Lecture Notes:
Introduction to Information Theory

Changho Suh[1]

December 4, 2019

[1]Changho Suh is an Associate Professor in the School of Electrical Engineering at Korea Advanced Institute of Science and Technology, South Korea (Email: chsuh@kaist.ac.kr).

September 2, 2019
Changho Suh (chsuh@kaist.ac.kr)

Lecture 1: Logistics and overview

About instructor

Welcome to EE623: Information Theory! My name is Changho Suh, an instructor of the course. A brief introduction of myself. A long time ago, I was one of the students in KAIST like you. I spent six years at KAIST to obtain the Bachalor and Master degrees all from Electrical Engineering in 2000 and 2002, respectively. And then I left academia, joining Samsung electronics. At Samsung, I worked on the design of wireless communication systems like 4G-LTE systems. Spending four and a half years, I then left industry, joining UC-Berkeley where I obtained the PhD degree in 2011. I then joined MIT as a postdoc, spending around one year. And then I came back to KAIST. While in Berkeley and MIT, I worked on a field, so called *information theory*, which I am going to cover in this course. This is one of the main reasons that I am teaching this course.

Today's lecture

In today's lecture, we will cover two very basic stuffs. The first is logistics of this course: details as to how the course is organized and will proceed. The second thing to cover is a brief overview to this course. In the second part, I am going to tell you a story of how the information theory is developed, as well as what the theory is.

My contact information, office hours and TAs' information

See syllabus uploaded on the course website. One special note: if you cannot make it neither for my office hours nor for TAs' ones, please send me an email to make an appointment in different time slots.

Prerequisite

The basic prerequisite for this course is familiarity with the concept on *probability and random processes*. In terms of a course, this means that you should have taken the following course: EE210, which is an undergraduate-level course on probability. This is the one that is offered in Electrical Engineering. Some of you from other departments might take a different yet equivalent course (e.g., MAS250). This is also okay.

There must be a reason as to why the concept of probability is crucial for this course. Here is why. Information theory that we will cover in this course was developed in the context of *communication*, which I am going to tell you with greater details in the second part of this lecture. So the communication is the field where one can see relationship between information theory and probability.

Now what is communication? Communication is the transfer of information from one end (called *transmitter*) to the other (called *receiver*), over a physical medium (like an air) between the two ends. The physical medium is so called the *channel*. Here the channel is the one that relates the concept of probability to communication. If you think about how the channel behaves, then you can easily see why. First of all, the channel is a sort of *system* (in other words, a function) which takes a transmitted signal as an input and a received signal as an output. Here one big problem arises in the system. The problem is that it is not a *deterministic* function.

If the channel is deterministic and one-to-one mapping, then one can easily reconstruct the input from the output. So there is no problem in transferring information from the transmitter to the receiver. However, the channel is not deterministic in reality. It is actually a *random* function. In other words, there is a random entity (also known as *noise* in the field) added into the system. In typical communication systems, the noise is additive: a received signal is the sum of a transmitted signal and the noise. In general, we have no idea of the noise. In mathematics or statistics, there is a terminology which indicates such a random quantity. That is, *random variable* or *random process*, which the probability forms the basis for. This is the very reason that this course requires a deep understanding on probability. Some of you may have no idea of the random processess although you took EE210. Please don't be offended. It is nothing but a sequence of random variables. Whenever the concept on random process comes up, I will provide detailed explanations and/or materials which serve you to understand the contents required.

If you think that you lack this prerequisite, then come and consult with me so that I can help you as much as possible. If you did not take this source, but if you took relevant courses held in other departments, then probably it would be okay.

There is another important course which help understanding this course. That is EE528, which is a course on random processes. This is a graduate-level course. It deals with the probability in a deeper manner and also covers many important concepts on random processes. So if you have enough energy, passion and time, I strongly recommend you to take this course simultaneously. But notice that is not a prerequisite.

Course website

We have a course website on the KLMS system. You can simply login with your portal ID. If you want to only sit in this course (or cannot register the course for some reason), please let me or one of TAs know your email address. Upon request, we are willing to distribute course materials to the email address that you sent us.

Text

The course consists of three parts. A textbook that we will use for the first and second parts is: Cover and Thomas's book, which is the most renowned text in the field, titled "Elements of Information Theory." I am going to call it CT for short. We will also use a celebrated paper written by the Father of Information Theory: Claude E. Shannon. Here is the title of the paper: "A mathematical theory of communication". We will also use lecture notes by Prof. Robert G. Gallager, which we are going to use for the first few weeks. You can download them from the course website. On top of these three, I am going to provide you with lecture slides which I will use during lectures, as well as course notes like the one that you are now reading. Perhaps these materials will be posted at night on a day before class.

There is no textbook for the last part. But don't worry. I will instead provide you with lecture slides and course notes which contain every details and thus are self-contained. Reading these materials would suffice for you to understand all the contents covered in the last part.

Problem sets

There will be weekly or bi-weekly problem sets. So there would be seven to eight problem sets in total. Solutions will be usually available at the end of the due date. This means that in principle, we do not accept any late submission. We encourage you to cooperate with each other in solving the problem sets. However, you should write down your own solutions by yourselves.

You are welcome to flag confusing topics in the problem sets; this will not lower your grade.

Exams

As usual, there will be two exams: midterm and final. Please see syllabus for the schedule that our institution assigns by default. Please let us know if someone cannot make it for the schedule. With convincing reasons, we can change the schedule or can give a chance for you to take an exam in a different time slot that we will organize individually.

Two things to notice. First, for both exams, you are allowed to use one cheating sheet, A4-sized and double-sided. So it is a kind of semi-closed-book exam. Second, for your convenience, we may provide an instruction note for each exam (if you wish), which contains detailed guidelines as to how to prepare for the exam. Such information includes: (1) how many problems are in the exam; (2) what types of problems are dealt with in what contexts; (3) the best way to prepare for such problems.

Course grade

Here is a rule for the course grade that you are mostly interested in perhaps. The grade will be decided based on four factors: problem sets (22%); midterm (32%); final (40%); and interaction (6%). Here the interaction means any type of interaction. So it includes attendance, in-class participation, questions, and discussion.

Overview

Now let's move onto the second part. Here is information for reading materials: Chapter 1 in Gallager's notes, Shannon's paper, and Chapter 2 in CT. In this part, I will tell you how the information theory was developed and what the theory is. As I mentioned earlier, information theory was developed in the context of communication. So first I am going to tell you how the information theory was developed in that context. Perhaps next time, I will provide you with specific topics that we will learn throughout the course.

History of communication systems

To talk about a story of how information theory was developed, we need to first know about the history of communication systems which in turn led to the foundation of information theory. Before getting into details, recall the terminologies that I mentioned earlier. Remember that communication is the transfer of information from one end to the other end. Here the one end is called transmitter, and the other end is called receiver. Something that lies in between is a physical medium, called the channel.

As you can easily image, communication has a long history. Even in the beginning of the world, there was a communication, which is a dialogue between people. The dialogue, also called conversation, is definitely a type of communication although it is a very naive way of communication and has nothing to with electrical engineering. Actually there was a breakthrough in the history of communication, which is now related to the communication systems that have something to do with electrical engineering and so we are interested in. This breakthrough was the invention of telegraph. *Morse* code[1] is the first such example: a very simple transmission scheme that is initially used in telegraph. Actually this invention was based on a simple observation (discovered in physics) that electrical signals, like voltage or current signals can be transmitted

[1] Regarding the code, don't be confused with computer programming languages (such as C++ and Python) that have nothing to do with it. *Code* is a sort of terminology used in the context of communication which indicates a transmission scheme.

over wires such as copper lines. So it is the first communication system that have something to do with electrical signals. Actually this is the main reason as to why communication systems have been studied within the field of electrical engineering where most of you guys are in. Later this technology was further developed. In the 1870s, Alexander Graham Bell invented a more advanced version of such systems, called *telephone*.

Communication systems were further upgraded. The upgrade was based on another finding in physics: not only we can send electrical signals over wires, but we can also send them in a *wireless* environment through electromagnetic waves, simply called radio waves. This finding inspired an Italian engineer at that time, named Guglielmo Marconi. He developed a wireless version of telegraph, called wireless telegraphy. Later this technology was further developed, which led to the invention of radio and TV.

State of the affairs in early 20th century

These are the communication systems that were developed in the early 20th century: telegraph, telephone, wireless telegraph, radio and TV. Actually at this time, there was one guy who could make some interesting observations on these communication systems, which in turn made him do some great work in communication. The guy was Claude E. Shannon, known as the Father of Information Theory. He made the following two observations.

The first is that most communication systems at that time are analog. They directly dealt with signals of interest. For example, in radio systems, signals of interest are audio waveforms. In TV, such interested signals are image pixels. Obviously these are analog continuous signals because the signals are based on voltage or current signals which are definitely analog.

The second observation that he made is that engineering designs are pretty ad-hoc, being tailored for each specific application. This means that design principles were completely different depending on signals of interest.

Claude E. Shannon

What he felt from such observations is that such communication systems are really annoying. He did not like the communication systems because such systems are designed in a pretty ad-hoc manner (no systematic design principle!), in other words, in a completely different manner depending on each specific application of interest. He actually believed that there must be one simple & beautiful framework that can unify all such different communication systems. So he tried to unify the ad-hoc approach. As a specific effort, he raised the following three big questions.

Shannon's three big questions

The first question is the most fundamental question regarding the *possibility* of unification.

Question 1: Is there a general unified methodology for designing communication systems?

The second question is a natural follow-up question which helps addressing the first question. He thought that if unification is possible, then there may exist a *common currency* (like dollar in the context of economics) w.r.t. (with respect to) information. In the communication systems, we have a variety of different information sources, like text, voice, video, images. The second question that he asked is related to the existence of such common currency.

Question 2: Is there a common currency of information that can represent all such different information sources?

The last question is with respect to the task of interest: communication.

Question 3: Is there a limit to how fast one can communicate?

He had spent around eight years to address the above questions. Perhaps he had quite a difficult time as the long period of eight years indicates. But he could make a progress in the end. He could address all the big questions in a single stroke. In the process of doing this, he could develop a theory, later people call "information theory".

What Shannon did

Specifically what he did are three-folded. First of all, he showed that the answer to the second question is yes! He came up with a common currency of information that can represent all of different types of information sources. And then based on the common currency, he addressed the first question. He developed a single beautiful framework that can unify all such different communication systems that were prevalent in the days. Under this unified framework, he then answered the last question. He demonstrated that there exists a limit on the amount of information (represented in terms of the common currency - to be detailed later) that one can maximally communicate. Not only that, he could also characterize the limit.

Especially in the process of characterizing the limit, he could make an interesting observation that the limit is a *sole* function of a channel, regardless of any transmission and reception strategy. This means that given a channel, there exists a fundamental limit on the amount of information that we can transmit under which communication is possible, and below which communication is impossible no matter what and whatsoever. This quantity is never changed given a channel. It is like a fundamental *law* that is dictated by the *Nature*. So it is like a law in physics. Shannon was excited about this. So he theorized the law in a mathematical framework, and called it "a mathematical theory of communication", which became the title of his landmark paper. Later people call it information theory or Shannon theory.

Looking ahead

Next time, I will tell you how Shannon could do such things. Then, I am going to tell you what specific topics we are going to cover in this course.

September 4, 2019
Changho Suh (chsuh@kaist.ac.kr)

Lecture 2: Overview

Recap

Last time I told you about a story of how Claude Shannon came up with information theory. The story dates back to the early 20th century when a variety of different communication systems are prevalent. Motivated by the fact that most of communication systems at that time were designed in an ad-hoc manner, Shannon intended to develop a single framework that can unify all the different systems. To this end, he first came up with a common currency of information. Using this, he then established a unified framework. Based on the framework, he characterized the limit to how fast one can communicate. In the process of doing this, he was particularly excited by the fact that given a channel determined by the *Nature*, the limit does not change no matter what transmitter and receiver do.

Shannon theorized all of these in a single mathematical framework. He then called it "a mathematical theory of communication", which people later called information theory or Shannon theory. This was how information theory was invented.

Today's lecture

Today we are going to go a little bit deeper. Specifically I will tell you how Shannon did all of these in detail. Based on this, I will then list up specific topics that we are going to cover in this course.

Communication architecture

Recall the three terminologies that we introduced last time: transmitter, receiver, and something that lies in between transmitter and receiver, which is the channel. Now consider information signals that one wishes to transmit, such as text, voice, and image pixels. Here is another terminology which indicates such signals: "information source". What Shannon thought in the first place is that there must be something which processes the information source before transmission. He abstracted this process with a black box that he called "encoder". Of course there must be something at receiver which attempts to recover the information source from the received signals that the channel yields. He abstracted the process at the receiver end with another black box that he called "decoder". This is the very first basic block diagram that Shannon imagined for a communication architecture. See Fig. 1. From a Shannon's viewpoint, a communication system is nothing but a collection of encoder and decoder, so designing a communication system is concerning how to develop such encoder and decoder.

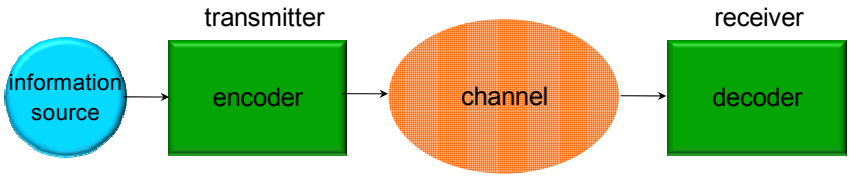


Figure 1: A basic communication architecture.

Representation of information source

With this simple picture in his mind, Shannon wished to unify all the different communication systems that were prevalent in the early 20th century. Most engineers at that time attempted to transmit information sources (which are of course different depending on applications of interest) *directly* without any significant modification. Shannon thought that this is the very reason as to why there were different communication systems.

He then thought that for the purpose of unification, there must be at least something that can represent different information sources. He believed that the common thing would lead to a universal framework. It turns out Shannon's work on master thesis in MIT could play a significant role to find a universal way of representing information source. His master thesis was about Boolean algebra. What he did specifically is that any logical relationship in circuit systems can be represented with 0/1 logic, in other words, binary string.

Inspired by this, Shannon thought that the same thing may happen in the context of communication systems, meaning that any type of information source can be represented with binary string. He showed that it is indeed the case. Specifically what he showed is that the binary string that he called "bits" can represent the meaning of information.

For instance, suppose that information source is an English text that comprises English letters. Now how to represent each English letter with a binary string? Here one key observation is that there are only a *finite* number of candidates that each letter can take on. This number is the total number of English alphabets, which is 26.[1] From this, we see that $\lceil \log_2 26 \rceil = 5$ number of bits suffices to represent each letter.

A two-stage architecture

This observation led Shannon to introduce bits as a common currency of information. Specifically here is what he did. He first thought that we need a block which converts information source into bits of our interest. Motivated by this, Shannon came up with a two-stage architecture in which the encoder is split into two parts. Here the role of the first block is to convert information source into bits. Shannon called the block "source encoder", as it is a part of the encoder as well as is related to how information *source* looks like. The role of the second block is to convert bits into a signal that can actually be transmitted over a channel. Shannon called it "channel encoder" because it is obviously concerning a channel. See the upper part in Fig. 2.

Similarly, receiver consists of two stages. But the way it is structured is opposite. In other words, we first convert the received signals into the bits that we sent at the transmitter (channel decoder). Next, we reconstruct the information source from the bits (source decoder). As you can see, this block is nothing but an inverse function of source encoder. Obviously source encoder should be one-to-one mapping; otherwise, there is no way to reconstruct the information source. See the lower part in Fig. 2.

Actually there is a terminology which indicates the part spanning from channel encoder, channel, to channel decoder. We call it "digital interface". Here one thing to notice is that this digital interference is universal in a sense that it has nothing to do with type of information source because the input to the digital interference is always bits. So in that regard, it is indeed a unified communication architecture.

Two questions on the fundamental limits

[1] Here we ignore any special characters such as space.

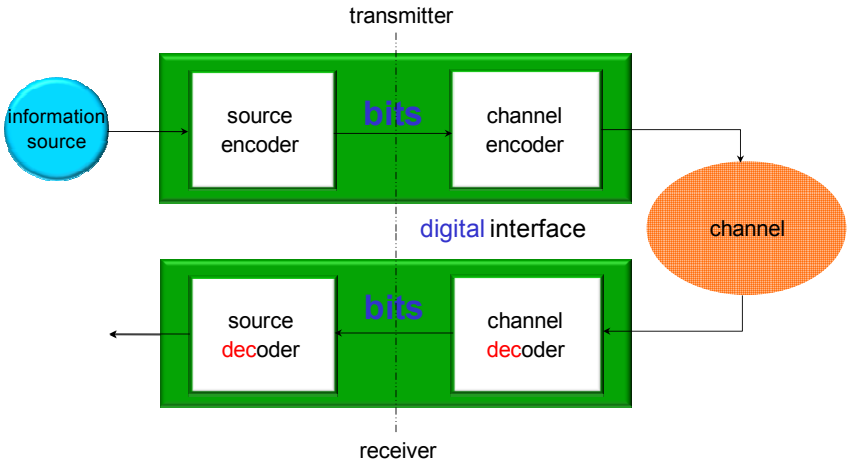


Figure 2: A two-stage communication architecture.

Keeping the two-stage architecture in his mind, Shannon tried to address the third question: is there a limit to how fast one can communicate? To this end, Shannon first thought about the goal of communication: transferring information as much as possible. Actually if information source itself is bits, then we only need to worry about the maximum number of bits that can be transmitted over a channel. But here notice in the two-staged architecture that we have another block in front, that is, source encoder which converts the information source into bits.

This naturally led him to worry about *efficiency* of the first block. In order to maximize the amount of information to send, first we need to minimize the number of bits that can represent the information source. This led Shannon to worry about *what the most efficient way of representing the information source is*.

To make this more explicit, Shannon split the question on the limit into two. The first is about efficiency of the first block. What is the minimum number of bits that can represent information source? The second question is about transmission capability. What is the maximum number of bits that can be transmitted over a channel?

Shannon developed two theorems to answer the two questions. He first developed a theorem that he called "source coding theorem" to identify what the minimum number of bits is. He also developed another theorem that he called "channel coding theorem", in which the maximum transmission capability is characterized.

Source coding theorem

Let us first explore what the source coding theorem is. Information source is a kind of sequence. For example, it is a sequence of English alphabets, a sequence of audio signals, or a sequence of image pixels. Let that sequence be $\{S_i\}$. Actually one important view that Shannon took is that he interpreted this sequence as a sequence of *random* quantities, in other words, a sequence of random variables, or simply a random process. The rationale behind this viewpoint is that the information source is the one that has *uncertainty* from a receiver perspective. Note that it is unknown to the receiver. Then, one can readily expect that the minimum number of bits that represent the random process depends on the probabilistic property that the random process has: *joint distribution*.

For simplicity, let's think about a simple case in which the joint distribution of the random process is greatly simplified. One such case is the one in which S_i's are independent and identically distributed, simply called the i.i.d. case. It turns out in this case, the source coding theorem is

very simple to state.

Let S be a generic random variable which indicates an individual random variable that forms the random process. In the source coding context, each random variable is called "symbol". It turns out the minimum number of bits is related to one important notion that is very well-known in other fields such as physics and chemistry. That is "entropy". The entropy is a kind of measure of disorder or randomness. It turns out this measure plays a crucial role to establish the source coding theorem, formally stated below.

Theorem 0.1 (Source coding theorem in the i.i.d. case) *The minimum number of bits that can represent the source per symbol is the entropy of the random variable S, denoted by*

$$H(S) := \sum_{s \in \mathcal{S}} p(s) \log_2 \frac{1}{p(s)}. \tag{1}$$

where $p(s)$ denotes the distribution[2] of S, and $\mathcal{S}$ (that we call "caligraph of S") indicates the range, which is the set of all possible values that S can take on.

Source code example

To give you a concrete feel about why the source coding theorem makes sense, let me give you a source code example in which one can achieve the limit promised as (1). The design of source code means the specification of functional relationship between input S and output, say $f(S)$, in the source encoder. In the source coding context, such output $f(S)$ is called "codeword". Let's think about a simple case in which the information source is DNA sequence where each symbol S can take on one of the four letters: A, C, T, G. For simplicity, let's assume an unrealistic yet simple setting in which the random process $\{S_i\}$ is independent and identically distributed (i.i.d.), each S_i being according to the following distribution:

$$S = \begin{cases} A, & \text{with probability (w.p.) } \frac{1}{2}; \\ C, & \text{w.p. } \frac{1}{4} \\ T, & \text{w.p. } \frac{1}{8}; \\ G, & \text{w.p. } \frac{1}{8}. \end{cases}$$

Now the question is: how to design $f(S)$ such that the number of bits that represents S, reflected in the average length of codeword $\mathbb{E}[f(S)]$, is minimized? Note that there are four letters in total. So it suffices to use two bits to represent each symbol. One naive way is to take the following simple mapping rule: $A \to 00$; $C \to 01$; $T \to 10$; $G \to 11$. This way, we can achieve 2 bits/symbol. But the source coding theorem says that we can actually do better, as the claimed limit reads:

$$H(S) = \frac{1}{2} \cdot 1 + \frac{1}{4} \cdot 2 + \frac{1}{8} \cdot 3 + \frac{1}{8} \cdot 3 = 1.75. \tag{2}$$

As the theorem promised, there is indeed a code that can achieve the limit. The code is based on the following observations: (i) the letter A occurs more frequently than other letters; (ii) the length of codeword is not necessarily fixed. This naturally leads to an idea: assigning a short-length codeword to frequent letters while assigning a long-length codeword to less frequent letters. Now the question is how to implement such an idea? For visualization purpose, let us introduce a so-called "binary code tree" with which one can easily implement a mapping from S to $f(S)$.

[2]It is a probability mass function, simply called pmf, for the case where S is a discrete random variable.

Binary code tree: A binary tree is the one in which every internal node has only two branches. Notice that a node which has no branch is called the *leaf*. See an example in Fig. 3. Now let me relate the binary tree to a code. Suppose we assign a symbol to a leaf. We can then establish the functional relationship by specifying the pattern of the corresponding codeword as follows: the sequence of binary labels (associated with branches) from the root to that leaf. For example, suppose we label 0 on an upper branch, 1 on a lower branch, and assign a symbol A to the top leaf. Then, $f(A) = 0$, as we have only one branch, labeled 0, that links the root to the leaf. Similarly $f(C) = 10$. Note that there are two branches (labeled 1 and 0 subsequently) which connect the root to the leaf assigned to C.

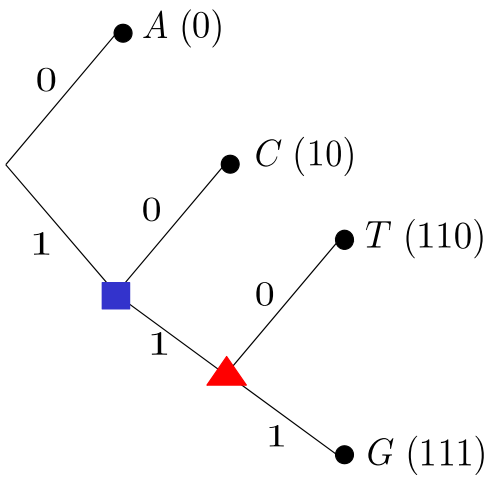


Figure 3: Representation of an optimal source code via a binary code tree.

Now how to implement an optimal mapping rule (which achieves 1.75 bits/sym) via a binary code tree. As mentioned earlier, the idea is to assign a short-length codeword to frequent letters. Obviously we should assign the most frequent letter A to the top leaf, since it has the shortest codeword length. Now what about for the second most frequent letter C? One may want to assign it to an internal node marked in a blue square in Fig. 3. But it is not valid. Why? That way, the codeword pattern ran out - this is problematic because we have only two leaves although we need four in total. So we must have another two branches generating from the internal node. We can then assign the C to the second top leaf of codeword length 2. Similarly the next frequent letter T (or G) cannot be assigned to a follow-up internal node marked in a red triangle in Fig. 3. So the node should have another set of two branches. The remaining letters T and G are now assigned to the two remaining leaves. With this mapping rule, one can readily see that

$$\begin{aligned}\mathbb{E}[\text{length}(f(S))] &= \Pr(S=A)\text{length}(f(A)) + \Pr(S=C)\text{length}(f(C)) \\ &\quad + \Pr(S=T)\text{length}(f(T)) + \Pr(S=G)\text{length}(f(G)) \\ &= \frac{1}{2}\cdot 1 + \frac{1}{4}\cdot 2 + \frac{1}{8}\cdot 3 + \frac{1}{8}\cdot 3 = 1.75 = H(S).\end{aligned}$$

Channel coding theorem

Now let us move onto the channel coding theorem. It says that the maximum number of bits that can be transmitted over a channel is the capacity denoted by C. There is a mathematical definition for such C. In fact, the definition relies on a bunch of concepts and notions that

we will need to study. One such important notion is "mutual information." We will deal with definitions of those later on.

Course outline

The two theorems are parts of the main topics that we are going to cover in this course. Specifically, the course consists of three parts. In Part I, we will study basic concepts on key notions in the field: (i) entropy (which played a fundamental role in establishing the source coding theorem); (ii) mutual information (a crucial notion for the channel coding theorem); (iii) Kullback-Leibler (KL) divergence (another key notion which plays a similar role with mutual information yet is known to more powerful in a widening array of other disciplines such as statistics, mathematics, and machine learning). In Part II, we will prove the two major theorems using the notions of entropy and mutual information. It turns out information theory can serve to address some important issues that arise in a wide variety of fields, not limited to communication, ranging from data science, computational biology, recently to machine learning and deep learning. In Part III, we will explore applications of information theory in machine learning and deep learning. In particular, we will focus on the roles of the above key notions in the design of supervised and unsupervised learning algorithms.

September 9, 2019
Changho Suh (chsuh@kaist.ac.kr)

Lecture 3: Entropy

Recap

During the past lectures, I told you about a story of how Shannon came up with information theory. Shannon's effort was initiated by a strong motivation for unifying distinct communication systems that were prevalent in the early 20th century. As an initial step towards unification, he introduced *bits* as a means to represent possibly different information sources and then proposed a unified two-stage architecture in which the first block converts an information source into bits and the second block generates a signal that can actually be transmitted over a channel. Within this framework, Shannon quantified the limit on the amount of information that one can communicate. In the process of doing this, he established two fundamental theorems which in turn laid the foundation of information theory. The first is the source coding theorem which demonstrates that the minimum number of bits that can represent an information source is the entropy of the source. The second is the channel coding theorem which characterizes the maximum number of bits that can be transmitted over a channel.

Today's lecture

Prior to proving the source and channel coding theorems which we are going to cover in depth in Part II, we will first study four key notions which form the basis of the theorems : (i) entropy; (ii) mutual information; (iii) Kullback-Leibler (KL) divergence; (iv) cross entropy. It turns out these notions play a crucial role to address important issues that arise in a widening array of other disciplines such as statistics, physics, computational biology and machine learning. So it is worthwhile being familiar with detailed properties about the notions for many purposes.

Today we will study in depth about the first notion: Entropy. Specifically what we are going to do are three folded. First we will review the definition of entropy that we already introduced last time. I will then introduce a couple of intuitive interpretations on the meaning of entropy, which may help us to understand the source coding theorem as to why the limit on the number of bits required to represent the information source must be the entropy. Next we will study some key properties which are useful in a variety of contexts not limited to the source coding theorem. Later in Part II you will see how the entropy comes up in the proof of the source coding theorem. Please be patient until I show you such connection.

Definition of entropy

The entropy is defined with respect to (w.r.t.) a random quantity which has uncertainty. More formally, it is concerned about a random variable or a random process depending on the dimension of a random quantity. For simplicity, let us focus on a simple case in which the random quantity is a random variable. Later we will cover a general case dealing with a random process.

More precisely the entropy is defined w.r.t. a *discrete*[1] random variable. Let X be a discrete random variable and $p(x)$ be its probability mass function (pmf). Let $\mathcal{X}$ (that we call "caligraph

[1] We say that a random variable is *discrete* if its range (the set of values that the random variable can take on) is finite or at most countably infinite. See Ch. 2.1 in BT for details.

ex") be its range: the set of values that X can take on. The entropy is defined as:

$$H(X) := \sum_{x \in \mathcal{X}} p(x) \log_2 \frac{1}{p(x)} \qquad \text{bits.} \tag{1}$$

In this course, mostly we will use only one type of a logarithmic function, that is log base 2. For simplicity, we will drop the "2" throughout the course.

Actually there is an alternative expression for the entropy formula that I strongly recommend you to remember. It is the one which is much simpler and thus easy to remember. In addition, it helps greatly simplifying the proof of some important properties that we are going to investigate later on. Note that the entropy is sort of a weighted sum of $\log \frac{1}{p(x)}$. So it can be represented as:

$$H(X) := \mathbb{E}\left[\log \frac{1}{p(X)}\right]. \tag{2}$$

where the expectation is taken over the distribution of X: $p(x)$.

Interpretation #1

Actually there are some well-known interpretations on the meaning of the entropy which are quite intuitive and therefore can give some insights into why the entropy is related to the limit promised by the source coding theorem. Here we will investigate two of them. The first is:

Entropy is a measure of the uncertainty of a random quantity.

Let me give you a concrete example where we can see this interpretation makes sense. Consider two experiments: (i) tossing a fair coin; (ii) rolling a fair dice. One simple random variable that one can naturally think of for the first experiment is a function that maps the head (or tail) event to 0 (or 1). Since the coin is fair, we have:

$$X = \begin{cases} 0, & \text{w.p. } \frac{1}{2}; \\ 1, & \text{w.p. } \frac{1}{2}, \end{cases}$$

where "w.p" stands for "with probability". On the other hand, a natural random variable in the second experiment is a function that maps a dice result to the same number, which in turn yields:

$$X = \begin{cases} 1, & \text{w.p. } \frac{1}{6}; \\ 2, & \text{w.p. } \frac{1}{6}; \\ 3, & \text{w.p. } \frac{1}{6}; \\ 4, & \text{w.p. } \frac{1}{6}; \\ 5, & \text{w.p. } \frac{1}{6}; \\ 6, & \text{w.p. } \frac{1}{6}. \end{cases}$$

Now which random variable is more uncertain, in other words, more random? Your intuition points to the second one! No wonder. Actually the entropy supports your answer with explicit numbers. Note that $H(X) = 1$ in the first experiment while $H(X) = \log 6 > 1$ in the latter. Entropy can indeed *quantify* such uncertainty.

Let me give you another example. Suppose now we have a bent coin. Then, it yields a different head event probability, say $p \neq \frac{1}{2}$:

$$X = \begin{cases} 0, & \text{w.p. } p; \\ 1, & \text{w.p. } 1-p. \end{cases} \tag{3}$$

Now the question is: Does this random variable has more uncertainty, compared to the fair coin case? To see this clearly, consider an extreme case in which $p \ll 1$ and thus:

$$H(X) = p \log \frac{1}{p} + (1-p) \log \frac{1}{1-p} \approx 0. \tag{4}$$

Here we used the fact that $\lim_{p \to 0} p \log \frac{1}{p} = 0$. Why? Remember the Lopital theorem that you learned about from the course on calculus. This implies that the bent-coin case is definitely more certain. This makes sense. Why? Very small p ($\ll 1$) means that the tail event happens almost all the time, and thus the result is well predictable.

Interpretation #2

The second interpretation is related to a common way of removing uncertainty. What does this mean? Let me give you an example. Suppose we meet a person for the first time. Then the person is completely unknown to us. Here we can think of a common way to remove such randomness: asking questions. With answers to the questions, we can then somehow remove randomness w.r.t. him/her. With enough questions, we may have complete knowledge about him/her. So from this, we see that the number of questions required to have complete knowledge represents the amount of randomness: the more questions required, the more uncertain. This naturally leads to:

Entropy is related to the number of questions required to determine the value of X.

Sometimes the number of questions (on average) exactly matches $H(X)$. Here is an example where that happens. Suppose

$$X = \begin{cases} 1, & \text{w.p. } \frac{1}{2}; \\ 2, & \text{w.p. } \frac{1}{4}; \\ 3, & \text{w.p. } \frac{1}{8}; \\ 4, & \text{w.p. } \frac{1}{8}. \end{cases}$$

A straightforward calculation gives $H(X) = 1.75$. Now suppose that questions are of the yes-or-no type. Then, the minimum number of questions (on average) required to determine the value of X is $H(X)$. Here is the optimal way of asking questions that achieves $H(X)$. We first ask whether or not $X = 1$. If yes, $X = 1$; otherwise, we ask if $X = 2$ or not. If yes, $X = 2$; otherwise we ask if $X = 3$ or not. Let $f(x)$ be the number of questions when $X = x$. Then, this way yields:

$$\begin{aligned} \mathbb{E}\left[f(X)\right] &= \frac{1}{2} f(1) + \frac{1}{4} f(2) + \frac{1}{8} f(3) + \frac{1}{8} f(4) \\ &= \frac{1}{2} \cdot 1 + \frac{1}{4} \cdot 2 + \frac{1}{8} \cdot 3 + \frac{1}{8} \cdot 3 = 1.75. \end{aligned}$$

Some of you may recognize that this is the same as the source code example that we examined last time.

Key properties

The entropy has several important properties. We can identify those properties by making some important observations.

Recall the bent-coin example. See (**??**) for the distribution of the associated random variable X. Consider the entropy calculated in (**??**). First we note that the entropy is a function of p which fully describes X. See Fig. **??** which plots $H(X)$ as a function of p. From this, one can

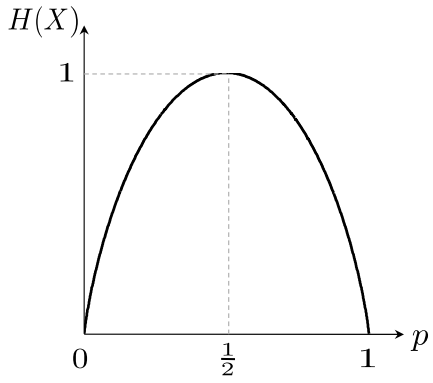


Figure 1: Entropy of the binary random variable X with $\Pr(X = 0) = p$ and $\Pr(X = 1) = 1 - p$

make two observations: (i) the minimum entropy is 0; (ii) the entropy is maximized when $p = \frac{1}{2}$, i.e., X is uniformly distributed.

Consider another example in which $X \in \mathcal{X} = \{1, 2, \ldots, M\}$ and is uniformly distributed:

$$X = \begin{cases} 1, & \text{w.p. } \frac{1}{M}; \\ 2, & \text{w.p. } \frac{1}{M}; \\ \vdots & \\ M, & \text{w.p. } \frac{1}{M}. \end{cases}$$

In this case,

$$H(X) = \sum \frac{1}{M} \log M = \log M = \log |\mathcal{X}|$$

where $|\mathcal{X}|$ indicates the cardinality (the size of the set) of $\mathcal{X}$. This leads to another observation: the entropy of the uniformly distributed random variable $X \in \mathcal{X}$ is $\log |\mathcal{X}|$.

The above three observations lead us to conjecture the following two: for $X \in \mathcal{X}$,

$$\begin{aligned} &\textit{Property 1:} \quad H(X) \geq 0. \\ &\textit{Property 2:} \quad H(X) \leq \log |\mathcal{X}|. \end{aligned}$$

It turns that these properties indeed hold. The first is easy to prove. Using the definition of entropy and the fact that $p(X) \leq 1$, we get: $H(X) = \mathbb{E}[\log \frac{1}{p(X)}] \geq \mathbb{E}[\log 1] = 0$. The proof of the second property is also easy once we rely on one of the very popular inequalities called *Jensen's inequality*[2], formally stated below.

Theorem 0.1 (Jensen's inequality) *For a concave*[3] *function* $f(\cdot)$*,*

$$\mathbb{E}\left[f(X)\right] \leq f\left(\mathbb{E}[X]\right).$$

Proof: The proof is immediate for a simple binary case, say $X \in \mathcal{X} = \{x_1, x_2\}$. By setting $p(x_1) = \lambda$ and $p(x_2) = 1 - \lambda$, we get: $\mathbb{E}[X] = \lambda x_1 + (1 - \lambda)x_2$ and $\mathbb{E}[f(X)] = \lambda f(x_1) + (1 - \lambda) f(x_2)$. Now the definition of the concavity (see the associated footnote for the definition or Fig. **??**) completes the proof. The generalization to an arbitrary $\mathcal{X}$ is not that hard. The idea is by induction. Try this in Problem Set 1. □

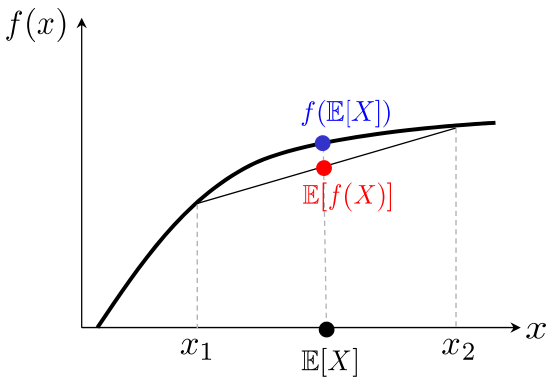


Figure 2: Jensen's inequality for a concave function: $\mathbb{E}[f(X)] \leq f(\mathbb{E}[X])$

Using the definition of entropy and the fact that log is concave, we get:

$$\begin{aligned} H(X) &= \mathbb{E}\left[\log \frac{1}{p(X)}\right] \\ &\leq \log\left(\mathbb{E}\left[\frac{1}{p(X)}\right]\right) \\ &= \log\left(\sum_{x\in\mathcal{X}} p(x)\cdot\frac{1}{p(x)}\right) \\ &= \log|\mathcal{X}| \end{aligned}$$

where the inequality is due to Jensen's inequality.

Joint entropy

Now let's move on to the entropy defined w.r.t. multiple (say two) random variables. It is said to be the *joint* entropy as it involves multiple quantities, and is defined as follows: for two discrete random variables, $X \in \mathcal{X}$ and $Y \in \mathcal{Y}$,

$$H(X,Y) := \sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} p(x,y)\log\frac{1}{p(x,y)}.$$

We see that the only distinction w.r.t. the single random variable case is that the *joint* distribution $p(x,y)$ comes into picture. Similarly, an alternative expression is:

$$H(X,Y) = \mathbb{E}\left[\log\frac{1}{p(X,Y)}\right]$$

where the expectation is now taken over $p(x,y)$.

Chain rule

[2]Jensen's inequality is one of the most well-known inequalities widely used in mathematics, as well as the one that underlies many of the basic results in information theory.

[3]We say that a function f is *concave* if for any (x_1,x_2) and $\lambda\in[0,1]$, $\lambda f(x_1)+(1-\lambda)f(x_2) \leq f(\lambda x_1+(1-\lambda)x_2)$. In other words, for a concave function, the weighted sum w.r.t. functions evaluated at two points is less than or equal to the function at the weighted sum of the two points. See Fig. **??**.

There is a key property regarding the joint entropy, called the *chain rule*, which shows the relationship between the multiple random variables. For the two random variable case, it reads:

$$\textit{Property 3 (chain rule):}\ H(X,Y) = H(X) + H(Y|X)$$

where $H(Y|X)$ is so called *conditional entropy* and defined as:

$$H(Y|X) := \sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} p(x,y)\log\frac{1}{p(y|x)} = \mathbb{E}\left[\frac{1}{p(Y|X)}\right]$$

where the expectation is taken over $p(x,y)$. The proof of the chain rule is straightforward:

$$\begin{aligned}
H(X,Y) &= \mathbb{E}\left[\log\frac{1}{p(X,Y)}\right] \\
&\overset{(a)}{=} \mathbb{E}\left[\log\frac{1}{p(X)p(Y|X)}\right] \\
&\overset{(b)}{=} \mathbb{E}_{X,Y}\left[\log\frac{1}{p(X)}\right] + \mathbb{E}\left[\log\frac{1}{p(Y|X)}\right] \\
&\overset{(c)}{=} \mathbb{E}_{X}\left[\log\frac{1}{p(X)}\right] + \mathbb{E}\left[\log\frac{1}{p(Y|X)}\right] \\
&= H(X) + H(Y|X)
\end{aligned}$$

where (a) follows from the definition of the conditional distribution (see Ch. 1 & 2 in BT); (b) follows from linearity of expectation; (c) follows from $\sum_{y\in\mathcal{Y}} p(x,y) = p(x)$ (why?). The last step is due to the definition of entropy and conditional entropy.

Here we provide an interesting interpretation on the chain rule. Remember that entropy is a measure of uncertainty. So we can interpret *Property 3* as follows. The uncertainty of (X,Y) (reflected in $H(X,Y)$) is the sum of the following two: (i) the uncertainty of X (reflected in $H(X)$); (ii) the uncertainty that remains in Y given X (reflected in $H(Y|X)$). See Fig. **??** for visual illustration. By mapping the *area* of a Venn diagram to the amount of uncertainty w.r.t. an associated random variable, we see that the interpretation makes sense. Here the area of the blue circle indicates $H(X)$; the area of the red part denotes $H(Y|X)$; and the entire area indicates $H(X,Y)$.

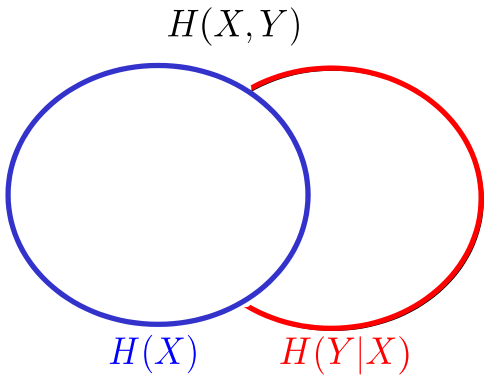


Figure 3: A Venn diagram interpretation of the chain rule.

Lastly let me leave one remark on conditional entropy. Another way of expressing conditional

entropy (which I strongly recommend you to remember) is:

$$\begin{aligned} H(Y|X) &= \sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} p(x,y)\log\frac{1}{p(y|x)} \\ &= \sum_{x\in\mathcal{X}} p(x)\sum_{y\in\mathcal{Y}} p(y|x)\log\frac{1}{p(y|x)} \\ &= \sum_{x\in\mathcal{X}} p(x)H(Y|X=x) \end{aligned}$$

where the last equality is due to the conventional definition of:

$$H(Y|X=x) := \sum_{y\in\mathcal{Y}} p(y|x)\log\frac{1}{p(y|x)}.$$

Here $H(Y|X=x)$ is the entropy defined w.r.t. Y when $X=x$. So the conditional entropy can be interpreted as the weight sum of $H(Y|X=x)$, which sort of helps you to remember the formula as well as to provide an easy way to calculate.

Look ahead

We are now ready to prove the source coding theorem, but not for channel coding theorem. As I mentioned earlier, the proof of the channel coding theorem requires the concept on another notion: *mutual information*. So next time we will study details on mutual information, ranging from its definition to several key properties which serve to prove the theorem as well as turn out to play crucial roles in other disciplines. We will also study about another key notion: the KL divergence. Actually it is quite instrumental to proving the source coding theorem. Moreover, it has played a significant role in other fields, in particular statistics, as it can serve as a quantity that measures a sort of distance between distributions, which is of strong interest to the field.

September 11, 2019
Changho Suh (chsuh@kaist.ac.kr)

Lecture 4: Mutual information & KL divergence

Recap

Last time we learned about one key notion, *entropy*, which serves to play a central role in establishing the source coding theorem that we will investigate in depth in Part II. Specifically we started with the definition of entropy for the single random variable case and then extended to a general case in which more random variables are involved. We then studied one important rule which dictates the relationship across multiple random variables: the *chain rule*. For two random variables, say X and Y, the chain rule says:

$$H(X,Y) = H(X) + H(Y|X) \tag{1}$$

where $H(Y|X)$ denotes conditional entropy. Remember the definition of conditional entropy:

$$H(Y|X) := \sum_{x \in \mathcal{X}} H(Y|X=x) \tag{2}$$

where $H(Y|X=x)$ is the entropy w.r.t. $p(y|x)$.

At the end of last time, I mentioned that the proof of the channel coding theorem which we will also investigate in Part II requires knowledge on another key notion: *mutual information*. I also mentioned that there is another important notion worth being studied in depth: the *Kullabck-Leibler (KL) divergence*.

Today's lecture

So today we will study details on such two notions: mutual information and the KL divergence. Specifically what we are going to do are four-folded. First we will study the definition of mutual information. We will then investigate some key properties of mutual information which turn out to play crucial roles in establishing the channel coding theorem as well as addressing important issues that arise in other disciplines. Next we will learn about the KL divergence and an interesting relationship with mutual information. Lastly we will discuss a connection of mutual information to the channel coding theorem.

Observation

An interesting observation we made w.r.t. the chain rule (the Venn diagram interpretation in Fig. 1) turns out to yield a natural definition for mutual information. The interpretation was: the randomness of two random variables X and Y (reflected in the total area of two Venn diagrams) is the sum of the randomness of one variable, say X, (reflected in the area of the blue Venn diagram) and the uncertainty that remains about Y conditioned on X (reflected in the crescent-shaped red area). By the chain rule, the crescent-shaped red area can be represented as: $H(Y|X)$.

