Anders Christansen Sørby

June 2019

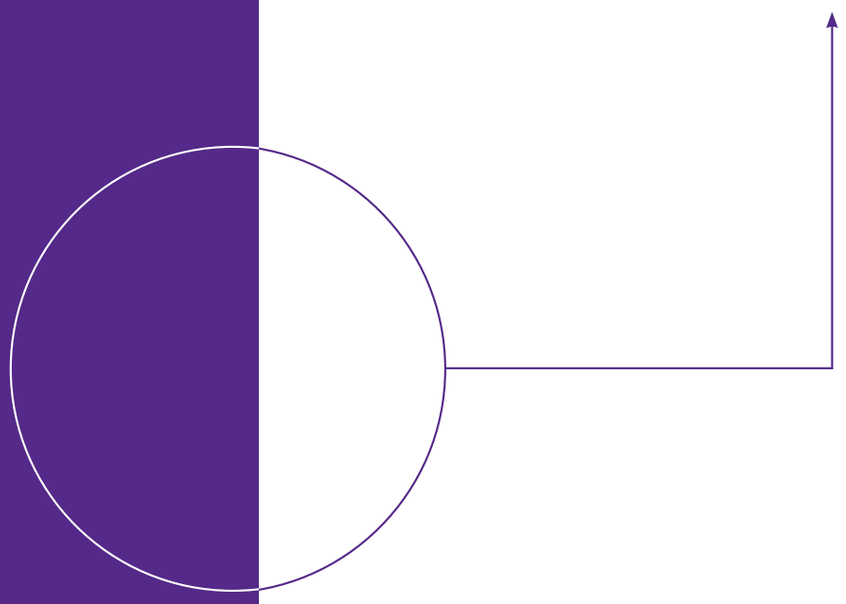
Master's thesis

NTNU
Norwegian University of
Science and Technology

Master's thesis

NTNU
Norwegian University of
Science and Technology
Faculty of Information Technology and Electrical
Engineering
Department of Mathematical Sciences

Master's thesis 2019 Anders Christansen Sørby



NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

Anders Christansen Sørby

Applied Physics and Mathematics
Submission date: June 2019
Supervisor: Bo Henry Lindqvist

Norwegian University of Science and Technology
Department of Mathematical Sciences

Til familie og venner

To my family and friends

Sammendrag

Når menneskehjernener prosesserer input fra sansene er de i stand til å unastrengt forestille seg nye instanser og scenarioer fra bare en liten mengde inputopplevelse. Generative Adverseriale Nettverk (GAN) klarer til en viss grad å oppnå denne forstillingsevnen for datasett. Vi utforsker denne teknikken og dens anvendelser innen bildeprosessering og generering. Dette har anvendelsesområder som medisin, fysikk og kunstig intelligens.

For eksempel implementerer vi pix2pix algoritmen for å transformere satellittbilder til kart. Denne algoritmen er anvendelsesuavhengig og kan takle særdeles forskjellige problemer uten mye tilpasning. Dette viser at det er mulig å lage generaliserte metoder for kompliserte domene til domene transformasjoner.

Hovedvekten i denne avhandlingen vil allikevel falle på den omfattende litteraturstudien av GAN-varianter. Her vil vi dekke over forskjellige tapsfunksjoner for GAN, som Wasserstein-metrikken, funksjonelle gradienter for fininstilling av GAN, måter å kontrollere generert output, som betinget GAN, Syklisk GAN og InfoGAN, og til slutt en Bayesiansk utvidelse av GAN som bidrar med usikkerhet og inferens til GAN.

Summary

When human brains process input from the senses they are able to effortlessly imagine new instances and scenarios from only a small amount of input experience. Generative Adversarial Networks (GANs) manages to some extent to achieve this imagination ability for datasets. We explore this technique and its applicability in image processing and generation. This has applications in areas like medicine, physics, and artificial intelligence.

For example we implement the pix2pix algorithm for transforming satellite images into maps. This algorithm is application independent and can handle vastly different problems without significant tweaking. This shows that it is possible to create generalized methods to do complicated domain to domain transformations.

The main weight of this thesis will nonetheless fall on the extensive literature study of GAN variants. Here we cover different loss functions for GANs, like the Wasserstein metric, functional gradients for fine tuning GANs, ways of controlling the generated output, like conditional GAN, CycleGAN, and InfoGAN, and finally a Bayesian extension of GAN that provides uncertainty and inference to GANs.

Preface

I would like to thank my family, supervisor and friends for discussing the topics of my master thesis and supporting me in my process. The project started off on a vague note with great ambitions. It ended with reality and time constraints finally putting a stop to new diversions. It has been a pleasure and a pain to work on this and I suspect I have learned a something about time management, my own limitations, and the enormous effort that goes into world class research. This is in addition to all the technical knowledge and insight gained from writing this thesis.

I believe there is a lot of potential for better writing in technical texts like mathematics or computer science. Concepts are not best taught by just describing them precisely, but by conveying intuition and using creative formulations. That way the reader not only obtains the facts of the text, but learns to represent those structures in her brain. The analogue to this in machine learning is that we often add noise in various contexts to avoid the model to collapse or stagnate in a local optimum.

Although this thesis is not explicitly about artificial intelligence, the topic of deep learning and generative adversarial networks is seen by many as a stepping stone to more general artificial intelligence. I believe that the advent of increasingly sophisticated methods for processing and synthesising data will significantly alter the way we live, our society, and the way we see the world. That is why I think it is necessary to keep in mind the underlying philosophical ideas, societal norms, and psychological issues we are indirectly imposing into the development of these methods and practical applications. Otherwise we risk amplifying the destructive tendencies of human societies and individuals rather than soothing them.

The last five years has been a journey of ups and downs, moments of inspiration and distraction, and challenges and ease that has brought me to the point where I am today. It is sometimes easy to forget that we are just one of many minds trying to live their lives and achieve their potential. The distribution of the properties that leads to success in these areas can never be fair and it is important to remember this initial game of chance while keeping focus on our own agency. In the end the only real thing we have is the present.

During my studies in Trondheim I have met many interesting people that has taken part in shaping me. With the knowledge, skills and experience I have now my attitude and perception would have been so different if I started studying now. I especially enjoyed my exchange to Tokyo University (東京大学) which has played a huge part in shaping my academic interests. Without it my thesis would have been quite different.

Table of Contents

List of Tables

List of Figures

Abbreviations

Symbol	=	definition
NN	=	Neural Network
GAN	=	Generative Adversarial Network
WGAN	=	Wasserstein GAN
BGAN	=	Bayesian GAN
NS GAN	=	Non Saturating GAN
JS GAN	=	Jensen-Shannon loss GAN
cGAN	=	Conditional GAN
AE	=	Auto Encoder
IS	=	Inception Score
FID	=	Fréchet Inception Distance

Chapter 1

Introduction

To tackle problems in the world learning is essential. Machine learning is the quest for automating this up until recently exclusive trait of sentient beings. We are going to consider a specific class of learning algorithms to complete this task called Generative Adversarial Networks (GAN). First we need to introduce some general classes for learning. Then we are going to present deep learning before we can start with the main topic of this thesis in chapter 2. This builds on some of the work in my project thesis, Sørby (2019).

Supervised learning is when you want to learn a task which you know the answer to. Essentially you have a dataset for which you know what you want the output value to be. For example you can have a dataset of pictures of dogs and cats. If you additionally know for each instance which pictures are of dogs and which are of cats you have a labeled dataset. This means that for each instance we can give a score of how well the algorithm is working. The point of this is that after learning you can now label new unseen data. However it is not necessary that an algorithm that can do well in training will be able to label unseen data correctly; which is called generalization. There is essentially no learning without generalization.

Usually you do not have a labeled dataset and labeling is a tedious and expensive task that has to be done by humans. Unsupervised learning tackles the problem of extracting useful information from an unlabeled dataset. This can be thought of as learning properties of the distribution the dataset is drawn from. GAN provides a way to do unsupervised learning by learning a distribution.

Then there is also a sort of intermediate version of learning which is called semi supervised learning. Here only a small part of the data has labels, but we want to use all of the data to train the model.

Understanding features in satellite images can be useful for many applications. For example for examining the population of unmapped urban areas like slums. In section 4.2 we present a model which can automatically transform satellite images into maps.

This chapter serves as a reference chapter for many of the concepts used later in this text. It will be necessary to understand the concepts presented in this chapter to fully understand the rest of the text, but it can be safely reviewed later.

1.1 Research interest in GAN

Initially we will briefly discuss how the GAN community has developed over the years. That includes what the current application areas are and what the ambitions for the future are. We close it off with some predictions.

The research interest in GAN has had an exceptional growth the last few years. This is probably caused by its impressive results and that it is seemingly a step towards more general artificial intelligence. The cumulative number of published papers with a named GAN variant can be seen in the graph in figure 1.1.

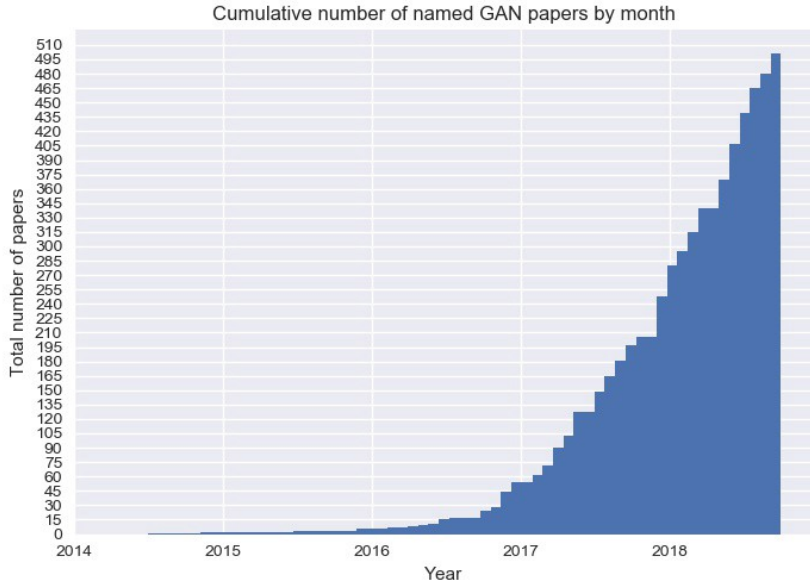


Figure 1.1: Cumulative number of unique named GAN variations published since its release compiled by Gavranovi (2019). At the time of writing there are 502 named GANs in the GAN Zoo.

There is an overwhelming amount of material being published about GAN. A big part of this research is based around applying theory from several different areas of mathematics and statistics to improve the capabilities, stabilize training and widen the application area. In chapter 3 we are going to explore several of these papers. However, those papers represent only the tip of the iceberg.

In the survey by Hong et al. (2019) they give an overview of most of the GAN variants available and their properties. This has been very useful for outlining this thesis.

GAN has wide ranging application areas and its results are quite impressive. The development has gone very fast as well. In the original GAN paper they generate some blurry faces, but in Karras et al. (2018), just 4 years later, they generate very realistic fake human faces.

1.2 Deep neural networks

We will now give a short introduction to some of the basics of deep learning and neural networks. For a more complete reference we recommend the book *Deep Learning* by Goodfellow et al. (2016). Deep learning is a machine learning subfield which is characterized by the use of large models with many layers; hence the name deep. The intention of this is to learn more complicated tasks like image or voice recognition. The quintessential model in deep learning is called a deep neural network (DNN), or just neural network (NN), and is in some sense a chain of linear predictors connected by nonlinear functions. It is essentially a way to parameterize functions.

Then we have some input x, which can be for example a color image in the shape of a three-tensor or a text string in the shape of a vector (one-tensor). However there are very many variations to this form and the following form is perhaps the simplest, namely the fully connected NN, which can be seen in a graphical format in figure 1.2 as well as in equation format

$$f(x) = \mu_L(W_L(\cdot) + b_L) \circ \ldots \circ \mu_1(W_1 x + b_1). \tag{1.1}$$

Here the network is represented as a chain of functions (which can also be thought and referred to as links or layers)

$$\begin{aligned} &f_1(x) = \mu_1(W_1 x + b_1), \\ &f_\ell(x) = \mu_\ell(W_\ell(f_{\ell-1}(x)) + b_\ell) \text{ for } \ell = 2, \ldots, L \end{aligned} \tag{1.2}$$

where μ_ℓ is some nonlinear activation function working elementwise on input of any dimension and W_ℓ and b_ℓ are the weight matrices and bias vectors respectively. Examples of common nonlinear activation functions are the rectifier $\mathbf{ReLU}(x) = \max(x, 0)$ and the sigmoid $\sigma(x) = \frac{1}{1+e^{-x}}$. For generality and convenience we are going to denote the parameters (or simply the weights) as θ and θ_ℓ for all parameters or the parameters of a specific layer respectively.

It is also possible to consider the neural network as a Directed Acyclic Graph (DAG) which is graphically supported by figure 1.2. In this view each node represents one component of the output from each layer and each edge or arrow represents multiplication with a weight. All the edges meeting in a node is summed over and then put into the activation function.

1.3 Additional layer structures

We can imagine replacing one of the layers with another structure that can take the same input and give an output of correct dimension to the next layer. It may even take input from other earlier layers like in residual nets or dense nets. There are many different structures that can be introduced as a layer in a neural network. It is mostly limited by our imagination. In fact the only constraints are that the layer at least preserves some of the information of the input and that it is weakly differentiable.

We call the collection of all the layers and their configuration parameters the architecture of the network. Different architectures are better at learning different tasks. There is

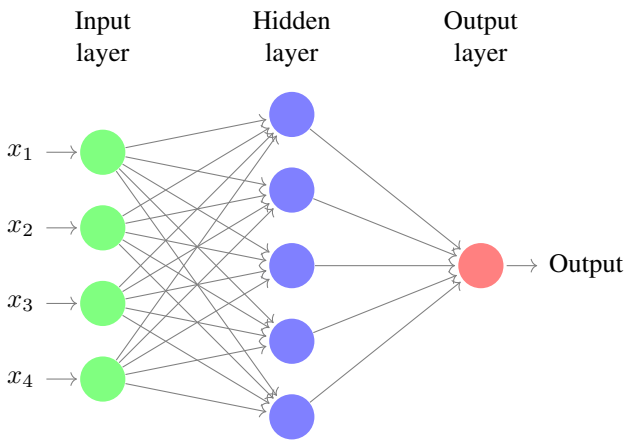


Figure 1.2: A visual representation of a one hidden layer fully connected neural network without bias vectors.

at the moment no theoretical basis for choosing the best architecture for a given task. The practice is mostly based on intuition and experimental experience.

In the following section we are going to present some layer structures that we are going to use in experiments or to explain the theory of different approaches.

