Worcester State University

MA 470: Capstone Experience

Fall 2018

Bayesian and Measure Theoretic Extensions of Generative Adversarial Networks

Author: Michael Rose

Abstract

Generative Adversarial Networks are a class of machine learning algorithms which have the ability to learn rich, hierarchical probability distributions over a variety of different data types. This is achieved by simultaneously training two neural networks and pitting them against each other in an adversarial framework. This paper explores the background information needed to understand these algorithms and shows extensions of the original idea.

1 Introduction

The game Go originated in China 3000 years ago and is widely considered to be the oldest board game continuously played to this day. The rules are simple: Each player takes turns to place black or white stones on a 19×19 board, trying to capture their opponent's stones or to surround empty space to make points of territory.

While the rules are simple, Go is very complex. There are approximately

$$2002031940862976956714479730135578509969862591524303826112350007$$
$$7348906207401543395415870817978902800457543055297838678738457 04$$
$$5887237708512899422163924031484980226164357409684272 61$$

legal board positions in Go, or $2 \cdot 10^{170}$.[1] To put this into perspective, there are about $1.58 \cdot 10^{55}$ possible chess board configurations (without pawn promotion), and it is estimated that there are about $7.1 \cdot 10^{79}$ atoms in the known observable universe. [2] This means Go contains more than a googol ($1.0 \cdot 10^{100}$) times as many board configurations as are in Chess and almost a googol ($2.8 \cdot 10^{90}$) times as many board configurations as there are atoms in the universe.

Due to its complexity, Go has been viewed as the most challenging of classical games for Artificial Intelligence. In a game with such simple rules that lack constraints and massive numbers of possibilities, it is not reasonable to just search through the possibilities like was done with Chess. Despite decades of research, the strongest computer Go programs were only able to play at the level of amateurs. When Deep Blue beat the Chess grandmaster Garry Kasparov in 1997, many experts still believed that a computer could never figure out how to play Go at a similar level of mastery.

In October 2015, Google Deepmind's AlphaGo became the first computer program to defeat a professional Go player, beating Fan Hui 5 - 0.[3] AlphaGo went on to play the legendary Lee Sedol, winner of 18 world titles in Go, and won in March 2016 to an audience of 200

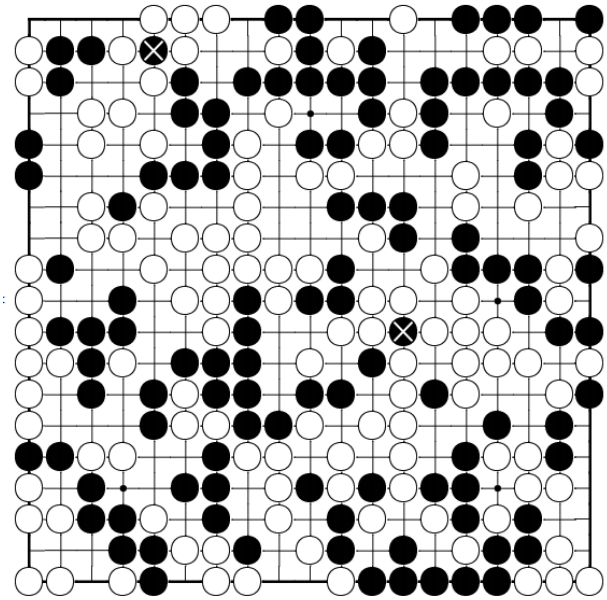


Figure 1: A Go Board

million people worldwide. Winning with a record of 4 - 1, it played a handful of highly inventive winning moves, several of which were so surprising that they overturned hundreds of years of received wisdom and have since been examined extensively by players of all levels. While winning a world championship, AlphaGo managed to teach the world completely new knowledge about what is perhaps the most studied and contemplated game in history.

How did Deepmind do it? AlphaGo uses a Monte Carlo tree search algorithm to find its moves based on knowledge previously 'learned' through machine learning. The learning is thanks to its model structure, something known as a multilayer perceptron (covered in this paper). Multilayer perceptrons are also known as neural networks.

AlphaGo uses a neural network to predict the best possible next move. Once it takes in the information from its opponent, it iteratively chooses the next best possible move. The original AlphaGo was initially trained on human gameplay data, given a database of around 30 million moves. Once it achieved proficiency, it played matches against itself and updated its knowledge of Go in the process.

In December 2017, Deepmind released a new iteration of AlphaGo known as AlphaGo Zero. This version was provided the rules of the game, a random initialization and no more data. It then played games against itself. Within 24 hours it had achieved a superhuman level of play in the games of chess, shogi, and Go - beating former world champion programs Stockfish, Elmo and AlphaGo in each case. [4]

What does this mean? A neural network was able to discover deep patterns about the complex game of Go. More generally, it was able to discover a rich, hierarchical model of a

probability distribution over a very high dimensional data set. It could estimate the most probable move that would lead to victory at each step of the game, regardless of the intensely large data space it needed to explore.

Since then, the field of Deep Learning has exploded in popularity and led to a large-scale revival in artificial intelligence research. Deep Learning is mainly focused around ways to use neural networks to generate models of other types of high dimensional data, such as natural language, images, video, audio, time series, and more.

Typically, a neural network is focused on learning a representation of some data and then making decisions about any given new data. In the AlphaGo example, the representation was the latent space of the game of Go and the decision was to make the best next move such that we maximize the probability of winning the game.

Once we know the patterns that make up the data, what if we don't have a directive for it? What if we don't care about making any decision, and we just want to create something?

Enter Generative Adversarial Networks (GANs). GANs are about creating things. They are neural networks that can draw pictures, or compose symphonies, or write scripts. As an example of the difference between a regular neural network and a generative adversarial network - a regular neural network, given data of paintings, would be able to tell you if some new painting was made by Van Gogh. A GAN, given the same data, would be able to paint you a new painting in the style of Van Gogh (GAN Gogh if you will).

A GAN is concerned with learning the patterns hidden within data, and then using them to create new samples of data. They have been applied in a variety of use cases, including:

- producing samples of photorealistic images
- visualizing interior design from quick sketches
- creating textures for computer graphics
- modeling patterns of motion in video
- reconstructing 3D models from images
- increasing definition for magnified images
- improving astronomical images by removing noise
- controversially, placing celebrity faces on other people's bodies in videos
- generating promising drug candidates
- creating new music in the style of certain artists
- repairing images and videos with missing pixels / frames

1.1 Image Examples

Figure 2: trained on faces then asked to create a new face. None of these people exist.

Figure 3: trained on paintings in the style of a particular artist, asked to recreate a real photo

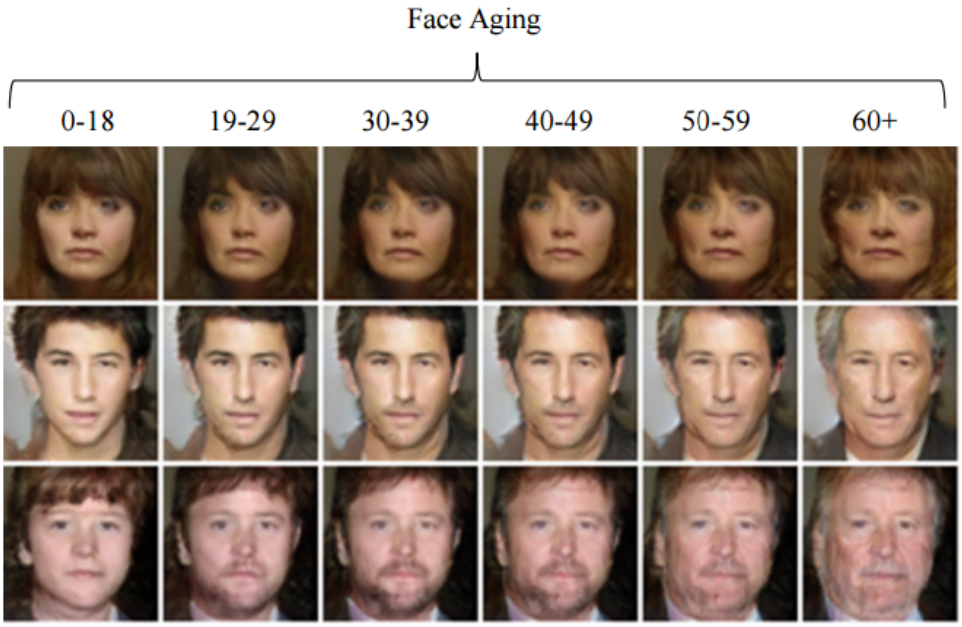


Figure 4: trained on different age groups, then asked to show aging process for a new image

Figure 5: A painting made with a GAN that sold for 432,500 USD at auction

2 Probability Primer

2.1 Random Variables

Random variables are variables whose possible values are the outcome of a random phenomenon. We can say that a random variable is a function that maps the outcomes of some unpredictable process to a numerical quantity. An example would be to consider the outcome of a dice roll, which we could model as a random variable where the outcome is some member of the set $\{1, 2, 3, 4, 5, 6\}$.

When the image of our random variable X is finite or countably infinite, we say that it is a

discrete random variable. If our image is uncountably infinite, then X is called a continuous random variable. [5]

Formally, we can define a random variable in the following manner:

A random variable $X : \Omega \mapsto E$ is a measurable function from Ω, a set of possible outcomes to E, a measurable space. Generally $E = \mathbb{R}$.

2.2 Distributions

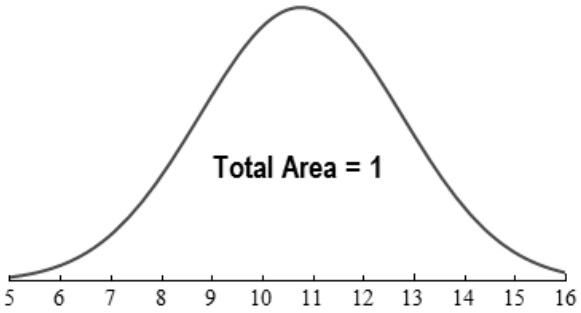


Figure 6: A Probability Distribution

Given a random variable, we can ask how likely it is that our random variable X is equal to some value. This is equivalent to asking for the probability of an event $\omega : X(\omega) = n$ or $P(X = n)$. If we were to record all of the resulting probabilities, we would have a probability distribution.

Suppose we were to integrate the probability distribution from $-\infty$ to ∞. Then, in order to be a valid probability distribution function, it must equal 1. If we were to take the integral of a probability distribution from $-\infty$ to some value x, we would have an area section known as a cumulative distribution which is described by a probability distribution's given cumulative distribution function.

As probability distributions are essentially distributions of the outcome of random variables, they are similarly divided into two classes: Discrete probability distributions, in which the set of outcomes is a discrete list of the probabilities of the outcomes and continuous probability distributions in which the set of possible outcomes take on values in a continuous range, typically $\mathbb{R}$, where the probability of any individual event is 0.

Throughout this paper, we will see a variety of random variables expressed as probability distributions. Our notation will look like this:

$$X_i \sim \text{Distribution}(a_1, a_2, ..., a_n)$$

where X_i is a random variable, Distribution is the name of our distribution, and $a_1, ..., a_n$ are our parameters. For example, if we say

$$X \sim \text{Exp}(\lambda)$$

then we have a function of the form:

$$y = \lambda e^{-\lambda x}$$

This example was an exponential distribution. There are many different named distributions, and each distribution has its own probability density function which will determine its characteristics.

2.3 Multiplication Rule

If two probabilities have outcomes which are completely independent of each other, then we can say

$$P(A, B) = P(A) \cdot P(B)$$

For example: in a dice roll each roll has no effect on the other rolls of the die. Therefore, the probability of getting two 6s in a row is given by

$$P(6, 6) = P(6) \cdot P(6) = (P(6))^2 = (\tfrac{1}{6})^2 = \tfrac{1}{36}$$

2.4 Baye's Theorem

Baye's Theorem is an equation that describes the probability of an event, based on the prior knowledge of conditions that are related to the event.

Mathematically, $P(A|B) = \frac{P(B|A)P(A)}{P(B)}$ where A, B are events and $P(B) \neq 0$.

In this equation $P(A|B)$ is the probability of event A, given that event B is true and $P(A), P(B)$ are the probabilities of observing A or B independently of each other.

In the Bayesian interpretation, probability measures a degree of belief. Baye's Theorem then links the degree of belief in our proposition before and after accounting for the evidence. Therefore:

In the equation $P(A|B) = \frac{P(B|A)P(A)}{P(B)}$:

- $P(A)$ is the initial degree of belief in A and we call this our prior.
- $P(B)$ is the information that has already been given. This is called our evidence.
- $P(A|B)$ is our degree of belief in A after we have accounted for event B. We call this our posterior.
- $P(B|A)$ is the degree of belief we have in B given that A has happened. This is called our likelihood.

Putting it all together we get:

$$\text{Posterior} = \frac{\text{likelihood} * \text{prior}}{\text{evidence}}$$

Baye's Rule can be derived from the definition of conditional probability:

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

$P(B) \neq 0$, $P(A \cap B)$ is the joint probability of both A and B.

Then:

$$P(A|B)P(B) = P(A \cap B) = P(B \cap A) = P(B|A)P(A)$$

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \text{ and } P(B|A) = \frac{P(A|B)P(B)}{P(A)}$$

2.5 Expectation

Expectation is another name for expected value. Intuitively, we can think of the expected value of a random variable as the long run average vale of repetitions of the experiment it represents.