Here we see an overlap between the blue and red areas. The area of the overlapped part depends how large $H(Y|X)$ is: the larger the overlapped area, the smaller $H(Y|X)$. A small $H(Y|X)$ somehow indicates a strong dependency between X and Y. Hence, one can say that: the larger the overlapped area, the larger dependency between the two. In view of this, the overlapped

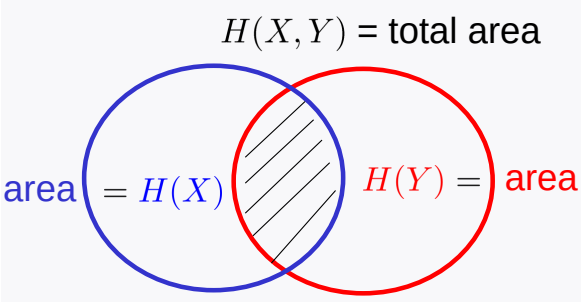


Figure 1: A Venn diagram interpretation of the chain rule.

area captures the amount of information that X and Y share in common, sort of common information.

Definition of mutual information

This leads to a natural definition for the overlapped area so as to capture such common information:

$$I(X;Y) := H(Y) - H(Y|X). \tag{3}$$

In the literature, this notion is called *mutual information* instead of common information.

Obviously from the picture in Fig. 1, one can define it as $I(X;Y) := H(X) - H(X|Y)$ because the alternative indicates the same overlapped area. But by convention we follow the definition of (3): The entropy of the right-hand-side term inside $I(\cdot;\cdot)$ minus the conditional entropy of the right-hand-side term conditioned on the left-hand-side term.

Key properties

As entropy has important properties (described with the non-negativity bound $H(X) \geq 0$ and the cardinality bound $H(X) \leq \log|\mathcal{X}|$), mutual information has similar key properties:

$$\begin{aligned}
&\textit{Property 1:} \quad I(X;Y) = I(Y;X);\\
&\textit{Property 2:} \quad I(X;Y) \geq 0;\\
&\textit{Property 3:} \quad I(X;Y) = 0 \iff X \perp\!\!\!\perp Y.
\end{aligned}$$

The first property (named the *symmetry property*) is obvious from the picture. The proof is also straightforward:

$$\begin{aligned}
I(X;Y) &:= H(Y) - H(Y|X)\\
&\overset{(a)}{=} H(Y) - (H(X,Y) - H(X))\\
&\overset{(b)}{=} H(Y) + H(X) - (H(Y) + H(X|Y))\\
&\overset{(c)}{=} I(Y;X)
\end{aligned}$$

where (a) and (b) follows from the chain rule (1); and (c) is due to the definition of mutual information (3).

The second property is also very much intuitive, as the mutual information captures the size of the overlapped area and therefore it must be non-negative. But the proof is not that straightforward. It requires a bunch of steps as well as the usage of an important inequality that we learned last time: Jensen's inequality. See the next section for a detailed proof.

The third property also makes sense. Why? The mutual information being 0 means no correlation between X and Y, implying independence between the two. But the proof is not quite trivial either. See a section followed by the next section for a proof.

Proof of $I(X;Y) \geq 0$ & its implication

Starting with the definition of mutual information, we get:

$$\begin{aligned}
I(X;Y) &:= H(Y) - H(Y|X) \\
&\overset{(a)}{=} \mathbb{E}_Y\left[\log \frac{1}{p(Y)}\right] - \mathbb{E}_{X,Y}\left[\log \frac{1}{p(Y|X)}\right] \\
&\overset{(b)}{=} \mathbb{E}_{X,Y}\left[\log \frac{1}{p(Y)}\right] - \mathbb{E}_{X,Y}\left[\log \frac{1}{p(Y|X)}\right] \\
&\overset{(c)}{=} \mathbb{E}\left[\log \frac{p(Y|X)}{p(Y)}\right] \\
&\overset{(d)}{=} \mathbb{E}\left[-\log \frac{p(Y)}{p(Y|X)}\right] \\
&\overset{(e)}{\geq} -\log \mathbb{E}\left[\frac{p(Y)}{p(Y|X)}\right] \\
&= -\log\left\{\sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} p(x,y)\frac{p(y)}{p(y|x)}\right\} \\
&\overset{(f)}{=} -\log\left\{\sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} p(x)p(y)\right\} \\
&\overset{(g)}{=} -\log 1 = 0
\end{aligned}$$

where (a) follows from the definition of entropy and joint entropy; (b) is due to the total probability law $\sum_{x\in\mathcal{X}} p(x,y) = p(y)$ (why?); (c) is due to the linearity of the expectation operator; (d) comes from $\log x = -\log \frac{1}{x}$; (e) is due to the fact that $-\log(\cdot)$ is a *convex* function and Jensen's inequality; (f) follows from the definition of conditional distribution $p(y|x) := \frac{p(x,y)}{p(x)}$; and (g) is due to an axiom of the probability distribution: $\sum_{x\in\mathcal{X}} p(x) = \sum_{y\in\mathcal{Y}} p(y) = 1$.

This non-negativity property has another very intuitive implication. Applying the definition of mutual information and then re-arranging the two terms $H(Y)$ and $H(Y|X)$ properly, we get:

$$H(Y) \geq H(Y|X) \tag{4}$$

Remember one interpretation of entropy: a measure of uncertainty. So $H(Y)$ can be viewed as the uncertainty of Y, while $H(Y|X)$ being interpreted as the remaining uncertainty of Y after X being revealed. Our intuition says: Given side information like X that is given as conditioning, we know more and more about Y (the uncertainty is removed further) and therefore, such conditional entropy must be reduced. In short, *conditioning reduces entropy*. The above property proves this intuition.

Some curious students may want to ask: What if X is realized as a certain value $X = x$? In such case, does the particular form of conditioning still reduces entropy:

$$H(Y) \geq H(Y|X = x)? \tag{5}$$

Please think about it while solving the last problem in PS1.

Proof of $I(X;Y) = 0 \iff X \perp\!\!\!\perp Y$

To prove this, first recall one procedure that we had while proving the second property above:

$$\begin{aligned} I(X;Y) &:= H(Y) - H(Y|X) \\ &= \mathbb{E}\left[-\log \frac{p(Y)}{p(Y|X)}\right] \\ &\geq -\log \mathbb{E}\left[\frac{p(Y)}{p(Y|X)}\right]. \end{aligned}$$

Remember that the last inequality is due to *Jensen's inequality*. As you will figure out while solving Problem 3 in PS1, the sufficient and necessary condition for the equality to hold in the above is:

$$\frac{p(Y)}{p(Y|X)} = c \text{ (constant)}.$$

Please check this in PS1. The condition then implies that

$$p(y) = cp(y|x) \;\; \forall x \in \mathcal{X}, y \in \mathcal{Y}.$$

Using the axiom of probability distribution (the sum of the probabilities being 1), we get $c = 1$ and therefore:

$$p(y) = p(y|x) \;\; \forall x \in \mathcal{X}, y \in \mathcal{Y}.$$

Due to the definition of *independence* between two random variables, the above implies that X and Y are independent. Hence, this completes the proof:

$$I(X;Y) = 0 \iff X \perp\!\!\!\perp Y.$$

Interpretation on $I(X;Y)$

Let me say a few words about $I(X;Y)$. Using the chain rule and the definitions of entropy and joint entropy, one can rewrite $I(X;Y) := H(Y) - H(Y|X)$ as

$$\begin{aligned} I(X;Y) &= H(Y) + H(X) - H(X,Y) \\ &= \mathbb{E}\left[\log \frac{1}{p(Y)}\right] + \mathbb{E}\left[\log \frac{1}{p(X)}\right] - \mathbb{E}\left[\log \frac{1}{p(X,Y)}\right] \\ &= \mathbb{E}\left[\log \frac{p(X,Y)}{p(X)p(Y)}\right]. \end{aligned} \tag{6}$$

This leads to the following interesting observation:

$$\begin{aligned} p(X,Y) \text{ close to } p(X)p(Y) &\implies I(X;Y) \approx 0; \\ p(X,Y) \text{ far from } p(X)p(Y) &\implies I(X;Y) \text{ far above } 0. \end{aligned}$$

This naturally enables us to interpret mutual information as a sort of *distance measure* that captures how far the joint distribution $p(X,Y)$ is from the product distribution $p(X)p(Y)$. In statistics, there is a very well-known divergence measure that reflects soft of distance between two distributions: the *KL divergence*. So it turns out mutual information can be represented as the KL divergence. Before detailing such representation, let us first investigate the definition of the KL divergence[1].

Definition of the KL divergence

[1]The book of CT employs a different naming for the KL divergence: *relative entropy*. This naming is popular (but only) in the Information Theory Society. It is not the case in other societies though; the naming of the KL divergence is much more prevalent. Hence, I recommend you to use the naming of KL divergence.

Let $Z \in \mathcal{Z}$ be a discrete random variable. Consider two probability distributions w.r.t. Z: $p(z)$ and $q(z)$ where $z \in \mathcal{Z}$. The KL divergence between the two distributions are defined as:

$$\begin{aligned} \mathsf{KL}\left(p \| q\right) &:= \sum_{z \in \mathcal{Z}} p(z) \log \frac{p(z)}{q(z)} \\ &= \mathbb{E}_{p(Z)} \left[\log \frac{p(Z)}{q(Z)} \right]. \end{aligned} \tag{7}$$

Mutual information in terms of the KL divergence

Applying the definition (7) to (6), we then get:

$$\begin{aligned} I(X;Y) &= \mathbb{E}\left[\log \frac{p(X,Y)}{p(X)p(Y)} \right] \\ &= \mathbb{E}_{p(X,Y)} \left[\log \frac{p(X,Y)}{p(X)p(Y)} \right] \\ &\overset{(a)}{=} \mathbb{E}_{p(Z)} \left[\log \frac{p(Z)}{q(Z)} \right] \\ &\overset{(b)}{=} \mathsf{KL}\left(p(Z) \| q(Z)\right) \\ &= \mathsf{KL}\left(p(X,Y) \| p(X)p(Y)\right) \end{aligned}$$

where (a) comes from my own definition: $Z := (X, Y)$ (note that $p(x)p(y)$ is a valid probability distribution. Why?); and (b) is because of the definition of the KL divergence.

Properties of the KL divergence

As mutual information has the three properties, the KL divergence has three similar properties:

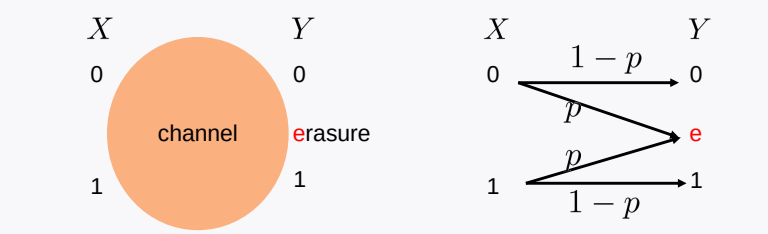
$$\begin{aligned} &\textit{Property 1:} \quad \mathsf{KL}\left(p||q\right) \neq \mathsf{KL}\left(q||p\right); \\ &\textit{Property 2:} \quad \mathsf{KL}\left(p||q\right) \geq 0; \\ &\textit{Property 3:} \quad \mathsf{KL}\left(p||q\right) = 0 \iff p = q. \end{aligned}$$

The first property is similar to the one for mutual information but in a completely opposite manner. Unlike mutual information, the KL divergence is *not symmetric*. Notice in the definition (7) of the KL divergence that the expectation is taken only over the first probability distribution p. This breaks up symmetry. The second and third properties are very much similar to the ones for mutual information. It turns out the proofs are also similar. Please check this in PS2.

Connection of mutual information to channel capacity C

Now let us discuss a connection of mutual information to the channel coding theorem. As I claim earlier, there is a very strong connection between the two.

For this, let us consider one very concrete exemplary channel, named the *binary erasure channel* (BEC for short). Actually this is the first toy-example channel which Shannon thought of. Here is how it looks like. See Fig. 2. The input to the channel, say X, is binary, taking either 0 or 1. The output, say Y, is ternary, taking either 0, 1 or some garbage information, called *erasure* (simply denoted by e). As I mentioned in the first lecture, the channel is sort of an enemy that introduces an uncertainty in the form of noise. In mathematics, the behavior of such uncertainty in the channel can be described by conditional distribution $p(y|x)$. In the BEC, the output is



(a) binary erasure channel (BEC) (b) erasure probability p

Figure 2: Binary Erasure Channel.

the same as the input w.p. $1-p$; otherwise, it is an erasure no matter what x is. In terms of conditional distribution, it is explained as:

$$p(y|x) = \begin{cases} 1-p, & \text{for } (x,y) = (0,0); \\ p, & \text{for } (x,y) = (0,\mathsf{e}); \\ p, & \text{for } (x,y) = (1,\mathsf{e}); \\ 1-p, & \text{for } (x,y) = (1,1); \\ 0, & \text{otherwise.} \end{cases} \tag{8}$$

A pictorial description of the BEC is in Fig. 2(b). Here a value placed above each arrow indicates the transition probability for a transition reflected by the arrow.

To see the connection, let us compute mutual information between the input and the output.

$$I(X;Y) = H(Y) - H(Y|X).$$

As you can see, it requires a computation of the entropy $H(Y)$ of a ternary random variable Y. It turns out that $H(Y)$ is a bit complicated to compute, while a simpler calculation comes from an alternative expression:

$$I(X;Y) = H(X) - H(X|Y). \tag{9}$$

Now to compute $H(X)$, we need to know about $p(x)$. However, $p(x)$ is not given here. So let us make a simple assumption: X is uniformly distributed, i.e., (in other words), $X \sim \mathsf{Bern}(\frac{1}{2})$. Here "$\sim$" refers to "is distributed according to"; Bern denotes the distribution of a binary (or Bernoulli[2]) random variable; the value inside $\mathsf{Bern}(\cdot)$ indicates the probability that the variable takes 1, simply called the Bernoulli parameter. Assuming $X \sim \mathsf{Bern}(\frac{1}{2})$, the entropy of X is simply $H(X) = 1$ and the conditional entropy $H(X|Y)$ is calculated as:

$$\begin{aligned} H(X|Y) &\overset{(a)}{=} \Pr(Y = \mathsf{e})H(X|Y = \mathsf{e}) + \Pr(Y \neq \mathsf{e})H(X|Y \neq \mathsf{e}) \\ &\overset{(b)}{=} \Pr(Y = \mathsf{e})H(X|Y = \mathsf{e}) \\ &\overset{(c)}{=} p \end{aligned}$$

where (a) is due to the definition of conditional entropy; (b) follows from the fact that $Y \neq \mathsf{e}$ completely determines X (no randomness) and therefore $H(X|Y \neq \mathsf{e}) = 0$; (c) follows from the

[2]The Bernoulli random variable is named after a Swiss mathematician in the 1600s: Jacob Bernoulli. He employs such a simple binary random variable to discover one of the foundational laws in mathematics, called the *Law of Large Numbers* (LLN). This is the reason that such a binary random variable is also called the Bernoulli random variable. Later we will have a chance to investigate the LLN. Please be patient until we get to the point, unless you are familiar with the LLN.

fact that $Y = \mathsf{e}$ does not provide any information about X and hence $X|Y = \mathsf{e}$ has exactly the same distribution as X, so $H(X|Y = \mathsf{e}) = H(X) = 1$. Applying this to (9), we get:

$$I(X;Y) = H(X) - H(X|Y) = 1 - p.$$

Now this is exactly where we can see the connection between mutual information and capacity C: the maximum number of bits that can be transmitted over a channel. It turns out that $I(X;Y) = 1 - p$ is the capacity of the BEC. Remember that we assume the distribution of X in computing $I(X;Y)$. It turns out that for a general channel which can be characterized by an arbitrary $p(y|x)$, such $p(x)$ can play a role as an optimization variable and the channel capacity is characterized as:

$$C = \max_{p(x)} I(X;Y). \tag{10}$$

This is the very statement of the channel coding theorem. From this, we see that mutual information is quite instrumental to describing the theorem. Later in Part II, we will prove the theorem.

Look ahead

Next time, we will embark on Part II. And we will start proving the source coding theorem.

September 16, 2019
Changho Suh (chsuh@kaist.ac.kr)

Lecture 5: Source coding theorem for i.i.d. sources (1/3)

Recap

In the first two lectures, we studied Shannon's two-staged architecture in which the encoder is split into two parts: (i) source encoder; (ii) channel encoder. The reason that he proposed such architecture is that he wished to convert an information source of a possibly different type into a common currency of information: bits. He then established two theorems which deal with efficiency of the two blocks and therefore characterize the limits on the amount of information that one can transmit over a channel: (i) source coding theorem; (ii) channel coding theorem. We are now ready to embark on Part II, wherein the goal is to prove the two theorems. In the next five lectures including today's one, we are going to prove the source coding theorem.

Today's lecture

The source coding theorem quantifies the minimum number of bits required to represent an information source without any information loss. The information source can be a collection of dots and lines (in the case of Morse code), an English text, speech signals, video signals or image pixels. So it consists of *multiple* components. For example, a text consists of multiple English letters. Speech signals (waveform) contain multiple points, each indicating the magnitude of the signal at a specific time instant. Also it can be viewed as a *random* from a perspective of a receiver who does not know about the signal. This leads us to model the source as a *random process*: a collection of random variables. Let $\{X_i\}$ be such process. Here X_i is called a "symbol" in the source coding context. For simplicity, we will start with a simple case in which X_i's are i.i.d. (independent and identically distributed). Let X be a generic random variable which represents every individual instance X_i. In the i.i.d. case, the source coding theorem says:

Minimum number of bits required to represent the i.i.d. source $\{X_i\}$ per symbol is $H(X)$.

Today we will attempt to prove this theorem. Once we are done with the i.i.d. case, we will extend to a general case in which the source respects a realistic non-i.i.d. distribution.

Symbol-by-symbol source encoder

Since an information source consists of multiple components, an input to source encoder comprises multiple symbols. So the encoder acts on multiple symbols in general. To understand what it means, let us think about a concrete example where source encoder acts on three consecutive symbols. Here what it means by acting on multiple symbols is that an output is *a function of the three consecutive symbols*. But for simplicity, we are going to consider a much simpler case for the time being in which the encoder *acts on each individual symbol, being independent of other symbols*. It means that the encoder produces bits in a *symbol-by-symbol basis*: a symbol X_1 yields a corresponding binary string, and independently another binary string w.r.t. the next symbol X_2 follows, and this goes on similarly for other follow-up symbols. The reason that we consider this simple yet restrictive setting is that this case turns out to provide significant insights into a general case. Actually it contains every key insight needed for generalization. So building upon the insights that we will obtain from this simple case, we will attack the general case later on. Please be patient until we get to that point.

The simple case allows us to greatly simplify notations. First it suffices to focus only on one symbol, say X. The encoder is nothing but a function of X. Let's call that function C. Please don't be confused with the same notation that we used to indicate channel capacity. The reason that we employ the same notation is that the output $C(X)$ is called "codeword". Let $\ell(X)$ be the length of codeword $C(X)$. For example, consider $X \in \{a, b, c, d\}$ in which $C(a) = 0, C(b) = 10, C(c) = 110, C(d) = 111$. In this case, $\ell(a) = 1, \ell(b) = 2, \ell(c) = 3, \ell(d) = 4$. Note that $\ell(X)$ is a function of a random variable X, hence it is also a random variable. So we are interested in a representative quantity of such varying quantity, which is the *expected* codeword length:

$$\mathbb{E}\left[\ell(X)\right] = \sum_{x \in \mathcal{X}} p(x)\ell(x).$$

Optimization problem

The efficiency of source encoder is well reflected by such expected codeword length. Hence, we wish to minimize the expected codeword length. So the optimization problem of our interest is:

$$\min_{\ell(x)} \sum_{x \in \mathcal{X}} p(x)\ell(x). \tag{1}$$

There are many practical ways to estimate the distribution $p(x)$ of an information source. So let us assume that $p(x)$ is given. In this case, $\ell(x)$'s are only variables that we can optimize over.

Now are we done with the problem formulation? Of course, not! We should definitely worry about constraints that the optimization variables $\ell(x)$'s should satisfy. What are constraints on $\ell(x)$ then? The obvious constraints are: $\ell(x) \geq 1$ and $\ell(x) \in \mathbb{N}$. Are we now done? No! If it is done, the solution to this problem becomes trivial. It would be 1. One can set $\ell(x) = 1$ for all x's to obtain 1. But this is too good to be true. In fact, there is another constraint on $\ell(x)$, which is concerning the condition that a *valid* code should satisfy.

A naive condition: Non-singularity

For the validity of a code, the encoder function must be *one-to-one*. Why? Remember what we discussed in Lecture 2. The reason is that otherwise, there is no way to reconstruct the input from the output. In the source coding context, a code is said to be *non-singular* if it is one-to-one mapping. Mathematically, the non-singularity condition reads:

$$C(x) = C(x') \implies x = x'.$$

Here is an example that respects this condition:

$$C(a) = 0;\ C(b) = 010;\ C(c) = 01;\ C(d) = 10. \tag{2}$$

Note that every codeword is distinct, ensuring one-to-one mapping.

Now is this non-singularity condition enough to ensure the validity of a code? Unfortunately, no. Actually what we care about is a *sequence* of multiple symbols. What we get in the output is the sequence of binary strings which corresponds to a *concatenation* of such multiple symbols: $X_1X_2\cdots X_n \Longrightarrow C(X_1)C(X_2)\cdots C(X_n)$. Remember we assume the *symbol-by-symbol* encoder; hence we get $C(X_1)C(X_2)\cdots C(X_n)$ instead of $C(X_1X_2\cdots X_n)$. So one should be able to reconstruct the sequence $X_1X_2\cdots X_n$ of input symbols from that output $C(X_1)C(X_2)\cdots C(X_n)$. But it turns out that in the above example (2), there is some ambiguity in decoding the sequence

of input symbols. Why? Here is a concrete example where one can see this. Suppose that the output sequence reads:

$$\text{output sequence: } 010$$

Then, what are the corresponding input sequence? One possible input would be simply "b" ($C(b) = 010$). But there are also some other patterns that yield the same output: "ca" ($C(c)C(a) = 010$) and "ad" ($C(a)C(d) = 010$). We have multiple candidates that agree upon the same output. This is problematic because we cannot tell which input sequence is put into. In other words, we cannot *uniquely* figure out what the input is.

A stronger condition: Unique decodability

What additional condition do we need to satisfy in order to make a code valid? What we need is that for any encoded bit sequence, there must be no decoding ambiguity, in other words, there must be only one matching input sequence. This property is called *unique decodability*. This is equivalent to the one-to-one mapping constraint now w.r.t. the *sequence* of source symbols with an arbitrary length. Here is a mathematical expression for unique decodability: for any n and m,

$$C(x_1)C(x_2)\cdots C(x_n) = C(x_1')C(x_2')\cdots C(x_m') \implies x_1x_2\cdots x_n = x_1'x_2'\cdots x_m'.$$

Example

Let me give you an example where the unique decodability condition holds:

$$C(a) = 10;\ C(b) = 00;\ C(c) = 11;\ C(d) = 110. \tag{3}$$

One may wonder how to check unique decodability. Here is how. Suppose the output sequence reads:

$$\text{output sequence: } 10110101111\cdots$$

First we read a binary string until we find a *matching codeword* or a *codeword which includes the string in part*. In this example, the first read must be 10 because there is only one corresponding codeword: $C(a)$. So, the corresponding input is "a". What about the next read? Here an ambiguity arises in the next two bits: 11. We have two possible candidates: (i) a matching codeword $C(c) = 11$; (ii) another codeword $C(d) = 110$ which includes the string 11 in part. Here the "11" is either from "c" or from "d". It looks this code is not uniquley decodable. But it is actually uniquely decodable - we can tell which one is the correct one. How? Via looking at the future string! What does this mean? Suppose we see one more bit after "11", i.e., we read 110. Still no way to identify which is correct one! However, suppose we see two more bits after "11", i.e., we read 1101. We can then tell which symbol is actually put into. That is, "d"! Why? Another possibility "cb" ($C(c)C(b) = 1100$) does not agree upon the 1101. So it is eliminated. We repeat this. If one can uniquely decode the input sequence with this way, then the code is said to be uniquely decodable. Actually one can readily verify that the above mapping (3) ensures unique decodability[1].

Constraints on $\ell(x)$ due to uniquely decodable property?

[1]There is a rigorous way of checking unique decodability, proposed by Sardinas and Patternson. Please check Problem 5.27 in CT for details.

Recall our goal: finding constraints on $\ell(x)$ in the optimization problem (1). So we now need to worry about what is an appropriate mathematical constraint on $\ell(x)$'s that respects the property of unique decodability. How to translate the unique decodability property into a mathematical constraint in terms of $\ell(x)$'s? It turns out the translation is a bit difficult.

But there is a good news. The good news is that there is an indirect and much easier way to identify the constraint. A subclass of uniquely decodable codes, called *prefix-free codes* comes to rescue! The prefix-free code that we will describe in greater details soon satisfies the following two properties: (i) it yields exactly the same constraint as the constraint that the uniquely decodable code should satisfy (meaning that if a code is uniquely decodable, then it must respect the constraint due to prefix codes as well); (ii) it offers a much easier way to identify the constraint that the valid code should satisfy.

Here the first property means that the constraint due to prefix-free codes is a *sufficient and necessary condition* for a valid uniquely-decodable code. So it suffices to consider the prefix-free code when coming up with the constraint due to the valid code. The proof of the first property is not that simple. So you will be asked to prove it in PS2. But don't worry. There would be many subproblems which will help you to prove it without much difficulty although the proof itself is highly non-trivial.

To understand what the second property means, we should figure out what the prefix-free code is in detail. So I will first explain what the code is and also will leave a side remark on one big advantage against non prefix-free yet uniquely decodable codes.

Prefix-free codes

Recall the previous example of (3) in which the code is uniquely decodable. Actually this is a perfect example which poses some issue w.r.t. decoding complexity, thereby motivating prefix codes. What is that complexity issue? The issue is that as we saw in the prior example, decoding the second input symbol requires looking at a *future* string, so decoding is not instantaneous. Actually this issue can be aggravated further. In the worst case, we can think of the following output sequence:

$$1100000000000000000000000000000000000001.$$

In this case, decoding even the first symbol, we have to take a look at many future strings.

The prefix-free code that I will define soon is the one in which there is no such complexity issue. Here is an example of such code:

$$C(a) = 0;\ C(b) = 10;\ C(c) = 110;\ C(d) = 111. \tag{4}$$

One key property of this code is that *no codeword is a prefix of any other codeword.* This is why the code is named the *prefix-free code.* As you can readily figure out, the code in the previous example (3) is not prefix-free although it is indeed uniquely-decodable: there exists some codeword such that it is a prefix of some other codeword. This was the main reason that decoding such a code requires looking at future strings, in the worst case, we need to take a look at the *entire* string to decode even the first input symbol. On the other hand, the prefix-free code like (4) has no such codeword that is a prefix of any other codeword. This enables us to have no such complexity issue in decoding because there is no ambiguity in decoding and therefore we don't need to look at any future string to decode an input. So the code is instantaneously decodable. That's why such code is also called *instantaneous.*

Look ahead

Recall the optimization problem (1). As I mentioned earlier, the good news is that: (i) the constraint on $\ell(x)$ that the prefix-free code should satisfy is the same as that due to the uniquely decodable code; (ii) it is easy to identify the constraint due to the prefix-free code. So next time, we will figure out the constraint that the prefix-free code property should respect. We will then attack the optimization problem.

September 18, 2019
Changho Suh (chsuh@kaist.ac.kr)

Lecture 6: Source coding theorem for i.i.d. sources (2/3)

Recap

Last time, we tried to prove the source coding theorem for i.i.d. sources. As an initial effort, we focused on a simple symbol-by-symbol encoder setting in which the code acts on each individual symbol independently. We then set out the goal: designing a code C such that $\mathbb{E}[\ell(X)]$ is minimized. To achieve the goal, we formulated an optimization problem:

$$\min_{\ell(x)} \sum_{x\in\mathcal{X}} p(x)\ell(x)$$
$$\text{subject to some constraint on } \ell(x).$$

Here the key to solving the problem is to come up with mathematical constraints on $\ell(x)$ that a valid code (fully specified by the unique-decodability property) should respect.

At the end of the last lecture, I mentioned that it is a bit difficult to come up with such constraints. So we planned to take a different approach, being motivated by the following facts (which I claimed but did not prove): (1) constraints on $\ell(x)$ that prefix-free codes (a subclass of uniquely-decodable codes) satisfy are equivalent to those due to uniquely-decodable codes; (2) deriving mathematical constraints on $\ell(x)$ induced by the prefix-free code property is relatively easier. We deferred the proof of the first property to PS2.

Today's lecture

Today, we are going to derive the constraint due to the prefix-free code property, and will attack the optimization problem accordingly.

Review of prefix-free codes

Let us start by reviewing the prefix-free code example that we introduced last time:

$$C(a) = 0;\ C(b) = 10;\ C(c) = 110;\ C(d) = 111. \tag{1}$$

Notice that no codeword is a prefix of any other codeword. So it is indeed prefix-free.

From codeword to binary code tree

Let me introduce a pictorial representation of the code (1) that will guide us to easily figure out mathematical constraints on $\ell(x)$. Actually the picture that I will introduce is the one that you saw earlier: the *binary code tree*. The binary code tree is the one in which there are only two branches for each node (either the root or an internal node). As mentioned earlier, there is one-to-one correspondence between a code mapping rule and a representation of a binary code tree.

Let me explain how to draw a binary code tree given a code mapping rule. We start with the root and draw two branches that originate from the root. We then assign a label of 0 on the top branch, while assigning a label of 1 on the bottom. We may want to take the other way around: 1 for the top and 0 for the bottom. This is our own choice. We then have two nodes.

Next we assign each node, a sequence of binary labels that are along the path from the root to that node. So 0 is assigned to the top node; 1 is assigned to the bottom. We then check if there is a codeword which matches one of the two sequences 0 and 1. We see that the codeword $C(a)$ matches the sequence 0 assigned to the top node. So we assign "a" to the top node. We repeat this for the bottom. Note that this node has no matching codeword. If there is no matching node but there still exist codewords, then we generate two additional branches originating from the node. We then assign a label of 0 on the top, a label of 1 on the bottom. Again we can take the other way around. We now assign to the top node a sequence of binary labels on the branches connecting the root to the node: "10". On the bottom, we assign "11". Next we check if there is a codeword that is identical either to "10" or to "11". We see that $C(b) = 10$. So we assign "b" to the top node. Note that there is no matching codeword for "11". So we split the bottom into two, assigning "110" on the top and "111' on the bottom. Finally we assign "c" to the top, "d" to the bottom. See Fig. 1 for visual representation.

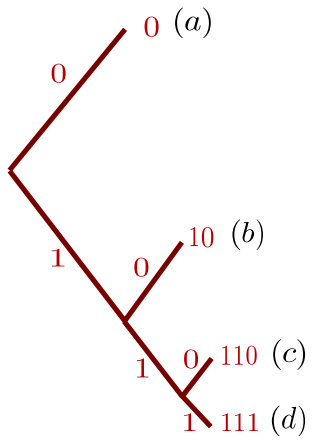


Figure 1: The codeword representation via a binary code tree.

The binary-code-tree representation gives insights into identifying a mathematical constraint on $\ell(x)$ that the prefix-free code should admit. Specifically the following two observations help.

Observation #1

The first observation is regarding the location of nodes to which symbols are assigned. A tree consists of two types of nodes. One is an ending node (terminal) which has no further branch. We call that ending node a *leaf*. The second is a node from which another branch generates. We call it an internal node. Keeping these in our mind, let's take a look at the earlier binary code tree illustrated in Fig. 1. Notice that all codewords are assigned to leaves only. In other words, there is no codeword that is assigned to an internal node. Can you see why that is the case? Actually it is obvious. In a prefix-free code, there is no codeword that is a prefix of any other codeword. If there were a codeword that is assigned to an internal code, then we would violate the prefix-free code property because that codeword is definitely a prefix of some other codeword which lives in the associated leaf. So the first observation that one can make from the prefix-free code is:

Observation #1: Codeword must be a leaf in a binary code tree.

Observation #2

Now let's move on to the second observation that serves to relate the prefix-free code property to the mathematical constraint on $\ell(x)$. That is,

Observation #2: Any codeword can be mapped to a subinterval in $[0, 1]$.

What does this mean? Let me explain what it means by drawing another picture on top of the binary code tree in Fig. 1. Here an interval $[0, 1]$ is associated with the entire set of codewords. See Fig. 2. We then draw a line that is crossing the root, and assign the average value of two ends in the associated interval ($\frac{0+1}{2} = 0.5$) to a point that intersects the line and the $[0, 1]$ interval. Now we assign a codeword to the $[0, 0.5]$ interval. We see that there is only one codeword above the root level. So we map the codeword $C(a)$ to the $[0, 0.5]$ interval. Below the root level, there are multiple codewords though. We draw another line on the centered interior node in the bottom level. We then assign $\frac{0.5+1}{2} = 0.75$ to a point that intersects the line and the $[0.5, 1]$ interval. Here we do the same thing. We assign $C(b)$ to the $[0.5, 0.75]$ interval. We repeat this until we map every codeword. See the resulting picture in Fig. 2. Here we see that every codeword is mapped into a subinterval of $[0, 1]$:

$$C(a) \leftrightarrow [0, 0.5];$$
$$C(b) \leftrightarrow [0.5, 0.75];$$
$$C(c) \leftrightarrow [0.75, 0.875];$$
$$C(d) \leftrightarrow [0.875, 1].$$

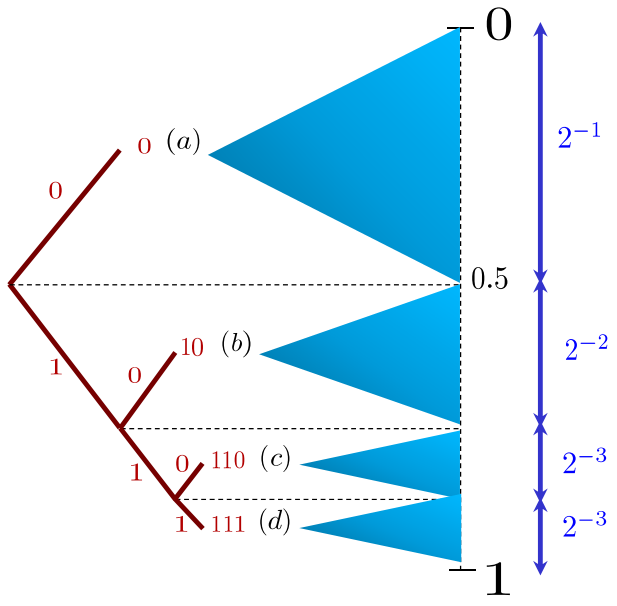


Figure 2: Observation #2: Any codeword can be mapped to a subinterval in $[0, 1]$.

This naturally leads to the following two facts: (1) subinterval size $= 2^{-\ell(x)}$; (2) there is no overlap between subintervals. The second fact comes from the first observation: an interior node cannot be a codeword. This leads to the following constraint:

$$\sum_{x \in \mathcal{X}} 2^{-\ell(x)} \leq 1. \tag{2}$$

This is called *Kraft's inequality.*

Optimization problem

Using Kraft's inequality, we can now formulate the optimization problem as:

$$\min_{\ell(x)} \sum_{x\in\mathcal{X}} p(x)\ell(x)$$
$$\text{subject to } \sum_{x\in\mathcal{X}} 2^{-\ell(x)} \leq 1, \;\; \ell(x) \in \mathbb{N}, \;\; \ell(x) \geq 1.$$

Here one can ignore the constraint $\ell(x) \geq 1$? Why? Otherwise, the Kraft's inequality $\sum_{x\in\mathcal{X}} 2^{-\ell(x)} \leq 1$ is violated. So the simplified problem reads:

$$\begin{aligned} &\min_{\ell(x)} \sum_{x\in\mathcal{X}} p(x)\ell(x) \\ &\text{subject to } \sum_{x\in\mathcal{X}} 2^{-\ell(x)} \leq 1, \quad \ell(x) \in \mathbb{N}. \end{aligned} \tag{3}$$

Non-convex optimization problem

In fact, this optimization problem is very hard to solve. It is categorized into a certain type of problems which are known to be very difficult in general. To see this, let us start by remembering one definition that we learned in Lecture 3: *concave* functions. We say that a function is concave if $\forall x_1, x_2$ and $\lambda \in [0,1]$, $\lambda f(x_1) + (1-\lambda) f(x_2) \leq f(\lambda x_1 + (1-\lambda)x_2)$. Actually there is another type of functions which are defined in a very similar yet opposite manner: *convex* functions. We say that a function f is convex if $-f$ is concave, i.e., $\forall x_1, x_2$ and $\lambda \in [0,1]$, $\lambda f(x_1) + (1-\lambda) f(x_2) \geq f(\lambda x_1 + (1-\lambda)x_2)$. Note that the inequality has an opposite direction compared to the one used in defining concave functions.

Consider the optimization problem (3) now with the concept of convex functions. Here the objective function is *convex* in $\ell(x)$, and also the left-hand-side of the inequality constraint (when it is massaged such that the right-hand-side is 0) is $\sum_{x\in\mathcal{X}} 2^{-\ell(x)} - 1$, which is convex in $\ell(x)$. We say that the optimization problem is *convex* if the objective function and the left-hand-sides in constraints are convex in variables that we optimize over. The problem is said to be *non-convex* otherwise, in other words, if it includes any non-convex objective function and/or non-convex function in the inequalities.

We see in (3) that the objective function and the function in the inequality constraint are convex. What about the integer constraint $\ell(x) \in \mathbb{N}$? Now we need to worry about the definition of convexity w.r.t. a set. Suppose we have two points, say A and B in a set. We say that the set is *convex* if any linear combination of the two points A and B is also an element of the set; otherwise, *non-convex*. Now is $\mathbb{N}$ convex or not? It is non-convex! Why? Suppose we have two integer points, say 1 and 2. Now consider one linear combination of 1 and 2, say 1.5. Obviously 1.5 is not an integer.

The integer constraint $\ell(x) \in \mathbb{N}$ is *non-convex*. So the optimization problem is non-convex. Especially when the non-convex constraint is an integer constraint, the problem is known to be notoriously difficult. It turns out the optimization problem of our interest is indeed extremely hard to solve. So it has been open so far.

Approximate!

What did Shannon do in order to overcome this challenge? Here is what he did. Since the problem is extremely hard, he just wanted to say something about the solution. He believed that although the problem is hard, it may be doable to come up with something which is similar to the exact solution. To this end, what he attempted to do is to *approximate* the solution. In

other words, he intended to come with upper and lower bounds (on the solution) which are close enough.

A lower bound

First consider a lower bound. Notice that the optimization problem of our interest aims to *minimize* the object function. So one can easily expect that with a larger (more relaxed) search space for $\ell(x)$, the solution would be smaller than or equal to the exact solution in the original problem! This motivated Shannon to broaden the search space as it yields a lower bound. Which constraint did Shannon want to remove, in an effort to broaden the search space then? Shannon was smart. Of course, he has chosen the *integer constraint* which contains a non-convex set and hence makes the problem quite challenging. Here is the relaxed version of the problem that Shannon formulated:

$$\begin{aligned} \underline{\mathcal{L}} := \min_{\ell(x)} & \sum_{x\in\mathcal{X}} p(x)\ell(x) \\ \text{subject to } & \sum_{x\in\mathcal{X}} 2^{-\ell(x)} \leq 1. \end{aligned} \tag{4}$$

Look ahead

So far we have formulated the optimization problem which aims at minimizing the expected codeword length subject to Kraft's inequality and an integer constraint on $\ell(x)$. Relaxing the integer constraint, we translated the non-convex optimization into a tractable convex optimization problem. Next time we will derive a lower bound by solving such convex optimization. We will then derive an upper bound, and based on the lower and upper bounds, we will attempt to prove the source coding theorem.

September 23, 2019
Changho Suh (chsuh@kaist.ac.kr)

Lecture 7: Source coding theorem for i.i.d. sources (3/3)

Recap

Last time we formulated an optimization problem which aims at minimizing the expected codeword length subject to Kraft' inequality (that prefix-free codes imply) and the integer constraint on $\ell(x)$ (an intrinsic constraint due to the nature of the problem). We then realized that it is a *non-convex* optimization problem which is intractable in general. In an effort to make some progress, we employed an approach that Shannon took, which is to approximate: Deriving lower and upper bounds as close as possible. At the end of the last lecture, we introduced a trick which allows us to come up with a lower bound. The trick is to relax constraints, i.e., broadening search space in an effort to yield a better solution. Specifically we removed the integer constraint which was the cause for non-convexity, and hence could translate the problem into a tractable convex optimization problem:

$$\begin{aligned} \underline{\mathcal{L}} := \min_{\ell} & \sum_{x\in\mathcal{X}} p(x)\ell(x) \\ \text{subject to } & \sum_{x\in\mathcal{X}} 2^{-\ell(x)} \le 1. \end{aligned} \tag{1}$$

Today's lecture

While we derived a lower bound in class, CN6 did not contain details on the derivation of the bound, although LS6 did. So here we will provide such derivation. We will then derive an upper bound introducing another trick. Based on the lower and upper bounds, we will later complete the proof of the source coding theorem.

A lower bound

Actually the convex optimization problem (in this case, the objective function and the functions that appear in constraints are convex) is very well studied. So there are many ways of solving the problems. One of them is the one that you already learned about from a course on Calculus. That is, the *Lagrange multiplier method*. The way it works is very simple. To describe how it works, we first need to define the Lagrange function. The Lagrange function is defined in terms of the optimization variables $\ell(x)$'s and some other variable called *Lagrange multiplier*, say λ:

$$\mathcal{L}(\ell(x),\lambda) = \sum_{x\in\mathcal{X}} p(x)\ell(x) + \lambda\left(\sum_{x\in\mathcal{X}} 2^{-\ell(x)} - 1\right).$$

The number of Lagrange multipliers is the same as the number of constraints involved. So here we have only one. Notice that the Lagrange function is the sum of two terms: (i) the objective function; (ii) Lagrange multiplier times the left-hand-side of the inequality constraint in the canonical form[1].

[1]The canonical form of inequality constraints is the one in which the right-hand-side is 0 and the inequality direction is $\le$.

Now then how does the Lagrange multiplier method work? We take a derivative of the Lagrange function w.r.t. optimization variables $\ell(x)$'s. We also take a derivative w.r.t. the Lagrange multiplier λ. Setting these derivatives to zero, we get:

$$\frac{\mathcal{L}(\ell(x),\lambda)}{d\ell(x)} = 0;$$
$$\frac{\mathcal{L}(\ell(x),\lambda)}{d\lambda} = 0.$$

It turns out solving these under the constraint of $\lambda \geq 0$ leads to the solution. This is how it works. But in this lecture, we are not going to use this method. Why? There are two reasons: (i) this way is a bit complicated and messy; (ii) the method is not quite intuitive as to why it should work[2].

Instead we are going to use an alternative approach which turns out to be much simpler and intuitive. But don't worry - in PS3, you will have a chance to take the Langrange multiplier method to solve the problem. Before describing the method in details, let us first simplify the optimization problem further. Here one thing to notice is that we do not need to consider the case when the strict inequality holds: $\sum_{x\in\mathcal{X}} 2^{-\ell(x)} < 1$. Here is why. Suppose there exists an optimal solution, say $\ell^*(x)$, such that the strict inequality holds: $\sum_{x\in\mathcal{X}} 2^{-\ell^*(x)} < 1$. Then, one can always come up with a better solution, say $\ell'(x)$, such that: for some x_0,

$$\begin{aligned}
&\ell'(x_0) < \ell^*(x_0);\\
&\ell'(x) = \ell^*(x) \ \ \forall x \neq x_0;\\
&\sum_{x\in\mathcal{X}} 2^{-\ell'(x)} = 1.
\end{aligned}$$

Here we reduced $\ell^*(x_0)$ a bit for one particular symbol x_0, in an effort to increase $2^{-\ell^*(x_0)}$ so that we achieve $\sum 2^{-\ell'(x)} = 1$. Note that this is indeed a better solution as it yields a smaller objective solution due to $\ell'(x_0) < \ell^*(x_0)$. This is obviously contradiction! This implies that an optimal solution occurs only when the equality constraint holds. Hence, it suffices to consider the equality constraint. So the optimization can be simplified as:

$$\begin{aligned}
&\underline{\mathcal{L}} := \min_{\ell(x)} \sum_{x\in\mathcal{X}} p(x)\ell(x)\\
&\text{subject to } \sum_{x\in\mathcal{X}} 2^{-\ell(x)} = 1.
\end{aligned} \tag{2}$$

Now the approach that we will take here is based on a method called "change of variables". Let $q(x) = 2^{-\ell(x)}$. Then, the equality constraint becomes $\sum_{x\in\mathcal{X}} q(x) = 1$, and $\ell(x)$ in the objective function should be replaced with $\log \frac{1}{q(x)}$:

$$\begin{aligned}
&\underline{\mathcal{L}} = \min_{q(x)} \sum_{x\in\mathcal{X}} p(x) \log \frac{1}{q(x)}\\
&\text{subject to } \sum_{x\in\mathcal{X}} q(x) = 1, \quad q(x) \geq 0.
\end{aligned} \tag{3}$$

[2]Actually there is a deep underlying theorem, so called the *strong duality theorem*, which proves that such approach leads to the optimal solution under convexity constraints. Since we will not deal with the proof in this course, it is reasonable not to take the approach which relies on such deep theorem. But if you are interested in the theorem, please refer to course notes (see CN12) for convex optimization which I uploaded on the course website. These are the notes which I wrote for the course; so if you don't understand, please consult with me.