1.3.1 Convolutional layers

One notable possibility for a layer is a convolutional layer. A convolutional layer can be seen as a less connected layer than the fully connected layer where only the local relationships of the input are examined. The concept revolves around doing a convolution operation with some kernel K (also called filters), which corresponds to the weights, for some input x. For example we can consider a two-dimensional kernel, $K \in \mathbb{R}^{N\times M}$, which normally means that the input only has one channel (like a black and white image). An illustration of this can be seen in figure 1.3. This means that the kernel sweeps over the input one neighborhood of size $N \times M$ at a time. In the 2-dimensional case we call the output matrix S and the equation of each element of S becomes

$$S = (K * X)(i, j) = \sum_{n}^{N} \sum_{m}^{M} X(i - n, j - m)K(n, m). \tag{1.3}$$

Then as usual we apply an activation function elementwise to produce the output $f_\ell(X) = \mu_\ell(S)$. In this case the output is in the shape of a matrix (two-tensor) not a vector. For three-tensor input (like a color image) the output will also be a three-tensor. To be able to connect this to the next layer one can either flatten the tensor into a vector and possibly lose structural information or keep the structure and let the next layer handle higher order inputs. Also note that again we apply some nonlinear activation function μ_l. A NN containing at least one convolutional layer is usually called a *Convolutional Neural Network* (CNN).

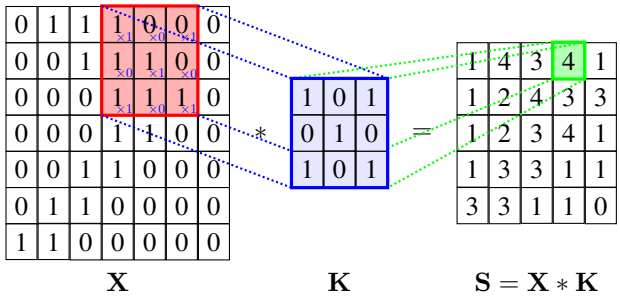


Figure 1.3: An illustration of a 2D convolution layer operating on a matrix input.

Additional configuration parameters include number of filters, padding, and strides. Filters are the number of simultaneous kernels to be applied to the input and represent the size of the output space. Padding means adding a boundary of zeros around the input when performing the convolution. It is necessary when you want the output to have a specific shape. For example if you want it to retain the original shape of the input you would add extra zeros to the edges. Strides is the number of spaces between each place the filter is applied.

1.3.2 Recurrent units

A NN where the output of a layer is sent back as input to an earlier layer and then iterated arbitrarily many times is called a *Recurrent Neural Network* (RNN). This is necessary when you want to produce sequences of arbitrary length as for example in text generation. It can also be used to analyze time series data.

1.3.3 Dropout

Sometimes we want to introduce noise in the network to force it to be more robust. This can be done by randomly turning off some nodes when forward propagating the network which is called dropout (see figure 1.4); first introduced by Hinton et al. (2012). Note that this only affects the training of the network and not when we are using it in practice.

This will force the network to not rely on a few nodes to to make use of all the nodes. It is thought that this will prevent the network from relying on unreliable features in the data. This can for example be a particular bias in the dataset. If you have two classes, say images of Huskys and Retrivers, and all the images with Huskys has snow in the background then it is much easier for the network to learn to simply detect the snow than actually recognise the Husky features.

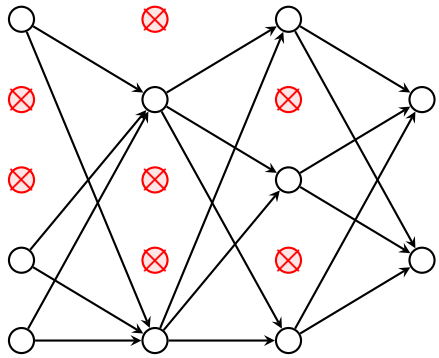


Figure 1.4: Visualization of dropout on three fully connected layers.

Later we will use dropout to introduce noise in the generator of a GAN.

1.3.4 Max pooling

It might not always be the best strategy to collect all the information from the previous layers. Sometimes you would want to filter out the input with the presumed highest significance. A max pooling layer is structurally quite similar to a convolutional layer. It will scan over the input tensor and give a shrunken down output similar to a convolutional layer, but instead of computing a weighted sum we simply take the maximum of each neighbourhood.

1.3.5 Batch normalization

A problem with training neural networks is that due to the large datasets and computational demands have to reduce the training updates to only smaller batches of data at a time. This means that the distribution of outputs from a layer within a batch may vary a lot during training. To ensure more stability during training it is possible to normalize the outputs with a layer called batch normalization by Ioffe and Szegedy (2015). This subtracts the mean and divides by the standard deviation for every output within the batch. Although the reasons for the effectiveness of this technique is debated it has been empirically shown to improve stability and performance of training.

1.4 Training

The network can be trained on a dataset $\{(x_i, y_i)\}_{i=1}^{n}$ (or rather a training set), where x_i are called samples and y_i are called labels, such that $f(x_i) = y_i$ for all i or some probability that the sample has a certain label. We can imagine that all samples are collected from a greater sample space $\mathcal{X}$ and all labels from a greater label space $\mathcal{Y}$, and there is a probability distribution $\mathcal{D}$ over $\mathcal{X} \times \mathcal{Y}$. Note that while in general the space of possible NNs and $\mathcal{X}$ and $\mathcal{Y}$ is infinite we sometimes need to restrict ourselves to finite spaces when doing analyzes.

Additionally we have a separate test set $\{(x_i, y_i)\}_{i=n+1}^{n+n_{\text{test}}}$ which we will use to verify the generalization ability of the network. The setting where all the labels of the samples are known is called supervised learning. Training or learning differs from normal optimization in that we wish to optimize the performance on the test set indirectly by optimizing the performance on the training set. We can not know anything about the generalization ability of the network without a separate test set which has not been used in training. An important note from this is that when we are learning we do not know the underlying distribution of data.

For simplicity and notational convenience we will sometimes use slightly different versions of the loss function. This will be indicated with different indexes on the function or different arguments to the function. Remember that we denoted the parameters of the network as θ. To train the network we first need to compute some loss function $\mathcal{L}(\theta, x_i, y_i)$, which in this case applies to a single sample. This will give a score of the accuracy of the network on a given sample. The training problem then becomes to minimize the loss

$$\min_{\theta} \mathcal{L}(\theta, x_i, y_i) + R(\theta) \tag{1.4}$$

where $R(\theta)$ is some optional regularizer. Regularizers may force the model to generalize better, but for the remainder of this chapter we are going to omit it for simplicity. A popular choice for loss function is the cross entropy, which compares the entropy of two probability distributions, and follows from the maximum likelihood principle. In this case the empirical distribution from the training set and the prediction distribution from the model is compared

$$\mathcal{L}(\theta) = -\mathbb{E}_{(x,y)\sim\mathcal{D}_{\text{data}}} \left[\log(p_{\text{model}}(y|x))\right]. \tag{1.5}$$

This way we reduce a machine learning problem to a normal optimization problem by introducing the empirical distribution $\mathcal{D}_{\text{data}}$. There are however many other candidates for loss functions.

Then to update the weights accordingly it is normal to use a variant of *Stochastic Gradient Descent* (SGD) and backpropagation to compute the gradient for each layer. The training set is split into smaller batches of size m that we can use to calculate an approximation of the gradient; a stochastic gradient. The samples in each batch can be drawn randomly to reduce bias from the ordering of the samples in the dataset. It requires that the total loss over all training samples, $\mathcal{L}_{\text{total}}(\theta, \{x_i\}, \{y_i\})$, is an average over the loss for each individual sample. This means that in general it follows the form

$$\mathcal{L}_{\text{total}}(\theta, \{x_i\}, \{y_i\}) = \frac{1}{m} \sum_i \mathcal{L}(\theta, x_i, y_i) \tag{1.6}$$

$$\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}_{\text{total}}(\theta, \{x_i\}, \{y_i\}) \tag{1.7}$$

where the hyperparameter η is called the learning rate and m is called the batch size. This is then performed iteratively over the entire training set, where one iteration over the training set is called an epoch.

Chapter 2

Generative adversarial networks

When you have data, X, but no labels, Y, you can use unsupervised learning to extract value from it. Unlabeled data is the default state of all data found in nature; it is not until we want to extract some meaning or causal link from the data that we can associate labels to that data. Adding those labels to a dataset often needs to be done by humans since some of the point of machine learning is to automate difficult or tedious labeling tasks. This is expensive and time consuming. There is however much useful information that can be extracted from a dataset without having a specific prediction objective in mind. One interesting property can be to be able to generate new samples from a dataset. Generative Adversarial Networks (GAN) first introduced by Goodfellow et al. (2014) makes it possible to generate new samples similar to those found in a dataset.

It consists of two NNs, the generator G and the discriminator D and an adversarial training structure. The intuitive interpretation is that they are adversaries in a game where the generator tries to fool the discriminator and the discriminator tries to catch the generator.

2.1 An intuitive description

To give a more intuitive basis for understanding GAN consider the following story. The generator is a con artist making fake Monet paintings. She wants to sell them at art galleries, but she has to fool the art critic, the discriminator, to do so. The discriminator will walk through the gallery and give a score to all the paintings of whether he thinks they are real Monet paintings or created by a con artist. If they get a too low score no one will buy them. The generator wants to earn as much money as possible, so she needs to improve her skills so that she best can fool the discriminator.

Initially the generator does not know how to paint at all, let alone how to paint Monet paintings. She has in fact never seen a Monet painting. Luckily the discriminator does not know how to recognise Monet paintings either. So the generator produces a batch of paintings with random strokes and content and sends them off to the gallery for judgement. At the same time a batch of real Monet paintings arrive at the art gallery. Now the

inexperienced discriminator has to walk around and give scores to all the paintings. When he is finished he will give a report to his supervisor. The supervisor always knows which paintings are real and fake. She will score the discriminators performance and send him a report with all the errors he made on both the real and the fake images. The supervisor is secretly friends with the generator as well and will give her a score and a report as well. The discriminator and generator both read their reports and find ways to improve their technique. Then the process is repeated. The generator makes a batch, the discriminator will score them and they get a new report from the supervisor. Over time as this process is repeated many times the discriminator and generator get better and better at their job each time trying to outsmart the other.

In this story the supervisor is analogous with the loss function and her report with backpropagating the gradient. This is something we will describe in further detail in the following chapters.

2.2 An example using MNIST

As an example consider the MNIST dataset. It consists of 70 000 (28x28) back and white images of handwritten digits (0 to 9). This has been used as a benchmarking dataset for decades and is the go-to dataset when exploring a new method. Each digit is written in its own way, but overall the digits define a sort of MNIST digit font. This is a qualitative feature of the dataset. It does not contain every way of writing a digit which means that it has a certain bias. How can we learn this qualitative feature of digits?

We can think of a digit as certain shape formed onto a sheet of paper. For example consider the digit one. It can be written as simply a vertical line or more elaborately with extra lines at the top and bottom. Manually designing a recognition algorithm for recognising this digit is practically impossible - and futile. Humans nonetheless have little difficulty learning to recognize this digit and then being able to write it themselves. This new digit will not be a perfect copy of the original and it is not intended to be. Rather the human has learned which features of the digit are important and which can be varied. It must therefore have an internal notion of the distribution of these digits.

Now lets try to make our generator learn how to generate digits that are not exact copies of any of the original digits in the dataset, but still would be recognized as digits by a human. As in any learning algorithm we will start with some initial random weights, then apply some input and compute the corresponding output. This output is then put into a loss function. However we have no way of defining the loss function for whether an output looks like a digit from MNIST. The closest thing to such a loss function would be a human. However it would not be able to provide any meaningful gradient for the generator to learn. The GAN solution is to learn the loss function as well.

In the context of the story in the previous section the discriminator needs to learn how to detect real Monet paintings. Learning the loss function means learning a classifier that can tell whether the generator is producing good digits or not. This classifier, the discriminator, can be thought of as a loss function for the generator. As in human learning we would want the discriminator to behave pedagogically. It should not only tell the generator what is correct or wrong, but provide it with helpful feedback so that it can improve. That means providing meaningful gradients for backpropagation. We shall see

in section 2.5 that this is not always the case and is actually a challenge when training GANs in practice.

2.3 Formal definition

In the following formulation we will say that the discriminator takes in a sample and gives out a guessed probability that it comes from the real dataset; essentially $D : \mathcal{X} \to [0, 1]$. That means that if $D(x) = 0$ the discriminator is completely sure that x is fake. If $D(x) = 1$ it is completely sure that x is real. We are going to denote the parameters of the discriminator as θ_d when necessary.

The generator on the other hand takes noise, $z \sim \mathcal{D}_z$, for example $\mathcal{D}_z = \mathcal{N}(0, \sigma^2 I)$, as input and produces samples as output, $G : \mathcal{D}_z \to \mathcal{X}$. The noise gives randomness to the otherwise deterministic function G. We call the input vector z a latent vector. There are many ways to design this latent vector space which we will discuss later. As with the discriminator we denote the parameters of the generator as θ_g when necessary. For ease of notation we are going to use just θ when we are referring to the parameters of the entire model.

First we will consider how to tell if the generator is doing a good job. We want the generator to try to fool the discriminator as often as possible. What we could do is simply maximize the probability, $D(G(z))$, that the discriminator guesses that generated samples are from the real distribution. This is equivalent to minimizing $1 - D(G(z))$ which will be convenient later. Then on the other hand we want the discriminator to classify generated samples as fake which means maximizing $1 - D(G(z))$. Applying expectation to this allows us to define a generator objective

$$\min_G \max_D \mathbb{E}_{z \sim \mathcal{D}_z}[1 - D(G(z))]. \tag{2.1}$$

This formulation has a number of problems, but we are going to see a variant of this in section 3.2. For example the output ranges only from 0 to 1 which is not beneficial for optimization. A typical statistical approach would be to take the logarithm of the probability and mirrors the log likelihood formulation from statistics. Then the output will now range over $(-\infty, 0]$. An information theoretic way to see this is that we are trying to fit a distribution to our data and according to the maximum entropy principle we should choose the distribution with the highest entropy. In conclusion we now have the complete objective for the generator

$$\mathcal{L}_g(G, D) = \mathbb{E}_{z \sim \mathcal{D}_z}[\log(1 - D(G(z)))] \tag{2.2}$$

which the generator needs to minimize

$$\min_G \mathcal{L}_g(G, D). \tag{2.3}$$

As you may note we have not yet included the dataset in the formulation which will be essential. It is enough to train the generator to fool the discriminator and the discriminator to identify the generator's mistakes if the discriminator already has a good intuition of how samples from the dataset look like. We could for example train the discriminator

beforehand to identify the samples from the dataset. However we do not have any good counter examples to the samples of the dataset.

Let's say you had a perfect discriminator before starting to train the generator. There is no reason to expect that this is the best way to train the generator. Rather we let the discriminator and generator be equally bad at the start of training and then as training progresses it will be better at detecting. What the optimal discriminator generator relationship during training is not known. We let the discriminator learn to detect samples from the dataset alongside trying to catch the generated samples. We can formulate the discriminator's objective as

$$\mathcal{L}_d(G, D) = \mathbb{E}_{x\sim\mathcal{D}_{\text{data}}}[\log D(x)] + \mathbb{E}_{z\sim\mathcal{D}_z}[\log(1 - D(G(z)))]. \tag{2.4}$$

which is going to be maximized

$$\max_D \mathcal{L}_d(G, D). \tag{2.5}$$

We may for example choose to learn the generator's objective (2.2) or the discriminator's objective (2.4) more often than the other. It is not know which strategy is best, but experimental experience and some intuition tells us that equal amount of learning updates seems best.