For the finite discrete case:

Let X be a random variable with a finite number of finite outcomes $x_1, x_2, ..., x_k$ occurring with probabilities $p_1, p_2, ..., p_k$ respectively.

$$\mathrm{E}[X] = \sum_{i=1}^{k} x_i p_i = x_1 p_1 + x_2 p_2 + ... + x_k p_k$$

For the countably infinite case:

Let X be a random variable with a countable set of finite outcomes $x_1, x_2, ...$ occurring with probabilities $p_1, p_2, ...$ respectively such that the infinite sum $\sum_{i=1}^{\infty} |x_i| p_i$ converges. Then the expected value of X is defined as the series

$$\mathrm{E}[X] = \sum_{i=1}^{\infty} x_i p_i$$

For the continuous case:

If X is a random variable whose cumulative distribution function admits a density $f(x)$, then the expected value is defined by the following:

$$\mathrm{E}[X] = \int_{\mathbb{R}} x f(x) dx$$

As an example, suppose wish to know the expected value of rolling a die. Since each roll is a discrete value such that $X \in \{1, 2, 3, 4, 5, 6\}$, we get the following:

$$\begin{aligned} \mathrm{E}[X] &= \sum_{i=1}^{6} x_i p_i && (1) \\ &= 1 \cdot \frac{1}{6} + 2 \cdot \frac{1}{6} + 3 \cdot \frac{1}{6} + 4 \cdot \frac{1}{6} + 5 \cdot \frac{1}{6} + 6 \cdot \frac{1}{6} && (2) \\ &= [1 + 2 + 3 + 4 + 5 + 6] \cdot \frac{1}{6} = \frac{21}{6} = 3.5 && (3) \end{aligned}$$

Thus the expected long term average of a series of dice rolls is 3.5.

For the continuous case, suppose we earned a degree in medieval poetry. Given our exquisite command of the English language practiced during the middle ages, we wish to brighten the lives of those who will listen to us. As a result, we get a job at a call center.

Bored during your shift one day, you begin to ponder the length of time before an incoming call. You start with the assumption of a constant rate of incoming phone calls during certain hours (for example, 50 calls / hour from noon til 7, 30 calls / hour afterwards). Then, letting X be the time until the next call:

$$\mathrm{E}[X] = \int_0^\infty \lambda e^{-\lambda x} dx = -\lambda e^{-\lambda x} - \frac{e^{-\lambda x}}{\lambda}\Big|_0^\infty = \frac{1}{\lambda}$$

where λ is the rate of the incoming calls.

3 Measure Theory

3.1 Metric Space

Definition 1. *A metric space is an ordered pair (M, d) where M is a set and d is a metric on M. In other words, a function $d : M \times M \to \mathbb{R}$ such that the following properties hold:*

- $d(x, y) \geq 0$ *non-negativity*
- $d(x, y) = 0 \iff x = y$ *identity of indiscernables*
- $d(x, y) = d(y, x)$ *symmetry*
- $d(x, z) \leq d(x, y) + d(y, z)$ *triangle inequality*

A Metric Space is a set for which distances between all members of the set are defined. Taken together, these distances are called a metric on the set. The most common example of a metric space is 3D Euclidean space. A "metric" is a generalization of the Euclidean metric arising from 4 properties (explained above) of Euclidean distance. The Euclidean metric defines the distance between 2 points as the length of the straight line segment connecting them. [6]

3.2 Neighborhoods

Definition 2. *For a metric space (M, d), a set V is a neighborhood of a point p if there exists an open ball (for 3D space, the space bounded by a sphere such that the boundary points that constitute the sphere are not included) with center p and radius $r > 0$ such that*

$$B(p; r) = \{x \in M | d(x, p) < r\} \text{ is contained in } V$$

Intuitively, a neighborhood of a point is a set of points containing that point where one can move some amount in any direction away from that point without leaving the set.

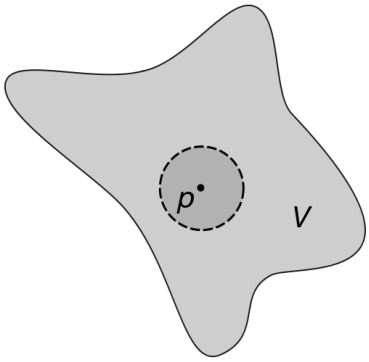


Figure 7: V is a neighborhood of point p

3.3 Continuity

Definition 3. *a function f is continuous at a point c of its domain if, for any neighborhood $N_1(f(c))$ there exists a neighborhood $N_2(c)$ such that $f(x) \in N_1(f(c))$ whenever $x \in N_2(c)$*

A function is continuous at a point c if the range of the restriction of f to a neighborhood of c shrinks to a single point $f(c)$ as the width of the neighborhood around c shrinks to 0.

3.4 Lipschitz Continuity

Definition 4. *Given two metric spaces (M_1, d_1) and (M_2, d_2) where d_2 denotes the metric on the set M_1 and d_2 is the metric on the set M_2, a function $f : M_1 \to M_2$ is called Lipschitz continuous if there exists a real constant $K \geq 0$ such that, for all x_1 and x_2 in M_1:*

$$d_2(f(x_1), f(x_2)) \leq K d_1(x_1, x_2), \text{ or}$$

$$\frac{d_2(f(x_1), f(x_2))}{d_1(x_1, x_2)} \leq K$$

The constant K is referred to as a Lipschitz constant for the function f.

Definition 5. *A real valued function $f : \mathbb{R} \to \mathbb{R}$ is called Lipschitz continuous if there exists a positive real constant K such that for all real x_1, x_2*

$$|f(x_1) - f(x_2)| \leq K|x_1 - x_2|$$

In this case, M_2 is the set of real numbers $\mathbb{R}$ with the metric $d_2(y_1, y_2) = |y_1 - y_2|$ and M_1 is a subset of $\mathbb{R}$ with the same metric.

Intuitively, a Lipschitz continuous function is limited in how fast it can change.

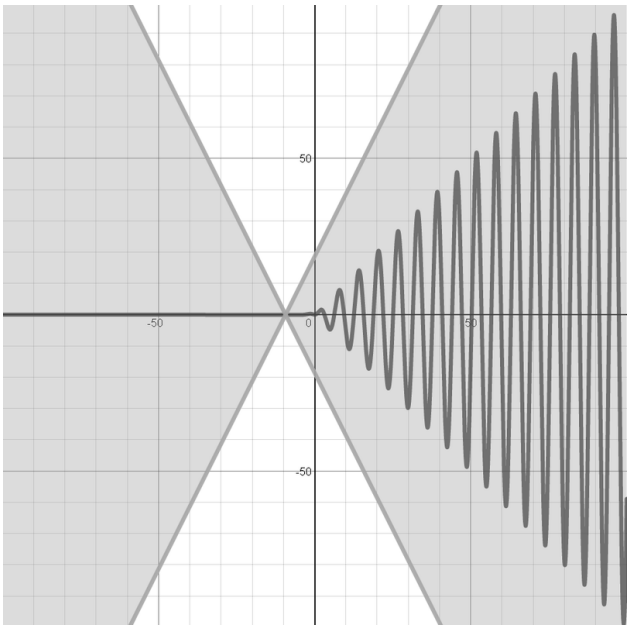


Figure 8: A Lipschitz Continuous Function in which K=2

For real-valued functions of several real variables, this holds if and only if the absolute value of the slopes of all secant lines are bounded by K. The set of lines of slope K passing through a point on the graph of the function forms a circular cone, and a function is Lipschitz if and only if the graph of the function everywhere lies completely outside the cone. For intuition on the cone, see figure 8.

3.5 Boundedness

Definition 6. *For a function f defined on a set X, f is bounded if there exists a real number B such that $|f(x)| \leq B$ for all x in X.*

Put succinctly, f is bounded if the set of its values is bounded. For a set to be bounded it must have both upper and lower bounds. For example, a set of real numbers is bounded if it is contained within a finite interval.

If $f(x) \leq C$ for all x in X, then the function is bounded above by C. If $f(x) \geq A$ for all x in X, then the function is bounded below by A. If both of the above are true, $A \leq f(x) \leq C$ and $f(x)$ is bounded within [A, C].

3.6 Dense Sets

Definition 7. *Suppose (M, d) is a metric space. Then a subset $S \subset M$ is called dense in M if for every $\epsilon > 0$ and $x \in M$,*

$$B(x, \epsilon) \cap S \neq \emptyset$$

where B is a ball.

Intuitively, we can think of a subset $S \subset M$ as dense in M if wherever we look in M, we find an element of S really close by. Another way of putting this is that no matter what point in M we choose and no matter how small of an open ball we constrain our neighborhood to, there will always be an element $s \in S$ in $B(m, \epsilon)$.

As an example, consider the rational numbers $\mathbb{Q} \subset \mathbb{R}$ as dense in the real numbers $\mathbb{R}$.

3.7 Compact Space

Definition 8. *For any metric space (M, d), being compact is equivalent to (M, d) being complete and totally bounded.*

Compactness is a property that generalizes the notion of a subset of Euclidean space being closed (such as [a, b] containing points a and b) and bounded.

Intuitively, a space is complete if there are no points missing from it (either inside or at the boundary).

Definition 9. *Let (M, d) be a metric space. A sequence (x_n) in M is called a Cauchy sequence if for any $\epsilon > 0$, there is an $n_\epsilon \in \mathbb{N}$ such that $d(x_a, x_b) < \epsilon$ for any $a \geq n_\epsilon, n \geq n_\epsilon$.*

Definition 10. *A metric space is said to be **complete** if every Cauchy sequence in M converges to a point in M.*

Definition 11. *A metric space is totally bounded if and only if for every real number $\epsilon > 0$, there exists a finite collection of open balls in M of radius ϵ whose union contains M.*

Intuitively, a totally bounded space means that it can be covered by finitely many subsets of every fixed size. The smaller the size fixed, the more subsets may be needed, but it will always be a finite number of subsets.

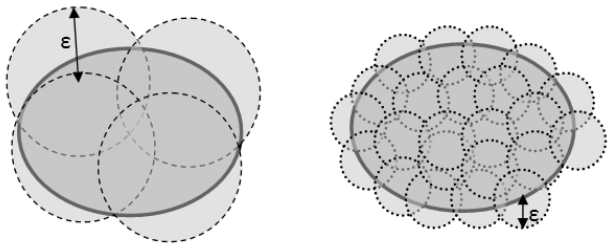


Figure 9: M can be covered with finitely many $B(x, \epsilon)$ for any $\epsilon > 0$

3.8 Monotonic Functions

Definition 12. *A function f defined on a subset of the real numbers with real values is called monotonic if and only if it is either entirely non-increasing or entirely non-decreasing.*

Definition 13. *A function is called monotonically increasing if for all x and y such that $x \leq y$ one has $f(x) \leq f(y)$.*

Definition 14. *A function is called monotonically decreasing if for all x and y, whenever $x \leq y$, then $f(x) \geq f(y)$.*

3.9 Convex Functions

Definition 15. *Let X be a convex set in a real vector space and let $f : X \to \mathbb{R}$ be a function. Then f is called convex if for all $x_1, x_2 \in X$ and for all $t \in [0, 1]$:*

$$f(tx_1 + (1-t)x_2) \leq tf(x_1) + (1-t)f(x_2)$$

A real-valued function defined on an n-dimensional interval is called convex if the line segment between any two points on the graph of the function lies above or on the graph, in a Euclidean space (or vector space), of at least two dimensions.

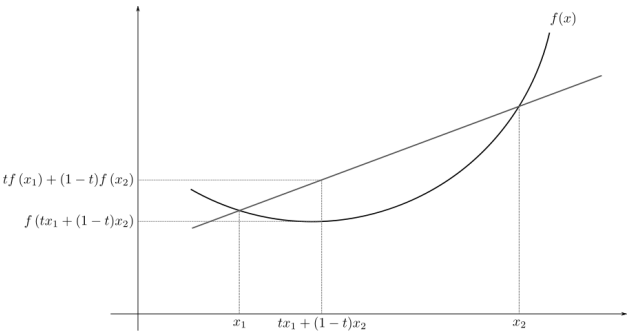


Figure 10: convex function

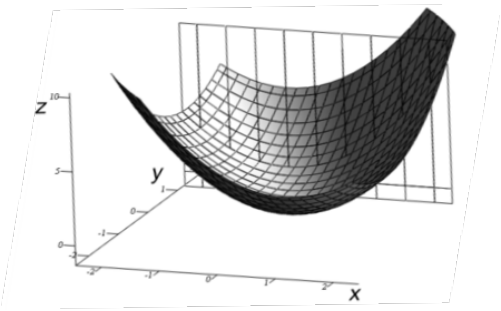


Figure 11: bivariate convex function

Convex functions have many useful properties. Here are some that are directly useful in the scope of this paper:

First, suppose f is a function of one real variable defined on an interval and let

$$R(x_1, x_2) = \frac{f(x_1)-f(x_2)}{x_1-x_2}$$

Note that $R(x_1, x_2))$ is the slope of the line in the two-dimensional plot and that R is symmetric in (x_1, x_2).

1. f is convex if and only if $R(x_1, x_2)$ is monotonically non-decreasing in x_1 for every fixed x_2 (or vice versa).

2. A convex function f of one real variable defined on some open interval C is continuous on C and Lipschitz Continuous on any closed subinterval.

3. A differentiable function of one variable is convex on an interval if and only if its derivative is monotonically non-decreasing on that interval. A function that is differentiable and convex is also continuously differentiable.

4. A differentiable function of one variable is convex on an interval if and only if the function lies above all of its tangents:

   $$f(x) \geq f(y) + f'(y)(x - y)$$

   for all x and y in the interval. In particular, if $f'(c) = 0$, then c is a global minimum of $f(x)$

5. Jensen's Inequality applies to every convex function f. If X is a random variable taking values in the domain of f, then $\mathrm{E}(f(x)) \geq f(\mathrm{E}(X))$ where E represents expectation.

3.10 Infimum and Supremum

A partially ordered set is a set in which there is a binary relation indicating that, for certain pairs of elements in the set, one of the elements precedes the other in the ordering. The word partial in the name indicates that not every pair of elements needs to be comparable.

The infimum of a subset S of a partially ordered set X is the greatest element in X that is less than or equal to all the elements of S, if such an element exists. The term *Greatest Lower Bound* is commonly used for infimum.

The supremum of a subset S of a partially ordered set X is the least element in X that is greater than or equal to all the elements of S, if such an element exists. The term *Least Upper Bound* is commonly used for supremum.