Here one thing to notice is that the constraint of $q(x) \geq 0$ is newly introduced as $q(x) = 2^{-\ell(x)}$ - remember that whenever we apply "change of variable", we should worry about intrinsic constraints on the new variables which did not appear in the original optimization problem. Now observe $\sum_{x \in \mathcal{X}} q(x) = 1$. What does this remind you of? Probability mass function! Recall the axiom that the pmf should satisfy (if you don't remember, please refer to Ch 1 and 2 in BT).

Here the objective function looks like the one that you saw earlier. That is, entropy $H(X)$! Now what I claim is that the solution to the optimization problem is $H(X)$ and the minimizer $q^*(x)$ (that minimizes the objective function) is $p(x)$. Here is the proof.

Let's consider subtraction of the objective function from $H(X)$. We then get:

$$\begin{aligned}
&\sum_{x \in \mathcal{X}} p(x) \log \frac{1}{q(x)} - \sum_{x \in \mathcal{X}} p(x) \log \frac{1}{p(x)} \\
&= \sum_{x \in \mathcal{X}} p(x) \log \frac{p(x)}{q(x)} \\
&= \mathbb{E}_p \left[\log \frac{p(X)}{q(X)} \right]
\end{aligned}$$

Now what does this remind you of? That is the Kullback-Leibler (KL) divergence that you encountered in Lecture 4! Applying one key fact regarding the KL divergence that it is non-negative (this is what you will prove in one of the subproblems in PS2), one can immediately see that the objective function is minimized as $\underline{\mathcal{L}} = H(X)$, and the minimizer $q^*(x) = p(x)$.

An upper bound

Now let us move on to an upper-bound issue. Remember the way to come up with a lower bound. That is to broaden the search space. Then, what is a natural corresponding way to come up with an upper bound? The answer is: the other way around! That is to *reduce* the search space. Here we will take one very naive way of reducing the search space. That is to make one particular choice for optimization variables $\ell(x)$'s.

Now the question is: Which particular choice that we want to make for $\ell(x)$? Obviously a random choice for $\ell(x)$ will lead to a possibly very loose upper bound. So we need to be very careful about the choice. One can imagine that a good choice might be similar to the optimal solution in the relaxed optimization problem (without the integer constraint). In this regard, the minimizer $\ell^*(x)$ for the relaxed optimization can shed some lights on this. Remember the minimizer in the case:

$$q^*(x) = p(x)$$

Since $q(x) := 2^{-\ell(x)}$, in terms of $\ell(x)$, it would be:

$$\ell^*(x) = \log \frac{1}{p(x)}.$$

If $\ell^*(x)$'s were integers, then we are more than happy as we can obtain the exact solution to the original optimization problem. In general, however, $\ell^*(x)$'s are not necessarily integers. One natural choice for $\ell(x)$ is then to take an integer which is as close to $\ell^*(x)$ as possible. So one can think of two options: (i) $\lfloor \log \frac{1}{p(x)} \rfloor$; (ii) $\lceil \log \frac{1}{p(x)} \rceil$. Which one do you want to take? Actually the first choice is not valid. Why? Think about the Kraft's inequality. Hence, the natural choice

is the second one. Under the second choice, we then get:

$$\begin{aligned}\mathcal{L}^* &\le \sum_{x\in\mathcal{X}} p(x)\left\lceil \log\frac{1}{p(x)}\right\rceil \\ &\le \sum_{x\in\mathcal{X}} p(x)\left(\log\frac{1}{p(x)}+1\right) \\ &= H(X)+1.\end{aligned}$$

Are the bounds tight?

In summary, what we can say for $\mathcal{L}^*$ is :

$$H(X) \le \mathcal{L}^* \le H(X)+1.$$

First take a look at the lower bound. Note the lower bound is tight when $\log\frac{1}{p(x)}$'s are integers (i.e., $p(x)$'s are integer powers of 2) and therefore $\ell^*(x)$ can be chosen as $\log\frac{1}{p(x)}$ without violating the integer constraint. However, this is a very particular case because in general $p(x)$ is not limited to that particular type. As for the upper bound $H(X)+1$: one thing that we can say is that if $H(X)$ is large enough, the gap of 1 would be negligible. However, if $H(X)$ is comparable to (or much smaller than) 1, then the bounds are quite loose. For instance, consider a case in which X is a binary r.v. with $\Pr(X=0)=p$ where $p \ll 1$. In this case, $H(X)$ is close to 0; hence, the bounds play almost no role in the case.

General source encoder

While we have made significant effort to approximate $\mathcal{L}^*$, we figured out that the bounds are not tight in general. So our effort seems to play no role. While the bounds themselves are useless, the techniques that we used for deriving the bounds turn out to play a crucial role in proving the source coding theorem. Here are details on why.

Actually we have restricted ourselves to a very special setting in source encoding. We focused on a *symbol-by-symbol* encoder which acts on each individual symbol independently. However, the encoder can take an arbitrary length of input sequence to yield an output. So in general it can act on multiple symbols. For example, one can take an n-length sequence of $Z_n := (X_1, X_2, \ldots, X_n)$, say *super-symbol*, and generate an output like $C(X_1, X_2, \ldots, X_n)$. In fact, the sequence length n is sort of design parameters, the one that we can choose as per our wish.

Very interestingly, if one can apply the bounding techniques that we have learned to this general setting, we can readily prove the source coding theorem. Here is detail. Let $\mathcal{L}_n^*$ be the minimum expected codeword length w.r.t. the super-symbol Z_n:

$$\mathcal{L}_n^* = \min_{\ell(z)} \sum_{z\in\mathcal{Z}} p_{Z_n}(z)\ell(z).$$

Applying the same bounding techniques to $\mathcal{L}_n^*$, we now get:

$$H(Z_n) \le \mathcal{L}_n^* \le H(Z_n)+1.$$

Since the codeword length *per symbol* is of our interest, what we do care about is the one divided by n:

$$\frac{H(Z_n)}{n} \le \frac{\mathcal{L}_n^*}{n} \le \frac{H(Z_n)+1}{n}.$$

Remember that the length n of super-symbol is of our design choice and hence we can choose it so that it is an arbitrary integer. In an extreme case, we can make it arbitrarily large. Actually this is exactly the way that we achieve the limit. Note that the lower and upper bounds coincide with $\frac{H(Z_n)}{n}$ as n tends to infinity.

We are now almost done with the proof. What remains is to calculate such matching quantity. Using the chain rule (a generalized version of the chain rule - check in PS2), we get:

$$\begin{aligned}\frac{H(X_1, X_2, \ldots, X_n)}{n} &= \frac{H(X_1) + H(X_2|X_1) + \cdots + H(X_n|X_1, \cdots, X_{n-1})}{n} \\ &\overset{(a)}{=} \frac{H(X_1) + H(X_2) + \cdots + H(X_n)}{n} \\ &\overset{(b)}{=} H(X)\end{aligned}$$

where (a) follows from the independence of $(X_1, X_2, \ldots, X_n)$ and the fact that $H(X_2|X_1) = H(X_2)$ when X_1 and X_2 are independent (check in PS2); and (b) is due to the fact that $(X_1, X_2, \ldots, X_n)$ are identically distributed.

Hence,

$$\lim_{n\to\infty} \frac{\mathcal{L}_n^*}{n} = H(X).$$

This proves the source coding theorem for i.i.d. sources: the minimum number of bits that represents an information source per symbol on average is indeed $H(X)$.

Look ahead

So far we have proved the source coding theorem. We formulated an optimization problem and then showed that the solution to the optimization problem is entropy. What this result implies is that there *exist* optimal codes that can achieve the limit. Actually we never talked about *explicit sequence pattern* of such optimal codes. In other words, we never discussed how to design such codes. Next time, we will discuss this issue in depth.

September 25, 2019
Changho Suh (chsuh@kaist.ac.kr)

Lecture 8: Source code design

Recap

So far we have proved the source coding theorem for the i.i.d. source case. In an effort to gain some insights, we first considered a simple yet restricted setting in which source encoder acts on each individual symbol, i.e., the *symbol-by-symbol* encoder setting. We then formulated an optimization problem which aims at minimizing the expected codeword length under the setting. Realizing the difficulty of such problem, we attempted to approximate the solution via deriving reasonably tight lower and upper bounds. Applying simple yet power lower-&-upper tricks, we could come up with the bounds which differ by 1.

We then applied these bounding techniques to a general setting in which source encoder can act now on multiple symbols, the number of which can possibly be arbitrarily large. We ended up with:

$$\frac{H(X_1,\ldots,X_n)}{n} \leq \frac{\mathcal{L}_n^*}{n} \leq \frac{H(X_1,\ldots,X_n)+1}{n}$$

where $\mathcal{L}_n^*$ indicates the minimum expected codeword length w.r.t. a super symbol $Z_n := (X_1,\ldots,X_n)$. This led us to obtain: In the limit of n,

$$\frac{\mathcal{L}_n^*}{n} \longrightarrow H(X).$$

Today's lecture

Here what this result implies is that there *exist* optimal codes that can achieve the limit. Actually we never talked about *explicit sequence pattern* of such optimal codes. In other words, we never discussed how to design such codes. In this lecture, we will discuss this issue in details.

Regimes in which one can achieve the limit

Recall the setting where we can achieve the limit. That is the one in which we take a super-symbol and the length of the super-symbol tends to infinity. Now how to design such codes? To this end, we need to know about two things: the *length* and *pattern* of codewords. Actually the minimizer of the relaxed optimization problem gives insights into the length of optimal codewords. Recall:

$$\ell^*(X^n) = \log \frac{1}{p(X^n)}.$$

So to figure out what the length looks like in the interested regime of large n, we should take a look at the behavior of $p(X^n)$ for that regime.

Behavior of $p(X^n)$ for large n

To gain insights, we focus on the binary alphabet case in which $X_i \in \{a,b\}$ and $\Pr(X_i = a) = p$. Later we will consider the general alphabet case. Obviously the sequence consists of a certain combination of two symbols: a's and b's. For very large n, there is an interesting behavior on two

quantities regarding the sequence. One is the faction of symbol "a", represented as the number of a's divided by n. This is an empirical mean of occurrences of symbol "a". The second is the fraction of symbol "b", the number of b's divided by n.

To see this clearly, let's compute the probability of observing X^n. Since X^n is i.i.d., $p(X^n)$ is simply the product of individual probabilities:

$$p(X^n) = p(X_1)p(X_2)\cdots p(X_n).$$

Here each probability is p or $1-p$ depending on the value of X_i. So the result would be of the following form:

$$p(X^n) = p^{\{\#\text{ of } a\text{'s}\}}(1-p)^{\{\#\text{ of } b\text{'s}\}}.$$

Now consider $\log \frac{1}{p(X^n)}$ that we are interested in:

$$\log \frac{1}{p(X^n)} = \{\#\text{ of } a\text{'s}\} \cdot \log \frac{1}{p} + \{\#\text{ of } b\text{'s}\} \cdot \log \frac{1}{1-p}.$$

Dividing by n on both sides, we then get:

$$\frac{1}{n} \log \frac{1}{p(X^n)} = \frac{\{\#\text{ of } a\text{'s}\}}{n} \cdot \log \frac{1}{p} + \frac{\{\#\text{ of } b\text{'s}\}}{n} \cdot \log \frac{1}{1-p}. \tag{1}$$

Now what can we say about this in the limit of n? Your intuition says: $\frac{\{\#\text{ of } a\text{'s}\}}{n} \to p$ as $n \to \infty$! One can naturally expect that when $n \to \infty$, the fraction would be very close to $\Pr(X = a)$. Actually this is indeed the case. Relying on a very well-known theorem, so called the *Weak Laws of Large Numbers* (WLLN[1]), one can prove this. Here is the proof. Let $Y_i = \mathbf{1}(X_i = a)$ where $\mathbf{1}\{\cdot\}$ is an indicator function that returns 1 when the event $\cdot$ is true, 0 otherwise. Consider:

$$S_n := \frac{Y_1 + Y_2 + \cdots + Y_n}{n}.$$

The S_n indicates the fraction of a's. Obviously it is a sequence of n. Now consider the *convergence* of the sequence. Remember from the course on Calculus that you learned about the convergence of sequences, which are *deterministic*.

But here the story is a bit different. The reason is that S_n is a *random variable* (not deterministic). So now we need to consider the convergence w.r.t. a *random process*. Actually there are multiple types of convergence w.r.t. random processes. One type of the convergence that is needed to be explored in our problem context is the *convergence in probability*. Actually what the WLLN that I mentioned above says is that

$$\text{WLLN: } S_n \textit{ converges to } \mathbb{E}[Y] = p \textit{ in probability.}$$

What it means by this in words is very simple. It means that S_n converges to p *with high probability*. But in mathematics, the meaning should be rigorously stated. What it means by this in mathematics is that for any $\epsilon > 0$,

$$\Pr\left(|S_n - p| \leq \epsilon\right) \to 1.$$

In other words, S_n is within $p \pm \epsilon$ with high probability (w.h.p.).

[1]This is exactly the law which I mentioned in Lecture 4 Jacob Bernoulli discovered!

Now applying the WLLN to $\frac{\{\#\text{ of } a\text{'s}\}}{n}$ and $\frac{\{\#\text{ of } b\text{'s}\}}{n}$, we get: for any $\epsilon_1, \epsilon_2 > 0$,

$$\frac{\{\#\text{ of } a\text{'s}\}}{n} \text{ is within } p \pm \epsilon_1,$$
$$\frac{\{\#\text{ of } b\text{'s}\}}{n} \text{ is within } 1-p \pm \epsilon_2,$$

as $n \to \infty$. Applying this to (1), we can then say that in the limit of n,

$$\begin{aligned}\frac{1}{n}\log\frac{1}{p(X^n)} &= (p \pm \epsilon_1)\log\frac{1}{p} + (1-p \pm \epsilon_2)\log\frac{1}{1-p} \\ &= H(X) \pm \epsilon\end{aligned}$$

where $\epsilon := \epsilon_1 \log\frac{1}{p} + \epsilon_2 \log\frac{1}{1-p}$.

Manipulating the above, we get:

$$p(X^n) = 2^{-n(H(X)\pm\epsilon)} \text{ holds w.h.p.} \tag{2}$$

Here the key observation is that for very large n, ϵ can be made arbitrarily close to 0 and thus $p(X^n)$ is almost the same for all such X^n. We call such sequences *typical sequences*. The set that contains typical sequences is said to be a *typical* set, formally defined as below.

$$A_\epsilon^{(n)} := \{x^n : 2^{-n(H(p)+\epsilon)} \le p(X^n) \le 2^{-(H(p)-\epsilon)}\}.$$

Notice that $p(X^n) \approx 2^{-nH(X)}$ is almost uniformly distributed. It turns out this property plays a crucial role to design optimal codes. In the sequel, we will use this property to design such codes.

Prefix-free code design

What (2) suggests is that any arbitrary sequence is asymptotically *equiprobable*. Remember in LS2 that a good source code assigns a short-length codeword to a frequent symbol, while assigning a long-length codeword to a less frequent symbol. Here we see that any sequence is almost equally probable. This implies that the optimal length of a codeword assigned for any arbitrary sequence would be roughly the same as:

$$\ell^*(X^n) = \log\frac{1}{p(X^n)} \approx nH(X).$$

So one can consider a binary code tree in which the depth of the tree is roughly $nH(X)$ and codewords are assigned to leaves. Recall for a prefix-free code that codewords live only in leaves.

Now let's check if we can map all the possible sequence patterns of X^n into leaves. To this end, we need to check two values: (1) total number of leaves; (2) total number of possible input sequences. First of all, the total number of leaves is roughly $2^{nH(X)}$. Now what about the second value? Obviously the total number of input sequence patterns is 2^n because each symbol can take on one of the two values ("a" and "b") and we have n of them. But here one thing that we have to keep in our mind is that in the limit of n, the sequence X^n behaves in a particular manner, more specifically, in a manner that $p(x^n) \approx 2^{-nH(X)}$; hence, the number of such sequences is not the maximum possible value of 2^n. Then, now the question is: how many sequences such that $p(x^n) \approx 2^{-nH(X)}$? To see this, consider:

$$\sum_{x^n : p(x^n) \approx 2^{-nH(X)}} p(x^n) \approx |\{x^n : p(x^n) \approx 2^{-nH(X)}\}| \times 2^{-nH(X)}.$$

Obviously, the aggregation of all the probabilities of such sequences cannot exceed 1; hence,

$$|\{x^n : p(x^n) \approx 2^{-nH(X)}\}| \lesssim 2^{nH(X)}.$$

Note that the cardinality of such set does not exceed the total number of leaves $\approx 2^{nH(X)}$. So by using parts of leaves, we can map all of such sequences. This completes the design of source code. See Fig. 1. Finally let's check if this code indeed achieves the fundamental limit of $H(X)$. Notice that the length of every codeword is $\approx nH(X)$. So the expected codeword length per symbol would be $\approx \frac{nH(X)}{n} = H(X)$.

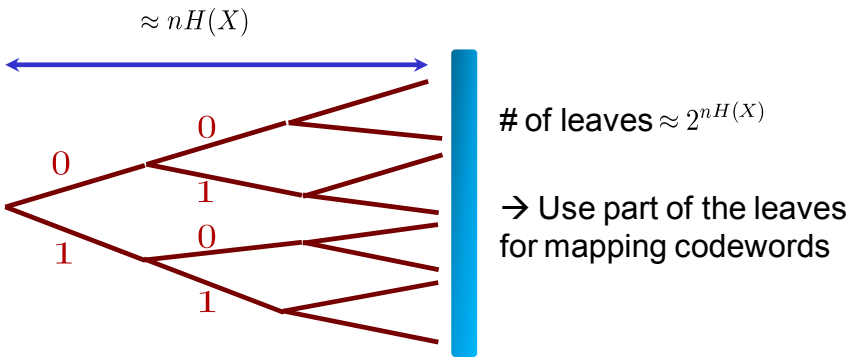


Figure 1: Design of optimal prefix-free codes

Extension to non-binary sources

So far we have considered the binary alphabet case. A natural question arises: how about for non-binary sources? Actually using the WLLN that we studied earlier, one can readily prove that:

$$\frac{1}{n}\log\frac{1}{p(X^n)} \longrightarrow H(X) \quad \text{in prob.}$$

although X is an *arbitrary* r.v., not limited to the binary one. Please check this is indeed the case in PS3. Roughly speaking, this implies that $p(X^n) \approx 2^{-nH(X)}$ w.h.p. - any arbitrary sequence is asymptotically equally probable. So using this and applying the code design rule based on a binary code tree, we can easily construct an optimal source code - every input sequence is mapped to a leaf in a binary code tree with depth $\approx nH(X)$ and hence, the expected codeword length per symbol is $H(X)$. The sequence pattern of a codeword is determined by which leaf the codeword is assigned to.

General non-i.i.d. sources

So far we have considered only i.i.d. sources. However, in practice, many of the information sources are far from i.i.d. One such example is an English text. Here is a concrete example where one can see this clearly. Suppose the first and second letters are "t" and "h" respectively. Then, what do you guess about the third letter? Yes, many of you would expect "e"! The reason is that there are many "the"s in an usual English text. What this example implies is that symbols that form a text are actually quite *correlated* with each other, meaning that the sequence is *dependent*. In fact, many other information sources are like that, not i.i.d. at all. Then, now the question is: What about for general non-i.i.d. sources? What is the corresponding source coding theorem? How to design optimal codes?

Actually we can readily answer these questions by applying the bounding techniques that we learned. Suppose we employ a super symbol-based source code of the super-symbol size n. Let $\mathcal{L}_n^* = \mathbb{E}[\ell(X^n)]$. Applying exactly the same lower-&-upper bound techniques that we learned, one can show that

$$H(X_1, X_2, \cdots, X^n) \leq \mathcal{L}_n^* \leq H(X_1, X_2, \cdots, X_n) + 1.$$

Dividing the above by n, we get:

$$\frac{H(X_1, X_2, \ldots, X^n)}{n} \leq \frac{\mathcal{L}_n^*}{n} \leq \frac{H(X_1, X_2, \ldots, X_n) + 1}{n}.$$

Here what we can say is that the expected codeword length per symbol is definitely related to the following quantity:

$$\lim_{n\to\infty} \frac{H(X_1, X_2, \ldots, X_n)}{n}.$$

Now the question is: Does the limit exist? If the limit exists, then we are done! The answer is: Not always. Actually there are some very contrived examples where the limit does not exist - one such example is in page 75 in CT. But the good news is: in many practically-relevant cases of our interest, the limit does exist. So let's consider only these cases where the limit exists. Then, we can state the source coding theorem as follows:

$$\textit{Minimum \# of bits that represent general source per symbol} = \lim_{n\to\infty} \frac{H(X_1, X_2, \ldots, X_n)}{n}.$$

Actually there is a terminology which indicates such limit. To understand the rationale behind the terminology that you will see soon, let's consider a plot in which x-axis and y-axis indicate n and $H(X_1, X_2, \ldots, X_n)$ respectively. See Fig. 2. What the above limit means is the *slope*. In other words, it means the growth *rate* of the sequence uncertainty w.r.t. n. Hence, it is called the *entropy rate*.

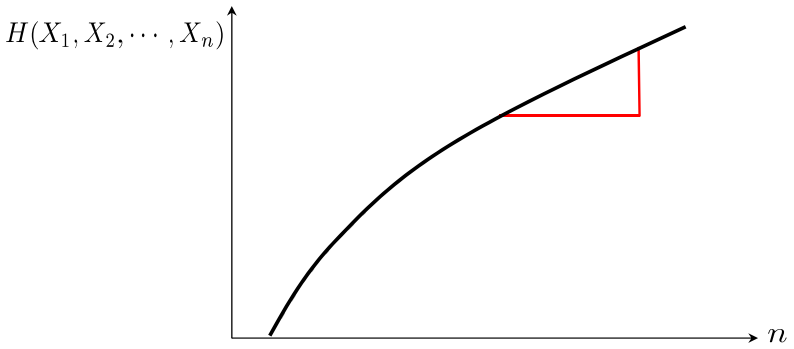


Figure 2: Entropy rate.

Look ahead

In the latter part of this lecture, we have proved the source coding theorem for general information source (not limited to the i.i.d. case): the minimum number of bits that represent information source per symbol is *entropy rate*. Next time, we will explore how to construct optimal codes that achieve such limit.

September 30, 2019
Changho Suh (chsuh@kaist.ac.kr)

Lecture 9: Source coding theorem for general sources

Recap

Last time, we studied how to construct an optimal code that can achieve $H(X)$ promised by the source coding theorem for the i.i.d source case. The construction was based on a super-symbol-based approach in which the source encoder acts on multiple input symbols, say n symbols, and the parameter n is of our design choice. The idea of the construction was to make the super-symbol size n arbitrarily large. In the interested regime of large n, what we found with the help of the WLLN is:

$$\frac{1}{n}\log\frac{1}{p(X^n)} \overset{\text{in prob.}}{\longrightarrow} H(X) \tag{1}$$

i.e., $\frac{1}{n}\log\frac{1}{p(X^n)}$ becomes in between $H(X)-\epsilon$ and $H(X)+\epsilon$ for any $\epsilon>0$, as n tends to infinity. Inspired by the fact that the codeword length solution for the lower bound in the optimization problem that we formulated earlier is $\ell^*(x^n)=\log\frac{1}{p(x^n)}$, we focused on the quantity of $\frac{1}{p(x^n)}$. From (1), we observed that in the limit of n, the quantity becomes:

$$\log\frac{1}{p(X^n)} \approx nH(X).$$

This then motivated us to consider a prefix free code in which an input sequence x^n is mapped to one of the leaves that reside in the level with the tree depth of $\approx nH(X)$. This ensures the expected codeword length per symbol to be $\approx H(X)$, matching with the promised limit. We also checked that the number of possible input sequences such that $p(x^n)\approx 2^{-nH(X)}$ is less than the total number $2^{nH(X)}$ of leaves. This ensured a mapping for every such input sequence.

Today's lecture

We have thus far considered only i.i.d. sources. However, in practice, many of the information sources are far from i.i.d. Remember the English text example that we introduced last time. So a natural question arises: What about for general non-i.i.d. sources? What is the corresponding source coding theorem? How to design optimal codes? In this lecture, we will address these questions.

Source coding theorem for general sources

Actually we can readily characterize the limit by applying the bounding techniques that we learned in previous lectures. Suppose we employ a super symbol-based source code of the super-symbol size n. Let $\mathcal{L}_n^*=\mathbb{E}\left[\ell(X^n)\right]$. Applying exactly the same lower-&-upper bound techniques that we learned, one can show that

$$H(X_1,X_2,\ldots,X^n) \le \mathcal{L}_n^* \le H(X_1,X_2,\ldots,X_n)+1.$$

Dividing the above by n, we get:

$$\frac{H(X_1,X_2,\ldots,X^n)}{n} \le \frac{\mathcal{L}_n^*}{n} \le \frac{H(X_1,X_2,\ldots,X_n)+1}{n}.$$

Here what we can say is that the expected codeword length per symbol is definitely related to the following quantity:

$$\lim_{n\to\infty} \frac{H(X_1, X_2, \ldots, X_n)}{n}.$$

Now the question is: Does the limit exist? If the limit exists, then we are done! However, the answer is: Not always. Actually there are some very contrived examples where the limit does not exist - one such example is in page 75 in CT. But there is a good news. The good news is: in many practically-relevant cases of our interest, the limit does exist. So let's consider only these cases where the limit exists. We can then state the source coding theorem as follows:

$$\textit{Minimum \# of bits that represent a general source per symbol} = \lim_{n\to\infty} \frac{H(X_1, X_2, \ldots, X_n)}{n}.$$

Actually there is a terminology which indicates such limit. To understand the rationale behind the terminology that you will see soon, let's consider a plot in which x-axis and y-axis indicate n and $H(X_1, X_2, \ldots, X_n)$ respectively. See Fig. 1. What the above limit means is the *slope*. In other words, it means the growth *rate* of the sequence uncertainty w.r.t. n. Hence, it is called the *entropy rate*.

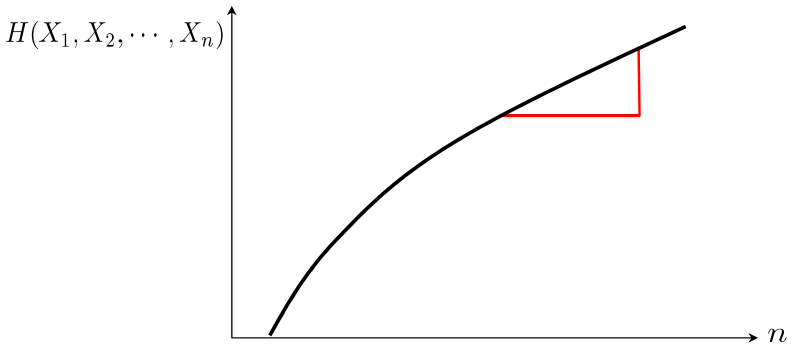


Figure 1: Entropy rate.

Stationary process

Now how to compute the entropy rate? It turns out in many practical information sources, the entropy rate is easy to compute. One such practical example is the one in which information source is a *stationary process*. Here is the definition of the stationary process. We say that a random process is *stationary* if $\{X_i\}$ is a *statistical copy* (e.g., having the same joint distribution) of its shifted version $\{X_{i+\ell}\}$ for any non-negative integer ℓ. One concrete example is again an English text. Notice that the statistics of 10-year-old text would be almost the same as that of a nowaday's text; for instance, the frequency of "the" in an old text would be roughly the same as that of a current text.

As claimed earlier, the limit can be simplified a bit further when specialized to the stationary process case. To see this, we consider:

$$\begin{aligned} H(X_1, X_2, \ldots, X_n) &= H(X_1) + H(X_2|X_1) + \cdots + H(X_n|X_1, \ldots, X_{n-1}) \\ &= \sum_{i=1}^{n} H(X_i|X_1, X_2, \ldots, X_{i-1}) \\ &= \sum_{i=1}^{n} H(X_i|X^{i-1}) \end{aligned}$$

where the first equality is due to the chain rule and the last comes from the shorthand notation that we introduced earlier: $X^{i-1} := (X_1, \ldots, X_{i-1})$. Let $a_i = H(X_i|X^{i-1})$. Now we can see two properties of the deterministic sequence $\{a_i\}$. First it is non-negative. Second, it is non-increasing, i.e., $a_{i+1} \leq a_i$. To see this, consider:

$$\begin{aligned} a_{i+1} &= H(X_{i+1}|X_1, X_2, \ldots, X_i) \\ &\leq H(X_{i+1}| \quad X_2, \ldots, X_i) \\ &= H(X_i|X_1, \ldots, X_{i-1}) \\ &= a_i \end{aligned}$$

where the inequality follows from the fact that conditioning reduces entropy and the second last equality is due to the stationarity of the process. These two properties imply that the deterministic sequence $\{a_i\}$ has a limit. Why? Please think about the *definition of the existence of a limit* that you learned perhaps from the course on Calculus or somewhere else. Otherwise, look for wikipedia.

Now let's consider

$$\frac{H(X_1, \ldots, X_n)}{n} = \frac{1}{n} \sum_{i=1}^{n} a_i.$$

Notice that this quantity indicates the *running average* of the sequence $\{a_i\}$. Keeping in your mind that $\{a_i\}$ has a limit, what your intuition says is that the running average will converge to the limit as well because almost all the components that form the running average will converge to the limit. It turns out this is indeed the case. So the entropy rate of the stationary process is:

$$H(\mathcal{X}) = \lim_{i \to \infty} H(X_i|X^{i-1}). \tag{2}$$

Please see PS3 for the rigorous proof of this.

Now how to design optimal codes for such a stationary process? We can apply the same methodology that we developed earlier. The first thing to do is to check that such a stationary sequence is also asymptotically equiprobable. In fact, one can resort to a generalized version of the WLLN to prove that this is indeed the case:

$$p(X^n) \approx 2^{-nH(\mathcal{X})}. \tag{3}$$

The proof of this is not that simple, requiring some non-trivial tricks. So we will not deal with the proof here. But if you are interested, I am happy to issue a problem for the proof in a later homework, say PS4 or PS5. Please let me know your interest in class or via any communication route. Recalling what we learned last time, what (3) suggests is that the optimal code assigns the same codeword length for every sequence and the length should read roughly $nH(\mathcal{X})$. The sequence patterns will be determined by the binary code tree of depth $\approx nH(\mathcal{X})$ in which codewords are mapped to leaves.

From theory to practice

So far we have characterized the limit (entropy rate) on the number of bits that represent information source (stationary process in almost all practically-relevant cases), and also learned about how to design optimal codes that achieve the limit. But here one thing to notice is that we focused on a somewhat idealistic scenario in which the super-symbol size n can be made arbitrarily large. In reality, n should be *finite* though. Why? Two reasons. One is that the

hardware that we will employ for the use of a code obviously cannot support an infinite size of n. The other is that the amount of information source available is definitely limited, as pointed out by a student in class. So one natural question that we can ask in this context is then: What if n is finite? What are optimal codes for finite n? Actually this is exactly the question that Shannon raised right after establishing the source coding theorem. Also this question was shared with a bunch of professors in MIT at that time including Prof. Robert Fano. Prof. Fano also shared this question with students who took the information theory course that he held at that time.

As I mentioned earlier, for the finite n case, we need to solve a *non-convex* optimization problem in order to give a concrete answer. I told you that this problem is extremely hard, and a closed-form solution to the limit has been open thus far. So Prof. Fano never expected that any one would come up with a solution to this very difficult problem.

But surprisingly, one of the students in class, named David Huffmnan, came up with a very simple and straightforward algorithm which leads to the optimal code. Huffman did not provide the exact closed-form solution to the limit, but instead could provide a very explicit design rule, called *algorithm*, which leads to the optimal code. This code is called the *Huffman code*. One amazing fact is that he did this as a term project in the class, implying that you may also be able to do something great right now!

While the Huffman code provides the optimal code in a practical scenario where n is finite, it still has limitations in implementation. One such limitation is that the codes require *knowledge of source statistics*. In other words, to design Huffman code, we need to know about the joint distribution of a super symbol: $p(x_1, x_2, \ldots, x_n)$. But in practice, it may not be that easy to obtain the statistical knowledge. So one follow-up question that arises in this more practical scenario is: What if source statistics is *unknown*? Are there any optimal codes that require no priori knowledge on source statistics?

A universal code

Two very smart Jewish people could answer this question: Lempel & Ziv. They developed a *universal* code that does not require any statistical knowledge on information sources. This code is called *Lempel-Ziv* code. Since this code can be applied to any type of information source while still approaching the limit (entropy rate), many system designers were very interested in implementing their code. So it turns out the code was implemented in a variety of systems as the following protocols named: gzip, pkzip, UNIX compression.

The idea of the Lempel-Ziv code is very simple. The idea is the one that many people in our daily life are using. Let us explain the idea in one specific context: "closing email phrases". As a closing phrase, people usually use the following:

"I look forward to your reply.";

"I look forward to seeing you.";

"I look forward to hearing from you.";

"Your prompt repay would be appreciated.", *etc.*

Actually if one intends to represent this English phrase as it is, then we may need a bunch of bits, of which the number is definitely larger than the number of alphabets in a phrase. But the Lempel-Ziv code says that we can do much better than this. The idea is based on *dictionary*. One noticeble property is that we have only a few number of closing phrases because people usually use very similar phrases. Inspired by this, one can imagine a dictionary that maps each phrase into an index like 1, 2, 3. This dictionary forms the basis of the Lempel-Ziv code.

Roughly speaking, here is how the Lempel-Ziv code works. First of all, we make a dictionary from some pilot sequence which are part of the entire sequence. We then share this between source encoder and decoder. The source encoder encodes only indices and the decocder decoes the indices using the shared dictionary. This way, we do not need to know about the statistics of information sources. Also it has been shown that this code can achieve the limit (entropy rate) as the dictionary size increases.

In the interest of time, we will not cover more details on the Huffman code and the Lempel-Ziv code. For those who are interested in, I will upload a supplementary material w.r.t. the Huffman code. Unfortunately, I don't have a course note for the Lempel-Ziv code. So if you are interested in, please look for wikipedia.

Look ahead

So far we have proved the source coding theorem for general information source, and learned about how to design optimal codes that achieve the limit. We also briefly overviewed some practical codes such as the Huffman code and the Lempel-Ziv code. Next time, we will move onto another core topic in Part II: the channel coding theorem.

October 2, 2019
Changho Suh (chsuh@kaist.ac.kr)

Lecture 10: Overview of channel coding theorem

Recap

In the first two lectures, we learned about the two-stage communication architecture that Shannon developed. The role of the first stage that we call *source coding* is to convert an information source (which is possibly of a different type) into a common currency of information, *bits*. The role of the second stage that we call *channel coding* is to convert the bits into a signal so that it can be reliably transmitted over a channel to the receiver. During the past five lectures, we studied the *source coding theorem* which quantifies the fundamental limit on the number of bits required to represent an information source without loss of meaning. In the upcoming five lectures (including today's one), we will study the *channel coding theorem* which quantifies the fundamental limit on the number of bits that can be reliably transmitted over a channel.

Today's lecture

Remember the verbal statement of the channel coding theorem that we made in Lecture 2. Like a law in physics, there is a law in communication systems: Given a channel, the maximum number of bits that can be transmitted over a channel is *determined*, regardless of any operations done at the transmitter and the receiver. In other words, there is a fundamental limit on the number of bits, under which communication is possible and above which communication is impossible, no matter what we do and whatsoever. The limit is so called *channel capacity*. In today's lecture, we will study what this statement means in a more explicit and precise manner. To this end, we will study the following: (i) a specific problem setup that Shannon considered; (ii) a mathematical model of channels; (iii) precise meaning of *possible* (or *impossible*) communication. Once the statement becomes clearer, we will later prove the theorem.

Problem setup

The goal of the communication problem is to deliver an information of binary string type (bits) to the receiver as much as possible. Let's start by investigating the statistics of the binary string. Fortunately, unlike the information source in the source coding context, one can make a very simple assumption on the statistics of the binary string.

To see this, let's recall the source coding setup that we considered in the past. Notice that an information source is of arbitrary statistics depending on applications of our interest. So the source code design was tailored for the statistics of the information source. On the other hand, we have a good news in the channel coding part. The good news is that the statistics of the input binary string is not arbitrary - it follows a particular statistics under a reasonable assumption. Here the reasonable assumption is that we use an *optimal* source code. Now then can you see why the binary string is of a particular statistics?

To see this, let's recall the source coding theorem that we proved. Let the binary string $(b_1, b_2, \ldots, b_m)$ be an output of an optimal source encoder. Then, using the source coding theorem, we get:

$$\begin{aligned} m &= H(\text{information source}) \\ &= H(b_1, b_2, \ldots, b_m) \end{aligned} \tag{1}$$

where the second equality follows from the fact that a source encoder is one-to-one mapping and the fact $H(X) = H(f(X))$ for an one-to-one mapping function f (Why?). Now observe that

$$\begin{aligned} H(b_1, b_2, \ldots, b_m) &= H(b_1) + H(b_2|b_1) + \cdots + H(b_m|b_1, \ldots, b_{m-1}) \\ &\leq H(b_1) + H(b_2) + \cdots + H(b_m) \\ &\leq m \end{aligned} \tag{2}$$

where the first inequality is due to the fact that conditioning reduces entropy and the last inequality comes from the fact that b_i's are binary r.v. This together with (1) suggests that the inequalities in (2) are actually tight. This implies that b_i's are independent (from the first inequality) and identically distributed $\sim \mathsf{Bern}(\frac{1}{2})$ (from the second inequality). Hence, we can make the following simple assumption: a binary string, an input to a channel encoder, is i.i.d., each being according to $\mathsf{Bern}(\frac{1}{2})$.

For notational simplicity, information theorists introduced a simple notation which indicates such an i.i.d. random process $\{b_i\}$. They expressed it with a single r.v., say W, using the following mapping rule:

$$\begin{aligned} (b_1, \ldots, b_m) &= (0, 0, \ldots, 0) \longrightarrow W = 1; \\ (b_1, \ldots, b_m) &= (0, 0, \ldots, 1) \longrightarrow W = 2; \\ &\vdots \\ (b_1, \ldots, b_m) &= (1, 1, \ldots, 1) \longrightarrow W = 2^m. \end{aligned}$$

This allows us to express the input only with a single r.v. W and the distribution of W is *uniform*. Why uniform?

Things to design

The design of the digital communication system means the design of two things: (i) channel encoder, say f; (ii) channel decoder, say g. See Fig. 1. One crucial thing to consider in designing the encoder and the decoder is the behavior of *channel*. As mentioned earlier, the channel is an enemy against the communication system because it induces errors, making the system a *random* function. So the encoder and the decoder should be designed so as to protect against such errors. A natural way to protect against the errors is to add some redundancy. Notice in our conversation that we *say again* when the communication is not successful. Inspired by this, we can think of a *sequence* of symbols instead of a single quantity for the channel encoder output, say $(X_1, X_2, \ldots, X_n)$. Here we use a shorthand notation $X^n := (X_1, X_2, \ldots, X_n)$ for simplicity, and n is called *code length*. Note here that n has nothing to do with the super-symbol size employed in the source coding context. Denote by Y^n the output of the channel. The goal of the decoder g is to infer W given the channel output Y^n. Here as we can easily imagine, there is a challenge in decoding W. The challenge is that Y^n is not a *deterministic* function of X^n. Actually if the channel were deterministic, then the function g is nothing but an inverse function of the concatenation of the two: f and channel. But the problem is that the channel is not deterministic. So the design of g is not that simple. Actually the design of g requires a deep understanding of how the channel behaves. So before getting into details as to how to design f and g, let's briefly talk about how the channel behaves as well as how to model it.

Channel modeling

Let X_i and Y_i be input and output of the channel at time i, respectively. One typical way to capture the randomness of the channel is via a conditional distribution: $p(y|x)$. To give you an idea, let us consider one concrete example: binary symmetric channel (BSC). The other example

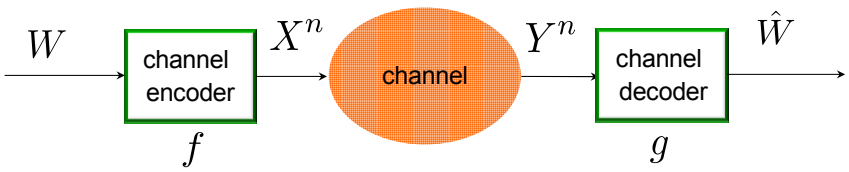


Figure 1: Channel coding problem setup

that one can think of is: binary erasure channel (BEC). But this is the one that we studied in the past (Lecture 4). So let's consider only the BSC.

In the BSC, Y_i is a flipped version of X_i with probability, say p (called crossover probability): $p(y|x) = p$ when $y \neq x$. So the relation between X_i and Y_i is captured by the following:

$$Y_i = \begin{cases} X_i, & \text{w.p. } 1-p; \\ X_i \oplus 1, & \text{w.p. } p. \end{cases}$$

Another simpler way to represent this is:

$$Y_i = X_i \oplus Z_i$$

where $Z_i \sim \mathsf{Bern}(p)$. Usually we consider a so called mememoryless channel in which the noises are independent across different time instants. So one natural assumption is that Z_i's are i.i.d.

Two performance metrics

Let us go back to our main business, which is to design f and g such that communication is possible. But to this end, we need to fully understand what it means by *possible* communication. To explain what the possible communication means, we need to rely on two performance metrics and the relationship between the two.

The first performance metric is the one that captures the amount of bits that we wish to transmit. Suppose $W \in \{1, 2, \ldots, M\}$. Then, the total number of bits that we intend to send is $\log M$. Since code length is n, the number of channels (time instants) that we use is n and hence, the number of bits transmitted per channel use (called *data rate*) is

$$R = \frac{\log M}{n} \quad \text{bits/channel use.}$$

The second performance metric is the one which captures the decoding quality. Since the channel is random, we cannot always guarantee the decoded message (say $\hat{W}$) to be the same as the original message W. So the decoding quality can be captured by the probability of an error event:

$$P_e := \Pr\{W \neq \hat{W}\}.$$

The smaller P_e, the better the decoding quality.

Optimization problem

With these two performance metrics, Shannon came up with a very natural optimization problem to be explained soon. He then came up with a concept of *possible* communication. Let us first take a look at the optimization problem. One can easily expect that there should be tradeoff

relationship between R and P_e. The larger R, the larger P_e and vice versa. So one natural optimization problem is: Given R and n:

$$P_e^* := \min_{f,g} P_e.$$

What Shannon realized is that unfortunately, this is again a *non-convex* optimization problem, being very difficult to solve. So Shannon took a different approach, as he did for the source coding theorem. This is, to *approximate*! In other words, he attempted to develop upper and lower bounds on P_e^*.

In the process of doing this, he discovered something very interesting and came up with a concept of possible communication. What he found is that when R is below some value, an *upper* bound on P_e^* can be made arbitrarily close to 0 as n increases. This implies that in that case, actual P_e^* can be made very small. He also found that when R is above some value, an *lower* bound on P_e^* *cannot* be made arbitrarily close to 0 no matter what we do. This implies that in that case, actual P_e^* cannot be made very small.

From this observation, he then came up with the concept of *possible* communication. We say that communication is *possible* if one can make P_e^* arbitrarily close to 0 as $n \to \infty$; otherwise communication is said to be *impossible*. He also came up with a follow-up natural concept of *achievable* rate. We say that data rate R is *achievable* if we can make $P_e^* \to 0$ as $n \to \infty$ given R; otherwise R is said to be not achievable.

Channel coding theorem

Moreover, Shannon made a very interesting observation. What he observed is that the threshold below which communication is possible and above which communication impossible is *sharp*. In other words, there is a sharp *phase transition* on the achievable rate. He then called that limit *channel capacity* and denoted it by C. This forms the channel coding theorem:

The maximum achievable rate is channel capacity C.

Now from the next lecture on, we will try to prove this theorem. Specifically we will prove two things:

$$R \le C \Longrightarrow P_e \to 0$$
$$R > C \Longrightarrow P_e \nrightarrow 0.$$

The first is called the *achievability* proof or the *direct* proof. The second is called the *converse* proof. Notice that the contraposition of the second statement is:

$$P_e \to 0 \Longrightarrow R \le C,$$

which is exactly the converse of the first statement.