These two objectives combines into the following loss function

$$\mathcal{L}_{JS}(D, G) = \mathbb{E}_{x\sim\mathcal{D}_{\text{data}}}[\log D(x)] + \mathbb{E}_{z\sim\mathcal{D}_z}[\log(1 - D(G(z)))]. \tag{2.6}$$

This particular choice of GAN loss is sufficient. As we shall see later maximizing this over D is equivalent to approximating the Jensen-Shannon divergence between the induced distribution of the dataset and the generator. There are many alternatives to this formulation however that might solve issues during training. One of them is the Wasserstein metric which we will discuss in section 3.2.

The formulation we have constructed so far is equivalent to playing a zero sum minimax game (see Maschler et al. (2018)) with the formulation

$$\min_G \max_D \mathcal{L}_{JS}(G, D). \tag{2.7}$$

Here $\mathcal{L}_{JS}(G, D)$ can be interpreted as the joint loss function for the generator and the discriminator. However not all formulations of GAN are minimax games.

Note that it is also possible to add a regularizer R_θ to the formulation (2.7). This is some form of restriction on the parameters to ensure that the result has some properties we want. We are going to see examples of this in table 3.2, section 3.2 and section 3.6.

This is very useful because the generator learns the distribution of the data. It essentially learns to sample new versions of the data making it possible to use much fewer examples in training. For example given a dataset of images the generator can generate new images similar to the data. This also means that the model is not dependent on having labeled examples; it can do unsupervised learning.

We have defined how the GAN works in terms of loss functions and optimization. In order to learn a specific task it is necessary to choose appropriate architectures for the generator and discriminator. To train a GAN we can compute gradients and update the

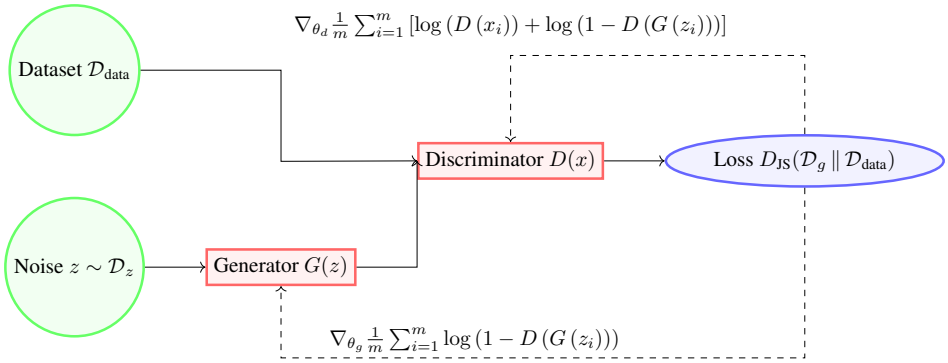


Figure 2.1: A basic setup for a generative adversarial model using the JS-loss.

parameters with backpropagation. There are several algorithms for updating the parameters like SGD or Adam. For generality we are going to denote this update step as $\mathcal{A}(\theta, v)$ where θ are the parameters and v is a gradient. For example the SGD update step is

$$\mathcal{A}(\theta, v) = \theta - v \tag{2.8}$$

where v is a stochastic gradient.

In algorithm 1 we outline the normal training steps for updating the generator and discriminator. Note that we can configure the discriminator to be updated more often than the generator. We also visualize the training in figure 2.1. Full lines indicate inputs and outputs and dashed lines indicates backpropagating gradients.

2.4 Basic theoretical results

In this section we will present the theoretical results given in the original paper. Note that these results are placed in an idealized setting and assumes no limitations on the representability of NNs. We will use concepts from information theory in some results. There is a reference for this in appendix B.

The generator defines implicitly a distribution $\mathcal{D}_g$ of generated samples $G(z) = \tilde{x} \sim \mathcal{D}_g$ where $z \sim \mathcal{D}_z$. $\mathcal{D}_g$ represents a probability density function for the random variable, $\tilde{x}$, of samples produced by G.

Let $\mathcal{D}_{\text{data}}(x)$ be the probability density function induced by the dataset. This distribution represents the larger idea of the data. It fills in the gaps between samples of the dataset. In the case of MNIST we would assume that this is the distribution of digits, but this is not obvious. With the metric used we are implicitly choosing an approximation to the maximum entropy principle. It will be the distribution representing the data that has the least amount of prior hypotheses.

The optimal discriminator for a fixed generator can be found by considering the inte-

Data: The learning rate η, the dimensionality n_z of the latent noise $\mathcal{D}_z$, initial weights and architecture for G and D, the number of epochs n_{epochs} and batches per epoch n_{batches}, the number steps for training the discriminator for every update of the generator n_{disc}, and the batch size m.

for $n_{epochs} \cdot n_{batches}$ *iterations* **do**
  First train the discriminator.
  for n_{disc} *iterations* **do**
    Sample m noise samples $\{z_i\}^m \sim \mathcal{D}_z$.
    Sample m data samples $\{x_i\}^m$.
    Update the discriminator by backpropagating the gradient
    Note that since we are maximizing we need to change the sign of the gradient
    $v \leftarrow -\nabla_{\theta_d} \left[\mathcal{L}_d(G, D) + R_\theta\right]$
    $\theta_d \leftarrow \mathcal{A}(\theta_d, v)$.
  end
  Then train the generator.
  Sample m noise samples $\{z_i\}^m \sim \mathcal{D}_z$.
  Update the generator by backpropagating the gradient
  $v \leftarrow \nabla_{\theta_g} \left[\mathcal{L}_g(G, D) + R_\theta\right]$
  $\theta_g \leftarrow \mathcal{A}(\theta_d, v)$.
end

Algorithm 1: General algorithm for training GANs. We use general loss functions $\mathcal{L}_d$ for the discriminator and $\mathcal{L}_g$ for the generator. We include a regularizer R_θ for compatibility with some methods.

gral version of (2.6) with appropriate measures

$$\begin{aligned}\mathcal{L}_{JS}(G,D) &= \int \log(D(x))\,\mathcal{D}_{\text{data}}(x)dx + \int \log(1-D(G(z)))\,\mathcal{D}_z(z)dz \\ &= \int \log(D(x))\,\mathcal{D}_{\text{data}}(x) + \log(1-D(x))\,\mathcal{D}_g(x)dx.\end{aligned} \tag{2.9}$$

The inside of this integral is a function of the form $y \to \alpha\log(y) + \beta\log(1-y)$ with respect to D where $y \in [0,1]$. This obtains its maximum at $\frac{\alpha}{\alpha+\beta}$ which means that the optimal discriminator for this formulation is

$$D_G^*(x) = \frac{\mathcal{D}_{\text{data}}(x)}{\mathcal{D}_{\text{data}}(x) + \mathcal{D}_g(x)}. \tag{2.10}$$

This is essentially the distribution where the discriminator is most likely to classify a given sample correctly. To calculate this value the discriminator needs to have knowledge about both the distribution of the dataset and the generator. Using this discriminator does not necessarily provide a good training update to the generator.

We also know what the optimal end state of the game is. Putting the optimal discriminator into the game of (2.7) yields

$$\begin{aligned}C(G) &= \max_D \mathcal{L}_{JS}(G,D) \\ &= \mathbb{E}_{x\sim\mathcal{D}_{\text{data}}}\left[\log\frac{\mathcal{D}_{\text{data}}(x)}{\mathcal{D}_{\text{data}}(x)+\mathcal{D}_g(x)}\right] + \mathbb{E}_{x\sim\mathcal{D}_g}\left[\log\frac{\mathcal{D}_g(x)}{\mathcal{D}_{\text{data}}(x)+\mathcal{D}_g(x)}\right] \\ &= D_{KL}(\mathcal{D}_g(x)\|\,\mathcal{D}_{\text{data}}(x)+\mathcal{D}_g(x)) + D_{KL}(\mathcal{D}_{\text{data}}(x)\|\,\mathcal{D}_{\text{data}}(x)+\mathcal{D}_g(x)).\end{aligned} \tag{2.11}$$

Note that the final line of the above expression is very close to the definition of the Jensen-Shannon divergence. If we add $2\log 2$ we get exactly the formulation in (6). This is possible because of logarithm rules and that the expectation of a constant is the same constant. This means that we can write

$$C(G) = -\log 4 + D_{JS}(\mathcal{D}_g\,\|\,\mathcal{D}_{\text{data}}). \tag{2.12}$$

Since the JS divergence is always positive this function has a global optimum when $\mathcal{D}_g = \mathcal{D}_{\text{data}}$ which is $C^* = -\log 4$. This means that the generator has learned to perfectly represent the data distribution.

In terms of game theory the global optimum is called the Nash equilibrium. This is when there is no benefit for either party of the game to change strategy. At this point the discriminator will be no better than a random guess i.e. $D_G^*(x) = \frac{1}{2}$.

At last we also know that the training algorithm will converge in an idealized setting. This is possible to show if we consider updates on the distribution $\mathcal{D}_g$ itself and always let the discriminator converge to the optimal form. If $\mathcal{L}_{JS}(G,D) = U(\mathcal{D}_g, D)$ then U is convex in $\mathcal{D}_g$. Then by considering the subderivatives of U with regards to each argument separately, we can apply gradient descent to $\mathcal{D}_g$ with an optimal D. This is known to converge with sufficiently small steps. As stated before these results do not hold in practice because of limitations in the NNs representability.

Two natural questions to ask remain. Is the distribution we induce from the dataset really the same as the idea we wanted to represent with the dataset? For example given a dataset of faces, like CelebA, it is not obvious what kind of general idea we want to represent with this dataset. Is it a special kind of faces that are represented by this dataset or is it all possible human faces? The extension of the dataset $\mathcal{D}_{\text{data}}$ is in reality induced by the architecture and training of the discriminator. The discriminator determines the rule for what looking like data from the dataset means and thereby implicitly defining what $\mathcal{D}_{\text{data}}$ is. It is therefore important to keep in mind that this might not be the distribution we intended.

Secondly and related is the question whether the generator is fully capturing the distribution or just a smaller mode. In an ideal setting the generator has an infinite representability and can approximate any data with arbitrarily small details. In practice this is far from the truth and if the generator has too many parameters there is a risk it might just memorize the dataset more or less exactly. This would for most loss functions mean that the GAN has converged, but it results in a useless generator. It is therefore necessary that learning to generate new samples is easier than to remember the dataset in the context of a learning algorithm.

2.5 Difficulties with training GAN

As discussed previously the GAN has a global optimum when the distribution induced by the generator is the same as the distribution induced by the dataset. However in practice there are a number of things that can go wrong and hinder the GAN to reach the global optimum. This is mostly related to the discriminator and generator reaching a local optimum. There are some specific named problems that are worth mentioning.

2.5.1 Mode collapse

If the generator starts to produce samples from only one class, for example just eights in the case of the MNIST dataset, we call this mode collapse. This can happen if it is a much easier task for the generator to learn to generate only one type of examples than to generate all kinds. This is essentially a local minima in the game between the generator and discriminator. The discriminator will give a high score to the generator because it produces high quality eights (samples of one kind). The generator will not have an incentive to generate samples of a different kind because these will be of a poorer quality and not give a good score in the discriminator. This is not the behavior we want the generator to exhibit, but it might be easier to achieve than to actually learn the distribution. We need to configure the loss function and hyperparameters such that this is discouraged.

2.5.2 Vanishing gradient

The gradient might be really small and not update the weights significantly. This usually means that our training method has reached a local minimum or is in an area where there is only a tiny gradient. The cause of this might be that the generator or discriminator lacks the representability to produce better samples. Another possible cause could be

if the generator or discriminator is much better than their opponent. For example the discriminator might be nearly perfectly classifying the presented samples, but in such a way that the gradient is small.

2.5.3 Failure to converge

The parameters of the GAN might start to oscillate, get really large or small, or generally fail to converge. This is a known problem from normal optimization and might be caused by gradients growing towards infinity or start to oscillate.

There is little real guarantee that GAN training will converge with standard training updates like Stochastic Gradient Descent. This is because the typical training updates for the generator and discriminator might not actually bring the entire GAN game towards convergence. For example imagine that the discriminator and generator are trained with strategies that are directly opposite. The discriminator might be adjusted for the exact strategy the generator is applying leading them to cancel out each others improvement. If this is in addition not bringing the overall game closer to convergence they might be stuck in a loop. A simple example of this is for example if one agent, like the discriminator, is minimizing xy with gradient descent and the other is minimizing $-xy$ with gradient descent. This will only lead to oscillation and never hit the global optimum at $(x, y) = (0, 0)$.

2.6 Applications

In this section we are going to give a short introduction of the utility of GANs. The most useful consequence of training a GAN is that we obtain a generator for a distribution based on our dataset. This means that we can generate new unseen samples from the dataset. To effectively train NNs with supervised learning it is required to have large labeled datasets. This is usually not available. GANs can therefore be used as a dataset augmentation method. For example when training NNs on image classification it is usual to augment the dataset by adding translated, scaled or somewhat distorted versions of the original images to the dataset. This makes the NN more robust for these kinds of small changes in the input. Adding GAN generated images would allow for even more variations in the original samples.

There are several ways to configure GAN to do variants of supervised learning. This can involve turning the discriminator into a classifier as well. In this scenario the discriminator not only classifies whether samples are generated or not, but also assigns them a label. A special label is then given to samples that are considered to be generated. This means that GAN can be used in normal supervised settings as well. It may even generalize better since the classifier, the discriminator, is also trained on all the generated samples. This means that we may require fewer training samples to succeed.

Another related application is in semi supervised learning. We can not compute a supervised loss for these samples since we do not have the correct label for these samples. To use the supervised learning framework we can rather estimate the labels of those samples. We are going to see an example of this in section 3.7.2.

As we shall see in section 3.4 we can extend the GAN definition to let the generator also take data as input. This can for example be pictures that we want to convert to a different format or style. We can also use it to remove noise from an input signal or increase the resolution of images.

Chapter 3

Extensions and innovations

In this chapter we want to discuss some of the properties and possibilities with adversarial training of generative models found in published papers. Many of these techniques are orthogonal in the sense that they can be combined without interfering with each other.

We are going to present several improvements and variations to GAN. We are going to look at a different loss function for GAN in the Wasserstein GAN section 3.2. Then we are going to consider several ways of regularizing a GAN to achieve different behaviour. This is necessary for Wasserstein GAN and InfoGAN. Then we are going to consider a more application oriented path with conditional GAN in section 3.4, Image to image GAN in section 3.4.1, and Cyclic GAN in section 3.5. All of these represents an extension to the original to increase the control and utility of GANs. Bayesian GAN in section 3.7 provides inference and robustness by not restricting itself to a single point in the parameter space, but rather a distribution of parameters. The gradient layer method in section 3.3 provides a way to fine tune outputs with functional gradients.

