary. Each position in this vector contains the TF of the corresponding term. In order to minimize the contributions from ubiquitous terms, we use product of TF and IDF instead of TF alone.

COSINE SIMILARITY

In section 2.5.6 we saw that dot product between two vectors measures the agreement between them.

Given two vectors $\vec{a}$ and $\vec{b}$ we know $\vec{a} \cdot \vec{b} = \|\vec{a}\| \, \|\vec{b}\| \, cos \, (\theta)$, where the operator $\| \cdot \|$ implies length of a vector, and θ is the angle between the 2 vectors (see Figure 2.7). The cosine is at its maximum possible value, 1, when the vectors are pointing in the same direction and the angle between them is zero. It progressively becomes smaller as the angle between the vectors increases until the two vectors become perpendicular to each other when the cosine becomes zero, implying no correlation - vectors are independent of each other.

The magnitude of dot product itself is also proportional to the length of the two vectors. Hence, we do not want to use the dot product par se as a measure of similarity between the vectors. Because then two long vectors would have a high score of similarity even if they are not aligned in direction. Rather, we want to use the cosine, defined as

$$cosine\_similarity\left(\vec{a},\vec{b}\right) = \frac{\vec{a}^T\vec{b}}{\|\vec{a}\|\|\vec{b}\|} \tag{4.13}$$

The cosine similarity between document vectors is often used to measure similarity between two documents. It is a principled way of measuring the degree of term sharing between the two documents.

4.5.2 Latent Semantic Analysis (LSA)

Cosine similarity and such techniques suffer from a significant drawback. To see this, examine the cosine similarity between d_5 and d_6. It will be zero. But it is obvious to a human that the documents are similar.

What went wrong? Answer: we are measuring only direct overlap between terms in documents. The words "gun" and "violence" occurred together in many of the *other* documents to indicate some degree of similarity between them. But cosine similarity between document vectors does not look at such secondary evidence.

This is the blind spot that LSA tries to overcome.

Words are known by the company they keep. This means, if terms appear together in many documents, they are likely to share some semantic similarity. For instance, guns and violence in the above examples. Such terms should be grouped together into a common pool of semantically similar terms. We will call this pool a *topic*. Document similarity should be measured in terms of common topics rather than common terms. Thus, we have expanded the notion of shared terms between documents to shared topics between documents.

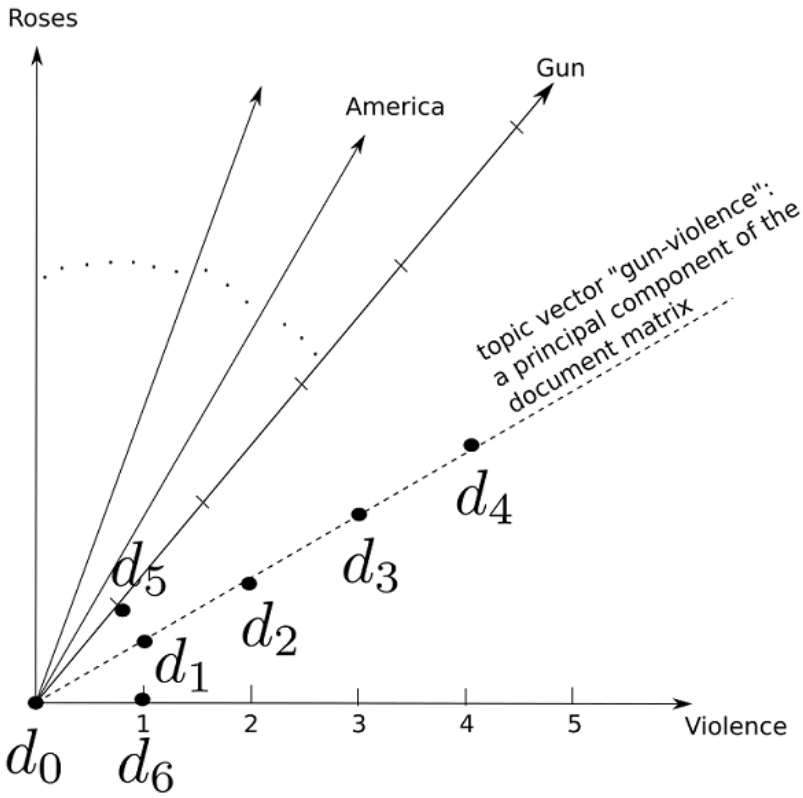


Figure 4.8: Document Vectors from our toy dataset $d_0, \cdots d_6$. Each word in the vocabulary corresponds to a separate dimension. Dots show projections of document feature vectors on the plane formed by the axes corresponding to the terms "gun(s)" and "violence"

How do we automatically identify topics from an universe of documents? Answer: we look for terms that occur together in many documents. Geometrically, this would manifest as follows: if we project the points representing document vectors on the subspace corresponding to the topic, we will see correlation (see Figure 4.8). This means - recall section 4.3 - we expect principal components along the topics. We can identify these principal components by taking SVD on the data matrix with individual document vectors along its rows. Then, the right eigenvectors would be along topics. Projecting the document vectors on the subspace of these eigenvectors would yield topic modeling of the document. As indicated before, we can set the smaller eigenvalues to zero to eliminate noise and unimportant topics.

The document matrix (with document vectors as rows) -we will omit prepositions, conjunctions, commas etc - looks like Thus, the oveall steps are as follows (see Listing 4.5.3 for python code):

Table 4.1: Documents matrix for toy example dataset. Rows correspond to documents. Columns correspond to terms. Each cell contains the term frequency. The terms "Gun" and "Violence" occur equal number of times in most documents, indicating clear correlation (Gun-Violence is a topic). Principal Components (right eigenvectors) will identify topics.

	violence	gun	America	...	Roses
$\mathbf{d}_0$	0	0	0	...	2
$\mathbf{d}_1$	1	1	1	...	0
$\mathbf{d}_2$	2	2	0	...	0
$\mathbf{d}_3$	3	3	0	...	0
$\mathbf{d}_4$	5	5	0	...	0
$\mathbf{d}_5$	0	1	0	...	0
$\mathbf{d}_6$	1	0	0	...	0

- Create document term matrix, X, of dimension say $m \times n$. Its rows correspond to documents (m documents), columns correspond to terms (n terms).
- Perform SVD on document term matrix. This yields U, S and V matrices. V is $n \times n$ orthogonal matrix. S is a diagonal matrix.
- Columns of resulting V matrix yield topics. These are principal vectors for the rows of X, i.e., eigenvectors of $X^T X$. Initially V has n , i.e., n topics.
- The successive elements of each topic vector (column of V matrix) tell us the contribution of corresponding terms to that topic. Each of these columns is $n \times 1$, depicting the contributions of the n terms in the system.
- The diagonal elements of S tell us the weights (importance) of corresponding topics. These are the eigenvalues of $X^T X$, i.e., principal values of the row vectors of X.
- We inspect weights and choose a cutoff. All topics below that weight are discarded. The corresponding columns of V are thrown away. This yields a V matrix with fewer columns (but same number of rows). These are the topic vectors of interest to us. We have reduced dimensionality of the problem. Let us say, the number of retained topics is t. The reduced V matrix is $m \times t$
- By projecting (multiplying) the original doc-term matrix X to this new V matrix, we will get a $m \times t$ doc-topic matrix (it has same number of rows as X, but fewer columns). This is theprojection of the doc-term matrix to the topic space, i.e., topic based representation of thedocument vectors.
- Rows of the doc-topic matrix will henceforth be taken as document representations. Document similarities will be computed by taking cosine similarity of these rows, as opposed to rows of the original doc-term matrix. This cosine-similarity, in the topic space, will capture many indirect connections that were not visible in the original input space.

4.5.3 Python/Numpy code to compute LSA on a toy dataset

Fully functional code for this section, can be found at

https://nbviewer.jupyter.org/github/krishnonwork/mathematical-methods-in-deep-learning-ipython/blob/master/python/ch4/4.5.3-svd-lsa-toy-dataset-numpy.ipynb.

Here we will present annotated code snippets to explain the idea.

Listing 4.8: Latent Semantic Analysis on the toy dataset from Table 4.1

```
terms = ["violence", "gun", "america", "roses"]
X = np.array([[0, 0, 0, 2],
              [1, 1, 1, 0],
              [2, 2, 0, 0],
              [3, 3, 0, 0],
              [5, 5, 0, 0],
              [0, 1, 0, 0],
              [1, 0, 0, 0]])

U, S, V_t = np.linalg.svd(X)
V = V_t.T

rank = 1
U = U[:, :rank]
V = V[:, :rank]

topic0_term_weights = list(zip(terms, V[:, 0]))

def cosine_similarity(vec_1, vec_2):
    vec_1_norm = np.linalg.norm(vec_1)
    vec_2_norm = np.linalg.norm(vec_2)
    return np.dot(vec_1, vec_2) / (vec_1_norm * vec_2_norm)

d5_d6_cosine_similarity = cosine_similarity(X[5], X[6])
doc_topic_projection = np.dot(X, V)
d5_d6_lsa_similarity = cosine_similarity(doc_topic_projection[5],
                                         doc_topic_projection[6])
```

consider only 4 terms for simplicity

Document Term matrix. Each row describes a document. Each column contains TF scores for one term. IDF is ignored for simplicity.

Perform SVD on the doc-term matrix. Columns of resulting matrix V correspond to topics. These are eigenvectors of X^TX, i.e., principal vectors of the doc-term matrix. Thus, a topic corresponds to the direction of maximum variance in doc feature space.

S indicates the diagonal matrix of principal values. These signify topic weights (importance). We choose a cut-off and discard all topics below that weight - dimensionality reduction. Thus, only first few columns of V are retained. Principal values (topic weights) for this dataset are shown in output. Only 1 topic retained in this example.

Elements of topic vector show contributions of corresponding terms to the topic

Cosine similarity in feature space fails to capture d, d6 similarity. LSA succeeds.

4.5.4 Python/Numpy code to compute and visualize LSA/SVD on a 500 × 3 dataset

Supposing we have a set of 500 documents over a vocabulary of 3 terms. It is a unrealistically short vocabulary, but it allows us to easily visualize the space of document vectors. Each document vector is 3×1 vector and there are 500 such vectors. Together they form a 500×3 data matrix X. In the python code, we create an artificial dataset like this. In this dataset, the terms $x0$ and $x1$ are correlated. $x0$ occurs randomly between 0 and 100 times in a document.

$x1$ occurs twice as many times as $x0$ except for small random fluctuations. The third term's frequency varies between 0 and 5. From section 4.5, we know that $x0$, $x1$ together form a single topic while $x2$ by itself forms another topic. We expect a principal component along each topic. The code below creates the dataset, computes SVD, plots the dataset and shows the first two principal components in red and purple respectively. The third singular value is small compared to the first. We can ignore that dimension - it corresponds to the small random variation within the $x0 - x1$ topic.

Listing 4.9: Latent Semantic Analysis using SVD

```
num_examples = 500

x0 = np.rint(np.random.normal(0, 100, num_examples))
random_noise = np.rint(np.random.normal(0, 2, num_examples))
x1 = 2*x0 + random_noise
x2 = np.rint(np.random.normal(0, 5, num_examples))
X = np.column_stack((x0, x1, x2))
U, S, V_t = np.linalg.svd(X)
V = V_t.T
```

3D dataset - first two axes are linearly correlated
third axis has small near zero random values

Third singular value relatively small - ignore

First two principal vectors each represent a topic
Projecting data points on them
yield document descriptor in terms of the two topics

Output:

```
Singular values are: 5305.37495081 , 109.572182265 , 19.4568281491
```

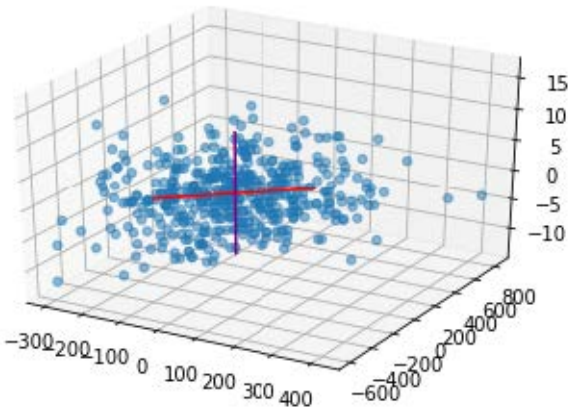


Figure 4.9: Latent Semantic Analysis. Note: the scale is very different along the third axis, purple line is much smaller than red line.

4.6 Summary

In this chapter, we studied several linear algebraic tools in machine learning and data science.

- We learnt that the direction (unit vector) which maximizes (minimizes) the quadratic form $\hat{x}^T A \hat{x}$ is the eigenvector corresponding to the largest (smallest) eigenvalue of matrix A. The magnitude of the quadratic form when $\hat{x}$ is along those directions is the largest (smallest) eigenvalue of A.
- We learnt that given a set of points, $X = \{\vec{x}^{(0)}, \vec{x}^{(1)}, \vec{x}^{(2)}, \cdots, \vec{x}^{(n)}\}$ in an $n+1$ dimensional space, we can define the mean vector and covariance matrix as

$$\vec{\mu} = \frac{1}{n}\sum_{i=0}^{n} \vec{x}^{(i)}$$

$$C = \frac{1}{n}\sum_{i=0}^{n} \left(\vec{x}^{(i)} - \vec{\mu}\right)\left(\vec{x}^{(i)} - \vec{\mu}\right)^T$$

  The variance along an arbitrary direction (unit vector) $\hat{l}$ is $\hat{l}^T C \hat{l}$. This is a quadratic form. Consequently, the maximum (minimum) variance of a set of data points in multidimensional space occurs along the eigenvector corresponding to the largest (smallest) eigenvalue of the covariance matrix. This direction is called the first principal axis of the data. The subsequent eigenvectors, sorted in order of decreasing eigenvalues, will be mutually orthogonal (perpendicular) and will yield subsequent direction of maximum variance. This technique is known as *Principal Component Analysis (PCA)*.

  In many real life cases, the larger variances correspond to true underlying pattern of the data while smaller variances correspond to noise (e.g., measurement error). Projecting the data on the principal axes corresponding to the larger eigenvalues would yield a lower dimensional data that is relatively noise free. Also, the projected data points match the true underlying pattern more closely, hence yield better insights. This is known as *dimensionality reduction*.

- We studied *Singular Value Decomposition (SVD)* - a technique that allows us to decompose an arbitrary *m* × *n* matrix A as a product of 3 matrices: $A = U\Sigma V^T$ where U, V are orthogonal and Σ is diagonal. Matrix V has the eigenvectors of $A^T A$ as its columns. U has eigenvectors of AA^T as columns. Σ has the eigenvalues of $A^T A$ (sorted in decreasing order) in its diagonal.
  - SVD provides us with a numerically stable way to solve the linear system of equations $A\vec{x} = \vec{b}$. In particular, for non-square matrices, it provides the closest approximations, viz., $\vec{x}$ that minimizes $||A\vec{x} = \vec{b}||$.
  - Given a dataset X whose rows are data vectors corresponding to individual instances and columns correspond to feature values, $X^T X$ yields the covariance matrix. Thus eigenvectors of $X^T X$ yields the principal components of the data. Since SVD of X has eigenvectors of $X^T X$ as columns of the matrix V, SVD is an effective way of computing PCA.

- Finally we studied a real life machine learning, data science application for *document retrieval*.
    - We learnt the bag of words model where documents are represented by document vectors that contain the term frequency (number of occurrences) of each term in the document.
    - We studied *TF-IDF* an *cosine similarity* techniques for document matching and retrieval. We also studied the drawbacks of these techniques.
    - Finally we studied *Latent Semantic Analysis (LSA)* which does *topic modeling*. Here, we do PCA on the document vectors to identify topics. Projecting document vectors onto topic axes allows LSA to see latent (indirect) similarities beyond direct overlapping of terms.

5

Probability Distributions for Machine Learning and Data Science

In machine learning, we take large feature vectors as inputs. and ascribe them to one or more of pre-defined classes. Such a machine is called a *classifier*.

As stated earlier, we can view the feature vectors as points in a high dimensional space. Now suppose, with each point in the input space we associate the probabilities of belonging to each of the possible classes. Then, given any input we will simply pick the class with the highest probability.

In effect, we have a classifier. Thus, we are modelling the probability distributions of the classes over input space.

Such probability distribution modelling classifiers give us more insight into the actual underlying phenomenon compared to classifiers that simply emit the class to which the input belongs. To see this, consider the problem of identifying horses, zebras and apes from photos. Given an input image, the non-probabilistic classifier simply emits a best guess as to whether its the photo of a horse or a zebra or an ape. The probabilistic classifier on the other hand, emits probabilities of the input image being a horse, zebra and ape. From them we might observe that for every horse image, the probability of a zebra is relatively high and vice versa, while for an ape image, the probabilities of both horse and zebra are low. This leads to the insight that horse and zebra looks somewhat similar while ape looks quite different from both of them.

Probabilistic classifiers typically treat the input points as random samples from some (usually unknown) multivariate probability distribution. The unknown probability distribution is expected to have distinct modes for each class. This means that at any point in the input space, the probability of one class is significantly higher than probability of other classes - if this condition is not satisfied, classification becomes error prone. This is why, in most practical classifiers, the input undergoes a non-linear transform first. The transform maps the input to a more classification friendly space and the probability distribution is modelled on this space rather than raw input.

Of course, we do not know that underlying probability distribution. All we have are the training data points. Sometimes we know the classes associated with the training data points (*supervised*) sometimes we do not even have that (*unsupervised*). Coming up with a probability distribution that is maximally compatible with the training data is the crux of the problem here.