Looking ahead

Next time, we will prove the achievability for a simple example. Gaining insights from the example, we will later prove the acheivability for a general channel specified by an arbitrary conditional distribution $p(y|x)$.

October 7, 2019
Changho Suh (chsuh@kaist.ac.kr)

Lecture 11: Achievability proof of the binary erasure channel

Recap

Last time we introduced the channel coding problem setup. Specifically we came up with a precise mathematical statement of channel coding theorem which was only vaguely stated in the beginning of this course. To this end, we first took into account two performance metrics: (i) data rate $R := \frac{\log M}{n}$ (bits/channel use); (ii) probability of error $P_e := \Pr(W \neq \hat{W})$. Here the message $W \in \{1, 2, \ldots, M\}$ and n indicates code length (the number of channels used). In an effort to understand the tradeoff relationship between the two performance metrics, we then considered the following optimization problem: Given R and n:

$$P_e^* = \min_{f,g} P_e$$

where f and g indicate channel encoder and decoder respectively. Unfortunately, Shannon could not solve this problem. Instead he looked into upper and lower bounds. In the process of doing this, he made a very interesting observation: If R is less then a threshold, say C, then P_e can be made arbitrarily close to 0 as $n \to \infty$; otherwise, $P_e \nrightarrow 0$ no matter what we do. This leads to a natural concept of *achievable rate*: data rate R is said to be *achievable* if given R we can make $P_e \to 0$. This finally formed the channel coding theorem:

$$\text{Maximum achievable rate } = C.$$

The channel coding theorem requires the proof of two parts. The first one is: $R \leq C \Longrightarrow P_e \to 0$. To this end, we need to come up with an *achievable* scheme such that given $R \leq C$, one can make $P_e \to 0$. Hence, the proof is so called the *achievability* proof. The second part to prove is: $R > C \Longrightarrow P_e \nrightarrow 0$. Notice that the contraposition of this statement is: $P_e \to 0 \Longrightarrow R \leq C$, which is exactly the opposite of the statement for the first part proof. Hence, it is so called the *converse* proof.

Today's lecture

In today's lecture, we will attempt to prove the achievability. Specifically we will try to gain some insights from a simple channel example: the binary erasure channel (BEC). It turns out the BEC provides significant insights into proving the achievability for arbitrary channels. Once we are equipped with enough insights, we will later attack a fairly general channel model setting.

Binary erasure channel (BEC)

Remember in the BEC that the channel output reads:

$$Y_i = \begin{cases} X_i, & \text{w.p. } 1-p; \\ \mathsf{e}, & \text{w.p. } p, \end{cases}$$

where e stands for "erasure" (garbage) and p indicates the erasure probability. As mentioned earlier, we consider a *memoryless channel* in which noises are independent across different time instances.

First of all, let's guess what the channel capacity C is. Actually one can make a naive guess based on the following observation: the channel is perfect w.p. $1-p$; erased w.p. p; and the erasure (garbage) does not provide any information w.r.t. what we transmit. This naturally leads to: C is at most $1-p$. Then, a question arises: Is this achievable? Let's think about a very easy case in which the transmitter knows all erasure patterns across all time instances beforehand. Of course this is far from reality. But just for simplicity, let's think about this case for the time being. In this case, one can readily achieve $1-p$. The achievable scheme is: Transmit a bit whenever the channel is perfect. Since the perfect channel probability is $1-p$, by the Law of Large Number (LLN), we can achieve $1-p$ as n tends to infinity.

Now let's consider the realistic scenario in which we *cannot predict* the future events and thus the transmitter has no idea of erasure patterns. In this case, we cannot apply the above naive transmission scheme because each transmission of a bit is not guaranteed to be successful due to the lack of the knowledge of erasure patterns. Perhaps one may imagine that it is impossible to achieve $1-p$. But it turns out that very interestingly one can still achieve $1-p$ even in this realistic setup.

How to encode?

Here is a transmission scheme. First fix R arbitrarily close to $1-p$. Now given R, what is M (the cardinality of the range of the message W)? Since $R := \frac{\log M}{n}$, one needs to set as: $M = 2^{nR}$. So the message W takes one of the values among $1, 2, \ldots, 2^{nR}$.

Next, how to encode the message W? In other words, what is a mapping rule between W and X^n? Here we call X^n *codeword*[1]. It turns out Shannon's encoding rule, to be explained in the sequel, enables us to achieve $R = 1-p$. Shannon's encoding is extremely simple - it looks even dumb! (You'll see why soon). The idea is to generate a *random* binary string given any $W = w$. In other words, for any $W = w$, every component of $X^n(w)$ follows a binary r.v. with parameter $\frac{1}{2}$ and those are independent with each other, i.e., $X_i(w)$'s are i.i.d. $\sim \mathsf{Bern}(\frac{1}{2})$ across all i's. These are i.i.d. also across all w's. This looks like a really dumb scheme. But surprisingly Shannon showed that this dumb scheme can achieve the rate of $1-p$. Here there is a terminology which indicates a collection (book) of $X_i(w)$'s. It is called *codebook*.

How to decode?

Let's move onto the decoder side. Obviously the decoder input is a received signal Y^n (channel output). Here one can make a reasonable assumption on the decoder: the codebook (the collection of $X_i(w)$'s) is known. This assumption is realistic because this information is the one that suffices to share only at the beginning of communication. Once this information is shared, we can use this all the time until communication is terminated.

Now the question is: How to decode W from Y^n, assuming that the codebook is known? What is an optimal way of decoding W? Remember the second performance metric: the probability of error $P_e := \Pr(W \neq \hat{W})$. Since the channel is not deterministic (i.e., *random*), the only thing that we can do for the best is to minimize the probability of error as small as possible. So the optimal decoder can be defined as the one that *minimizes the probability of error*. An alternative way to define the optimal decoder is: Optimal decoder is the one that *maximizes the success probability*: $\Pr(W = \hat{W})$. But here the received signal is given because it is an input to the decoder. So the success probability should be a conditional probability:

$$\Pr(W = \hat{W} | Y^n = y^n).$$

[1]The word of *codeword* was used to indicate the output of the source encoder in prior lectures. It is a sort of convention to use the same word also to indicate the output of *channel* encoder.

Then, a more formal definition of the optimal decoder is:

$$\hat{W} = \arg\max_{w} \Pr(W = w | Y^n = y^n).$$

Actually there is a terminology which indicates such optimal decoder. Notice that $\Pr(W = w|Y^n = y^n)$ is the probability *after* an observation y^n is made; and $\Pr(W = w)$ is so called the *a priori* probability because it is the one known *beforehand*. Hence, $\Pr(W = w|Y^n = y^n)$ is called the *a posteriori* probability. Observe that the optimal decoder is the one that Maximizes A Posteriori (MAP) probability. So it is called the MAP decoder. The MAP decoder acts as an optimal decoder for many interesting problems not limited to this problem context. As long as a problem of interest is an *inference problem* (*infer* X from Y when X and Y are *probabilistically* related), then the optimal decoder is always the MAP decoder.

In fact, this MAP decoder can be simplified further in many cases including our case as a special case. Here is how it is simplified. Using the definition of conditional probability, we get:

$$\begin{aligned}\Pr(W = w|Y^n = y^n) &= \frac{\Pr(W = w, Y^n = y^n)}{\Pr(Y^n = y^n)} \\ &= \frac{\Pr(W = w)}{\Pr(Y^n = y^n)} \cdot \Pr(Y^n = y^n|W = w).\end{aligned}$$

Here one thing to notice is that $\Pr(W = w) = \frac{1}{2^{nR}}$ - it is irrelevant to w. Also $\Pr(Y^n = y^n)$ is not a function of w. This implies that it suffices to consider only $\Pr(Y^n = y^n|W = w)$ in figuring out when the above probability is maximized. Hence, we get:

$$\begin{aligned}\hat{W} &= \arg\max_{w} \Pr(W = w|Y^n = y^n) \\ &= \arg\max_{w} \Pr(Y^n = y^n|W = w).\end{aligned}$$

Observe that given $W = w$, X^n is known as $x^n(w)$. Hence,

$$\begin{aligned}\hat{W} &= \arg\max_{w} \Pr(Y^n = y^n|W = w) \\ &= \arg\max_{w} \Pr(Y^n = y^n|X^n = x^n(w), W = w).\end{aligned}$$

Also given $X^n = x^n(w)$, Y^n is a sole function of the channel, meaning that given $X^n = x^n(w)$, Y^n is independent of w. Hence,

$$\begin{aligned}\hat{W} &= \arg\max_{w} \Pr(Y^n = y^n|X^n = x^n(w), W = w) \\ &= \arg\max_{w} \Pr(Y^n = y^n|X^n = x^n(w)).\end{aligned}$$

Notice that $\Pr(Y^n = y^n|X^n = x^n(w))$ is nothing but a conditional distribution, given by the channel, which is easy to compute. Actually there is another name for the conditional distribution: *likelihood*. So the decoder is called the Maximum Likelihood (ML) decoder.

How to derive the ML decoder?

It turns out the ML decoder is extremely simple in our problem context. Let's see this through a very simple example where $n = 4$ and $R = \frac{1}{2}$. In this example, $M = 2^{nR} = 4$. Consider a particular codebook example as follows:

$$\begin{aligned}X^n(1) &= 0000; \\ X^n(2) &= 0110; \\ X^n(3) &= 1010; \\ X^n(4) &= 1111.\end{aligned}$$

Suppose $y^n = 0\mathsf{e}00$. Then, we can easily compute the likelihood $\mathbb{P}(y^n|x^n(w))$. For instance,

$$\mathbb{P}(y^n|x^n(1)) = (1-p)^3 p$$

because the channel is perfect three times (time 1, 3 and 4), while being erased at time 2. On the other hand,

$$\mathbb{P}(y^n|x^n(2)) = 0$$

because the third bit 1 does not match $y_3 = 0$, and this is the event that would never happen, meaning that the probability must be 0. In other words, the second message is *incompatible* with the received signal y^n. Then, what is the ML decoding rule? Here is how it works.

1. We eliminate all the messages which are incompatible with the received signal.
2. If there is only one message that survives, then we declare the survival is the correct message that the transmitter sent.

However, this procedure is not sufficient to describe the ML decoding rule. The reason is that we may have a different erasure pattern that confuses the rule. To see this clearly, consider the following concrete example. Suppose that the received signal is now $y^n = (0\mathsf{ee}0)$. Then, we see that

$$X^n(1) = (1-p)^2 p^2;$$
$$X^n(2) = (1-p)^2 p^2.$$

Now the two messages (1 and 2) are compatible and those likelihood functions are *equal*. In this case, what we can do for the best is to flip a coin, choosing one out of the two in a random manner. This forms the ML decoding rule.

3. If there are multiple survivals, choose one randomly.

Look ahead

Next time, we will demonstrate that we can achieve the rate of $1-p$ under the optimal ML decoding rule.

October 14, 2019
Changho Suh (chsuh@kaist.ac.kr)

Lecture 12: Achievability proof of BEC & BSC

Recap

Last time, we claimed that the capacity of the BEC with erasure probability is

$$C_{\mathsf{BEC}} = 1 - p.$$

We then attempted to prove achievability: if $R < 1 - p$, we can make P_e arbitrarily close to 0 as $n \to \infty$. We employed a *random* codebook where each component $X_i(w)$ of the codebook follows $\mathsf{Bern}(\frac{1}{2})$ and i.i.d. across all $i \in \{1, \ldots, n\}$ and $w \in \{1, \ldots, 2^{nR}\}$. We also employed the optimal decoder: the maximum likelihood (ML) decoder in our problem context where the message W is uniformly distributed. We then tried to complete achievability proof, by showing that $P_e \to 0$ under the problem setup.

Today's lecture

Today we are going to complete this proof. We will then consider another channel example which turns out give further insights into a general channel setting. The channel is the one that we saw earlier: the binary symmetric channel.

Probability of error

The probability of error is a function of codebook $\mathcal{C}$. So here we are going to investigate the *average* error probability $\mathbb{E}_{\mathcal{C}}[P_e(\mathcal{C})]$ taken over all possible random codebooks, and will demonstrate that $\mathbb{E}_{\mathcal{C}}[P_e(\mathcal{C})]$ approaches 0 as $n \to \infty$. This then implies the existence of an optimal *deterministic* codebook, say $\mathcal{C}^*$, such that $P_e(\mathcal{C}^*) \to 0$ as $n \to \infty$ (Why? See PS4 for details), which completes achievability proof. Consider:

$$\begin{aligned}
\mathbb{E}_{\mathcal{C}}[P_e(\mathcal{C})] &\overset{(a)}{=} \sum_c \Pr(\mathcal{C} = c) \Pr(\hat{W} \neq W | \mathcal{C} = c) \\
&\overset{(b)}{=} \Pr(\hat{W} \neq W) \\
&\overset{(c)}{=} \sum_{w=1}^{2^{nR}} \Pr(W = w) \Pr\left(\hat{W} \neq w | W = w\right)
\end{aligned} \tag{1}$$

where (a) follows from the fact that $\Pr(\mathcal{C} = c)$ indicates the probability that codebook is chosen as a particular realization c; (b) and (c) are due to the total probability law (check Ch. 1 and 2 in BT if you do not get it).

Now consider $\Pr(\hat{W} \neq w | W = w)$. Notice that $[X_1(w), \ldots, X_n(w)]$'s are identically distributed over all w's. This means that the way that the wth codeword is constructed is exactly the same for all w's. This implies that $\Pr(\hat{W} \neq w | W = w)$ is irrelevant of what the particular value of w is, meaning that

$$\Pr(\hat{W} \neq w | W = w) \text{ is the same for all } w.$$

This together with (1) gives:

$$\begin{aligned}\mathbb{E}_{\mathcal{C}}[P_e(\mathcal{C})] &= \Pr\left(\hat{W} \neq 1 | W = 1\right) \\ &= \Pr\left(\bigcup_{w=2}^{2^{nR}} \{\hat{W} = w\} | W = 1\right).\end{aligned} \tag{2}$$

Here the second equality is due to the fact that $\{\hat{W} \neq 1\}$ implies that $\hat{W} = w$ for some $w \neq 1$. In general, the probability of the *union* of multiple events is not that simple to compute. Actually it is quite complicated especially when it involves a large number of multiple events. Even for the three-event case (A, B, C), the probability formula is not that simple:

$$\begin{aligned}\Pr(A \cup B \cup C) =& \Pr(A) + \Pr(B) + \Pr(C) \\ &- \Pr(A \cap B) - \Pr(A \cap C) - \Pr(B \cap C) + \Pr(A \cap B \cap C).\end{aligned}$$

Even worse, the number of associated multiple events here in (2) is $2^{nR}-1$ and this will make the formula of that probability very complicated. So the calculation of that probability is not that simple. Of course, Shannon realized the difficulty of the calculation. To make some progress, he took an indirect approach as he did in the proof of the source coding theorem. He did not attempt to compute the probability of error exactly. Instead, he intended to *approximate* it! Specifically he tried to derive an *upper* bound because at the end of day what we want to show is that the probability of error approaches 0. Note that if an upper bound goes to zero, then the exact quantity also approaches 0. We have a very well-known upper bound w.r.t. the union of multiple events. That is, the *union bound*: for events A and B,

$$\Pr(A \cup B) \leq \Pr(A) + \Pr(B).$$

The proof is immediate because $\Pr(A \cap B) \geq 0$.

Now applying the union bound to (2), we get:

$$\begin{aligned}\mathbb{E}_{\mathcal{C}}[P_e(\mathcal{C})] &\leq \sum_{w=2}^{2^{nR}} \Pr\left(\hat{W} = w | W = 1\right) \\ &= (2^{nR} - 1)\Pr\left(\hat{W} = 2 | W = 1\right)\end{aligned}$$

where the equality follows from the fact that the codebook employed is symmetric w.r.t message indices. The event of $\hat{W} = 2$ implies that message 2 is *compatible*; otherwise, $\hat{W}$ cannot be chosen as 2. Using the fact that for two events A and B such that A implies B, $\Pr(A) \leq \Pr(B)$, we get:

$$\begin{aligned}\mathbb{E}_{\mathcal{C}}[P_e(\mathcal{C})] &\leq (2^{nR} - 1)\Pr\left(\text{message 2 is } \textit{compatible} | W = 1\right) \\ &\leq 2^{nR}\Pr\left(\text{message 2 is } \textit{compatible} | W = 1\right).\end{aligned}$$

Notice that the fact that message 2 is compatible implies that for time i in which the transmitted signal is not erased, $X_i(2)$ must be the same as $X_i(1)$; otherwise, message 2 cannot be compatible. To see this clearly, let $\mathcal{B} = \{i : Y_i \neq \mathsf{e}\}$. Then, what the above means is that we get:

$$\begin{aligned}\mathbb{E}_{\mathcal{C}}[P_e(\mathcal{C})] &\leq 2^{nR}\Pr\left(\bigcap_{i \in \mathcal{B}} \{X_i(1) = X_i(2)\} | W = 1\right) \\ &= 2^{nR}\left(\frac{1}{2}\right)^{|\mathcal{B}|}\end{aligned} \tag{3}$$

where the equality is due to the fact that $\Pr(X_i(1) = X_i(2)) = \frac{1}{2}$ and the codebook is independent across all i's.

Now here one key thing to notice is that due to the LLN, for sufficiently large n:

$$|\mathcal{B}| \approx n(1-p).$$

This together with (3) yields:

$$\mathbb{E}_{\mathcal{C}}[P_e(\mathcal{C})] \lesssim 2^{n(R-(1-p))}.$$

Hence, $\mathbb{E}_{\mathcal{C}}[P_e(\mathcal{C})]$ can be made arbitrarily close to 0 as $n \to \infty$, as long as $R < 1-p$. This completes achievability proof.

Binary symmetric channel (BSC)

Before generalizing to arbitrary channels, let's exercise a bit more, exploring the binary symmetric channel (BSC). The reason is that the techniques employed in the BEC achievability proof are not enough for generalization, while the techniques that we will learn in the BSC achievability proof turn to provide enough insights into extension.

Remember in the BSC that:

$$Y_i = X_i \oplus Z_i$$

where Z_i's are i.i.d. $\sim \mathsf{Bern}(p)$. Without loss of generality, assume that $p \in (0, \frac{1}{2})$; otherwise, we can flip all 0's and 1's to 1's and 0's, respectively.

Let's start with claiming what the capacity is:

$$C_{\mathsf{BSC}} = 1 - H(p)$$

where $H(p) := p \log \frac{1}{p} + (1-p) \log \frac{1}{1-p}$. For the rest of this lecture, we will prove achievability: if $R < 1 - H(p)$, one can make the probability of error arbitrarily close to 0 as $n \to \infty$.

Encoder & Decoder

The encoder that we will employ is exactly the same as before: the *random* code where $X_i(w)$'s are i.i.d. $\sim \mathsf{Bern}(\frac{1}{2})$ across all i's and w's. We will also use the optimal decoder, which is the ML decoder:

$$\hat{W}_{\mathsf{ML}} = \arg\max_{w} \mathbb{P}\left(y^n | x^n(w)\right).$$

But the way that we compute the likelihood function is different from that in the BEC case. Let's see this difference through the following example. Suppose that $(n, R) = (4, \frac{1}{2})$ and the codebook is:

$$\begin{aligned} X^n(1) &= 0000; \\ X^n(2) &= 1110; \\ X^n(3) &= 1010; \\ X^n(4) &= 1111. \end{aligned}$$

Suppose that the received signal y^n is (0100). Then, the likelihood functions are:

$$\begin{aligned} \mathbb{P}(y^n|x^n(1)) &= (1-p)^3 p^1; \\ \mathbb{P}(y^n|x^n(2)) &= (1-p)^2 p^2; \\ \mathbb{P}(y^n|x^n(3)) &= (1-p) p^3; \\ \mathbb{P}(y^n|x^n(4)) &= (1-p) p^3. \end{aligned}$$

Here one thing to notice is that unlike the BEC case, all the messages are *compatible* because the likelihood functions are strictly positive. So we need to now compare all the functions to choose the message that maximizes the function. Since we assume that $p \in (0, \frac{1}{2})$ without loss of generality, in the above example, the first message is the maximizer. Notice that it has the minimum number of flips (marked in red) and the flipping probability is smaller than $\frac{1}{2}$, and hence, the corresponding likelihood function is maximized.

From this, one can see that the number of non-matching bits (number of flips) plays a crucial role in a decision:

$$d(x^n, y^n) := |\{i : x_i \neq y_i\}| .$$

This is so called the *Hamming* distance. Using this, the ML decoder can be re-written as:

$$\hat{W}_{\mathsf{ML}} = \arg\min_w d(x^n(w), y^n).$$

Now what remains to do is to show that one can make the probability of error arbitrarily close to 0 as $n \to \infty$, when using the ML decoder. However, here we will not employ the ML decoder because of two reasons. The first is that the error probability analysis is a bit complicated. The second one is more critical: the proof technique is not extensible to general cases in which channels are arbitrary. Now then how can we prove the achievabiliy without using the ML decoder? Fortunately, there is a different yet *suboptimal* decoder under which the achievability proof can be greatly simplified as well as which still achieves $1 - H(p)$. More importantly, the suboptimal decoder that we will describe soon is *extensible*. So we are going to employ such suboptimal decoder to prove the achievability.

A suboptimal decoder

The suboptimal decoder that we will employ is inspired by a key observation that one can make. Notice that the input to the decoder is Y^n and codeword is available at the decoder, i.e., $X^n(w)$ is known for all w's. Now observe that

$$Y^n \oplus X^n = Z^n$$

and we know the statistics of Z^n: i.i.d. $\sim \mathsf{Bern}(p)$.

Suppose that actually transmitted signal is $X^n(1)$. Then,

$$Y^n \oplus X^n(1) = Z^n \ \sim \mathsf{Bern}(p).$$

On the other hand, for $w \neq 1$,

$$Y^n \oplus X^n(w) = X^n(1) \oplus X^n(w) \oplus Z^n.$$

Here the key observation is that the statistics of the resulting sequence is $\mathsf{Bern}(\frac{1}{2})$. Notice that the sum of any two independent Bernoulli r.v. is $\mathsf{Bern}(\frac{1}{2})$, as long as at least one r.v. follows $\mathsf{Bern}(\frac{1}{2})$ (Why?). This motivates us to employ the following decoding rule.

1. Compute $Y^n \oplus X^n(w)$ for all w's.
2. Eliminate messages such that the resulting sequence is *not typical* w.r.t. $\mathsf{Bern}(p)$. More precisely, let

   $$A_\epsilon^{(n)} := \{z^n : 2^{-n(H(p)+\epsilon)} \leq p(z^n) \leq 2^{-n(H(p)-\epsilon)}\}$$

   be a typical set w.r.t. $\mathsf{Bern}(p)$. Eliminate all w's such that $Y^n \oplus X^n(w) \notin A_\epsilon^{(n)}$.

3. If there is only one message that survives, then declare the survival is the correct message. Otherwise, declare an error.

Note that the error event is of two types: (i) multiple survivals; (ii) no survival.

Probability of error

We are now ready to analyze the probability of error to complete achievability proof. For notational simplicity, let $\bar{P}_e := \mathbb{E}_{\mathcal{C}}[P_e(\mathcal{C})]$. Using exactly the same argument that we made in the BEC case, we can get:

$$\bar{P}_e := \mathbb{E}_{\mathcal{C}}[P_e(\mathcal{C})] = \Pr\left(\hat{W} \neq 1 | W = 1\right).$$

As mentioned earlier, the error event is of the two types: (i) multiple survivals; (ii) no survival. The multiple-survival event implies that there exists $w \neq 1$ such that $Y^n \oplus X^n(w) \in A_\epsilon^{(n)}$. The no-survival event implies that even the $Y^n \oplus X^n(1)$ w.r.t. the correct message "1" is not a typical sequence, meaning that $Z^n \notin A_\epsilon^{(n)}$. Hence, we get:

$$\begin{aligned}
\bar{P}_e &= \Pr\left(\hat{W} \neq 1 | W = 1\right) \\
&\leq \Pr\left(\bigcup_{w \neq 1} \{Y^n \oplus X^n(w) \in A_\epsilon^{(n)}\} \cup \{Z^n \notin A_\epsilon^{(n)}\} | W = 1\right) \\
&\overset{(a)}{\leq} \sum_{w=2}^{2^{nR}} \Pr\left(\{Y^n \oplus X^n(w) \in A_\epsilon^{(n)}\} | W = 1\right) + \Pr\left(Z^n \notin A_\epsilon^{(n)} | W = 1\right) \\
&\overset{(b)}{=} (2^{nR} - 1) \Pr\left(Y^n \oplus X^n(2) \in A_\epsilon^{(n)} | W = 1\right) + \Pr\left(Z^n \notin A_\epsilon^{(n)} | W = 1\right) \\
&\overset{(c)}{\approx} (2^{nR} - 1) \Pr\left(Y^n \oplus X^n(2) \in A_\epsilon^{(n)} | W = 1\right)
\end{aligned} \tag{4}$$

where (a) follows from the union bound; (b) is by symmetry of the codebook; and (c) follows from the fact that Z^n is a typical sequence w.h.p due to the WLLN. Now observe that

$$Y^n \oplus X^n(2) = X^n(1) \oplus X^n(2) \oplus Z^n \sim \mathsf{Bern}\left(\frac{1}{2}\right).$$

How to compute $\Pr(Y^n \oplus X^n(2) \in A_\epsilon^{(n)} | W = 1)$? To compute this, we need to consider two quantities: (i) the total number of possible sequences that $Y^n \oplus X^n(2) \sim \mathsf{Bern}(\frac{1}{2})$ can take on; (ii) the size of the typical set $|A_\epsilon^{(n)}|$. Specifically, we have:

$$\Pr\left(Y^n \oplus X^n(2) \in A_\epsilon^{(n)} | W = 1\right) = \frac{\left|A_\epsilon^{(n)}\right|}{\text{total number of } \mathsf{Bern}(\frac{1}{2}) \text{ sequences}}$$

Note that the total number of $\mathsf{Bern}(\frac{1}{2})$ sequences is 2^n and $|A_\epsilon^{(n)}| \leq 2^{n(H(p)+\epsilon)}$ (Why?). Hence,

$$\Pr\left(Y^n \oplus X^n(2) \in A_\epsilon^{(n)} | W = 1\right) \leq \frac{2^{n(H(p)+\epsilon)}}{2^n}.$$

This together with (4), we get:

$$\bar{P}_e \lesssim 2^{n(R-(1-H(p)-\epsilon))}.$$

Since ϵ can be made arbitrarily close to 0, we can conclude that if $R < 1 - H(p)$, then $\bar{P}_e \to 0$ as $n \to \infty$. This completes the proof.

Look ahead

We have proved achievability for BSC. We will defer the converse proof for the time being. The reason is that it turns out the converse proof technique can be applied to a very general problem context directly. So next time, we will extend achievability proof to the general case in which the channel is characterized by an arbitrary conditional distribution $p(y|x)$. Later we will prove converse for the general case.

October 16, 2019
Changho Suh (chsuh@kaist.ac.kr)

Lecture 13: Achievability proof of DMC

Recap & outline

Last time, we have proved the achievability for BSC. We will defer the converse proof for the time being. The reason is that it turns out the converse proof technique can be applied to a very general problem context directly. So we will first extend achievability proof to the general case in which the channel is characterized by an arbitrary conditional distribution $p(y|x)$. Later we will prove converse for the general case.

Discrete memoryless channel (DMC)

The general channel that we will consider is the one so called the *discrete memoryless channel* (DMC). Let's start with the definition of the channel. We say that a channel is DMC if input and output are on discrete-alphabet sets and the following condition is satisfied:

$$p(y_i|x_i, x^{i-1}, y^{i-1}, W) = p(y_i|x_i).$$

This condition is called *memoryless*. Notice that given the current channel input x_i, the current output y_i is independent of the past input/output (x^{i-1}, y^{i-1}) and any other things including the message W. Here one key property that we need to keep in our mind is:

$$p(y^n|x^n) = \prod_{i=1}^{n} p(y_i|x_i). \tag{1}$$

This can be proved by using the memoryless property (check in PS5). It turns out this property plays a crucial role in proving achievability as well as converse. This will be clearer later.

Guess on the capacity formula

Let's start by guessing the capacity formula for the DMC. Remember the capacity formulas of the BEC and BSC: $C_{\mathsf{BEC}} = 1 - p$; $C_{\mathsf{BSC}} = 1 - H(p)$. Here one thing that we can say about is that these capacities are closely related to one key notion that we introduced earlier: *mutual information*. Specifically what we can easily show is: when $X \sim \mathsf{Bern}(\frac{1}{2})$,

$$C_{\mathsf{BEC}} = 1 - p = I(X;Y);$$
$$C_{\mathsf{BSC}} = 1 - H(p) = I(X;Y).$$

Also one can verify that for an arbitrary distribution of X,

$$C_{\mathsf{BEC}} = 1 - p \geq I(X;Y);$$
$$C_{\mathsf{BSC}} = 1 - H(p) \geq I(X;Y).$$

Check this in PS5. Hence, what one can guess on the capacity formula from this is:

$$C_{\mathsf{DMC}} = \max_{p(x)} I(X;Y). \tag{2}$$

It turns out it is indeed the case. For the rest of this lecture, we will prove achievability: if $R < \max_{p(x)} I(X;Y)$, then the probability of error can be made arbitrarily close to 0 as $n \to \infty$.

Encoder

The encoder that we will employ is a *random* code, meaning that $X_i(w)$'s are generated in an i.i.d. fashion according to some input distribution $p(x)$. We know in the BEC and BSC that the input distribution is fixed as $\mathsf{Bern}(\frac{1}{2})$. And one can verify that $\mathsf{Bern}(\frac{1}{2})$ is the *maximizer* of the above optimization problem (2). This motivates us to choose $p(x)$ as:

$$p^*(x) = \arg\max_{p(x)} I(X;Y).$$

It turns out this choice enables us to achieve the capacity guessed in (2).

Decoder

As before, we also assume that the codebook is known at the decoder. We use the suboptimal decoder that we employed in the BSC case. Remember that the suboptimal decoder is based on a typical sequence and the fact that $Y^n \oplus X^n = Z^n$. But one significant distinction in the general DMC case is that Y^n is a kind of *arbitrary random function* of X^n. So now what we can do is to take a look at *pairs* of $(Y^n, X^n(w))$ and to check if we can see a particular behavior of the pair associated with the true codeword, compared to the other pairs w.r.t. the wrong codewords.

To illustrate this clearly, let's consider the following situation. Suppose that $X^n(1)$ is actually transmitted, i.e., the message 1 is the correct one. Then, the joint distribution of the correct pair $(x^n(1), y^n)$ would be:

$$\begin{aligned} p(x^n(1), y^n) &= p(x^n(1))p(y^n|x^n(1)) \\ &\overset{(a)}{=} \prod_{i=1}^{n} p(x_i(1))p(y_i|x_i(1)) \\ &= \prod_{i=1}^{n} p(x_i(1), y_i) \end{aligned}$$

where (a) follows from the key property (1). This implies that the pairs $(x_i(1), y_i)$'s are i.i.d over i's. Then, using the WLLN on the i.i.d. sequence of pairs, one can easily show that

$$\frac{1}{n} \log \frac{1}{p(X^n(1), Y^n)} \longrightarrow H(X,Y) \quad \text{in prob.}$$

Check in PS5. From this, one can say that

$$p(x^n(1), y^n) \approx 2^{-nH(X,Y)}$$

for sufficiently large n (Why?).

On the other hand, the joint distribution of the wrong pair $(x^n(w), y^n)$ for $w \neq 1$ would be:

$$p(x^n(w), y^n) = p(x^n(w))p(y^n).$$

The reason is that y^n is associated only with $(x^n(1), \text{channel noise})$, which is *independent* of $x^n(w)$. Remember that we use a random code in which codewords are independent with each

other. Also here one can verify that y^n is also i.i.d. (check!). Again using the WLLN on $x^n(w)$ and y^n, one can get:

$$p(x^n(w), y^n) \approx 2^{-n(H(X)+H(Y))}$$

for sufficiently large n. This motivates the following decoder. Let

$$\begin{aligned} A_\epsilon^{(n)} = \Big\{(x^n, y^n) : 2^{-n(H(X)+\epsilon)} &\leq p(x^n) \leq 2^{-n(H(X)-\epsilon)} \\ 2^{-n(H(Y)+\epsilon)} &\leq p(y^n) \leq 2^{-n(H(Y)-\epsilon)} \\ 2^{-n(H(X,Y)+\epsilon)} &\leq p(x^n, y^n) \leq 2^{-n(H(X)+H(Y)-\epsilon)}\Big\}. \end{aligned}$$

Then, the decoding rule is as follows:

1. Eliminate all the messages w's such that $(x^n(w), y^n) \notin A_\epsilon^{(n)}$.
2. If there is only one message that survives, then declare that the survival is the correct message.
3. Otherwise, declare an error.

Similar to the previous case, note that the error event is of two types: (i) multiple survivals (or one wrong survival); (ii) no survival.

Probability of error

We are now ready to analyze the probability of error to complete the achievability proof. Using exactly the same argument that we made in the BEC case, we can get:

$$\bar{P}_e := \mathbb{E}_{\mathcal{C}}[P_e(\mathcal{C})] = \Pr\left(\hat{W} \neq 1 | W = 1\right).$$

Now as mentioned earlier, the error event is of the two types: (i) multiple survivals; (ii) no survival. The multiple-survival event implies the existence of the wrong pair being a jointly typical pair, meaning that there exists $w \neq 1$ such that $(X^n(w), Y^n) \in A_\epsilon^{(n)}$. The no-survival event implies that even the correct pair is not jointly typical, meaning that $(X^n(1), Y^n) \notin A_\epsilon^{(n)}$. Hence, we get:

$$\begin{aligned} \bar{P}_e &= \Pr\left(\hat{W} \neq 1 | W = 1\right) \\ &\leq \Pr\left(\bigcup_{w \neq 1} \{(X^n(w), Y^n) \in A_\epsilon^{(n)}\} \cup \{(X^n(1), Y^n) \notin A_\epsilon^{(n)}\} | W = 1\right) \\ &\overset{(a)}{\leq} \sum_{w=2}^{2^{nR}} \Pr\left(\{(X^n(w), Y^n) \in A_\epsilon^{(n)}\} | W = 1\right) + \Pr\left((X^n(1), Y^n) \notin A_\epsilon^{(n)} | W = 1\right) \\ &\overset{(b)}{\leq} 2^{nR} \Pr\left((X^n(2), Y^n) \in A_\epsilon^{(n)} | W = 1\right) + \Pr\left((X^n(1), Y^n) \notin A_\epsilon^{(n)} | W = 1\right) \\ &\overset{(c)}{\approx} 2^{nR} \Pr\left((X^n(2), Y^n) \in A_\epsilon^{(n)} | W = 1\right) \end{aligned}$$

where (a) follows from the union bound; (b) is by symmetry of the codewords w.r.t. message indices; and (c) follows from the fact that $(X^n(1), Y^n))$ is jointly typical for sufficiently large n w.h.p due to WLLN. Now observe that $X^n(2)$ and Y^n are independent. So the total number of pair patterns w.r.t. $(X^n(2), Y^n)$ would be:

$$\begin{aligned}\text{total number of } (X^n(2), Y^n) \text{ pairs} &\approx 2^{nH(X)} \cdot 2^{nH(Y)} \\ &= 2^{n(H(X)+H(Y))}.\end{aligned}$$

On the other hand, the cardinality of the jointly typical pair set $A_\epsilon^{(n)}$ is:

$$|A_\epsilon^{(n)}| \approx 2^{nH(X,Y)}$$

(Why?). Hence, we get:

$$\begin{aligned}\Pr\left(Y^n \oplus X^n(2) \in A_\epsilon^{(n)} | W = 1\right) &= \frac{\left|A_\epsilon^{(n)}\right|}{\text{total number of } (X^n(2), Y^n) \text{ pairs}} \\ &\approx \frac{2^{n(H(X,Y)-H(X)-H(Y))}}{2^n} \\ &= \frac{2^{-nI(X;Y)}}{2^n}.\end{aligned}$$

Using this, we get:

$$\bar{P}_e \lesssim 2^{n(R-I(X;Y))}.$$

So one can conclude that if $R < I(X;Y)$, then $\bar{P}_e \to 0$ as $n \to \infty$. Since we choose $p(x)$ such that $I(X;Y)$ is maximized, we see that $\max_{p(x)} I(X;Y)$ is achievable, which completes the proof.

Look ahead

So far we have proved achievability for discrete memoryless channels. Next time, we will prove converse to complete the proof of channel coding theorem.

October 28, 2019
Changho Suh (chsuh@kaist.ac.kr)

Lecture 14: Converse proof

Recap & outline

Last time, we have proven achievability for discrete mememoryless channels which are described by conditional distribution $p(y|x)$:

$$R < \max_{p(x)} I(X;Y) \Longrightarrow P_e \to 0.$$

Today's lecture

Today we will prove converse:

$$P_e \to 0 \Longrightarrow R < \max_{p(x)} I(X;Y).$$

The proof relies on two important inequalities: (1) Fano's inequality; (2) data processing inequality (DPI). Let us start by studying the two inequalities.

Fano's inequality

Fano's inequality is one of the well-known inequalities that plays a significant role in the context of *inference* problems. Here the inference problem is the one of which the task is to infer input from output given that the input and output are related in a *probabilistic* manner. Notice in the inference problems that the input and output are probabilistically related, and so the only thing that we can do w.r.t. the input from the output is to *guess* about the input. In a mathematical language, the guess refers to *inference*. Actually the communication problem of our interest can be viewed as an inference problem because the goal in the problem is to infer the message W from the received signal Y^n that is *stochastically* related to W. In this inference problem context, Fano's inequality captures relationship between the following performance metrics:

$$P_e := \Pr(\hat{W} \neq W) \ \& \ \ H(W|\hat{W}).$$

Here what one can easily expect is that the smaller P_e, the smaller $H(W|\hat{W})$. Fano's inequality captures such relation. Remember in the converse converse that we need to come up with a lower bound on P_e because at the end of the day, we intend to show that $P_e \nrightarrow 0$ (when $R > C$), and the fact that a lower bound of P_e does not goes to 0 ensures the actual P_e not to converge to 0. Here Fano's inequality plays a role to provide such a lower bound. Actually a lower bound on P_e in terms of $H(W|\hat{W})$ can be written as an upper bound on $H(W|\hat{W})$ in terms of P_e. And the upper bound formula is well-known and we will focus on such upper-bound presentation:

$$\textit{Fano's inequality: } H(W|\hat{W}) \leq 1 + P_e \cdot nR. \tag{1}$$

Notice that this indeed captures the tradeoff relationship between P_e and $H(W|\hat{W})$: the smaller P_e, the more contracted $H(W|\hat{W})$. The proof of this is very simple. Let

$$E = \mathbf{1}\{\hat{W} \neq W\}.$$

By the definition of the probability of error and the fact that $\mathbb{E}[\mathbf{1}\{\hat{W} \neq W\}] = \Pr\{\hat{W} \neq W\}$, $E \sim \mathsf{Bern}(P_e)$. Starting with the fact that E is a function of $(W, \hat{W})$, we have:

$$\begin{aligned} H(W|\hat{W}) &= H(W, E|\hat{W}) \\ &\overset{(a)}{=} H(E|\hat{W}) + H(W|\hat{W}, E) \\ &\overset{(b)}{\leq} 1 + H(W|\hat{W}, E) \\ &\overset{(c)}{=} 1 + \Pr(E = 1) \cdot H(W|\hat{W}, E = 1) \\ &= 1 + P_e \cdot H(W|\hat{W}, E = 1) \\ &\overset{(d)}{\leq} 1 + P_e \cdot nR \end{aligned}$$

where (a) is due to a chain rule; (b) follows from the cardinality bound on entropy; (c) comes from the definition of conditional entropy; and (d) follows from the cardinality bound on entropy.

Data processing inequality (DPI)

The second inequality that plays a role in the converse proof is the data processing inequality, DPI for short. In words, the DPI says that any processed data of the original data cannot improve the quality of inference. In our problem context, it means that the interference quality on W based on X^n (let's view this as original data) cannot be worse than that based on a processed data, say Y^n. Actually the precise mathematical statement of the DPI is made in the context of a Markov process. So let's first study what the Markov process is. We say that a random process, say $\{X_i\}$, is a Markov process if

$$p(x_{i+1}|x_i, x_{i-1}, \ldots, x_1) = p(x_{i+1}|x_i).$$

Here what this condition means is that given the current state x_i, the future state x_{i+1} and the past states x^{i-1}'s are independent with each other. There is a very well-known graphical representation of such a Markov process. Suppose that (X_1, X_2, X_3) forms a Markov process. Then, we represent it as:

$$X_1 - X_2 - X_3.$$

The rationale behind this representation is based on the following observation: Given X_2, it can be removed from the above graph as it is known; but this removal leads (X_3, X_1) to be *disconnected*, meaning that these have nothing to do with each other, in a statistical language, being independent. Note that the graph looks like a chain. So it is also called a Markov *chain*.

In our problem context, there is a Markov chain. First of all, one can show that (W, X^n, Y^n) forms a Markov chain:

$$W - X^n - Y^n.$$

The proof is straightforward. Given X^n, Y^n is a sole function of the noise induced by the channel, which has nothing to do with the message W, meaning that (Y^n, W) are independent with each other.

The DPI is the one defined w.r.t. the Markov chain. Specifically it captures the relationship between the following mutual information: $I(W; X^n)$, $I(W; Y^n)$, $I(X^n; Y^n)$. Actually $I(W; X^n)$ captures somehow common information between W and X^n, hence it can be interpreted as the amount of information regarding W that one can infer from X^n. So it can be viewed as *inference quality* on W when the inference is based on X^n. Similarly one can view $I(W; Y^n)$ as inference

quality on W when the inference is based on Y^n. Remember the verbal statement of the DPI: any processed data (say Y^n) of the original data (say X^n) cannot improve inference quality on W, meaning that

$$I(W;Y^n) \leq I(W;X^n).$$

The proof of this is very simple. Starting with a chain rule and non-negativity of mutual information, we have:

$$\begin{aligned} I(W;Y^n) &\leq I(W;Y^n,X^n) \\ &= I(W;X^n) + I(W;Y^n|X^n) \\ &\overset{(a)}{=} I(W;X^n) \end{aligned} \tag{2}$$

where (a) follows from the fact that $W - X^n - Y^n$. There is another DPI w.r.t. $I(W;Y^n)$ and another mutual information $I(X^n;Y^n)$. Note that X^n is closer to Y^n compared to W. So one can guess:

$$I(W;Y^n) \leq I(X^n;Y^n).$$

It turns out this is indeed the case. The proof is also very simple.

$$\begin{aligned} I(W;Y^n) &\leq I(W,X^n;Y^n) \\ &= I(X^n;Y^n) + I(W;Y^n|X^n) \\ &= I(X^n;Y^n). \end{aligned} \tag{3}$$

From (2) and (3), one can summarize that the mutual information between two ending terms in the Markov chain does not exceed the mutual information between any two terms that lie in-between the two ends.

Looking at our problem setting, there is another term: $\hat{W}$. It turns out this together with the above Markov chain $(W - X^n - Y^n)$ forms a longer Markov chain:

$$W - X^n - Y^n - \hat{W}.$$

Notice that given Y^n, $\hat{W}$ is completely determined regardless of (W, X^n) because it is a function of Y^n. Now applying the DPI, one can readily verify that

$$I(W;\hat{W}) \leq I(W;Y^n) \tag{4}$$
$$I(W;\hat{W}) \leq I(W;X^n) \tag{5}$$
$$I(W;\hat{W}) \leq I(X^n;\hat{W}) \tag{6}$$
$$I(W;\hat{W}) \leq I(X^n;Y^n) \tag{7}$$
$$I(W;\hat{W}) \leq I(Y^n;\hat{W}). \tag{8}$$

Converse proof

We are now ready to prove the converse with the two inequalities that we learned. Starting with

the fact that $nR = H(W)$ (why?), we have:

$$\begin{aligned}
nR &= H(W) \\
&= I(W;\hat{W}) + H(W|\hat{W}) \\
&\overset{(a)}{\leq} I(W;\hat{W}) + 1 + P_e \cdot nR \\
&\overset{(b)}{=} I(W;\hat{W}) + n\epsilon_n \\
&\overset{(c)}{\leq} I(X^n;Y^n) + n\epsilon_n \\
&= H(Y^n) - H(Y^n|X^n) + n\epsilon_n \\
&\overset{(d)}{=} H(Y^n) - \sum_{i=1}^{n} H(Y_i|X_i) + n\epsilon_n \\
&\overset{(e)}{\leq} \sum_{i=1}^{n} [H(Y_i) - H(Y_i|X_i)] + n\epsilon_n \\
&= \sum_{i=1}^{n} I(X_i;Y_i) + n\epsilon_n \\
&\overset{(f)}{\leq} nC + n\epsilon_n
\end{aligned}$$

where (a) follows from Fano's inequality (1); (b) comes from the definition of ϵ_n that we set as:

$$\epsilon_n := \frac{1}{n}(1 + nP_eR);$$

(c) is due to the DPI (7); (d) follows from the memoryless channel property ($p(Y^n|X^n) = \prod_{i=1}^{n} p(Y_i|X_i)$ and hence $H(Y^n|X^n) = \sum_{i=1}^{n} H(Y_i|X_i)$); (e) conditioning reduces entropy; and (f) is due to the definition $C := \max_{p(x)} I(X;Y)$. Now dividing by n on both sides, we get:

$$R \leq C + \epsilon_n.$$

If $P_e \to 0$, then $\epsilon_n := \frac{1}{n}(1 + P_enR)$ tends to 0 as $n \to \infty$. Hence, we get:

$$R \leq C,$$

which completes the proof.