4 Statistical Modeling

Statistical Modeling is the act of creating a mathematical model that embodies a set of assumptions about some process that generates data. What makes a statistical model different from the broader class of mathematical models is that a statistical model generally relates one or more random variables. Given that our models are non-deterministic due to the use of random variables, statistical models are used to deduce properties of an underlying data generating process by creating an approximation of the process with a probability model or models.

Formally, a statistical model can be thought of as a pair (S, P) where S is the set of possible observations (the sample space), and P is a set of probability distributions on S. The set P is parameterized as such: $P = \{P_\theta : \theta \in \Theta\}$ where the set Θ defines the parameters of our model. Motivating this definition, we can assume that there is some true probability distribution(s) that model a data generating process. We can choose P to represent a set of distributions which contain a distribution that can approximate our data generating distribution. We don't necessarily need P to contain our true data generating process, and in practice this is almost a given. [8]

Some of the classes of assumptions that are made about a model include:

- Distributional Assumptions - We can make assumptions that the model follows some manner of probability distribution. We can also relate terms that involve error as having a distribution. For example, we may say a data generating process contains samples $X_i \sim \text{Cauchy}(0, 1)$ or that errors are normally distributed.
- Structural Assumptions - Model selection typically involves making a structural as-

sumption about the form of the functional relationship. For example, in linear regression we are making an assumption that the data is linear.

- Cross-Variation Assumptions - We may make the assumption that the observations or errors are statistically independent.

4.1 Maximum Likelihood

Maximum Likelihood Estimation is a method of estimating the parameters of a statistical model, given observations. The goal is to find the parameter values which maximize the likelihood function $\mathcal{L}(\theta|x)$.

As an example, suppose Worcester State had 5000 undergraduate students and we wanted to measure their typing speed. It would be unreasonable to measure the speed of all the students.

Instead we can assume that the typing speeds of the students are normally distributed with some unknown mean and variance. The mean and variance can then be estimated with MLE while only knowing the typing speeds of some of the students, a sample of the population of students. MLE would accomplish this by taking the mean and variance as parameters and finding particular parametric values such that we make the observed results the most probable given our model.

If we are fitting some data to a discrete distribution:

Let X be a discrete random variable with a probability mass function (discrete form of probability density function) p depending on a parameter θ. Then the function

$$\mathcal{L}(\theta|x) = p_\theta(x)$$

considered as a function of θ is the likelihood function of θ given the outcome x of the random variable X.

For fitting data to a continuous distribution, similarly:

Let X be a random variable following an absolutely continuous probability distribution with a density function f depending on a parameter θ. Then the function:

$$\mathcal{L}(\theta|x) = f_\theta(x).$$

Then, once we have the likelihood function, we wish to find

$$\hat{\theta} \in \{\arg\max_{\theta \in \Theta} \mathcal{L}(\theta; x)\}$$

if a maximum exists. In practice, we often opt to work with the natural logarithm of the likelihood function (called the **log-likelihood**).

$$\hat{\ell}(\theta; x) = \ln \mathcal{L}(\theta; x).$$

4.2 Sampling

Sampling is the act of gathering n observations from a population N (typically $n << N$) in order to estimate characteristics of the whole population.

For example, we may want to learn more about the salaries of workers in a country. We could take a sample of people from the population of the country and gather characteristics of each person (our observation) like name, age, salary, location, job title, etc. Then we would have a set of samples (each person), each with a set of explanatory variables (their characteristics).

Sampling plays a big part in the work to come. We often wish to take a batch or small set of samples from a larger population in order to help fit our model to some data.

We will also deal with a set of methods known as Monte Carlo Methods. These are methods in which we rely on repeated random sampling to obtain numerical results. The idea is that we can use random sampling to solve problems that may be deterministic in principle. An example would be to solve an analytically intractable integral. This is known as Monte Carlo Integration and it works by randomly choosing points at which the integrand is evaluated. By comparing those points that fall within the integrals area and those that do not, as our number of randomly placed points points increase we get a better idea of the proportion of the area within the integral and outside of the integral. [7]

4.3 Classification

It is often the case that we have a sample observation and we want to place it into some subcategory of a population. We call this subcategory a class. The properties that help us categorize an observation into a class are often called explanatory variables.

For example, say we wanted to split the US populace into those who enjoy mathematics and those who don't. Then we could create a model to distinguish the explanatory variables that are likely to lead to someone enjoying mathematics. Then, given a new person, our model

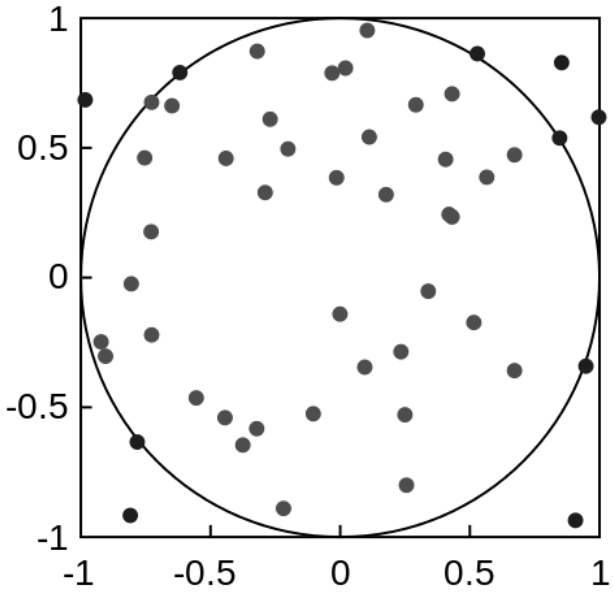


Figure 12: An example of a Monte Carlo approximation of Pi

would look at their characteristics and place them into a bin: those that like mathematics and those that do not.

4.4 Logistic Regression

Often when talking about classification, references to a logistic regression model will be made.

To begin with, observe the sigmoid function:

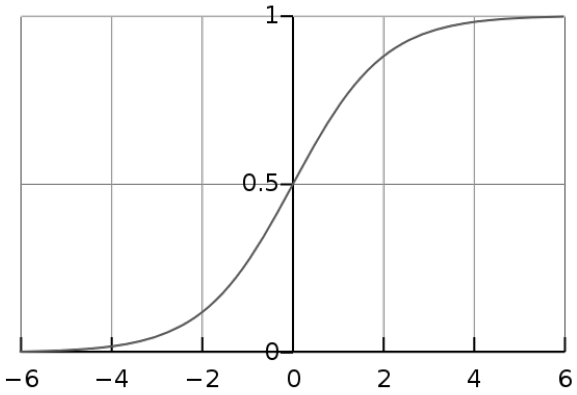


Figure 13: A Sigmoid Function

A sigmoid function is of the form:

$$S(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{e^x+1}$$

which takes any real input $x \in \mathbb{R}$ and outputs a value between 0 and 1.

Then let us assume that x is a linear function on some interval with a single explanatory variable z. In the picture above, this interval is $[-6, 6]$. Then x can be expressed in the following way:

$$x = \beta_0 + \beta_1 z$$

and we can rewrite our sigmoid function as

$$p(z) = \frac{1}{1+e^{-(\beta_0+\beta_1 z)}}$$

This equation can be interpreted as the probability of the dependent variable equaling a success ($p(z) = 1$) as opposed to a failure ($p(z) = 0$) for some case.

Now we can define the inverse of the sigmoid function, the logit (log odds):

$$g(p(z)) = \text{logit}(p(z)) = \ln(\frac{p(z)}{1-p(z)}) = \beta_0 + \beta_1 z$$

Then, after exponentiation, we get what is called our odds:

$$\frac{p(x)}{1-p(x)} = e^{\beta_0+\beta_1 z}$$

The odds of our dependent variable x equaling a case (like 0 or 1) given some linear combination z of the predictors is equivalent to the exponential function of our linear regression expression.

This shows how our logit function can serve as a "link" between the probability of a specific case and our linear function. In the original linear regression, our output variable had a range within $(-\infty, \infty)$, and now our output range is contained as a probability in $[0, 1]$.

A logistic regression model also generalizes to multiple explanatory variables easily as well. We can revise the expression

$$\beta_0 + \beta_1 z$$

to

$$\beta_0 + \beta_1 z_1 + \beta_2 z_2 + ... + \beta_n z_n = \beta_0 + \sum_{i=1}^{n} \beta_i z_i$$

4.5 Training Models

We often wish to create some mathematical model where, given an input sample $\mathbf{x}$ containing predictors $\mathbf{x} \in \{x_1, x_2, ..., x_n\}$ we want to predict some value y. Our y value could be real valued (regression) or one value from a set of possible values (classification). Suppose we have some equation of the form $y = \beta_0 + \sum_{i=1}^{n} \beta_i x_i + \epsilon_i$ where β_0 is our intercept, β_i is our coefficient for some predictor x_i, and ϵ_i is irreducible error in the relationship. This is called a linear model.

When we train a model, we are effectively choosing the best parameters such that we maximize or minimize some metric with respect to the data. An example of a commonly used metric is minimizing the sum of squared residuals $S = \sum_{i=1}^{n} r_i^2$. A residual is the difference between the actual data given value of our dependent variable and the value predicted by our model. Mathematically, for a single parameter linear model $y = \beta_0 + \beta_1 x + \epsilon$, $r_i = y_i - f(x, \beta_1)$ where r_i is our residual, y_i is our actual data value at some point x, and $f(x_i, \beta_i)$ is the value predicted by our model.

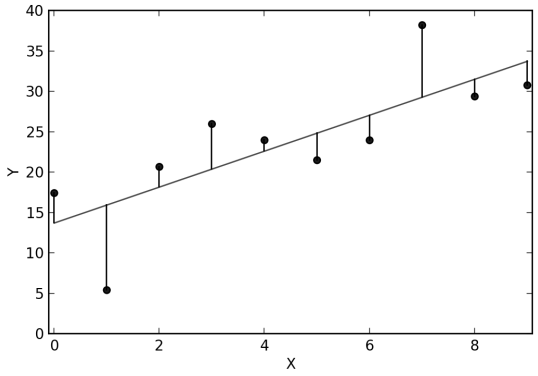


Figure 14: Residuals for a Linear Model

We may use a variety of different metrics to measure how well our model fits. We call the function that we are trying to maximize or minimize in order to fit a model our objective function. This may be called a variety of things, such as loss function, cost function, reward function, profit function, utility function, fitness function or others depending on the context in which it is applied.

The process of training a model is finding some method, typically a numerical method, to minimize or maximize our objective function. When we say that a model is learning some

metric, it is really optimizing some objective function subject to or not subject to some constraint. Then if we have a linear model like $y = \beta_0 + \sum_{i=1}^{n} \beta_i x_i + \epsilon_i$, after training it on some data we will minimize our objective function (the sum of squares of the residuals, called ordinary least squares) such that our linear model gives the best fit on our data.

5 Discriminative and Generative Models

Suppose we have a set of data consisting of n explanatory variables, one of which is categorical ($y \in \{0, 1, ..., m\}$). Each row of data would look like the following:

$$\text{Sample} = [x_1, x_2, ..., x_{n-1}, y]$$

Now suppose that we wish to train some model where, given a new sample with explanatory variables x_1 through x_{n-1} we could predict which class y it would be most likely to fall into.

5.1 Discriminative Models

A discriminative model would learn the conditional probability distribution $p(y|x)$ - the probability that our class is y given our data sample x. Therefore, we are evaluating a function of the form $f(x) = \max_y p(y|x)$. We are finding the class in which there is the highest probability of our sample x belonging to it. A model of this form is trained on some data input with marked classification boundaries y_i for each sample x. When given a new sample, it will generate an array of the probability that our sample x belongs to class y for all y_i and choose the class with the highest probability. We can think of this as modeling the boundaries between classes in high dimensional space and seeing where our new sample falls within these boundaries.

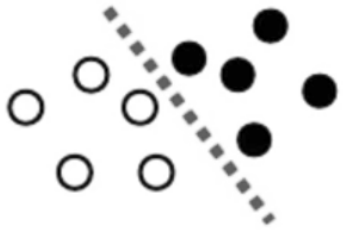


Figure 15: Class Demarcation in a Discriminative Model

5.2 Generative Models

A generative model would learn the joint probability distribution $p(x, y)$ - the probability of having our sample and our class y together. Therefore we are effectively creating a series of probability distributions for each class y. Since generative models model the joint probability $p(x, y)$, we can choose to get back a distribution rather than a single class. From that distribution we can sample (x, y) points and generative new likely data samples. This means a generative model can generate new samples.

Additionally, we can use Baye's Rule to learn the conditional probability distribution $p(y|x)$ from our joint distribution $p(x, y)$ in the following manner:

$$p(x, y) = p(x|y)p(y) = p(y|x)p(x)$$
$$p(y|x) = \frac{p(x,y)}{p(x)}.$$

This means that a generative model can effectively work as both a generative model and a discriminative model with a simple transformation. When we are training this model, we can say we are evaluating $f(x) = p(x, y) \propto \max_y p(x|y)p(y) \propto \max_y p(y|x)p(x)$.

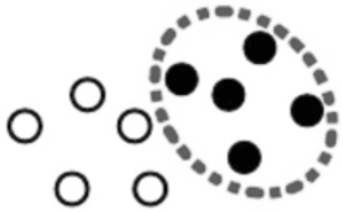


Figure 16: Class Encapsulation in a Generative Model

6 Generative Adversarial Networks

Generative Adversarial Networks were introduced in 2015 with the release of a paper by Ian Goodfellow et. al. [9] In the paper, the authors proposed a new framework for training a generative model on some data. This is done by simultaneously training two models: A generative model G which tries to model $p(x, y)$ of our data generating process, and a discriminative model D which tries to estimate the probability that our sample came from the training data as opposed to our generator.

Our generator is being trained to maximize the probability that our discriminator function will mis-classify a sample that it generates for a sample from the actual data generating process. This back and forth battle will stop when the generator produces samples that

are so good that the discriminator can not tell the difference between a sample from the generator and a sample from the original data. This back and forth between the two models is a game theoretic two player minimax game.