To solve this problem, we postulate a parameterized probability distribution function from a known family of distributions. That family is chosen based on our physical knowledge of the problem. Then we estimate the parameters that would maximize the probability of occurrence of the training data. This is called *evidence maximization* - we are choosing the parameters such that the likelihood of observing the data that we did observe is maximized. We are essentially *fitting* a probabilistic model to the training data. It yields a probability distribution function over the feature space. At inference time, given an input vector, we can use the fitted (learnt) model to estimate the probabilities of the input vector belonging to each of our classes. Optionally, we can emit the class with maximum probability as the class to which the input belongs.

Viewed in this way, probabilistic point distributions are connected to machine learning and data science. Probabilistic point distributions become even more important in the context of unsupervised and minimally supervised learning, specifically *VariationalAuto Encoders (VAEs)*, *Generative Adversarial Networks (GANs)* , which we will study in later chapters.

This chapter is dedicated to studying probability distributions (including multivariates) and their role in machine learning.

In this chapter we will start from the basic idea of probabilities and histogram. From there we will move on to discrete probability distributions, continuous probability densities, multivariate probability distributions, joint probabilities. We will study the *Gaussian (Normal) distribution, uni and multivariate*, in much detail. We will also study some other distributions that are often used in machine learning like *Binomial, Multinomial, Categorical, Bernoulli, Beta and Dirichlet* distributions. We will study *entropy, cross entropy* and their significance in machine learning. All concepts will be presented in a machine learning centric way, with examples from machine learning applications.

An equally important goal of this chapter is to familiarize the reader with *PyTorch distributions, the statistical package of PyTorch* which we will use throughout the book. All the distributions discussed will be accompanied by code snippets from PyTorch distributions. TBD(Sujay): Please put a pointer to the public github for this chapter.

5.1 Probability - the classical frequentist view

Consider a mythical city called Statsville. Suppose we pick up a random adult inhabitant of Statsville. What are the chances of this person's height being above 6 ft? Below 3 ft? Between 5 ft. 5 inches and 6 ft? What are the chances of this person's weight being between 50 and 70 Kgs[16]1? Above 100 Kgs? What are the probabilities of the person's home exactly 6 Kms from the city center? What are the chances of the person's weight being in the 50-70 Kg range *and*

[16] The physicists would rather use the term *mass* here, but we have chosen to stick to the more folksy word weight

height in the 5 ft 5 inches to 6 ft range? What are the chances of the person's weight being above 90 Kgs and his/her home being within 5 Kms of the city center?

All these questions can be answered in the frequentist paradigm by adopting the following approach.

Count the size of the population belonging to the desired event (satisfying the criterion or criteria of interest). For instance, the number of Statsville adults whose heights are above 6 ft. Divide it by the total size of the population (here, number of adults in Statville). This is the probability (chance) of that criterion/criteria being satisfied.

Formally,

$$\begin{aligned}\text{probability of an event} &= \frac{\text{size of population belonging to that event}}{\text{total size of population}} \\ &= \frac{\text{Number of favorable outcomes}}{\text{Number of possible outcomes}}\end{aligned} \tag{5.1}$$

For instance, suppose there are 100, 000 adults in the city. Of them, 25, 000 are 6 ft. or above in height. Then the size of population satisfying event of interest (aka number of favorable outcomes) is 25, 000. The total size of population (aka number of possible outcomes) is 100, 000. Then

$$\text{probability of a random adult Statsville resident having height above 6ft} = \frac{\text{number of adult Statsville residents with height above 6 ft}}{\text{total number of adult Statsville residents}} = \frac{25000}{100000} = 0.25$$

Since the total population is always a superset of the population belonging to any event, the denominator is always greater than or equal to the numerator. Consequently, *probabilities are always lesser than or equal to* 1.

5.1.1 Random Variables

Whenever one talks of probability, a relevant question to ask is: probability of what? The simplest answer is the probability of "occurrence of an event". For example, in the previous subsection we discussed the probability of the event that the weight of an adult Statsville resident is less than 60 Kgs.

A little thought will reveal that an event always corresponds to a numerical entity of interest taking a particular value or lying in a particular range of values. This entity is called a random variable. For instance, the weights of adult Statsville residents can be a random variable. We talk of the probability of it being less than 60 Kgs etc. When predicting the performance of stock markets, the Dow Jones index maybe a random variable. We talk of the probability of this random variable crossing 19000 etc. When discussing the spread of a virus, the total number of infected people maybe a random variable. We talk of the probability of this being less than 2000 etc. The defining characteristic of a random variable is that, every allowed value (or range of values) is associated with a probability (of the random variable taking that value or value range). For instance, we may allow a set of only 3 weight ranges for adults of Statsville: (i) S_1: Below 60 Kgs (ii) S_2: Between 60 and 90 Kgs (iii) S_3: Above 90 Kgs. Then we can have a corresponding random variable X, representing quantized weight. It can take one of only 3 values: $X = 1$ – corresponding to weight in S_1, $X = 2$ - corresponding to weight in S_2 and $X = 3$ - corresponding to weight in S_3. Each value comes with a fixed

probability, e.g., $p(X = 1) = 0.25$, $p(X = 2) == 0.5$ and $p(X = 3) = 0.25$ respectively in the example from section 5.1. Such random variables, that takes value from a countable set are known as *discrete* random variables. Random variables can also be *continuous*. For instance, mass is a *continuous* random variable. It can take any value from near zero (mass of some sub-atomic particle) to trillions of tons (mass of black holes). For a continuous random variable X, we associate a probability with its value being in an infinitesimally small range, $p(x \leq X < x + \delta x)$ with $\delta x \to 0$. This is explained in more detail in section 5.6.

In this book, we will always use upper-case letters to denote random variables. Usually the same letter in lower case will refer to a specific value of the random variable. E.g., $p(X = x)$ or $p(X \in \{x, x + \delta x\})$ etc. Note that we will use the letter X to denote a random variable as well as a data set - this popular but confusing convention is rampant in the literature. The meaning should be obvious from context however.

5.1.2 Population Histograms

Histograms help us to visualize discrete random variables.

Let us continue with our example. We are only interested in 3 weight ranges for adults of Statsville:

(i) S_1: Below 60 Kgs (ii) S_2: Between 60 and 90 Kgs (iii) S_3: Above 90 Kgs.

Suppose the counts of Statsville adults in these weight ranges be as shown in the following table

Table 5.1: Frequency Table for weights of adults in the city of Statsville

S_1: **Below 60 Kgs**	S_2: **Between 60 and 90 Kgs**	S_3: **Above 90 Kgs**
25, 000	**50, 000**	**25, 000**

The same information can be visualized by the histogram 5.1.

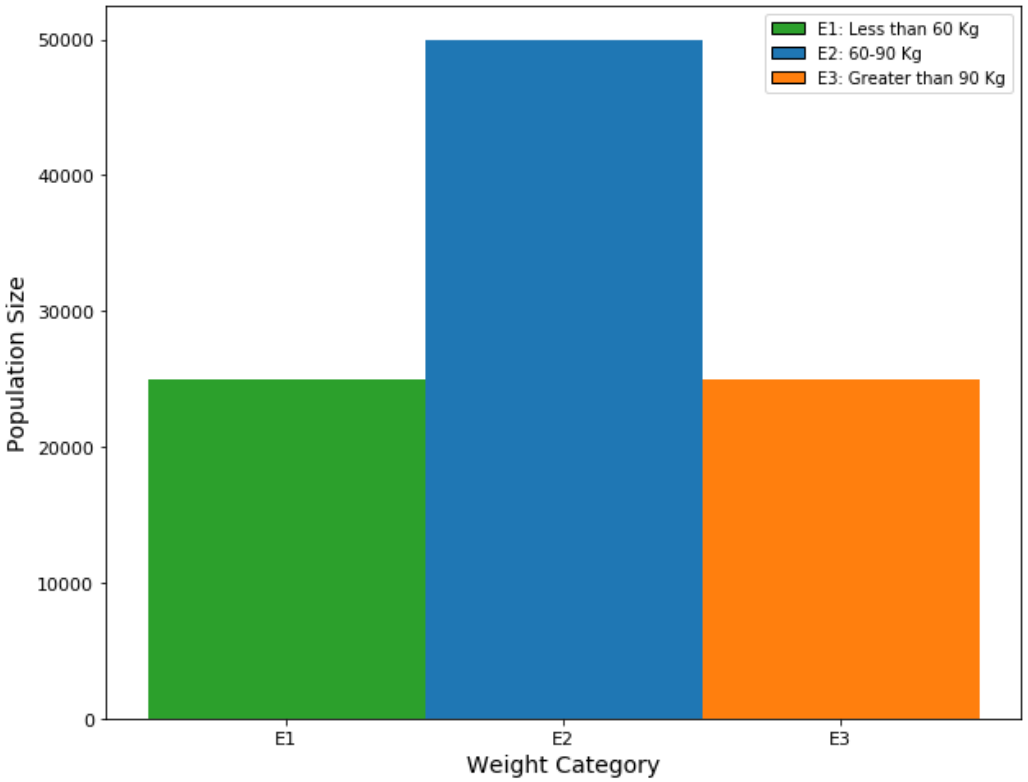


Figure 5.1: Histogram depicting weights of adults in Statsville, matching table 5.1

The X-axis of the histogram corresponds to possible values of the discrete random variable from section 5.1.1. The Y-axis shows the "size of the population" in the corresponding weight range.

There are $25,000$ people in the range S_1, $50,000$ people in the range S_2, 25000 people in the range S_3. Together, these categories account for the entire adult population of Statsville - every adult belongs to one category or other. This can be verified by adding the Y-axis values for all the categories - $25,000 + 50,000 + 25,000 = 100,000$ - the adult population of Statsville.

5.2 Probability Distributions

Histogram 5.1 or its equivalent, Table 5.1 can easily be converted to probabilities, as shown in Table 5.2. The table shows the probabilities corresponding to allowed values of the discrete random variable X representing the quantized weight of a randomly chosen adult resident of Statsville.

Table 5.2 represents what is formally known as a probability distribution - a mathematical function, which takes a random variable as input and outputs the probability of it taking any allowed value. It must be defined over all possible values of the random variable.

Table 5.2: Probability Distribution for quantized weights of Statsville adults

S_1: **Below** 60 **Kgs**	S_2: **Between** 60 **and** 90 **Kgs**	S_3: **Above** 90 **Kgs**
$p(X = 1) = 25{,}000/100{,}000 = 0.25$	$p(X = 2) = 50{,}000/100{,}000 = 0.5$	$p(X = 3) = 25{,}000/100{,}000 = 0.25$

Note that the set of ranges $\{S_1, S_2, S_3\}$ is exhaustive in the sense that all possible values of X belongs to one range or other, we cannot have a weight that does not belong to any of them. In set theoretical terms, the union of these ranges, viz., $S_1 \cup S_2 \cup S_3$ covers a space that contains there entire population (all possible values of X)[17]

Also, the ranges are mutually exclusive, in the sense that any given observation of X can belong to only a single range, never more. In set theoretic terms, the intersection of any pair of ranges is null, i.e., $S_1 \cap S_2 = S_1 \cap S_3 = S_2 \cap S_3 = \phi$ [18].

For a set of exhaustive and mutually exclusive events, the function yielding the probabilities of these events is a probability distribution.

The probability distribution in this tiny example comprises three probabilities:

$P(X = 1) = 0.25$ $P(X = 2) = 0.5$, $P(X = 3) = 0.25$.

This is shown in Fig. 5.2 which is a 3 point graph.

[17] The set theoretic operator $\cup$ denotes set union.
[18] The set theoretic operator $\cap$ denotes set intersection

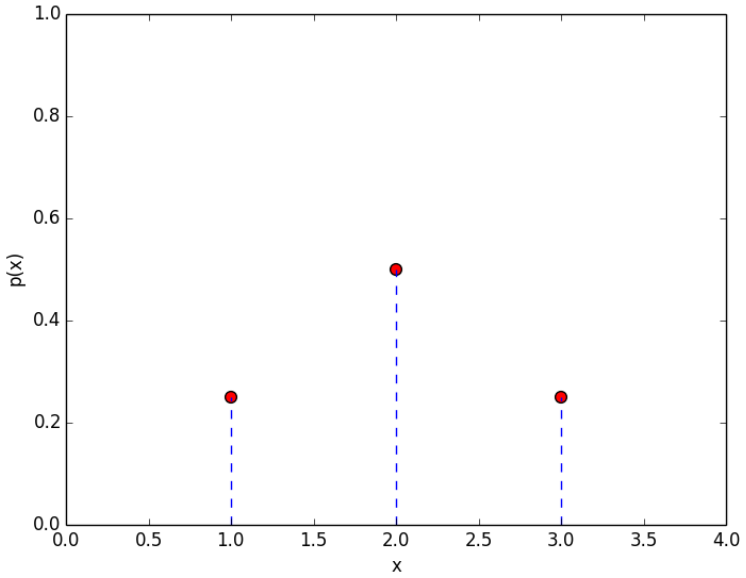


Figure 5.2: Probability Distribution graph for weights of adults in Statsville, matching table 5.2. Event $E_1 \equiv X = 1 \Rightarrow$ weight in range S_1, Event $E_2 \equiv X = 2 \Rightarrow$weight in range S_2, Event $E_3 \equiv x = 3 \Rightarrow$ weight in range S_3

5.3 Impossible and certain events, Sum of probabilities of exhaustive, mutually exclusive events, Independent events

In this section we will briefly touch upon some basic notions related to probability theory.

5.3.1 Probabilities of Impossible and Certain Events

The probability of an impossible event is zero (e.g., the probability that sunrise will happen in the West). Probability of an event that occurs with certitude is 1 (e.g., the probability that sunrise will happen in the East). Improbable events have low, but non-zero, probabilities (e.g., the probability of this author beating Roger Federer in competitive tennis) may be 0.001 or something. Highly probable events have probabilities close to, but not exactly equal to 1 (e.g., the probability of Roger Federer beating this author in competitive tennis) maybe 0.999.

5.3.2 Exhaustive and mutually exclusive events

Consider the events E_1, E_2, E_3 which corresponds to the quantized weight of a Statsville adults from section 5.2 belonging to the ranges S_1, S_2 and S_3 respectively (equivalently, E_1 is the event corresponding to $X = 1$, E_2 is the event corresponding to $X = 2$, E_3 is the event corresponding to $X = 3$) . They are exhaustive - their union covers the entire population space. This means, all quantized weights of Statsville adults will belong to one or other of the ranges S_1, S_2, S_3. Also, they are mutually exclusive, meaning their mutual intersections are null. This means, no

member of the population can belong to more than one of the ranges. E.g., if the weight belongs to S_1 it cannot belong to S_2 or S_3. For such events, the following holds true.

> *Sum of probabilities of mutually exclusive events yields the probability of one or the other of them occurring.*

For instance, for events E_1, E_2, E_3:

$$p\ (E_1 \text{ or } E_2) = p\ (E_1) + p\ (E_2) \tag{5.2}$$

THE SUM RULE

The sum of probabilities of exhaustive, mutually exclusive set of events is always 1, e.g.,

$$p\ (E_1) + p\ (E_2) + p\ (E_3) = p\ (E_1 \text{ or } E_2 \text{ or } E_3) = 1$$

This is intuitively obvious. We are merely saying that *we can say with certainty that either* E_1 *or* E_2 *or* E_3 *will occur*.

In general, given a set of exhaustive, mutually exclusive events E_1, E_2, ...E_n,

$$\sum_{i=1}^{i=n} p\ (E_i) = 1 \tag{5.3}$$

5.3.3 5.3.3 Independent Events

Consider the two events: $E_1 \equiv$ "weight of an adult inhabitant of Statsville is below 60 Kgs" and $G_1 \equiv$ "home of an adult inhabitant of Statsville is within 5 Kms of the city center". These events do not influence each other at all. Knowledge that a member of the population weighs less than 60 Kgs does not reveal anything about his/her home distance from city center and vice versa. We say, E_1 and G_1 are independent events.

5.4 Joint Probabilities and their distributions

Given an adult Statsville resident, let E_1 be, as before, the event that his/her weight is below 60Kgs. The corresponding probability is $p\ (E_1)$. Also, let G_1, be the event that the distance of his/her home from city center is less than 5 Kms. The corresponding probability is $p\ (G_1)$.

Now consider the probability that a resident weights below 60 Kg **and** the distance of his/her home from city center is less than 5 Kms. This probability, denoted $p\ (E_1, G_1)$ is the so called joint probability.

> *Formally, the joint probability of a set of events is the probability of all those events occurring together.*

The Product Rule: *Joint probability of independent events can be obtained by multiplying the individual probabilities.*. Thus, for the current example, $p\ (E_1, G_1) = p\ (E_1)\ p\ (G_1)$.

Let us now study joint probabilities in a bit more detail, with a slightly more elaborate example. We have consolidated the weight categories, the corresponding population and probability distributions in Table 5.3 for quick reference. Similarly, we quantize the distance of home from city center into 3 ranges: $D_1 \equiv$ below 5 Kms, $D_2 \equiv$ between 5 and 15 Kms and $D_3 \equiv$ above 15ms.

Table 5.3: Population Probability Distribution table for weights of for adult residents of the city of Statsville. E_1, E_2, E_3 are exhaustive, mutually exclusive events, $p\ (E_1) + p\ (E_2) + p\ (E_3) = 1$.

Below 60 Kgs (range S_1)	Between 60 and 90 Kgs (range S_2)	Above 90 Kgs (range S_3)
Event $E_1 \equiv weight \in S_1$	Event $E_2 \equiv weight \in S_2$	Event $E_3 \equiv weight \in S_3$
population size = 25, 000	population size = 50, 000	population size = 25, 000
$p(E_1) = 25{,}000/100{,}000 = 0.25$	$p(E_2) = 50{,}000/100{,}000 = 0.5$	$p(E_3) = 25{,}000/100{,}000 = 0.25$

Table 5.4 shows the corresponding population and probability distribution.