Look ahead

So far we have proved the channel coding theorem for a fairly general channel: discrete memoryless channel. But in the proof, we only showed the existence of an optimal code such that under the code P_e can be made arbitrarily close to 0 as $n \to 0$. Next time, we will discuss explicit *deterministic* codes which approach (or exactly achieve) the capacity. This together with the proof of the channel coding theorem forms all the contents of Part II. So right after the discussion on deterministic codes, we will embark on Part III.

October 30, 2019
Changho Suh (chsuh@kaist.ac.kr)

Lecture 15: Overview of Part III and supervised learning

Recap

Last time, we have proven the converse for discrete mememoryless channels to complete the proof of the channel coding theorem:

$$C = \max_{p(x)} I(X;Y).$$

This completes Part II. Before moving onto Part III, let me tell you an interesting story behind the development of the channel coding theorem, which later motivated some efforts on explicit code constructions that we did not deal with in this course.

Initial reactions on Shannon's work

The story started from interesting initial reactions that people had at the time on Shannon's work. Actually people's initial responses were not that good. Many people, especially communication systems *engineers*, did not appreciate the work. The reasons are three-folded.

First and mostly importantly, many of the engineers did not understand what Shannon was talking about. Perhaps they were not familiar with the concept of reliable (possible) communication, achievable data rate and capacity. This might prevent them from grasping the gist of the theorem.

Second, even the smart engineers who understood the result had a pessimistic viewpoint on the development of an optimal code because the optimal code, as per the achievability proof of the theorem, seems to require a sufficiently long code-length to achieve the capacity. So they believed that complexity is way too high, considering the technology of the day; and hence, the optimal code is far from implementation.

Lastly, even the smart and optimistic engineers who appreciated the potential of implementation possibly with an advanced technology emerging in the future were not that positive. The reason is that Shannon did not really tell people *exactly how to design* an optimal code. His idea for the achievability proof is based on *random coding* - he never talked about how to *explicitly construct* such an optimal code. In other words, the proof only showed the *existence* of optimal codes.

Two major efforts

Due to these reasons, Shannon and his advocators including a few smart MIT folks made significant efforts to develop explicit & deterministic optimal codes possibly with low implementation complexity. Actually Shannon himself failed to come up with a good deterministic code. Instead his advocators at MIT came up with some good codes. Here we will discuss two major efforts by them.

One effort was made by Robert G. Gallager[1]. He developed a code called *"Low Density Parity Check code"* (LDPC code for short) in 1960. Obviously it is an explicit and deterministic code, meaning that it provides a detailed guideline as to how to design such a code. The performance of the code is also remarkable. It has been shown that it approaches the capacity as the code

[1]He is the right person who wrote the Gallager's note that I uploaded on the website. He is actually my academic grandfather - an advisor of my PhD advisor.

length tends to infinity, although it does not match the capacity exactly. Encouragingly (perhaps especially to you guys), he developed such a code during his "PhD study" as the first piece of work - meaning that you can also do that kind of great works in the near future!

Unfortunately, his work did not receive enough credits at that time. The main reason is that the LDPC code was still of high implementation complexity considering the digital signal processing (DSP) technology of the day.[2] 30 years later, however, the code was revived, receiving significant attention. The reason is that the code *became* an *efficient* code – the DSP technology had been evolved, finally enabling the code to be implemented. Later the code was more appreciated with a more advanced DSP technology – it is now widely being employed in a variety of systems such as LTE, WiFi, DMB[3] and storage systems.

At the time when Gallager developed the LDPC code, he was not fully satisfied with his result though. The reason was that his code is not guaranteed to *exactly achieve* the capacity even in the limit of code length. This motivated one of his PhD students, Erdal Arikan, to work on developing the *capacity-achieving* deterministic code. It turned out Arikan could develop such a code, called *Polar code*. It is a first capacity-achieving deterministic code. Interestingly, he could develop the code in 2007, 30+ years later than his PhD day when the motivation began.[4] Due to its great performance and low-complexity nature, it is now being seriously considered for implementation in a variety of systems.

These are the two major efforts. Due to the interest of time, we will not cover these in this course. If you are interested in details, I can instead share some course notes w.r.t. the Polar code which I drafted in earlier semesters. Please let me know your interest. Unfortunately, I made no course note for the LDPC code.

Summary of Part I and II

So far we have studied the core contents of information theory. In Part I, we learned three key notions that arise in the field: (i) entropy; (ii) mutual information; (iii) the Kullback-Leibler (KL) divergence. In Part II, we explored the roles of such notions in the context of Shannon's two major theorems: source coding theorem and channel coding theorem. Specifically we found that the entropy serves to characterize the fundamental limit on the number of bits that represent information source, promised by source coding theorem; the mutual information serves as a key measure for the channel capacity, characterized in channel coding theorem.

Goal of Part III

Encouragingly, the key notions have proved quite instrumental in a widening array of other fields and disciplines, ranging from social networks (e.g., community detection via Facebook's friendship graph), computational biology (e.g., DNA and RNA sequencing), to machine learning and deep learning.

The goal of Part III is to demonstrate such roles in one recent spotlight field that has been revolutionized in the past decades: *machine learning and deep learning*. In particular, we aim at investigating: (i) the role of entropy for one of the prominent methodologies of machine learning, called *supervised learning*; (ii) the role of KL divergence for another popular methodology, called *unsupervised learning*; (iii) the role of mutual information in the design of machine learning

[2] Nonetheless, he became an MIT faculty right after graduation mainly due to that piece of work. It is fortunate that there are some serious & patient scholars who appreciate the *potential* of the great piece of work which does not demonstrate any immediate impact upon the world.

[3] It is the name of the technology for a digital broadcast system, standing for Digital Multmedia Broadcasting.

[4] Remarkably he devoted most of his time to the development of the code, and finally succeeded. How dramatic the story is!

algorithms that ensure *fairness* for disadvantageous against advantageous groups.

During upcoming lectures

During the upcoming lectures (possibly three), we will focus on the first: the role of entropy for supervised learning. Specifically we will first study what supervised learning is and then formulate an optimization problem based on the methodology. Next we will demonstrate that the entropy plays a central role to define an objective function that leads to an optimal architecture for such optimization problem. Lastly we will learn how to solve the optimization problem via one powerful and computationally-efficient algorithm, called the *gradient descent algorithm*. The last process also includes *programming* implementation via one of the prominent and most intuitive deep learning frameworks, named Pytorch. If you are not familiar with Pytorch, please read the pytorch tutorial (uploaded on the course website), as well as try the last problem of PS5.

Machine learning

Machine learning is about an algorithm which is defined to be a set of instructions that a computer system can execute. Formally speaking, machine learning is the study of *algorithms* with which one can train a computer system so that the trained system can perform a specific task of interest. Pictorially, it means the following; see Fig. 1.

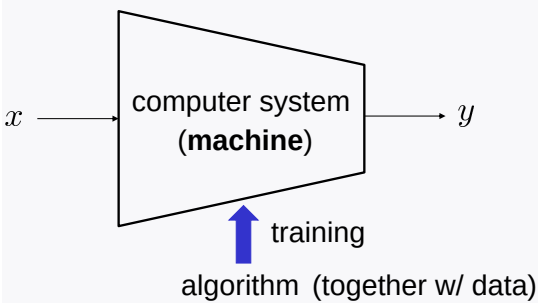


Figure 1: Machine learning is the study of algorithms which provide a set of explicit instructions to a computer system (machine) so that it can perform a specific task of interest. Let x be the input which indicates information employed to perform a task. Let y be the output that denotes a task result.

Here the entity that we are interested in building up is a computer system, which is definitely a *machine*. Since it is a system (i.e., a function), it has the input and the output. The input, usually denoted by x, indicates information which is employed to perform a task of interest. The output, usually denoted by y, indicates a task result. For instance, if a task of interest is legitimate-emails filtering against spam emails, then x could be multi-dimensional quantities[5]: (i) frequency of a keyword like *dollar signs* \$\$\$; (ii) frequency of another keyword, say *winner*. And y could be an email entity, e.g., $y = +1$ indicates a legitimate email while $y = -1$ denotes a spam email. In machine learning, such y is called a *label*. Or if an interested task is cat-vs-dog classification, then x could be *image-pixel values* and y is a binary value indicating whether the fed image is a cat (say $y = 1$) or a dog ($y = 0$).

Machine learning is about designing *algorithms*, wherein the main role is to train (teach) the computer system (machine) so that it can perform such task well. In the process of designing

[5]In machine learning, such quantity is called the *feature*. Usually this refers to a key component that well describes characteristics of data.

algorithms, we use something that is *data*.

Why called "Machine Learning"?

One can easily see the rationale of the naming via changing a viewpoint. From a *machine's perspective*, one can say that a *machine learns* the task from data. Hence, it is called *machine learning*, ML for short. This naming was originated in 1959 by Arthur Lee Samuel. See Fig. 2.

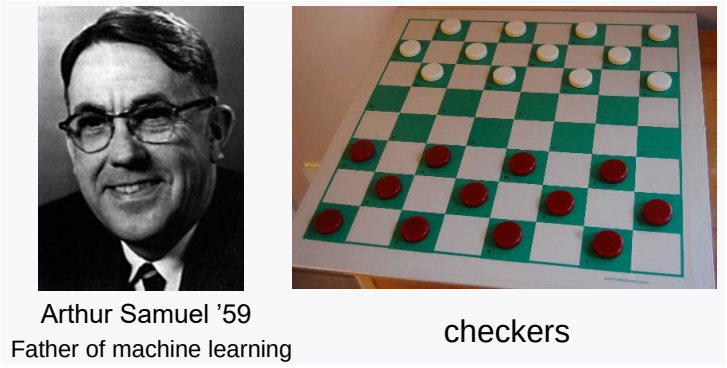


Figure 2: Arthur Lee Samuel is an American pioneer in the field of artificial intelligence, known mostly as the Father of machine learning. He found the field in the process of developing computer checkers which later formed the basis of AlphaGo.

Arthur Samuel is actually one of the pioneers in the broader field of *Artificial Intelligence* (AI) which includes machine learning as a sub-field. The AI field is the study of creating *intelligence* by *machines*, unlike the *natural intelligence* displayed by intelligent beings like humans and animals. Since the ML field is about building up a human-like machine, it is definitely a sub-field of AI.

In fact, Arthur Samuel found the ML field in the process of developing a human-like computer player for a board game, called *checkers*; see the right figure in Fig. 1. He proposed many algorithms and ideas while developing computer checkers. It turns out those algorithms could form the basis of *AlphaGo*, a computer program for the board game Go which defeated one of the 9-dan professional players, Lee Sedol with 4 wins out of 5 games in 2016.

Mission of machine learning

Since ML is a sub-field of AI, its end-mission is *achieving artificial intelligence*. So in view of the block diagram in Fig. 1, the goal of ML is to design an algorithm so that the trained machine *behaves like intelligence beings*.

Supervised learning

There are some methodologies which help us to achieve the goal of ML. One specific yet very popular method is the one called:

Supervised Learning.

What supervised learning means is *learning* a function $f(\cdot)$ (indicating a functional of the machine) with the help of a *supervisor*. See Fig. 3.

What the supervisor means in this context is the one who provides input-output samples. Obviously the input-output samples form the *data* employed for training the machine, usually

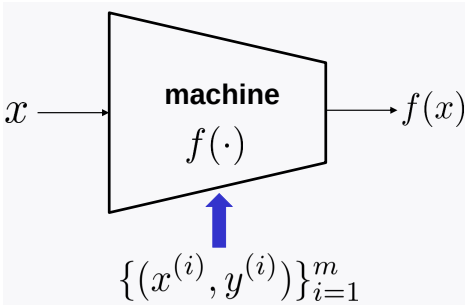


Figure 3: Supervised Learning: Design a computer system (machine) $f(\cdot)$ with the help of a supervisor which offers input-output pair samples, called a training dataset $\{(x^{(i)}, y^{(i)})\}_{i=1}^{m}$.

denoted by:

$$\{(x^{(i)}, y^{(i)})\}_{i=1}^{m}, \tag{1}$$

where $(x^{(i)}, y^{(i)})$ indicates the ith input-output sample (or called a *training sample* or an *example*) and m denotes the number of samples. Using this notation (1), one can say that supervised learning is to:

$$\text{Estimate } f(\cdot) \text{ using the training samples } \{(x^{(i)}, y^{(i)})\}_{i=1}^{m}. \tag{2}$$

Optimization

A common way to estimate $f(\cdot)$ is through *optimization*. To understand what this means, let us first explain what optimization is.

Optimization is to choose an *optimization variable* that minimizes (or maximizes) a certain quantity of interest possibly given some constraints. There are two important quantities in the definition. One is the optimization variable which affects the interested quantity and is subject to our design. This is usually a multi-dimensional quantity. The second is the certain quantity of interest that we wish to minimize (or maximize). This is called the *objective function* in the field, and an one-dimensional scalar.

Objective function

Now what is the *objective function* in the supervised learning framework? To figure this out, we need to know about the *objective* that supervised learning wishes to achieve. In view of the goal (2), what we want is obviously:

$$y^{(i)} \approx f(x^{(i)}), \quad \forall i \in \{1, \ldots, m\}.$$

A natural question that arises is then: How to quantify *closeness* (reflected in the "$\approx$" notation) between the two quantities: $y^{(i)}$ and $f(x^{(i)})$? One very common way that has been used in the field is to employ a function, called a *loss* function, usually denoted by:

$$\ell(y^{(i)}, f(x^{(i)})). \tag{3}$$

One obvious property that the loss function $\ell(\cdot, \cdot)$ should have is that it should be small when the two arguments are close, while being zero when the two are identical. Using such loss

function (3), one can then formulate an optimization problem as:

$$\min_{f(\cdot)} \sum_{i=1}^{m} \ell(y^{(i)}, f(x^{(i)})). \tag{4}$$

How to introduce optimization variable?

Now what is *optimization variable*? Unfortunately, there is no variable. Instead we have a different quantity that we can optimize over: *the function* $f(\cdot)$, marked in red in (4). The question is then: How to introduce optimization variable? A common way employed in the field is to represent the function $f(\cdot)$ with *parameters (or called weights)*, denoted by w, and then consider such weights as an optimization variable. Taking this approach, one can then translate the problem (4) into:

$$\min_{w} \sum_{i=1}^{m} \ell(y^{(i)}, f_w(x^{(i)})) \tag{5}$$

where $f_w(x^{(i)})$ denotes the function $f(x^{(i)})$ parameterized by w.

The above optimization problem depends on how we define the two functions: (i) $f_w(x^{(i)})$ w.r.t. w; (ii) the loss function $\ell(\cdot, \cdot)$. In machine learning, lots of works have been done for the choice of such functions.

A choice for $f_w(\cdot)$

Around at the same time when the machine learning (ML) field was founded, one architecture was suggested for the function $f_w(\cdot)$ in the context of simple binary classifiers in which y takes one among the two options only, e.g., $y \in \{-1, +1\}$ or $y \in \{0, 1\}$. The architecture is called:

$$\textit{Perceptron},$$

and was invented in 1957 by one of the pioneers in the AI field, named Frank Rosenblatt. Interestingly, Frank Rosenblatt was a *psychologist*. He was interested in how brains of intelligent beings work and his study on brains led him to come up with the perceptron, and therefore gave significant insights into *neural networks* that many of you guys heard of.

How brains work

Here are details on how the brain structure inspired the perceptron architecture. Inside a brain, there are many *electrically excitable cells*, called *neurons*; see Fig. 4. Here a red-circled one indicates a neuron. So the figure shows three neurons in total. There are three major properties about neurons that led to the architecture.

The first is that a neuron is an *electrical* quantity, so it has a *voltage*. The second property is that neurons are connected with each other through mediums, called *synapses*. So the main role of synapses is to deliver electrical voltage signals across neurons. Depending on the connectivity strength level of a synapse, a voltage signal from one neuron to another can increase or decrease. The last is that a neuron takes a particular action, called *activation*. Depending on its voltage level, it generates an all-or-nothing pulse signal. For instance, if its voltage level is above a certain threshold, then it generates an impulse signal with a certain magnitude, say 1; otherwise, it produces nothing.

Perceptron

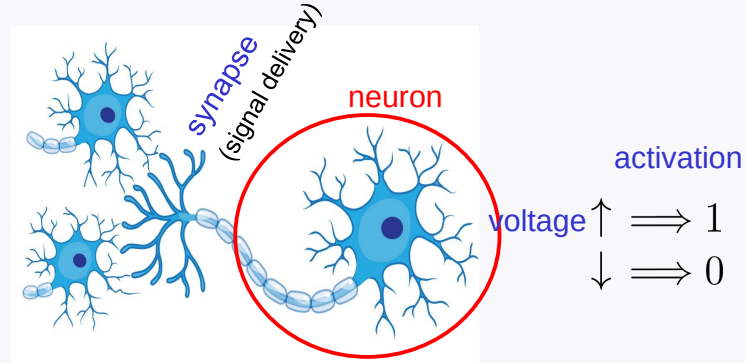


Figure 4: Neurons are electrically excitable cells and are connected through synapses.

The above three properties led Frank Rosenblatt to propose the perceptron architecture, as illustrated in Fig. 5.

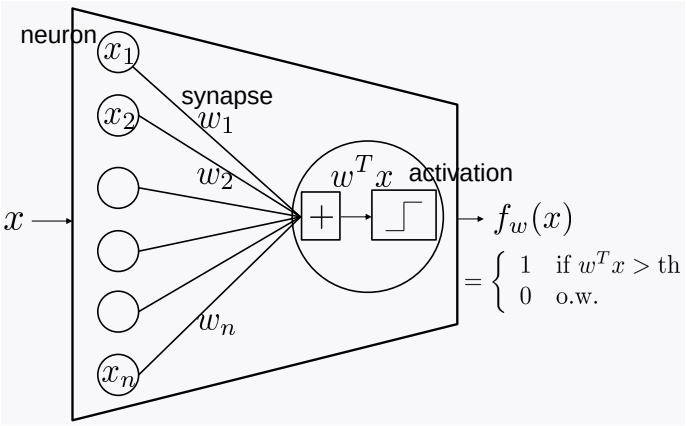


Figure 5: The architecture of perceptron.

Let x be an n-dimensional real-valued signal: $x := [x_1, x_2, \ldots, x_n]^T$. Suppose each component x_i is distributed to each neuron, and let x_i indicates a voltage level of the ith neuron. The voltage signal x_i is then delivered through a synapse to another neuron (placed on the right in the figure, indicated by a big circle). Remember that the voltage level can increase or decrease depending on the connectivity strength of a synapse. To capture this, a weight, say w_i, is multiplied to x_i so $w_i x_i$ is a delivered voltage signal at the terminal neuron. Based on an empirical observation that the voltage level at the terminal neuron increases with more connected neurons, Rosenblatt introduced an adder which simply aggregates all the voltage signals coming from many neurons, so he modeled the voltage signal at the terminal neuron as:

$$w_1 x_1 + w_2 x_2 + \cdots + w_n x_n = w^T x. \tag{6}$$

Lastly in an effort to mimic the *activation*, he modeled the output signal as

$$f_w(x) = \begin{cases} 1 & \text{if } w^T x > \text{th}, \\ 0 & \text{o.w.} \end{cases} \tag{7}$$

where "th" indicates a certain threshold level. It can also be simply denoted as

$$f_w(x) = \mathbf{1}\{w^T x > \text{th}\}. \tag{8}$$

Activation functions

Taking the percentron architecture in Fig. 5, one can then formulate the optimization problem (5) as:

$$\min_{w} \sum_{i=1}^{m} \ell(y^{(i)}, \mathbf{1}\{w^T x^{(i)} > \text{th}\}). \tag{9}$$

This is an initial optimization problem that people came up with. However, people figured out there is an issue in solving this optimization. The issue comes from the fact that the objective function contains an indicator function, so it is *not differentiable*. It turns out non-differentiability makes the problem difficult to solve. This will be clearer soon in a later lecture. Please be patient until we get to the point. What can we do then? One typical way that people have taken in the field is to *approximate* the activation function. There are many ways for approximation. From below, we will investigate one of them.

Approximate the activation!

One popular way is to use the following function that makes a *smooth* transition from 0 to 1:

$$f_w(x) = \frac{1}{1 + e^{-w^T x}}. \tag{10}$$

Notice that $f_w(x) \approx 0$ when $w^T x$ is very small; it then grows exponentially with an increase in $w^T x$; later grows logarithmically; and finally saturates as 1 when $w^T x$ is very large. Actually the function (10) is a very popular one used in statistics, called the *logistic*[6] function. There is another name for the function, which is the *sigmoid*[7] function.

There are two good things about the logistic function. First it is differentiable. The second is that it can serve as the *probability* for the output in the binary classifier, e.g., $\Pr(y = 1)$ where y denotes the ground-truth label in the binary classifier. So it is interpretable.

Look ahead

Under the choice of the logistic activation, what is then a good choice for a *loss* function? It turns out the entropy plays an important role in the design of an *optimal* loss function in some sense. Next time we will investigate in what sense it is optimal. We will then figure out how the entropy comes up in the design of the optimal loss function.

[6]The word *logistic* comes from a Greek word which means a slow growth, like a logarithmic growth.
[7]Sigmoid means resembling the lower-case Greek letter sigma, S-shaped.

November 4, 2019
Changho Suh (chsuh@kaist.ac.kr)

Lecture 16: Logistic regression

Recap

Last time we formulated an optimization problem for supervised learning based on the percetron architecture:

$$\min_{w} \sum_{i=1}^{m} \ell(y^{(i)}, f_w(x^{(i)})). \tag{1}$$

As an activation function, we considered a logistic function that is widely used in the field:

$$f_w(x) = \frac{1}{1+e^{-w^T x}}. \tag{2}$$

I then mentioned that a concept related to entropy appears in the design of the *optimal* loss function.

Today's lecture

Today we will prove this claim. Specifically what we are going to do are three folds. We will first investigate what it means by the *optimal* loss function. We will next figure out how entropy comes up in the design of the optimal loss function. Lastly we will also learn how to solve the optimization problem.

Optimality in a sense of maximizing likelihood

A binary classifier with the logistic function (2) is called *logistic regression*. See Fig. 1 for illustration.

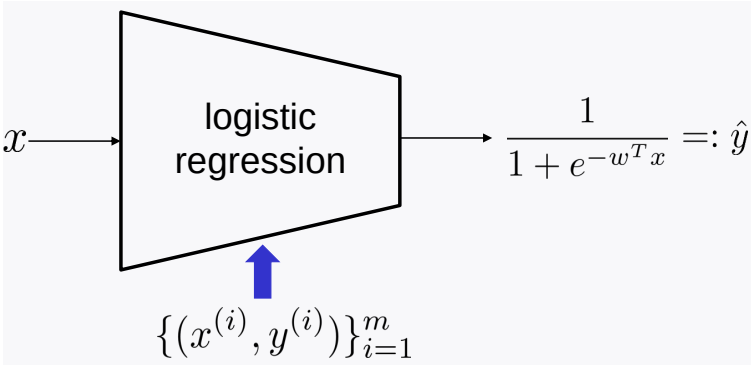


Figure 1: Logistic regression

Notice that the range of the output $\hat{y}$ of the logistic regression is in between 0 and 1:

$$0 \leq \hat{y} \leq 1.$$

Hence, one can interpret this as a *probability* quantity. Now the optimality of a classifier can be defined under the following assumption inspired by such interpretation:

$$\text{Assumption}: \hat{y} = \Pr(y = 1|x). \tag{3}$$

To understand what it means in detail, consider the *likelihood* of the ground-truth classifier:

$$\Pr\left(\{y^{(i)}\}_{i=1}^m \mid \{x^{(i)}\}_{i=1}^m\right). \tag{4}$$

Notice that the classifier output $\hat{y}$ is a function of weights w. Hence, we see that assuming (3), the likelihood (4) is also a function of w.

We are now ready to define the optimal w. The optimal weight, say w^*, is defined as the one that *maximizes the likelihood* (4):

$$w^* := \arg\max_w \Pr\left(\{y^{(i)}\}_{i=1}^m | \{x^{(i)}\}_{i=1}^m\right). \tag{5}$$

Accordingly the *optimal loss function*, say $\ell^*(\cdot,\cdot)$ is defined as the one that yields:

$$\arg\min_w \sum_{i=1}^m \ell^*\left(y^{(i)}, \hat{y}^{(i)}\right) = \arg\max_w \Pr\left(\{y^{(i)}\}_{i=1}^m | \{x^{(i)}\}_{i=1}^m\right). \tag{6}$$

As I mentioned earlier, an entropy-related concept appears in $\ell^*(\cdot,\cdot)$. Let us now figure this out.

Finding the optimal loss function $\ell^*(\cdot,\cdot)$

Usually samples are obtained from different data $x^{(i)}$'s. Hence, it is reasonable to assume that such samples are independent with each other:

$$\{(x^{(i)}, y^{(i)})\}_{i=1}^m \text{ are independent over } i. \tag{7}$$

Under this assumption, we can then rewrite the likelihood (4) as:

$$\begin{aligned}
\Pr\left(\{y^{(i)}\}_{i=1}^m | \{x^{(i)}\}_{i=1}^m\right) &\overset{(a)}{=} \frac{\Pr\left(\{(x^{(i)}, y^{(i)})\}_{i=1}^m\right)}{\Pr\left(\{x^{(i)}\}_{i=1}^m\right)} \\
&\overset{(b)}{=} \frac{\prod_{i=1}^m \mathbb{P}\left(x^{(i)}, y^{(i)}\right)}{\prod_{i=1}^m \mathbb{P}(x^{(i)})} \\
&\overset{(c)}{=} \prod_{i=1}^m \mathbb{P}\left(y^{(i)}|x^{(i)}\right)
\end{aligned} \tag{8}$$

where (a) and (c) are due to the definition of conditional probability; and (b) comes from the independence assumption (7). Here $\mathbb{P}(x^{(i)}, y^{(i)})$ denotes the probability distribution of the input-output pair of the system:

$$\mathbb{P}(x^{(i)}, y^{(i)}) := \Pr(X = x^{(i)}, Y = y^{(i)}) \tag{9}$$

where X and Y indicate random variables of the input and the output, respectively.

Recall the probability-interpretation-related assumption (3) made with regard to $\hat{y}$:

$$\hat{y} = \Pr(y = 1|x).$$

This implies that:

$$\begin{aligned} y = 1: &\quad \mathbb{P}(y|x) = \hat{y}; \\ y = 0: &\quad \mathbb{P}(y|x) = 1 - \hat{y}. \end{aligned}$$

Hence, one can represent $\mathbb{P}(y|x)$ as:

$$\mathbb{P}(y|x) = \hat{y}^y (1 - \hat{y})^{1-y}.$$

Now using the notations of $(x^{(i)}, y^{(i)})$ and $\hat{y}^{(i)}$, we then get:

$$\mathbb{P}\left(y^{(i)}|x^{(i)}\right) = (\hat{y}^{(i)})^{y^{(i)}} (1 - \hat{y}^{(i)})^{1-y^{(i)}}.$$

Plugging this into (8), we get:

$$\begin{aligned} &\Pr\left(\{y^{(i)}\}_{i=1}^m | \{x^{(i)}\}_{i=1}^m\right) \\ &= \prod_{i=1}^m \mathbb{P}(x^{(i)}) \prod_{i=1}^m (\hat{y}^{(i)})^{y^{(i)}} (1 - \hat{y}^{(i)})^{1-y^{(i)}}. \end{aligned} \tag{10}$$

This together with (5) yields:

$$\begin{aligned} w^* &:= \arg\max_w \prod_{i=1}^m (\hat{y}^{(i)})^{y^{(i)}} (1 - \hat{y}^{(i)})^{1-y^{(i)}} \\ &\overset{(a)}{=} \arg\max_w \sum_{i=1}^m y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}) \\ &\overset{(b)}{=} \arg\min_w \sum_{i=1}^m -y^{(i)} \log \hat{y}^{(i)} - (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}) \end{aligned} \tag{11}$$

where (a) comes from the fact that $\log(\cdot)$ is a non-decreasing function and $\prod_{i=1}^m (\hat{y}^{(i)})^{y^{(i)}} (1 - \hat{y}^{(i)})^{1-y^{(i)}}$ is positive; and (b) is due to changing a sign of the objective while replacing max with min.

In fact, the term inside the summation in the last equality in (11) respects the formula of another key notion in information theory: *cross entropy*. In particular, in the context of a loss function, it is named the cross entropy loss:

$$\ell_{\mathsf{CE}}(y, \hat{y}) := -y \log \hat{y} - (1 - y) \log(1 - \hat{y}). \tag{12}$$

Hence, the optimal loss function that yields the maximum likelihood solution is the cross entropy loss:

$$\ell^*(\cdot, \cdot) = \ell_{\mathsf{CE}}(\cdot, \cdot).$$

Remarks on cross-entropy loss (12)

Let me say a few words about why the loss function (12) is called the *cross-entropy* loss. Actually this comes from the definition of *cross entropy*. The cross entropy is defined w.r.t. two random variables. For simplicity, let us consider two binary random variables, say $X \sim \mathsf{Bern}(p)$ and

$Y \sim \mathsf{Bern}(q)$ where $X \sim \mathsf{Bern}(p)$ indicates a binary random variable with $p = \Pr(X = 1)$. For such two random variables, the cross entropy is defined as:

$$H(p, q) := -p \log q - (1 - p) \log(1 - q). \tag{13}$$

Notice that the formula of (12) is exactly the same as the term inside summation in (11), except for having different notations. Hence, it is called the *cross entropy loss*.

Then, you may now wonder why $H(p, q)$ in (13) is called *cross entropy*. Of course, there is a rationale. The rationale comes from the following fact that you were asked to prove in PS5:

$$H(p, q) \geq H(p) := -p \log p - (1 - p) \log(1 - p) \tag{14}$$

where $H(p)$ denotes the entropy of $\mathsf{Bern}(p)$. Also you were asked to verify that the equality holds when $p = q$. So from this, one can interpret $H(p, q)$ as an *entropic*-measure of discrepancy a*cross* distributions. Hence, it is called *cross entropy*.

How to solve (11)?

In view of (11), the optimization problem for logistic regression can be written as:

$$\min_w \sum_{i=1}^{m} -y^{(i)} \log \frac{1}{1 + e^{-w^T x^{(i)}}} - (1 - y^{(i)}) \log \frac{e^{-w^T x^{(i)}}}{1 + e^{-w^T x^{(i)}}}. \tag{15}$$

Let $J(w)$ be the normalized version of the objective function:

$$J(w) := \frac{1}{m} \sum_{i=1}^{m} -y^{(i)} \log \hat{y}^{(i)} - (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}). \tag{16}$$

It turns out the above optimization belongs to *convex optimization* that we are familiar with. In other words, $J(w)$ is convex in optimization variable w. So for the rest of this lecture, we will first prove the convexity of the objective function, and then discuss how to solve such convex optimization.

Proof of convexity

One can readily show that convexity preserves under addition (why? think about the definition of convex functions). So it suffices to prove the following two:

$$\text{(i)} - \log \frac{1}{1 + e^{-w^T x}} \text{ is convex in } w;$$
$$\text{(ii)} - \log \frac{e^{-w^T x}}{1 + e^{-w^T x}} \text{ is convex in } w.$$

Since the second function in the above can be represented as the sum of a linear function and the first function:

$$- \log \frac{e^{-w^T x}}{1 + e^{-w^T x}} = w^T x - \log \frac{1}{1 + e^{-w^T x}},$$

it suffices to prove the convexity of the first function.

Notice that the first function can be rewritten as:

$$- \log \frac{1}{1 + e^{-w^T x}} = \log(1 + e^{-w^T x}). \tag{17}$$

In fact, proving the convexity of the function (17) is a bit involved if one relies directly on the definition of convex functions. It turns out there is another way to prove. That is based on the computation of the second derivative of a function, called the Hessian. How to compute the Hessian? What is the dimension of the Hessian? For a function $f : \mathbf{R}^d \to \mathbf{R}$, the gradient $\nabla f(x) \in \mathbf{R}^d$ and the Hessian $\nabla^2 f(x) \in \mathbf{R}^{d\times d}$. If you are not familiar, check from the vector calculus course or from Wikipedia.

A well-known fact says that if the Hessian of a function is positive semi-definite (PSD)[1], then the function is convex. We will not prove this here. But if you are interested, you will have a chance to prove it in PS6. If this is too much, you may want to simply remember this fact only. No need to prove, but the statement itself is very instrumental. Here we will use this fact to prove the convexity of the function (17).

Taking a derivative of the RHS formula in (17) w.r.t. w, we get:

$$\nabla_w \log(1 + e^{-w^T x}) = \frac{-xe^{-w^T x}}{1 + e^{-w^T x}}.$$

This is due to a chain rule of derivatives and the fact that $\frac{d}{dz} \log z = \frac{1}{z}$, $\frac{d}{dz} e^z = e^z$ and $\frac{d}{dw} w^T x = x$. Taking another derivative of the above, we obtain a Hessian as follows:

$$\begin{aligned}
\nabla_w^2 \log(1 + e^{-w^T x}) &= \nabla_w \left(\frac{-xe^{-w^T x}}{1 + e^{-w^T x}} \right) \\
&\overset{(a)}{=} \frac{xx^T e^{-w^T x}(1 + e^{-w^T x}) - xx^T e^{-w^T x} e^{-w^T x}}{(1 + e^{-w^T x})^2} \\
&= \frac{xx^T e^{-w^T x}}{(1 + e^{-w^T x})^2} \\
&\succeq 0
\end{aligned} \tag{18}$$

where (a) is due to the derivative rule of a quotient of two functions: $\frac{d}{dz} \frac{f(z)}{g(z)} = \frac{f'(z)g(z) - f(z)g'(z)}{g^2(z)}$. Here you may wonder why $\frac{d}{dw}(-xe^{-w^T x}) = xx^T e^{-w^T x}$. Why not xx, $x^T x^T$ or $x^T x$ in front of $e^{-w^T x}$? One rule-of-thumb that I strongly recommend is to simply try all the candidates and choose the one which does not have a syntax error (matrix dimension mismatch). For instance, xx (or $x^T x^T$) is just an invalid operation. $x^T x$ is not a right one because the Hessian must be an d-by-d matrix. The only candidate left without any syntax error is xx^T! We see that xx^T has the single eigenvalue of $\|x\|^2$ (Why?). Since the eigenvalue $\|x\|^2$ is non-negative, the Hessian is PSD, and therefore we prove the convexity.

Gradient decent algorithm

Now how to solve the convex optimization problem (11)? Since there is no constraint in the optimization, w^* must be the *stationary* point, i.e., the one such that

$$\nabla J(w^*) = 0. \tag{19}$$

But there is an issue in deriving such optimal point w^* from the above. The issue is that analytically finding such point is not doable because it turns out there is no closed-form solution (check!). However, there is a good news. The good news is that there developed several algorithms which allow us to find such point efficiently without the knowledge of the closed-form

[1]We say that a *symmetric* matrix, say $Q = Q^T \in \mathbf{R}^{d\times d}$, is positive semi-definite if $v^T Q v \geq 0$, $\forall v \in \mathbf{R}^d$, i.e., all the eigenvalues of Q are non-negative. It is simply denoted by $Q \succeq 0$.

solution. One prominent algorithm that has been widely used in the field is: the *gradient decent algorithm*.

Here is how the algorithm works. It is an *iterative* algorithm. Suppose that at the t-th iteration, we have an estimate of w^*, say $w^{(t)}$. We then compute the gradient of the function evaluated at the estimate: $\nabla J(w^{(t)})$. Next we update the estimate along a direction being *opposite* to the direction of the gradient:

$$w^{(t+1)} \longleftarrow w^{(t)} - \alpha \nabla J(w^{(t)}) \tag{20}$$

where $\alpha > 0$ indicates a stepsize (or called a learning rate). If you think about it, this update rule makes sense. Suppose $w^{(t)}$ is placed right relative to the optimal point w^*, as illustrated in Fig. 2.

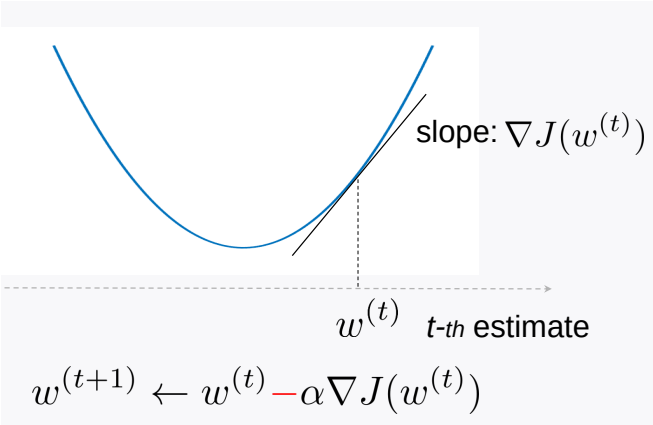


Figure 2: Gradient decent algorithm.

Then, we should move $w^{(t)}$ to the *left* so that it is closer to w^*. The update rule actually does this, as we *subtract* by $\alpha \nabla J(w^{(t)})$. Notice that $\nabla J(w^{(t)})$ points to the *right* direction given that $w^{(t)}$ is placed right relative to w^*. We repeat this procedure until it converges. It turns out: as $t \to \infty$, it actually converges:

$$w^{(t)} \longrightarrow w^*, \tag{21}$$

as long as the stepsize is chosen properly, like the one delaying exponentially w.r.t. t. We will not touch upon the proof of this convergence. Actually the proof is not that simple - even there is a big field in statistics which intends to prove the convergence of a variety of algorithms (if it is the case).

Look ahead

So far we have formulated an optimization problem for supervised learning, and found that *cross entropy* serves as a key component in the design of the optimal loss function. We also learned how to solve the problem via a famous algorithm, called gradient decent. As mentioned earlier, there would be programming implementation of such algorithm. Next time, we will study such implementation details in the context of a simple classifier via Pytorch.

November 6, 2019
Changho Suh (chsuh@kaist.ac.kr)

Lecture 17: Algorithm implementation

Recap

We have thus far formulated an optimization problem for supervised learning:

$$\min_w \sum_{i=1}^m \ell_{\mathsf{CE}}\left(y^{(i)}, \frac{1}{1+e^{-w^T x^{(i)}}}\right) \tag{1}$$

where

$$\ell_{\mathsf{CE}}(y, \hat{y}) = -y \log \hat{y} - (1-y)\log(1-\hat{y}), \tag{2}$$

$$\hat{y} := \frac{1}{1+e^{-w^T x}}. \tag{3}$$

We proved that the cross entropy loss $\ell_{\mathsf{CE}}(\cdot,\cdot)$ is the optimal loss function in a sense of maximizing likelihood. We also showed that the normalized version $J(w)$ of the above objection function is convex in w, and therefore, it can be solved via one prominent algorithm that enables us to efficiently find the unique stationary point: *gradient decent*.

$$J(w) := \frac{1}{m}\sum_{i=1}^m -y^{(i)} \log \hat{y}^{(i)} - (1-y^{(i)})\log(1-\hat{y}^{(i)}). \tag{4}$$

Today's lecture

Today we will study how to implement the algorithm in the context of a simple classifier. Specifically we will first investigate what that simple classifier setting of our focus is. We will then study four implementation details w.r.t. such classifier: (i) dataset that we will use for training and testing[1]; (ii) a deep neural network[2] model & ReLU[3] activation; (iii) Softmax: a natural extension of a logistic activation for multiple (more than two) classes; (iv) Adam: An advanced version of gradient decent algorithm that is widely used in practice. Lastly we will learn how to do programming for the classifier via one deep learning framework: Pytorch.

Handwritten digit classification

The simple classifier that we will focus on for implementation exercise is a handwritten digit classifier wherein the task is to figure out a digit from a handwritten image; see Fig. 1. The figure illustrates an instance in which a image of digit 2 is correctly recognized.

For training a model, we employ a popular dataset, named the MNIST (Modified National Institute of Standards and Technology)[4] dataset. It contains $m = 60,000$ training images and

[1] In the machine learning field, *testing* refers to performance evaluation of a trained machine learning model. For this purpose, we often employ an *unseen* dataset, called *test dataset*. Here *unseen* means never being employed during training.

[2] So far we have studied the single layer neural network. By convention, the input layer is not counted, so it is called a single layer network instead of a two-layer one. We say that a neural network is deep if it has at least one hidden layer (two layers).

[3] Yes, it is the one that you encountered in PS5, defined as $\mathsf{ReLU}(x) = \max(0, x)$.

[4] It was created by re-mixing the examples from NIST's origional dataset. Hence, the naming was suggested. It was prepared by one of the deep learning heroes, named Yann LeCun.

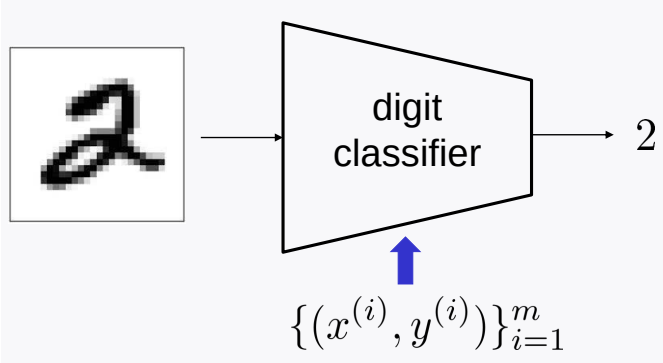


Figure 1: Handwritten digit classification.

$m_{\text{test}} = 10,000$ testing images. Each image, say $x^{(i)}$, consists of 28×28 pixels, each indicating a gray-scale level ranging from 0 (white) to 1 (black). It also comes with a corresponding label, say $y^{(i)}$, that takes one of the 10 classes $y^{(i)} \in \{0, 1, \ldots, 9\}$. See Fig. 2.

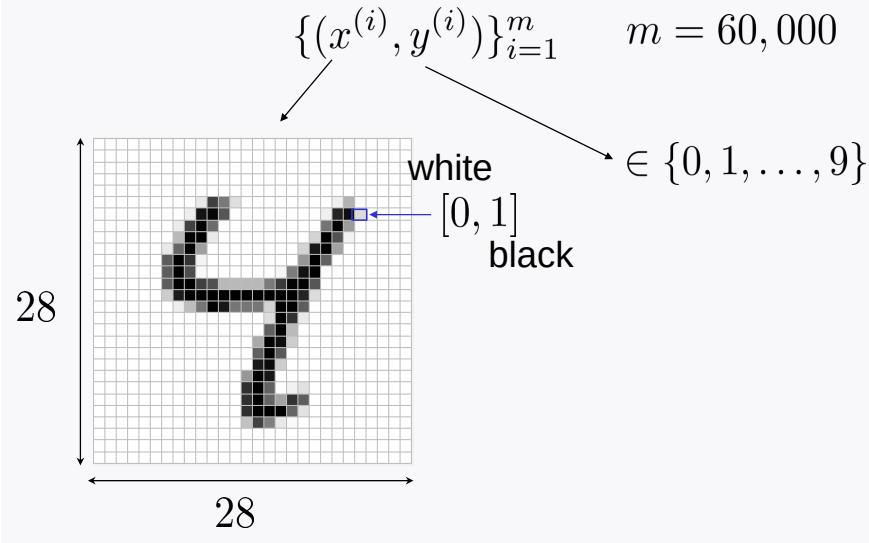


Figure 2: MNIST dataset: An input image is of 28-by-28 pixels, each indicating an intensity from 0 (whilte) to 1 (black); each label with size 1 takes one of the 10 classes from 0 to 9.

A deep neural network model

As a model, we employ an advanced version of logistic regression that we have learned so far. There are two reasons for the use of the advanced version.

One is that logistic regression based on the perceptron architecture is a *linear* classifier, meaning that $f_w(\cdot)$ is restricted to a linear function class. So one may guess that the performance of the logistic regression based on the restricted architecture may not be that good in many applications. It turns out this is the case. This motivated people to develop a perceptron-like yet more advanced neural network which consists of *many* layers, called a deep neural network (DNN). While it has been invented in 1960s, the performance benefit of DNNs started to be greatly appreciated only during the past decade due to a big event happened in 2012. The big event was the winning of ImageNet recognition competition by Geoffrey Hinton (a

very well-known figure in the AI field, known as the Godfather of deep learning) and his two PhD students. What was astonishing at the time is that their DNN can achieve *human-level* recognition performances, which were never attained earlier. Hence, this anchored the start of deep learning revolution. It turns out digit classification of our interest here is one such application where a linear classifier does not perform well. So we will employ a DNN yet a very simple version with only two layers: one-hidden layer and output layer, illustrated in Fig. 3.