Thorough theoretical investigation is necessary to improve training of GANs as well as developing new techniques. In the paper by Arjovsky and Bottou (2017) they establish some deeper theoretical results for GANs.

Somewhat related to GAN is the Auto Encoder (AE). It learns to break down a sample, like an image, into a lower dimensional representation. Then it has to recreate it from that lower dimensional representation. It is analogous with compression.

Name	Discriminator Objective	Generator Objective
JS GAN	$\mathcal{L}_d^{\text{GAN}} = -\mathbb{E}_{x\sim\mathcal{D}_{\text{data}}}[\log D(x)] - \mathbb{E}_{z\sim\mathcal{D}_z}[\log(1 - D(G(z))]$	$\mathcal{L}_g^{\text{GAN}} = \mathbb{E}_{z\sim\mathcal{D}_z}[\log(1 - D(G(z))]$
NS GAN	$\mathcal{L}_d^{\text{NSGAN}} = -\mathbb{E}_{x\sim\mathcal{D}_{\text{data}}}[\log D(x)] - \mathbb{E}_{z\sim\mathcal{D}_z}[\log(1 - D(G(z))]$	$\mathcal{L}_g^{\text{NSGAN}} = -\mathbb{E}_{z\sim\mathcal{D}_z}[\log(1 - D(G(z))]$
LS GAN	$\mathcal{L}_d^{\text{LSGAN}} = -\mathbb{E}_{x\sim\mathcal{D}_{\text{data}}}[(D(x) - 1)^2] - \mathbb{E}_{z\sim\mathcal{D}_z}[(1 - D(G(z))^2]$	$\mathcal{L}_g^{\text{LSGAN}} = -\mathbb{E}_{z\sim\mathcal{D}_z}[(1 - D(G(z))^2]$
WGAN	$\mathcal{L}_d^{\text{WGAN}} = \mathbb{E}_{x\sim\mathcal{D}_{\text{data}}}[D(x)] - \mathbb{E}_{z\sim\mathcal{D}_z}[D(G(z))]$	$\mathcal{L}_g^{\text{WGAN}} = -\mathbb{E}_{z\sim\mathcal{D}_z}[D(G(z))]$
BEGAN	$\mathcal{L}_d^{\text{BEGAN}} = \mathbb{E}_{x\sim\mathcal{D}_{\text{data}}}[\\|x - \text{AE}(x)\\|_1] - k_t\,\mathbb{E}_{z\sim\mathcal{D}_z}[\\|G(z) - \text{AE}(G(z))\\|_1]$	$\mathcal{L}_g^{\text{BEGAN}} = \mathbb{E}_{z\sim\mathcal{D}_z}[\\|G(z) - \text{AE}(G(z))\\|_1]$

Table 3.1: Different varieties of GAN loss functions. BEGAN uses an autoencoder as discriminator. AE stands for autoencoder.

There are numerous GAN loss functions. We include a limited overview compiled by Lucic et al. (2017) in table 3.1. The Non Saturating GAN (NS GAN) is similar to the JS GAN, but the generator tries to maximize its objective not minimize. It is therefore not a minimax game. To signify this difference JS GAN is also referred to as MM (MiniMax) GAN. BEGAN uses an autoencoder as a discriminator which tries to compress and reconstruct the input. We therefore need to compare the reconstructed output with the input.

Just as important is different ways of regularizing GANs which can be seen in table 3.2. This includes techniques for doing supervised learning where we need to compare generated outputs to ground truths. In that case we can calculate pixelwise distance using absolute (L1) or squared (L2) distances.

Name	Discriminator Objective
L1	$\mathbb{E}_{(x,y)\sim\mathcal{D}_{\text{data}},z\sim\mathcal{D}_z}[\|y - G(x,z)\|_1]$
L2	$\mathbb{E}_{(x,y)\sim\mathcal{D}_{\text{data}},z\sim\mathcal{D}_z}[\|y - G(x,z)\|_2]$
WGAN-GP	$\mathbb{E}_{\tilde{x}\sim\mathcal{D}_g,x\sim\mathcal{D}_{\text{data}}}[(\|\nabla D(\alpha x + (1-\alpha)\tilde{x})\|_2 - 1)^2]$
DRAGAN	$\mathbb{E}_{\tilde{x}\sim\mathcal{D}_{\text{data}}+\mathcal{N}(0,c)}[(\|\nabla D(\tilde{x})\|_2 - 1)^2]$
InfoGAN	$\mathbb{E}_{x\sim\mathcal{D}_{\text{data}},c'\sim p(c)}[\log Q(c'\|x)] + H(c)$

Table 3.2: Different varieties of GAN regularizers. For L1 and L2 we are in a supervised setting.

3.1 Performance metrics

You cannot trust the value of the loss function to determine if the generator has produced good samples or not. As we have seen many things can go wrong while training that can make it look like the training is going well, while in fact the samples produced are of poor quality. It is often necessary for manual inspection of generated samples to conclude whether the generator is behaving as desired or not. This is however tedious and subjective. It is much more convenient to have a numerical value. There has therefore been suggested several metrics for computing a GAN score. The two most common are the Inception Score (IS) by Salimans et al. (2016) and the Fréchet Inception Distance (FID) by Heusel et al. (2017).

3.1.1 Inception score

This score was developed with two considerations in mind. The conditional label distribution of samples containing meaningful objects should have low entropy. The second consideration was that the variability of the samples should be high.

To calculate the distribution of labels of generated data, $p(y)$, and labels conditioned on generated data, $p(y|x)$, we employ *Inception Net* trained on the Image Net dataset. In the case where you want to compare methods in a domain which is not like Image Net you could use another dataset. The inception score is then given by

$$\mathrm{IS}(G) = \exp(\mathbb{E}_{x\sim\mathcal{D}_g}[D_{\mathrm{KL}}(p(y|x)\|p(y))]). \tag{3.1}$$

This is however not a metric. It has been shown that this score correlates well with human evaluations.

3.1.2 Fréchet Inception Distance

The Fréchet Inception Distance (FID) embeds generated samples into the feature space given by a layer of Inception Net. It considers the feature space as a multivariate Gaussian distribution. We can then compute the mean and covariance of the distributions generated by the dataset and the generator. The FID is then simply a comparison between these two distributions given by the Fréchet distance as follows

$$\mathrm{FID}(x, g) = \|\mu_x - \mu_g\|_2^2 + \mathbf{tr}\left(\Sigma_x + \Sigma_g - 2(\Sigma_x\Sigma_g)^{\frac{1}{2}}\right). \tag{3.2}$$

The FID can also detect intra-class mode dropping. This is when the generator produces only a few types of each class. This could give a good IS, but gives a bad FID.

A problem with both of these scores is that they do not account for overfitting the data. A GAN that memorizes the training data, but can not produce any new samples would receive perfect score from both of these methods.

3.2 Wasserstein GAN

Using a different metric may solve many of the difficulties with training GANs. In this section we are going to construct the Wasserstein metric and demonstrate its useful properties as a loss function when training GANs. To do this rigorously and general we need to introduce some concepts from topology and optimal transport. Defining every concept thoroughly is however outside the scope of this text. For a more complete reference the book *Optimal transport: old and new* by Villani (2008) is a good option.

3.2.1 The Wasserstein metric

First there is the Borel set which is any set that can be constructed by taking countable unions, intersections or set difference of open sets in a topological space. Combined these sets form the Borel σ-algebra which is the smallest algebra containing all the open sets of a topological space.

Given a metric space (Ω, d_Ω) where $d_\Omega(x, y)$ is a metric and Ω is such that all probability measures are Radon measures. A Radon measure is a general measure with the following properties.

Definition 3.2.1. *Let m be a measure on a σ-algebra of Borel sets of a Hausdorff topological space X. A measure m is inner regular if for every open set $U \subset \Omega$*

$$m(U) = \sup_{\textit{Compact subset } K \textit{ of } U} m(K)$$

the supremum of every compact subset of U. It is outer regular if for any Borel set B,

$$m(B) = \inf_{\text{Open set } U \text{ containing } B} m(U)$$

the infimum of $m(U)$ over all open sets containing B. It is locally finite if every point of Ω has a neighbourhood U where $m(U)$ is finite. Then m is called a Radon measure if it is both inner regular, outer regular, and locally finite.

The Wasserstein distance between two Radon probability measures, μ and ν, is in general

$$W_p(\mu, \nu) := \left(\inf_{\gamma \in \Gamma(\mu,\nu)} \int_{M \times M} d_\Omega(x, y)^p \, \mathrm{d}\gamma(x, y) \right)^{1/p} \tag{3.3}$$

where $\Gamma(\mu, \nu)$ is the set of all joint probability measures where the marginals are μ and ν respectively. It can be shown that this satisfies all the axioms for a metric and therefore defines a metric space.

This can be reformulated using expected value, and setting $p = 1$, to the earth mover (EM), or Wasserstein-1, metric. The name earth mover comes from the intuition that we are measuring the minimal work of shuffling one pile of probability to another. Work in this context being the mass and distance each piece of earth needs to be moved. Let p_g and p_r be two arbitrary probability density functions defined on Ω. Then we can consider the earth mover metric

$$W(p_g, p_r) = \inf_{\mathcal{D} \in \Pi(p_r, p_g)} \mathbb{E}_{(x,y)\sim\mathcal{D}} \left[d_\Omega(x, y) \right] \tag{3.4}$$

where $\Pi(p_r, p_g)$ is the set of all joint probabilities that has p_r and p_g as marginal distributions respectively. By using the Kantrovich-Rubinstein duality we can get an even simpler formulation

$$W(p_g, p_r) = \sup_{\|f\|_L \leq 1} \left(\mathbb{E}_{x\sim p_g}[f(x)] - \mathbb{E}_{x\sim p_r}[f(x)] \right) \tag{3.5}$$

where $\|f\|_L \leq 1$ means that f must be 1-Lipschitz; essentially $\|f(x_1) - f(x_2)\| \leq \|x_1 - x_2\|$ for any x_1, x_2. This constraint is a challenge to enforce. It can be solved by using clipping on the weights.

In practice we need to approximate this formulation by maximizing and averaging. We also restrict our search to some space of parameterized functions f_θ like neural networks with some predetermined architecture. We sample m samples from each of the distributions $\{x_i\} \sim p_r$ and $\{\tilde{x}_i\} \sim p_g$

$$\tilde{W}(p_g, p_r) = \max_\theta \left(\frac{1}{m} \sum_{i=1}^{m} f_\theta(x_i) - \frac{1}{m} \sum_{i=1}^{m} f_\theta(\tilde{x}_i) \right) \tag{3.6}$$

and f_θ needing to be Lipschitz constrained. As in the case of the JS-loss GAN (2.6) we can use the discriminator of a GAN to approximate this distance between the distribution of the dataset $\mathcal{D}_{\text{data}}$ and the distribution of the generator $\mathcal{D}_g$.

This metric induces a weaker topology on the space of probability distributions than for example the Kullback-Leibler divergence or Jensen-Shannon divergence. This ensures that given a locally Lipschitz continuous mapping f_θ (for example a NN) $W(p_r, p_\theta)$

is continuous everywhere and differentiable almost everywhere. This gives rise to the Wasserstein GAN (WGAN). It tackles the problem that training GANs can be difficult because of failure to converge and local minima.

In the original GAN the Jensen-Shannon (JS) divergence (6) was implicitly used as a loss. This metric turns out to produce troublesome gradients in many cases. The Wasserstein metric on the other hand has a gradient defined almost everywhere.

3.2.2 Suitability as a loss function for GANs

This is the basis for the Wassersten GAN (WGAN) by Arjovsky et al. (2017) the setup of which can be seen in figure 3.1. The main point about Wasserstein GAN is that it will reduce the risk of mode collapse.

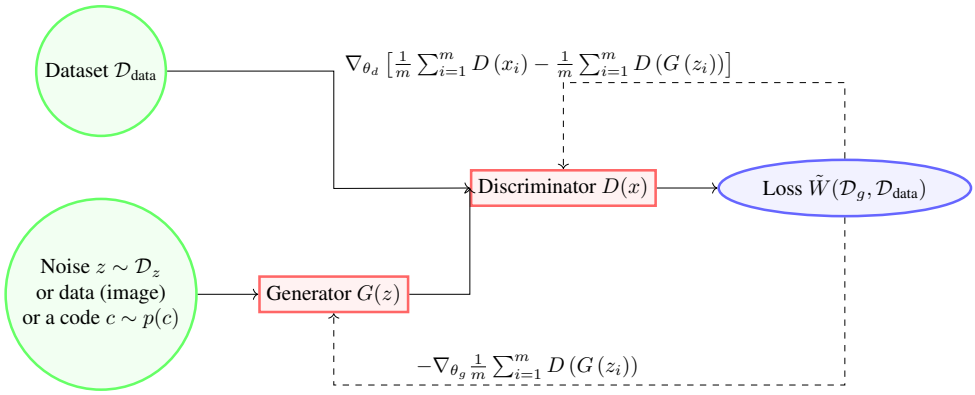


Figure 3.1: A basic setup for a Wasserstein GAN including the alternative gradients.

Enforcement of the Lipschitz constraint is not straight forward. Initially it was enforced by weight clipping. Weight clipping is a simple technique to keep the weights of the network from growing to much. It works by choosing a hyperparameter c that will set an absolute upper bound for each individual weight

$$\textbf{clip}(w; c) = \text{sign}(w) \min(c, |w|). \tag{3.7}$$

Later works by Gulrajani et al. (2017) and Adler and Lunz (2018) introduce a better alternative called Gradient Penalty (WGAN-GP). This works by constraining the gradient of the discriminator with respect to the generated input to be close to 1. Then this can be converted into a regularizer like so

$$\min_G \max_D \left\{\mathbb{E}_{x\sim\mathcal{D}_{\text{data}}}[D(x)] - \mathbb{E}_{z\sim\mathcal{D}_z}[D(G(z))] + \lambda(\|\nabla_\xi D(\xi)\|_2 - 1)^2\right\} \tag{3.8}$$

where $\xi = \varepsilon x + (1-\varepsilon)\tilde{x}$ is a combination of samples from the generator $\tilde{x}$ and the dataset x for some hyperparameter $\varepsilon \in [0, 1]$. This is a restriction on how the discriminator can interpret samples given by the generator. For example we want the discriminator to use

visual features in an image to detect whether it is generated or not. We do not want it to focus on things like invisible noise or small perturbations.

Actually it appears that the form of the loss might not be so important as enforcing the Lipschitz constraint. In a paper by Qin et al. (2018) they argue that all loss functions perform rather similarly when joined with Lipschitz regularization.