Table 5.4: Population Probability Distribution table for distance of home from city center for adult residents of the city of Statsville. G_1, G_2, G_3 are exhaustive, mutually exclusive events, $p\ (G_1) + p\ (G_2) + p\ (G_3) = 1$.

Below 5 Kms (range D_1)	Between 5 and 15 Kms (range D_2)	Above 15 Kms (range D_3)
Event $G_1 \equiv distance \in D_1$	Event $G_2 \equiv distance \in D_2$	Event $G_3 \equiv distance \in D_3$
population size = 20, 000	population size = 60, 000	population size = 20, 000
$p\ (G_1) = 20{,}000/100{,}000 = 0.20$	$p\ (G_2) = 60{,}000/100{,}000 = 0.6$	$p\ (G_3) = 20{,}000/100{,}000 = 0.20$

The joint probability distribution of the events $\{E_1, E_2, E_3\}$and $\{G_1, G_2, G_3\}$is shown in Table 5.5.

Table 5.5: Joint Probability Distribution of independent events. Sum of all elements in the table is 1.

	Below 60Kgs (E_1)	Between 60 and 90Kgs (E_2)	Above 90Kgs (E_3)
Below 5 Kms (G_1)	$p\ (E_1, G_1)$ $= 0.25 \times 0.2 = 0.05$	$p\ (E_2, G_1)$ $= 0.5 \times 0.2 = 0.1$	$p\ (E_3, G_1)$ $= 0.25 \times 0.2 = 0.05$
Between 5 and 15Kms (G_2)	$p\ (E_1, G_2)$ $= 0.25 \times 0.6 = 0.15$	$p\ (E_2, G_2)$ $= 0.5 \times 0.6 = 0.3$	$p\ (E_3, G_2)$ $= 0.25 \times 0.6 = 0.15$
Above 15 Kms (G_3)	$p\ (E_1, G_3)$ $= 0.25 \times 0.2 = 0.05$	$p\ (E_2, G_3)$ $= 0.25 \times 0.2 = 0.1$	$p\ (E_3, G_3)$ $= 0.25 \times 0.2 = 0.05$

Notes on Table 5.5

We can make the following statements:

- The sum total of all elements in Table 5.5 is 1. In other words, $p(E_i, G_j)$

  is a proper probability distribution - indicating the probabilities of event E_i and event G_j occurring together.

  Here $(i, j) \in \{1, 2, 3\} \times \{1, 2, 3\}$.

  The symbol $\times$ here denotes the Cartesian product. The Cartesian product of two sets $\{1, 2, 3\} \times \{1, 2, 3\}$ is the set $\{(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3), (3, 1), (3, 2), (3, 3)\}$.

- $p(E_i, G_j) = p(E_i)\, p(G_j) \quad \forall (i, j) \in \{1, 2, 3\} \times \{1, 2, 3\}$[19]. This is because the events are independent.

In general, given a set of independent events $E_1, E_2, \ldots E_n$, the joint probability $p(E_1, E_2, \ldots E_n)$ of all the events occurring together is the product of their individual probabilities of occurring

$$p(E_1, E_2, \cdots, E_n) = p(E_1)\, p(E_n) \cdots p(E_1) = \prod_{i=1}^{i=n} p(E_i) \tag{5.4}$$

5.4.1 Marginal Probabilities

Suppose we did not have the individual probabilities $p(E_i)$ and $p(G_j)$. All we have is the joint probability distribution, i.e., Table 5.5. Can we find the individual probabilities from them? If yes, how?

To answer this question, consider a particular row or column of Table 5.5, say the top row. In this

row, the E values iterate over all possibilities (the entire space of E's). The G though, is fixed at

G_1. If G_1 is to occur, there are only 3 possibilities, it occurs with E_1,E_2 or E_3. The corresponding joint probabilities are $p(E_1, G_1)$, $p(E_2, G_1)$ and $p(E_3, G_1)$. If we add them, we have considered all situations under which G_1 can occur, i.e., we have obtained the probability of event G_1 occurring.

Thus, $p(G_1)$ can be obtained by adding all probabilities in the row corresponding to G_1 and writing it in the margin (this is why it is called "marginal" probability). Similarly, by adding all the probabilities in the middle column, we obtain the probability $p(E_2)$ and so forth. Table 5.6 shows

[19] The symbol $\forall$ indicates "for all"

Table 5.6: Joint Probability Distribution with marginal probabilities shown

	Below 60 **Kgs** (E_1)	**Between** 60 **and** 90 **Kgs** (E_2)	**Above** 90 **Kgs** (E_3)	**Marginals for** G**'s**
Below 5 Kms (G_1)	$p(E_1, G_1) = 0.25 \times 0.2 = 0.05$	$p(E_2, G_1) = 0.5 \times 0.2 = 0.1$	$p(E_3, G_1) = 0.25 \times 0.2 = 0.05$	$p(G_1) = 0.05 + 0.1 + 0.05 = 0.2$
Between 5 and 15 Kms (G_2)	$p(E_1, G_2) = 0.25 \times 0.6 = 0.15$	$p(E_2, G_2) = 0.5 \times 0.6 = 0.3$ $p(E_3, G_2) = 0.25 \times 0.6 = 0.15$	$p(G_2) = 0.15 + 0.3 + 0.15 = 0.6$	
Above 15 Kms (G_3)	$p(E_1, G_3) = 0.25 \times 0.2 = 0.05$	$p(E_2, G_3) = 0.25 \times 0.2 = 0.1$	$p(E_3, G_3) = 0.25 \times 0.2 = 0.05$	$p(G_3) = 0.05 + 0.1 + 0.05 = 0.2$
Marginals for E**'s**	$p(E_1) = 0.05 + 0.15 + 0.05 = 0.25$	$p(E_2) = 0.1 + 0.3 + 0.1 = 0.5$	$p(E_3) = 0.05 + 0.15 + 0.05 = 0.25$	

Table 5.5 updated with marginal probabilities. In general, given a set of exhaustive, mutually exclusive events E_1, E_2, ...E_n, another event G along with joint probabilities $p(E_1, G)$, $p(E_2, G)$,...$p(E_n, G)$,

$$p(G) = \sum_{i=1}^{i=n} p(E_i, G) \tag{5.5}$$

In some sense, by summing over all possible values of E_is we are factoring out the Es. This is because Es are mutually exclusive and exhaustive, summing over them results in a certain event which gets factored out (remember probability of a certain event is 1).

5.4.2 Dependent Events and their Joint Probability Distribution

So far, the events we have considered jointly are "weights" and "distance of home from city center".

These are independent of each other - their joint probability is the product of their individual probabilities. Now, let us study a different situation, when the variables are connected, knowing the value of one actually helps us in predicting the other. For instance, weight and height of adult residents of Statsville, These are not independent, typically taller people weigh more and vice versa. As usual, we will create a toy example to understand the idea. We will quantize heights into 3 ranges, $H_1 \equiv$ Below 5 ft. 5 inches, $H_2 \equiv$ between 5 ft. 5 inches and 6 ft. and $H_3 \equiv$ above 6 ft..

Let z be the random variable corresponding to height. We have 3 possible events with respect to height, $F_1 \equiv z \in H_1$, $F_2 \equiv z \in H_2$ and $F_3 \equiv z \in H_3$. The joint probability distribution of height and weight is shown in Table 5.7.

Table 5.7: Joint Probability Distribution of dependent events

	Below 60 **Kgs** (E_1)	**Between** 60 **and** 90 **Kgs** (E_2)	**Above** 90 **Kgs** (E_3)
Below 5 **ft.** 5 **inches** (F_1)	$p\ (E_1, F_1) = 0.25$	$p\ (E_2, F_1) = 0$	$p\ (E_3, F_1) = 0$
Between 5 **ft.** 5 **inches and** 6 **ft.** (F_2)	$p\ (E_1, F_2) = 0$	$p\ (E_2, F_2) = 0.5$	$p\ (E_3, F_2) = 0$
Above 6 (F_3)	$p\ (E_1, F_3) = 0$	$p\ (E_2, F_3) = 0$	$p\ (E_3, F_3) = 0.25$

Notes on Table 5.7:

- The sum total of all elements in Table 5.7 is 1. In other words, $p(E\text{i}, F\text{j})$ is a proper probability distribution - indicating the probabilities of event E_i and event F_j occurring together. Here $(i, j) \in \{1, 2, 3\} \times \{1, 2, 3\}$.
- $p(E\text{i}, F\text{j}) = 0$ if $i \neq j\ \forall\ (i, j) \in \{1, 2, 3\} \times \{1, 2, 3\}$. This essentially means the events are perfectly correlated, occurrence of E_1 implies occurrence of F_1 and vice versa, occurrence of E_2 implies occurrence of F_2 and vice versa, occurrence of E_3 implies occurrence of F_3 and vice versa. In other words, every adult resident of Statsville who weighs below 60 Kgs is also below 5 ft. 5 inches in height etc[6].

5.5 Geometrical View: Sample point distributions for dependent and independent variables

Let us take a graphical view of the point distributions corresponding to Table 5.5 and Table 5.7. We will observe that there is a fundamental difference between how point distributions for independent and dependent variables look and it is connected to Principal Component Analysis (PCA) and dimensionality reduction we studied in section 4.3. We also provide the python numpy code to sample from the joint distributions and create the 2D point population shown.

We have 3 weight related events, E_1, E_2, E_3, and 3 distance events G_1, G_2, G_3. Hence the joint distribution has 3 × 3 = 9 possible events(E_i, G_j), $\forall\ (i, j) \in \{1, 2, 3\} \times \{1, 2, 3\}$ as shown in Table 5.5. To visualize the point distribution, we have drawn 1000 samples of joint events. Owing to the relatively high probability, the event (E_2, G_2) will likely occur the maximum number of times. On the other hand, events (E_1, G_1), (E_3, G_1), (E_1, G_3) etc will occur relatively infrequently.

Now, each of these 9 events is represented by a small 1.0×1.0 sized rectangle (bucket for the joint event) around it. The joint event sample is placed at a random location somewhere within its bucket (i.e., all points within bucket have an equal probability of getting selected).

Graphical views of the point distribution for the independent and non-independent joint variable are shown in Figure 5.3 and Figure 5.4 respectively. Notice that the concentration of points is higher inside high probability buckets and vice versa.

We see that the sample point distribution for the independent events is spread symmetrically all over the domain, while that for the dependent events is spread narrowly around a particular direction.

This holds true in general, and for higher dimensions too. This is the mental picture to have with respect to independent versus non-independent point distributions.

This is kind of intuitive, if we sample independent events (uncorrelated) - all possible combinations of events are equally likely to occur. This precludes concentration of points in a small region of the space – $\{E_1, G_1\}$, $\{E_1, G_2\}$, $\{E_1, G_3\}$, ...$\{E_3, G_3\}$ are all equally likely. In other words, the joint probability is diffused all over the population space (see Fig. 5.3 for instance). On the other hand, if the events are correlated, the joint probability will be concentrated in certain regions of the joint space and near zero values elsewhere. For instance, see Fig. 5.4 where (E_1, G_1), (E_2, G_2), (E_3, G_3) are far more likely than the other combinations.

If this did not remind you of PCA (section 4.3) you should go and re-read that section. Dependent events such as the one shown in Fig. 5.4 is a good candidate for dimensionality reduction – the two dimensions are essentially carrying the same information - if we know one we can derive the other. We could drop one of the highly correlated dimensions without compromising anything.

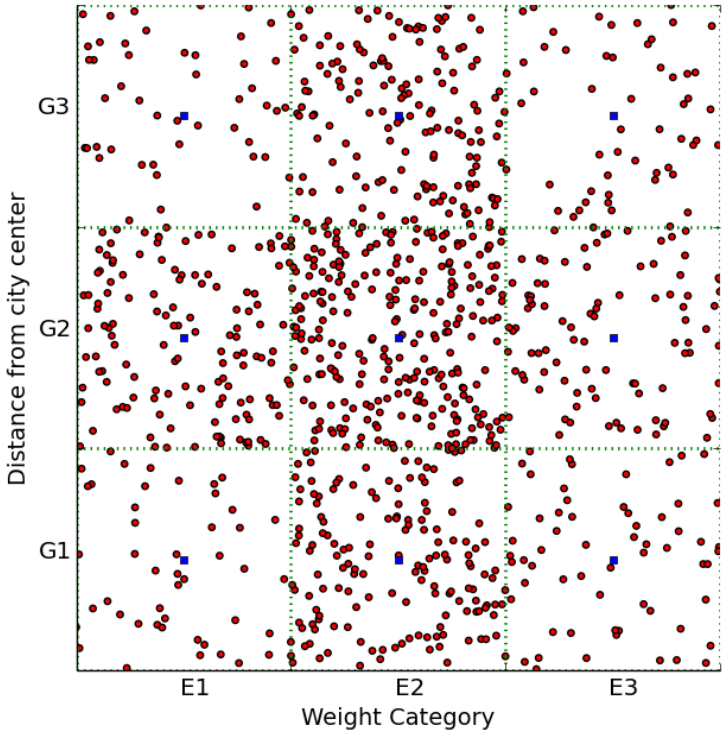


Figure 5.3: Graphical Visualization of the joint probability distribution from Table 5.5 (independent events). Green rectangles depict the buckets. Blue dots at center of buckets correspond to events. The central bucket event (E_2, G_2) corresponds to the highest probability event and has the biggest concentration of points. The 4 corner buckets correspond to lowest probability events and have lower density of points. The events are independent and the points are symmetrically spread out, not clustering close to any straight line. Not suitable for PCA based dimensionality reduction.

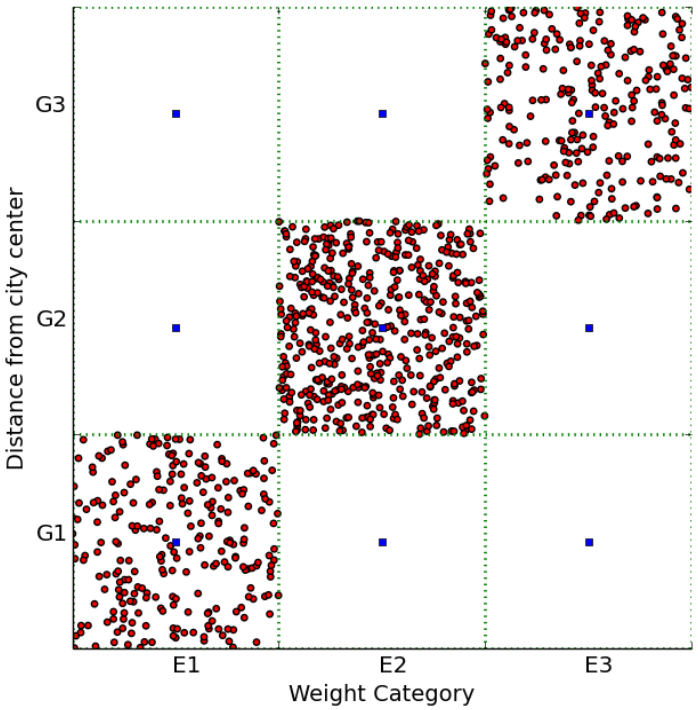


Figure 5.4: Graphical Visualization of the joint probability distribution from Table 5.7 (non-independent events). Green rectangles depict the buckets. Blue dots at center of buckets correspond to events. The central bucket event (E_2, G_2) corresponds to the highest probability event and has the biggest concentration of points. Off diagonal buckets have zero probabilities and no points. The events are correlated (not independent). The points are clustered around the diagonal. Suitable for PCA based dimensionality reduction.

5.5.1 Python Numpy code to draw random samples from a discrete joint probability distribution

Listing 5.1: Snippet from python numpy code to draw point distributions. Buckets are selected according to specified probability distribution using numpy choice function. Point coordinates within bucket are chosen via uniform random sampling using numpy uniform function.

```
bucket_centers = np.array(list(itertools.product(range(3),
                                                 range(3))))
                 + [0.5, 0.5]
bucket_probabilities_indep = np.array([0.05, 0.1, 0.05,
                                       0.15, 0.3, 0.15,
                                       0.05, 0.1, 0.05])
bucket_probabilities_dep = np.array([0.25, 0.0, 0.0,
                                     0.0, 0.5, 0.0,
                                     0.0, 0.0, 0.25])
bucket_probabilities = bucket_probabilities_indep

bucket_indices = \
    np.random.choice(np.array(range(len(bucket_centers))),
                     size=1000, p=bucket_probabilities)
x = [np.random.uniform(low=bucket_centers[i][0] - 0.5,
                       high=bucket_centers[i][0] + 0.5)
     for i in bucket_indices]
y = [np.random.uniform(low=bucket_centers[i][1] - 0.5,
                       high=bucket_centers[i][1] + 0.5)
     for i in bucket_indices]
```

use Numpy cartesian product to create bucket centers

set probabilities for bucket centers as desired

Choose 1000 sample bucket indices as per specified bucket probabilities

uniform random sample within small rectangle around bucket center

5.6 Continuous Random Variables and Probability Density

So far we have spoken about quantities like weight, height and distance and their probabilities - in terms of discrete buckets. For instance, we have quantized weight into 3 buckets, below 60 Kgs, between 60 and 90 Kgs and above 90 Kgs and assigned probabilities to each bucket. What if we want to know probabilities at a more granular level, say, $0-10$ Kgs, $10-20$ Kgs, $20-30$Kgs etc? Well, we will have to create more buckets. Each bucket will cover a narrower range of values (i.e., a smaller portion of the population space), but there will be more of them. We will then (following frequentist approach) count the number of adult Statsvilleans in each bucket, divide that by the total population size and call that probability of belonging to that bucket. What if we want even further granularity? Well, we will create even more buckets each of which covers an even smaller portion of the population space. In the limit, we will have an infinite number of buckets, each one covering an infinitesimally small portion of the population space. Together they still cover the population space - very large number of very

small pieces can cover arbitrary regions. At this limit, the probability distribution function is called a probability density function.