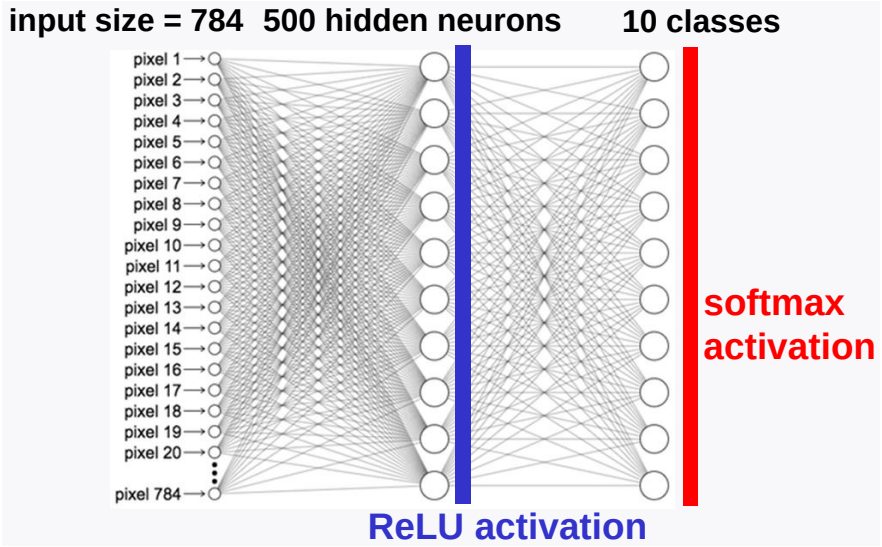


Figure 3: A two-layer fully-connected neural network where input size is $28 \times 28 = 784$, the number of hidden neurons is 500 and the number of classes is 10. We employ ReLU activation for the hidden layer, and softmax activation for the output layer; see Fig. 4 for details.

Each neuron in the hidden layer respects exactly the same procedure as that in the perceptron: a linear operation followed by activation. For activation, the logistic function or its shifted version, called the tanh function (spanning -1 and +1), were frequently employed in the early days. However, many practitioners recently found that another function, named *Rectified Linear Unit* (ReLU for short), is more powerful in enabling faster training while yielding a better or equal performance. As mentioned earlier, the functional of ReLU is: $\mathsf{ReLU}(x) = \max(0, x)$. Hence, a common rule-of-thumb that have been applied in the deep learning field is to use ReLU activation in all hidden layers. So we adopt the rule-of-thumb, as illustrated in Fig. 3.

Softmax activation for output layer

The second reason that we employ an advanced version of logistic regression is w.r.t. the number of classes in our interested classifier. Logistic regression is intended for the *binary* classifier. However, in our digit classifier, there are 10 classes in total. So logistic regression is not directly applicable. One natural extension of logistic regression for a general classifier with more than two classes is to use a generalized version, called softmax. See Fig. 4.

Let z be the output of the last layer in a neural network prior to activation:

$$z := [z_1, z_2, \ldots, z_c]^T \in \mathbf{R}^c \tag{5}$$

where c denotes the number of classes. The softmax function is then defined as:

$$\hat{y}_j := [\mathsf{softmax}(z)]_j = \frac{e^{z_j}}{\sum_{k=1}^{c} e^{z_k}} \quad j = 1, 2, \ldots, c. \tag{6}$$

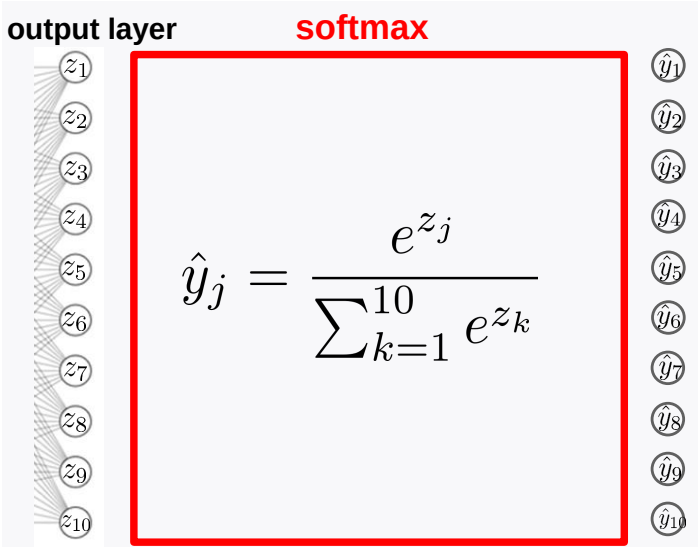


Figure 4: Softmax activation for output layer. This is a natural extension of logistic activation intended for the 2-class case.

Note that this is a natural extension of a logistic function: for $c = 2$,

$$\begin{aligned} \hat{y}_1 := [\mathsf{softmax}(z)]_1 &= \frac{e^{z_1}}{e^{z_1} + e^{z_2}} \\ &= \frac{1}{1 + e^{-(z_1 - z_2)}} \\ &= \sigma(z_1 - z_2) \end{aligned} \tag{7}$$

where $\sigma(\cdot)$ is a logistic function. Viewing $z_1 - z_2$ as the binary classifier output $\hat{y}$, this coincides exactly with the logistic regression.

Here $\hat{y}_i$ can be interpreted as the probability that the ith example belong to the class i. Hence, like the binary classifier, one may want to assume:

$$\hat{y}_i = \Pr(y = [0, \ldots, \underbrace{1}_{i\text{th position}}, \ldots, 0]^T | x), \; i = 1, \ldots, c. \tag{8}$$

As you may expect, under this assumption, one can verify that the optimal loss function (in a sense of maximizing likelihood) is again the cross entropy loss:

$$\ell^*(y, \hat{y}) = \ell_{\mathsf{CE}}(y, \hat{y}) = \sum_{j=1}^{c} -y_j \log \hat{y}_j$$

where y indicates a label of one-hot vector type. For instance, in the case of label= 2 with

$c = 10$, y takes:

$$y = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{matrix}$$

The proof of this is almost the same as that in the binary classifier. So we will omit the proof here. Instead you will have a chance to prove it in PS6.

Due to the above rationales, the softmax activation has been widely used for many classifiers in the field. Hence, we will also use the conventional function in our digit classifier.

Adam optimizer

Let us discuss a specific algorithm that we will employ in our setting. As mentioned earlier, we will use an advanced version of gradient decent algorithm, called Adam optimizer. To see what the optimizer operates, let us first recall the vanilla gradient decent:

$$w^{(t+1)} \leftarrow w^{(t)} - \alpha \nabla J(w^{(t)})$$

where $w^{(t)}$ indicates the estimated weight in the tth iteration, and α denotes the learning rate. Notice that the weight update relies only on the *current* gradient, reflected in $\nabla J(w^{(t)})$. Hence, in case $\nabla J(w^{(t)})$ fluctuates too much over iterations, the weight update oscillates significantly, thereby bringing about unstable training. To address this, people often use a variant algorithm that exploits *past* gradients for the purpose of stabilization. That is, Adam optimizer.

Here is how the Adam works. The weight update takes the following formula instead:

$$w^{(t+1)} = w^{(t)} + \alpha \frac{m^{(t)}}{\sqrt{s^{(t)} + \epsilon}} \tag{9}$$

where $m^{(t)}$ indicates a weighted average of the current and past gradients:

$$m^{(t)} = \frac{1}{1 - \beta_1^t} \left(\beta_1 m^{(t-1)} - (1 - \beta_1) \nabla J(w^{(t)}) \right). \tag{10}$$

Here $\beta_1 \in [0, 1]$ is a hyperparameter that captures the weight of past gradients, and hence it is called the *momentum*. So the notation m stands for momentum. The factor $\frac{1}{1-\beta_1^t}$ is applied in front, in an effort to stabilize training in initial iterations (small t). Check the detailed rationale behind this in PS6.

$s^{(t)}$ is a normalization factor that makes the effect of $\nabla J(w^{(t)})$ almost constant over t. In case $\nabla J(w^{(t)})$ is too big or too small, we may have significantly different scalings in magnitude. Similar to $m^{(t)}$, $s^{(t)}$ is defined as a weighted average of the current and past values:

$$s^{(t)} = \frac{1}{1 - \beta_2^t} \left(\beta_2 s^{(t-1)} - (1 - \beta_2) (\nabla J(w^{(t)}))^2 \right) \tag{11}$$

where $\beta_2 \in [0, 1]$ denotes another hyperparameter that captures the weight of past values, and s stands for *square*.

Notice that the dimensions of $w^{(t)}$, $m^{(t)}$ and $s^{(t)}$ are the same. So all the operations that appear in the above (including division in (9) and square in (11)) are *component-wise*. In (9), ϵ is a tiny value introduced to avoid division by 0 in practice (usually 10^{-8}). Here (β_1, β_2) is exactly the `betas` that you encountered in PS5.

Pytorch: MNIST data loading

Now let us study how to do Pytorch programming for implementing the simple digit classifier that we have discussed so far. First, MNIST data loading. MNIST is a very famous dataset, so it is offered by many libraries including `torchvision.datasets`. So in order to download the dataset into your machine, you may want to use the following command:

```
train_dataset = torchvision.datasets.MNIST(root='../datasets',
                                           train = True,
                                           transform = transforms.ToTensor(),
                                           download = True)
```

where `'../datasets'` indicates the directory that the MNIST dataset would be downloaded (`'../'` means the directory prior to the current direction that you are working on), and `transforms.ToTensor()` denotes conversion from raw images into tensor data that we will play with.

During training, we often employ a part of the entire examples to compute a gradient of a loss function. Such part is called the *batch*. Hence, the dataset is usually reorganized in the form of batches. To this end, you can use the following:

```
train_loader = torchvision.utils.data.DataLoader(dataset= train_dataset,
                                                 batch_size = batch_size,
                                                 shuffle = True)
```

In the MNIST dataset, the size of `train_dataset` should be $m = 60,000$ and the size of `train_loader` should read $\frac{m}{\texttt{batch\_size}}$.

Pytorch: 2-layer neural network

The simple two-layer neural network model, illustrated in Fig. 3, can be implemented by the following code:

```
import torch.nn as nn

class Net(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(Net, self).__init__
        self.fc1 = nn.Linear(input_size,hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size,num_classes)

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x

my_model = Net(input_size,hidden_size,num_classes).to(device)
```

where `device` denotes `cuda` if available; `cpu` otherwise. See the pytorch tutorial for details. Here `nn.Linear(in, out)` is a built-in class implemented in `torch.nn`. The main functional is a linear operation where the associated matrix is of size `out`-by-`in` and a bias vector is of size `out`. For instance, the command "`x = self.fc1(x)`" means the following:

$$\texttt{x} \leftarrow \texttt{W x + b}$$

where $\texttt{W} \in \mathbf{R}^{\texttt{hidden\_size} \times \texttt{input\_size}}$ and $\texttt{b} \in \mathbf{R}^{\texttt{hidden\_size}}$. `nn.ReLU()` is another built-in class which contains the following functional: $\max(0, \texttt{input argument})$. For example, the command "`x = self.relu(x)`" means: $\texttt{x} \leftarrow \max(0, \texttt{x})$.

Pytorch: Cross entropy loss

The cross entropy loss that we learned is also implemented in `torch.nn`. The corresponding built-in class is:

```
nn.CrossEntropyLoss()
```

To use this class for our classifier, you may want to consult with the following commands:

```
criterion = nn.CrossEntropyLoss()
loss = criterion(output,label)
```

where `output` denotes an output of `my_model` defined earlier, and `label` indicates the ground-truth class of an associated image $\in \{0, 1, \ldots, 9\}$. Here one important to notice is that the `output` is the result *prior to* softmax activation.

Pytorch: Adam optimizer

The Adam optimizer is also implemented in `torch.optim` via the following built-in class:

```
torch.optim.Adam()
```

This is how to use in our classifier:

```
my_model = Net(input_size,hidden_size,num_classes).to(device)
optimizer = torch.optim.Adam(my_model.parameters(),lr=learning_rate,betas=betas)
```

Remember that Adam optimizer includes three important parameters: `learning rate`, `b1` and `b2`. While these are hyperparameters that we can choose as per our wish, a default choice is: `lr:=learning rate=0.001`, `betas:=(b1,b2)=(0.9,0.999)`.

Pytorch: Training

For training, we should first enable the "train mode" via:

```
my_model.train()
```

To initialize the gradient of `optimizer`, we use:

```
optimizer.zero_grad()
```

To compute a gradient of a loss function, we use:

```
loss = criterion(output,label)
loss.backward()
```

To update the weight as per (9), we use:

```
optimizer.step()
```

Pytorch: Testing

For testing, we should transit to "test mode" via:

```
my_model.eval()
```

One important distinction relative to training is that testing does not involve any *gradient computation*. Hence, once we are under the "test mode" (through the above command), all the operations should be programmed within the following command:

```
with torch.no_grad():
```

To make a prediction from `output` of `my_model`, we use:

```
_, predicted = torch.max(output.data, 1)
```

To counter the number of correct predictions, we use:

```
correct += (predicted == label).sum().item()
```

Look ahead

Now we are done with the first application that we would deal with in Part III. Next time, we will move onto the second application of information theory: *unsupervised learning*.

November 11, 2019
Changho Suh (chsuh@kaist.ac.kr)

Lecture 18: Unsupervised learning: Generative models

Recap

During the past three lectures, we have studied some basic and trending contents on supervised learning. The goal of supervised learning is to estimate a functional $f(\cdot)$ of an interested computer system (machine) from input-output samples, as illustrated in Fig. 1.

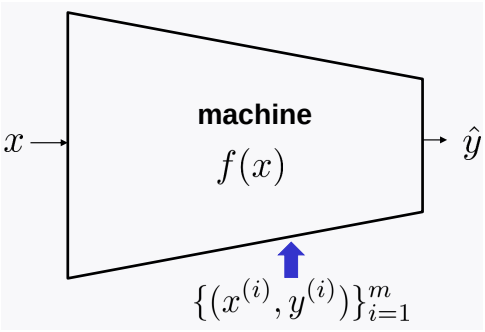


Figure 1: Supervised learning: Learning the functional $f(\cdot)$ of an interested system from data $\{(x^{(i)}, y^{(i)})\}_{i=1}^m$.

In an effort to translate a *function* optimization problem (a natural formulation of supervised learning) into a parameter-based optimization problem that we are familiar with, we expressed the function with parameters (or called weights) assuming a certain architecture of the system.

The certain architecture was: *Perceptron*. Taking the logistic function, we obtained logistic regression. We then proved that cross entropy is the optimal loss function in a sense of maximizing the likelihood of the ground-truth system w.r.t. training data. We also considered the Deep Neural Network (DNN) architecture for $f(\cdot)$, which has been shown to be more expressive. Since there is no theoretical basis on the choice of activation functions in the DNN context, we investigated only a rule-of-thumb which is common to use in the field: Taking ReLU at all hidden neurons while taking the logistic (or softmax) function at the output layer.

With regard to how to solve the optimization, we explored one popular algorithm that enables us to solve the problem numerically: *gradient decent*. We have also studied a more complicated yet powerful version of gradient decent, Adam optimizer, which exploits past gradients to yield much stable training. Lastly we studied how to do programming for implementing such algorithm via Pytorch.

Unsupervised learning

Now what is next? Actually we face one critical challenge that arises in supervised learning. The challenge is that it is not that easy to collect *labeled* data in many realistic situations. In general, gathering labeled data is very expensive, as it usually requires extensive human-labour-based annotations. So people wish to do something without such labeled data. Then, a natural question that arises is: What can we do *only* with $\{x^{(i)}\}_{i=1}^m$?

This is where the concept of *unsupervised learning* kicks in. Unsupervised learning is a methodology for learning something about the data $\{x^{(i)}\}_{i=1}^m$. You may then ask: What is *something*?

There are a few candidates for such something to learn in the field. One very popular candidate is *probability distribution*. In fact, this is a sort of the most complex yet most fundamental information. The probability distribution allows us to create realistic signals as we wish. The unsupervised learning method for learning such fundamental entity is: *generative models*. This is actually the most famous unsupervised learning method, which has received a particularly significant attention in the field nowadays. So in this course, we are going to focus on this method.

Generative Adversarial Networks (GANs)

Among many generative models in the literature, we will focus on one very popular generative model: *Generative Adversarial Networks (GANs)*. The GANs were invented by one of youngest heroes in the history of the AI field, named Ian Goodfellow; see Fig. 2. He was a bachelor and

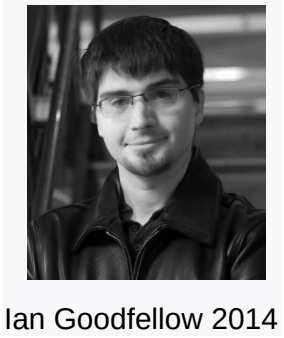


Figure 2: Ian Goodfellow, a young yet big figure in the modern AI field. He is best known as the inventor of the Generative Adversarial Networks (GANs), which made a big wave in the history of the AI field.

master student at Stanford, working with Prof. Andrew Ng. But he moved to the University of Montreal for PhD study to join Joshua Bengio's group. During his PhD, he could develop the *"GANs"*. The GANs are shown to be extremely instrumental in a wide variety of applications, even not limited to the AI field. Such applications include: image creation, human image synthesis, image inpainting, coloring, super-resolution image synthesis, speech synthesis, style transfer, robot navigation, to name a few. Since it works pretty well, as of Oct. 3 2019, the state of California even passed a bill that would ban the use of GANs to make fake pornography without the consent of the people depicted.

In view of information theory, the GAN is also an interesting framework. It turns out the GAN is closely related to the KL divergence and mutual information that we have learned thus far.

During upcoming lectures ...

During upcoming lectures (possibly three lectures including this), we will investigate such connection. Specifically what we are going to do are four-folded. First we will study what generative models are in a mathematical framework. We will then formulate a corresponding optimization problem; in particular will put an emphasis on one particular framework, which led to the GAN. Next we will make a connection to the KL divergence and mutual information. Lastly we will learn how to solve the GAN optimization and implement it via Pytorch. Today we will focus on the first two.

Generative models

A generative model is defined as a mathematical model which allows us to generate *fake* data which has a *similar distribution* as that of *real* data. See Fig. 3 for a pictorial representation. The model parameters are learned via real data so that the learned model outputs fake data

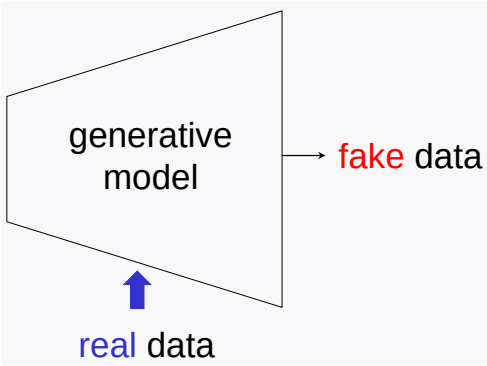


Figure 3: A generative model is the one that generates *fake* data which resembles *real* data. Here what *resembling* means in a mathematical language is that it has a *similar distribution*.

that resemble real data. Here an input signal (which should be placed in front of the model yet not appeared in the figure) can be either an *arbitrary random* signal or a specifically synthesized signal that forms the skeleton of fake data. The type of the input depends on applications of interest - this will be detailed later on.

Remarks on generative models

In fact, the problem of designing a generative model is one of the most important problems in *statistics*, so it has been a classical age-old problem in that field. This is because the major goal of the field of statistics is to figure out (or estimate) the probability distribution of data that arise in the real world (that we call real data), and the generative model plays a role as a underlying framework in achieving the goal. Actually the model can do even more - it provides a concrete function block (called the *generator* in the field) which can create realistic fake data. There is a very popular name in statistics that indicates such problem, that is the *density estimation problem*. Here the density refers to the probability distribution.

Actually this problem was not that popular in the AI field until very recently, precisely 2014 when the GANs were invented.

How to formulate an optimization problem?

Now let us relate generative models to optimization. As mentioned earlier, we can feed some input signal (that we call *fake input* or *latent signal*) which one can arbitrarily synthesize. Common ways employed in the field to generate them are to use Gaussian or uniform distributions. Another way will be explored in PS7. Since it is an *input* signal, we may wish to use a conventional "x" notation. So let us use $x \in \mathbf{R}^k$ to denote a fake input where k indicates a dimension of the latent signal.

Notice that this has a conflict with real data notation $\{x^{(i)}\}_{i=1}^m$. To avoid the conflict, let us use a different notation, say $\{y^{(i)}\}_{i=1}^m$, to denote real data. Please don't be confused with labeled data - these are not labels any more. In fact, the convention in the machine learning field is to use a notation z to indicate the latent signal while maintaining real data notation as $\{x^{(i)}\}_{i=1}^m$. This may be another way to go; perhaps this is the way that you should take especially when writing papers. Anyhow let us take the first unorthodox yet reasonable option for this course.

Let $\hat{y} \in \mathbf{R}^n$ be a fake output. Considering m examples, let $\{(x^{(i)}, \hat{y}^{(i)})\}_{i=1}^m$ be such fake input-output m pairs and let $\{y^{(i)}\}_{i=1}^m$ be m real data examples. See Fig. 4.

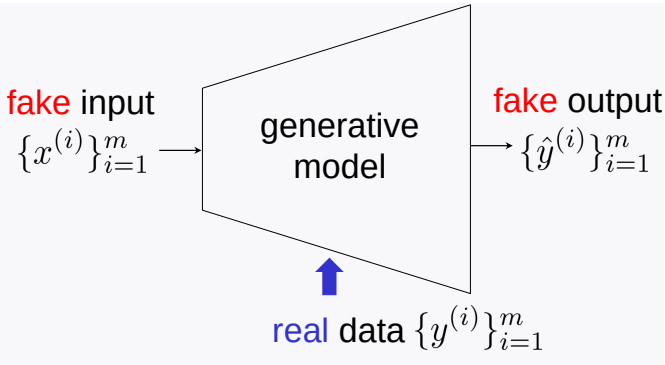


Figure 4: Problem formulation for generative models.

Goal

Let $G(\cdot)$ be a function of the generative model. Then, the goal of the generative model can be stated as follows: Designing $G(\cdot)$ such that

$$\{\hat{y}^{(i)}\}_{i=1}^m \approx \{y^{(i)}\}_{i=1}^m \textit{ in distribution.}$$

Here what does it mean by "in distribution"? To make it clear, we need to quantify closeness between two distributions. One natural yet prominent approach employed in the statistics field is to take the following two steps:

1. Compute empirical distributions or estimate distributions from $\{y^{(i)}\}_{i=1}^m$ and $\{(x^{(i)}, \hat{y}^{(i)})\}_{i=1}^m$. Let such distributions be:

   $$\mathbb{Q}_Y, \mathbb{Q}_{\hat{Y}}$$

   for real and fake data, respectively.

2. Next employ a well-known *divergence measure* in statistics which can serve to quantify the closeness of two distributions. Let $\mathsf{D}(\cdot, \cdot)$ be one such divergence measure. Then, the similarity between $\mathbb{Q}_Y$ and $\mathbb{Q}_{\hat{Y}}$ can be quantified as:

   $$\mathsf{D}(\mathbb{Q}_Y, \mathbb{Q}_{\hat{Y}}).$$

Taking the above natural approach, one can concretely state the goal as: Designing $G(\cdot)$ such that

$$\mathsf{D}(\mathbb{Q}_Y, \mathbb{Q}_{\hat{Y}}) \text{ is minimized.}$$

Optimization under the approach

Hence, under the approach, one can formulate an optimization problem as:

$$\min_{G(\cdot)} \mathsf{D}(\mathbb{Q}_Y, \mathbb{Q}_{\hat{Y}}). \tag{1}$$

As you may easily notice, there are some issues in solving the above problem (1). One issue is that it is a *function* optimization problem which we are not familiar with. As mentioned earlier, one natural way to resolve this issue is to parameterize the function with a neural network:

$$\min_{G(\cdot)\in\mathcal{N}} \mathsf{D}(\mathbb{Q}_Y, \mathbb{Q}_{\hat{Y}}). \tag{2}$$

where $\mathcal{N}$ indicates a class of neural network functions.

There are two more yet non-trivial issues. The first is that the objective function $\mathsf{D}(\mathbb{Q}_Y, \mathbb{Q}_{\hat{Y}})$ is a very complicated function of the knob $G(\cdot)$. Note that $\mathbb{Q}_{\hat{Y}}$ is a function of $G(\cdot)$, as $\hat{y} = G(x)$. So the objective is a twice folded composite function of $G(\cdot)$. The second is perhaps the most fundamental issue. It is not clear as to how to choose a divergence measure $D(\cdot, \cdot)$.

Look ahead

It turns out there are some ways to address the above issues. Very interestingly, one such way leads to an optimization problem for GANs! So next time, we will study what that way is, and then will take the way to derive an optimization problem for GANs.

November 13, 2019
Changho Suh (chsuh@kaist.ac.kr)

Lecture 19: Generative Adversarial Networks (GANs)

Recap

Last time we started investigating *unsupervised learning*. The goal of unsupervised learning is to learn something about the data, which we newly denoted by $\{y^{(i)}\}_{i=1}^m$, instead of $\{x^{(i)}\}_{i=1}^m$. Depending on target candidates for something to learn, there are a few unsupervised learning methods. Among them, we explored one very prominent method which aims to learn the *probability distribution*. That is, *generative models*. We then formulated an optimization problem for generative models:

$$\min_{G(\cdot)\in\mathcal{N}} \mathsf{D}(\mathbb{Q}_Y, \mathbb{Q}_{\hat{Y}}), \tag{1}$$

where $\mathbb{Q}_Y$ and $\mathbb{Q}_{\hat{Y}}$ indicate the empirical distributions (or the estimates of the true distributions) for real and fake data, respectively; $G(\cdot)$ denotes the function of a generative model; $\mathsf{D}(\cdot,\cdot)$ is a divergence measure; and $\mathcal{N}$ is a class of neural network functions. We then mentioned two major issues that arise in the problem: (i) the objective is a very complicated function of $G(\cdot)$; (ii) it is not clear as to how to choose $\mathsf{D}(\cdot,\cdot)$.

At the end of the last lecture, I claimed that there are some ways to address such issues, and very interestingly, one such way leads to an optimization problem for a recently-developed powerful generative model, named Generative Adversarial Networks (GANs).

Today's lecture

Today we are going to explore details on the GANs. Specifically what we are going to do are three-folded. First we will investigate what that way leading to the GANs is. We will then take the way to derive an optimization problem for the GANs. Lastly we will demonstrate that the GANs indeed have close connection to the KL divergence and mutual information.

What is the way that led to GANs?

Remember one challenge that we faced in the optimization problem (1): $\mathsf{D}(\mathbb{Q}_Y, \mathbb{Q}_{\hat{Y}})$ is a complicated function of $G(\cdot)$. To address this, Ian Goodfellow, the inventor of GANs, took an *indirect way* to represent $\mathsf{D}(\mathbb{Q}_Y, \mathbb{Q}_{\hat{Y}})$. He was thinking how $\mathsf{D}(\mathbb{Q}_Y, \mathbb{Q}_{\hat{Y}})$ should behave, and his insightful observation (that we will discuss soon) led him to come up with an *indirect way* of mimicking the behaviour. It turned out the way enabled him to explicitly compute $\mathsf{D}(\mathbb{Q}_Y, \mathbb{Q}_{\hat{Y}})$. Below are details.

How $\mathsf{D}(\mathbb{Q}_Y, \mathbb{Q}_{\hat{Y}})$ should behave?

First of all, Goodfellow imagined how $\mathsf{D}(\mathbb{Q}_Y, \mathbb{Q}_{\hat{Y}})$ should behave. What he imagined is that if one can *easily discriminate* real data y from fake data $\hat{y}$, then the divergence must be *large*; otherwise, it should be small. This naturally motivated him to:

> Interpret $\mathsf{D}(\mathbb{Q}_Y, \mathbb{Q}_{\hat{Y}})$ as a measure that can quantify *the ability to discriminate*.

In order to express such ability, he believed there must be something that plays a role of dis-

criminating. So he introduced an entity that can play such role, and named it:

$$\textit{Discriminator}.$$

Goodfellow considered a simple *binary-output* discriminator which takes as an input, either real data y or fake data $\hat{y}$. He then wanted to design $D(\cdot)$ such that $D(\cdot)$ well *approximates* the probability that the input $(\cdot)$ is *real* data:

$$D(\cdot) \approx \Pr((\cdot) = \text{real data}).$$

Noticing that

$$\Pr(y = \text{real}) = 1;$$
$$\Pr(\hat{y} = \text{real}) = 0,$$

he wanted to design $D(\cdot)$ such that:

$$D(y) \text{ is as large as possible, close to 1;}$$
$$D(\hat{y}) \text{ is as small as possible, close to 0.}$$

See Fig. 1.

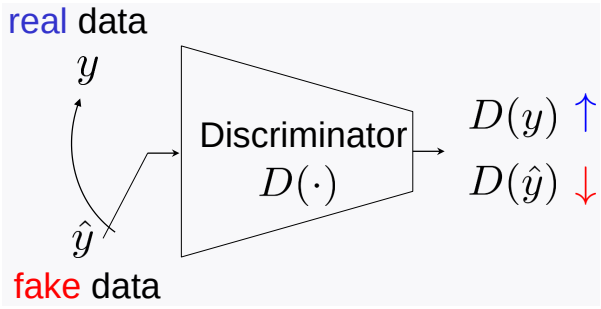


Figure 1: Discriminator wishes to output $D(\cdot)$ such that $D(y)$ is as large as possible while $D(\hat{y})$ is as small as possible.

How to quantity the ability to discriminate?

Keeping the picture in Fig. 1 in his mind, he then wanted to quantify the ability to discriminate. To this end, he first observed that if $D(\cdot)$ can easily discriminate, then we should have:

$$D(y) \uparrow; \;\; 1 - D(\hat{y}) \uparrow.$$

One naive way to capture the ability is simply *adding* the above two terms. But Goodfellow did not take the naive way. Instead he took the following *logarithmic* summation:

$$\log D(y) + \log(1 - D(\hat{y})). \tag{2}$$

Actually I was wondering why he took this particular way, as his paper does not mention about the rationale behind the choice. In NIPS 2016, he gave a tutorial on GANs, mentioning that the problem formulation was inspired by a paper published in AISTATS 2010:

$$\textit{http://proceedings.mlr.press/v9/gutmann10a.html}$$

Reading the paper, I realized that the logarithmic summation (2) comes from Eq. (3) in the paper. Actually this is also related to the *binary cross entropy*. Think about why.

Making the particular choice, the ability to discriminate for m examples can be quantified as:

$$\frac{1}{m}\sum_{i=1}^{m}\log D(y^{(i)}) + \log(1 - D(\hat{y}^{(i)})). \tag{3}$$

A two-player game

Goodfellow then imagined a *two-player game* in which Player 1, Discriminator $D(\cdot)$, wishes to maximize the quantified ability (3), while Player 2, Generator $G(\cdot)$, wants to minimize (3). See Fig. 2.

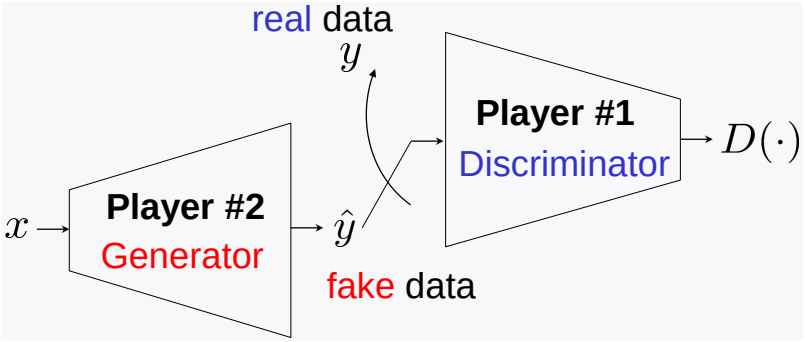


Figure 2: A two-player game.

Optimization for GANs

This naturally motivated him to formulate the following min max optimization problem:

$$\min_{G(\cdot)}\max_{D(\cdot)}\frac{1}{m}\sum_{i=1}^{m}\log D(y^{(i)}) + \log(1 - D(\hat{y}^{(i)})). \tag{4}$$

You may wonder why not max min (meaning that we first take "min" and then "max"). That may be another way to go, but Goodfellow made the above choice. Actually there is a reason why he took the way. This will be clearer soon. Here notice that the optimization is over the two *functions* of $D(\cdot)$ and $G(\cdot)$, meaning that it is still a *function* optimization problem. Luckily the year of 2014 (when the GAN paper was published) was after the starting point of the deep learning revolution, the year of 2012. Also at that time, Goodfellow was a PhD student of Joshua Bengio, one of the deep learning heroes. So he was very much aware of the power of neural networks: "Deep neural networks can well represent any arbitrary function." This motivated him to parameterize the two functions with neural networks, which in turn led to the following optimization problem:

$$\min_{G(\cdot)\in\mathcal{N}}\max_{D(\cdot)\in\mathcal{N}}\frac{1}{m}\sum_{i=1}^{m}\log D(y^{(i)}) + \log(1 - D(\hat{y}^{(i)})), \tag{5}$$

where $\mathcal{N}$ denotes a class of neural network functions. This is exactly the optimization problem for GANs!

Related to original optimization?

Remember what I mentioned earlier: the way leading to the GAN optimization is an indirect way of solving the original optimization problem:

$$\min_{G(\cdot)} \mathsf{D}(\mathbb{Q}_Y, \mathbb{Q}_{\hat{Y}}). \tag{6}$$

Then, a natural question that arises is: How are the two problems (5) and (6) related? It turns out these are very much related. This is exactly where the choice of min max (instead of max min) plays the role; the other choice cannot establish a connection - it was a smart choice. It has been shown that assuming that deep neural networks can represent any arbitrary function, the GAN optimization (5) can be translated into the original optimization form (6). We will prove this below.

Simplification & manipulation

Let us start by simplifying the GAN optimization (5). Since we assume that $\mathcal{N}$ can represent any arbitrary function, the problem (5) becomes *unconstrained*:

$$\min_{G(\cdot)} \max_{D(\cdot)} \frac{1}{m} \sum_{i=1}^{m} \log D(y^{(i)}) + \log(1 - D(\hat{y}^{(i)})). \tag{7}$$

Notice that the objective is a function of $D(\cdot)$, and the two functions $D(\cdot)$'s appear but with *different* arguments: one is $y^{(i)}$, colored in blue; the other is $\hat{y}^{(i)}$, colored in red. So in the current form (7), the inner (max) optimization problem is not quite tractable to solve. In an attempt to make it tractable, let us express it in a different manner using the following notations.

Define a random vector Y which takes one of the m real examples with probability $\frac{1}{m}$ (uniform distribution):

$$Y \in \{y^{(1)}, \dots, y^{(m)}\} =: \mathcal{Y}; \;\; \mathbb{Q}_Y(y^{(i)}) = \frac{1}{m}, \;\; i = 1, 2, \dots, m,$$

where $\mathbb{Q}_Y$ indicates the probability distribution of Y. Similarly define $\hat{Y}$ for fake examples:

$$\hat{Y} \in \{\hat{y}^{(1)}, \dots, \hat{y}^{(m)}\} =: \hat{\mathcal{Y}}; \;\; \mathbb{Q}_{\hat{Y}}(\hat{y}^{(i)}) = \frac{1}{m}, \;\; i = 1, 2, \dots, m,$$

where $\mathbb{Q}_{\hat{Y}}$ indicates the probability distribution of $\hat{Y}$. Using these notations, one can then rewrite the problem (7) as:

$$\min_{G(\cdot)} \max_{D(\cdot)} \sum_{i=1}^{m} \mathbb{Q}_Y(y^{(i)}) \log D(y^{(i)}) + \mathbb{Q}_{\hat{Y}}(\hat{y}^{(i)}) \log(1 - D(\hat{y}^{(i)})). \tag{8}$$

Still we have different arguments in the two $D(\cdot)$ functions.

To address this, let us introduce another quantity. Let $z \in \mathcal{Y} \cup \hat{\mathcal{Y}}$. Newly define $\mathbb{Q}_Y(\cdot)$ and $\mathbb{Q}_{\hat{Y}}(\cdot)$ such that:

$$\mathbb{Q}_Y(z) := 0 \text{ if } z \in \hat{\mathcal{Y}} \setminus \mathcal{Y}; \tag{9}$$

$$\mathbb{Q}_{\hat{Y}}(z) := 0 \text{ if } z \in \mathcal{Y} \setminus \hat{\mathcal{Y}}. \tag{10}$$

Using the z notation, one can then rewrite the problem (8) as:

$$\min_{G(\cdot)} \max_{D(\cdot)} \sum_{z \in \mathcal{Y} \cup \hat{\mathcal{Y}}} \mathbb{Q}_Y(z) \log D(z) + \mathbb{Q}_{\hat{Y}}(z) \log(1 - D(z)). \tag{11}$$

We now see that the same arguments appear in the two $D(\cdot)$ functions.

Solving the inner optimization problem

We are ready to solve the inner optimization problem in (11). Key observations are: $\log D(z)$ is concave in $D(\cdot)$; $\log(1 - D(z))$ is concave in $D(\cdot)$; and therefore, the objective function is concave in $D(\cdot)$. This implies that the objective has the *unique maximum* in the function space $D(\cdot)$. Hence, one can find such maximum by searching for the one in which the derivative is zero. Taking a derivative and setting it to zero, we get:

$$\text{Derivative} = \sum_z \left[\frac{\mathbb{Q}_Y(z)}{D^*(z)} - \frac{\mathbb{Q}_{\hat{Y}}(z)}{1 - D^*(z)} \right] = 0.$$

Hence, we get:

$$D^*(z) = \frac{\mathbb{Q}_Y(z)}{\mathbb{Q}_Y(z) + \mathbb{Q}_{\hat{Y}}(z)}. \tag{12}$$

Plugging this into (11), we obtain:

$$\min_{G(\cdot)} \sum_{z \in \mathcal{Y} \cup \hat{\mathcal{Y}}} \mathbb{Q}_Y(z) \log \frac{\mathbb{Q}_Y(z)}{\mathbb{Q}_Y(z) + \mathbb{Q}_{\hat{Y}}(z)} + \mathbb{Q}_{\hat{Y}}(z) \log \frac{\mathbb{Q}_{\hat{Y}}(z)}{\mathbb{Q}_Y(z) + \mathbb{Q}_{\hat{Y}}(z)}. \tag{13}$$

Connection to KL divergence

Let us massage the objective function in (13) to express it as:

$$\min_{G(\cdot)} \underbrace{\sum_{z \in \mathcal{Y} \cup \hat{\mathcal{Y}}} \mathbb{Q}_Y(z) \log \frac{\mathbb{Q}_Y(z)}{\frac{\mathbb{Q}_Y(z) + \mathbb{Q}_{\hat{Y}}(z)}{2}} + \mathbb{Q}_{\hat{Y}}(z) \log \frac{\mathbb{Q}_{\hat{Y}}(z)}{\frac{\mathbb{Q}_Y(z) + \mathbb{Q}_{\hat{Y}}(z)}{2}}}_{} - 2 \log 2. \tag{14}$$

Notice that the above underbraced term can be expressed with a well-known divergence measure that we are familar with: the KL divergence. Hence, we get:

$$\min_{G(\cdot)} \sum_{z \in \mathcal{Y} \cup \hat{\mathcal{Y}}} \mathbb{Q}_Y(z) \log \frac{\mathbb{Q}_Y(z)}{\frac{\mathbb{Q}_Y(z) + \mathbb{Q}_{\hat{Y}}(z)}{2}} + \mathbb{Q}_{\hat{Y}}(z) \log \frac{\mathbb{Q}_{\hat{Y}}(z)}{\frac{\mathbb{Q}_Y(z) + \mathbb{Q}_{\hat{Y}}(z)}{2}} - 2 \log 2 \tag{15}$$

$$= \min_{G(\cdot)} \mathsf{KL}\left(\mathbb{Q}_Y \| (\mathbb{Q}_Y + \mathbb{Q}_{\hat{Y}})/2\right) + \mathsf{KL}\left(\mathbb{Q}_{\hat{Y}} \| (\mathbb{Q}_Y + \mathbb{Q}_{\hat{Y}})/2\right) - 2 \log 2. \tag{16}$$

Slightly manipulating the above, we obtain an equivalent optimization:

$$G^*_{\mathsf{GAN}} = \arg\min_{G(\cdot)} \frac{1}{2} \left\{ \mathsf{KL}\left(\mathbb{Q}_Y \| (\mathbb{Q}_Y + \mathbb{Q}_{\hat{Y}})/2\right) + \mathsf{KL}\left(\mathbb{Q}_{\hat{Y}} \| (\mathbb{Q}_Y + \mathbb{Q}_{\hat{Y}})/2\right) \right\}. \tag{17}$$

Also note that the objective coincides with Jensen-Shannon divergence that you encountered in PS2.

Connection to mutual information

Looking at (17), we can also make a connection to mutual information. To see this, recall what you were asked to prove in PS2 and midterm. That is, for two random variables, say T and Z,

$$I(T; Z) = \sum_{t \in \mathcal{T}} P_T(t) \mathsf{KL}(P_{Z|t} \| P_Z) \tag{18}$$

where P_T and P_Z denote the probability distributions of T and Z, respectively; and $P_{Z|t}$ indicates the conditional distribution of Z given $T = t$.

Now suppose that $T \sim \mathsf{Bern}\left(\frac{1}{2}\right)$ and we define Z as:

$$Z = \begin{cases} Y, & T = 0; \\ \hat{Y}, & T = 1. \end{cases}$$

Then, we get:

$$P_{Z|0} = \mathbb{Q}_Y,\ P_{Z|1} = \mathbb{Q}_{\hat{Y}},\ P_Z = \left(\mathbb{Q}_Y + \mathbb{Q}_{\hat{Y}}\right)/2$$

where the last is due to the total probability law (why?). This together with (17) and (18) gives:

$$G^*_{\mathsf{GAN}} = \arg\min_{G(\cdot)} I(T; Z). \tag{19}$$

Look ahead

So far we have formulated an optimization problem for GANs and made an interesting connection to the KL divergence and mutual information. Next time, we will study how to solve the GAN optimization (5) and implement it via Pytorch.

November 18, 2019
Changho Suh (chsuh@kaist.ac.kr)

Lecture 20: GAN implementation

Recap

Last time we investigated Goodfellow's approach to deal with an optimization problem for generative models, which in turn led to GANs. He started with quantifying the ability to discriminate real against fake samples:

$$\frac{1}{m}\sum_{i=1}^{m}\log D(y^{(i)})+\log(1-D(\hat{y}^{(i)})) \tag{1}$$

where $y^{(i)}$ and $\hat{y}^{(i)} := G(x^{(i)})$ indicate real and fake samples (the outputs of Generator), respectively; $D(\cdot)$ denotes the output of Discriminator; and m is the number of examples. He then introduced two players: (i) Player 1, Discriminator, who wishes to maximize the ability; (ii) Player 2, Generator, who wants to minimize it. This naturally led to the optimization problem for GANs:

$$\min_{G(\cdot)\in\mathcal{N}}\max_{D(\cdot)\in\mathcal{N}}\frac{1}{m}\sum_{i=1}^{m}\log D(y^{(i)})+\log(1-D(G(x^{(i)}))) \tag{2}$$

where $\mathcal{N}$ denotes a class of neural network functions. Lastly we demonstrated that the problem (2) can be stated in terms of the KL divergence or mutual information, thus making a connection to information theory.

Now two questions arise: (i) How to solve the problem (2)?; (ii) How to do programming, say via Pytorch?

Today's lecture

Today we will answer these two questions. Specifically we are going to cover four stuffs. First we will investigate a practical method to solve the problem (2). We will then do one case study for the purpose of exercising such method. In particular, we will study how to implement an MNIST-like image generator. Next we will study one important implementation detail: Batch Normalization, particularly useful for very deep neural networks. Lastly we will learn how to do programming via Pytorch.

Parameterization

Solving the problem (2) starts with parameterizing the two functions $G(\cdot)$ and $D(\cdot)$ with neural networks:

$$\min_{w}\max_{\theta}\underbrace{\frac{1}{m}\sum_{i=1}^{m}\log D_\theta(y^{(i)})+\log(1-D_\theta(G_w(x^{(i)})))}_{=:J(w,\theta)} \tag{3}$$

where w and θ indicate parameters for $G(\cdot)$ and $D(\cdot)$, respectively. Now the question of interest is: Is the parameterized problem (3) the one that we are familiar with? In other words, is $J(w,\theta)$ is *convex* in w? Is $J(w,\theta)$ is *concave* in θ? It turns out unfortunately, it is not the case. In general, the objective is highly non-convex in w and also highly non-concave in θ.

