An Introduction to Autoencoders, Meta-Learning, GANs, and Wasserstein GANs

A StartOnAI Tutorial

Anaiy Somalwar, Felix Liu

July 21, 2020

StartOnAI

Contents

Contents **ii**

1 Autoencoders **1**
1.1 Introduction . . . 1
1.2 Theory . . . 1
1.3 Applications . . . 2

2 Meta Learning **3**
2.1 Introduction . . . 3
2.2 Theory . . . 3
2.3 Applications . . . 4

3 Generative Adversarial Neural Networks (GANS) **5**
3.1 What Are GANS? . . . 5
A Brief History . . . 5
How Do They Work? . . . 5

4 The Wasserstein GAN (WGAN) **10**
4.1 What is a Wasserstein GAN? . . . 10
Introduction . . . 10
Research Progress . . . 10
KL Divergence . . . 10
Jensen-Shannon Divergence . . . 12
4.2 The Wasserstein GAN . . . 13
Motivation . . . 13
Wasserstein Distance . . . 13
Transport Plans . . . 14
Summary . . . 15

Bibliography **16**

Autoencoders 1

1.1 Introduction 1
1.2 Theory 1
1.3 Applications 2

1.1 Introduction

Autoencoders, first developed in the late 1980's, are a type of artificial neural networks that are often used in unsupervised learning, more specifically, representation learning, or feature learning. While autoencoders are used for a multitude of purposes, some of the most common include noise reduction and dimensionality reduction. Before we delve into the mathematical implications of autoencoders, let us first see what one looks like.

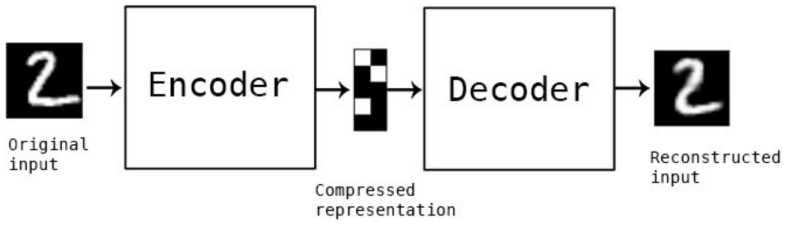


Figure 1.1: This is a visualization of an autoencoder for MNIST (https://towardsdatascience.com/auto-encoder-what-is-it-and-what-is-it-used-for-part-1-3e5c6f017726)

As we can see, the most basic autoencoder has two parts, the encoder and the decoder. In simple terms, the encoder is the function or layer in which a model learns how to compress data into a smaller dimension, and the decoder is the way a model learns how to re-create the data from before the encoder. The way to measure the difference between the original inputs and the reconstructed inputs is called the reconstruction loss.

1.2 Theory

Now that we have briefly looked at the components of an autoencoder, let's discuss the theory behind it. The simplest type of autoencoder is a feedforward network that very closely resembles a layer of an MLP. Just like individual layers in MLPs, there is a function that relates the inputs and outputs of that layer and is thus a form of unsupervised learning. Autoencoders that are used as networks often follow a similar pattern : an encoder, several middle layers that can vary as MLPs to RNNs, and the decoder, which is essentially an inverse function of the encoder. The point of this network is to allow the middle layers to "improve" the input, or make it closer to the output mathematically. While we mentioned that one of the trademark transformations from this network lead to noise reductions, they also sometimes re-create the input data into something that human experts in that field did not expect, which is one of the more common ways breakthroughs in these fields occur. After all, one of the main reasons why AI is so different from other studies of various fields is

because the model creates the rules for you and expertise in that subject is not necessary. One great way of visualizing autoencoders is to see it in code and to understand what part of a model is the autoencoder. Let's take a look at an autoencoder for mnist:

```
encoder = Sequential([
    Flatten(input_shape = (28, 28)),
    Dense(512),
    LR(),
    Dropout(0.5),
    Dense(256),
    LR(),
    Dropout(0.5),
    Dense(128),
    LR(),
    Dropout(0.5),
    Dense(64),
    LR(),
    Dropout(0.5),
    Dense(LATENT_SIZE),
    LR()
])
```

Listing 1.1: This is what the encoder part for MNIST looks like.

```
decoder = Sequential([
    Dense(64, input_shape = (LATENT_SIZE,)),
    LR(),
    Dropout(0.5),
    Dense(128),
    LR(),
    Dropout(0.5),
    Dense(256),
    LR(),
    Dropout(0.5),
    Dense(512),
    LR(),
    Dropout(0.5),
    Dense(784),
    Activation("sigmoid"),
    Reshape((28, 28))
])
```

Listing 1.2: This is what the decoder part for MNIST looks like

As you can see, the only difference between the decoder and encoder of MNIST are their order; the decoder is simply the opposite of the encoder. While this autoencoder is quite complex in the number of layers it has, it does a purpose that less complex autoencoders will also do fairly well, reduce the noise in images.