When we want to train using an adversarial modeling framework, it is typical that we use a type of model known as a multilayer perceptron. The Universal Approximation Theorem (discussed below) shows that a multilayer perceptron can effectively approximate any continuous function. We also use them because they are continuous, differentiable functions that are conducive to training using a technique known as backpropagation. [9]

6.1 Minimax Theorem

For every two person, zero-sum (each participant's gain or loss is exactly balanced by the losses or gains of the other participants) game with finitely many strategies, there exists a value V and a mixed strategy for each player such that

- Given player 2's strategy, the best payoff possible for player 1 is V, and
- Given player 1's strategy, the best payoff possible for player 2 is -V

Equivalently, player 1's strategy guarantees them a payoff of V regardless of player 2's strategy, and similarly player 2 can guarantee a payoff of -V.

The name minimax arises because each player minimizes the maximum payoff possible for the other. Since the game is zero-sum, they also minimize their own maximum loss (and maximize their minimum payoff).

Formally, $\overline{v_i} = \min_{a_{-i}} \max_{a_i} v_i(a_i, a_{-i})$.

6.2 An Analogy of the Generator and Discriminator

Suppose we have two people, a currency expert and a counterfeiter. The expert has a nice office job at a bank where they regularly take samples of the local currency and check for fakes. The counterfeiter also has an nice office job at the same bank, but their salary is not nearly as nice as the experts! Given that the counterfeiter fancies themselves to be rather clever, they decide to make their own fortune and begin designing and printing fake currency.

The expert is pretty good at telling the difference between real and fake money already, but has room to improve. The counterfeiter on the other hand, is completely new to making fake money and must learn attention to detail. The first time the counterfeiter tries to pass

off their work as real currency, there is a very low probability that the expert won't be able to tell. As a result, the expert finds some of the fake money in their weekly inspection batch and decides the best thing to do would be to throw a tantrum. The counterfeiter, overhearing their coworker's lament (the expert is a real loudmouth), goes home and works through some youtube tutorials on photoshop.

The next week, the counterfeiter releases another batch of fake money (after some lessons) into the local economy and it eventually ends up in front of the expert. The expert notices that the detail work has gotten better, but is still fairly certain that the currency is fake. The counterfeiter, not deterred goes home and works through more youtube photoshop tutorials. In the process of judging these fake currencies, the expert is also learning more about what to look for to tell real money from fake money.

After many iterations of judgement and training, the counterfeiter releases another batch of fake money into their local economy. This time, the expert (who is now truly deserving of the title expert), is stumped and assumes that all the money in their inspection batch is real. The game between the counterfeiter and the expert has gotten to the point where the counterfeiter is so good at making fake money that the expert can not tell real money apart from the fake money. Similarly, the counterfeiter is so good at making fake money, there is likely no better counterfeiter in the world.

Hence, they have reached an equilibrium in which the counterfeiter has minimized the chance that the expert can tell their fakes. Similarly, the expert has maximized their ability to tell the difference between real and fake currency. As a result, the counterfeiter gets obscenely wealthy and the expert loses their job for allowing so much fake currency to permeate through the local economy.

6.3 Perceptron

The adversarial network is the most straight forward to apply when the generator and the discriminator are models known as multilayer perceptrons.

A perceptron is an algorithm that allows us to learn a binary classifier (given an input, returns a class 0 or 1). It takes in a vector of real valued weights from an input, runs the weight vector through a function and gives some output $\in [0, 1]$. Graphically:

Formally, a perceptron is a function that maps an input $x \in \mathbb{R}$ to an output value $f(x)$ where

$$f(x) = \begin{cases} 1 & \text{w} \cdot x + b > 0 \\ 0 & otherwise \end{cases}$$

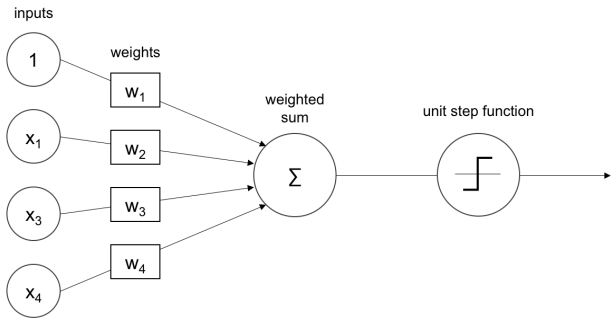


Figure 17: A Single Layer Perceptron with a Heaviside Step Activation Function

In this definition:

- w is a vector of real-valued weights
- $w \cdot x$ is the dot product $\sum_{i=1}^{m} w_i x_i$
- m is the number of inputs to the perceptron
- b is the bias.

Since $f(x) = 1$ in our main case and 0 otherwise, we are using a Heaviside Step function as our activation function, but we could use any variety of linear or nonlinear activation functions to determine our output. The activation function of our perceptron defines the output of our given set of inputs. The bias shifts the decision boundary away from the origin and does not depend on the input value.

6.4 Multilayer Perceptrons

A multilayer perceptron is a series of perceptrons that have an input, at least one hidden layer (but typically many more), and an output layer. In this overview we will focus on multilayer perceptrons with one hidden layer, but they can easily get generalized to account for more hidden layers.

Generalizing our idea of a perceptron to a multilayer perceptron, we say that a simple multilayer perceptron is a function $f : \mathbb{R}^D \mapsto \mathbb{R}^L$ where D is the input vector x size and L is the output vector $f(x)$ such that

$$f(x) = G(b^{(2)} + W^{(2)}(s(b^{(1)} + W^{(1)}x)))$$

where, for our hidden layer $h(x) = s(b^{(1)} + W^{(1)}x)$:

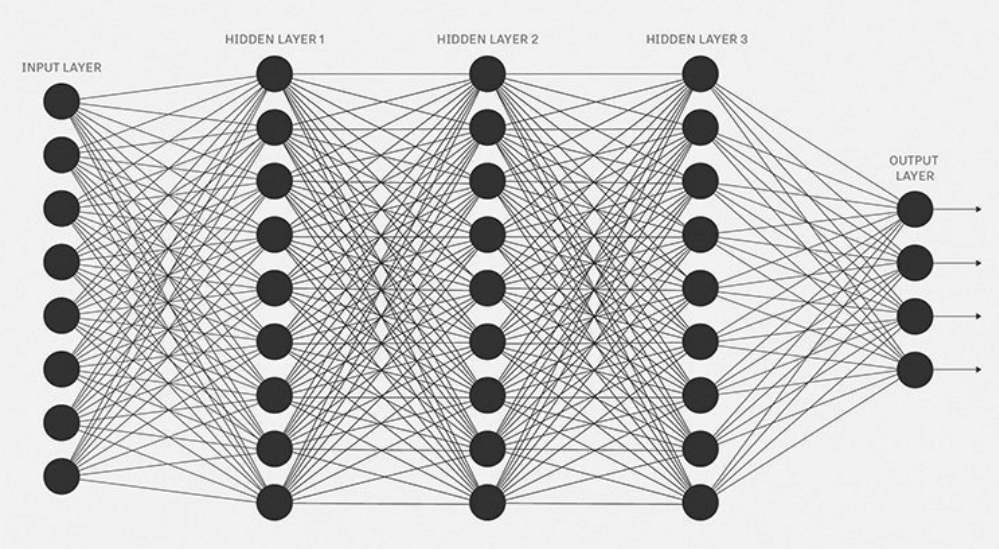


Figure 18: A Multilayer Perceptron

- $W^{(1)} \in \mathbb{R}^{D_{input} \times D_{hidden}}$ is the weight matrix that connects our input weights to our hidden layer weights.
- $b^{(1)}$ is a bias vector which dictates the shift in our decision boundaries
- s is our hidden layer activation function. Common functions include the Rectified Linear Unit (a fancy name for $s(z) = \max(0, z)$), tanh and the sigmoid function.

And, for our output layer $O(x) = G(b^{(2)} + W^{(2)}h(x))$:

- G is our choice of activation function for our output layer. In this output, we could use the sigmoid function, effectively turning our multilayer perceptron into a logistic regression classifier. In the single layer perceptron above, our output activation function was the Heaviside Step function.
- bias vector $b^{(2)}$
- weight matrix $W^{(2)}$

Multilayer perceptrons can be generalized to any depth (as many inputs, hidden layers, and outputs as needed). The field of deep learning explores applications of higher depth multilayer perceptrons. If we wish to have a classifier with multiple classes, we can simply choose a softmax function for our output activation function. This will return, for some input, the most probable class (of many classes) that it belongs to.[10]

The softmax function is a generalization of the logistic function where the output is k distinct linear functions, and the predicted probability for the j^{th} class given sample x and a weights vector W is:

$$p(y = j|x) = \frac{e^{x \cdot w_j}}{\sum_{k=1}^{K} e^{(x \cdot w_k)}}$$

6.5 Universal Approximation Theorem

The universal approximation theorem states that a multilayer perceptron with a single hidden layer containing a finite number of perceptron nodes can approximate continuous functions on compact subsets of $\mathbb{R}^n$, under mild assumptions on the activation function. Essentially, a simple multilayer perceptron can represent a variety of interesting functions when given appropriate parameters. This theorem was proved in 1989 by George Cybenko.

Theorem 6.1. *Let $\gamma(.)$ be a nonconstant, bounded, and continuous function. Let I_m denote the m-dimensional unit hypercube $[0, 1]^m$. The space of continuous functions on I_m is denoted by $C(I_m)$. Then, given any $\epsilon > 0$ and any function $f \in C(I_m)$, there exists an integer N, real constants $v_i, b_i \in \mathbb{R}$ and real vectors $w_i \in \mathbb{R}^m$ where $i = 1, ..., N$ such that we may define:*

$$F(x) = \sum_{i=1}^{N} v_i \gamma(w_i^T x + b_i)$$

as an approximate realization of the function f where f is independent of γ; that is

$$|F(x) - f(x)| < \epsilon \quad \forall x \in I_m.$$

Alternatively, we could say that functions of the form $F(x)$ are dense in $C(I_m)$. This theorem also holds when we replace I_m with any compact subset of $\mathbb{R}^m$.

In the equation above $\gamma(w_i^T x + b_i)$ represents a multilayer perceptron where $w_i^T x + b_i$ is the summation step and γ is the output activation function.

6.6 Backpropagation

Backpropagation is a method that is used in multilayer perceptrons to calculate a gradient (a multivariable generalization of a derivative) that is used to adjust the weights used in our layers. This gradient allows us to gauge how changes in weight vectors lead to changes in some error function.

The word backpropagation itself is shorthand for "Backward Propogation of Errors". This is because after each output, the error is measured, and the weights of the connections between perceptron nodes in each layer are updated accordingly. Hence, the weights are updated in relation to the error function backwards through the network.

Our error function, or loss function, is typically the square of the Euclidean distance between our data vector y and our multilayer perceptron output vector y'. This allows us to calculate the difference between the multilayer perceptron's output and its expected output according to the data, after some gradient (change in weights) has been applied. Later in the paper we will see that the error function can be generalized to any f-divergence.

The Euclidean distance can be expressed as:

$$E(y, y') = \frac{1}{2}||y - y'||^2$$

Then our partial derivative with respect to the multilayer perceptron outputs is:

$$\frac{\partial E}{\partial y'} = y' - y$$

and then we can write out error function over n training examples as the average of losses over individual examples:

$$E = \frac{1}{2n} \sum_x ||y(x) - y'(x)||^2$$

Repeated use of backpropagation allows us to train a multilayer perceptron by continually adjusting the weight vectors in a way that reduces our error function to a minimum. We are optimizing our multilayer perceptron by minimizing the error function.

We achieve this minimization with a two phase cycle: propagation and a weight update.

First we give an input vector of weights to our network, and it makes its way through each of the layers until it reaches the output layer. Then the output layer is compared to the desired output with our error function (Euclidean Distance). The resulting output layer is a vector, so we can calculate the error values for each node in the output layer. Then the error values are propagated backwards from the output, through the network, until each perceptron node has an associated error value that reflects its contribution to the output.

Then backpropagation uses these error values for each node to calculate the gradient (change in) the loss function (Euclidean Distance). This gradient is then fed to an optimization method, which in turn uses the gradient to update the weight vectors and the process repeats.

Let N be a multilayer perceptron with z connections, m inputs and n outputs. Then let:

- $x_1, x_2, ...$ denote vectors in $\mathbb{R}^m$ (our inputs)

- $y_1, y_2, ...$ denote vectors in $\mathbb{R}^n$ (our outputs)
- $w_0, w_1, w_2, ...$ denote vectors in $\mathbb{R}^z$ (our weights)

Our multilayer perceptron corresponds to a function $y = f_N(w, x)$ which, given a weight w will map an input x to some output y.

Then our function takes as input a sequence of training examples (x_1, y_1), ..., (x_p, n_p) and produces a sequence of weights $w_0, w_1, ..., w_p$ starting from some initial weight w_0 which is usually randomly generated.

Then these weights are computed one by one. First we compute w_i using (x_i, y_i, w_{i-1}) for $i = 1, ..., p$. Then our output is w_p, giving us a new function $x \mapsto f_N(w_p, x)$.

The computation is the same for each $i \in [1, p]$. To compute the weights from w_1 onwards, we consider (x_1, y_1, w_0) and apply the optimization technique gradient descent to the function $w \mapsto E(f_N(w, x_1), y_1)$ to find a local minimum starting at $w = w_0$.

6.7 Gradient Descent

Gradient Descent is an iterative optimization algorithm for finding the local minimum of a function. In order to do this, it takes a series of steps proportional to the negative of a given gradient. If we wish to find a local maximum, it could also take a series of steps proportional to the positive of a given gradient. The maximum version is called Gradient Ascent.

As the surfaces they explore can be very high dimensional with many minima and maxima, these algorithms generally converge to local minima and local maxima, making its output nondeterministic.

Intuition for this technique can be gained through an analogy.

Suppose a hiker is stuck on a mountain and is trying to get down. There is a lot of fog, so they can't really see which direction leads to the bottom. As a result, they must use locally available information to find their way down.