Then, what can we do? Actually there is nothing we can do more beyond what we know. We only know how to find a stationary point via a method like gradient decent. So one practical way to go is to simply look for a stationary point, say (w^*, θ^*), such that

$$\nabla_w J(w^*, \theta^*) = 0, \ \nabla_\theta J(w^*, \theta^*) = 0,$$

while cross-fingering that such point yields a near optimal performance. It turns out very luckily it is often the case in reality, especially when employing neural networks for parameterization. There have been huge efforts by many smart theorists in figuring out why that is the case. However, a clear theoretical understanding is still missing despite their serious efforts.

Alternating gradient decent

One practical method to attempt to find (yet not necessarily guarantee to find) such a stationary point in the context of the min-max optimization problem (3) is: *alternating gradient decent*. Here is how it works. At the tth iteration, we first update Generator's weight:

$$w^{(t+1)} \leftarrow w^{(t)} - \alpha_1 \nabla_w J(w^{(t)}, \theta^{(t)})$$

where $w^{(t)}$ and $\theta^{(t)}$ denote the weights of Generator and Discriminator at the tth iteration, respectively; and α_1 is the learning rate for Generator. Given $(w^{(t+1)}, \theta^{(t)})$, we next update Discriminator's weight as per:

$$\theta^{(t+1)} \leftarrow \theta^{(t)} + \alpha_2 \nabla_\theta J(w^{(t+1)}, \theta^{(t)})$$

where α_2 is the learning rate for Discriminator. Here one important thing to notice is that the direction along which the Discriminator's weight is updated should be *aligned* with the gradient, reflected in the plus sign colored in blue. Why? Lastly we repeat the above two until converged.

In practice, we may wish to control the frequency of Discriminator weight update relative to that of Generator. To this end, we often employ the $k:1$ alternating gradient decent:

1. Update Generator's weight:

$$w^{(t+1)} \leftarrow w^{(t)} - \alpha_1 \nabla_w J(w^{(t)}, \theta^{(t\cdot k)}).$$

2. Update Discriminator's weight k times while fixing $w^{(t+1)}$: `for i=1:k`,

$$\theta^{(t\cdot k+i)} \leftarrow \theta^{(t\cdot k+i-1)} + \alpha_2 \nabla_\theta J(w^{(t+1)}, \theta^{(t\cdot k+i-1)}).$$

3. Repeat the above.

You may wonder why we update Discriminator more frequently than Generator. Usually more updates in the *inner* optimization yield better performances in practice. Further, we often employ the Adam counterpart of the algorithm together with batches. Details are omitted although we will apply such practical version for programming assignment in PS7.

A practical tip on Generator

Before moving onto a case study for implementation, let me say a few words about Generator optimization. Given Discriminator's parameter θ: the Generator wishes to minimize:

$$\min_w \frac{1}{m_{\mathcal{B}}} \sum_{i \in \mathcal{B}} \log D_\theta(y^{(i)}) + \log(1 - D_\theta(G_w(x^{(i)})))$$

where $\mathcal{B}$ indicates a batch of interest and $m_{\mathcal{B}}$ is the batch size (the number of examples in the interested batch). Notice that $\log D_\theta(y^{(i)})$ in the above is irrelevant of Generator's weight w. Hence, it suffices to minimize the following:

$$\min_w \underbrace{\frac{1}{m_{\mathcal{B}}} \sum_{i \in \mathcal{B}} \log(1 - D_\theta(G_w(x^{(i)})))}_{\textit{generator loss}}$$

where the underbraced term is called "generator loss". However, in practice, instead of minimizing the generator loss directly, people often rely on the following *proxy*:

$$\min_w \frac{1}{m_{\mathcal{B}}} \sum_{i \in \mathcal{B}} -\log D_\theta(G_w(x^{(i)})).$$

You may wonder why. There is a technically detailed rationale behind the use of such proxy for the generator loss. Check this in PS7.

Task

Now let us discuss one case study for implementation. The task that we will focus on is the one related to the simple digit classifier that we exercised on in Lecture 17. The task is to generate MNIST-like images, as illustrated in Fig. 1. Here we intend to train Generator with MNIST

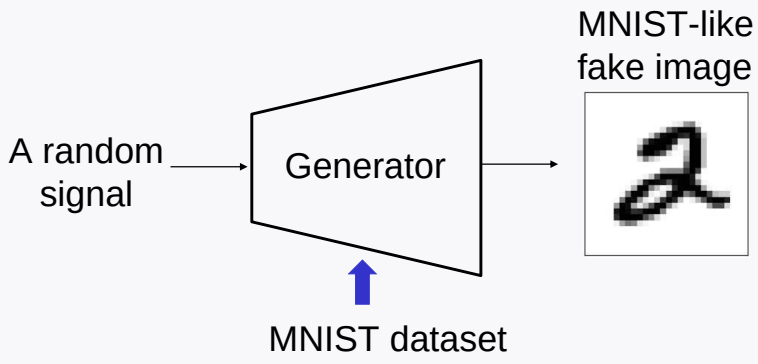


Figure 1: MNIST-like image generation.

dataset so that it outputs a MNIST-like fake image when fed by a random input signal.

Model for Generator

As a Generator model, we employ a 5-layer fully-connected (linear) neural network with four hidden layers, as depicted in Fig. 2. For activation at each hidden layer, we employ ReLU. Remember that an MNIST image consists of 28-by-28 pixels, each indicating a gray-scaled value that spans from 0 to 1. Hence, for the output layer, we use 784 ($= 28 \times 28$) neurons and logistic activation to ensure the range of $[0, 1]$.

Here the employed network has five layers, so it is deeper than the 2-layer case that we used earlier. In practice, for a somewhat deep neural network, each layer's signals can exhibit quite different scalings. It turns out such dynamically-swinged scaling yields a detrimental effect upon training: *unstable* training. So in practice, people often apply an additional procedure (prior to ReLU) so as to control such scaling in our own manner. The procedure is called: Batch

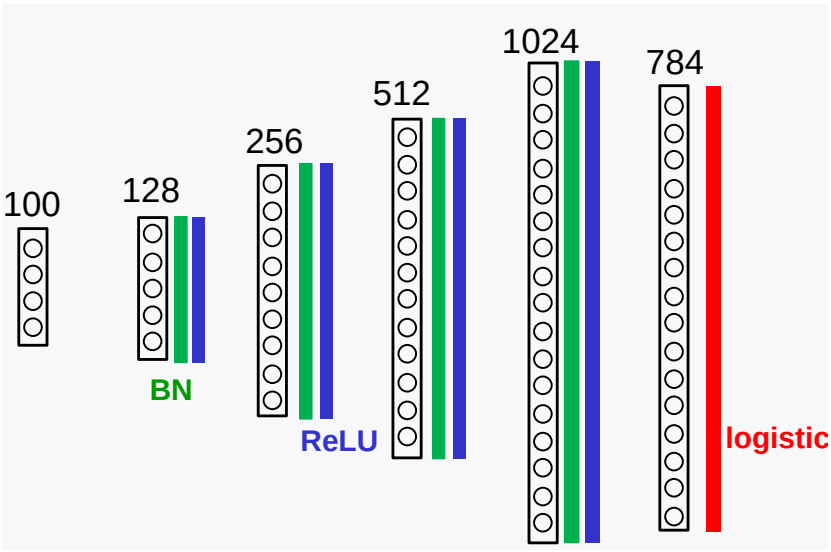


Figure 2: Generator: A 5-layer fully-connected neural network where the input size (the dimension of a latent signal) is 100; the numbers of hidden neurons are 128, 256, 512, 1024; and the output size is 784 ($= 28 \times 28$). We employ ReLU activation for every hidden layer, and logistic activation for the output layer to ensure 0-to-1 output signals. We also use Batch Normalization prior to ReLU at each hidden layer. See Fig. 3 for details.

Normalization.

Batch Normalization

Here is how it works; see Fig. 3. For illustrative purpose, focus on one particular hidden layer.

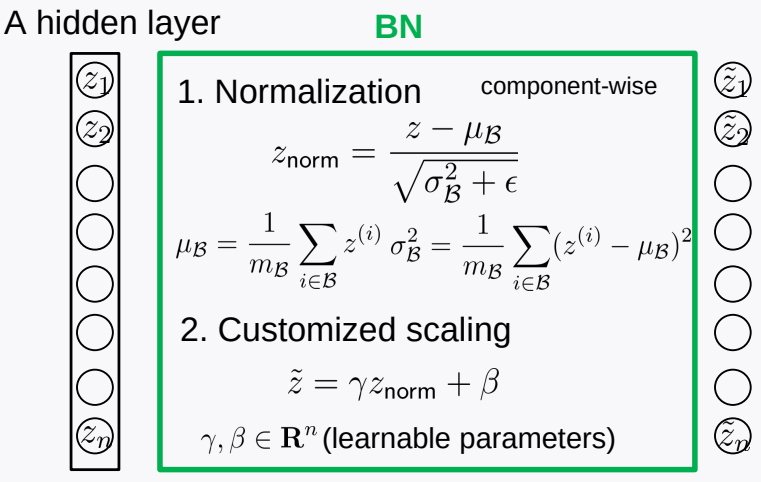


Figure 3: Batch Normalization (BN). First we do zero-centering and normalization with the mean $\mu_{\mathcal{B}}$ and the variance $\sigma_{\mathcal{B}}^2$ computed over the examples in an associated batch $\mathcal{B}$. Next we do customized scaling by introducing two new parameters that would also be learned during training: $\gamma \in \mathbf{R}^n$ and $\beta \in \mathbf{R}^n$.

Let $z := [z_1, \ldots, z_n]^T$ be the output of the considered hidden layer prior to activation. Here n denotes the number of units (neurons) in the hidden layer.

The Batch Normalization (BN for short) consists of two steps. First we do zero-centering and normalization using the mean and variance w.r.t. examples in an associated batch $\mathcal{B}$:

$$\mu_{\mathcal{B}} = \frac{1}{m_{\mathcal{B}}} \sum_{i \in \mathcal{B}} z^{(i)}, \quad \sigma^2_{\mathcal{B}} = \frac{1}{m_{\mathcal{B}}} \sum_{i \in \mathcal{B}} (z^{(i)} - \mu_{\mathcal{B}})^2 \tag{4}$$

where $(\cdot)^2$ indicates a *component-wise* square, and hence $\sigma^2_{\mathcal{B}} \in \mathbf{R}^n$. In other words, we generate the normalized output, say z_{norm}, as:

$$z_{\mathsf{norm}} = \frac{z^{(i)} - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \tag{5}$$

where division and multiplication are all component-wise. Here ϵ is a tiny value introduced to avoid division by 0 (typically 10^{-5}).

Second, we do customized scaling as per:

$$\tilde{z}^{(i)} = \gamma z^{(i)}_{\mathsf{norm}} + \beta \tag{6}$$

where $\gamma, \beta \in \mathbf{R}^n$ indicate two new scaling parameters which are learnable via training. Notice that again the operations in (6) are all component-wise.

The BN lets the model learn the optimal scale and mean of the inputs for each hidden layer. It turns out this technique plays a significant role in stabilizing and speeding up training especially for a very deep neural network. This has been verified experimentally by many practitioners, although no clear theoretical justification has been provided thus far.

Model for Discriminator

As a Discriminator model, we use a simple 3-layer fully-connected network with two hidden layers; see Fig. 4. Here the input size must be the same as that of the flattened real (or fake) image. Again we employ ReLU at hidden layers and logistic activation at the output layer.

Pytorch: How to use BN?

Now let us talk about how to do Pytorch programming for implementation. MNIST data loading is exactly the same as before. So we omit it. Instead let us first discuss how to use BN.

As you expect, Pytorch provides a built-in class for BN: `nn.BatchNorm1d()`. Here is how to use

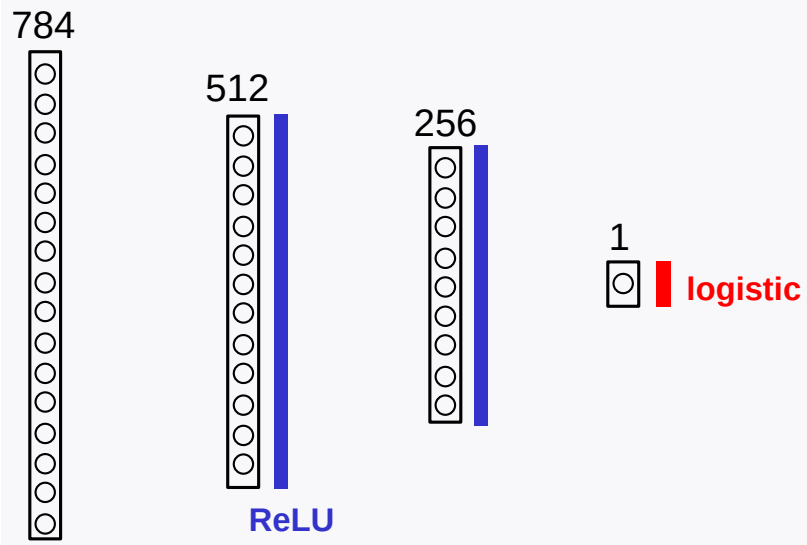


Figure 4: Discriminator: A 3-layer fully-connected neural network where the input size (the dimension of a flattened vector of a real (or fake) image) is 784 (= 28 × 28); the numbers of hidden neurons are 512, 256; and the output size is 1. We employ ReLU activation for every hidden layer, and logistic activation for the output layer.

such class in our setting:

```
class Generator(nn.Module):
    def __init__(self, latent_dim):
        super(Generator, self).__init__
        self.fc1 = nn.Linear(latent_dim,128)
        self.bn1 = nn.BatchNorm1d(128)
        self.relu = nn.ReLU()
                ⋮
    def forward(self, x):
        x = self.fc1(x)
        x = self.bn1(x)
        x = self.relu(x)
                ⋮
```

where `latent_dim` is the dimension of the latent signal (which we set as 100) and `nn.BatchNorm1d(128)` indicates a class that does BN for 128 hidden neurons.

Pytorch: Optimizers for Generator & Discriminator

We use Adam optimizers with `lr=0.0002` and `(b1,b2)=(0.5,0.999)`. Since we have two models

(Generator and Discriminator), we employ two optimizers accordingly:

```
generator = Generator(latent_dim)
discriminator = Discriminator()
optimizer_G = torch.optim.Adam(generator.parameters(),lr=lr,betas=(b1,b2))
optimizer_D = torch.optim.Adam(discriminator.parameters(),lr=lr,betas=(b1,b2))
```

where `Discriminator` is another neural network model which respects the structure in Fig. 4. Think about how to do programming for the model.

Pytorch: Generator input

As a generator input, we use a random signal with the *Gaussian* distribution. In particular, we use:

$$x \in \mathbf{R}^{\texttt{latent\_dim}} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}_{\texttt{latent\_dim}}).$$

Here is how to generate such Gaussian random signal in Pytorch:

```
x = torch.normal(mean=0,std=1,size=(batch_size,latent_dim))
```

Pytorch: Binary cross entropy loss

Consider the batch version of the GAN optimization (3):

$$\min_{w} \max_{\theta} \frac{1}{m_{\mathcal{B}}} \sum_{i \in \mathcal{B}} \log D_\theta(y^{(i)}) + \log(1 - D_\theta(G_w(x^{(i)}))) \tag{7}$$

Now introduce the ground-truth real-vs-fake indicator vector $[1, 0]^T$ (real=1, fake=0). Then, here the term $\log D_\theta(y^{(i)})$ can be viewed as the minus binary cross entropy between the real/fake indicator vector and its prediction counterpart $[D_\theta(y^{(i)}), 1 - D_\theta(y^{(i)})]^T$:

$$\begin{aligned} \log D_\theta(y^{(i)}) &= 1 \cdot \log D_\theta(y^{(i)}) + 0 \cdot \log(1 - D_\theta(y^{(i)})) \\ &= -\ell_{\mathsf{BCE}}(1, D_\theta(y^{(i)})). \end{aligned}$$

On the other hand, another term $\log(1 - D_\theta(\hat{y}^{(i)}))$ can be interpreted as the minus binary cross entropy between the fake-vs-real indicator vector (fake=0, real=1) and its prediction counterpart:

$$\begin{aligned} \log(1 - D_\theta(\hat{y}^{(i)})) &= 0 \cdot \log D_\theta(\hat{y}^{(i)}) + 1 \cdot \log(1 - D_\theta(\hat{y}^{(i)})) \\ &= -\ell_{\mathsf{BCE}}(0, D_\theta(\hat{y}^{(i)})). \end{aligned}$$

From this, we see that the cross entropy plays a role in computation of the objective function. In Lecture 17, we used a built-in class: `nn.CrossEntropyLoss()`. However, here we will use another slightly different built-in class: `nn.BCELoss()`. Here is to how to use in our setting:

```
CE_loss = nn.BCELoss()
loss = CE_loss(output,real_fake_indicator)
```

where `output` denotes an output of `discriminator` defined earlier, and `real_fake_indicator` is real/fake indicator vector (real=1, fake=0). Here one important thing to notice is that the

`output` is the result *after* logistic activation; and `real_fake_indicator` is also a *vector* with the same dimension as `output`. The function in `nn.BCELoss()` automatically detects the number of examples in an associated batch, thus yielding a normalized version (through division by $m_{\mathcal{B}}$).

Pytorch: Generator loss

Recall the proxy for for the generator loss that we will use:

$$\min_{w} \frac{1}{m_{\mathcal{B}}} \sum_{i \in \mathcal{B}} -\log D_{\theta}(G_{w}(x^{(i)})).$$

To implement this, we use:

```
g_loss = CE_loss(discriminator(gen_imgs),valid)
```

where `gen_imgs` indicate fake images (corresponding to $G_w(x^{(i)})$'s) and `valid` denotes an all-1's vector with the same dimension as `gen_imgs` (why?).

Pytorch: Discriminator loss

Recall the batch version of the optimization problem:

$$\max_{\theta} \frac{1}{m_{\mathcal{B}}} \sum_{i \in \mathcal{B}} \log D_{\theta}(y^{(i)}) + \log(1 - D_{\theta}(G_{w}(x^{(i)})))$$

Taking the minus sign in the objective, we obtain the equivalent minimization optimization:

$$\min_{\theta} \underbrace{\frac{1}{m_{\mathcal{B}}} \sum_{i \in \mathcal{B}} -\log D_{\theta}(y^{(i)}) - \log(1 - D_{\theta}(G_{w}(x^{(i)})))}_{\textit{discriminator loss}}$$

where the discriminator loss is defined as such minus version. Here is how to implement the discriminator loss:

```
real_loss = CE_loss(discriminator(real_imgs),valid)
fake_loss = CE_loss(discriminator(gen_imgs.detach()),fake)
d_loss = real_loss + fake_loss
```

where `real_imgs` indicate real images (corresponding to $y^{(i)}$'s) and `fake` denotes an all-0's vector with the same dimension as `gen_imgs`. Here we apply `.detach()` into `gen_imgs` which was associated with `generator`. This process is needed to enable a different computation w.r.t. discriminator loss.

Look ahead

Now we are done with the second application that we would deal with in Part III. Next time, we will move onto the last application of information theory: *fairness algorithms*.

November 20, 2019
Changho Suh (chsuh@kaist.ac.kr)

Lecture 21: Fairness machine learning

Recap

During the past six lectures, we have explored two prominent methodologies that arise in machine learning: (i) supervised learning; (ii) unsupervised learning. The goal of supervised learning is to estimate a functional $f(\cdot)$ of an interested system from input-output example pairs, as illustrated in Fig. 1(a).

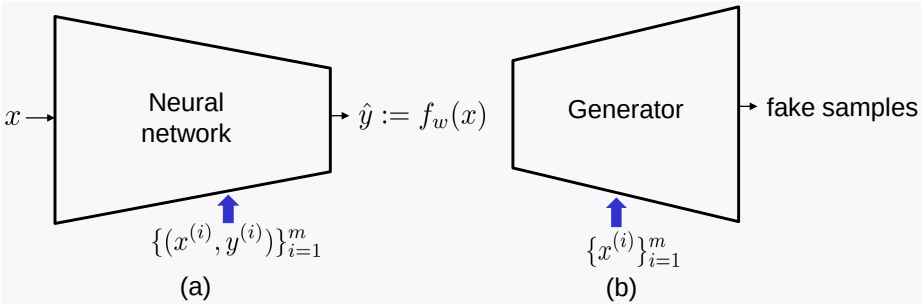


Figure 1: (a) Supervised learning: Learning the functional $f(\cdot)$ of an interested system from input-output example pairs $\{(x^{(i)}, y^{(i)})\}_{i=1}^m$; (b) Generative modeling (an unsupervised learning methodology): Generating fake data that resemble real data, reflected in $\{x^{(i)}\}_{i=1}^m$.

In an effort to translate a function optimization problem (a natural formulation of supervised learning) into a parameter-based optimization problem, we parameterized the function with a deep neural network. We also investigated some common practices adopted by many practitioners: Employing ReLU activation at hidden layers and softmax at the output layer; using the *cross entropy* loss (the optimal loss function in a sense of maximum likelihood); and applying an advanced version of gradient descent, named Adam optimizer.

We then moved onto unsupervised learning. We put a special emphasis on one famous unsupervised learning method: generative modeling, wherein the goal is to generate fake examples so that their distribution is as close as possible to that of real examples; see Fig. 1(b). In particular, we focused on one powerful generative model, named Generative Adversarial Network (GAN), which have played a revolutionary role in the modern AI field. We explored an interesting connection to two information-theoretic notions: KL divergence and mutual information. We also studied a practical method for solving the GAN optimization: Alternating gradient descent. Moreover, we learned how to do Pytorch implementation for GAN.

Next application

Now what is next? As the last application of information theory, we will explore one recent trending topic that arises in the modern machine learning: *Fairness machine learning.* There are three reasons that I pick up the topic as the last application.

The first reason is motivated by the recent trend in the machine learning field. As machine learning becomes prevalent in our daily lives involving a widening array of applications such as medicine, finance, job hiring and criminal justice, one morally & legally motivated need

in the design of machine learning algorithms is to ensure *fairness* for disadvantageous against advantageous groups. The fairness issue has received a particular attention from the learning algorithm by the US Supreme Court that yields unbalanced recidivism (criminal reoffending) scores across distinct races, e.g., predicts higher scores for blacks against whites; see Fig. 2. Hence, I wish to touch upon such trending & important topic for this course. I hope this may

Figure 2: The current machine learning algorithm employed in the US Supreme Court predicts higher recidivism scores for blacks against whites. If you want to know more, see `https://www.propublica.org/article/machine-bias-risk-assessments-incriminal-sentencing`

be of great interest to you guys. The second is w.r.t. a connection to information theory. It turns out that *mutual information* plays a key role in formulating an optimization problem for fairness machine learning algorithms. The last reason is that the optimization problem is closely related to the GAN optimization that we learned in the past lectures. You may see the perfect coherent sequence of applications, from supervised learning, GAN to fairness machine learning.

During upcoming lectures

For a couple of upcoming lectures (probably three lectures including today's one), we will investigate fairness machine learning in depth. Specifically what we are going to cover are four-folded. First off, we will figure out what fairness machine learning is. We will then study two prominent fairness concepts that have been established in the recent literature. We will also formulate an optimization for fairness machine learning algorithms which respect the fairness constraints based on the concepts. Next we will demonstrate that mutual information forms the basis of such optimization and the optimization can be rewritten as the GAN optimization that we learned in the prior application. Lastly we will learn how to solve the optimization and implement via Pytorch. Today we will cover the first two.

Fairness machine learning

Fairness machine learning is a subfield of machine learning that focuses on the theme of *fairness*. In view of the definition of machine learning, fairness machine learning can concretely be defined as a field of algorithms that train a machine so that it can perform a specific task in a *fair* manner.

Like traditional machine learning, there are two methodologies for fairness ML. One is fairness supervised learning, wherein the goal is to develop a *fair* classifier using input-output sample pairs: $\{(x^{(i)}, y^{(i)})\}_{i=1}^{m}$. The second is the unsupervised learning counterpart. In particular, what is a proper counterpart for generative modeling? A natural one is: *fair generative modeling* in which the goal is to generate *fairness-ensured* fake data which are also realistic. Due to the interest of time, we will focus only on the fairness supervised learning in this course.

Two major fairness concepts

In order to develop a *fair* classifier, we first need to understand what it means by *fairness*. Actually the fairness is a terminology that arises in *law* that deals with *justice*. So it has a long and rich history, and there are numerous concepts prevalent in the field of law. Here we focus only on two major and prominent concepts on fairness, which have received particular attention in the modern machine learning field.

The first is *disparate treatment (DT)*. This means an unequal treatment that occurs *directly* because of some sensitive information (such as race, sex, and religion), often called *sensitive attributes* in the literature. It is also called *direct discrimination* ("Gikjeop Chabyeol" in Korean), since such attributes directly serve to yield discrimination.

The second is *disparate impact (DI)*. This means an action that adversely affects one group against another *even with formally neutral rules* wherein sensitive attributes are never used in classification and therefore the DT does not occur. It is also called *indirect discrimination* ("Ganjeop Chabyeol" in Korean), since a disparate action is made *indirectly* through biased historical data.

Criminal reoffending predictor

Now how to design a fair classifier that respects the above two fairness concepts: DT and DI? For simplicity, let us explore this in the context of a simple yet concrete classification setting: *Criminal reoffending prediction*, wherein the task is to predict whether or not an interested individual with criminal records[1] would reoffend within two years. This is indeed the classification being done by the US Supreme Court for the purpose of deciding parole ("Gaseokbang" in Korean).

A simple setting

For illustrative purpose, here we consider a simplified version of the predictor wherein only a few information are employed for prediction. See Fig. 3.

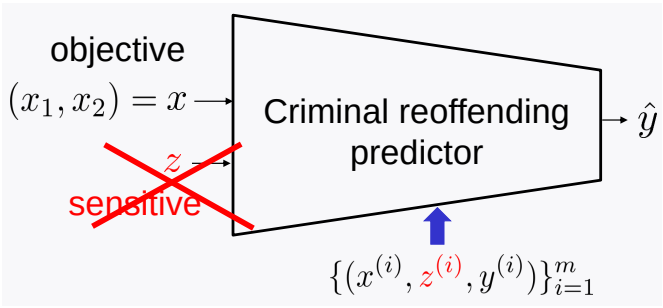


Figure 3: A criminal reoffending predictor.

There are two types of data employed: (i) *objective* data; (ii) *sensitive* data (or called *sensitive attributes*). For objective data that we denote by x, we consider only two features, say x_1 and x_2. Let x_1 be the number of prior criminal records. Let x_2 be a criminal type, e.g., misdemeanour or felony. For *sensitive* data, we employ a different notation, say z. Here we consider a simple scalar and binary case in which z indicates a race type only among white ($z = 0$) and black ($z = 1$). Let $\hat{y}$ be the classifier output which aims to represent the ground-truth conditional distribution $p(y|x, z)$. Here y denotes the ground-truth label: $y = 1$ means reoffending within 2 years; $y = 0$ otherwise. This is a supervised learning setup, so we are given m example triplets:

[1] A casual terminology for criminal records is *priors*.

$\{(x^{(i)}, z^{(i)}, y^{(i)})\}_{i=1}^m$.

How to avoid disparate treatment?

Firs of all, how to deal with disparate treatment? Recall the DT concept: An unequal treatment *directly* because of sensitive attributes. Hence, in order to avoid the DT, we should ensure that the prediction should not be a function of the sensitive attribute. A mathematical precise expression for this is:

$$p(y|x, z) = p(y|x) \quad \forall z. \tag{1}$$

Now how to ensure the above? The solution is very simple: Not using the sensitive attribute z at all in the prediction, as illustrated with a red-colored "x" mark in Fig. 3. Here an important thing to notice is that the sensitive attribute is offered as part of training data although it is not used as part of input. So $z^{(i)}$'s can be employed while designing an algorithm.

What about disparate impact?

How about for the other fairness concept: disparate impact? How to avoid the DI? Again recall the DI concept: An action that adversely affects one group against another even with formally neutral rules. Actually it is not that clear as to how to implement this mathematically.

To gain some insights, let us investigate the precise mathematical definition of DI. To this end, let us introduce a few notations. Let Z be a random variable that indicates a sensitive attribute. For instance, consider a binary case, say $Z \in \{0, 1\}$. Let $\tilde{Y}$ be a binary hard-decision value of the predictor output $\hat{Y}$ at the middle threshold: $\tilde{Y} := \mathbf{1}\{\hat{Y} \geq 0.5\}$. Observe a ratio of likelihoods of positive example events $\tilde{Y} = 1$ for two cases: $Z = 0$ and $Z = 1$.

$$\frac{\Pr(\tilde{Y} = 1|Z = 0)}{\Pr(\tilde{Y} = 1|Z = 1)}. \tag{2}$$

One natural interpretation is that a classifier is more fair when the ratio is closer to 1; becomes unfair if the ratio is far away from 1. The DI is quantified based on this, so it is defined as:

$$\mathsf{DI} := \min\left(\frac{\Pr(\tilde{Y} = 1|Z = 0)}{\Pr(\tilde{Y} = 1|Z = 1)}, \frac{\Pr(\tilde{Y} = 1|Z = 1)}{\Pr(\tilde{Y} = 1|Z = 0)}\right). \tag{3}$$

Notice that $0 \leq \mathsf{DI} \leq 1$ and the larger DI, the more fair the situation is.

Two cases

In view of the mathematical definition (3), reducing disparate impact means maximizing the mathematical quantity (3). Now how to design a classifier so as to maximize the DI then? Depending on situations, the design methodology can be different. To see this, think about two extreme cases.

The first is the one in which training data is already fair:

$$\{(x^{(i)}, z^{(i)}, y^{(i)})\}_{i=1}^m \rightarrow \text{large } \mathsf{DI}.$$

In this case, a natural solution is to simply rely on a conventional classifier that aims to maximize prediction accuracy. Why? Because maximizing prediction accuracy would well respect training data, which in turn yields large DI. The second is a non-trivial case in which training data is far from being fair:

$$\{(x^{(i)}, z^{(i)}, y^{(i)})\}_{i=1}^m \rightarrow \text{small } \mathsf{DI}.$$

In this case, the conventional classifier would yield small DI. This is indeed a challenging scenario where we need to take some non-trivial action for ensuring fairness.

In fact, the second scenario can often occur in reality, since there could be biased *historical* records which form the basis of training data. For instance, the Supreme Court can make some biased decisions for blacks against whites, and these are likely to be employed as training data. See Fig. 4 for one such unfair scenario. In Fig. 4, a hallowed (or black-colored-solid) circle

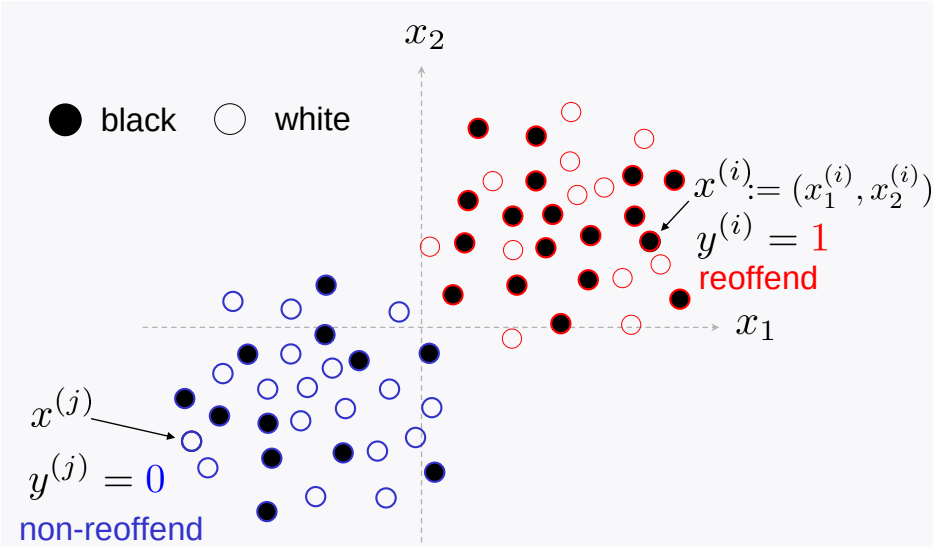


Figure 4: Data points visualization: A hallowed (or black-colored-solid) circle indicates a data point of an individual with white (or black) race; the red (or blue) colored edge denotes $y = 1$ reoffending (or $y = 1$ non-reoffending) label.

indicates a data point of an individual with white (or black) race; and the red (or blue) colored edge (ring) denotes the event that the interested individual reoffends (or non-reoffends) within two years. This is an *unfair* situation. Notice that for positive examples $y = 1$, there are more black-colored-solid circles than hallowed ones, meaning sort of biased historical records favouring whites against blacks. Similarly for negative examples $y = 0$, there are more hallowed circles relative to solid ones.

How to ensure large DI?

Now how to ensure large DI under all possible scenarios including the above unfair challenging scenario? To gain insights, first recall an optimization problem that we formulated earlier for the design of a conventional classifier:

$$\min_{w} \frac{1}{m} \sum_{i=1}^{m} \ell_{\mathsf{CE}}(y^{(i)}, \hat{y}^{(i)}) \tag{4}$$

where $\ell_{\mathsf{CE}}(\cdot, \cdot)$ indicates binary cross entropy loss, and w denotes weights (parameters) for a classifer. Here one natural approach to ensure large DI is to incorporate an DI-related constraint in the optimization (4). Maximizing DI is equivalent to minimizing $1 - \mathsf{DI}$ (since $0 \leq \mathsf{DI} \leq 1$). So we can resort to a very well-known technique in the optimization field: *regularization*. That is to add the two objectives with different weights.

Regularized optimization

Here is a regularized optimization:

$$\min_w \frac{1}{m}\sum_{i=1}^{m} \ell_{\mathsf{CE}}(y^{(i)}, \hat{y}^{(i)}) + \lambda \cdot (1 - \mathsf{DI}) \tag{5}$$

where λ denote a regularization factor that balances predication accuracy against the DI-associated objective (minimizing $1-\mathsf{DI}$). However, here an issue arises in solving the regularized optimization (5). Recalling the definition of DI

$$\mathsf{DI} := \min\left(\frac{\Pr(\tilde{Y}=1|Z=0)}{\Pr(\tilde{Y}=1|Z=1)}, \frac{\Pr(\tilde{Y}=1|Z=1)}{\Pr(\tilde{Y}=1|Z=0)}\right),$$

we see that DI is a complicated function of w. We have no idea as to how to express DI in terms of w.

Another way

Since directly expressing DI as a function of w is not doable, one can rely on another way to go. Another way that I will take is inspired by information theory, particularly by *mutual information*. Notice that $\mathsf{DI} = 1$ means that the sensitive attribute Z is *independent* of the hard decision $\tilde{Y}$ of the prediction. Remember one key property of mutual information: Mutual information between two input random varibles being zero is the "sufficient and necessary condition" for the independence between the two inputs. This motivates us to represent the constraint of $\mathsf{DI} = 1$ as:

$$I(Z;\tilde{Y}) = 0. \tag{6}$$

This captures the complete independence between Z and $\tilde{Y}$. Since the predictor output is $\hat{Y}$ (instead of $\tilde{Y}$), we consider another stronger condition that concerns $\hat{Y}$ directly:

$$I(Z;\hat{Y}) = 0. \tag{7}$$

Notice that the condition (7) is indeed stronger than (6), i.e., (7) implies (6). This is because

$$\begin{aligned} I(Z;\tilde{Y}) &\overset{(a)}{\leq} I(Z;\tilde{Y},\hat{Y}) \\ &\overset{(b)}{=} I(Z;\hat{Y}) \end{aligned} \tag{8}$$

where (a) is due to the chain rule and non-negativity of mutual information; and (b) is because $\tilde{Y}$ is a function of $\hat{Y}$: $\tilde{Y} := \mathbf{1}\{\hat{Y} \geq 0.5\}$. Notice that (7) together with (8) gives (6).

Strongly regularized optimization

In summary, the condition (7) indeed enforces the $\mathsf{DI} = 1$ constraint. This then motivates us to consider the following optimization:

$$\min_w \frac{1}{m}\sum_{i=1}^{m} \ell_{\mathsf{CE}}(y^{(i)}, \hat{y}^{(i)}) + \lambda \cdot I(Z;\hat{Y}). \tag{9}$$

Now the question of interest is: How to express $I(Z;\hat{Y})$ in terms of classifier parameters w? It turns out interestingly there is a way to express it. The idea is to use a GAN trick that we learned!

Look ahead

Next time we will review the GAN trick w.r.t. $I(Z;\hat{Y})$ and then use the trick to explicitly formulate an optimization.

November 25, 2019
Changho Suh (chsuh@kaist.ac.kr)

Lecture 22: Fairness-GAN

Recap

Last time, we introduced the last application of information theory: A fair classifier. As an example of a fair classifier, we considered a recidivism predictor wherein the task is to predict if an interested individual with prior criminal records (priors for short) would reoffend within two years; see Fig. 1 for illustration.

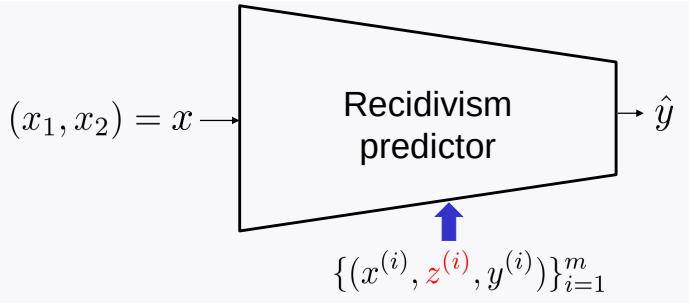


Figure 1: A simple recidivism predictor: Predicting a recidivism score $\hat{y}$ from $x = (x_1, x_2)$. Here x_1 indicates the number of prior criminal records; x_2 denotes a criminal type (misdemeanor or felony); and z is a race type among white ($z = 0$) and black ($z = 1$).

In order to avoid *disparate treatment* (one famous fairness concept), we made the sensitive attribute not to be included as part of input. To reflect another fairness constraint (*disparate impact* - DI for short), we introduced a regularized term into the original optimization dealing only with prediction accuracy, thus arriving at:

$$\min_w \frac{1}{m} \sum_{i=1}^m \ell_{\mathsf{CE}}(y^{(i)}, \hat{y}^{(i)}) + \lambda \cdot I(Z; \hat{Y}) \tag{1}$$

where $\lambda \geq 0$ is a regularization factor that balances prediction accuracy (reflected in the binary cross entropy terms) against the fairness constraint, reflected in $I(Z; \hat{Y})$. Remember that $I(Z; \hat{Y}) = 0$ is a sufficient condition for $\mathsf{DI} = 1$. At the end of the last lecture, I claimed that one can express the mutual information $I(Z; \hat{Y})$ of interest in terms of an optimization parameter w, thereby enabling us to train the model parameterized by w. I then told you that the idea for translation is to use a GAN trick that we learned in the past lectures.

Today's lecture

Today we will support the claim. Specifically we are going to cover three stuffs. First we will figure out what I meant by the GAN trick is. Next we will use the GAN trick to formulate an optimization yet in a simple binary sensitive attribute setting. Lastly we will extend to the general non-binary sensitive attribute case.

The GAN trick

Recall the inner optimization in GAN:

$$\max_{D(\cdot)} \frac{1}{m} \sum_{i=1}^{m} \log D(y^{(i)}) + \log(1 - D(\hat{y}^{(i)}))$$

where $y^{(i)}$ and $\hat{y}^{(i)}$ indicate the ith real and fake samples, respectively; and $D(\cdot)$ is a discriminator output. Also remember that we could make an interesting connection with mutual information:

$$I(T; Z) = \max_{D(\cdot)} \frac{1}{m} \sum_{i=1}^{m} \log D(y^{(i)}) + \log(1 - D(\hat{y}^{(i)})) \tag{2}$$

where $T = \mathbf{1}\{\text{Discriminator's input is real}\}$ and Z is a random variable that takes a real sample if $T = 1$; a fake sample if $T = 0$. Please do not be confused with the different Z in (1).

Here what I meant by the GAN trick is the other way around (reverse), meaning that we start with the mutual information of interest and then express it in terms of an optimization problem similarly to (2). Now let us apply this trick to our problem setting. For illustrative purpose, let us start with a simple binary sensitive attribute setting.

Mutual information vs. KL divergence

Notice in our optimization (1) that Z is a sensitive-attribute indicator, which plays the same role as T in (2). Similarly $\hat{Y}$ in (1) serves the same role as Z in (2). So one can expect that $I(Z; \hat{Y})$ in (1) would be expressed similarly as in (2). It turns out it is indeed the case. But there is a slight distinction in a detailed expression. To see what the distinction is, let us start by manipulating $I(Z; \hat{Y})$ from scratch.

Starting with the key relationship between mutual information and KL divergence, we get:

$$\begin{aligned}
I(Z; \hat{Y}) &= \mathsf{KL}\left(P_{\hat{Y},Z} \| P_{\hat{Y}} P_Z\right) \\
&\overset{(a)}{=} \sum_{\hat{y} \in \hat{\mathcal{Y}}, z \in \mathcal{Z}} P_{\hat{Y},Z}(\hat{y}, z) \log \frac{P_{\hat{Y},Z}(\hat{y}, z)}{P_{\hat{Y}}(\hat{Y}) P_Z(z)} \\
&= \sum_{\hat{y} \in \hat{\mathcal{Y}}, z \in \mathcal{Z}} P_{\hat{Y},Z}(\hat{y}, z) \log \frac{P_{\hat{Y},Z}(\hat{y}, z)}{P_{\hat{Y}}(\hat{Y})} + \sum_{\hat{y} \in \hat{\mathcal{Y}}, z \in \mathcal{Z}} P_{\hat{Y},Z}(\hat{y}, z) \log \frac{1}{P_Z(z)} \\
&\overset{(b)}{=} \sum_{\hat{y} \in \hat{\mathcal{Y}}, z \in \mathcal{Z}} P_{\hat{Y},Z}(\hat{y}, z) \log \frac{P_{\hat{Y},Z}(\hat{y}, z)}{P_{\hat{Y}}(\hat{Y})} + \sum_{z \in \mathcal{Z}} P_Z(z) \log \frac{1}{P_Z(z)} \\
&\overset{(c)}{=} \sum_{\hat{y} \in \hat{\mathcal{Y}}, z \in \mathcal{Z}} P_{\hat{Y},Z}(\hat{y}, z) \log \frac{P_{\hat{Y},Z}(\hat{y}, z)}{P_{\hat{Y}}(\hat{Y})} + H(Z)
\end{aligned}$$

where (a) is due to the definition of KL divergence; (b) comes from the total probability law; and (c) is due to the definition of entropy.

Observation

For the binary sensitive attribute case, we have:

$$\begin{aligned} I(Z;\hat{Y}) &= \sum_{\hat{y}\in\hat{\mathcal{Y}},z\in\mathcal{Z}} P_{\hat{Y},Z}(\hat{y},z)\log\frac{P_{\hat{Y},Z}(\hat{y},z)}{P_{\hat{Y}}(\hat{Y})} + H(Z) \\ &= \sum_{\hat{y}\in\hat{\mathcal{Y}}} P_{\hat{Y},Z}(\hat{y},1)\log\underbrace{\frac{P_{\hat{Y},Z}(\hat{y},1)}{P_{\hat{Y}}(\hat{y})}}_{=:D^*(\hat{y})} + \sum_{\hat{y}\in\hat{\mathcal{Y}}} P_{\hat{Y},Z}(\hat{y},0)\log\underbrace{\frac{P_{\hat{Y},Z}(\hat{y},0)}{P_{\hat{Y}}(\hat{y})}}_{=1-D^*(\hat{y})} + H(Z). \end{aligned} \tag{3}$$

Notice that the first log-inside term, defined as $D^*(\hat{y})$, has the close relationship with the second log-inside term in the above: the sum of the two is 1. This reminds us of the maximization formula in (2). So one may conjecture that $I(Z;\hat{Y})$ can be expressed in terms of an optimization problem as follows:

Claim 1:

$$I(Z;\hat{Y}) = \max_{D(\cdot)} \sum_{\hat{y}\in\hat{\mathcal{Y}}} P_{\hat{Y},Z}(\hat{y},1)\log D(\hat{y}) + \sum_{\hat{y}\in\hat{\mathcal{Y}}} P_{\hat{Y},Z}(\hat{y},0)\log\left(1-D(\hat{y})\right) + H(Z). \tag{4}$$

Proof: It turns out it is indeed the case. The proof is very simple. Taking the derivative w.r.t. $D(\hat{y})$, we get:

$$\frac{P_{\hat{Y},Z}(\hat{y},1)}{D^\star(\hat{y})} - \frac{P_{\hat{Y},Z}(\hat{y},0)}{1-D^\star(\hat{y})} = 0 \;\; \forall\hat{y}$$

where $D^\star(\hat{y})$ is the optimal solution to the optimization (4). Hence, we get:

$$D^\star(\hat{y}) = \frac{P_{\hat{Y},Z}(\hat{y},1)}{P_{\hat{Y},Z}(\hat{y},1)+P_{\hat{Y},Z}(\hat{y},0)} = \frac{P_{\hat{Y},Z}(\hat{y},1)}{P_{\hat{Y}}(\hat{y})}$$

where the second equality is due to the total probability law. Since this is exactly the same as $D^*(\hat{y})$ that we defined in (3), we complete the proof. ■

How to compute $I(Z;\hat{Y})$?