1.3 Applications

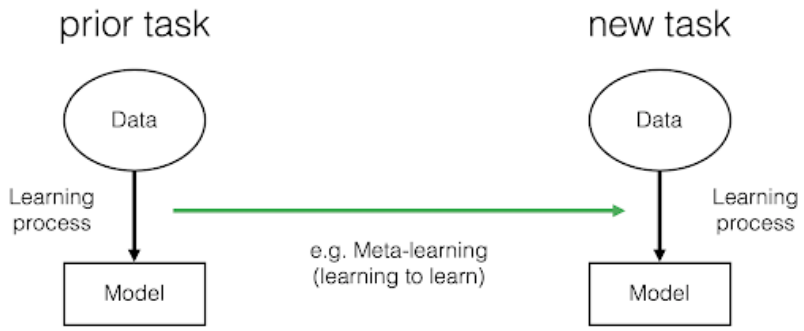
While we had mentioned that autoencoders are used for dimensionality reduction and denoising, they also have many other purposes. Autoencoders are also used for anamoly detection and outliers, for example their deletion, image processing (often as a preprocessing step for image classification models), and data compression. Overall, autoencoders can be used almost in any type of model to improve it slightly or significantly.

Meta Learning 2

2.1 Introduction 3
2.2 Theory 3
2.3 Applications 4

2.1 Introduction

Meta learning is a subfield under machine learning, which is a subfield under artificial intelligence. Meta learning is the study of using learning algorithms on data to optimize current algorithms and improve learning and generalization. The goal of meta learning is to make it so models with "knowledge" on similar problems could be easily adapted to other problems not in regards to accuracy particularly, but more in terms with the speed of learning.

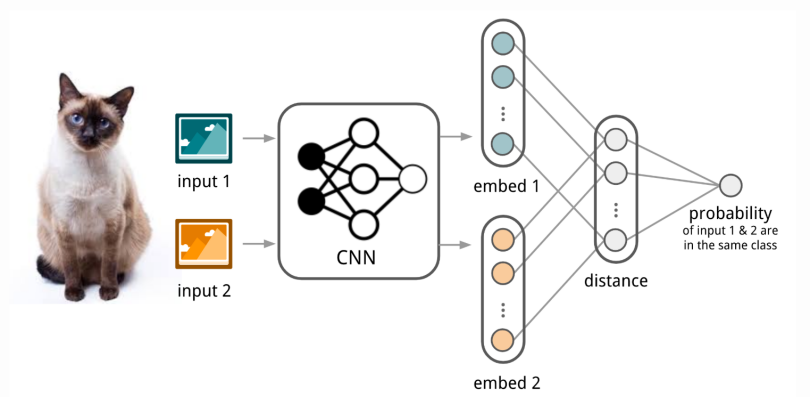


[Flennerhag, 2019]

Figure 2.1: This is the general setup of meta learning.

2.2 Theory

There are three different types of meta learning - metric-based meta learning, model-based meta learning, and optimization-based meta learning. The main principle behind metric-based meta learning is very similar to the KNN unsupervised learning model; the prediction y-hat is a weighted matrix of input values by a kernel. Similarly to the KNN model, without having a good kernel function, it is not possible to have great results. One famous model that incorporates metric-based meta learning is the Convolutional Siamese Neural Network, which outputs the probability that two input images are in the same class.



[Weng, 2018]

Figure 2.2: This is the architecture of the Convolutional Siamese Neural Network.

While metric-based meta learning relies on a function that correlates the input data to the output data, modle-based meta learning does not make those assumptions; it is used for fast learning by varying its internal architecture. Within model-based meta learning, there are different architectures, namely memory augmented neural nets and meta networks. Augmented neural networks, which includes Turing machines, use memory that is not internal to increase the rate of learning of models. While this sounds similar to LSTMs or RNNs, they are different as RNNs and LSTMs only use internal memory. Finally, there remains optimization-based meta learning, which is used to increase the accuracy of models. The most common optimization-based meta learning model is the "LSTM Meta Learner" or a model that contains elements of an LSTM and memory augmented neural networks. These models are used for fast learning and improved optimization.

2.3 Applications

Meta learning can be used with many different purposes, which is one reason why it is so commonly referred to in creating or improving machine learning / deep learning models. Since meta learning is more of a technique, it can be used in any model; however, in the past it has often been dealt within projects that involve time series data where LSTMS are one of the forefront models. However, it is also fairly common in image classification tasks. Overall, meta learning has a wide scope within machine learning and AI and is definitely something that should be considered while optimizing models.

Generative Adversarial Neural Networks (GANS)

3

3.1 What Are GANS?

3.1 What Are GANS? 5
A Brief History 5
How Do They Work? 5

A Brief History

GANs, or generative adversarial nets, are a type of learning framework that was first proposed in 2014 by Ian Goodfellow and his colleagues. In his paper, GANs are described as a "[frameworks] for estimating generative models via an adversarial process, in which we simultaneously train two models." [3] More specifically, we train a generative model, G, and a discriminative, D, that compete against each other in which each model attempts to outperform the other; this is a type of minimax game.