Using the method of gradient descent, our hiker will look at the steepness of the hill around them, and move in the direction of the steepest downwards descent. Alternatively, if they wanted to find a peak, they could move in the direction of steepest ascent. Using the descent method, our hiker would eventually find their way down the mountain.

In this analogy, our hiker is the gradient descent algorithm, and the path taken down the mountain is a sequence of parameter settings that our algorithm explores. The steepness of

the hill represents the slope of the error surface at that point. The direction the hiker goes aligns with the gradient of the error surface at that point.

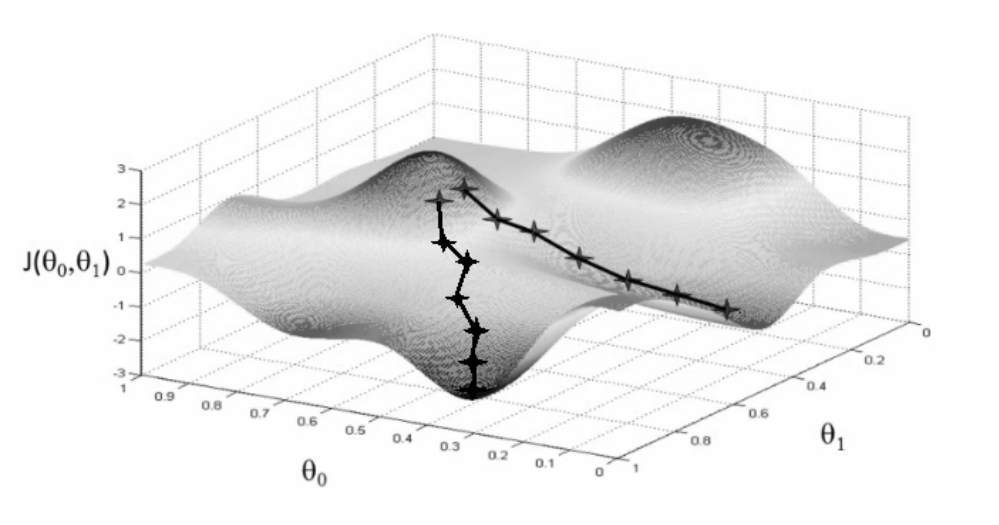


Figure 19: Gradient Descent on a Surface

Gradient Descent is based on the observation that if some multi-variable function $F(\vec{x})$ is defined and differentiable in a neighborhood of a point a, then $F(\vec{x})$ decreases fastest if one goes from a in the direction of the negative gradient of F at a, $-\nabla F(a)$.

It follows that, if

$$a_{n+1} = a_n - \gamma \nabla F(a_n) \quad \text{for } \gamma \in \mathbb{R}^+ \text{ small enough}$$

then $F(a_n) \geq F(a_{n+1})$. In other words, the term $\gamma \nabla F(a)$ is subtracted from a because we want to move against the gradient towards the minimum of the function.

With this in mind, we can guess x_0 for a local minimum of F, and consider the sequence $x_0, x_1, x_2, ...$ such that

$$x_{n+1} = x_n - \gamma_n \nabla F(x_n), n \geq 0.$$

Then we have a monotonic sequence $F(x_0) \geq F(x_1) \geq F(x_2) \geq ...$, and our sequence x_n should converge to a local minimum.

Convergence to a local minimum can be guaranteed if we use the Barzilai-Borwein Method in which

$$\gamma_n = \frac{(x_n - x_{n-1})^T [\nabla F(x_n) - \nabla F(x_{n-1})]}{||\nabla F(x_n) - \nabla F(x_{n-1})||^2}$$

or if there are other assumptions on the function F such as being convex or ∇F being Lipschitz Continuous.

In Gradient Descent we are determining $x_{n+1} = x_n - \gamma_n \nabla F(x_n)$ for $x_0, x_1, x_2...$ until our algorithm converges. Since $\nabla F(x_n)$ is calculated based on the complete training set, we would need to make a pass through our training set for every step of gradient descent. This can be excessively computationally costly.

In stochastic gradient descent, we take small batches of our training samples and perform $\nabla F(x_n)$ on the sample. Then for the next update we take another random sample of our training data. The use of small batches cuts down on the computational power needed to run the algorithm, and as a result most problems that use gradient descent in practice use a stochastic variant.

6.8 Adversarial Networks

Now that we have seen the minimax theorem, multilayer perceptrons and how backpropagation and gradient descent allow us to train our model, we can learn how the adversarial nets framework works.

Let us define G as a differentiable function represented by a multilayer perceptron with parameters $\theta_g = \{W_g^{(2)}, b_g^{(2)}, W_g^{(1)}, b_g^{(1)}\}$, and similarly D as a differentiable function represented by a multilayer perceptron with parameters $\theta_d = \{W_d^{(2)}, b_d^{(2)}, W_d^{(1)}, b_d^{(1)}\}$. In this case, G is our generator and D is our discriminator.

To learn the generator's probability distribution p_g over data x, we first define a prior input noise variable $p_z(z)$. We then represent a mapping to data space as $G(z, \theta_g)$.

We also have our second multilayer perceptron $D(x, \theta_d)$ which will output a single scalar: the probability that x came from the data rather that p_g.

We will train D to maximize the probability of assigning the correct label to both training examples and samples from our generator.

Simultaneously, we will also train G to minimize $\log(1 - D(G(z)))$.

Therefore, we can place our "dueling" multilayer perceptrons into our two player minimax value function $V(G, D)$:

$$\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))]$$

In the value function above, $\mathbb{E}$ denotes an equilibrium.

Our generator G implicitly defines a probability distribution p_g as the distribution of the samples $G(z)$ obtained when $z \sim p_z$. As our minimax function is converging to an equilibrium, $G(z)$ converges to a good estimator of p_{data}, given enough capacity and training time.

6.9 Algorithm: Stochastic Gradient Descent Training of Generative Adversarial Nets

Let k be the number of steps applied to the discriminator, with $k = 1$ the default. Then:

For number of training iterations, do:

- For k steps, do:
  - Sample minibatch of m noise samples $\{z_1, ..., z_m\}$ from noise prior $p_g(z)$
  - Sample minibatch of m training samples $\{x_1, ..., x_m\}$ from data generating distribution $p_{data}(x)$
  - Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} [\log(D(x_i)) + \log(1 - D(G(z_i)))]$$

  - end for k steps
- Sample minibatch of m noise samples $\{z_1, ..., z_m\}$ from noise prior p_z
- Update the generator by descending it's stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log(1 - D(G(z_i)))$$

- End for number of training iterations

Now we are going to show that this minimax game as a global optimum where $p_g = p_{data}$. Using this, we will be able to prove that the algorithm optimizes our minimax equation.

Proposition 1. *For a fixed G, the optimal discriminator D is:*

$$D^*_G(x) = \frac{p_{data}(x)}{p_{data}(x) + p_g(x)}.$$

Proof. The training criterion for the discriminator D, given any generator G, is to maximize our value function $V(G, D)$.

Then

$$\begin{aligned} V(G,D) &= \int_x p_{data}(x)\log(D(x))dx + \int_z p_z(z)\log(1-D(G(z)))dz && (4)\\ &= \int_x p_{data}(x)\log(D(x)) + p_g(x)\log(1-D(x))dx && (5)\end{aligned}$$

For any $(a, b) \in \mathbb{R}^2\backslash\{0, 0\}$, the function $y \mapsto a\log(y) + b\log(1-y)$ achieves its maximum in $[0, 1]$ at $\frac{a}{a+b}$. The discriminator does not need to be defined outside of $\sup(p_{data}) \cap \sup p_g$.

□

We can now reformulate our minimax function. Since our training criterion for D is maximizing our value function:

$$C(G) = \max(V(G, D)).$$

Note that the training objective for D can be interpreted as maximizing the log-likelihood for the conditional probability $P(y|x)$ where y indicates whether x comes from p_{data} ($y = 1$) or from p_g ($y = 0$). Then we can formulate our minimax value function as follows:

$$\begin{aligned} C(G) &= \mathbb{E}_{x\sim p_{data}}[\log D^*_G(x)] + \mathbb{E}_{z\sim p_z}[\log(1-D^*_G(G(z)))] && (6)\\ &= \mathbb{E}_{x\sim p_{data}}[\log D^*_G(x)] + \mathbb{E}_{z\sim p_g}[\log(1-D^*_G(x))] && (7)\\ &= \mathbb{E}_{x\sim p_{data}}[\log(\frac{p_{data}(x)}{p_{data}(x)+p_g(x)})] + \mathbb{E}_{z\sim p_g}[\log(1-\frac{p_{data}(x)}{p_{data}(x)+p_g(x)})] && (8)\\ &= \mathbb{E}_{x\sim p_{data}}[\log(\frac{p_{data}(x)}{p_{data}(x)+p_g(x)})] + \mathbb{E}_{z\sim p_g}[\log(\frac{p_g(x)}{p_{data}(x)+p_g(x)})] && (9)\end{aligned}$$

Now that we have our criteria for our generator in a more useful form:

Theorem 6.2. *The global minimum of the virtual training criterion C(G) is achieved if and only if $p_g = p_{data}$. At that point, C(G) achieves the value $-\log(4)$.*

For the proof of Theorem 1 we need some background knowledge.

6.10 Kullback-Leibler Divergence

The Kullback-Leibler Divergence (also called relative entropy) is a measure of how one probability distribution is different from a second, reference probability distribution. A Kullack-Leibler divergence of 0 indicates that the two distributions are identical, and $KL \geq 0$ for all distributions P, Q.

In the language of Bayesian Inference, $D_{KL}(P||Q)$ is a measure of the information gained when one revises one's beliefs from the prior distribution Q to the posterior probability distribution P. In this case, $Q = p_g$ and $P = p_{data}$. In other terms, it is the amount of information lost when Q is used to approximate P.

For Discrete Distributions:

$$D_{KL}(P||Q) = -\sum_i P(i)\log(\tfrac{Q(i)}{P(i)}) = \sum_i P(i)\log(\tfrac{P(i)}{Q(i)}).$$

For Continuous Distributions:

$$D_{KL}(P||Q) = \textstyle\int_{-\infty}^{\infty} p(x)\log(\tfrac{p(x)}{q(x)})dx.$$

It is the expectation of the logarithmic difference between the probabilities P and Q, where the expectation is taken using the probabilities P. The Kullback-Leibler Divergence has its origins in Information theory, and has a different formulation:

$$\begin{aligned} D_{KL} &= \sum_x p(x)\log(p(x)) - \sum_x p(x)\log(q(x)) && (11)\\ &= H(P,Q) - H(P) && (12)\end{aligned}$$

where $H(P,Q)$ is the cross entropy of P and Q, and $H(P)$ is the entropy of P.

The Kullback-Leibler Divergence does not obey the triangle inequality, nor is it symmetric ($D_{KL}(P||Q) \neq D_{KL}(Q||P)$). Therefore, it is placed in a class of measures known as Bregman Divergences.

We can see this asymmetry more clearly in figure 20.

In the top section of figure 20, we see that the difference between the area integrated for $p(x)$ and $q(x)$ is not symmetric while $p(x) \neq q(x)$.

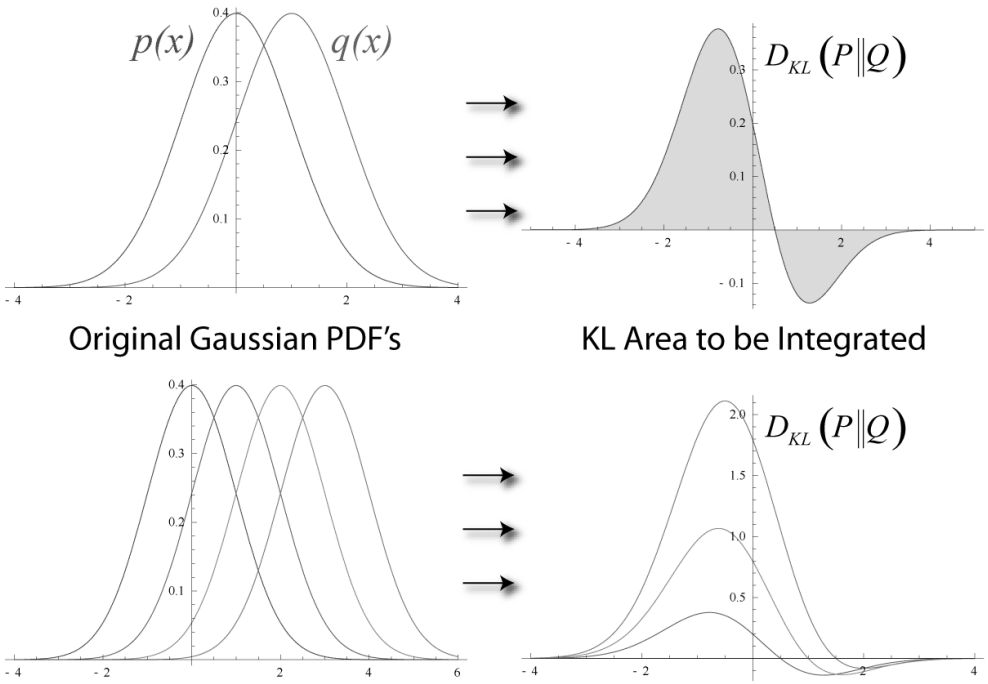


Figure 20: Visual Intuition for KL Divergence

In the bottom section of figure 20, we can see the area computed as a comparison between the first line and the others. As distance between our data distribution (the first line) and our reference distribution increases, our area of integration increases. As distance decreases, our area similarly decreases.

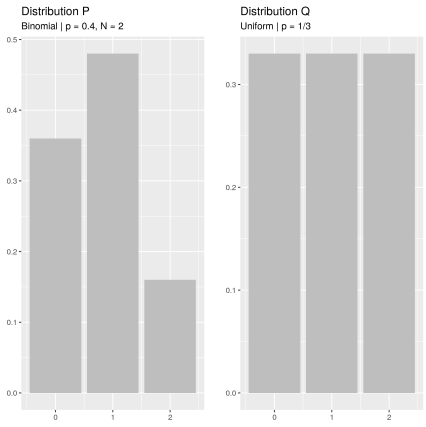
6.11 Example of Kullback-Leibler Divergence

Suppose we have 2 probability distributions. Let $P \sim \text{Binomial}(n, p)$ with $n = 2, p = 0.4$, and let $Q \sim \text{DiscreteUniform}(0, 1)$ with 3 possible outcomes, $x = 0, 1, 2$, each with probability $p = \frac{1}{3}$.