Notice that the formula (4) contains two probability quantities ($P_{\hat{Y},Z}(\hat{y},1), P_{\hat{Y},Z}(\hat{y},0)$) which are not available. The only things that we are given are: $\{(x^{(i)},z^{(i)},y^{(i)})\}_{i=1}^m$. So we need to now worry about what we can do with this information for computation of the probability quantities. First we can compute $\{\hat{y}^{(i)}\}_{i=1}^m$ from the samples. This together with the samples allows us to compute empirical distributions: $\mathbb{Q}_{\hat{Y},Z}(\hat{y}^{(i)},1)$ and $\mathbb{Q}_{\hat{Y},Z}(\hat{y}^{(i)},0)$. So as an estimate, we can take the empirical version of the true distributions:

$$\begin{aligned} \mathbb{Q}_{\hat{Y},Z}(\hat{y}^{(i)},0) &= \frac{1}{m}; \\ \mathbb{Q}_{\hat{Y},Z}(\hat{y}^{(i)},0) &= \frac{1}{m}. \end{aligned}$$

We have m samples in total. So the empirical distributions are uniform with the probability $\frac{1}{m}$. Why? Applying these empirical distributions to (4), we can approximate $I(Z;\hat{Y})$ as:

$$I(Z;\hat{Y}) \approx \max_{D(\cdot)} \frac{1}{m}\left\{\sum_{i:z^{(i)}=1}\log D(\hat{y}^{(i)}) + \sum_{i:z^{(i)}=0}\log\left(1-D(\hat{y}^{(i)})\right)\right\} + H(Z). \tag{5}$$

Notice that this approximation is exactly the same as the inner optimization formula in the GAN (2).

Fairness-GAN optimization

Now let us go back to our original optimization setting:

$$\min_{w} \frac{1}{m} \sum_{i=1}^{m} \ell_{\mathsf{CE}}(y^{(i)}, \hat{y}^{(i)}) + \lambda \cdot I(Z; \hat{Y}).$$

Applying the approximation (5) into the above, one can consider:

$$\min_{w} \max_{\theta} \frac{1}{m} \left\{ \sum_{i=1}^{m} \ell_{\mathsf{CE}}(y^{(i)}, \hat{y}^{(i)}) + \lambda \left(\sum_{i:z^{(i)}=1} \log D_\theta(\hat{y}^{(i)}) + \sum_{i:z^{(i)}=0} \log \left(1 - D_\theta(\hat{y}^{(i)})\right) \right) \right\} \tag{6}$$

where $D(\cdot)$ is parameterized with θ. The sensitive-attribute entropy $H(Z)$ in (5) is removed here, since it is irrelevant of the optimization parameters (θ, w). Observe now that the objective function has an explicit relationship with the optimization parameters (θ, w). Hence, the parameters are trainable via some practical algorithms such as gradient descent.

Extension to non-binary sensitive attribute

So far we have considered the binary sensitive attribute setting. However, in practice, this may not be necessarily the case. For instance, there are many race types such as Black, White, Asian, Hispanic, to name a few. Also there could be multiple sensitive attributes like gender and religion. In order to reflect such practical scenarios, we now consider a sensitive attribute with an *arbitrary alphabet size*. Notice that one can represent a collection of any multiple sensitive attributes as a single random variable yet with an arbitrary alphabet size. Hence, we consider such setting: $Z \in \mathcal{Z}$ where $|\mathcal{Z}|$ is not limited to two.

Recalling the key relationship between mutual information and KL divergence, we get:

$$\begin{aligned} I(Z; \hat{Y}) &= \mathsf{KL}\left(P_{\hat{Y},Z} \| P_{\hat{Y}} P_Z\right) \\ &= \sum_{\hat{y} \in \hat{\mathcal{Y}}, z \in \mathcal{Z}} P_{\hat{Y},Z}(\hat{y}, z) \log \frac{P_{\hat{Y},Z}(\hat{y}, z)}{P_{\hat{Y}}(\hat{y}) P_Z(z)} \\ &= \sum_{\hat{y} \in \hat{\mathcal{Y}}, z \in \mathcal{Z}} P_{\hat{Y},Z}(\hat{y}, z) \log \underbrace{\frac{P_{\hat{Y},Z}(\hat{y}, z)}{P_{\hat{Y}}(\hat{y})}}_{=:D^*(\hat{y},z)} + H(Z). \end{aligned} \tag{7}$$

Defining the log-inside term in the above as $D^*(\hat{y}, z)$, we obtain:

$$\sum_{z \in \mathcal{Z}} D^*(\hat{y}, z) = 1 \;\; \hat{y} \in \hat{\mathcal{Y}}.$$

This is due to the total probability law. Now what does this remind you of? You should be reminded of the claim that appeared in the extra problem in midterm!

Claim 2:

$$I(Z; \hat{Y}) = \max_{D(\hat{y},z): \sum_{z \in \mathcal{Z}} D(\hat{y},z)=1} \sum_{\hat{y} \in \mathcal{Y}, z \in \mathcal{Z}} P_{\hat{Y},Z}(\hat{y}, z) \log D(\hat{y}, z) + H(Z). \tag{8}$$

Proof: For those who do not remember details and/or do not prove the claim, let me provide a proof here. Since we have multiple equality constraints, marked in blue in the above, we first define the Lagrange function:

$$\mathcal{L}(D(\hat{y},z),\nu(\hat{y})) = \sum_{\hat{y}\in\hat{\mathcal{Y}},z\in\mathcal{Z}} P_{\hat{Y},Z}(\hat{y},z)\log D(\hat{y},z) + \sum_{\hat{y}\in\hat{\mathcal{Y}}}\nu(\hat{y})\left(1-\sum_{z\in\mathcal{Z}} D(\hat{y},z)\right)$$

where $\nu(\hat{y})$'s are Lagrange multipliers. Notice that there are the number $|\hat{\mathcal{Y}}|$ of Lagrange multipliers. Since the objective function is *concave* in $D(\cdot,\cdot)$, we can solve the problem via the KKT conditions:

$$\frac{d\mathcal{L}(D(\hat{y},z),\nu(\hat{y}))}{dD(\hat{y},z)} = \frac{P_{\hat{Y},Z}(\hat{y},z)}{D^\star(\hat{y},z)} - \nu^\star(\hat{y}) = 0 \;\; \forall \hat{y},z$$
$$\frac{d\mathcal{L}(D(\hat{y},z),\nu(\hat{y}))}{d\nu(\hat{y})} = 1 - \sum_{z\in\mathcal{Z}} D^\star(\hat{y},z) = 0 \;\; \forall \hat{y}.$$

Notice that plugging the following,

$$D^\star(\hat{y},z) = \frac{P_{\hat{Y},Z}(\hat{y},z)}{P_{\hat{Y}}(\hat{y})}, \qquad \nu^\star(\hat{y}) = P_{\hat{Y}}(\hat{y}).$$

we satisfy the KKT conditions. This implies that $D^\star(\hat{y},z)$ is indeed the optimal solution. Since $D^\star(\hat{y},z)$ is exactly the same as $D^*(\hat{y},z)$ that we defined in (7), we complete the proof. ■

Fairness-GAN optimization: General case

Again for computation of $I(Z;\hat{Y})$, we rely on the empirical version of the true distribution $P_{\hat{Y},Z}(\hat{y},z)$:

$$\mathbb{Q}_{\hat{Y},Z}(\hat{y}^{(i)},z^{(i)}) = \frac{1}{m}. \tag{9}$$

So we get:

$$I(Z;\hat{Y}) \approx \max_{D(\hat{y},z):\sum_{z\in\mathcal{Z}} D(\hat{y},z)=1} \sum_{i=1}^{m}\frac{1}{m}\log D(\hat{y}^{(i)},z^{(i)}) + H(Z). \tag{10}$$

Lastly by parameterizing $D(\cdot,\cdot)$ with θ and excluding $H(Z)$ (irrelevant of (θ,w)), we obtain the following optimization:

$$\min_{w}\max_{\theta:\sum_{z\in\mathcal{Z}} D_\theta(\hat{y},z)=1} \frac{1}{m}\left\{\sum_{i=1}^{m}\ell_{\mathsf{CE}}(y^{(i)},\hat{y}^{(i)}) + \lambda\sum_{i=1}^{m}\log D_\theta(\hat{y}^{(i)},z^{(i)})\right\}. \tag{11}$$

While it is similar to the GAN, it comes with a couple of distinctions. First the Generator is replaced with the Classifier. Hence, it includes additional loss term that captures prediction accuracy, reflected in $\sum_{i=1}^{m}\ell_{\mathsf{CE}}(y^{(i)},\hat{y}^{(i)})$. Also we now have multiple (the number of $|\mathcal{Z}|$) outputs in the Discriminator. Here each $D_\theta(\hat{y}^{(i)},z^{(i)})$ can be interpreted as:

$$\Pr\left(\hat{y}^{(i)} \text{ belongs to } z^{(i)}\right).$$

Fairness-GAN architecture

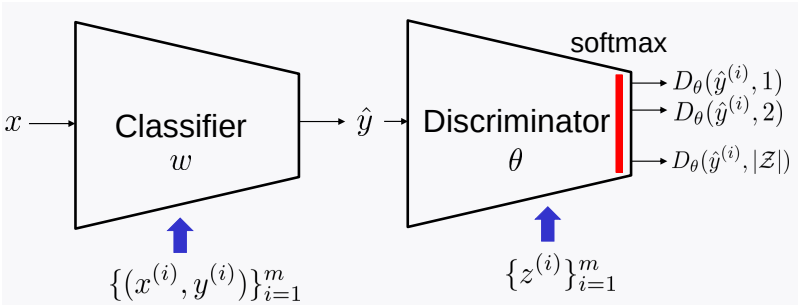


Figure 2: Fairness-GAN architecture.

In view of the final optimization formula (11), the Fairness-GAN architecture can be illustrated as in Fig. 2. Notice that we have the number $|\mathcal{Z}|$ of outputs in the Discriminator. In an attempt to ensure the equality constraint in (11), we may want to use the softmax activation in the output layer.

Look ahead

We are now done with the optimization problem formulation for Fairness-GAN. Next time, we will study how to solve the optimization (11), as well as how to implement it via Pytorch.

December 2, 2019
Changho Suh (chsuh@kaist.ac.kr)

Lecture 23: Fairness-GAN implementation

Recap

Last time, we formulated an optimization that respects two fairness constraints: disparate treatment (DT) and disparate impact (DI). Given m example triplets $\{(x^{(i)}, z^{(i)}, y^{(i)})\}_{i=1}^m$:

$$\min_w \frac{1}{m} \sum_{i=1}^m \ell_{\mathsf{CE}}(y^{(i)}, \hat{y}^{(i)}) + \lambda \cdot I(Z; \hat{Y})$$

where $\hat{y}^{(i)}$ indicates the Classifier output, depending only on $x^{(i)}$ (not on the sensitive attribute $z^{(i)}$ due to the DT constraint); and λ is a regularization factor that balances prediction accuracy against the DI constraint, reflected in $I(Z; \hat{Y})$. Using the GAN trick, we could approximate $I(Z; \hat{Y})$ in the form of optimization; for the binary sensitive attribute case, the approximation reads:

$$I(Z; \hat{Y}) \approx \max_{D(\cdot)} \frac{1}{m} \left\{ \sum_{i: z^{(i)}=1} \log D(\hat{y}^{(i)}) + \sum_{i: z^{(i)}=0} \log \left(1 - D(\hat{y}^{(i)})\right) \right\} + H(Z).$$

We then parameterized $D(\cdot)$ with θ to obtain:

$$\min_w \max_\theta \frac{1}{m} \left\{ \sum_{i=1}^m \ell_{\mathsf{CE}}(y^{(i)}, \hat{y}^{(i)}) + \lambda \left(\sum_{i: z^{(i)}=1} \log D_\theta(\hat{y}^{(i)}) + \sum_{i: z^{(i)}=0} \log \left(1 - D_\theta(\hat{y}^{(i)})\right) \right) \right\}. \quad (1)$$

Two questions that arise are: (i) how to solve the optimization (1)?; and (ii) how to implement the algorithm via Pytorch?

Today's lecture

Today we will address these two questions. Specifically what we are going to do are four-folded. First we will investigate a practical algorithm that allows us to attack the optimization (1). We will then do a case study for the purpose of exercising the algorithm: recidivism prediction. In the process, we will put a special emphasis on one implementation detail: synthesizing *unfair* dataset that we will use in our experiments. Lastly we will learn how to implement programming via Pytorch.

Observation

Let us start by translating the optimization (1) into the one that is more programming-friendly:

$$\begin{aligned}
&\min_w \max_\theta \frac{1}{m}\left\{\sum_{i=1}^m \ell_{\mathsf{CE}}(y^{(i)},\hat{y}^{(i)}) + \lambda\left(\sum_{i:z^{(i)}=1}\log D_\theta(\hat{y}^{(i)}) + \sum_{i:z^{(i)}=0}\log\left(1-D_\theta(\hat{y}^{(i)})\right)\right)\right\}\\
&\overset{(a)}{=} \min_w \max_\theta \frac{1}{m}\left\{\sum_{i=1}^m \ell_{\mathsf{CE}}(y^{(i)},\hat{y}^{(i)}) + \lambda\left(\sum_{i=1}^m z^{(i)}\log D_\theta(\hat{y}^{(i)}) + (1-z^{(i)})\log\left(1-D_\theta(\hat{y}^{(i)})\right)\right)\right\}\\
&\overset{(b)}{=} \min_w \max_\theta \frac{1}{m}\left\{\sum_{i=1}^m \ell_{\mathsf{CE}}\left(y^{(i)},\hat{y}^{(i)}\right) - \lambda\sum_{i=1}^m \ell_{\mathsf{CE}}\left(z^{(i)},D_\theta(\hat{y}^{(i)})\right)\right\}\\
&= \min_w \max_\theta \underbrace{\frac{1}{m}\left\{\sum_{i=1}^m \ell_{\mathsf{CE}}\left(y^{(i)},G_w(x^{(i)})\right) - \lambda\ell_{\mathsf{CE}}\left(z^{(i)},D_\theta(G_w(x^{(i)}))\right)\right\}}_{=:J(w,\theta)}
\end{aligned}$$

where (a) is due to $z^{(i)} \in \{0,1\}$; and (b) is due to the definition of the binary cross entropy loss $\ell_{\mathsf{CE}}(\cdot,\cdot)$. Notice that $J(w,\theta)$ contains two cross entropy loss terms, each being a non-trivial function of $G_w(\cdot)$ and/or $D_\theta(\cdot)$. Hence, in general, $J(w,\theta)$ is highly non-convex in w and highly non-concave in θ.

Alternating gradient descent

Similar to the prior GAN setting, what we can do in this context is to apply the only technique that we are aware of: *alternating gradient descent*. And then hope for the best. So we employ the $k:1$ alternating gradient descent:

1. Update Classifier (Generator)'s weight:
$$w^{(t+1)} \leftarrow w^{(t)} - \alpha_1 \nabla_w J(w^{(t)},\theta^{(t\cdot k)}).$$
2. Update Discriminator's weight k times while fixing $w^{(t+1)}$: `for i=1:k`,
$$\theta^{(t\cdot k+i)} \leftarrow \theta^{(t\cdot k+i-1)} + \alpha_2 \nabla_\theta J(w^{(t+1)},\theta^{(t\cdot k+i-1)}).$$
3. Repeat the above.

Again one can use the Adam optimizer possibly together with the batch version of the algorithm.

Optimization used in our experiments

Here is the optimization that we will use in our experiments:

$$\min_w \max_\theta \frac{1}{m}\left\{\sum_{i=1}^m (1-\lambda)\ell_{\mathsf{CE}}\left(y^{(i)},G_w(x^{(i)})\right) - \lambda\ell_{\mathsf{CE}}\left(z^{(i)},D_\theta(G_w(x^{(i)}))\right)\right\}. \tag{2}$$

In order to restrict the range of λ into $0 \le \lambda \le 1$, we apply the $(1-\lambda)$ factor to the prediction accuracy loss term.

Like the prior GAN setting, let us define two loss terms. One is "Generator (or Classifier) loss":

$$\min_{w} \max_{\theta} \underbrace{\frac{1}{m}\left\{\sum_{i=1}^m (1-\lambda)\ell_{\mathsf{CE}}\left(y^{(i)},G_w(x^{(i)})\right) - \lambda\ell_{\mathsf{CE}}\left(z^{(i)},D_\theta(G_w(x^{(i)}))\right)e\right\}}_{\text{"generator (classifier) loss"}}.$$

Given w, the Discriminator wishes to maximize:

$$\max_{\theta} -\frac{\lambda}{m}\sum_{i=1}^{m}\ell_{\mathsf{CE}}\left(z^{(i)}, D_{\theta}(G_{w}(x^{(i)}))\right).$$

This is equivalent to minimizing the minus of the objective:

$$\min_{\theta} \underbrace{\frac{\lambda}{m}\sum_{i=1}^{m}\ell_{\mathsf{CE}}\left(z^{(i)}, D_{\theta}(G_{w}(x^{(i)}))\right)}_{\text{“discriminator loss”}}. \tag{3}$$

This is how we define the “Discriminator loss”.

Performance metrics

Unlike to the prior two settings (supervised learning and GAN), here we need to introduce another performance metric that captures the degree of fairness. To this end, let us first define the hard-decision value of the prediction output w.r.t. an unseen (test) example:

$$\tilde{Y}_{\mathsf{test}} := \mathbf{1}\{\hat{Y}_{\mathsf{test}} \geq 0.5\}.$$

The test accuracy is then defined as:

$$\frac{1}{m_{\mathsf{test}}}\sum_{i=1}^{m_{\mathsf{test}}}\mathbf{1}\{y_{\mathsf{test}}^{(i)} = \tilde{y}_{\mathsf{test}}^{(i)}\}$$

where m_{test} denotes the number of test examples. This is an empirical version of the ground truth $\Pr(Y_{\mathsf{test}} = \tilde{Y}_{\mathsf{test}})$.

Now how to define a fairness-related performance metric? Recall the mathematical definition of DI:

$$\mathsf{DI} := \min\left(\frac{\Pr(\tilde{Y}=1|Z=0)}{\Pr(\tilde{Y}=1|Z=1)}, \frac{\Pr(\tilde{Y}=1|Z=1)}{\Pr(\tilde{Y}=1|Z=0)}\right). \tag{4}$$

Here you may wonder how to compute two probability quantities of interest: $\Pr(\tilde{Y}=1|Z=0)$ and $\Pr(\tilde{Y}=1|Z=1)$. Again we use its empirical version relying on the Law of Large Number:

$$\Pr(\tilde{Y}=1|Z=0) = \frac{\Pr(\tilde{Y}=1, Z=0)}{\Pr(Z=0)} \approx \frac{\sum_{i=1}^{m_{\mathsf{test}}}\mathbf{1}\{\tilde{y}_{\mathsf{test}}^{(i)}=1, z_{\mathsf{test}}^{(i)}=0\}}{\sum_{i=1}^{m_{\mathsf{test}}}\mathbf{1}\{z_{\mathsf{test}}^{(i)}=0\}}.$$

The above approximation is getting more and more accurate as m_{test} gets larger. Similarly we can approximate $\Pr(\tilde{Y}=1|Z=1)$. This way, we can evaluate the DI (4).

A case study

Now let us exercise what we have learned so far with a simple example. As a case study, we consider the same simple setting that we introduced earlier: recidivism prediction, wherein the task is to predict if an interested individual reoffends within 2 years, as illustrated in Fig. 1.

Synthesizing unfair dataset

One thing that we need to be careful about in the context of fairness machine learning is w.r.t. *unfair* dataset. Here for simplicity, we will employ a *synthetic* dataset, not real-world dataset.

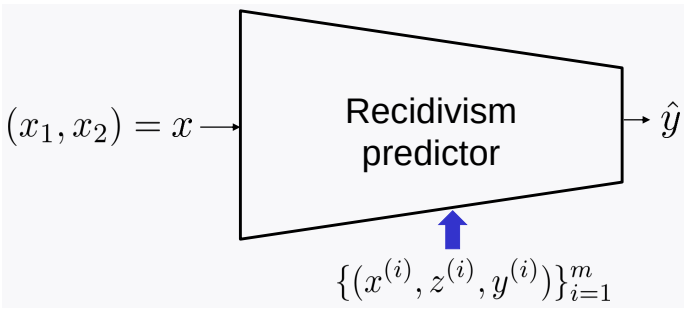


Figure 1: Predicting a recidivism score $\hat{y}$ from $x = (x_1, x_2)$. Here x_1 indicates the number of prior criminal records; x_2 denotes a criminal type: misdemeanor or felony; and z is a race type among white ($z = 0$) and black ($z = 1$).

In fact, there is a real-world dataset that concerns the recidivism prediction, called the COMPAS dataset. But this contains many attributes, so it is a bit complicated. Hence, we will take a particular yet simple method to synthesize a much simpler unfair dataset.

Recall the visualization of an unfair data scenario that we investigated in Lecture 21 and will form the basis of our synthetic dataset (to be explained in the sequel); see Fig. 2. In Fig. 2,

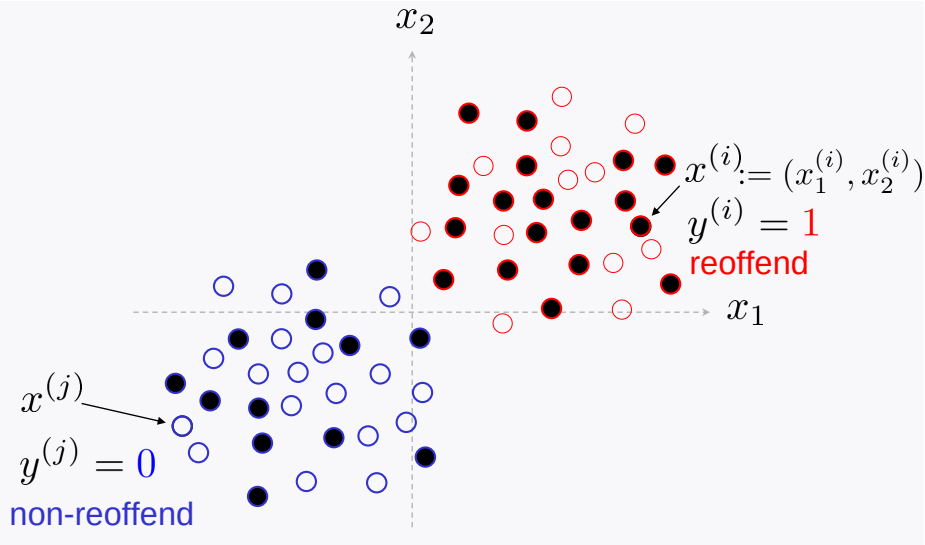


Figure 2: Data points visualization: A hallowed (or black-colored-solid) circle indicates a data point of an individual with white (or black) race; the red (or blue) colored edge denotes $y = 1$ reoffending (or $y = 1$ non-reoffending) label.

a hallowed (or black-colored-solid) circle indicates a data point of an individual with white (or black) race; and the red (or blue) colored edge (ring) denotes the event that the interested individual reoffends (or non-reoffends) within two years. This is indeed an *unfair* scenario: for $y = 1$, there are more black-colored-solid circles than hallowed ones; similarly for $y = 0$, there are more hallowed circles relative to solid ones.

To generate such an unfair dataset, we employ a simple method. See Fig. 3. We first generate m labels $y^{(i)}$'s so that they are i.i.d. each being according to $\mathsf{Bern}(\frac{1}{2})$. For indices of positive examples ($y^{(i)} = 1$), we then generate i.i.d. $x^{(i)}$'s according to $\mathcal{N}((1,1), 0.5^2\mathbf{I})$; and i.i.d. $z^{(i)}$'s

as per Bern(0.8), meaning that 80% are blacks ($z = 1$) and 20% are whites ($z = 0$) among the positive individuals. Notice that the generation of $x^{(i)}$'s is not quite realistic - the first and second components in $x^{(i)}$ do not precisely capture the number of priors and a criminal type. You can view this generation as sort of a crude *abstraction* of such realistic data. On the other hand, for negative examples ($y^{(i)} = 0$), we generate i.i.d. $(x^{(i)}, z^{(i)})$'s with different distributions: $x^{(i)} \sim \mathcal{N}((-1,-1), 0.5^2\mathbf{I})$ and $z^{(i)} \sim \text{Bern}(0.2)$, meaning that 20% are blacks ($z = 1$) and 80% are whites ($z = 0$). This way, $z^{(i)} \sim \text{Bern}(\frac{1}{2})$. Why?

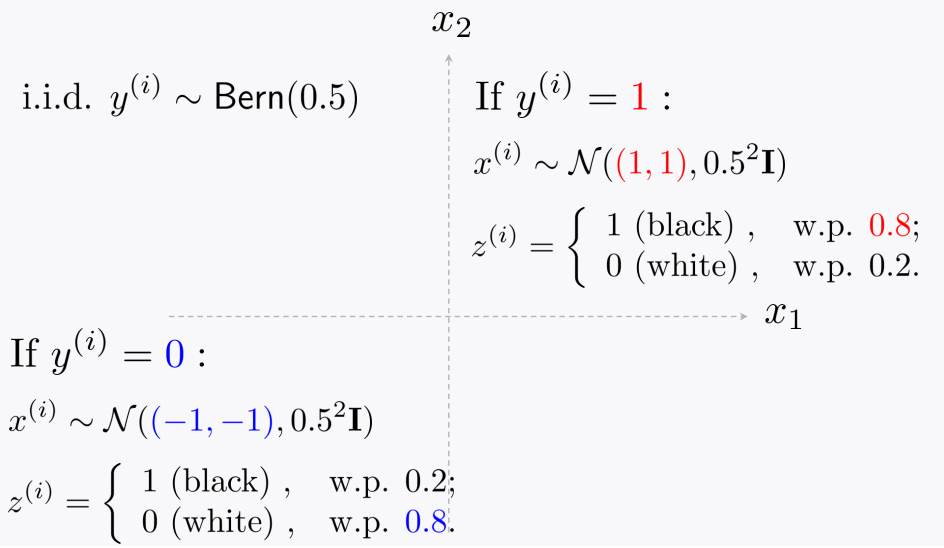


Figure 3: Synthesizing a simple unfair dataset.

Model architecture

Fig. 4 illustrates the architecture of Fairness-GAN. Since we focus on the binary sensitive attribute, we have a single output $D_\theta(\hat{y})$ in the Discriminator. For models of Classifier and

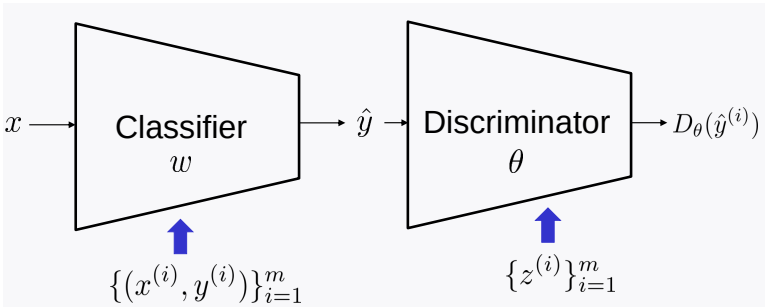


Figure 4: The architecture of Fairness-GAN.

Discriminator, we employ very simple single-layer neural networks with logistic activation in the output layer; see Fig. 5.

Pytorch: Synthesizing unfair dataset

Now let us discuss how to implement such details via Pytorch. First unfair dataset synthesis.

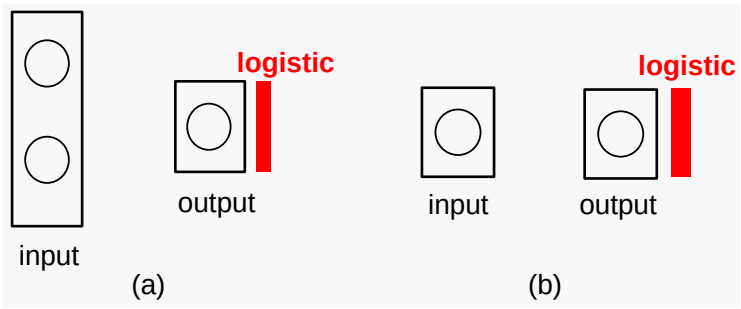


Figure 5: Models for (a) Classifier and (b) Discriminator.

To generate i.i.d binary random variables for labels, we use:

```
y_train = numpy.random.binomial(1,0.5,size=(train_size, ))
```

where the first two arguments of `(1,0.5)` specify Bern(0.8); and the null space followed by `train_size` indicates a single dimension. Remember we generate i.i.d. Gaussian random variables for $x^{(i)}$'s. To this end, one can use one of the following two ways:

```
x = torch.normal(mean=(1,1),std=0.5,size=(train_size,2))
x = numpy.random.normal(loc=(1,1),scale=0.5,size=(train_size,2))
```

Pytorch: Optimizers for Classifier & Discriminator

For Classifier, we use Adam optimizer with `lr_c=0.005` and `(b1,b2)=(0.9,0.999)`. For Discriminator, we use SGD with `lr_d=0.005`:

```
classifier = Classifier()
discriminator = Discriminator()
optimizer_G = torch.optim.Adam(classifier.parameters(),lr=lr_c,betas=(b1,b2))
optimizer_D = torch.optim.SGD(discriminator.parameters(),lr=lr_d)
```

where `Classifier()` and `Discriminator()` are neural network models which respect the structure in Fig. 5. Since you may be familiar with how to program the models, we omit details here.

Pytorch: Classifier (Generator) loss

Recall the optimization problem of interest:

$$\min_{w}\max_{\theta}\frac{1}{m}\left\{\sum_{i=1}^{m}(1-\lambda)\ell_{\mathsf{CE}}\left(y^{(i)},G_w(x^{(i)})\right)-\lambda\sum_{i=1}^{m}\ell_{\mathsf{CE}}\left(z^{(i)},D_\theta(G_w(x^{(i)}))\right)\right\}.$$

To implement the Classifier loss (the objective in the above), we use:

```
p_loss = CE_loss(y_pred,y_train)
f_loss = CE_loss(discriminator(y_pred.detach()),z_train)
g_loss = (1- lambda)*p_loss - lambda*f_loss
```

where `y_pred` indicates the Classifier output; `y_train` denotes a label; and `z_train` is a binary sensitive attribute. Note in this Fairness-GAN implementation that we use the full batch.

Pytorch: Discriminator loss

Recall the Discriminator loss that we defined in (3):

$$\min_{\theta} \frac{\lambda}{m} \sum_{i=1}^{m} \ell_{\mathsf{CE}} \left(z^{(i)}, D_{\theta}(G_{w}(x^{(i)})) \right).$$

To implement this, we use:

```
f_loss = CE_loss(discriminator(y_pred.detach()), z_train)
d_loss = lambda*f_loss
```

Pytorch: Evaluation

Recall the DI performance:

$$\mathsf{DI} := \min \left(\frac{\Pr(\tilde{Y}=1|Z=0)}{\Pr(\tilde{Y}=1|Z=1)}, \frac{\Pr(\tilde{Y}=1|Z=1)}{\Pr(\tilde{Y}=1|Z=0)} \right).$$

To evaluate the DI performance, we rely on the following approximation:

$$\Pr(\tilde{Y}=1|Z=0) \approx \frac{\sum_{i=1}^{m_{\mathsf{test}}} \mathbf{1}\{\tilde{y}_{\mathsf{test}}^{(i)}=1, z_{\mathsf{test}}^{(i)}=0\}}{\sum_{i=1}^{m_{\mathsf{test}}} \mathbf{1}\{z_{\mathsf{test}}^{(i)}=0\}}.$$

Here is how to implement this in details:

```
y_tilde = (y_pred >0.5).int().squeeze()
z0_ind = (z_train == 0.0)
z1_ind = (z_train == 1.0)
z0_sum = int(torch.sum(z0_ind))
z1_sum = int(torch.sum(z1_ind))
Pr_y1_z0 = float(torch.sum((y_tilde==1)[z0_ind]))/z0_sum
Pr_y1_z1 = float(torch.sum((y_tilde==1)[z1_ind]))/z1_sum
```

December 4, 2019
Changho Suh (chsuh@kaist.ac.kr)

Lecture 24: Remarks on how to do research

Today's lecture

Today is the last lecture day for the course. So I would like to share something which may be useful for your future careers. In particular, I wish to talk about two things w.r.t. how to do research. The first is my personal advice which might be helpful for you to advance modern technologies. The second is my own methodology w.r.t. how to efficiently read research papers.

Summary of the course

Prior to telling you my advice on your future career, let me summarize what we have learned so far. In Part I, we have studied three information-theoretic notions: (i) entropy; (ii) mutual information; (iii) KL divergence. In Part II, we explored important roles of the notions while proving Shannon's two celebrated theorems: (a) source coding theorem; (b) channel coding theorem.

In Part III, we could find interesting roles of such notions in a different trending field: machine learning. We investigated: (i) the core role of cross entropy in the design of a loss function for supervised learning; (ii) the fundamental role of the KL divergence in the design of a powerful unsupervised learning framework: GAN; (iii) the recently discovered role of mutual information in the design of machine learning algorithms that ensure fairness.

Be strong at fundamentals

One key message that we can see from what we have learned is that *fundamental concepts* (entropy, KL divergence, and mutual information in this context) play important roles. So in view of this, my advice is: *Be strong at fundamentals!* But you may be confused about my advice because there are many fundamentals and what I meant by fundamentals are not clear enough. So you may wonder which fundamentals which I recommend you to focus on. The fundamentals that I would like to put a special emphasis on are w.r.t. modern technologies. To explain what it means, let us first investigate fundamentals which played important roles in old days.

Fundamentals in old days

As we all know, old-days technologies were driven by 1st, 2nd and 3rd industrial revolutions. Such revolutions were due to breakthrough inventions, insired by *scientific results*. The key invention that led to the 1st industrial revolution was the *steam engine*. It is based on *theomodynamics* in which physics and chemistry play foundational roles. The invention that triggered the 2nd revolution was *electricity* which is based on *electromagnetism* which again physics provides a theory behind. The invention for the 3rd revolution was *computer* which forms the basis of our daily lives. The computer is based on the invention of *semiconductor* which again physics and chemistry provide the foundation for. These fundamentals are definitely important. However, these are not the ones that I meant. What I wish to put an emphasis on is w.r.t. modern-day technologies.

Fundamentals in modern days

The modern-days technologies are being driven by the 4th industrial revolution. As we all know, the main theme of the 4th industrial revolution is: *Artificial Intelligence (AI)*. Now what are the fundamentals that form the basis of AI technologies? Remember that the fields of machine learning and deep learning which provide key methodologies for achieving machine intelligence are all based on *optimization*, which is a branch of mathematics. Hence, one major driving force for AI technologies is: *mathematics*.

Four fundamentals in mathematics

In particular, the following four branches in mathematics play foundational roles in AI technologies. The first is obviously the optimization. The second is a field which provides instrumental tools with which one can translate the objective function and/or constraints into simple and tractable formulas. The field is: *linear algebra*. As you may be familiar with, many seemingly-complicated mathematical formulas can be expressed as simple terms that involve matrix multiplications and additions. The third is a field that plays a role in dealing with uncertainty that appears in random quantities. The field is: *probability*. The last is a field that serves to shed optimal architectural insights into machine learning models of interest. That is: *information theory*. Again remember the roles of information-theoretic notions that we investigated for supervised learning and unsupervised learning.

These are the very fundamentals which I believe are crucial for advancing the 4th industrial revolution empowered by AI technologies. So my advice is: Be strong at these fundamentals. One caveat here is that such fundamentals are highly likely to be built only when you are at school. Of course it is a bit exaggerated, but it seems indeed the case according to the experiences of my own and many others. You may be able to understand what this means *after* you graduate; not enough time would be given for you to deeply understand some principles and also your stamina would not be as good as that of now.

Another thing that you should be strong at

Is building up such fundamentals enough for you to advance AI technologies? Unfortunately it is not the case. There is another crucial thing that you should be strong at. Remember the end product of machine learning. In most cases, it yields "algorithms" instead of "closed-form solutions". In other words, it provides a *set of instructions* on what we should do in order to perform a task of interest, instead of directly telling us a task result. As you experienced, such set of instructions requires heavy computations; the computation by hand is almost impossible, so it should be done on a *computer*. This is exactly where another important thing kicks in. That is: *programming tools* such as Python and Pytorch that we have experienced in Part III. I highly recommend you to be strong at this as well. Fast implementations via great programming skills will help you to realize your ideas as well as to advance them.

One caveat w.r.t. programming tools is that unfortunately the tools are *changing over time*. This is because the capability and efficiency of programming languages depend on computation resource and power which are being advanced in a fast manner. Hence, you should follow up whenever such changes are made.

How to read research papers?

A second thing that I would like to share with you is regarding a methodology as to how to do research. In particular, I want to discuss on: *how to read research papers*. Two reasons that I picked up this topic. One is that very unfortunately, there is no systematic curriculum on this, although this may be of the greatest interest to students. So students simply attempt to read any seemingly-interesting (or randomly chosen) paper without any particular thought

and strategy in their minds. The second is that reading papers efficiently is perhaps the most difficult thing even for experienced students who are often exposed to research environments. These are the reasons that I decided to talk about this topic.

Disclaimer: The contents that I will share soon are my own opinions.

Two strategies

There are two strategies depending on what you wish to do. The first is a *passive approach* which can be applied to the case when you want to learn some basics and trends (not necessarily to write a paper) in an interested field. The second is an *active approach* which I think is applicable to the case when you wish to be an expert (or forced to write a paper) in a trending and non-mature field.

Passive approach

What I mean by the passive approach is to *rely on experts*. In other words, it is to wait until experts can teach in a very accessible manner, and then read well-written papers by such experts. It would be the best if such papers are of survey type. Two natural questions that you can ask then are: (i) how to figure out such experts?; (ii) how to read such well-written papers? Here are my answers.

How to figure out experts?

For the first question, I recommend the following three ways. The first is to ask professors, seniors and/or peers who are actually working in the field. Usually well-known figures are easily recognized by anyone in the field. So simply ask them. The second is to rely on Google: searching with proper keywords. Nowadays many (actually too many) blogs are floating around websites. So some famous relevant blogs may provide pointers and references which might hint for such experts. Once you locate someone, you may want to check their Google citation records. The last is to check organizers and tutorial/keynote speakers in flagship conferences and/or workshops. Those are likely to be experts. You may be confused when identified scholars are too young and so their track records may be a bit weak. In such case, you may want to check their advisors. Great advisors usually yield great students.

How to read well-written papers?

Once you figure out experts, the next is to read well-written papers by them. Usually most-cited survey papers are a good choice. Here are a couple of guidelines on how to read them.

The first and most important suggestion is to read *intensively*. Not skimming through a paper. Preferably, please read while writing down what you understand. You will have a better, clearer and deeper understanding when you translate their words into your own. I also recommend you to ponder on theorems prior to reading their proofs. Key messages are delivered mostly in the form of theorems, and understanding the main messages and their implications, reflected in the theorems, is the most important part. Diving into proofs (usually containing technical and non-insightful details) prior to understanding what the theorems mean will distract you without any insight, so you will lose your interest and/or burn out quickly. Once you grasp the main messages, you can then dive into technical details.

The second suggestion is to read the paper *several times* until you fully absorb contents. Personally I consider a very good paper as a *Bible*. Reading several times will you make have deep and diverse understandings on the contents. Also be familiar with proof techniques if any. A good paper usually contains very simple and insightful proof techniques. Simply a short proof

is not necessarily good unless it is insightful. I believe that a good proof comes with *insights* although it is rather long.

The last guideline is to respect experts' notations, logical flows, mindsets and writing-styles. Usually excellent writers spend lots of time in choosing their notations, making a storyline and refining their writings. So theirs are best picks and well-cultivated. You may also have your owns. But if you find theirs better than yours, then please do not hesitate to replace yours with them. Or if you have none or if you cannot judge, simply follow theirs. It is worth copying.

Active approach

Now let us move onto the second approach: *the active approach*. As mentioned earlier, this is intended for those who wish to write papers in a relatively new (emerging) field. In other words, it is for those who cannot wait until the field becomes mature so that experts start to world-tour for delivering keynotes and/or tutorials. The spirit of the approach is based on *trials-&-errors*. It is to read *many* recent papers and *do back-and-forth* until you figure out a big picture. As you may image, this is not that simple. It is in fact quite difficult. The keys to this approach are to: (i) identify relevant and good papers *quickly*; (ii) figure out main ideas of the identified papers *quickly*. Now how to do such quickly? How to do back-and-forth with multiple papers?

How to identify relevant/good papers quickly?

For the first question, I recommend the following two ways. The first is almost the same as the one mentioned in the passive approach. That is to ask professors, seniors and/or peers working in the field. But there is one caveat here. The caveat is that they should be *experts* in the field. This is because figuring out good papers in a new field is not that simple. So usually non-experts do not easily identify such papers, and even if they are able to do so with non-trivial efforts, they are not motivated to spend much time in searching for papers on behalf of others.

The second suggestion is to search in relevant conferences and workshops, yet in a very strategic manner. As you may imagine, there are many conferences/workshops and so there are many papers to check. So the key is to quickly figure out relevancy and quality of papers to make a short list that contains only the core papers worth delving into. Here is my recommendation to do this. First look only at the "title". If it contains some relevant keywords, it is grammatically correct, *and* looks fancy (at least not awkward), then start to read "abstract". Otherwise remove it from a list. There may be some non-well-written papers which yet contain ground-breaking results, but this is a very rare case. Also even if we miss it according to such a strict rule, later you will have enough chances to revisit the papers by others. Don't worry. Next, if an abstract of a surviving paper is relevant *and* well-written, we can then add it to a short list. Otherwise, throw it away. Usually authors make lots of efforts in writing and polishing an abstract, so if it includes some awkward sentences, then it would be likely to contain much worse sentences in the main body. This will make you have hard time figuring out the main idea of the paper. Not a good idea to keep the paper in the list.

Further short-list papers if needed

Once you apply the above rules, then you may end up with several or more papers. If the number of filtered papers is around (or beyond) 10, I recommend you to further short-list papers via investigating more information. Here is how. First set up your goal: what you want to know from a chosen paper. You then start reading "introduction" while keeping in mind your goal. If it is relevant to your goal *and* again it is well-written, then make it stay in the list. Otherwise,

throw it away.

Figure out main ideas of the short-listed papers

At this moment, the list should contain only several (or a couple of) papers. Otherwise, repeat the prior step with a more strict caliber until you have only several papers in the list. Once you pass the step, then search for the *punch-line sentence* in the introduction of each paper. If you do not locate or do not understand it, then start reading other sections until you figure out. Once you figure out, rephrase the punch-line sentence into your own and write it down in the first-page heading of the paper. If you feel very annoyed while figuring out, simply stop reading and move onto a next paper.

Do back-and-forth

Iterate the above procedure while summarizing the main ideas of possibly all the short-listed papers. Two things that you should keep in mind during the iteration process. The first is: Do not spend much time in reading "related works". Usually "related works" are heavy and are not along the main storyline of the paper. This is because authors are sort of forced to cite all the relevant (although not quite related) works to avoid any negative reviews potentially from non-cited paper authors. So you do not need to investigate all the contents in "related works". Rather it is mostly okay to simply ignore all of them. The second is w.r.t. the investigation of other references. If you *really need* to read another referenced paper in the process, then try to read it very quickly while keeping in mind you goal. Preferably do not be distracted with reading other references or with spending much time on them.

You can stop the iteration process until either: (a) you figure out a big picture; or (b) you find a very well-written paper (which I called the "anchor paper") that describes a whole picture clearly. The way to read the "anchor paper" is the same as the one that I mentioned w.r.t. the passive approach. This way, you will figure out a big picture.

Do your own research

Once you figure out a big picture, I recommend you to focus on doing your own research while not being too much distracted with reading other papers. Here is a guideline for the step.

1. Choose a challenge that you want to address;
2. Formulate a simple yet concrete problem that can address the challenge;
3. Try to solve the problem with conventional wisdom from first-principle thinking.

You may be stuck especially at the 3rd procedure in the above. If so, you may want to interact with your advisors, seniors and/or peers.

Two final remarks

Finally I would like to leave two remarks. The first is: Don't give up during the process. Doing research is not a simple process at all. It is very natural for you to feel discouraged and headaches several times while in the process. But the patience will get you rewarded in the end. The second is w.r.t. communication skills. As you may see in the guidelines, a key is to quickly figure out the writing quality and the main idea of a paper. So I strongly recommend you to work hard to improve reading comprehension and grammars.