Formally *a probability density function* $p(x)$ *for a continuous random variable* X*, is defined as the probability that* X *lies between* x *and* $x + \delta x$*, i.e.,*

$$p(x) = probability\ (x \leq X < x + \delta x)$$ [20]

There is a bit of theoretical nuance here. We are saying $p(x)$ is the probability of the random variable X lying between x and $x + \delta x$. This is not exactly same as saying that $p(x)$ is the probability that X is equal to x. But, δx being infinitesimally small, they mean the same thing, practically (though not mathematically). Occasionally, we will abuse terms for the sake of simplicity.

Consider the set of events $E = \{x \leq X < x + \delta x\}$ for all possible values of x. All possible values of x will range from negative infinity to infinity, i.e., $x \in [-\infty, \infty]$. There are infinite number of these events and each of them is infinitesimally narrow, but together they do cover the entire domain $x \in [-\infty, \infty]$. In other words, they are exhaustive. Also, they are mutually exclusive because x cannot belong to more than one of them at the same time. They are continuous counterparts of the discrete events E_1, E_2, E_3 we have seen before.

The fact that the set of events $E = \{x \leq X < x + \delta x\}$ in continuous space are exhaustive and mutually exclusive means we can apply equation 5.3 - only the sum will become integral as the variable is continuous. Thus we have the sum rule in continuous domain

$$\int_{x=-\infty}^{\infty} p(x)\, dx = 1 \tag{5.6}$$

Equation 5.6 is the continuous analog of equation 5.3. It is really saying that we can say with certainty that x lies somewhere in the interval $-\infty$ to ∞.

The random variable can also be multi-dimensional (i.e., vector). Then the probability distribution density function will be denoted as $p(\vec{x})$.

The sum rule still holds

[20] It is slightly unfortunate that the typical symbol for a random variable, viz., X, collides with that for a dataset (collection of data vectors), also X. But the context is usually enough to tell them apart.

$$\int_{\vec{x} \in D} p(\vec{x})\, d\vec{x} = 1 \tag{5.7}$$

where D is the domain of $\vec{x}$, i.e., the space containing all possible values of the vector $\vec{x}$. For instance, 2D vector $\begin{bmatrix} x \\ y \end{bmatrix}$ has the XY plane as its domain. It should be noted that the integral in quation 5.7 is a *multi-dimensional* integral (e.g., for 2D $\vec{x}$ it will be $\iint_{\vec{x} \in D} p(\vec{x})\, d\vec{x} = 1$).[21]

One might remember from elementary integral calculus that equation 5.6 actually corresponds to the area under the curve for $p(x)$ (or $p(\vec{x})$).

In higher dimensions, equation 5.7 actually corresponds to the volume under the hyper-surface for $p(\vec{x})$.

Thus, *the total area under a univariate probability density curve is always one*.

In higher dimensions, *the volume under the hyper-surface for a multivariate probability density functionis always one*.

5.7 Properties of distributions - Expected Value, Variance and Covariance

It was stated in the introduction to this chapter that machine learning models, including unsupervised models like variational auto encoders are often developed by fitting a distribution from a known family to the available training data. Thus, we will postulate a parameterized distribution from a known family and estimate the exact parameters that best fit the training data. Now most distribution families are parameterized in terms of some intuitive properties like mean, variance etc. Understanding these concepts and their geometric significance is essential to understanding the models based upon them. In this section, we will explain a few properties common to all distributions.

Later, when we study individual distributions, we will connect them to the parameters of those distributions. We will also show how to programmatically obtain the values of these for each individual distribution via the PyTorch distributions package.

5.7.1 Expected Value aka Mean

If one samples a random variable with a given distribution many many times, and take average of the sampled values , what value does one expect to end up with?

Well, the average will certainly be closer to the values that have higher probabilities (as these will show up more often during sampling). If we sample enough number of times, for a

[21] for cleanliness of notation, we will usually use a single integral sign to denote multi-dimensional integrals. The vector sign in the domain (e.g., $\vec{x} \in D$), as well the vector sign in $d\vec{x}$ indicates the multiple dimensions.

given probability distribution, this average will always settle down to a fixed value for that distribution - this is the *expected value* of the distribution.

Formally, given a discrete distribution where a discrete random variable X can take any value from the sets $\{x_1, x_2, \ldots x_n\}$[22] with respective probabilities $\{p(x_1), p(x_2), \ldots, p(x_n)\}$, the expected value is given by the formula

$$\mathbb{E}(X) = \sum_{i=1}^{n} p(x_i)\, x_i \tag{5.8}$$

Equation 5.8 is simply saying the *average value of a large number of samples drawn from the distribution is the probability weighted sum of all possible sample values*.

Where is the probability weight coming form? Why isn't this simply the average of all possible values? Well, when we sample, the higher probability values are going to appear more frequently than the lower probability values, so of course the average will be pulled closer to the higher probability values.

The idea extends over to multivariate random variables. Given a discrete distribution where a discrete multi-dimensional random variable X can take any value from the sets $\{\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_n\}$ with

respective probabilities $\{p(\vec{x}_1), p(\vec{x}_2), \ldots, p(\vec{x}_n)\}$, the expected value is given by the formula

$$\mathbb{E}(X) = \sum_{i=1}^{n} p(\vec{x}_i)\, \vec{x}_i \tag{5.9}$$

And the idea extends to continuous random variables too, where the sum gets replaced by an integral. Expected value of a continuous random variable X that takes values from $-\infty$ to ∞, i.e., $x \in \{-\infty, \infty\}$.

$$\mathbb{E}(X) = \begin{cases} \int_{x=-\infty}^{\infty} x\, p(x)\, dx \Rightarrow \text{for continuous univariate distributions} \\ \int_{\vec{x} \in D} \vec{x}\, p(\vec{x})\, d\vec{x} \Rightarrow \text{for continuous multivariate distributions} \end{cases} \tag{5.10}$$

EXPECTED VALUE AND CENTER OF MASS IN PHYSICS

In physics, we have the concept of center of mass or centroid. If we have a set of points, each with its own mass, then the entire point set can be replaced by a single point. This point is called the centroid. The position of the centroid is the weighted average of the positions of the individual points, weighted by their individual masses. If we mentally think of the probabilities of individual points as masses, the notion of expected value in statistics corresponds to the notion of centroid in physics.

---

[22] By popular convention, we use upper case to name the random variable, e.g., X, and lower case for the actual values it takes, e.g., x.

EXPECTED VALUE OF AN ARBITRARY FUNCTION OF A RANDOM VARIABLE

So far, we have seen expected value of the random variable itself. The notion can be extended to functions of the random variable.

The expected value of a function of a random variable is the probability weighted sum of the values of that function at all possible values of the random variable. Formally

$$\begin{aligned}
\mathbb{E}\left(f\left(X\right)\right) &= \sum_{i=1}^{n} f\left(x_i\right) p\left(x_i\right) \Rightarrow \text{for discrete univariate distributions} \\
\mathbb{E}\left(f\left(X\right)\right) &= \sum_{i=1}^{n} f\left(\vec{x}_i\right) p\left(\vec{x}_i\right) \Rightarrow \text{for discrete multivariate distributions} \\
\mathbb{E}\left(f\left(X\right)\right) &= \int_{x=-\infty}^{\infty} f\left(x\right)\ p\left(x\right) dx \Rightarrow \text{for continuous univariate distributions} \\
\mathbb{E}\left(f\left(X\right)\right) &= \int_{\vec{x}\in D} f\left(\vec{x}\right)\ p\left(\vec{x}\right) d\vec{x} \Rightarrow \text{for continuous multivariate distribution}
\end{aligned} \tag{5.11}$$

EXPECTED VALUE AND DOT PRODUCT

In equation 2.6 we studied the dot product between two vectors. Further, in section 2.5.6 we saw that dot product between two vectors measures the agreement between the two vectors. If both of them point in the same direction, the dot product is larger. In this section we will show that the expected value of a function of a random variable can be viewed as a dot product between a vector representing the probability and the another vector representing the function itself.

First let us consider the discrete case. Our random variable can take values x_i, $i \in \{1, n\}$. Now,

imagine a vector $\vec{f} = \begin{bmatrix} f\left(x_1\right) \\ f\left(x_2\right) \\ \cdots \\ f\left(x_n\right) \end{bmatrix}$ and a vector $\vec{p} = \begin{bmatrix} p\left(x_1\right) \\ p\left(x_2\right) \\ \cdots \\ p\left(x_n\right) \end{bmatrix}$. From equation 5.11 we see that the expected value of the function $\mathbb{E}\left(f\left(X\right)\right)$ of random variable X is same as $\vec{f}^{T}\vec{p} = \vec{f} \cdot \vec{p}$. This will be high when $\vec{f}$ and $\vec{p}$ are aligned - thus, the expected value of the function of random variable is high when the high function values coincide with high probabilities of the random variable and vice versa.

In the continuous case, these vectors have infinte number of components, the summation is replaced by an integral, but the idea stays the same.

EXPECTED VALUE OF LINEAR COMBINATIONS OF RANDOM VARIABLES

Expected value is a linear operator. This means, expected value of a linear combination of random variables is a linear combination (with same weights) of expected values of the random variables.

Formally

$$\mathbb{E}\left(\alpha_1 X_1 + \alpha_2 X_2 \ldots \alpha^n X^n\right) = \alpha_1 \ \mathbb{E}\left(X_1\right) + \alpha_2 \ \mathbb{E}\left(X_2\right) + \ldots \alpha^n \ \mathbb{E}\left(X^n\right) \tag{5.12}$$

5.7.2 Variance, Covariance, Standard Deviation

If we draw a very large number of samples from a given point distribution, we often like to know the spread of the point set. Spread is not merely a matter of measuring the largest distance between two points in the distribution. Rather, we want to know how densely packed the points are. If most of the points fit within a very small ball, then even if one or two points very far away from the ball, we would call that a small spread or high packing density.

Why is it important in machine learning? Lets start with a few informal examples. For instance, if we discover the points are tightly packed in a small region around a single point, we may want to replace the whole distribution with that point without causing much error. Or, if the points are packed tightly around a single straight line, we may replace the entire distribution with that line. This not only gives us a simpler (lower dimensional) representation, it often leads to a view of the data that is more amenable to understanding the "big picture. This is because, usually small variations about a particular point or direction is typically caused by noise while large variations are caused by meaningful things. By eliminating small variations and focusing on the large ones, we capture the main information content" [23]. This was the basic idea behind PCA and dimensionality reduction which we have seen in section 4.3.

Variance or its square root standard deviation measures how densely packed around the expected value *the points in the distribution are, aka the spread of the point distribution.* Formally, the variance of a probability distribution is defined as

$$var\left(X\right) = \begin{cases} \sum_{i=1}^{n} \left(x_i - \mu\right)^2 p\left(x_i\right) \Rightarrow \text{ for a discrete n point distribution} \\ \int_{x=-\infty}^{\infty} \left(x - \mu\right)^2 p\left(x\right) dx \Rightarrow \text{ for a continuous distribution} \end{cases} \tag{5.13}$$

By comparing equation 5.13 to equation 5.10 and equation 5.11 we see that variance is the expected value of the distance, $(x - \mu)^2$, of sample points x, from the mean μ. So if the more probable (more frequently occurring) sample points lie within a short distance of the mean the variance is small and vice versa. That is to say, variance measures how tightly packed the points are around the mean.

COVARIANCE - VARIANCE IN HIGHER DIMENSIONS

Extending the notion of expected value from the univariate case to multivariate case was straightforward.In the univariate case, we took probability weighted average of a scalar quantity, x. Theresulting expected value was a scalar, $\mu = \int_{x=-\infty}^{\infty} x \ p\left(x\right) dx$. In the

[23] 10This could be a possible explanation of why older people are often better at forming big picture view - there is too much data in their head and perhaps fewer neurons to retain everything in entirety. Their brain performs dimensionality reduction.

multivariate case, we took probability weighted average of a vector quantity, $\vec{x}$. The resulting expected value was a vector, $\vec{\mu} = \int_{\vec{x} \in D} \vec{x}\ p(\vec{x})\, d\vec{x}$.

Extending the notion of variance to the multivariate case is not so straightforward. This is because one can traverse the multidimensional random vector's domain (the space over which the vector is defined) along infinite number of possible directions - think how many possible directions one can walk along on a 2D plane - and the spread/packing-density can be different for each direction.

For instance, in Fig. 5.4, the spread along the main diagonal is much larger than the spread along direction perpendicular to it.

Covariance of a multidimensional point distribution is a matrix that allows us to easily measure the spread or packing density along any desired direction. It also allows us to easily figure out the direction along which maximum spread occurs and what that spread is. The main idea appears below.

Consider a multivariate random variable X which takes vector values $\vec{x}$. Let $\hat{l}$ be an arbitrary direction[24] along which one wants to measure the packing density of X. We studied in sec 2.5.2 and section 2.5.6 that the dot product of $\vec{x}$ along the direction $\hat{l}$, i.e., $\vec{x}^T \hat{l}$ is the projection or component ("effective value") of x along $\hat{l}$. Thus the spread or packing density of the random vector $\vec{x}$ along direction $\hat{l}$ is same as the spread of the dot product (aka component or projection) $\hat{l}^T \vec{x}$This projection $\hat{l}^T \vec{x}$ is a scalar quantity - we can use the univariate formula to measure its variance [12].

The expected value of the projection is

$$\vec{\mu}_l = \int_{\vec{x} \in D} \left(\hat{l}^T \vec{x}\right)\ p(\vec{x})\, d\vec{x} = \hat{l}^T \int_{\vec{x} \in D} \vec{x}\ p(\vec{x})\, d\vec{x} = \hat{l}^T \vec{\mu}$$

The variance is given by

$$var\left(\hat{l}^T \vec{x}\right) = \int_{\vec{x} \in D} \left(\hat{l}^T \vec{x} - \hat{l}^T \vec{\mu}\right)^2 d\vec{x}$$

Now, since the transpose of a scalar is the same scalar, we can write the square term within the integral as the product of the scalar $\hat{l}^T(\vec{x} - \vec{\mu})$ and its own transpose

$$var\left(\hat{l}^T \vec{x}\right) = \int_{\vec{x} \in D} \left(\hat{l}^T \vec{x} - \hat{l}^T \vec{\mu}\right)\left(\hat{l}^T \vec{x} - \hat{l}^T \vec{\mu}\right)^T d\vec{x} = \int_{\vec{x} \in D} \hat{l}^T (\vec{x} - \vec{\mu})\left(\hat{l}^T (\vec{x} - \vec{\mu})\right)^T d\vec{x}$$

Using equation 2.11

[24] as always, we will use overhead hats to denote unit length vectors signifying directions

$$var\left(\hat{l}^T\vec{x}\right) = \int_{\vec{x}\in D} \hat{l}^T\left(\vec{x}-\vec{\mu}\right)\left(\vec{x}-\vec{\mu}\right)^T\left(\hat{l}^T\right)^T d\vec{x} = \int_{\vec{x}\in D} \hat{l}^T\left(\vec{x}-\vec{\mu}\right)\left(\vec{x}-\vec{\mu}\right)^T\hat{l}\, d\vec{x}$$

Since $\hat{I}$ is independent of $\vec{x}$, we can take it out of the integral. Hence,

$$var\left(\hat{l}^T\vec{x}\right) = \hat{l}^T\left(\int_{\vec{x}\in D}\left(\vec{x}-\vec{\mu}\right)\left(\vec{x}-\vec{\mu}\right)^T d\vec{x}\right)\hat{l} = \hat{l}^T\mathbb{C}\left(\mathbb{X}\right)\hat{l}$$

where

$$\sqrt{\left(.6\times(20-100)^2+0.4\times(20-(-100))^2\right)} = 97.9795 \qquad (5.14)$$

For simplicity, we will drop the X in parenthesis and simply write $\mathbb{C}$ (X) as $\mathbb{C}$. An equivalent way of looking at the covariance matrix of a d dimensional random variable X taking vector values $\vec{x}$ is as follows:

$$\mathbb{C} = \begin{bmatrix} \sigma_{11} & \sigma_{12} & \sigma_{13}\cdots & \sigma_{1d} \\ \sigma_{21} & \sigma_{22} & \sigma_{23}\cdots & \sigma_{2d} \\ \vdots & & & \\ \sigma_{d1} & \sigma_{d2} & \sigma_{d3}\cdots & \sigma_{dd} \end{bmatrix} \qquad (5.15)$$

where

$$\sigma_{i,j} = \begin{cases} \int_{x_i\in D_i}\int_{x_j\in D_j}\left(x_i-\mu_i\right)\left(x_j-\mu_j\right)dx_i dx_j \Rightarrow \text{ for continuous distributions} \\ \sum_{i=1}^{n}\sum_{j=1}^{n}\left(x_i-\mu_i\right)\left(x_j-\mu_j\right) \Rightarrow \text{ for discrete } n \text{ point distributions} \end{cases}$$

is the co-variance of the i^{th} and j^{th} terms of the random vector $\vec{x}$.

$\mathbb{C}(X)$ or $\mathbb{C}$ is the *covariance matrix* of the random variable X. A little thought will reveal that equations 5.14 and 5.15 are equivalent.