[3]: Goodfellow et al. (2014), 'Generative Adversarial Networks'

In the paper, the generative model is intuitively thought of as a team of counterfeiters, which tries to produce fake money. On the other hand, the discriminative model is analogous to a team of police who are trying to detect, or "discriminate," fake money from real money. In this cycle of competition, both groups strive to improve; the counterfeiters try to create more realistic fakes, while the police become better at discerning counterfeits. Eventually, the counterfeit money becomes indistinguishable from the real money.

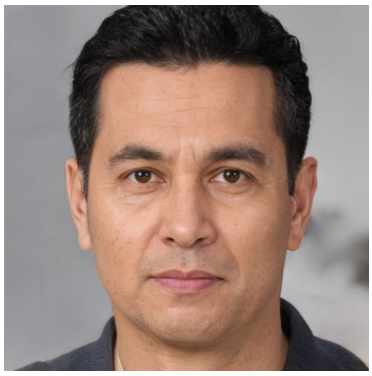


Figure 3.1: This image was generated by AI [2]

How Do They Work?

GANs are typically employed when both the generative and adversarial networks, G and D, are multi-layer perceptron algorithms, meaning that a network is constructed by layering perceptrons on top of each other.

The Setup

Creating the Models It is established that our adversarial models, G and D, are neural networks that compete against each other. In this

Credits for the above image go to: https://generated.photos/

particular situation, our data set contains images of cats, and we want our model G to learn how to create new images of cats given a data set X.

Let us begin by examining G. Since G is a generative model, which will produce images of cats in this case, it needs to take in some input that describes what kind of output we want. For consistency, let us call this input z, and refer to it as *noise*, which will come from some distribution $p_z(z)$. This noise determines some characteristics of the cat, like how furry it is, or how long the tail is. Furthermore, our model G, must also have some parameters that further determine what kind of output is generated; let us call these parameters θ_g. These parameters help tweak the final image to actually resemble a cat. Additionally, G can be thought of as some sort of function that maps z into the data space; therefore, we find it clearer to represent this as $G(z;\theta_g)$, where G is a "function" with parameters θ_g. This means that we can reduce our problem of learning to generate cats to a simpler question: how do we teach G a distribution (which we will call p_g) that resembles the data distribution $p_{\text{data}}(x)$. We will discuss how this is done through the use of a discriminator.

Moving on, we now define a discriminative model, D, that takes as input some data point and returns a scalar between 0 and 1. This scalar represents the probability that the input comes from the data set and not generated by G; this is equivalent to the probability that the input is not from the distribution p_g. For example, $D(x;\theta_d)$ should return a large number like 0.97 when x is an image of a cat from the data set, but return a small number like 0.24 when the image is a cat generated by G. A general overview of a GAN can be found in Figure 3.2.

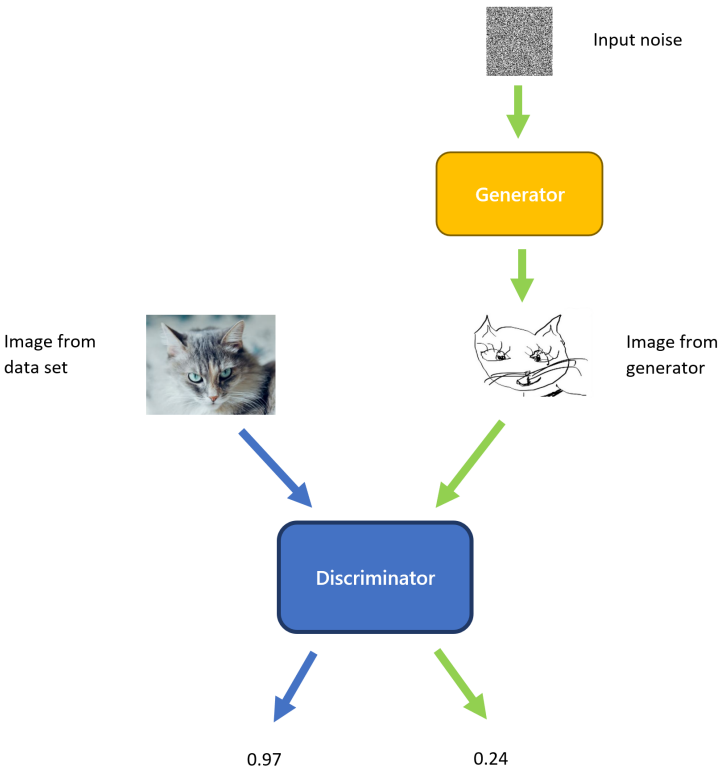


Figure 3.2: This is a generic overview of how a GAN works. Image credits for the cat on the left go to https://www.thesprucepets.com/siberian-cat-4178148[1]. Image credits for the cat on the right go to: https://www.quotev.com/story/4403096/Adopted-by-bvb-Finished-UNDER-EDIT/33[8]

Deriving the Loss Function We will now work towards deriving a suitable loss function for our models. This section will go through the derivation of the binary-cross entropy loss, and how it is adapted for use in a GAN.