3.2.3 Lipschitz constraint in Banach spaces

It is possible to generalize this penalty to use an arbitrary norm. That is what Adler and Lunz (2018) did which they called Banach Wasserstein GAN. They extend the formulation so that one can choose the norm best suited for the application. A Banach space is a complete normed space - a space where Cauchy sequences converge. Say we want the generated samples from a domain Ω to be extra penalized on the edges of the image. We consider the space of possible discriminators $D : \mathcal{X} \to \mathbb{R}$.

One benefit with this generalization is the ability to enforce that functions are from Sobolev spaces. These are spaces where solutions to partial differential equations are found. For example the space $W_{k,p}(\Omega)$ is of weak differentiable and Lebesgue integrable functions of order k and p respectively. They have the general norm for a vector x

$$\|x\|_{W_{k,p}} = \left(\int_\Omega \left(\mathcal{F}^{-1} \left[(1 + |\xi|^2)^{k/2} \mathcal{F}x \right] (t) \right)^p dt \right)^{\frac{1}{p}} \tag{3.9}$$

where $\mathcal{F}$ is the Fourier transform. It extracts the frequencies from any Lebesgue integrable function $\mathcal{F}f(x) = \int_{\mathbb{R}^n} f(x) e^{2\pi \langle x, \xi \rangle i} dx = \hat{f}(\xi)$. In the Fourier space multiplying with the identity function ξ is the same as calculating a weak gradient in the original function space. For example the space $W_{1,2}$ will have the norm

$$\|x\|_{W_{1,2}} = \left(\int_\Omega \left(f(t)^2 + |\nabla f(t)|^2 \right)^p dt \right)^{\frac{1}{2}}. \tag{3.10}$$

In this space there is put more emphasis on the edges. Since when $\|x_1 - x_2\|_{W_{1,2}}$ is small the absolute value of their derivative is also close which means that they both have the same kind of change in the function at that point which means that edges are preserved.

3.3 Gradient layer

There is also a paper by Nitanda and Suzuki (2018) where they use functional gradients to improve the training of WGANs. Essentially they note that the generator may have too little representative power and expands the problem to infinite dimensional spaces. This is realized as a gradient layer that through functional gradient descent minimizes the loss in the forward propagation of the network.

Let $G = G_2 \circ G_1$ be a generator NN and D be a discriminator NN. We split the generator at an arbitrary layer into a left G_1 and right G_2 part consistent with how we usually draw a network. The parameters of each part is denoted θ_{g1} and θ_{g2} respectively.

For regularization we use R_d to denote a gradient penalty regularizer like in (3.8). We denote the optimal discriminator for a fixed generator with parameters

$$\theta_d^* = \arg\max_{\theta_d} \mathbb{E}_{z\sim\mathcal{D}_z}[-D(G(z;\theta_g);\theta_d)] - \lambda R_d \tag{3.11}$$

as $D(x;\theta_d^*)$. It is possible to define this for more general losses as well. Correspondingly we denote the loss for a varying generator as

$$\mathcal{L}_d(G) = \max_{\theta_d} \mathbb{E}_{z\sim\mathcal{D}_z}[-D(G(z;\theta_g);\theta_d)] - \lambda R_d. \tag{3.12}$$

Note that this is the generators objective in the Wasserstein GAN definition.

We want to create a layer, ϕ, in the generator that minimizes the loss the same way as a training update will

$$\mathcal{L}_d(G_2 \circ \phi \circ G_1) \leq \mathcal{L}_d(G). \tag{3.13}$$

We are considering an infinite dimensional optimization problem over the space $L^2(\mathcal{D}_g)$ of square integrable functions with the measure induced by $\mathcal{D}_g$. Also including the inner product $\langle\cdot,\cdot\rangle_{L^2(\mathcal{D}_g)}$ which is the expectation of the inner product of two functions. This is realized for functions $\phi_1, \phi_2 \in L^2(\mathcal{D}_g)$ as follows

$$\langle\phi_1,\phi_2\rangle_{L^2(\mathcal{D}_g)} = \mathbb{E}_{x\sim\mathcal{D}_g}\left[\phi_1(x)^T\phi_2(x)\right]. \tag{3.14}$$

Then we want to find an intermediate function ϕ that minimizes the loss; essentially

$$\phi = \arg\min_{\phi'} \mathcal{L}_d(G_2 \circ \phi' \circ G_1) \tag{3.15}$$

The procedure of finding ϕ defines the *gradient layer*.

Let $\mathcal{L}(\phi) = \mathcal{L}_d(G_2 \circ \phi \circ G_1)$ for ease of notation in this section. What we can do to optimize the training loss in a functional manner is to perform gradient descent in a compositional manner. One application of the layer will transform an input into the next step of gradient descent

$$\phi_\eta(x;\theta_d) = x + \eta\nabla_{\theta_d} D(G_2(x;\theta_{g2});\theta_d) \tag{3.16}$$

where η is the hyperparameter for learning rate. We can then apply several of these functions in sequence to reach the optimum of the loss function. Note that it is dependent on the parameters of the discriminator and the generator. In this context ∇_{θ_d} represents a functional derivative which in this case is the normal derivative. This will improve the loss further and hence the quality of the samples.

Ideally we could compute the gradient using a functional derivative of the loss function. This is achieved by perturbing the input function ϕ with some arbitrary function v times a small constant t. The change caused by that small perturbation is our derivative. This can be stated using the envelope theorem and Lebesgues convergence theorem as follows

$$\frac{d}{dt}\mathcal{L}(\phi + tv)\Big|_{t=0} = -\mathbb{E}_{\tilde{x}\sim\mathcal{D}_g}\left[\nabla_x D(x;\theta_d^*)\Big|_{x=\phi(\tilde{x})}^T v(\tilde{x})\right]. \tag{3.17}$$

The envelope theorem introduced by Milgrom and Segal (2002) connects the parameterization of an objective function with the optimal point of the objective function.

3.3.1 Algorithms

The paper presents several algorithms for computing parts of the gradient layer. They are however memory intensive since they require to save all the parameters of the discriminator in each step. We denote as before by $\mathcal{A}(\theta_d, \phi)$ one step of a gradient based optimization method like SGD, Adam or RMSprop.

Data: Batch size m, the number of iterations T, the initial parameters $\theta_d^{(0)}$ of the discriminator, the number of iterations T_0 for the discriminator, the regularization parameter λ, and the learning rate η.
for $k = 0$ **to** $T - 1$ **do**
  $\theta_d \leftarrow \theta_d^{(k)}$
  for $k_0 = 0$ **to** $T_0 - 1$ **do**
    Take m random samples from the dataset $\{x_i\}_{i=1}^m$,
    from the generator $\{\tilde{x}_i\}_{i=1}^m \sim \mathcal{D}_g$,
    and from the uniform distribution $\{\varepsilon_i\} \sim U[0,1]^m$.
    $\{\tilde{x}_i\}_{i=1}^m \leftarrow \{\phi_\eta(\cdots \phi_\eta(\phi_\eta(\tilde{x}_i; \theta_d^{(k)}); \theta_d^{(k-1)}) \cdots ; \theta_d^{(1)})\}$
    $\{\xi_i\}_{i=1}^m \leftarrow \{\varepsilon_i x_i + (1-\varepsilon_i)\tilde{x}_i\}_{i=1}^m$
    $v \leftarrow \nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m [D(\tilde{x}_i; \theta_d) - D(x_i; \theta_d) + \lambda R_d(\xi_i)]$
    $\theta_d \leftarrow \mathcal{A}(\theta_d, v)$
  end
  $\theta_d^{(k+1)} \leftarrow \theta_d$
end
Result: $\theta_d^{(1)}, \ldots, \theta_d^{(T)}$

Algorithm 2: Fine tuning the generator and discriminator. We restrict ourselves to the Wasserstein loss in this context although it is possible to do it for more general loss functions.

With algorithm 2 we can fine tune the result from the generator by repeatedly applying the gradient layer. This will solve some of the problem caused by the generator not having infinitely many parameters. The result from this algorithm is a list of T updated discriminators.

What we are basically doing in the algorithm is improving the output of the generator with an ensamble of functional discriminator gradients. So why do we not just update the parameters like in a normal setting. The point is that these functional gradients should much better represents the underlying continuous data structure. A reference for functional calculus and functional gradients is given by Luenberger (2010).

This algorithm serves as an approximation to the more ideal functional gradient descent in algorithm 3. Here we are constructing new functions in each step of the algorithm by using composition. Essentially instead of updating the weights we preserve the weights from each iteration making them a part of the new function. This is memory intensive so

in practice it could be useful to only preserve a small portion of the weights.

Data: The initial generator G and the learning rate η.
$\phi_0 \leftarrow G$
for $k = 0$ **to** $T - 1$ **do**
  $\phi_{k+1} \leftarrow \phi_k - \eta \nabla_\phi \mathcal{L}(\phi_k)$
end
Result: ϕ_T

Algorithm 3: Functional gradient descent for finding the optimal generator.

The method shows promising results in experiments. It managed to learn toy datasets of Gaussian noise with only 100 iterations. In contrast JS-GAN and WGAN fail to learn these kinds of simple problems. In the paper they also train a WGAN-GP on the Cifar-10 and STL-10 datasets and then apply algorithm 2. It manages to improve the inception score with several points after only 20 to 30 iterations.

3.4 Conditional GAN

For many cases when you want to use GAN to not only generate new random samples from a distribution induced by a dataset, but learn a mapping from one kind of samples to another. An example of this is to learn a mapping from sketches to images and vice versa. This is the objective in the papers Mirza and Osindero (2014) and Isola et al. (2017) where they establish the conditional GAN (cGAN).

Given a labeled dataset $\{(x, y)\}$ of for example two kinds of images we want to find the relationship between them. We can also consider this as a pair of data that we want to find the relationship between. If we want to learn how to transform one kind into another i.e. learn a function $G(x) \approx y$ we can think of this as learning a conditional distribution $p(y|x)$. The intuitive interpretation of this is that we want to generate samples which are related to the data we are conditioning on.

This a sort of supervised method for GAN. We gain control over the output by restricting some of the input to be data we want. What we are essentially doing is that we replace some of the noise input with data x.

There are several ways to construct a GAN that can relate pairs of data. This has a close connection to supervised and semi supervised learning. In both cases you want to learn a function that connects two different spaces $f : \mathcal{X} \to \mathcal{Y}$. Usually those two spaces are of different size. For example in image classification you connect several instances of images to a specific class label.

3.4.1 Image to image translation

In the paper Isola et al. (2017) they present an algorithm called pix2pix which can transform one image into a different kind of image. For example turning a satellite image into a map or vice versa. It is an extension of conditional GAN described earlier.

In earlier work image to image translation has been subject to application specific construction. Pix2pix on the other hand is intended to be a simpler and more general framework for this. There is an improvement to this as well in section 3.5 called CycleGAN.

One important concept in this implementation is the convolutional PatchGAN discriminator. In this formulation the discriminator scans over an input image in small patches just like a convolutional layer. That way the discriminator is more locally dependent than if it were to consider the entire picture at once. It only cares about structures within the patches. That means that pixels that are further apart than the size of the patch are assumed independent. This means we are effectively modeling the image as a Markov random field.

We are modeling the texture of the input by only considering small neighbourhoods of pixels. That way transformations in one area of the output should be consistent with transformations in another area of the output. This is a property we want when converting between for example drawings and images like maps and satellite images. In a map there is roughly always the same rule for converting a satellite image into representations independent of the overall position in the map. A road is always a road and a pond is always a pond.

The generator is based on a convolutional neural network architecture called UNet from Ronneberger et al. (2015). It was initially designed for image segmentation for biomedical images. This was for example detecting cells in microscope images and then returning the area where the cells are located. The architecture of UNet consists of connecting blocks of convolutional layers with dropout and batch normalization. The original architecture can be seen in figure 3.2. The name comes from the U-shape it makes when presented as in the figure. UNet has become the state-of-the-art in many applications of deep learning.

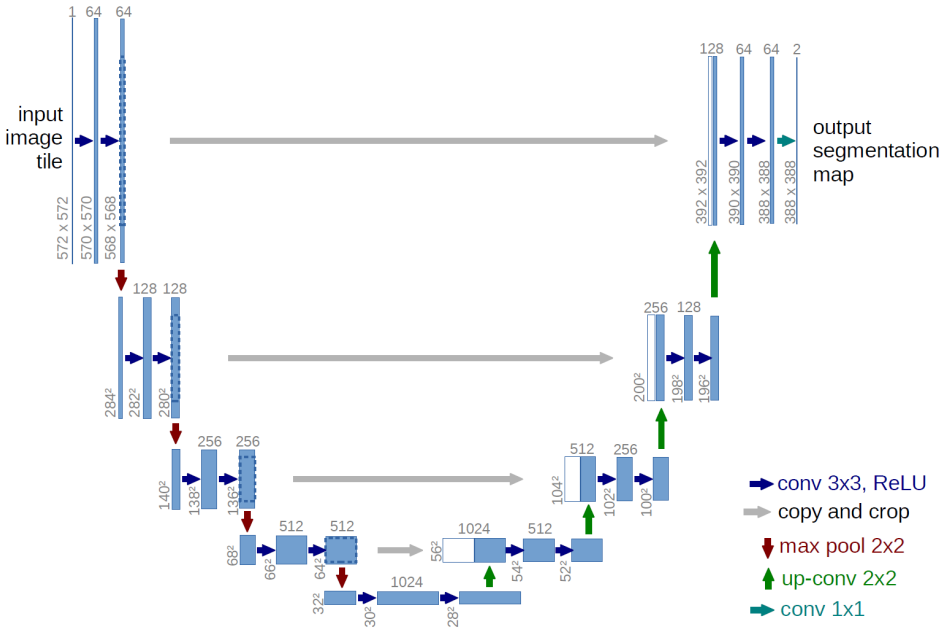


Figure 3.2: The UNet architecture from the original UNet paper on biomedical image segmentation.

A different view of the UNet architecture can be seen in figure 3.3. An important

feature of this architecture is the skip connections. That is output from a layer which is preserved and concatenated or added with the output from a later layer in addition to being passed to the next layer. This opens for the possibility that important information can travel between layers.

It follows an encoder decoder structure; much like an autoencoder. The input is encoded into a low dimensional representation and then rebuilt into a full size image again.

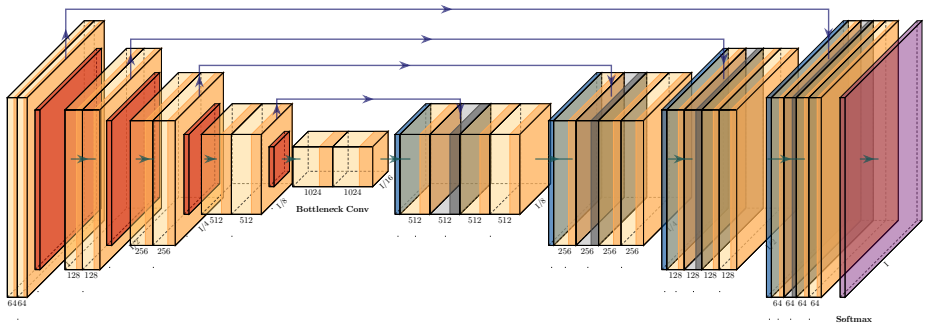


Figure 3.3: A 3D view of an UNet architecture. The numbers beneath each layer represents the number of filters in the convolution.