The following things are noteworthy:

- Fron equation 5.14, $\mathbb{C}$ is the product of a $d \times 1$ vector $(\vec{x}-\vec{\mu})$ and its transpose $(\vec{x}-\vec{\mu})^T$, a $1 \times d$ vector. Hence, $\mathbb{C}$ is a $d \times d$ matrix.
- This matrix is independent of the direction, $\hat{I}$, along which we are measuring variance or spread. In fact, we can pre-compute $\mathbb{C}$ and then when we need to measure the variance along any direction $\hat{I}$ we can evaluate the quadratic form $\hat{I}^T\ \mathbb{C}\ \hat{I}$ to obtain the variance along that direction.. Thus $\mathbb{C}$ is a generic property of the distribution, much like $\vec{\mu}$. $\mathbb{C}$ is called the covariance of the distribution.

- Covariance is the multivariate peer of the univariate entity variance.
- That covariance is the multivariate analog of variance is kind of evident by comparing the expressions in equation 5.13 and equation 5.14.

VARIANCE AND EXPECTED VALUE

As outlined above, variance is the expected value of the distance, $(\vec{x} - \vec{\mu})^2$, of sample points x, from the mean μ. This can be easily seen by comparing equation 5.13 to equation 5.10 and equation 5.11. This leads to the following formula (where the principle of expected value of linear combinations has been used)

$$var\,(X) = \mathbb{E}\left((X-\mu)^2\right) = \mathbb{E}\left(X^2\right) - \mathbb{E}\left(2\,\mu\,X\right) + \mathbb{E}\left(\mu^2\right)$$

Since μ is a constant, we can take it out of the expected value (special case of the principal of expected value of linear combinations). Thus we get,

$$var\,(X) = \mathbb{E}\left(X^2\right) - 2\vec{\mu}\;\;\mathbb{E}\,(X) + \vec{\mu}^2\;\;\mathbb{E}\,(1)$$

But $\mu = \mathbb{E}\,(X)$. Also, expected value of a constant is that constant. Thus, $\mathbb{E}\,(1) = 1$. Hence,

$$var\,(X) = \mathbb{E}\left(X^2\right) - 2\vec{\mu}^2 + \vec{\mu}^2\;\;\mathbb{E}\,(1) = \mathbb{E}\left(X^2\right) - \vec{\mu}^2$$

or

$$var\,(X) = \mathbb{E}\left(X^2\right) - \mathbb{E}\,(X)^2 \qquad (5.16)$$

5.8 Sampling from a Distribution

In section 5.5.1 we saw some code to sample probabilistic distributions. Now we will study the process of sampling in more details.

Sampling is the process of randomly choosing a subset of all possible values of a random variable.

In univariate cases, the sample value will be a scalar. In multivariate cases, the sample value will be a vector. The higher the probability of a particular value, the more its chances of getting chosen. Thus, in a set of of sampled values, higher probability values will be over represented.

The hope, of course, is that the subset will be a good representation of the entire population, i.e., analyzing the subset would yield insights about the entire population.

If we know a distribution, we can use closed form expressions to obtain its properties like mean and variance. Many standard distributions and closed form equations for obtaining their means and variance are discussed in section 5.9. But many a times we don't know the distribution.

We only have the sampled values. Under those circumstances, sample mean and sample variance is often used. They are computed as described below.

Equations 5.8, 5.9 and 5.10 showed us how to compute the expected value from the entire population. There the summations ran over all possible values of the random variable. In the continuous case, it ran over the entire domain of the continuous random variable. This yielded the true expected value.

If, on the other hand, the summations in the same formulae run over only the subset of sampled values, we get sampled mean. Similarly, consider equations 5.13 and 5.14. When the summation runs over all possible values of the random variable or (in the continuous case) the entire domain of the random variable, they yield true covariance. If instead we sum over the sample values we get sample variance or sample covariance.

In some situations, like Gaussian distributions, which we are going to study soon, it can be theoretically proved that the sample mean and variance are optimal (best possible guess to the true mean and variance given the sampled data). Also, sample mean approaches the true mean as the number of samples increases and with "enough" samples, we will get a pretty good approximation to the true mean. In the next subsection we will see some insights into how much is "enough".

LAW OF LARGE NUMBERS - HOW MANY SAMPLES ARE ENOUGH?

Informally speaking, the law of large number says that if we draw a large number of sample values from some probability distribution, their average should be close to the expected value of the distribution. In the limit, the average over an infinite number of samples will match the mean. In practise, we cannot draw an infinite number of samples, so there is no guarantee that the sample mean will coincide with the expected value (true mean) in real life sampling. But if the number of samples is "large", they will not be too different. This is not a matter of mere theory. Casinos design games where the probability of the house winning a bet against the guest is slightly higher.

Over the very large number of bets placed in a casino, this indeed happens -that is why casinos make money on the whole even though they can lose an individual bet.

How many samples is "large number of samples"? Well, it is not defined precisely. But one thing is known, if the variance is larger, then more number of samples need to be drawn to make the law of large numbers apply.

Let us illustrate this with an example. Consider a betting game. Suppose that the famous soccer club FC Barcelona, for reasons unknown, has agreed to play very very large number of matches against the Machine Learning Experts' Soccer Club of Sillicon Valley. One can place a bet of $100 on a team. If that team wins, one gets $200 back, i.e., makes $100. If that team loses, one loses the bet, i.e., makes -$100. The betting game is happening in a country where nobody knows anything about the reputations of these clubs. A better bets on FC Barcelona in the first game and wins $100. Can the better say, on the basis of this one observation, that by betting on Barcelona he/she expects to win $100 every time? Obviously, no.

Now suppose the better places 100 bets and wins $100 99 times and loses $100 once. Now, the better can expect, with some more confidence that he/she will win $100 or near about by betting on Barcelona. Based on these observations, the sample mean winnings from a bet

on FC Barcelona is $0.99 \times (100) + 0.01 \times (-100) = 98$. The sample standard deviation is $\sqrt{\left(.99 \times (98-100)^2 + 0.01 \times (98-(-100))^2\right)} = 19.8997$. Relative to the sample mean, the sample standard deviation is $19.8997/98 = 0.203$.

Now consider the same game, excepting now FC Barcelona is playing Real Madrid football club.

Since the two teams are evenly matched now (the theoretical win probability of Barcelona is 0.5), the results are no longer one sided. After 100 games, say FC Barcelona won 60 times and Real Madrid won 40 times. The sample mean winnings on a Barcelona bet is $0.6 \times (100) + 0.4 \times (-100) = 20$. The sample standard deviation is $\sqrt{\left(.6 \times (20-100)^2 + 0.4 \times (20-(-100))^2\right)} = 97.9795$. Relative to the sample mean, it is $97.9795/20 = 4.89897$. This is a much larger number than the previous 0.203. In this case, even after 100 trials, one cannot be very confident in predicting that the expected win is the sample mean $20.

The overall intuition is that, *if we take "sufficiently large" number of samples, their average will be close to the expected value. The exact definition of what constitutes "sufficiently large" number of samples is not known. However, larger the variance (relative to the mean), more samples are needed*.

5.9 Some famous probability distributions

In this section we will introduce some probability distributions and density functions that are often used in deep learning. We will be using PyTorch code snippets to demonstrate how to setup, sample and compute properties like expected values, variance/covariance etc for each of these distributions.

Note:

- In the code snippets, for every distribution, we evaluate the probability using
  - PyTorch distributions function call
  - raw evaluation from formula (for understanding the math

  Both should yield the same result. In practice, one should use the PyTorch distributions function call instead of the raw formula.
- In the code snippets, for every distribution,
  - we evaluate the theoretical mean and variance using PyTorch distributions function call
  - sample mean and variance

  When the sample set is large enough in size, sample mean and theoretical mean should be close. Ditto for variance.
- In machine learning, we often work with logarithm of the probability. Since the popular distributions are exponential in nature, this leads to simpler computations.

Fully functional code for these distributions, executable via Jupyter-notebook, can be found at

https://github.com/krishnonwork/mathematical-methods-in-deep-learning-ipython/tree/master/python/ch5

Now, let us dive into each of these probability distributions.

5.9.1 Uniform Random Distributions

Consider a continuous random variable x which can take any value from a fixed compact range, say $[a, b]$ *with equal probability, while probability of x taking a value outside the range is zero*. The corresponding $p(x)$ is a uniform probability distribution. Formally stated

$$p(x) = \begin{cases} \frac{1}{b-a} & \text{if } x \in [a, b] \\ 0 & \text{otherwise} \end{cases} \tag{5.17}$$

Equation 5.17 means $p(x)$ is constant, $1/b-a$, for x in between a and b and zero for other values of x. Note how the value of the constant is cleverly chosen so as to make the total area under the curve 1. This equation is depicted graphically in Fig. 5.5.

Listing 5.2: PyTorch code to compute logarithm of the probability of univariate uniform random distribution.

```
1  from torch.distributions import Uniform
2
3  a = torch.tensor([1.0], dtype=torch.float)
4  b = torch.tensor([5.0], dtype=torch.float)
5
6  ufm_dist = Uniform(a, b)
7
8  X = torch.tensor([2.0], dtype=torch.float)
9
10 def raw_eval(X, a, b):
11     return torch.log(1 / (b - a))
12
13 log_prob = ufm_dist.log_prob(X)
14 raw_eval_log_prob = raw_eval(X, a, b)
15 assert torch.isclose(log_prob, raw_eval_log_prob, atol=1e-4)
```

Import PyTorch Uniform distribution

Set distribution parameters

Instantiate Uniform distribution object

Instantiate single point test dataset

Evaluate probability using PyTorch

Evaluate probability using formula

Assert that probabilities match

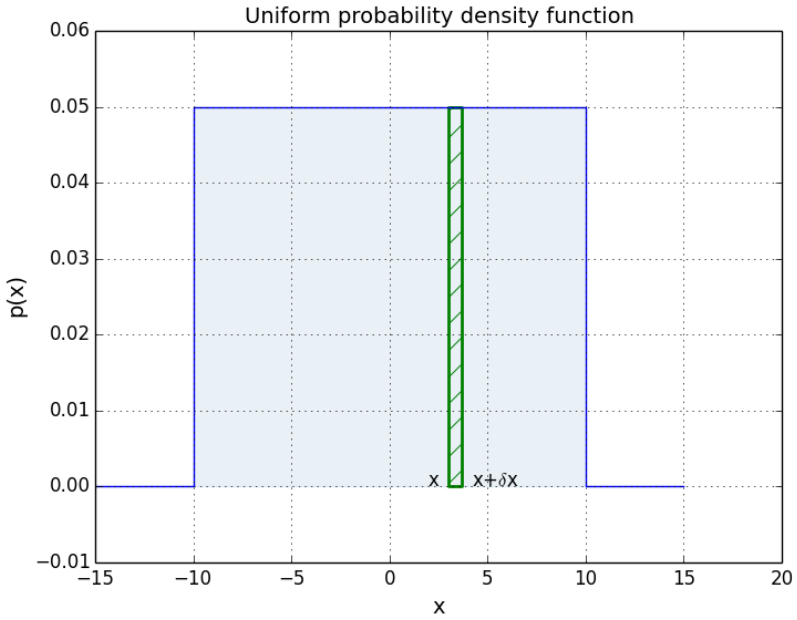


Figure 5.5: Univariate (single variable) uniform random probability density function. Probability $p(x)$ is constant, 0.05, in the interval [−10, 10] and zero everywhere outside the interval. Thus it depicts equation 5.17 with $b = 10, a = -10$. The area under the curve is the area of the blue rectangle of width 20 and height 0.05, $20 \times 0.05 = 1$. The green rectangle depicts an infinitessimally small interval corresponding to event $E = \{x \leq X < x + \delta x\}$.

If we draw a random sample x from this distribution, the probability that the value of the sample is between, say, 4 and $4 + \delta x$, with $\delta x \to 0$, is $p\,(4) = 0.05$. The probability that the value of the sample is between, say, 15 and $15 + \delta x$, with $\delta x \to 0$, is $p\,(15) = 0$

Fully functional code for uniform distribution, executable via Jupyter-notebook, canbe found at https://github.com/krishnonwork/mathematical-methods-in-deep-learning-ipython/blob/master/python/ch5/5.9.1-uniform-random-distribution.ipynb

EXPECTED VALUE OF UNIFORM DISTRIBUTION

We will do this for the univariate case although the computations can be easily extended to themultivariate case. Substituting the probability density function from equation 5.17 into the expression for expected value for a continuous variable, equation 5.10

$$\begin{aligned}\mathbb{E}_{uniform}\,(\mathrm{X}) &= \int_{-\infty}^{\infty} x\; p\,(x)\; dx = \int_{a}^{b} x \left(\frac{1}{b-a}\right)\; dx \\ &= \frac{1}{(b-a)} \int_{a}^{b} x\; dx = \frac{1}{(b-a)} \left[\frac{x^2}{2}\right]_a^b = \frac{(b^2-a^2)}{2\,(b-a)} \\ &= \frac{(a+b)}{2}\end{aligned} \tag{5.18}$$

(Note how the limits of integration changed because $p\,(x)$ is zero outside the interval $[a, b]$.) Overall, equation 5.18 agrees with our intuition. The expected value is bang in the middle of the uniform interval, as shown in Fig. 5.6.

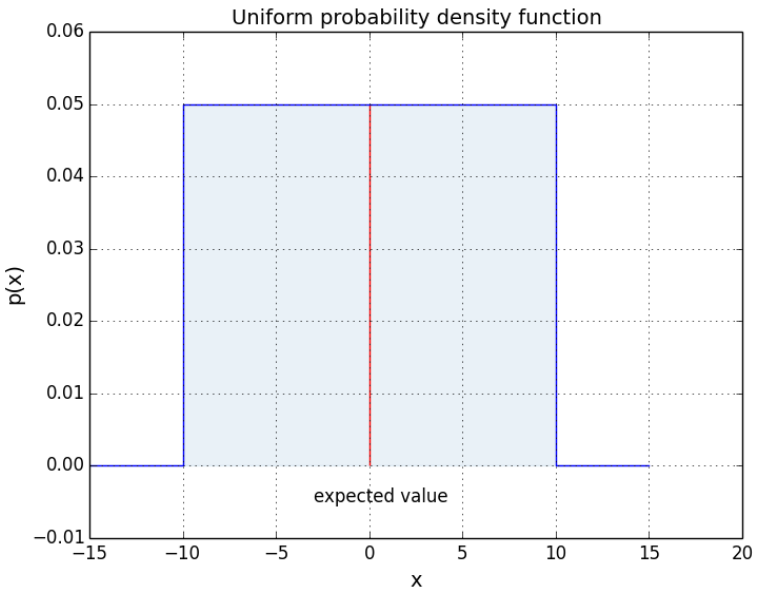


Figure 5.6: Univariate (single variable) uniform random probability density function. The solid red line in the middle indicates the expected value. Interactive visualizations (where one can change the parameters and observe how the graph changes as a result of that) can be found at https://github.com/krishnonwork/mathematical-methods-in-deep-learning-ipython/blob/master/python/ch5/5.9.1-uniform-random-distribution.ipynb

VARIANCE OF UNIFORM DISTRIBUTION

If we look at Fig 5.6, it is intuitively obvious that the packing density of the samples is related to the width of the rectangle. Smaller that width, tighter the packing and smaller the variance and vice versa. Let us see if the math supports that intuition.

$$\begin{aligned} var_{uniform}(x) &= \int_{x=-\infty}^{\infty} (x-\mu)^2 \, p(x) \, dx = \\ &= \int_{x=-\infty}^{\infty} \left(x - \frac{a+b}{2}\right)^2 \frac{1}{(b-a)} dx = \\ &= \frac{(b-a)^2}{12} \end{aligned} \tag{5.19}$$

Looking at Fig 5.5 we can see that the variance in equation 5.19 is proportional to the square of the width of the rectangle, i.e., $(b-a)^2$.

Listing 5.3: PyTorch code to compute mean and variance of univariate uniform random distribution.

```
num_samples = 100000                                  # Number of sample points

samples = ufm_dist.sample([num_samples])              # Obtain samples from ufm_dist instantiated in 5.2
                                                      # 100000 × 1 tensor
sample_mean = samples.mean()                          # Sample Mean
dist_mean = ufm_dist.mean                             # Mean via PyTorch function
assert torch.isclose(sample_mean, dist_mean, atol=0.2)

sample_var = ufm_dist.sample([num_samples]).var()     # Sample Variance
dist_var = ufm_dist.variance                          # Variance via PyTorch function
assert torch.isclose(sample_var, dist_var, atol=0.2)
```

MULTIVARIATE UNIFORM DISTRIBUTION

Uniform distribution can be multivariate too. In that case, the random variable will be a vector, $\vec{x}$ (i.e., not a single value, but a sequence of values). Its domain will be a multi-dimensional volume instead of the X axis and the graph will be higher dimensional than 2.

For example a two variable uniform random distribution.

$$p(x,y) = \begin{cases} \frac{1}{(b_1-a_1)(b_2-a_2)} & \text{if } (x,y) \in [a_1,b_1] \times [a_2,b_2] \\ 0 & \text{otherwise} \end{cases} \tag{5.20}$$

Note: Here $(x, y) \in [a_1, b_1] \times [a_2, b_2]$ indicates a rectangular domain on the two dimensional XY plane where x lies between a_1 and b_1 while y lies between a_2 and b_2. Equation 5.20 is depicted graphically in Fig 5.7.

In the general multidimensional case

$$p(\vec{x}) = \begin{cases} \frac{1}{V} & \text{if } \vec{x} \in D \\ 0 & \text{otherwise} \end{cases} \tag{5.21}$$

Here V is the volume of the hyper-dimensional box with base D. Equation 5.21 means $p(\vec{x})$ is constant for $\vec{x}$ in the domain D and zero for other values of x. When non-zero, it has a constant value, inverse of the volume V - this will make the total volume under the density function 1.