We begin by deriving the generic formula for binary cross-entropy loss. This means that we will have a data set X with labeled data points in the form (x_i, y_i), where x_i is the data point and y_i is the label. Since we are dealing with only two classes, we restrict y_i to be either 0 or 1.

Moving on, we must define favorable properties for our loss function, which we will denote by L. First of all, the loss must be high when we misclassify a point and near 0 when we correctly classify a point. We will denote the predicted label as $\hat{y}$ and the actual label as y[1] .

1: Notice that we drop the subscripts, because these can also be images produced by G, which are not part of our data set

$$L = \begin{cases} \begin{cases} 0 & \hat{y} = 0 \\ \infty & \hat{y} = 1 \end{cases} & y = 0 \\ \begin{cases} \infty & \hat{y} = 0 \\ 0 & \hat{y} = 1 \end{cases} & y = 1 \end{cases} \tag{3.1}$$

From Equation 3.1, we notice that we can modify the plot the of the logarithm on the interval [0,1] to achieve such a loss function. In particular, we can flip the graph of $y = \log(x)$ across the x axis to obtain a suitable loss function for the case where $y = 1$. For the case where $y = 0$, we obtain the loss function by reflecting our previous loss function across the line defined by $x = \frac{1}{2}$. The plots of these loss functions can be seen below:

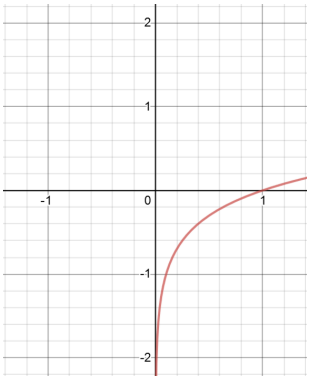
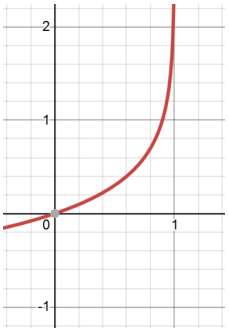
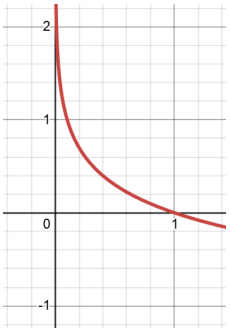


Figure 3.3: graph of $y = \log(x)$



(a) Loss function for $y = 0$. Plotted $L = -\log(\hat{y})$



(b) Loss function for $y = 1$. Plotted $L = -\log(1 - \hat{y})$

Figure 3.4: Graphs of the loss functions. Note: we only care about behavior on the interval $\hat{y} \in [0, 1]$

We can rewrite this in the form of an equation to obtain:

$$L = \begin{cases} -\log(1 - \hat{y}) & y = 0 \\ -\log(\hat{y}) & y = 1 \end{cases} \tag{3.2}$$

Notice that we can take advantage of the fact that y is only 0 or 1. This means we can reduce Equation 3.2 to one line:

$$\boxed{L = -\Big((1 - y)\big(\log(1 - \hat{y})\big) + (y)\big(\log(\hat{y})\big)\Big)} \tag{3.3}$$

We have successfully derived the general form for binary cross entropy loss. This means we can now apply this formula to a GAN.

Applying Binary Cross Entropy Loss to a GAN Here, it becomes necessary to define the parameters for our loss function. In particular, letting the parameters be y and $\hat{y}$, such that our loss function is $L(y, \hat{y})$, is very useful.

When training our discriminator, we will be attempting to minimize the loss function defined by Equation 3.3. This is equivalent to maximizing $-L(y, \hat{y})$. For ease of notation, we will now redefine L to be the following:

$$L(y, \hat{y}) = \big(1 - y\big)\big(\log(1 - \hat{y})\big) + \big(y\big)\big(\log(\hat{y})\big) \tag{3.4}$$

However, we still have y restricted to be either 0 or 1. In the case that $y = 1$, we know that the x in $D(x)$ is an image from our data set X. On the other hand, if $y = 0$, the x in $D(x)$ is from the distribution p_g, meaning it was an image produced by G. This means we can set $D(x) = D\big(G(z)\big)$ when $y = 0$. From here, we can get two distinct equations by instantiating (or "substituting") what we know into y and $\hat{y}$ as is seen in the following expressions[2] :

2: We can substitute $D(x) = \hat{y}$, because $\hat{y}$ represents a predicted label, which is essentially what $D(x)$ tells us.

$$L\Big(0, D\big(G(z)\big)\Big) = \log\Big(1 - D\big(G(z)\big)\Big) \tag{3.5}$$

$$L\Big(1, D(x)\Big) = \log\Big(D(x)\Big) \tag{3.6}$$

Now, in order to maximize the probability of D correctly labeling images from data set X, we want to maximize loss for when $y = 1$. This means maximizing $\log\big(1 - D(x)\big)$. Similarly, we also want to maximize the probability that $D(x)$ correctly labels images generated by G; this is when $y = 0$. Therefore, we also want to maximize $\log\Big(D\big(G(z)\big)\Big)$. This is equivalent to maximizing the sum of Equations 3.5 and 3.6. Mathematically, we write this as:

$$\max(L) \Rightarrow \max\left(\log\Big(D(x)\Big) + \log\Big(1 - D\big(G(z)\big)\Big)\right) \tag{3.7}$$