Then we would have a plot like this:

Sample	0	1	2
Distribution P	0.36	0.48	0.16
Distribution Q	0.33	0.33	0.33

We would calculate the Kullback-Leibler Divergence as follows:



$$
\begin{aligned}
D_{KL}(Q||P) &= \sum_i Q(i) \ln(\frac{Q(i)}{P(i)}) \\
&= 0.33 \ln(\frac{0.33}{0.36}) + 0.33 \ln(\frac{0.33}{0.48}) + 0.33 \ln(\frac{0.33}{0.16}) \\
&= -0.02596 + (-0.12176) + 0.24408 = 0.09637 \quad \text{nats}
\end{aligned}
$$

where a nat is a unit of information, and as nat $\to 0$, $P \to Q$.

6.12 Jensen-Shannon Divergence

The Jensen-Shannon Divergence is an extension of Kullback-Leibler Divergence, with the difference that it is symmetric and always finite. We can think of it as a symmetrized and smoothed version of the Kullback-Leibler Divergence, or an average of sorts.

$$JSD(P||Q) = \tfrac{1}{2} D(P||M) + \tfrac{1}{2} D(Q||M)$$

Where $M = \frac{1}{2}(P + Q)$ and $D(P||M)$ is the Kullback-Leibler Divergence.

Generalizing to n probability distributions:

$$JSD_{\pi_1,\pi_2,...,\pi_n}(P_1, P_2, ..., P_n) = H(\sum_{i=1}^{n} \pi_i P_i) - \sum_{i=1}^{n} \pi_i H(P_i)$$

Now we should have the required information for the proof of theorem 1.

6.13 Theorem 1 Proof

Proof. For $p_g = p_{data}$, $D^*_G(x) = \frac{1}{2}$ by our proposition ($D^*_G(x) = \frac{p_{data}(x)}{p_{data}(x)+p_g(x)}$). Then, by our Criteria for G, $C(G)$ we can plug in $D^*_G(x) = \frac{1}{2}$. Then:

$$\begin{aligned} C(G) &= \mathbb{E}_{x\sim p_{data}}[\log D^*_G(x)] + \mathbb{E}_{z\sim p_g}[\log(1 - D^*_G(x))] && (13)\\ &= \mathbb{E}_{x\sim p_{data}}[\log(\frac{1}{2})] + \mathbb{E}_{z\sim p_g}[\log(\frac{1}{2})] && (14)\\ &= \log(\frac{1}{2}) + \log(\frac{1}{2}) = -\log(4) && (15) \end{aligned}$$

To see that this is the best possible value of $C(G)$ (global minimum), reached only for $p_g = p_{data}$, observe that subtracting this expression from $C(G) = V(D^*_G, G)$, we get:

$$\begin{aligned} C(G) &= -\log(4) + KL(p_{data}||\frac{p_{data} + p_g}{2}) + KL(p_g||\frac{p_{data} + p_g}{2}) && (16)\\ &= -\log(4) + 2 * JSD(p_{data}||p_g) && (17) \end{aligned}$$

Where KL is the Kullback-Leibler Divergence and JSD is the Jensen-Shannon Divergence.

Since the Jensen-Shannon Divergence between two distributions is always non-negative and 0 only when $p_{data} = p_g$ (since these properties hold for Kullback-Leibler Divergences, they hold for Jensen-Shannon Divergences), we have shown that $C^* = -\log(4)$ is the global minimum of $C(G)$ and the only solution is $p_g = p_{data}$, when the generative model perfectly replicates the data generating process.

□

Now that we have the theoretical guarantee that $\min(C(G))$ occurs when $p_g = p_{data}$, we must check that our algorithm will converge; that $p_g \to p_{data}$.

6.14 Convergence of the Generative Adversarial Network Training Algorithm

A subderivative of a convex function $f : I \to \mathbb{R}$ at a point x_0 in the open interval I is a real number c such that

$$f(x) - f(x_0) \geq c(x - x_0)$$

for all $x \in I$. The subderivative is used to generalize the derivative to convex functions which are not necessarily differentiable.

The set $[a, b]$ of all subderivatives is called the subdifferential of the function f at x_0.

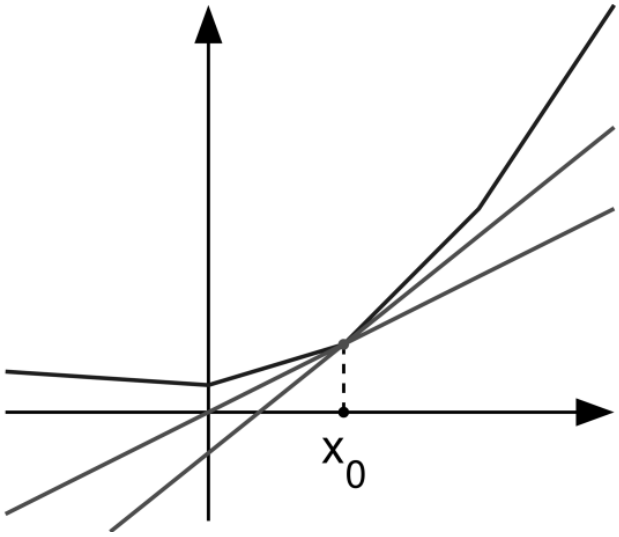


Figure 21: A Convex function with Subtangent Lines

The subdifferential of a supremum of convex functions include the derivative of the function at the point were the maximum is attained.

Proposition 2. *If G and D have enough capacity, and at each step of our algorithm, the discriminator is allowed to reach its optimum given G, and p_g is updated so as to improve the criterion*

$$\mathbb{E}_{x \sim p_{data}}[\log D_G^*(x)] + \mathbb{E}_{z \sim p_g}[\log(1 - D_G^*(x))]$$

then p_g converges to p_{data}.

Proof. Consider $V(G, D) = U(p_g, D)$ as a function of p_g as done in the above criterion. Note that $U(p_g, D)$ is convex in p_g.

Therefore, if $f(x) = \sup_{\alpha \in A} f_\alpha(x)$ and $f_\alpha(x)$ is convex in x for every α, then $\partial f_\beta(x) \in \partial f$ if $\beta = \arg\sup_{\alpha \in A} f_\alpha(x)$. This is equivalent to computing a gradient descent update for p_g at the optimal D given the corresponding G. Then $\sup_D U(p_g, D)$ is convex in p_g with a unique global optima as proven in theorem 1, and so with sufficiently small updates of p_g, p_g converges to p_{data}, concluding the proof. □

7 The Wasserstein Generative Adversarial Network

Now that we have seen generative adversarial networks, we know that they seek to minimize the Jensen-Shannon Divergence between two distributions, Q our generated approximation and P, our true data generating distribution. [11]

Unfortunately, there are some problems when training Generative Adversarial Networks.

- There is no explicit representation of $p_g(x)$.
- It's difficult to achieve equilibrium.
  - D must be synchronized well with G during training. In particular, G must not be trained too much without updating D.
  - This leads to GANs requiring lots of computational power and long training times.
- Our Jensen-Shannon Divergence fails when our distributions are disjoint.
- They are prone to mode collapse.
  - If G is trained too much without updating D, we may end up in a situation in which G memorizes some training examples or generates a small set of samples that always pass the discriminator's check. This would lead to very limited output in the generated samples.
  - We can consider this as a form of the generator being overtrained, and therefore not generalizing well with new outputs.

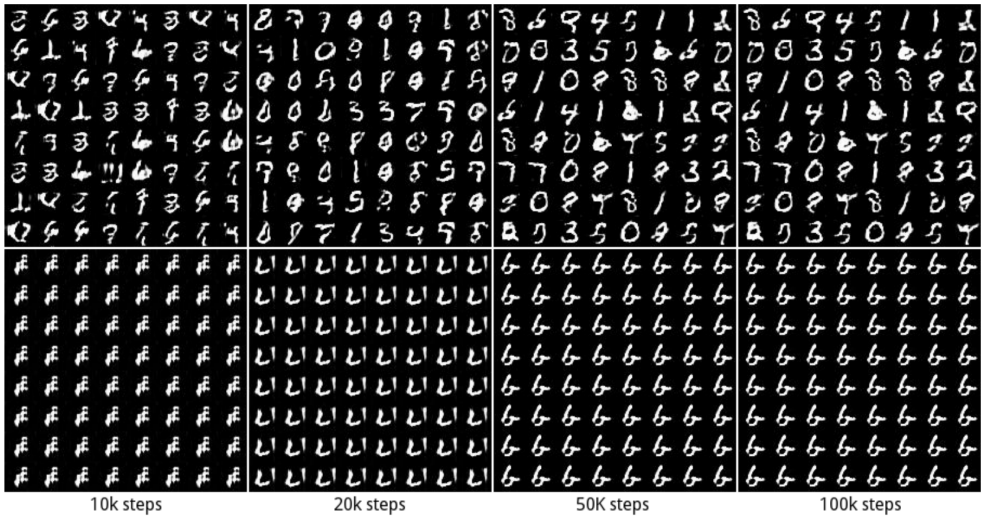


Figure 22: Mode Collapse on handwritten digits data

An extensible part of the generative adversarial network is that we could use any f-divergence as our objective function. When we swap out our Kullback-Leibler Divergence for the Wasserstein Divergence (also called Earth-Mover Distance), we help alleviate the last 3 problems stated above.

7.1 f-divergences

An f-divergence is a function $D_f(P||Q)$ that measures the difference between two probability distributions P and Q.

Let P and Q be two probability distributions over a space Ω such that Q is absolutely continuous with respect to P. Then, for a convex function f such that $f(1) = 0$, the f-divergence of Q from P is defined as:

$$D_f(P||Q) \equiv \int_\Omega f(\tfrac{dP}{dQ})dQ.$$

If P and Q are both absolutely continuous with respect to a reference distribution μ on Ω then their probability densities p and q satisfy $dP = pd\mu$ and $dQ = qd\mu$. In this case, the f-divergence can be written as:

$$D_f(P||Q) = \int_\Omega f(\tfrac{p(x)}{q(x)})q(x)d\mu(x)$$

The Kullback-Leibler Divergence and many other common divergences are special cases of f-divergences, coinciding with a particular choice of f. In the case of the KL-divergence, our corresponding $f(t) = t \log t$. We can see this below:

$$D_f(P||Q) = \int_\Omega f(\tfrac{p(x)}{q(x)})q(x)d\mu(x)$$

$$t = \tfrac{p(x)}{q(x)}, d\mu(x) = dx, \Omega \in [-\infty, \infty]$$

$$\begin{aligned} D_{KL}(P||Q) &= \int_\Omega \frac{p(x)}{q(x)} \log \frac{p(x)}{q(x)} q(x)dx && (18) \\ &= \int_\Omega q(x)\frac{p(x)}{q(x)} \log \frac{p(x)}{q(x)} dx && (19) \\ &= \int_{-\infty}^{\infty} p(x) \log(\frac{p(x)}{q(x)})dx \end{aligned}$$

(20)

7.2 Earth-Mover Distance

We can think of a probability distribution as being defined by how much mass is accumulated at a specific point. Imagine we have some distribution Q and we want to move the mass around in order to turn it into some distribution P. Then, moving mass m by distance d takes $m * d$ effort. The Earth-Mover distance is the minimum of the effort we need.

$$W(P||Q) = \inf_{\gamma \in \Pi(P,Q)} \mathbb{E}_{(x,y)\sim\gamma}[||x - y||]$$

where $\Pi(P, Q)$ denotes the set of all joint distributions $\gamma(x, y)$ whose marginals are respectively P and Q.

Intuitively, $\gamma(x, y)$ indicates how much "mass" must be transported from x to y in order to transform the distribution Q into the distribution P. This is akin to a transportation plan. Then when we take the infimum over $\Pi(P, Q)$ we are taking the minimum work transportation plan for all x, y to move $\gamma(x, y)$ mass from location x to location y.

For this plan to transform Q into P to work, we need a couple properties:

- The amount of mass that leaves x is $\int_y \gamma(x, y)dy$. This must equal $Q(x)$, the amount of mass originally at x.
- The amount of mass that enters y is $\int_x \gamma(x, y)dx$. This must equal $P(y)$, the amount of mass that ends up at y.

This shows why the marginals of $\gamma \in \Pi$ must be Q and P. For scoring this effort, we get the following:

$$\int_x \int_y \gamma(x, y)||x - y||dydx = \mathbb{E}_{(x,y)\sim\gamma}[||x - y||].$$

Computing the infimum of this over all valid γ gives the earth mover distance.

7.3 Earth-Mover Example

Why should we care about this distance metric?

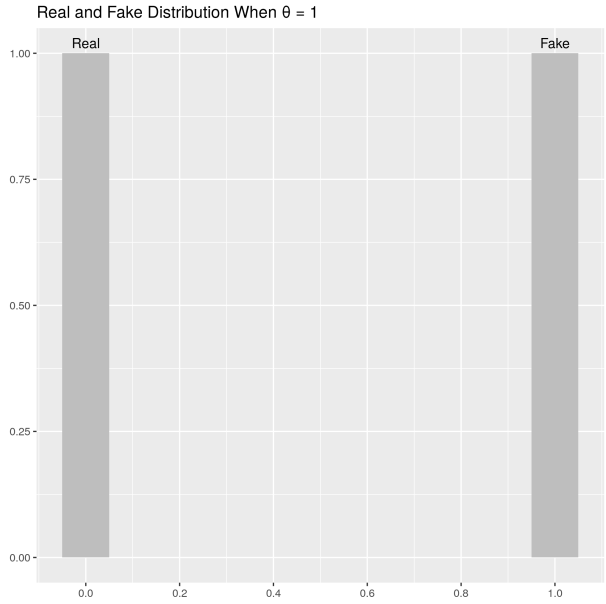


Figure 23: Earth-Mover Example — Two Disjoint Distributions

Consider two probability distributions defined over $\mathbb{R}^2$, each with a mass of 1. Let the true data distribution P_0 be defined $(0, y)$ where $y \sim \text{Uniform}[0, 1]$. Consider a false distribution P_θ where $P_\theta = (\theta, y)$ and likewise $y \sim \text{Uniform}[0, 1]$.