5.9.2 Gaussian (aka Normal) Distribution

This is probably the most famous distribution in the world. Let us consider, one more time, the weights of adult residents of Statsville. If Statsville is anything like a real city, the likeliest

weight will be around 75 Kgs, i.e., the largest percentage of the population will have this weight. Weights near this value (say 70 Kgs or 80 Kgs) will also be quite likely, although slightly less likely than 75 Kgs. Weights further away from 75 Kgs are still less likely and it goes on like that. The further one goes from 75 Kgs, the less will be the percentage of the population at that weight. Very *outlier* values, like 40 or 110 Kgs are going to be quite unlikely.

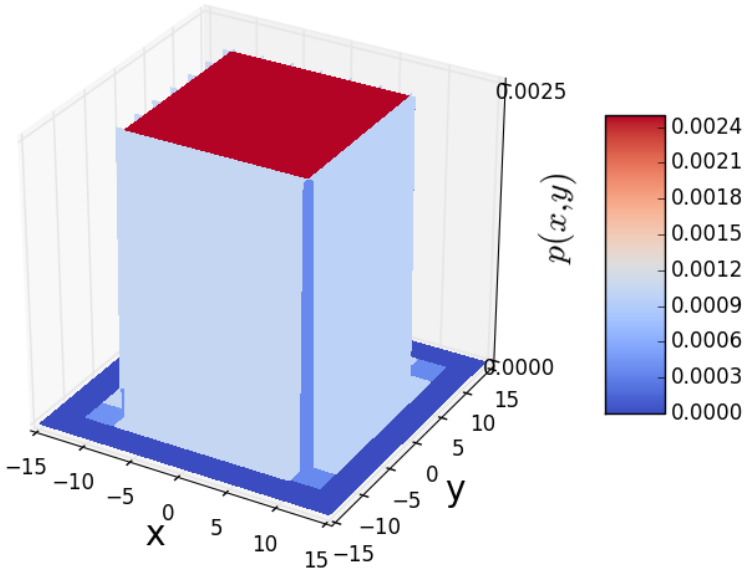


Figure 5.7: Bivariate uniform random probability density function. Probability $p\ (x, y)$ is constant, 0.0025, in the domain $(x, y) \in [-10, 10] \times [-10, 10]$ and zero everywhere outside the interval. The volume of the blue box of width 20×20 and height 0.0025, $20 \times 20 \times 0.0025 = 1$.

Informally speaking, this is how a Gaussian probability density function looks like. It is a *bell shaped curve*. The central value has the highest probability. The probability dies gradually as one moves away from the center. In theory, however, it never dies completely (i.e., the function $p(x)$ never exactly becomes equal to 0), although it becomes "almost" zero for all practical purposes. This behavior is described in mathematics as "asymptotically approaching zero". Figure 5.8 shows a Gaussian probability density function.

Formally

$$p(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{\frac{-(x-\mu)^2}{2\sigma^2}} \tag{5.22}$$

Here, μ and σ are parameters - μ corresponds to the center (e.g., in Fig. 5.8 is $\mu = 0$). The parameter σ controls the width of the bell. Larger σ implies that $p(x)$ dies slower as one moves away from the center.

The Gaussian (aka Normal) probability density function is so popular that we have a special symbol for it, $\mathcal{N}(\mu, \sigma^2)$.

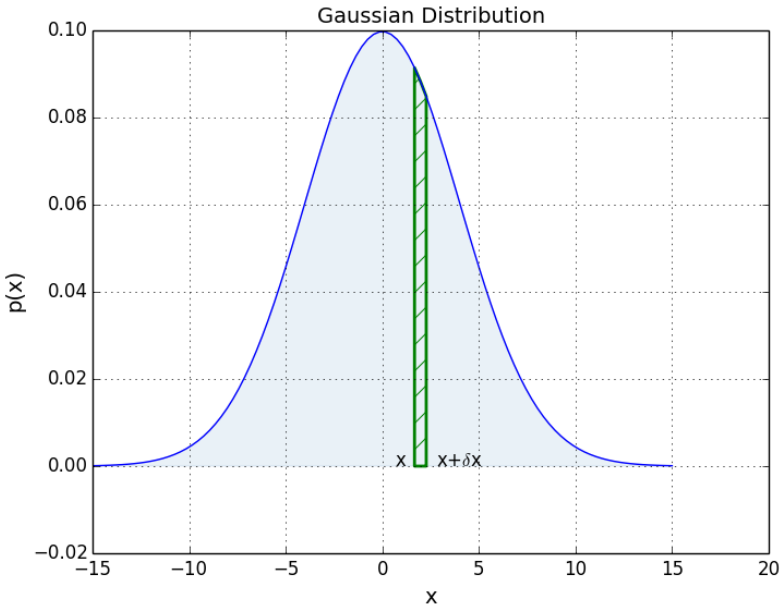


Figure 5.8: Univariate Gaussian random probability density function, $\mu = 0$ and $\sigma = 4$. Bell shaped curve - highest at the center and decreases more and more as one goes away from the center, approaching zero asymptotically. The value $x = 0$ has the highest probability, corresponding to the center of the probability density function. Note that the curve is symmetric. Thus for instance, probability of a random sample being in the vicinity of -5 is same as that of 5, viz., 0.04, i.e., $p(-5) = p(5) = 0.04$. Interactive visualization (where one can change the parameters and observe how the graph changes as a result of that) can be found at https://github.com/krishnonwork/mathematical-methods-in-deep-learning-ipython/blob/master/python/ch5/5.9.2-normal-distribution.ipynb

It can be proved (but is exceedingly tedious and will be skipped here) that

$$\int_{x=-\infty}^{\infty} \mathcal{N}(\mu, \sigma^2)dx = \int_{x=-\infty}^{\infty} \frac{1}{\sqrt{2\pi}\sigma} e^{\frac{-(x-\mu)^2}{2\sigma^2}} dx = 1$$

This establishes that $\mathcal{N}(\mu, \sigma^2)$ is a true probability (satisfying the sum rule in equation 5.7).

Listing 5.4: PyTorch code to compute logarithm of the probability of univariate normal (gaussian) distribution.

```
from torch.distributions import Normal
                Import PyTorch Univariate Normal (Gaussian) distribution
mu = torch.tensor([0.0], dtype=torch.float)  <-- Set distribution params
sigma = torch.tensor([5.0], dtype=torch.float)
                                Instantiate Univariate Normal distribution object
uvn_dist = Normal(mu, sigma)
                                                Instantiate single point test dataset
X = torch.tensor([0.0], dtype=torch.float)


def raw_eval(X, mu, sigma):
    K = 1 / (math.sqrt(2 * math.pi) * sigma)
    E = math.exp( -1 * (X - mu) ** 2 * (1 / (2 * sigma ** 2)))
    return math.log(K * E)
                                Evaluate probability using PyTorch
log_prob = uvn_dist.log_prob(X)
                                                Evaluate probability using formula
raw_eval_log_prob = raw_eval(X, mu, sigma)
assert log_prob == raw_eval_log_prob  <-- Assert that probabilities match
```

Fully functional code for normal distribution, executable via Jupyter-notebook, can be found at https://github.com/krishnonwork/mathematical-methods-in-deep-learning-ipython/blob/master/python/ch5/5.9.2-normal-distribution.ipynb

MULTIVARIATE GAUSSIAN

Gaussian distribution can be multivariate too. Then the random variable becomes a vector $\vec{x}$ as usual. The parameter μ also becomes a vector and the parameter σ becomes a matrix, Σ. We will state the formula here and will revisit this after studying expected values, variance and covariance.

$$p(\vec{x}) = \mathcal{N}(\vec{\mu}, \Sigma) = \frac{1}{(2\pi \det \Sigma)^{\frac{1}{2}}} e^{-\frac{1}{2}(\vec{x}-\vec{\mu})^T \Sigma^{-1} (\vec{x}-\vec{\mu})} \quad (5.23)$$

Equation 5.23 describes the probability density function for the random vector $\vec{x}$ to lie within the infinitesimally small volume with dimensions $\delta\vec{x}$ around the point $\vec{x}$ [25]. The vector $\vec{\mu}$ and the matrix Σ are parameters. As in the univariate case, $\vec{\mu}$ corresponds to the most likely value of the random vector. Fig 5.9 shows the Gaussian (aka Normal) distribution in 2 variables in the 3 dimensions. The shape of the base of the bell is controlled by the parameter Σ.

Listing 5.5: PyTorch code to compute logarithm of the probability of multivariate normal (gaussian) distribution.

```
from torch.distributions import MultivariateNormal
mu = torch.tensor([0.0, 0.0], dtype=torch.float)
C = torch.tensor([[5.0, 0.0], [0.0, 5.0]], dtype=torch.float)

mvn_dist = MultivariateNormal(mu, C)

X = torch.tensor([0.0, 0.0], dtype=torch.float)

def raw_eval(X, mu, C):
    K = (1 / (2 * math.pi * math.sqrt(C.det())))
    X_minus_mu = (X - mu).reshape(-1, 1)
    E1 = torch.matmul(X_minus_mu.T, C.inverse())
    E = math.exp(-1 / 2. * torch.matmul(E1, X_minus_mu))
    return math.log(K * E)

log_prob = mvn_dist.log_prob(X)
raw_eval_log_prob = raw_eval(X, mu, C)
assert log_prob == raw_eval_log_prob
```

Import PyTorch Multivariate Normal (Gaussian) distribution

Set distribution params

Instantiate Multivariate Normal distribution object

Instantiate single point test dataset

Evaluate probability using PyTorch

Evaluate probability using formula

Assert that probabilities match

EXPECTED VALUE OF GAUSSIAN DISTRIBUTION

Substituting the probability density function from equation 5.22 into the expression for expected value for a continuous variable, equation 5.10 we get

[25] imagine a tiny box (cuboid) whose sides are successive elements of $\delta\vec{x}$ with the top left corner of the box at $\vec{x}$

$$\mathbb{E}_{gaussian}(X) = \int_{-\infty}^{\infty} x \frac{1}{\sqrt{2\pi\sigma}} e^{\frac{-(x-\mu)^2}{2\sigma^2}} dx$$

$$= \frac{1}{\sqrt{\pi}} \int_{-\infty}^{\infty} \frac{(x-\mu)}{\sqrt{2}\sigma} e^{\frac{-(x-\mu)^2}{2\sigma^2}} dx + \mu \int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi\sigma}} e^{\frac{-(x-\mu)^2}{2\sigma^2}} dx$$

Substituting $y = \frac{-(x-\mu)}{\sqrt{2}\sigma}$

$$\mathbb{E}_{gaussian}(X) = \frac{\sqrt{2}\sigma}{\sqrt{\pi}} \int_{-\infty}^{\infty} y e^{-y^2} dy + \mu \int_{-\infty}^{\infty} p(x)\ dx$$

Substituting $u = y^2$ and using equation 5.6

$$\mathbb{E}_{gaussian}(X) = \frac{\sqrt{2}\sigma}{2\sqrt{\pi}} \int_{\infty}^{\infty} e^{-u} du + \mu$$

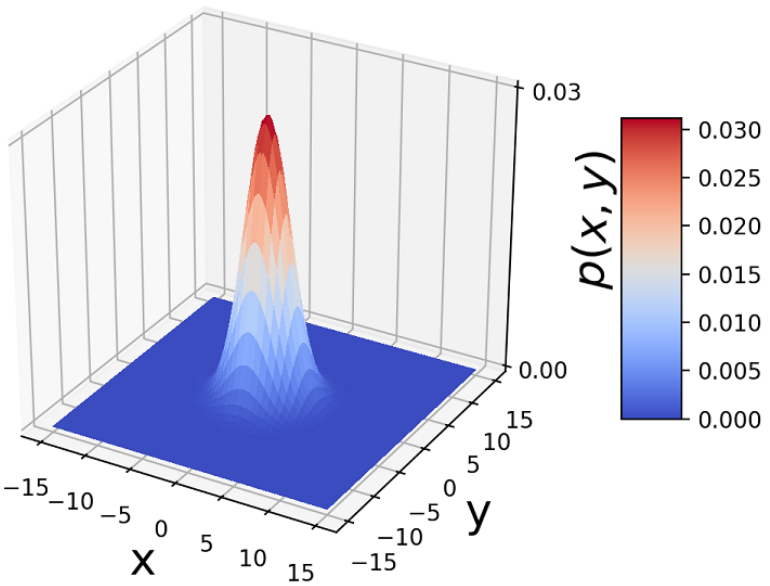


Figure 5.9: Bivariate Gaussian random probability density function. Bell shaped surface - highest at the center and decreases more and more as one goes away from the center, approaching zero asymptotically. $x = 0, y = 0$ has the highest probability corresponding to the center of the probability density function. Has a circular base, Σ matrix is a scalar multiple of Identity matrix I. Interactive visualization (where one can change the parameters and observe how the graph changes as a result of that) can be found at

https://github.com/krishnonwork/mathematical-methods-in-deep-learning-ipython/blob/master/python/ch5/5.9.2-normal-distribution.ipynb

Note that the limits of the integral in first term are identical. This happened because $u = y^2 \rightarrow \infty$ whether $y \rightarrow \infty$ or $y \; ! \; -\infty$. But an integral with same lower and upper limit is zero. Thus the first term is zero. Hence,

$$\mathbb{E}_{gaussian}(X) = \mu \tag{5.24}$$

Intuitively, this makes perfect sense. The probability density $p(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{\frac{-(x-\mu)^2}{2\sigma^2}}$ peaks (maximizes) at $x = \mu$. At this x, the exponent becomes zero, which makes the term $e^{\frac{-(x-\mu)^2}{2\sigma^2}}$ attain its maximum possible value of 1. This is bang in the middle of the bell, as shown in Fig. 5.10. . And of course the expected value will coincide with middle value if the density is symmetric and peaks at the middle.

Analogously, in the multivariate case, the Gaussian multi-dimensional random variable X that takes vector values $\vec{x}$ in the domain D, i.e, $\vec{x} \in D$, has an expected value

$$\mathbb{E}_{gaussian}(X) = \vec{\mu} \tag{5.25}$$

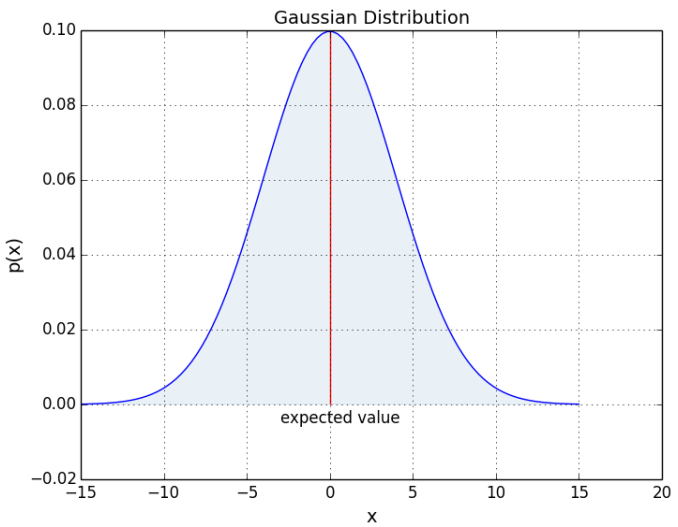


Figure 5.10: Univariate (single variable) normal (Gaussian) random probability density function, $\mu = 0$ and $\sigma = 4$. The solid red line in the middle indicates the expected value

VARIANCE OF GAUSSIAN DISTRIBUTION

The variance of the Gaussian distribution is obtained by substituting equation 5.22 in the integral form of equation 5.13. The mathematical derivation is shown in appendix 6.2, here we will only state the result.

The variance of a Gaussian distribution with probability density function $p(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{\frac{-(x-\mu)^2}{2\sigma^2}}$ is σ^2 and the standard deviation is the square root of that, viz., σ.

This makes intuitive sense. σ appears in the denominator of a negative exponent in the expression for probability density function $p(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{\frac{-(x-\mu)^2}{2\sigma^2}}$. As such $p(x)$ is an increasing function of σ, i.e., for a given x and μ, larger σ implies larger $p(x)$. In other words, larger σ implies that the probability decays more slowly as one moves away from the center, i.e., a fatter bell curve, a bigger spread and hence a larger variance. Figure 5.11 depicts this.

Listing 5.6: PyTorch code to compute mean and variance of univariate normal (gaussian) distribution.

```
num_samples = 100000                                  # Number of sample points

samples = uvn_dist.sample([num_samples])              # Obtain samples from uvn_dist instantiated in 5.4
                                                      # 100000 × 1 tensor
sample_mean = samples.mean()                          # Sample Mean
dist_mean = uvn_dist.mean                             # Mean via PyTorch function
assert torch.isclose(sample_mean, dist_mean, atol=0.1)

sample_var = uvn_dist.sample([num_samples]).var()     # Sample Variance
dist_var = uvn_dist.variance                          # Variance via PyTorch function
assert torch.isclose(sample_var, dist_var, atol=0.1)
```

Figure 5.11: 3 Gaussian probability densities with same μ but different σ s. Larger σ (variance) implies a fatter bell ⇒ more spread. Note that fatter curves are smaller in height as the total area under the curve must be 1.

COVARIANCE OF MULTIVARIATE GAUSSIAN DISTRIBUTION AND GEOMETRY OF THE BELL SURFACE

Comparing the equation 5.22 for univariate Gaussian probability density with equation 5.23 for multivariate Gaussian probability density, we intuitively feel that the matrix Σ is the multivariate peer of the univariate variance $\sigma 2$. Indeed, it is. Formally, for a multivariate Gaussian random variable with probability distribution given in equation 5.23, covariance matrix is given by the equation

$$\mathbb{C}_{gaussian}(X) = \Sigma \tag{5.26}$$

Indeed, as shown in Table 5.8, Σ regulates the shape of the base of the bell shaped probability density function.