This means that, in order to find the optimal discriminator, we maximize $D(x)$ and minimize $D\big(G(z)\big)$. We can easily see this if we graph the individual components of the loss function like in Figure 3.4. 3mm
Now that we have determined how our discriminator should behave, we will discuss how our generator will learn from this. Well, if we want our generator to produce images of cats that appear to be from our data set, this would mean that G would have to fool our discriminator. This is done when $D\big(G(z)\big)$ returns a value near 1, like 0.94. This would mean that the value we achieve from Equation 3.7 would approach $-\infty$ when the discriminator is fooled. In other words, in other fool the discriminator, G wants to minimize L.

This results in a mini-max between G and D such that G wants to minimize L, while D wants to maximize L. Putting these conditions together, we find that we find some sort of equation to represent this scenario.

$$\min_G \max_D \left(\log\Big(D(x)\Big) + \log\Big(1 - D\big(G(z)\big)\Big)\right) \tag{3.8}$$

Unfortunately, this equation only accounts for one data point in the entire data set X. To account for other data points, we must use some basic probability with expected values to reformulate our equation. For simplicity of notation, we define this to be $V(D, G)$, which is a value function representing the mini-max game. It turns out that this reformulated equation is what appears in Goodfellow's paper.

$$\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{\text{data}}(x)} \log D(x) + \mathbb{E}_{z \sim p_z(z)} \log D\big(G(z)\big) \tag{3.9}$$

In the above equation, $x \sim p_{\text{data}}(x)$ tells us that x comes from a the distribution of data $p_{\text{data}}(x)$, quite like how images generated by G come from the distribution p_g. Similarly, $z \sim p_z(z)$ tells us that z comes from some noise distribution defined as $p_z(z)$. $\mathbb{E}_{x \sim p_{\text{data}}(x)}$ tells us the expected value that x is in fact from the data set, while $\mathbb{E}_{z \sim p_z(z)}$ tells us the expected value that z is from the noise distribution.

Now that we have found a suitable loss function for our GAN, we can move to training the network.[3, 5, 4]

[3]: Goodfellow et al. (2014), 'Generative Adversarial Networks'
[5]: Kumar (), *Deep Learning 28: (2) Generative Adversarial Network (GAN) : Loss Derivation from Scratch*
[4]: IITM (), *Binary Entropy cost function*

Training the GAN The in-depth process of training a GAN will not be covered. This is because the training process operates through gradient descent (mini-batch is used in Goodfellow's paper) and back-propagation, which are both assumed prerequisites in reading this tutorial.

There are still some key things to note when training a GAN. For one, the discriminative and generative model must be trainined side-by-side. This can be done by first training D for one or two epochs, training G for one or two epochs, and then repeating. This prevents a "Helvetica scenario" in which data is overfitted, leading to "mode collapse." This situation is similar to "burning" in Markov chains.[3]

Unfortunately, p_g has no closed form, meaning it must be approximated through the training process. However, it is guaranteed that there does exist a global minimum, which is a distribution denoted by p_{data}. Furthermore, we are guaranteed to converge upon this local minimum when $p_g = p_{\text{data}}$. The proofs for these assertions are covered in Goodfellow's paper. [3]

3: The proofs involve measures of "distance" between distributions, which tell us how "different" p_g and p_{data} are. In particular, the proofs use the **KL divergence** (Kullback-Leibler divergence) and **JSD** (Jensen-Shannon Divergence) measures. Unfortunately, these measures both have their shortcomings, which will be discussed in the section on Wasserstein GANs.

Summary We begin setting up our two model G and D, where G is the generative model and D is the discriminative model. A variant of the binary cross entropy loss function is then used to model the mini-max game that our models play; this is Equation 3.9. The models are then trained using gradient descent until p_g is learned and it converges upon p_{data}, the distribution that results in a global minimum (this distribution is guaranteed to exist and be converged upon).

The Wasserstein GAN (WGAN) 4

4.1 What is a Wasserstein GAN? 10
Introduction 10
Research Progress 10
KL Divergence 10
Jensen-Shannon Divergence 12
4.2 The Wasserstein GAN 13
Motivation 13
Wasserstein Distance 13
Transport Plans 14
Summary 15

4.1 What is a Wasserstein GAN?

Introduction

The Wasserstein GAN, or WGAN, was first introducted in a 2017 paper by a research group consisting of Martin Arjovsky, Soumith Chintala, Léon Bottou, et al. This variation on the GAN succeeded in combating many of the problems faced by the original GAN proposed in Goodfellow's 2014 paper. One of these problems was mode collapse, which would occur if the adversarial model were not trained properly. Furthermore, the group also provided thorough theoretical work to further the application of GANs.