It is not enough to just use the regular discriminator loss for the generator in this setting. We also need to penalize the generator to produce images that are similar to the ground truth input. That is why we are using an additional L1 loss

$$\mathcal{L}_{L1}(G) = \mathbb{E}_{(x,y)\sim\mathcal{D}_{\text{data}},z\sim\mathcal{D}_z}[\|y - G(x,z)\|_1]. \tag{3.18}$$

The L1 regularization enforces sharp edges while L2 regularization would result in blurry outputs. Combining the conditional image input x and the noise z is not straight forward. In practice we could omit the noise and instead provide noise by using dropout layers. This was sufficient for this method and reduced some of the implementation complexity.

The complete formulation for a general loss is

$$\min_G \max_D \mathcal{L}(G,D) + \lambda\mathcal{L}_{L1}(G) \tag{3.19}$$

where λ is a hyperparameter. The architecture of this method is probably more determining of the result than the loss.

3.5 Cyclic GAN

We have seen that we can use GAN to learn to map images from one domain to another with conditional GAN. This can of course be done for the opposite direction as well. These are essentially inverse problems; mapping satellite images to maps and mapping maps to satellite images. This leads to the question of whether there is something that can be earned by jointly training these mappings. We can transform an image to a target domain and then

back to the original domain. Essentially we would want to minimize the features lost in the transformation. That is what Zhu et al. (2017) did when they created the CycleGAN. This can be seen as a natural extension of the conditional GAN. We are connecting a cGAN with its inverse formulation and thereby strengthening the relationship between input and output.

Lets say you have two domains, X and Y, which can be considered random variables. These can for example be satellite images and maps. In some sense it is the same information represented in a different format or style. Now we want to train two generators on mapping between the domains $G_{XY} : X \to Y$ and $G_{YX} : Y \to X$. We can do this by training two discriminators for each domain D_X and D_Y and using them to calculate a loss. Then in addition we can apply the generators in sequence and compare the result to the original input for each domain $G_{XY}(G_{YX}(y)) \approx y$ and $G_{YX}(G_{XY}(x)) \approx x$. We can train the generators to minimize the loss from each discriminator as well as minimizing the lost information when applying the generators in sequence. This can all be combined to a new joint loss function for both generators and discriminators

$$\begin{aligned}\mathcal{L}(G_{XY}, G_{YX}, D_X, D_Y) = \mathcal{L}_{\text{GAN}}(G_{XY}, D_Y) \\ + \mathcal{L}_{\text{GAN}}(G_{YX}, D_X) \\ + \lambda(\mathcal{L}_{\text{cycle}}(G_{XY}, G_{YX}) \\ + \mathcal{L}_{\text{cycle}}(G_{YX}, G_{XY})).\end{aligned} \tag{3.20}$$

Here $\mathcal{L}_{\text{GAN}}$ is some of the normal GAN objectives discussed earlier. It can be any of the many choices of GAN loss functions. The cyclic loss $\mathcal{L}_{\text{cycle}}(G_{XY}, G_{YX})$ tries to measure how much information is lost in a cycle of generators. In the original paper they simply compare the original data to the data returned from the cycle like so

$$\mathcal{L}_{\text{cycle}}(G_{XY}, G_{YX}) = \mathbb{E}_Y[\|G_{XY}(G_{YX}(Y)) - Y\|_1]. \tag{3.21}$$

The generator is however not supposed to be a deterministic mapping from one input to an output, but rather represents all the possible mappings when conditioned on a particular input and then sample from that distribution.

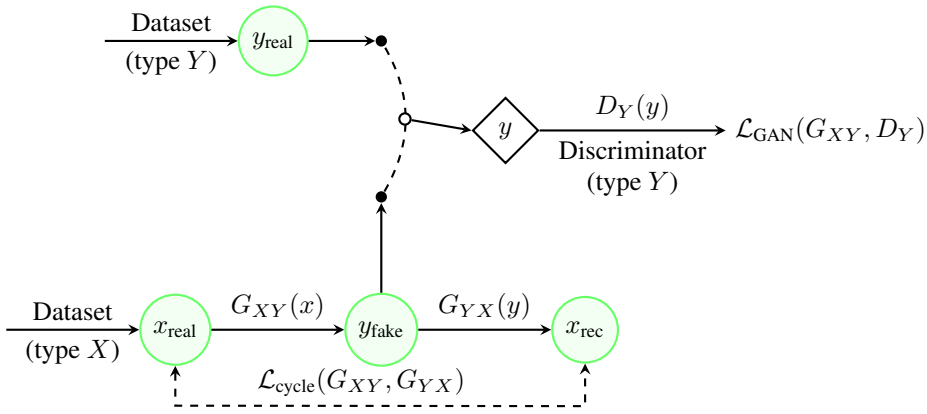


Figure 3.4: Visual representation of the $X \to Y \to X$ cycle of the Cycle GAN. The $Y \to X \to Y$ cycle is analogous to this just with X and Y flipped.

The combined loss from (3.20) consists of two normal GAN losses and two cycle losses. Each of these are providing updates to the generators and discriminators involved in the definition. In the original paper they experimented with using only some of the losses. This showed that using all the losses gave the best performance. The resulting formulation is

$$\min_{G_{XY}, G_{YX}} \max_{D_X, D_Y} \mathcal{L}(G_{XY}, G_{YX}, D_X, D_Y). \tag{3.22}$$

This method shows improved performance over pix2pix. Since it does not require paired data from each domain it is also more applicable.

3.6 InfoGAN

In the paper Chen et al. (2016) they develop an interpretable representation learning algorithm by combining information theory and GAN. We have included a reference to information theory in appendix B. For example the entropy $H(X)$ and mutual information $I(X;Y)$ are essential to the formulation of InfoGAN.

This construction makes the GAN learn a code for features by regularizing with the mutual information of the code and the generated sample. This requires to sample from the posterior distribution of the code which is unknown. We can instead approximate this distribution by updating it sequentially.

Let $c \sim p(c)$ be a random variable representing the distribution of the code we want to learn and $z \sim \mathcal{D}_z$ be the usual random variable for noise. The modified objective function (2.7) of the GAN, including the mutual information regularization term is of the form

$$\mathcal{L}_I(G, D) = \mathcal{L}(G, D) - \lambda I(c; G(z, c)). \tag{3.23}$$

However calculating the mutual information explicitly is not possible since we don't know the posterior distribution $P(c|x)$. What we can do instead is approximate it with

another distribution $Q(c|x)$ as follows

$$\begin{aligned} I(c; g(z,c)) &= H(c) - H(c|G(z,c)) \\ &= \mathbb{E}_x[\mathbb{E}_{c'}[\log P(c'|x)]] + H(c) \\ &= \mathbb{E}_x[D_{KL}(P\|Q) + \mathbb{E}_{c'}[\log Q(c'|x)]] + H(c) \qquad (3.24) \\ &\geq \mathbb{E}_x[\mathbb{E}_{c'}[\log Q(c'|x)]] + H(c) = L_I(Q,G) \qquad (3.25) \end{aligned}$$

The last inequality is because the Kullback-Leibler divergence is always positive. In (3.24) the inequality will approach equality as $\mathbb{E}_x[D_{KL}(P\|Q)] \to 0$ the approximated posterior $Q(\cdot|x)$ will approach the true posterior. This means that the approximated mutual information will equal the maximal mutual information $L_I(G,Q) = H(c) = I(c; G(z,c))$. This gives us a new objective function

$$\mathcal{L}_I(G,D,Q) = \mathcal{L}(G,D) - \lambda L_I(Q,G). \qquad (3.26)$$

Which leads to the following extended GAN formulation

$$\min_{G,Q} \max_{D} \mathcal{L}_I(G,D,Q). \qquad (3.27)$$

This means that when updating the generator in training we also need to update the auxiliary distribution Q.

This technique is known as Variational Information Maximization first introduced by Barber and Agakov (2003). This is in many ways analogous to the Expectation Maximization (EM) algorithm. We can approximate $L_I(Q,G)$ with Monte Carlo simulation and choosing an appropriate family of distributions for $Q(c|x)$. Since it is a highly complex distribution it is natural to parameterize it as a neural network. In the original paper they simply connect a new fully connected layer to the end of the convolutional layers of the discriminator.

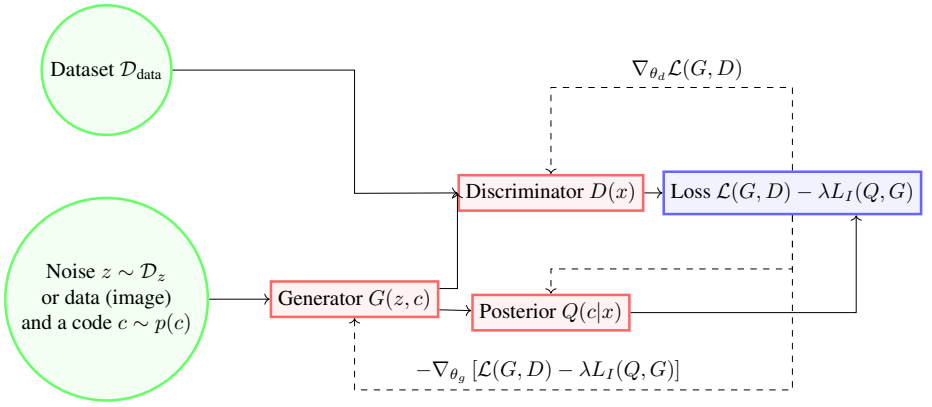


Figure 3.5: The basic setup of InfoGAN.

The result of this is that, when converged, the code c will have taken a natural role for representing the different varieties of samples produced by the generator. For example it

was trained on the MNIST dataset with c as a vector of one categorical component and two continuous components. A categorical distribution $c \sim \text{Cat}(K) = \{k_1, \ldots, k_K\}$ is a discrete uniform distribution i.e. $P(c = k_i) = \frac{1}{K}$ for all i. The categorical code $c_1 \sim \text{Cat}(K = 10)$ learned to represent the different digits. This is quite nice because the GAN was not supervised and had no knowledge of the labels. It learned to distinguish between the different digit classes as a consequence of maximizing the information content in the latent code. This means that the generator has learned to recognize the distinction between the digits and we can now explicitly tell it to generate a specific digit.

For a normal GAN there is no way of knowing what the properties of a sample will be given the input noise. However it is very desirable to be able to control what the output of the generator will be.

Another benefit of this extension is that it actually improves convergence in training based on empirical results. It can also be applied to most GAN architectures. However it is not so easy to apply it to a conditional GAN. In that context we already have a controlled input into the generator. For the code to become meaningful there needs to be a natural property that it can describe. It would be interesting to see what will happen if we try to model more complex relationships between components of the code and the generated samples.

3.7 Bayesian GAN

We are going to present and discuss the method in the paper by Saatci and Wilson (2017) where they introduce Bayesian inference to GANs. This not only gives a better explainability of the performance of GAN, but also improves it to a state-of-the-art level. In this section we are sometimes going to refer to the parameters of the generator and discriminator as a particular realization of the respective network. We also refer to the probability densities p for some distribution as simply the probability for simplicity.

We extend the definition of GAN to include a distribution of parameters, θ_g and θ_d, which is a distribution of all possible GANs with some predetermined architecture. This new model is called Bayesian GAN (BGAN). It provides a natural uncertainty estimation of the output which gives us more ability to asses the quality of the GAN. Further it relaxes some of the issues with GAN training, like mode collapse, since we are always considering an ensemble of generators and discriminators.

In a recent paper by He et al. (2018) they present an improvement on the BGAN and provide theoretical guarantees for convergence. The method is called Probabilistic GAN (ProbGAN) and can more robustly incorporate the loss functions in table 3.1. Both of these methods provide a probabilistic formulation for GANs.

To introduce distributions over the parameters we need some way of approximating the posteriors over the discriminator and the generator. Given a particular discriminator θ_d we want to know what the distribution of good generators θ_g is. This can be considered a posterior of generators over discriminators $p(\theta_g|\theta_d)$. That way we can consider a GAN in the context of Bayes rule

$$p(\theta_g|\theta_d) = \frac{p(\theta_d|\theta_g)p(\theta_g)}{p(\theta_d)}. \tag{3.28}$$

The data and hyperparameters are implicitly embedded in this formulation since it is part of defining the relationship between the discriminator and the generator. Note that we have not yet introduced the dataset as a component. The dataset partly defines the distribution of discriminators and then by extension the generators.

We can evaluate the network by sampling from the distribution of parameters and then evaluating it as normal function. Essentially the parameters of the generator are not a fixed point, but rather many possible generators where the probability mass is located around the configuration which at a particular point in training seems to be the best generators.

In the normal GAN setting we are always implicitly conditioning the training updates on the particular realization of the noise. Different samples of noise produce different generated samples which result in different training updates in result. Normally we produce so many samples at the same time that it should even out, but there is perhaps a performance gain from marginalizing over the noise.

In reality we need to restrict ourselves to a mini-batch of samples from the dataset and the generator. We take m_g samples from the generator and m_d samples from the dataset. However in most scenarios we would take the same amount of samples from the dataset and the generator $m_g = m_d = m$, but we will keep the indexes for generality. There are also two different approaches to defining the posteriors in the paper; an unsupervised formulation and a semi supervised formulation.

3.7.1 Unsupervised setting

We have a prior for the distribution of generators $p(\theta_g|\alpha_g)$ where α_g is a hyperparameter. The posterior for the generator can be estimated by considering the posterior conditional on the discriminator

$$p(\theta_g|z, \theta_d) \propto \left(\prod_{i=1}^{m_g} D(G(z_i; \theta_g); \theta_d) \right) p(\theta_g|\alpha_g). \tag{3.29}$$

We draw noise samples $z = \{z_i\}_{i=0}^{m_g}$ for the generator and use them to compute samples $G(z_i; \theta_g)$.

The discriminator also needs to consider the distribution of data x

$$p(\theta_d|z, x, \theta_g) \propto \prod_{i=1}^{m_d} D(x_i; \theta_d) \times \prod_{i=1}^{m_g} (1 - D(G(z_i; \theta_g); \theta_d)) \times p(\theta_d|\alpha_d). \tag{3.30}$$

Here α_d is a hyperparameter for the prior distribution of discriminators $p(\theta_d|\alpha_d)$. Note that this needs to be scaled appropriately to the size of the batch of data we are computing over.

We do not want the posterior to be dependent on the noise. To remove this we can marginalize over the noise using Monte Carlo

$$p(\theta_g|\theta_d) = \int p(\theta_g, z|\theta_d) dz = \int p(\theta_g|z, \theta_d) dz \approx \frac{1}{J_g} \sum_{j=1}^{J_g} p(\theta_g|z_j, \theta_d) \tag{3.31}$$

where we sample J_g noise samples $z_j \sim \mathcal{D}_z$. This gives us an estimate of the generators posterior and we can do the same procedure for the discriminator $p(\theta_d|\theta_g) \approx$

$\frac{1}{J_d}\sum_{j=1}^{J_d} p(\theta_d|z_j, x, \theta_g)$. Note that the index of the noise in this context represents an entire batch of noise suitable to be put into (3.29) or (3.30).