It is easy to see that the exponent in equation 5.23 is a quadratic form (introduced in section 4.1.). As such, it defines a hyper-ellipse, as shown in Table 5.8 and section 2.17. All the properties of quadratic form and hyper ellipse apply here. Following is a list of such properties.

Listing 5.7: PyTorch code to compute mean and variance of multivariate normal (gaussian) distribution.

```
num_samples = 100000                                  # Number of sample points

samples = mvn_dist.sample([num_samples])              # Obtain samples from mvn_dist instantiated in 5.5
                                                      # 100000 × 2 tensor
sample_mean = samples.mean()                          # Sample Mean
dist_mean = mvn_dist.mean                             # Mean via PyTorch function
assert torch.allclose(sample_mean, dist_mean, atol=1e-1)

sample_var = mvn_dist.sample([num_samples]).var()     # Sample Variance
dist_var = mvn_dist.variance                          # Variance via PyTorch function
assert torch.allclose(sample_var, dist_var, atol=1e-1)
```

Geometric Properties of the Gaussian Covariance Matrix Σ Consider a 2D version of equation 5.23. We rewrite $\vec{x} = \begin{bmatrix} x \\ y \end{bmatrix}$ and $\vec{\mu} = \begin{bmatrix} \mu_x \\ \mu_y \end{bmatrix}$ - 2D vectors both. Also $\Sigma = \begin{bmatrix} \sigma_{11} & \sigma_{12} \\ \sigma_{21} & \sigma_{22} \end{bmatrix}$ - a 2 × 2matrix. The probability density function from equation 5.23 becomes

$$p(x, y) = \mathcal{N}(\vec{\mu}, \mathbf{\Sigma}) = \frac{1}{(2\pi \det \mathbf{\Sigma})^{\frac{1}{2}}} e^{-\frac{1}{2}\left(\sigma_{11}x^2 + (\sigma_{11}+\sigma_{12})xy + \sigma_{22}y^2\right)} \tag{5.27}$$

(use the knowledge from chapter 3 to satisfy yourself that equation 5.27 is a 2D analog of equation 5.23).

If we plot the surface $p(x, y)$ against (x, y), it looks like a bell in 3D space. The shape of the bell's base, on the (x, y) plane, is governed by the 2 × 2 matrix Σ. In particular

- If Σ is a diagonal matrix with equal diagonal elements, the bell is symmetric in all directions, its base is circular
- If Σ is a diagonal matrix with unequal diagonal elements, the base of the bell is elliptical.
- The axes of the ellipse are aligned with coordinate axes.
- For general Σ matrix the base of the bell is elliptical. The axes of the ellipse are not necessarily aligned with coordinate axes.

- The eigenvectors of Σ yield the axes of the elliptical base of the bell surface

Now if we sample, the distribution from equation 5.27, we will get a set of points (x, y) on the base plane of the surface depicted in Fig 5.9. *The taller z coordinate (depicting $p\ (x, y)$) of the surface at a point (x, y), the higher its probability of getting selected in the sampling. If we draw a large number of sample, the corresponding point cloud will look more or less like the base of the bell surface.* Table 5.8 shows various point clouds formed by sampling Gaussian distributions with different covariance matrices Σ. It should be compared with Fig. 5.11.

GEOMETRY OF SAMPLED POINT CLOUDS - COVARIANCE AND DIRECTION OF MAXIMUM OR MINIMUM SPREAD

We saw above that if a multivariate distribution has a covariance matrix $\mathbb{C}$, its variance (spread) along any specific direction $\hat{I}$ is $\hat{I}^T\ \mathbb{C}\hat{I}$. What is the direction along which this spread is maximum?

Asking this is same as asking what is the direction $\hat{I}$ that maximizes the quadratic form $\hat{I}^T\ \mathbb{C}\hat{I}$.

In section 4.1 we saw that a quadratic form like this is maximized or minimized when the the direction $\hat{I}$ is aligned with the eigenvector corresponding to the maximum or minimum eigenvalue of the matrix $\mathbb{C}$. Thus, *the maximum spread of a distribution occurs along the eigenvector of the covariance matrix corresponding to its maximum eigenvalue*. This is what led to the PCA technique in section 4.3.

Below, we will discuss the covariance of the Gaussian distribution and geometry of the point cloud formed by sampling a multivariate Gaussian very large number of times. The reader may want to take a sneak peek at Table 5.8 which shows various point clouds formed by sampling Gaussian distributions with different covariance matrices Σ.

MULTIVARIATE GAUSSIAN POINT CLOUDS AND HYPER ELLIPSES

The numerator of the exponential term in equation 5.23, $(\vec{x} - \vec{\mu})^T\Sigma^{-1}(\vec{x} - \vec{\mu})$, is a quadratic form which we studied in section 4.1. It should also remind you of the hyper-ellipse we studied in section 2.17, equation 2.35 and equation 4.1.

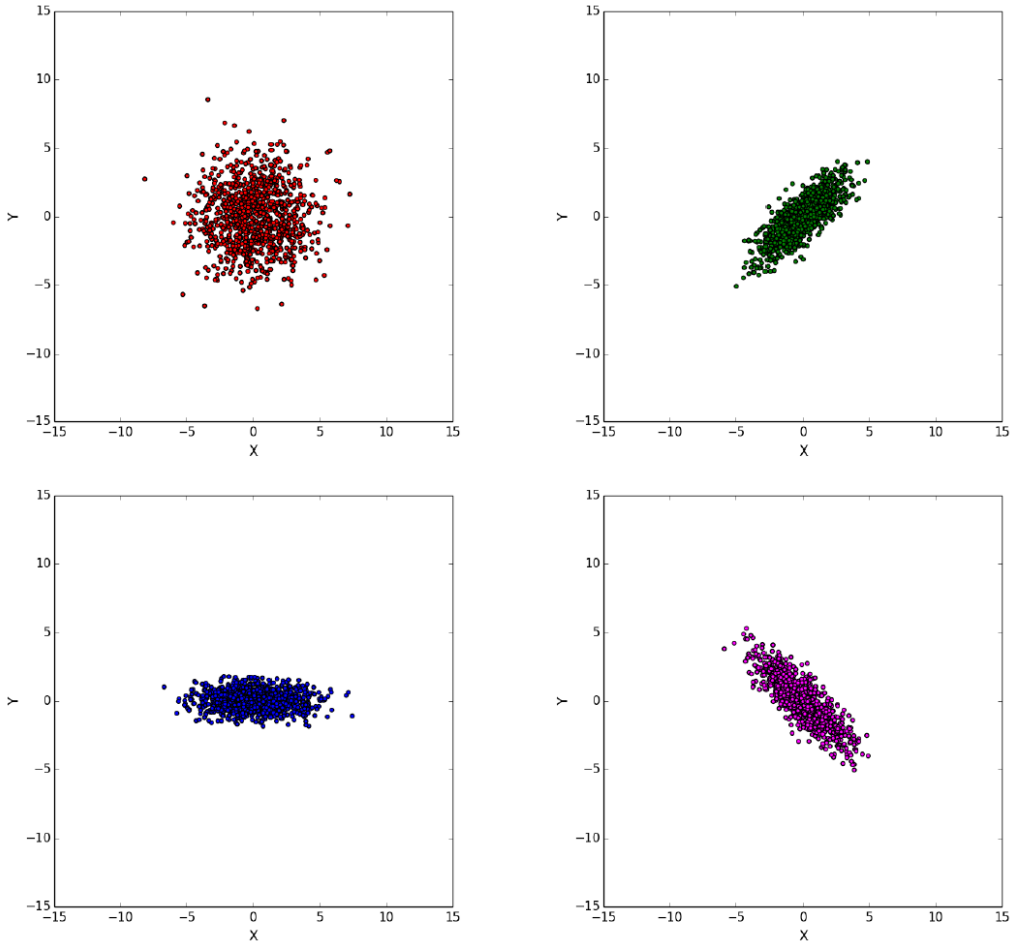


Table 5.8: Point clouds formed by sampling Multivariate Gaussians with same $\vec{\mu} = [0, 0]^T$ but different Σs. For the red point cloud $\Sigma = [[5, 0] , [0, 5]]$. For the blue point cloud $\Sigma = [[5, 0] , [0, 0.5]]$. For the green point cloud $\Sigma = [[2.75, 2.25] , [2.25, 2.75]]$. For the magenta point cloud $\Sigma = [[2.75, -2.25] , [-2.25, 2.75]]$. These point clouds correspond to the bases of the bell curves for multivariate Gaussian probability densities. All the point clouds except the red one may be replaced by an univariate Gaussian after rotation to align coordinate axes with eigenvectors of Σ (dimensionality reduction). See sections 4.3, 4.4, 4.5 for details. Interactive contour plots for the base of the bell curve can be found at https://github.com/krishnonwork/mathematical-methods-in-deep-learning-ipython/blob/master/python/ch5/5.9.2-normal-distribution.ipynb

Now, consider the plot of $p(\vec{x})$ against $\vec{x}$. This is a hyper-surface in n + 1 dimensional space, where the random variable $\vec{x}$ is n–dimensional. For instance, if the random Gaussian variable $\vec{x}$ is 2D, the $(\vec{x}, p(\vec{x}))$ plot in 3D is shown in Fig. 5.9. It is a bell-shaped surface. The

hyperellipse corresponding to the quadratic form in the numerator of the probability density function in equation 5.23 governs the shape and size of the base of this bell.

If the matrix Σ is diagonal (with equal diagonal elements) then the base will be *circular* - this is the special case shown in Fig 5.9. Otherwise the base of the bell will be elliptic. The eigenvectors of the covariance matrix Σ, correspond to the directions of the axes of the elliptical base. The eigen values correspond to the lengths of the axes.

5.9.3 Binomial Distribution

Suppose we have a database of people's photos. Also suppose that, we know 20% of the photos contain a celebrity, the remaining 80% do not. If we select 3 photos at random from this database, what is the probability that, say, 2 of them would contain a celebrity? This is the kind of problem that binomial distribution deals with.

In a computer vision centric machine learning setting, we would probably inspect the selected photos and try to predict whether they contain a celebrity. But for now, let us restrict ourselves to the simpler task of just blindly predicting the chances from aggregate statistics only.

If we select a single photo, the probability of that containing a celebrity is $\pi = 0.2$[26]. The probability of the photo not containing a celebrity is $1 - \pi = 0.8$. From that, we can compute the probability of, say, the first two sampled photos containing a celebrity but the last one containing a non-celebrity, i.e., the event $\{S, S, F\}$ (where S denotes success in finding a celebrity and F denotes failure in finding a celebrity). Using equation 5.4, the probability of the event $\{S, S, F\}$ is $\pi \times \pi \times (1 - \pi) = 0.2 \times 0.2 \times 0.8$. However, many other combinations are also possible.

All the possible combinations that can occur in a 3 trial are shown in Table 5.9. In that table, event ids 3, 5 and 6 correspond to two successes and one failure. They occur with probabilities $0.8 \times 0.2 \times 0.2$, $0.2 \times 0.8 \times 0.2$, $0.2 \times 0.2 \times 0.8$ respectively. If any one of them occur, we have 2 celebrity photos in 3 trials. Thus, using equation 5.3, the overall probability of selecting 2 celebrity photos in 3 trials is the sum of these event probabilities: $0.8 \times 0.2 \times 0.2 + 0.2 \times 0.8 \times 0.2 + 0.2 \times 0.2 \times 0.8 = 0.096$.

Table 5.9

Event Id	Event	probability
0	{F, F, F}	$(1 - \pi) \times (1 - \pi) \times (1 - \pi) = 0.8 \times 0.8 \times 0.8$
1	{F, F, S}	$(1 - \pi) \times (1 - \pi) \times \pi = 0.8 \times 0.8 \times 0.2$
2	{F, S, F}	$(1 - \pi) \times \pi \times (1 - \pi) = 0.8 \times 0.2 \times 0.8$
3	{F, S, S}	$(1 - \pi) \times \pi \times \pi = 0.8 \times 0.2 \times 0.2$

[26] This has nothing to do with the natural number π denoting the ratio of circumference to diameter of a circle, we are just re-using the symbol π following a popular convention.

4	{S, F, F}	$\pi \times (1-\pi) \times (1-\pi) = 0.2 \times 0.8 \times 0.8$
5	{S, F, S}	$\pi \times (1-\pi) \times \pi = 0.2 \times 0.8 \times 0.2$
6	{S, S, F}	$\pi \times \pi \times (1-\pi) = 0.2 \times 0.2 \times 0.8$
7	{S, S, S}	$\pi \times \pi \times \pi = 0.2 \times 0.2 \times 0.2$

In the general case, with larger than 3 trials, it would be impossibly tedious to enumerate all possible combinations of *success* and *f ailure* that can occur in a set of n trials. Fortunately, we can derive a formula. But before doing that, let us state the task of binomial distribution in more general terms.

Given a process which has binary outcome (success or failure) in any given trial, and the probability of success in a trial is a known constant, say π - Binomial distribution deals with the probability of observing k successes in n trials of the process.

Now, imagine events with n successive items, where each individual item can be either S or F. In table 5.9, we saw such events with $n = 3$. Each item has two possible values (S or F) and there are n items. Hence, altogether there can be $2 \times 2 \times \ldots 2 = 2^n$ possible events.

We are interested in only those events that have k occurrences of S (and therefore $(n-k)$ occurrences of F). How many of the n events are like that? Well, asking this is same as asking how many different ways can we choose k slots from a total of n possible slots? Another way to pose the same question is, how many different orderings of n items exist, where each item is either S or F and the total count of S is k? The answer, from combination theory is

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

Each of these events have a probability of $\pi^k \times (1-\pi)^{n-k}$. Hence, the overall probability of k successes in n trials is $\binom{n}{k}\ \pi^k \times (1-\pi)^{n-k}$.

Formally, if X is a random variable denoting the number of successes in n trials, with the probability of success in any single trial being some constant value π

$$p\,(X = k) = \binom{n}{k} \pi^k \times (1-\pi)^{n-k} \tag{5.28}$$

What all possible values can k take? Of course, we cannot have more than n successes in n trials, hence maximum possible value of k is n. All integer values between 0 and n are possible.

$$\sum_{k=0}^{k=n} p\,(X = k) = \sum_{k=0}^{k=n} \binom{n}{k} \pi^k \times (1-\pi)^{n-k}$$

The right hand side is expression for the generic term in the famous binomial expansion of $(a+b)^n$ with $a = \pi$ and $b = 1-\pi$. Hence we get

$$\sum_{k=0}^{k=n} p\left(X=k\right) = \sum_{k=0}^{k=n} \binom{n}{k} \pi^k \times (1-\pi)^{n-k} = (\pi + 1 - \pi)^n = 1^n = 1 \tag{5.29}$$

This agrees with intuition, since given n, k can only take values $0, 1, \ldots, n$, the sum of the probabilities on the left hand side of equation 5.29 corresponds to a certain event with probability 1.

Also, plugging in $n = 3$ and $k = 2$ and $\pi = 0.2$ into equation 5.28 yields $3!/2!\ 1!\ (0.2)^2\ (0.8)^{3-2} = 0.096$ exactly what we go from explicit enumeration.

Listing 5.8: PyTorch code to compute logarithm of the probability of binomial distribution.

```
from torch.distributions import Binomial

num_trials = 3
p = torch.tensor([0.2], dtype=torch.float)

binom_dist = Binomial(num_trials, probs=p)

X = torch.tensor([1], dtype=torch.float)

def nCk(n, k):
    f = math.factorial
    return f(n) * 1. / (f(k) * f(n-k))

def raw_eval(X, n, p):
    result = nCk(n, X) * (p ** X) * (1 - p) ** (n - X)
    return torch.log(result)

log_prob = binom_dist.log_prob(X)
raw_eval_log_prob = raw_eval(X, num_trials, p)
assert torch.isclose(log_prob, raw_eval_log_prob, atol=1e-4)
```

Import PyTorch Binomial distribution

Set distribution params

Instantiate Binomial distribution object

Instantiate single point test dataset

Evaluate probability using PyTorch

Evaluate probability using formula

Assert that probabilities match

Fully functional code for binomial distribution, executable via Jupyter-notebook, can be found at

https://github.com/krishnonwork/mathematical-methods-in-deep-learning-ipython/blob/master/python/ch5/5.9.3-binomial-distribution.ipynb

EXPECTED VALUE OF BINOMIAL DISTRIBUTION

We have seen that the binomial distribution deals with a random variable X which depicts the number of successes in n trials where the probability of success in a given trial is a constant π[27].

This X can take any integer value 0 to n. Hence,

$$\mathbb{E}(X) = \sum_{k=0}^{k=n} k\, p(X=k) = \sum_{k=0}^{k=n} k \binom{n}{k} \pi^k \times (1-\pi)^{n-k} = \sum_{k=0}^{k=n} k \frac{n!}{k!\,(n-k)!} \pi^k \times (1-\pi)^{n-k}$$

We can drop the first term which has the multiplier $k = 0$. Thus we get

$$\mathbb{E}(X) = \sum_{k=1}^{k=n} \frac{n!}{(k-1)!\,(n-k)!} \pi^k \times (1-\pi)^{n-k}$$

We can factor $n! = n\,(n-1)!$ and $\pi^k = \pi\,\pi^{k-1}$. Also, $n - k = (n-1) - (k-1)$. This gives us

$$\mathbb{E}(X) = \sum_{k=1}^{k=n} \frac{n\ (n-1)!}{(k-1)!\,((n-1)-(k-1))!} \pi\ \pi^{k-1} \times (1-\pi)^{n-k}$$

Substituting j for $k-1$ and m for $n-1$, we get

$$\mathbb{E}(X) = n\pi \sum_{j=0}^{j=m} \frac{m!}{j!\,(m-j)!} \pi^j \times (1-\pi)^{m-j} \qquad (5.30)$$

The quantity within the summation is similar to that in equation 5.29 - should sum to 1. This leaves us with

$$\mathbb{E}_{binomial}(X) = n\ \pi \qquad (5.31)$$

Equation 5.31 is saying that if π is the probability of success in a single trial, then the expected number of successes in n trials is $n\ \pi$. For instance, if the probability of success in a single trial is 0.2 then the expected number of successes in 100 trials is 20 - almost intuitively obvious.