Research Progress

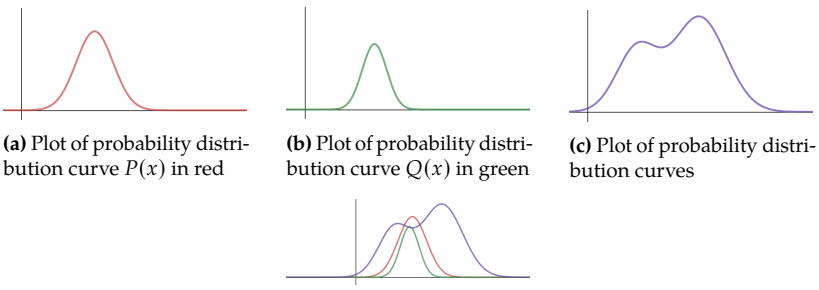
The traditional GAN proposed in 2014 relied on the **KL divergence** (Kullback-Leibler divergence) and **JSD** (Jensen-Shannon Divergence) measures to prove that the training algorithm would converge to a global minimum. Although these methods have an advantage over **variational auto-encoders**, these measures still have some flaws. This eventually led researches to the use of the Wasserstein distance measure, resulting in the creation of the WGAN.

KL Divergence

What is KL Divergence?

The Kullback-Leibler divergence is a measure of "distance" that describes how similar two probability distributions are. [1] We now go through a brief example to illustrate some basic intuition between KL divergence. We begin by defining probability distribution curves $P(x)$, $Q(x)$, and $R(x)$. The plots of these are shown below in Figure **??**.

1: One key property of this function is that the distance is asymmetric, meaning the distance from the distribution $P(x)$ to $Q(x)$ is not the same as the distance from $Q(x)$ to $P(x)$. Furthermore, the formula does not satisfy the triangle inequality; therefore, KL divergence is not a **distance metric**.



(a) Plot of probability distribution curve $P(x)$ in red

(b) Plot of probability distribution curve $Q(x)$ in green

(c) Plot of probability distribution curves

(d) Plot of all the the probability distributions above

Figure 4.1: These are the plots of $P(x)$, $Q(x)$, and $R(x)$

As seen in Figure 4.1d, we can easily see that the $P(x)$ looks much closer to $Q(x)$ than $R(x)$. This means that $D_{\mathrm{KL}}(P \parallel Q)$ is much smaller than $D_{\mathrm{KL}}(P \parallel R)$. This is the essence of KL divergence.

Mathematically, the formula for KL divergence is given by the following. Notice that we sometimes write $D_{\mathrm{KL}}(P \parallel Q)$ as $\mathrm{KL}(P \parallel Q)$ for simplicity of notation.[2]

$$D_{\mathrm{KL}}(P \parallel Q) = \mathrm{KL}(P \parallel Q) = \sum_x P(x) \log\left(\frac{P(x)}{Q(x)}\right) \tag{4.1}$$

2: This equation is for discrete probability curves. The formula for continuous probability curves involves an integral

Note that there are two kinds of KL divergence: **forward KL** and **reverse KL**. Forward KL divergence is found by calculating $\mathrm{KL}(P \parallel Q)$, while reverse KL is found by calculating $\mathrm{KL}(Q \parallel P)$.

Flaws of KL Divergence

Traditionally, when training a GAN, we are trying to teach our generative model, G, a probability distribution, p_g that replicates the distribution of the data, p_{data}. Consequently, this is equivalent to minimizing the KL divergence between p_g and p_{data}.

When using forward KL, we must find $\min\left(\mathrm{KL}(p_g \parallel p_{\text{data}})\right)$. For the sake of demonstration, assume p_g is a Gaussian while p_{data} is a bi-modal distribution. This is seen in Figure 4.2, where p_g is plotted in red and p_{data} is plotted in blue. In the image on the left, p_g is initialized to be some arbitrary Gaussian, which happens to start off with infinite divergence. After KL divergence is minimized, we end up at the image to the right.

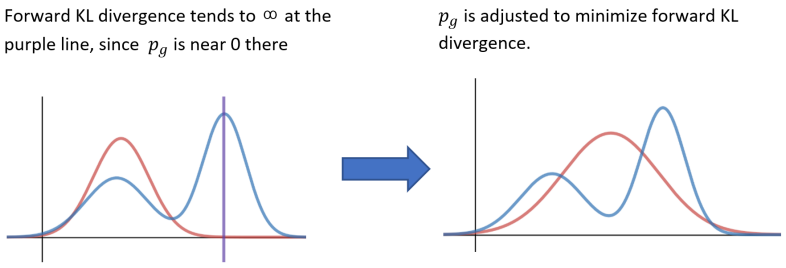


Figure 4.2: Training using forward KL divergence

Notice that the final plot of p_g shares endpoints with the distribution p_{data}. This means that the peak of p_g is the middle of p_{data}. This kind of behavior is known as **mean-seeking** and **zero-avoidance** behavior.