3.7.2 Semi supervised setting

Then on the other hand we can formulate the posteriors in a semi supervised setting for K-class classification. Here we have n unlabeled samples $\{x_i\}_{i=1}^n$ and a typically much smaller set $x^{(s)} \times y^{(s)} = \{(x_i^{(s)}, y_i^{(s)})\}_{i=1}^{n_s}$ of n_s labeled samples. In this context we extend the definition of the discriminator to include classification of samples. The discriminator now gives out a $K+1$ sized vector of the probability of belonging to one of the K classes or being generated by the generator. We denote the probability that a sample x_i is from a class y given by the discriminator as $D(x_i \to y; \theta_d)$. The class $y = 0$ means that the sample is generated. For the generators posterior we want the samples to be classified as anyone of the K classes. We therefore need to sum over all the nonzero classes which gives

$$p(\theta_g|z, \theta_d) \propto \left(\prod_{i=1}^{m_g}\sum_{y=1}^{K} D(G(z_i; \theta_g) \to y; \theta_d)\right) p(\theta_g|\alpha_g). \tag{3.32}$$

The generator wants the discriminator to classify its samples as any of the available. Note that it is enough that the discriminator is confused about which class generated samples are from, as long they are not classified as generated. This is a behavior we do not want, but the discriminator should be able to correct this by being better at classification.

We could also potentially extend this to a formulation similar to cGAN. By explicitly telling the generator which class to produce and the discriminator which class to detect we can have a Bayesian cGAN as well.

The discriminator needs to learn to correctly classify the labeled data hence the last line of the following formulation. It also needs to detect the generated samples and determine som nonzero label for the unlabeled data samples. Given the distribution of data x and the labeled samples $x^{(s)} \times y^{(s)}$ we can formulate the semi supervised posterior for the discriminator

$$\begin{aligned} p(\theta_d|z, x, x^{(s)}, y^{(s)}, \theta_g) \propto &\prod_{i=1}^{m_d}\sum_{y=1}^{K} D(x_i \to y; \theta_d) \\ &\times \prod_{i=1}^{m_g} D(G(z_i; \theta_g) \to 0; \theta_d) \\ &\times \prod_{i=1}^{n_s} D(x_i^{(s)} \to y_i^{(s)}; \theta_d) \times p(\theta_d|\alpha_d). \end{aligned} \tag{3.33}$$

We always include the entire labeled dataset in this formulation. This is because it is usually smaller than the unlabeled set and is essential for defining the relationship between the data and the labels.

These semi supervised posteriors can be marginalized in a similar manner to the unsupervised setting in (3.31).

3.7.3 Sampling from the posterior with SGHMC

Training a probabilistic version of GAN requires a different view of the training algorithm. Rather than just updating the weights with SGD and backpropagation we have many different realizations of the generator and discriminator. Each of these realizations are points in a distribution we are trying to learn. We can therefore say that the parameters of generator and discriminator are samples.

In each iteration we need to calculate a loss for all the parameter samples and then update the parameters. This update needs to be noisy so the parameter samples do not collapse into a single point. To sample from the posterior we use Stochastic Gradient Hamiltonian Monte Carlo (SGHMC) developed by Chen et al. (2014) which can be seen in algorithm 4. This is convenient since it resembles momentum based SGD. To represent the posteriors we collect and update a set of approximate samples from the posterior for the generator and the discriminator. We marginalize the noise as in (3.31). For every Monte Carlo simulation for marginalizing the noise we do S SGHMC updates to the parameters.

It is possible to approximate the distribution function of generator and discriminator parameters with Gaussian Mixture Approximation (GMA). This is necessary for certain kinds of sampling methods.

Experimental results show that BGAN outperforms both a supervised convolutional neural network and convolutional variants of JS GAN and WGAN on a semi supervised learning task in datasets MNIST, Cifar-10, SVHN, and CelebA.

Data: A friction term α, learning rate η, number of Monte Carlo (MC) samples from the discriminator J_d and the generator J_g, and S SGHMC samples.
We represent the posteriors as samples from the previous iteration $\{\theta_g^{j,s}\}_{j=1,s=1}^{J_g,S}$ and $\{\theta_d^{j,s}\}_{j=1,s=1}^{J_d,S}$.
First update the generators posterior samples.
for $j = 1$ **to** J_g **do**
  Sample noise $\{z_i\}_{i=1}^{J_g} \sim \mathcal{D}_z$ for J_g MC samples times m_g mini-batch size.
  Update the posterior samples of the generator $p(\theta_g|\theta_d)$ with SGHMC.
  for $s = 1$ **to** S **do**
    $\varepsilon \sim \mathcal{N}(0, 2\alpha\eta I)$.
    $v \leftarrow (1-\alpha)v + \eta \left(\sum_{i=0}^{J_g} \sum_{k=0}^{J_d} \nabla_{\theta_g} \log p(\theta_g|z_i, \theta_d^{k,s}) \right) + \varepsilon$.
    $\theta_g^{j,s} \leftarrow \theta_g^{j,s} + v$.
    Update the posterior samples with $\theta_g^{j,s}$.
  end
end
Then update the discriminators posterior samples.
for $j = 1$ **to** J_d **do**
  Sample noise $\{z_i\}_{i=1}^{J_d} \sim \mathcal{D}_z$ for J_d MC samples times m_d mini-batch size.
  Sample mini-batch of m_d data samples denoted as x.
  Update the posterior samples of the generator $p(\theta_d|\theta_g)$ with SGHMC.
  for $s = 1$ **to** S **do**
    $\varepsilon \sim \mathcal{N}(0, 2\alpha\eta I)$.
    $v \leftarrow (1-\alpha)v + \eta \left(\sum_{i=0}^{J_d} \sum_{k=0}^{J_g} \nabla_{\theta_d} \log p(\theta_d|z_i, x, \theta_g^{k,s}) \right) + \varepsilon$.
    $\theta_d^{j,s} \leftarrow \theta_d^{j,s} + v$.
    Update the posterior samples with $\theta_d^{j,s}$.
  end
end
Result: Posterior samples $\{\theta_g^{j,s}\}_{j=1,s=1}^{J_g,S}$ and $\{\theta_d^{j,s}\}_{j=1,s=1}^{J_d,S}$.

Algorithm 4: One iteration for sampling the Bayesian GAN. When we update the weights we are in reality performing backpropagation. In the semi supervised setting we also need to include the labeled dataset for estimating the posterior.

Chapter 4

Experiments

In this section we are going to present our attempts and results of implementing different GAN varieties and a framework for doing experiemtents.

We implemented the pix2pix algorithm as described in section 3.4.1 and built a framework for doing experiments. More specifically we reproduced the original problem of converting satellite images to maps. This was also intended to investigate applications of this technique.

We also implemented a Wasserstein GAN and tried to train it on the MNIST and Cifar10 datasets. Interestingly to implement the Wasserstein metric we could formulate it as a supervised loss where the labels were either 1 or -1 if samples came from either the dataset or the generator respectively. The discriminator was configured to output both positive or negative numbers. We drew m, the batch size, samples from the dataset and labeled them as 1 for training the discriminator. Similarly we drew m samples from generator and labeled them as -1. Then to compute the loss we simply multiply the output from the discriminator with the label and take the mean. That way we obtain the formulation as in (3.6).

This was however not successful as the model, with some initial hyperparameters, did not converge after over 100 000 epochs. We then decided that it was not worth the time and effort to correct the hyper parameters and finish training the model. The Wasserstein GAN implementation is therefore not essential to this thesis.

4.1 Experimental framework

We developed a framework for easy implementation, organization and execution of experiments with deep learning. It revolves around defining an object that contains all the relevant information for a particular model and then connecting it to other relevant objects and tasks. We will refer to this as the model framework, or just model, as they are intended to be interchangeable. First we are going to discuss what we think are some of the properties of an ideal deep learning framework.

Doing experiments in deep learning has some challenging technical demands. We need a flexible system that we can easily adapt to new scenarios and connect with previously implemented methods. For example there should be a standard framework for downloading, processing, and loading datasets. This minimizes the coding effort when using different datasets.

In general there is a trade-off between framework flexibility and ease of use. For example if you have a minimal framework you need to implement everything for each case. More ideally you would only need to implement the thing that is different in that case. This trade-off is at its optimum when the framework makes only the necessary assumptions about its usage. There are different philosophies for how to resolve this issue. For example with functional programming there is ideally no internal state and functions are therefore always predictable. However there are sometimes performance gains when allowing for deletion and states. This can often be circumvented with clever functional programming.

All normal functionality that you would expect should be easily available. This includes common model architectures and training algorithms. However the scope of an experimental framework extends beyond implementing the model. It also requires handling data and deploying variations of experiments. This is something you would usually implement yourself as you would want it tailored for your specific use.

Just as important is the performance when the system is implemented. Deep learning deals with huge datasets and often billions of parameters. This leads to huge memory and computational demands. The community has therefore chosen to create frameworks that first describe computation in abstract terms and then execute the computation in an optimized environment. This optimized environment may be a computational server with GPUs and large storage capacity.

Every experiment is started from the same entry point given some input configuration files or command line arguments. This input describes which model to be used, which action to be performed, various configuration options and the hyper parameters of the model. This is very convenient since varying the parameters, running several instances in parallel, and keeping track of the experimental history requires little effort.

For example in order to train a model we would use the *train_model* action. Then it would load the model framework specified by the *model* parameter. The framework specifies which dataset to be loaded and builds a model in some underlying framework like *Keras* or *Chainer*. This is only from instantiating the model framework.

The next step is then to perform the specified action on the model. Since all models are defined in the same general abstract framework, we should be able to apply general functions to them. This enables us to use many different frameworks for defining computation, model architecture, and training algorithms.

The results are also organized in a structured manner. All results are collected in a results folder. At the top level it is organised by the name of the model that is being experimented on. Then it is possible to name specific experimental configurations so that all the results with that name are put in a named folder. Saved models, plots, images, and other related data are stored together in that folder or sub folders.

Datasets are kept in a separate data directory. The framework is configured to automatically download, extract, and load specified datasets from that folder. This makes it very

easy to transfer code to a server or to a new computer. No manual action is needed when running the code. This represents a core philosophy of this framework. When the code is configured correctly there should be no manual action necessary to get it running. It means that when starting from scratch on a new computer, and issuing the same command, the result will be the same as on a computer where experiments has already been run.

All the code and configuration files is available at:
`https://github.com/Anderssorby/odin`.

4.2 Map generation from satellite images

Here we train a conditional GAN on a dataset of satellite images and maps. We wanted to do this to investigate if this was possible to apply this for creating maps of places without maps. For example organizations like *OpenStreetMap* are using a big network of human volunteers to map unmapped urban areas. It is often in the context of supporting organizations providing humanitarian aid. We made some effort to contact the *Humanitarian OpenStreetMap Team* (HOT) and Missing Maps to see if there was any potential for collaboration and application of this technique. This was however not so useful for their particular application. A more suiting technique would be some kind of object detection or object segmentation.

We note that the results from this experiment can be scaled to images of higher resolution by progressive growing of GANs. This means starting by producing low resolution versions of the desired images of the dataset. When a sufficient performance level is reached, we can add a new layer of higher dimension to the generator and train on higher resolution images. We do this procedure for many levels of resolution. This can in the end result in stunningly realistic images.

4.2.1 Dataset

We obtained one of the original datasets collected for the paper by Isola et al. (2017). It consists of square image tiles from Google Maps. Each tile is in both satellite and map format. The dataset consisted of only 1095 training sample pairs and 1097 validation sample pairs. Each image is then processed to have the desired 256 by 256 pixels resolution. This was to speed up training time and reduce memory usage when experimenting.

The maps are for the most part from urban areas with straight streets and sometimes parks or water. A sample from the training set of both the map representation and satellite representation of the same square can be seen in figure 4.1.

For efficiency all of the training and validation samples were processed into tensor format and collected in a HDF5 file. This reduced the loading time when starting a new process.

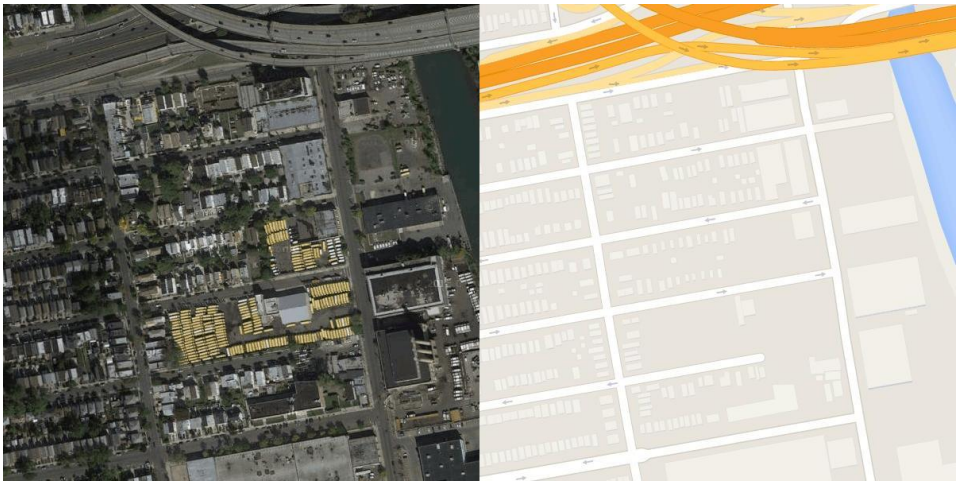


Figure 4.1: A sample from the training data.

4.2.2 Implementation

Initially we based our model on an unofficial open source implementation found on Github. This was in part to speed up the implementation. The final implementation however contains none of the original code and resembles it only in the resulting structure.

We built this into our experimental framework. It constructs a variant of the architecture described in section 3.4.1. This allows us to combine it with other general functions for importing datasets, testing, saving and loading the weights and architecture during and after training. Especially for GAN training we had a function for producing visual results from the generator at some intervals of epochs. This allowed us to monitor the progress of the training outside of looking at the loss.

We chose to focus on transforming the satellite images into maps as this seems to be more applicable. It is possible to reverse the problem and transform maps into satellite images. That is something they did in the original pix2pix paper. We considered implementing a CycleGAN as in section 3.5 which promises even better results. This is because it learns both the problem and the inverse problem. This allows for an additional cycle loss to determine how well it can restore satellite images that have been transformed into maps and back into its original form.

4.2.3 Results

The experiments were memory and computationally intensive and were run on a computing server with GPUs. In practice, to obtain good results with GAN, it usually requires manual tweaking and incremental development. We did not try to improve our results beyond showing that the method works to a convincing level.