We would like our generative adversarial net to learn to move θ to 0. Intuitively, as $\theta \to 0$, we expect the distance $d(P_0, P_\theta)$ to decrease. Unfortunately, this isn't the case, since our distributions are disjoint.[12]

Recall Kullback-Leibler Divergence:

$$D_{KL}(P||Q) = \int_{(x,y)} P(x, y) \log(\tfrac{P(x,y)}{Q(x,y)}) dy dx$$

when there is a point (x, y) where $P(x, y) > 0$ and $Q(x, y) = 0$, $KL(P||Q) = +\infty$.

For $KL(P_0||P_\theta)$ this is true at the point $(\theta, 0.5)$. For $KL(P_\theta||P_0)$ this is true at the point $(0, 0.5)$.

Then

$$KL(P_0||P_\theta) = KL(P_\theta||P_0) = \begin{cases} +\infty & if \theta \neq 0, \\ 0 & if \theta = 0 \end{cases}$$

Recall Jensen-Shannon Divergence:

$$JSD(P||Q) = \tfrac{1}{2}D(P||M) + \tfrac{1}{2}D(Q||M)$$

where $M = \frac{1}{2}(P+Q)$ and $D(P||M)$ is the Kullback-Leibler Divergence.

Consider $M = \frac{1}{2}(P_0 + P_\theta)$. Looking at one of the KL terms:

$$KL(P_0||M) = \textstyle\int_{(x,y)} P_0(x,y) \log \frac{P_0(x,y)}{M(x,y)} dydx.$$

For any x, y where $P_0(x,y) \neq 0$, $M(x,y) = \frac{1}{2}P_0(x,y)$. As a result, our integral becomes:

$$\begin{aligned} KL(P_0||M) &= \int_{(x,y)} P_0(x,y) \log 2 dydx & (21) \\ &= \log 2 \int_{(x,y)} P_0(x,y) dydx & (22) \\ &= \log 2 & (23) \end{aligned}$$

by definition of a probability density function.

Without loss of generality, the same is true for $KL(P_\theta||M)$, so the JS Divergence is

$$JSD(P_0, P_\theta) = \begin{cases} \log 2 & if \theta \neq 0, \\ 0 & if \theta = 0, \end{cases}$$

Now, for the Earth Mover Distance, we can see that the two distributions are translations of one another. Therefore, the best way to transport the mass is to move it in a straight line from $(0, y)$ to (θ, y). This gives us $W(P_0, P_\theta) = |\theta|$.

This example is useful because it shows that there are some families of distributions in which our distance metrics fail us. The Kullback Leibler Divergence blew up to $+\infty$. The Jensen Shannon Divergence gave us a false answer. The earthmover metric handled this problem with grace.

7.4 Intractability of the Earth-Mover Distance

All that was fun, but there is a problem. The infimum in our Earth-Mover distance is highly intractable. Thankfully, a result from the Kantorovich-Rubinstein duality shows that our definition:

$$W(P||Q) = \inf_{\gamma \in \Pi(P,Q)} \mathbb{E}_{(x,y)\sim\gamma}[||x - y||]$$

is equivalent to the following:

$$W(P||Q) = \sup_{||f||_L \leq 1} \mathbb{E}_{x\sim P}[f(x)] - \mathbb{E}_{x\sim Q}[f(x)]$$

where the supremum is taken over all 1-Lipschitz functions. As review from the measure theory section, a 1-Lipschitz function's slope never exceeds 1. The derivation of the Wasserstein distance from the Kantorovich-Rubinstein duality is involved and requires some knowledge of linear programming. You can see a full derivation here [13].

If we were to replace the supremum over 1-Lipschitz functions with the supremum over K-Lipschitz functions, then the supremum is $K\dot{W}(P||Q)$. This works because every K-Lipschitz function is a 1-Lipschitz function if we divide it by K, and the Wasserstein objective function is linear. Now we are at a point in which our objective function is still intractable, but it is now easier to approximate.

Suppose we have a parameterized function family $\{f_w\}_{w\in W}$, where w are the weights and W is the set of all possible weights. Also suppose that these functions are all K-Lipschitz for some K. Then:

$$\max_{w\in W} \mathbb{E}_{x\sim P}[f_w(x)] - \mathbb{E}_{x\sim Q}[f_w(x)]$$

$$\leq \sup_{||f||_L \leq K} \mathbb{E}_{x\sim P}[f(x)] - \mathbb{E}_{x\sim Q}[f(x)]$$

$$= K \cdot W(P, Q).$$

If $\{f_w\}_{w\in W}$ contains the true supremum among K-Lipschitz functions, this gives us the distance exactly. This is unlikely though, and in the case of our true supremum not being present our approximation's quality depends on what K-Lipschitz functions are missing from $\{f_w\}_{w\in W}$.

7.5 Implementation

Now that we have a good (enough) approximation, we can return to our generative model. Suppose we want to train $Q_\theta = G_\theta(Z)$ to match P. Given a fixed G_θ, we can compute the optimal f_w for the Wasserstein distance. We can then backpropagate through $W(P||G_\theta(Z))$ to get our gradient for θ:

$$\begin{aligned} \nabla_\theta W(P||Q_\theta) &= \nabla_\theta(\mathbb{E}_{x\sim P}[f_w(x)] - \mathbb{E}_{z\sim Z}[f_w(G_\theta(Z))]) && (24) \\ &= -\mathbb{E}_{z\sim Z}[\nabla_\theta f_w(G_\theta(Z))]. && (25) \end{aligned}$$

Now we have all the ingredients for a general training process:

- For a fixed θ, compute an approximation of $W(P||Q)$ by training f_w to convergence.
- Once an optimal f_w is found, we compute the θ gradient $-\mathbb{E}_{z\sim Z}[\nabla_\theta f_w(G_\theta(z))]$ by sampling several $z \sim Z$.
- Update θ and repeat.

This process only works when our function family $\{f_w\}_{w\in W}$ is K-Lipschitz. In order to guarantee this in the algorithm, we will use weight clipping. Each weight w is constrained to lie within $[-c, c]$ by clipping w after each update to w.

7.6 Algorithm : Wasserstein Generative Adversarial Network

In the algorithm below, we will consider our discriminator by the name critic.

Given the following:

- η : the learning rate (rate at which we update the gradient)
- c : the clipping parameter
- m : the batch size
- n_{critic} : the number of iterations of the critic per generator iteration
- w_0 : initial critic parameters
- θ_0 initial generator parameters

- RMSProp is our gradient descent algorithm

Then our algorithm is:

- While θ has not converged, do the following:
  - for $t = 0, ..., n_{critic}$ do the following:
    - Sample $\{x_i\}_{i=1}^m \sim P$, a batch from the real data
    - Sample $\{z_i\}_{i=1}^m \sim p(z)$, a batch of prior samples
    - $G_w = \nabla_w [\frac{1}{m} \sum_{i=1}^{m} f_w(x_i) - \frac{1}{m} \sum_{i=1}^{m} f_w(G_\theta(z_i))]$
    - $w = w + \eta * \text{RMSProp}(w, G_w)$
    - $w = clip(w, -c, c)$
  - end for
  - Sample $\{z_i\}_{i=1}^m \sim p(z)$ a batch of prior samples
  - $G_\theta = -\nabla_\theta \frac{1}{m} \sum_{i=1}^{m} f_w(G_\theta(z_i))$
  - $\theta = \theta - \eta * \text{RMSProp}(\theta, \text{G}_\theta)$
- end while

7.7 Comparison with Classical GANs

In classical Generative Adversarial Networks, our discriminator maximizes

$$\frac{1}{m} \sum_{i=1}^{m} \log(D(x_i)) + \frac{1}{m} \sum_{i=1}^{m} \log(1 - D(G_\theta(z_i)))$$

and we constrained $D(x)$ to be a probability $p \in (0, 1)$.

In Wasserstein Generative Adversarial Networks, nothing constrains f_w to output a probability. This is why the authors call f_w the critic instead of the discriminator. The loss is directly proportional to the quality of the generated outputs. The critic will assign large values to real data and small values to fake data. So if the value assigned by the critic

to the data produced by the generator is higher then it means the generator is producing high-quality outputs.

The classical Generative Adversarial Network paper shows that in the limit, the maximimum of the objective above is the Jensen-Shannon Divergence. In Wasserstein Generative Adversarial Networks, we get the Earth-Mover distance instead.

In practice with classical Generative Adversarial Networks we never actually train the generator G to convergence. Generally, the discriminator D is too strong and we need to alternate gradient updates between G and D to get reasonable generator updates.

In contrast, because the Earth-Mover distance is differentiable nearly everywhere, we can train f_w completely to convergence before each generator update. This gets us as accurate an estimate of $W(P||Q)$ as possible, which in turn gets us the most accurate gradient $\nabla_\theta W(P||Q)$ as well.

This accurate gradient helps us avoid mode collapse. A common problem leading to mode collapse is called the vanishing gradient problem. In training a neural network, when we go to update the training weights, sometimes the gradient is vanishingly small. This effectively prevents the weight from changing its value. This may lead to the generator getting stuck providing the same samples, which, if combined with a passing score from the discriminator, can lead to mode collapse. The accurate gradient also prevents the discriminator from getting too successful at telling the real and generated data apart, which would in turn make the gradient smaller and lead to less successful updating.

Therefore, our Wasserstein distance metric helps us avoid some of the main problems of classical Generative Adversarial Networks, namely convergence to an equilibrium, mode collapse, vanishing gradients, and dealing with disjoint distributions.

8 Bayesian Generative Adversarial Networks

When we train a generator, we are essentially approximating some high-dimensional distribution to mimic the real data generating distribution. This generator distribution can be multimodal and contain many peaks and valleys. In classical Generative Adversarial Networks, our training method is based on minimax optimization, and as a result it always estimates the whole posterior distribution over the network weights as a point mass centered on a single mode. This gives us one final generator as a point estimate (the maximum of the posterior distribution or the maximum likelihood). [14]

As a result of this, when we train our model, we can expect samples from our generator to be overly compact relative to the samples from our data generating distribution. This is because our generator is creating samples that contain characteristics that are the most

likely to occur in the data. The actual data itself contains more variety than that though, so when we use a point estimate we are constraining our possible outputs.

This doesn't have to be the case though. If we don't constrict our posterior to the most likely mode, each mode of a multimodal distribution becomes its own generator with its own meaningful interpretation. By fully representing the posterior distribution over the parameters of both the generator and discriminator, we can more accurately model the true data generating distribution.

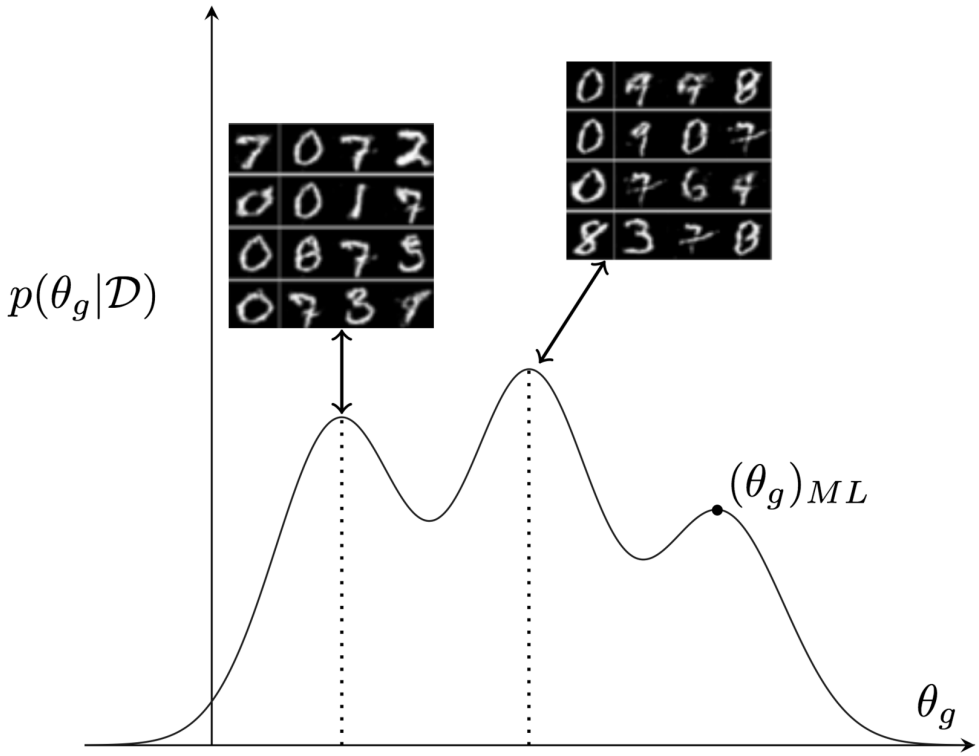


Figure 24: Multiple Generators for Multiple Peaks

8.1 A Problem

In order to introduce this idea, let us consider the following problem (similar to previous GANs).

Given some dataset $\xi_{data} = \{x_i\}_{i=1}^n$ of explanatory variables $x_i \sim p_{data}(x_i)$ we wish to estimate $p_{data}(x)$. We transform white noise $z \sim p(z)$ through a generator $G(z|\theta_g)$, parameterized by θ_g to produce candidate samples from the data distribution. We also use a discriminator $D(x|\theta_d)$ parameterized by θ_d, to output the probability that any x comes from the data distribution. These considerations hold for any general G and D, but typically we will choose multilayer perceptrons with weight vectors θ_g, θ_d for our generator and discriminator respectively.