We go through the same process as shown before, only instead we use reverse KL divergence. Initializing p_g in the same way, we find that minimizing the reverse KL divergence results in the behavior shown in Figure 4.3. Such behavior is called **mode-seeking** behavior.

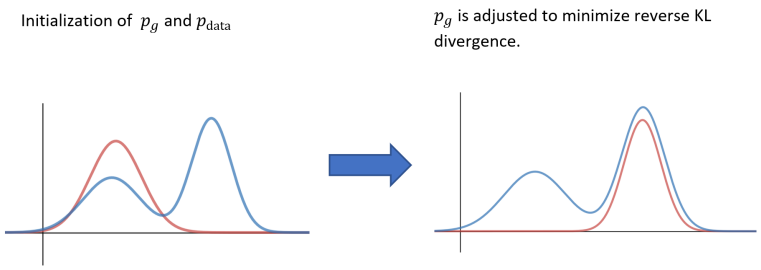


Figure 4.3: Training using reverse KL divergence

Unfortunately, we notice it is difficult to use KL divergence if p_g and p_{data} do not overlap. If the distributions do not overlap, then the KL divergence blows up to ∞ resulting in "mode collapse," as seen in Figure 4.4. The

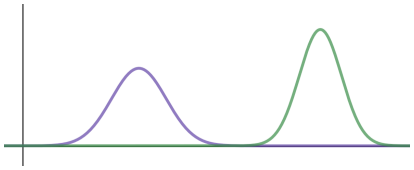


Figure 4.4: KL divergence blows up to ∞

following figure shows another case where p_g and p_{data} do not overlap and are discrete distributions.

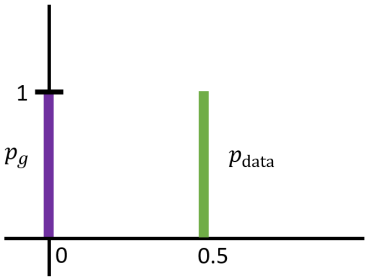


Figure 4.5: The probability distributions p_g and p_{data} are 1 at 0 and 0.5 respectively. The KL divergence blows up to ∞

This is where **JSD**, or Jensen-Shannon Divergence, comes into play. [6]

[6]: Kumar (), *Deep Learning 34: (1) Wasserstein Generative Adversarial Network (WGAN): Introduction*

Jensen-Shannon Divergence

Jensen-Shannon Divergence is used to combat the case where the distribution do not overlap and KL divergence can not be used; this is seen in Figure 4.4. In essence, Jensen-Shannon divergence is another way of defining of "distances" between distributions, where the divergence between distribution P and Q is given by $\text{JSD}(P \parallel Q)$. The mathematical formula for this is given below in Equation 4.2.

$$\text{JSD}(P \parallel Q) = \frac{1}{2}\big(KL(P \parallel M) + KL(Q \parallel M)\big) \qquad (4.2)$$
$$\text{where } M = \frac{P+Q}{2}$$

Using JSD, we have much more flexibility to train our GANs, as we can deal with the case when our distributions are far apart. This makes JSD superior to KL divergence.

However, in order to show that JSD is indeed superior to KL divergence, we will revisit the case where our distributions do not overlap and are far apart. This is seen in Figure 4.5. Substituting our observation from the figure into the formula for JSD and KL divergence, we find that:

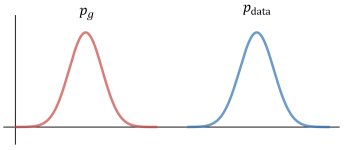
$$\begin{aligned}\text{JSD}(p_g \parallel p_{\text{data}}) &= \frac{1}{2}\left(p_g(x=0)\left(\log\left(\frac{p_g(x=0)}{\frac{p_g(x=0)+p_{\text{data}}(x=0)}{2}}\right)\right) + p_{\text{data}}(x=0.5)\left(\log\left(\frac{p_{\text{data}}(x=0.5)}{\frac{p_g(x=0.5)+p_{\text{data}}(x=0.5)}{2}}\right)\right)\right) \\ &= \frac{1}{2}\left(1\log(2) + 1\log(2)\right) = \boxed{\log(2)}\end{aligned}$$

This shows us that the JSD between two non-overlapping distributions converges to some concrete number, telling us more information than KL divergence, which blew up to ∞.[6]

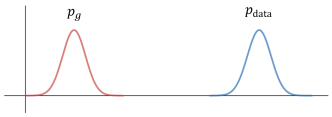
4.2 The Wasserstein GAN

Motivation

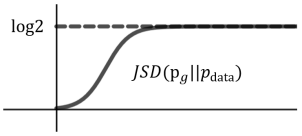
Unfortunately, JSD is still not a perfect definition for distances. To see why, let us observe what happens when we train our GAN using JSD. In the following figures, we will be dealing with the case where the distributions p_g and p_{data} do not overlap. As is evident, we see that the



(a) plot of p_g and p_{data}



(b) plot of p_g and p_{data}, but the distributions are farther away from each other



(c) Plot of the JSD as p_{data} moves farther from p_g

JSD converges to log(2). This means that if our discriminator D is too smart, then our GAN will not learn, as our loss function is 0; this is where JSD fails.