[27] once again, this has nothing to do with the natural number π denoting the ratio of circumference to diameter of a circle - we are simply reusing that symbol.

VARIANCE OF BINOMIAL DISTRIBUTION

Variance of a binomial random variable depicting the number of successes in n trials where the probability of success in a given trial is a constant π is

$$var_{binomial} = n\ \pi(1-\pi) \tag{5.32}$$

Proof follows along the same lines as that of the expected value.

Listing 5.9: PyTorch code to compute mean and variance of binomial distribution.

```
1  num_samples = 100000          ← Number of sample points
2
3  samples = binom_dist.sample([num_samples])   ← Obtain samples from binom_dist instantiated in 5.8
4                                  ↖ 100000 × 1 tensor
5  sample_mean = samples.mean()  ← Sample Mean
6  dist_mean = binom_dist.mean   ← Mean via PyTorch function
7  assert torch.isclose(sample_mean, dist_mean, atol=0.2)
8                                                  Sample Variance ↙
9  sample_var = binom_dist.sample([num_samples]).var()
10 dist_var = binom_dist.variance  ← Variance via PyTorch function
11 assert torch.isclose(sample_var, dist_var, atol=0.2)
```

5.9.4 Multinomial Distribution

Consider again the example problem we studied in section 5.9.3. We have a database of people's photos. Only, instead of two classes, celebrity and non-celebrity, we have 4 classes

- photos of Albert Einstein (class 1) - say 10% of the photos belong to this class
- photos of Marie Curie (class 2) - say 42% of the photos belong to this class
- photos of Carl Friedrich Gauss (class 3) - say 4% of the photos belong to this class
- other photos (class 4) - remaining 44% of the photos belong to this class

If we randomly select a photo from the database, i.e., perform a random trial,

- the probability of selecting class 1 (picking up an Einstein photo) is $\pi_1 = 0.1$
- the probability of selecting class 2 (picking up a Marie Curie photo) is $\pi_2 = 0.42$
- the probability of selecting class 3 (picking up a Gauss photo) is $\pi_3 = 0.04$
- the probability of selecting class 4 (picking up a photo of none of the above) is $\pi_4 = 0.44$

You will notice $\pi_1 + \pi_2 + \pi_3 + \pi_4 = 1$ - this is because the classes are mutually exclusive and exhaustive, one and exactly one of these classes must occur in each and every trial.

Given all this, let us ask the question: what is the probability that in a set of 10 random trials, class 1 occurs, say 1 time, class 2 occurs 2 times, class 3 occurs 1 time and class 4 occurs in the remaining 6 times. This is the kind of problem multinomial distributions deal with.

Formally,

- Let C_1, C_2, … , C_m be a set of m classes such that in any random trial, exactly one of these classes will get selected with respective probabilities π_1, π_2, … , π_m.
- Let X_1, X_2, …, X_m be a set of random variables. X_i corresponding to the number of occurrences of class C_i in a set of n trials.
- Then the multinomial probability function, depicting the probability that class C_1 is selected k_1 times, class C_2 is selected k_2 times, class C_3 is selected k_m times:

$$p\left(X_1 = k_1, X_2 = k_2, \cdots, X_m = k_m\right) = \frac{n!}{k_1!\, k_2! \cdots, k_m!} \pi_1^{k_1} \pi_2^{k_2} \cdots \pi_m^{k_m} \tag{5.33}$$

where

$$\sum_{i=1}^{m} k_i = n$$

$$\sum_{i=1}^{m} \pi_i = 1$$

One can verify that for $m = 2$ this becomes the binomial distribution (i.e, equation 5.28).

One noteworthy point here that if we look at any one of the m variables, X_1, X_2, … , X_m individually- its distribution is binomial.

Let us work out the final probability for the example we started with, viz., the probability that in a set of 10 random trials, class 1 occurs, say 1 time, class 2 occurs 2 times, class 3 occurs 1 times and class 4 occurs in the remaining 6 times. This is

$$p\left(X_1=1, X_2=2, X_3=1, X_4=6\right)=\frac{10!}{1!\,2!\,1!\,6!}\left(0.1\right)^1\left(0.42\right)^2\left(0.04\right)^1\left(0.44\right)^6=0.0129$$

Listing 5.10: PyTorch code to compute logarithm of the probability of multinomial distribution.

```
from torch.distributions import Multinomial

num_trials = 10
P = torch.tensor([0.1, 0.42, 0.04, 0.44], dtype=torch.float)

multinom_dist = Multinomial(num_trials, probs=P)

X = torch.tensor([1, 2, 1, 6], dtype=torch.float)

def raw_eval(X, n, P):
    f = math.factorial
    result = f(n)
    for p, x in zip(P, X):
        result *= (p ** x) / f(x)
    return math.log(result)

log_prob = multinom_dist.log_prob(X)
raw_eval_log_prob = raw_eval(X, num_trials, P)
assert torch.isclose(log_prob, raw_eval_log_prob, atol=1e-4)
```

Import PyTorch Multinomial distribution

Set distribution params

Instantiate Multinomial distribution object

Instantiate single point test dataset

Evaluate probability using PyTorch

Evaluate probability using formula

Assert that probabilities match

Fully functional code for multinomial distribution, executable via Jupyter-notebook, can be found at https://github.com/krishnonwork/mathematical-methods-in-deep-learning-ipython/blob/master/python/ch5/5.9.4-multinomial-distribution.ipynb

EXPECTED VALUE OF MULTINOMIAL DISTRIBUTION

Each of the random variables X_1, X_2, … , X_m, individually subscribe to a binomial distribution.

Accordingly, following the binomial distribution expected value formula from equation 5.31

$$\mathbb{E}_{multinomial}\left(X_i\right)=n\ \pi_i \tag{5.34}$$

VARIANCE OF MULTINOMIAL DISTRIBUTION

The variation of the random variables X_1, X_2, … , X_m, following the binomial distribution variance formula from equation 5.32

$$var_{multinomial}\left(X_i\right)=n\pi_i\left(1-\pi_i\right) \tag{5.35}$$

If each of the $X1$, $X2$,… , Xm is a scalar, then one can think of a random vector $X = \begin{bmatrix} X_1 \\ X_2 \\ \vdots \\ X_m \end{bmatrix}$. The expected value of such a random variable is

$$\mathbb{E}_{multinomial}(X) = \begin{bmatrix} n\pi_1 \\ n\pi_1 \\ \vdots \\ n\pi_m \end{bmatrix}$$

where the diagonal terms are like the binomial variance $\sigma_{ii} = n\ \pi_i (1 - \pi_i)\ \forall i \in [1,m]$ and off diagonal terms are $\sigma_{ij} = -n\ \pi_i\ \pi_j \forall\ (i, j) \in [1,m] \times [1,m]$. The cross covariance terms in the diagonal are negative because increase in one element implies decrease in others.

$$\mathbb{C}_{multinomial}(X) = \begin{bmatrix} \sigma_{11} & \sigma_{12} & \cdots \sigma_{1m} \\ \sigma_{21} & \sigma_{22} & \cdots \sigma_{2m} \\ & & \cdots \\ \sigma_{m1} & \sigma_{m2} & \cdots \sigma_{mm} \end{bmatrix} \tag{5.36}$$

Listing 5.11: PyTorch code to compute mean and variance of multinomial distribution.

```
num_samples = 100000                                   # Number of sample points
samples = multinom_dist.sample([num_samples])          # 100000 × 4 tensor; Obtain samples from multinom_dist instantiated in 5.10
sample_mean = samples.mean(axis=0)                     # 1 × 4 tensor; Sample Mean. axis=0 computes mean for each column
dist_mean = multinom_dist.mean                         # Mean via PyTorch function
assert torch.allclose(sample_mean, dist_mean, atol=0.2)
sample_var = multinom_dist.sample([num_samples]).var(axis=0)   # Sample Variance. axis=0 computes variance for each column
dist_var = multinom_dist.variance                      # Variance via PyTorch function
assert torch.allclose(sample_var, dist_var, atol=0.2)
```

5.9.5 Bernoulli Distribution

Bernoulli distribution is a special case of binomial distribution, where $n = 1$, i.e., a single success or fail type trial is performed [28]. The probability of success is π and probability of failure is $1-\ \pi$. In other words, let X be a discrete random variable which takes the value 1

[28] Here, we have gone against the standard practise of introducing the simpler Bernoulli distribution first

(success) with probability π and the value 0 (failure) with probability $1 - \pi$. The distribution of X is Bernoulli distribution.

$$p\,(X = 1) = \pi$$
$$p\,(X = 0) = 1 - \pi$$

Listing 5.12: PyTorch code to compute logarithm of the probability of Bernoulli distribution.

```
from torch.distributions import Bernoulli
                                                    # Import PyTorch Bernoulli distribution
p = torch.tensor([0.3], dtype=torch.float)          # Set distribution params
                                                    # Instantiate Bernoulli distribution object
bern_dist = Bernoulli(p)

X = torch.tensor([1], dtype=torch.float)            # Instantiate single point test dataset

def raw_eval(X, p):
    prob = p if X == 1 else 1-p
    return math.log(prob)

log_prob = bern_dist.log_prob(X)                    # Evaluate probability using PyTorch
raw_eval_log_prob = raw_eval(X, p)                  # Evaluate probability using formula
assert torch.isclose(log_prob, raw_eval_log_prob, atol=1e-4)   # Assert that probabilities match
```

Fully functional code for multinomial distribution, executable via Jupyter-notebook, can be found at https://github.com/krishnonwork/mathematical-methods-in-deep-learning-ipython/blob/master/python/ch5/5.9.5-bernoulli-distribution.ipynb

EXPECTED VALUE OF BERNOULLI DISTRIBUTION

If there are only 2 classes, *success* and *failure*, one cannot speak directly of expected value. If we run, say 100 trials and get 30 *successes* and 70 *failures*, the average is 0.3 *success* which is not even a valid outcome. One cannot have fractional *success* or *failure* in this binary system.

We can, however, talk of expected value of Bernoulli distribution if we introduce an artificial construct. That is, if we assign numerical values to these binary entities, say, *success* = 1 and *failure* = 0. Then the expected value of X is

$$\mathbb{E}\,(X) = \sum_{x \in \{0,1\}} x p\,(x) = 1 \cdot \pi + (1 - \pi) \cdot 0 = \pi \tag{5.37}$$

VARIANCE OF BERNOULLI DISTRIBUTION

Similarly, , if we assign numerical values to these binary entities, say, *success* = 1 and *failure* = 0, the variance of the Bernoulli distribution

$$var(X) = \sum_{x \in \{0,1\}} (x - \mathbb{E}(X))^2 p(x) = (1-\pi)^2 \pi + (0-\pi)^2 (1-\pi) = \pi(1-\pi) \quad (5.38)$$

Listing 5.13: PyTorch code to compute mean and variance of Bernoulli distribution.

```
num_samples = 100000                                   # Number of sample points

samples = bern_dist.sample([num_samples])              # Obtain samples from bern_dist instantiated in 5.12
                                                       # 100000 × 1 tensor
sample_mean = samples.mean()                           # Sample Mean
dist_mean = bern_dist.mean                             # Mean via PyTorch function
assert torch.isclose(sample_mean, dist_mean, atol=0.2)

                                                       # Sample Variance
sample_var = bern_dist.sample([num_samples]).var()
dist_var = bern_dist.variance                          # Variance via PyTorch function
assert torch.isclose(sample_var, dist_var, atol=0.2)
```

5.9.6 Categorical Distribution and one-hot vectors

Consider again the example problem introduced in section 5.9.4 to study multinomial distribution.

We have a database of photos. There are 4 classes of photos

- photos of Albert Einstein (class 1) - say 10% of the photos belong to this class
- photos of Marie Curie (class 2) - say 42% of the photos belong to this class
- photos of Carl Friedrich Gauss (class 3) - say 4% of the photos belong to this class
- other photos (class 4) - remaining 44% of the photos belong to this class
- If we randomly select a photo from the database, i.e., perform a single random trial,
- the probability of selecting class 1 (picking up an Einstein photo) is $\pi_1 = 0.1$
- the probability of selecting class 2 (picking up a Marie Curie photo) is $\pi_2 = 0.42$
- the probability of selecting class 3 (picking up a Gauss photo) is $\pi_3 = 0.04$
- the probability of selecting class 4 (picking up a photo of none of the above) is $\pi_4 = 0.44$

As before $\pi_1 + \pi_2 + \pi_3 + \pi_4 = 1$ - this is because the classes are mutually exclusive and exhaustive, one and exactly one of these classes must occur in each and every trial. In multinomial, we were performing n trials and we asked how many times each specific class will occur. What if we performed only 1 trial? Then we get categorical distribution. Categorical distribution is a special case of multinomial distribution (with number of trials $n = 1$).

It is also an extension of Bernoulli distribution where instead of just two classes *success* and *failure* we can have an arbitrary number of classes.

Formally,

- Let $C_1, C_2, \ldots, C_m$ be a set of m classes such that in any random trial, exactly one of these classes will get selected with respective probabilities $\pi_1, \pi_2, \ldots, \pi_m$. We sometimes

refer to the probabilities of all the classes together as a vector $\vec{\pi} = \begin{bmatrix} \pi_1 \\ \pi_2 \\ \vdots \\ \pi_m \end{bmatrix}$

- Let $X_1, X_2, \ldots, X_m$ be a set of random variables. X_i corresponding to the number of occurrences of class C_i in a set of n trials.
- Then the categorical probability function depicts the probabilities of each of the classes C_1, C_2 etc. occurring in a single trial.

ONE HOT VECTOR

One can use a one-hot vector to compactly express the outcome of a single trial of categorical distribution. This will be a vector with m elements. Exactly a single element is 1, all other elements are 0. The 1 indicates which of the m possible classes occurred in that specific trial. For instance, in the example of database of photos problem, if a Marie Curie photo comes up in a given trial, the corresponding one-hot vector could be $\vec{x} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$.

PROBABILITY OF CATEGORICAL DISTRIBUTION

We can think of a one-hot vector X as a random variable with a categorical distribution. Note that each individual class follows a Bernoulli distribution. The probability of class C_i occurring in any given trial

$$p(C_i) = \pi_i$$

We can express the probability distribution of all the classes together in a compact way.

$$p\left(X = \vec{x}\right) = \pi_1^{x_1} \pi_2^{x_2} \cdots \pi_m^{x_m} = \prod_{i=1}^{i=m} \pi_i^{x_i} \tag{5.39}$$

where $\vec{x}$ is a one-hot vector. Note that all but one of the powers in equation 5.39 is zero, hence the corresponding factor evaluates to one. The remaining power is 1. Hence the overall probability always evaluates to p_i where i is the index of the class that actually occurred in the trial.

EXPECTED VALUE

Since we are talking of classes here, expected value and variance does not make much sense in this context. We encountered a similar situation with Bernoulli distribution. There we assigned numerical values to each class and somewhat artificially defined expected value and variance. A similar idea can be applied here too. We can talk of expected value and variance of the one-hot vector (which consists of numerical values 0 and 1). But it remains a bit of an artificial construct.

Given a random variable X whose instances are one-hot vectors $\vec{x}$ following a categorical distribution with m classes with respective probabilities $\pi_1, \pi_2, \ldots, \pi_m$

$$\mathbb{E}(X) = \vec{\pi} = \begin{bmatrix} \pi_1 \\ \pi_2 \\ \vdots \\ \pi_m \end{bmatrix} \tag{5.40}$$

We will skip the variance of categorical distribution here.

5.10 Chapter Summary

In this chapter we took a first look at probability and statistics from a machine learning point of view. We also introduced the PyTorch distributions package and each concept has been illustrated with PyTorch distributions code samples immediately following the math. Also, fully functional code in the form of ipython notebooks that can be downloaded and run via python Jupyter notebook can be found at https://github.com/krishnonwork/mathematical-methods-in-deep-learning-ipython/tree/master/python/ch5

- We started with an intuitive introductions to probability, random variables and probability distributions following the frequentist paradigm.
- We studied various properties of probabilities, including the sum rule of probabilities, the concept of mutually exhaustive and independent events and the product rule
- We studied joint probabilities, marginal probabilities, sampling, geometric nature of sample point distributions for to independent vs correlated events and its connection to Principal Component Analysis.
- We studied expected value, variance and covariance matrices, their geometric significances and connection to machine learning
- We studied uni and multi variate forms Uniform, Gaussian, Binomial, Multinomial, Bernoulli and Categorical distributions. Physical and geometric significance of each parameter was explained. The Gaussian (aka Normal) distribution was studied in great detail, including geometry of its sample point clouds. In the github repository, we have provided an interactive visualizer for observing the shapes of Gaussian distributions, both in 1 and 2 dimensions, as one changes the parameter values. Take a look at the interactive visualization section in https://github.com/krishnonwork/mathematical-methods-in-deep-learning-ipython/blob/master/python/ch5/5.9.2-normal-distribution.ipynb