By placing distributions over θ_g and θ_d, we induce distributions over an uncountably infinite space of generators and discriminators, corresponding to every possible setting of their weight vectors. Therefore, our generator now represents a distribution over distributions of data. Now we can sample from our induced prior distribution over data instances as follows:

- 1. Sample $\theta_g \sim p(\theta_g)$
- 2. Sample $z_1, ..., z_m \sim p(z)$
- 3. Let $\hat{x_j} = G(z_j|\theta_g) \sim p_{generator}(x)$

8.2 Unsupervised Learning

In unsupervised learning, we create models that learn from test data that has not been labeled, classified, or categorized. As a result, instead of responding to feedback, unsupervised learning identifies commonalities in the data and reacts based on the presence or absence of such commonalities in each new piece of data. Generative Adversarial Networks as a whole are unsupervised methods. They do not require labels in their test data to generate a distribution. The majority of machine learning methods require labeled data, making them supervised learning methods. The reason this is brought up is because Bayesian GANs allow for a different type of learning, semi-supervised learning. This will be covered in the next section.

For the default (unsupervised) learning, we wish to infer posterior distributions over θ_g and θ_g. Therefore, we will iteratively sample from the following conditional posteriors:

Generator

$$p(\theta_g|z, \theta_d) \propto (\prod_{i=1}^{n_g} D(G(z_i|\theta_g)|\theta_d))p(\theta_g|\alpha_g)$$

Discriminator

$$p(\theta_d|z, X, \theta_g) \propto \prod_{i=1}^{n_d} D(x_i|\theta_d) \prod_{i=1}^{n_g} (1 - D(G(z_i|\theta_g)|\theta_d))p(\theta_d|\alpha_d)$$

where:

- $p(\theta_g|\alpha_g)$ and $p(\theta_d|\alpha_d)$ are priors over the parameters of the generator and discriminator with hyperparameters α_g and α_d respectively

- n_d and n_g are the numbers of minibatch samples for the generator and discriminator
- $X = \{x_i\}_{i=1}^{n_d}$

We can understand the formulation above by starting from the generative process for data samples. Suppose we wish to sample weights θ_g from the prior distribution $p(\theta_g|\alpha_g)$, and then condition on this sample of the weights to form a particular generator. We would sample some noise $z \sim p(z)$ and transform this noise through the network $G(z|\theta_g)$ to generate candidate samples.

The discriminator, conditioned on the weights θ_d, outputs a probability that these candidate samples came from the data distributions. Our first equation above shows that if the discriminator outputs high probabilities, then the posterior $p(\theta_g|z, \theta_d)$ will increase in a neighborhood of the sampled setting of θ_g and decrease for other settings.

For the second equation above, the posterior over the discriminator weights θ_d, the first two terms form a discriminative classification likelihood. This labels the data as belonging to the actual data or from the generator. The last term in this equation is the prior on θ_d.

8.3 Classical Generative Adversarial Networks and Maximum Likelihood

The approach taken above is a natural Bayesian generalization of the classical Generative Adversarial Network. If we were to use uniform priors for θ_d and θ_g and take the maximum of our posterior (called maximum a posteriori or MAP optimization) instead of sampling the posterior over the equations above, then our local optima for our generator would be the same as the generator for the classical generative adversarial network.

8.4 Marginalizing Noise

In non-Bayesian GANs, our updates are implicitly conditioned on a set of noise samples z. We can instead choose to marginalize z from our posterior updates using Monte Carlo methods:

$$p(\theta_g|\theta_d) = \int p(\theta_g, z|\theta_d)p(z|\theta_d)$$

$$\approx \frac{1}{J_g} \sum_{j=1}^{J_g} p(\theta_g|z_j, \theta_d)$$

where $p(z|\theta_d) = p(z)$ and $z_j \sim p(z)$. Similarly,

$$p(\theta_d|\theta_g) \approx \frac{1}{J_d} \sum_{j=1}^{J_d} p(\theta_d|z_j, X, \theta_g).$$

This has particularly nice features for Monte Carlo Integration.

- $p(z)$ is a white noise distribution from which we can take efficient and exact samples.
- $p(\theta_g|z, \theta_d)$ and $p(\theta_d|z, X, \theta_g)$ when viewed as a function of z, should be reasonably broad over z since z is used to produce candidate data samples in the generative procedure.

As a result, each term in the Monte Carlo sum typically makes a reasonable contribution to the total marginal posterior estimates.

8.5 Posterior Samples

By iteratively sampling from $p(\theta_g|\theta_d)$ and $p(\theta_d|\theta_g)$ at every step of a training cycle, we can, in limit, obtain samples from the approximate posteriors over θ_g and θ_d. This can be quite useful in practice.

For example, we can use different samples for θ_g to alleviate mode collapse and generate data samples with a more appropriate gradient size. We can also use a committee of generators to strengthen the training of the discriminator.

The samples for θ_d in turn can create a committee of discriminators which will amplify the overall adversarial signal - further improving the unsupervised learning process.

8.6 Semi-Supervised Learning

In a similar vein as unsupervised learning, there exists certain methods which allow for semi-supervised learning. These are methods which allow for making use of both unlabeled and labeled data. Typically we have a large amount of unlabeled data and a small amount of labeled data.

We can extend our Bayesian GAN to semi-supervised learning. In this setting, suppose we wish to perform K-Class classification, in which we are placing new samples into some class y from available class labels $y_s \in \{1, ..., K\}$. Suppose also that we have access to a set of n unlabeled observations $\{x_i\}$, and a much smaller set of N_s labeled observations, $\{x_{s,i}, y_{s,i}\}_{i=1}^{N}$.

Our goal is to jointly learn the structure from both the unlabeled and labeled examples in order to make better predictions of class labels for new test examples x_* than we would if we only had access to the labeled training points.

In this context, we can redefine our discriminator such that $D(x_i = y_i|\theta_d)$ gives the probability that sample x_i belongs to class y_i. We will reserve the class label 0 as the output of the generator. We can then infer the posterior over the weights as follows:

$$p(\theta_g|z, \theta_d) \propto (\prod_{i=1}^{n_g} \sum_{y=1}^{K} D(G(z_i|\theta_g) = y|\theta_d))p(\theta_g|\alpha_g) \ )$$

$$p(\theta_d|z, X, y_s, \theta_g) \propto \prod_{i=1}^{n_d} \sum_{y=1}^{K} D(x_i = y|\theta_d) \prod_{i=1}^{n_g} D(G(z_i|\theta_g) = 0|\theta_d) \prod_{i=1}^{N_s} (D(x_{s,i} = y_{s,i}|\theta_d))p(\theta_d|\alpha_d).$$

During each iteration, we use n_g samples from the generator, n_d unlabeled samples, and all of the N_s labeled samples where typically $N_s << n$. As in the section above, we can also approximately marginalize z with Monte Carlo sampling.

Now we can compute the predictive distribution for a class label y_* at a test input x_* using a model average over all collected samples with respect to the posterior over θ_d. Just like in the unsupervised learning case, we can marginalize the posteriors over θ_g and θ_d.

$$p(y_*|x_*, \xi_{data}) = \int p(y_*|x_*, \theta_d)p(\theta_d|\xi_{data})d\theta_d$$

$$\approx \frac{1}{T} \sum_{k=1}^{T} p(y_*|x_*, \theta_{d,k}),\ \theta_{d,k} \sim p(\theta_d|\xi_{data}).$$

8.7 Posterior Sampling with Stochastic Gradient Hamiltonian Monte Carlo Methods

We wish to marginalize our posterior distributions over the generator and discriminator weights. In other words, we wish to solve the integral above. For this purpose we will use Stochastic Gradient Hamiltonian Monte Carlo (SGHMC) for posterior sampling. Stochastic Gradient Hamiltonian Monte Carlo is a Monte Carlo method for sampling from some known distribution in order to approximate some other analytically intractable distribution. In this case, our posterior distribution over the generator and discriminator weights is being approximated. SGHMC is used because it involves a momentum parameter which will allow our sampler to explore the furthest reaches of our posterior distribution (even if we have large areas with low probability density). For a detailed look into SGHMC, check [15].

8.8 Algorithm: Posterior Sampling for the Bayesian Generative Adversarial Network

For the algorithm:

- α is the friction term for SGHMC (helps counter noise effect from gradient, regularizes momentum)
- η is the learning rate (rate at which we update the gradient)
- We take J_g and J_d Monte Carlo samples from the generator and discriminator respectively
- We take M SGHMC samples for each simple Monte Carlo sample.

- Represent posteriors with samples $\{\theta_g^{j,M}\}_{j=1,m=1}^{J_g,M}$ and $\{\theta_d^{j,M}\}_{j=1,m=1}^{J_d,M}$ from previous iteration
- for number of Monte Carlo iterations J_g do the following:
  - Sample J_g noise samples $\{z_1, ..., z_{J_g}\}$ from noise prior $p(z)$. Each z_i has n_g samples.
  - Update sample set representing $p(\theta_g|\theta_d)$ by running SGHMC updates for M iterations:

$$\theta_g^{j,m} = \theta_g^{j,m} + \Delta\theta_g^{j,m}$$
$$\Delta\theta_g^{j,m} = (1-\alpha)\Delta\theta_g^{j,m} + \eta\frac{\partial \log(\sum_i \sum_k p(\theta_g|z_i,\theta_d^{k,m}))}{\delta\theta_g} + n$$
$$n \sim \mathcal{N}(0, 2\alpha\eta)$$

  - Append $\theta_g^{j,m}$ to sample set
- end for
- for number of Monte Carlo iterations J_d do
  - Sample minibatch of J_d noise samples $\{z_1, ..., z_{J_d}\}$ from noise prior $p(z)$
  - Sample minibatch of n_d data samples x
  - Update sample set representing $p(\theta_d|z,\theta_g)$ by running SGHMC updates for M iterations:

$$\theta_d^{j,m} = \theta_d^{j,m} + \Delta\theta_d^{j,m}$$
$$\Delta\theta_d^{j,m} = (1-\alpha)\Delta\theta_d^{j,m} + \eta\frac{\partial \log(\sum_i \sum_k p(\theta_d|z_i,x,\theta_g^{k,m}))}{\delta\theta_d} + n$$
$$n \sim \mathcal{N}(0, 2\alpha\eta)$$

- Append $\theta_d^{j,m}$ to sample set

- end for

8.9 Comparison with Classical and Wasserstein GANs

We have shown that Bayesian GANs can explore multimodal distributions over the weight parameters of the generator. As a result, we capture a diverse set of complementary and interpretable representations of data.

These results also enable effective semi-supervised learning. This is crucial for reducing the dependency of multilayer perceptrons on large labeled datasets. Often labeling is not an option or comes at a high cost. In a semi-supervised setting we can greatly increase accuracy and effectiveness of our algorithm with many less labeled samples than unlabeled.

In the classical GAN, a major setback was mode collapse. In order to effectively train a GAN we needed a slew of tricks including explanatory variable normalization, minibatch discrimination, and the addition of Gaussian noise (to prevent issues with disjoint sets). Even then, training could still be arduous. In Bayesian GANs, we have a robustness to the mode collapse problem thanks to a Gaussian prior over the weights of our discriminator and generator. Our use of a Markov Chain Monte Carlo sampling procedure (Stochastic Gradient Hamiltonian Monte Carlo) also helps alleviate the risk of collapse and helps explore multiple modes (and the uncertainty within each mode).

In the Wasserstein GAN we also helped alleviate some of the problems of the classical GAN. The wasserstein GAN and the classical GAN also both rely on maximum likelihood estimation of their generator approximation to pick the final generator. As a result of this, they are left with the generator which generates the most probable samples from a given dataset, which is similarly restricted by the most likely features of the most likely images in the dataset. While this creates good samples, it does not lead to a generator that can fully approximate the total variation within a given dataset.

The Bayesian GAN models a multimodal distribution approximation to our data which gives a clearer picture of the variation and uncertainty present in the data. It also provides multiple generators and discriminators, allowing for different variants of output and different methods of training (such as heightening adversarial activity through multiple discriminators, or training discriminators with multiple generators).

References

[1] John Tromp Combinatorics of Go https://tromp.github.io/go/gostate.pdf

[2] Stack Overflow User Pulsar how scientists know the number of atoms in the universe? https://physics.stackexchange.com/questions/47941/dumbed-down-explanation-how-scientists-know-the-number-of-atoms-in-the-universe

[3] Google Deepmind https://deepmind.com/research/alphago/

[4] David Silver Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm https://arxiv.org/pdf/1712.01815.pdf

[5] Joe Blitzstein Introduction to Probability https://www.homeworkforyou.com/static$_m$*edia/uploadedfile*

[6] Walter Rudin Real and Complex Analysis https://59clc.files.wordpress.com/2011/01/real-and-complex-analysis.pdf

[7] Jay Kadane Principles of Uncertainty http://www.stat.cmu.edu/ kadane/principles.pdf

[8] Garreth James, Daniela Witten, Trevor Hastie, Rob Tibshirani Introduction to Statistical Learning https://www-bcf.usc.edu/ gareth/ISL/ISLR%20First%20Printing.pdf

[9] Ian Goodfellow et. al Generative Adversarial Nets https://arxiv.org/abs/1406.2661

[10] Ian Goodfellow et. al Deep Learning https://www.deeplearningbook.org/

[11] Martin Arjovsky et. al Wasserstein GAN https://arxiv.org/abs/1701.07875

[12] Alix Irpan Read-Through: Wasserstein GAN https://www.alexirpan.com/2017/02/22/wasserstein-gan.html

[13] Vincent Herrman Wasserstein GAN and the Kantorovich-Rubinstein Duality https://vincentherrmann.github.io/blog/wasserstein/

[14] Yunus Saatchi, Andrew Gordon Wilson Bayesian GAN https://arxiv.org/abs/1705.09558

[15] Tianqi Chen, Emily B. Fox, Carlos Guestrin Stochastic Gradient Hamiltonian Monte Carlo https://arxiv.org/abs/1402.4102