Wasserstein Distance

Earth Mover Distance

This distance is unique in that it measures the distance between distributions horizontally. Later on, this will become the stepping stone for developing Wasserstein distance. A basic intuition for Earth Mover can

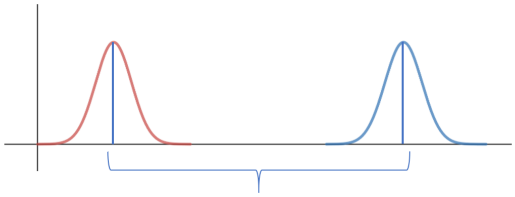


Figure 4.7: This roughly captures what Earth Mover Distance measures

be seen in Figure 4.7.

Equation for Wasserstein Distance

The equation for Wasserstein distance, W for two distribtions P and Q is given by the following:

$$\boxed{W(P,Q) = \inf_{\gamma \in \Pi(P,Q)} \mathbb{E}_{(x,y)\sim\gamma} \parallel x - y \parallel} \tag{4.3}$$

In the equation, $\Pi(P,Q)$ is a transport plan for P and Q (this will be covered shortly), the inf part tells us γ is the transport plan with lowest cost, $\mathbb{E}$ is the expected value of (x,y) sampled from γ, and $\parallel x - y \parallel$ is the distance between x and y.[7]

[7]: Kumar (), *Deep Learning 35: (2) Wasserstein Generative Adversarial Network (WGAN): Wasserstein metric*

Transport Plans

Say we want to shift two boxes along the number line. In order to define how we want to move these boxes, we need to come up with the notion of a transport plan Π. For example, if we have a box 1 at $x = 3$ and box 2 at $x = 8$, and we wanted the move the boxes to $x = 4$ and $x = 7$, we would have two possible transport plans.

Figure 4.8: Setup the boxes on the number line

We define our first transport plan, Π_1 to be the plan where box 1 moves to $x = 5$ and box 2 goes to $x = 7$. In this plan, box 1 travels 2 units and box 2 travels 1 unit; therefore, the cost for Π_1 is 3.

For transport plan Π_2, box 1 will move to $x = 7$ and box 2 will move to $x = 5$. Here, box 1 travels a total of 4 units and box 2 travels a total of 3 units; therefore the cost for Π_2 is 7. As a result, Wasserstein distance would have chosen the transport plan Π_1. This is extremely similar to Earth Mover Distance, as Wasserstein distance gives a rough measure of how far apart two distributions are.

Wrap-up

We see that Wasserstein distance does not suffer from converging to a value like log(2), as opposed to JSD. This means that we do not have to worry about the case where our distributions p_g and p_{data} do not overlap, because of how Wasserstein distance is defined.

Summary

We have finished discussing how Wasserstein distance works and why it is superior to other distance measures like KL divergence or JSD. As a result, a new and improved GAN will be trained by minimizing Wasserstein distance. This is the essence of a WGAN.

Bibliography

[1] Jackie Brown. *Siberian Cat: Cat Breed Profile*. [Online; accessed 19-July-2020]. URL: https://www.thesprucepets.com/siberian-cat-4178148 (cited on page 6).

[2] *generated.photos*. https://generated.photos/. Accessed: 7/17/20 (cited on page 5).

[3] Ian J. Goodfellow et al. 'Generative Adversarial Networks'. In: *ArXiv* abs/1406.2661 (2014) (cited on pages 5, 9).

[4] NPTEL-NOC IITM. *Binary Entropy cost function*. [Online; accessed 19-July-2020]. URL: https://www.youtube.com/watch?v=2ca_K2rgNVA (cited on page 9).

[5] Ahlad Kumar. *Deep Learning 28: (2) Generative Adversarial Network (GAN) : Loss Derivation from Scratch*. [Online; accessed 19-July-2020]. URL: https://www.youtube.com/watch?v=ZD7HtL1gook (cited on page 9).

[6] Ahlad Kumar. *Deep Learning 34: (1) Wasserstein Generative Adversarial Network (WGAN): Introduction*. [Online; accessed 19-July-2020]. URL: https://www.youtube.com/watch?v=_z9bdayg8ZI (cited on pages 12, 13).

[7] Ahlad Kumar. *Deep Learning 35: (2) Wasserstein Generative Adversarial Network (WGAN): Wasserstein metric*. [Online; accessed 20-July-2020]. URL: https://www.youtube.com/watch?v=y8LGAhzCOxQ (cited on page 14).

[8] Skyla. *Adopted by bvb (Finished) - UNDER EDIT: chapter 24*. [Online; accessed 19-July-2020]. URL: https://www.quotev.com/story/4403096/Adopted-by-bvb-Finished-UNDER-EDIT/33 (cited on page 6).