We ran the training for up to 6000 epochs but there was little visual improvement after 1000 epochs. The generator and discriminator objective loss was also rather stable at that point. Therefore we present the results from the weights at that point. Evaluation on 12

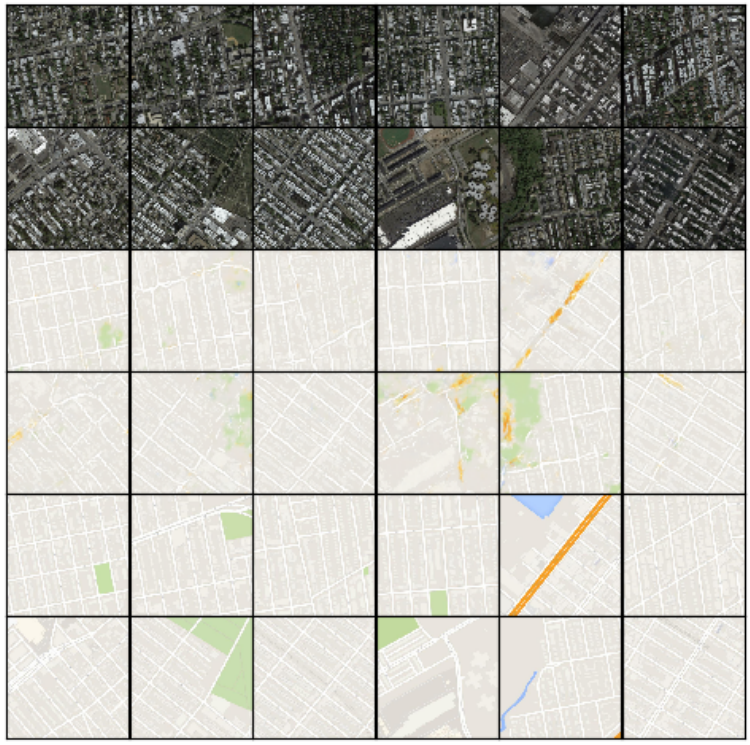


Figure 4.2: After training for 1000 epochs the generator produced these results on the validation dataset. The first two rows is the input. Row 3 and 4 is the generated output. Row 5 and 6 is the ground truth map.

samples from the validation set including input, output, and the ground truth can be seen in figure 4.2.

As you may observe in the figure the generator manages to capture the straight edges of streets. They end up being more squiggly and diffuse than in the ground truth maps. This could perhaps be enforced by having some sort of regularization like in a Banach WGAN. Information that is not explicitly present in the picture, like arrows indicating the direction of traffic flow, is not presented. This is probably due to the low resolution of the images as they are successful with this in the original paper.

It is hard to tell how well the generator has understood the features of the input images and how much is just memorizing earlier samples.

The generator has trouble distinguishing things like highways and wide streets. This is fair as there is generally no clear way for humans to distinguish them either in the context of producing a map. Areas with vegetation like forests or parks are also difficult to capture as some areas are made green and some are kept grey.

An input it fails a lot on is the one in row 2 column 4. Here we have something that looks like a sports arena and some nonstandard buildings. It is possible that further training and scaling up to higher resolutions would resolve this issue.

Still we deem these results quite impressive as they only a few years ago would have been considered almost impossible. Popular research papers about GAN usually have very high quality experimental results. These researchers usually have large computational resources and an experienced engineering team for support. This does not diminish their achievements, but research in deep learning has a tendency to require more and more resources and technical experience. This might be unfortunate for a fair and open research community in the years to come.

Chapter 5

Further work and ideas

In this section we will review the theory and applications presented so far. Then we will present some ideas for further application and innovation with regards to GAN or general machine learning. The development of GAN is far from over and it might be a stepping stone to even more advanced and applicable models in the future.

With regards to image processing and generation GAN has shown its strength. This is the application area where the results are easiest to judge.

5.1 Using GAN to enhance and augment object detection

Object detection is an application of machine learning that is attracting more and more interest. It has applications in medicine, industry and scanning through satellite images. There are several variants of the problem with varying degree of difficulty. It can be the task of detecting a single type of object like cars in a parking lot seen from above. The task is then to draw boxes around each car and then present all of the boxes. The result of this can for example be used to count the number of cars in a parking lot. This is a much more difficult problem than simply classifying whether a picture is of a car or not.

Further on the problem can be extended by introducing more classes of objects to detect. The task then extends to not only draw boxes around the objects of interest, but also classifying each object. For objects of more complicated geometry like lakes it can be useful to sketch the area as a geometric figure. This is called image segmentation.

In the paper by Nogues et al. (2018) they use GAN to extend a synthetic training set and make it look like real data. The task was to design an object detection algorithm for inspecting electric components on an assembly line. They employ a CycleGAN to convert images produced by a physics simulation engine into real world images. The actual object detection is also implemented into this algorithm with a Mask-RCNN.

We had ambitions of implementing a method similar to this, but lacked the time to do so. It would be a fully GAN enabled system where even the object detection was subject to adversarial training.

5.2 Octave convolution

For image analysis the conventional convolution has a lot of spatial redundancy. Octave convolution factorizes the input into high and low frequency separated by an octave and computes the convolution over these two frequencies while passing information between them. This is especially relevant for analyzing natural images. It is orthogonal to other convolution methods that suggest better topologies or reduce redundancy.

This technique is from a recent paper by Chen et al. (2019). It gives higher performance while reducing the computational cost of training. It is therefore ideal for designing the next generation of image processing systems.

5.3 Describing machine learning with category theory

Category theory is a mathematical framework for describing mathematical concepts in an intuitive and relationship oriented way. It can be used as a foundation for all of mathematics and has shown great ability to connect different theories of mathematics with similar language. Category theory is not intended to impose a specific view on concepts, but rather provide a general language that unifies different theories. A reference for category theory can be found in the book *Categories for the Working Mathematician* by MacLane (1971).

In the paper by Fong et al. (2017) they describe backpropagation in a category theoretical framework. This provides some intriguing insights into the mechanics of this algorithm and may lead to whole new ways of considering neural networks. Although it is important not to fixate too much on the theoretical aspects and not consider the applicational aspects of machine learning.

5.4 Overall impression of GAN

GAN has been described as the most interesting innovation in machine learning in the last decade. It has given us a general framework for learning different tasks of high complexity.

Generative methods like this might be essential for developing machine learning methods further as the domains and tasks become more and more complex. This is because complicated tasks and large models need increasing amounts of data to be able to compute good training updates and gradients. Generative methods, gradient learning methods, and functional gradients all provide opportunities for computing better learning updates with less data.

In addition the theoretical analysis of deep learning methods will be necessary to understand and guarantee the safety, security, and stability of these methods as they are applied in increasingly sensitive areas. Failure to understand the inner workings of the method may result in accidents and undesired outcomes. Therefore governments and organizations should also fund research for theoretical foundations of these methods; not only exciting applications.

Chapter 6

Conclusion

We have presented some of the the theory and ongoing research regarding generative adversarial networks. This has been a wide ranging and extensive study which shows the amount of challenges and opportunities given by GAN.

We have conducted experiments to demonstrate the power and potential applicability of GANs. There were however many experiments that we did not have the time to conduct. Training GAN is not straight forward even with many modern libraries and publicly available open source implementations of most papers about GAN. There is no free lunch in machine learning and it will always be necessary to understand the technique deeply to be able to master it.

The map generation experiment demonstrates that even complicated data relationships can be modeled with a GAN. Although this particular implementation is not suitable for any practical application. To actually be able to use such an automated technique to draw maps we would need to produce vector map data not images of maps. The map data needs to be scale invariant, contain information about what the objects in the map are, and have a clear definition of the boundaries of different objects. That is why we also investigated the possibility of using GAN for object detection which could be used to produce map data. There was however not enough time do start doing experiments on this.

Bibliography

Adler, J., Lunz, S., 2018. Banach wasserstein gan. In: Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., Garnett, R. (Eds.), Advances in Neural Information Processing Systems 31. Curran Associates, Inc., pp. 6754–6763.
URL http://papers.nips.cc/paper/7909-banach-wasserstein-gan.pdf

Arjovsky, M., Bottou, L., 2017. Towards principled methods for training generative adversarial networks. CoRR abs/1701.04862.

Arjovsky, M., Chintala, S., Bottou, L., 2017. Wasserstein gan. arXiv preprint arXiv:1701.07875.

Barber, D., Agakov, F. V., 2003. The IM Algorithm: A Variational Approach to Information Maximization.

Chen, T., Fox, E. B., Guestrin, C., Feb 2014. Stochastic Gradient Hamiltonian Monte Carlo. arXiv e-prints, arXiv:1402.4102.

Chen, X., Duan, Y., Houthooft, R., Schulman, J., Sutskever, I., Abbeel, P., 2016. Infogan: Interpretable representation learning by information maximizing generative adversarial nets. CoRR abs/1606.03657.
URL http://arxiv.org/abs/1606.03657

Chen, Y., Fan, H., Xu, B., Yan, Z., Kalantidis, Y., Rohrbach, M., Yan, S., Feng, J., 2019. Drop an octave: Reducing spatial redundancy in convolutional neural networks with octave convolution.

Cover, T. M., Thomas, J. A., 2006. Elements of Information Theory (Wiley Series in Telecommunications and Signal Processing). Wiley-Interscience, New York, NY, USA.

Fong, B., Spivak, D. I., Tuyéras, R., Nov 2017. Backprop as Functor: A compositional perspective on supervised learning. arXiv e-prints, arXiv:1711.10455.

Gavranovi, B., 2019. Graph from The GAN Zoo.
URL https://github.com/hindupuravinash/the-gan-zoo

Goodfellow, I., Bengio, Y., Courville, A., 2016. Deep Learning. MIT Press, http://www.deeplearningbook.org.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., Bengio, Y., 2014. Generative adversarial nets. In: Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N. D., Weinberger, K. Q. (Eds.), Advances in Neural Information Processing Systems 27. Curran Associates, Inc., p. 2672–2680.
URL http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf

Gulrajani, I., Ahmed, F., Arjovsky, M., Dumoulin, V., Courville, A. C., 2017. Improved training of wasserstein gans. In: Advances in Neural Information Processing Systems. pp. 5767–5777.

He, H., Wang, H., Lee, G.-H., Tian, Y., 2018. ProbGAN: Towards Probabilistic GAN with Theoretical Guarantees.

Heusel, M., Ramsauer, H., Unterthiner, T., Nessler, B., Hochreiter, S., 2017. Gans trained by a two time-scale update rule converge to a local nash equilibrium. In: Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., Garnett, R. (Eds.), Advances in Neural Information Processing Systems 30. Curran Associates, Inc., pp. 6626–6637.
URL http://papers.nips.cc/paper/7240-gans-trained-by-a-two-time-scale-update-rule-converge-to-a-local-nash-equilibrium.pdf

Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., Salakhutdinov, R. R., 2012. Improving neural networks by preventing co-adaptation of feature detectors. arXiv preprint arXiv:1207.0580.

Hong, Y., Hwang, U., Yoo, J., Yoon, S., Feb 2019. How generative adversarial networks and their variants work. ACM Computing Surveys 52 (1), 1–43.
URL http://dx.doi.org/10.1145/3301282

Ioffe, S., Szegedy, C., 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift.

Isola, P., Zhu, J.-Y., Zhou, T., Efros, A. A., 2017. Image-to-image translation with conditional adversarial networks. In: Proceedings of the IEEE conference on computer vision and pattern recognition. pp. 1125–1134.

Karras, T., Laine, S., Aila, T., 2018. A style-based generator architecture for generative adversarial networks.

Lucic, M., Kurach, K., Michalski, M., Gelly, S., Bousquet, O., Nov 2017. Are GANs Created Equal? A Large-Scale Study. arXiv e-prints, arXiv:1711.10337.

Luenberger, D. G., 2010. Optimization by vector space methods. Wiley.

MacLane, S., 1971. Categories for the Working Mathematician. Springer-Verlag, New York, graduate Texts in Mathematics, Vol. 5.

Maschler, M., Solan, E., Hellman, Z., Borns, M., Zamir, S., 2018. Game theory. Cambridge University Press.

Milgrom, P., Segal, I., 2002. ENVELOPE THEOREMS FOR ARBITRARY CHOICE SETS.

Mirza, M., Osindero, S., 2014. Conditional generative adversarial nets. arXiv preprint arXiv:1411.1784.

Nitanda, A., Suzuki, T., 2018. Gradient Layer: Enhancing the Convergence of Adversarial Training for Generative Models.

Nogues, F. C., Huie, A., Dasgupta, S., 2018. Object detection using domain randomization and generative adversarial refinement of synthetic images.

Qin, Y., Mitra, N., Wonka, P., 2018. How does lipschitz regularization influence gan training?

Ronneberger, O., P.Fischer, Brox, T., 2015. U-net: Convolutional networks for biomedical image segmentation. In: Medical Image Computing and Computer-Assisted Intervention (MICCAI). Vol. 9351 of LNCS. Springer, pp. 234–241, (available on arXiv:1505.04597 [cs.CV]).
URL http://lmb.informatik.uni-freiburg.de/Publications/2015/RFB15a

Saatci, Y., Wilson, A. G., 2017. Bayesian gan. In: Advances in neural information processing systems. pp. 3622–3631.

Salimans, T., Goodfellow, I., Zaremba, W., Cheung, V., Radford, A., Chen, X., 2016. Improved techniques for training gans. In: Advances in neural information processing systems. pp. 2234–2242.

Sørby, A. C., 2019. On methods for analyzing and improving deep learning.

Villani, C., 2008. Optimal Transport: Old and New. Grundlehren der mathematischen Wissenschaften. Springer Berlin Heidelberg.
URL https://books.google.no/books?id=hV8o5R7_5tkC

Zhu, J.-Y., Park, T., Isola, P., Efros, A. A., 2017. Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks.

Appendix

A Transfer learning

Transfer learning is about training on a task that is somewhat similar to the original task and then using the information obtained in that task to solve a more difficult task. This is for example useful if you have too little data to properly learn a task.

A task is defined as a supervised learning problem, $\mathcal{T} = (X, Y)$, where the point is to learn a function, f, such that $f(X) \approx Y$. Here X and Y are random variables, but we can also consider them as datasets. Usually you would minimize a loss function, $\mathcal{L}(f, X, Y)$, which is a measure of how close the function is to the appropriate form. However this is very dependent on the amount and quality of the data you have.

If you have a task $\mathcal{T}_1$ that is easy to learn and another task $\mathcal{T}_2$ that is difficult then transfer learning can leverage $\mathcal{T}_1$ to improve $\mathcal{T}_2$.

We are assuming that the data comes from a distribution which is similar to the one of the initial task. That way only small adjustments are needed to convert the problem into the domain we want to learn.

With neural networks transfer learning usually consists of training a network on a supervised task where we already have a good labeled dataset. After training we can reuse those weights in a new neural network. We keep the old layers and attach new layers that are going to tackle the specifics of the new task. We do not update the weights of the old layers during training to prevent them from being destroyed from noisy training. This is called freezing. When the model starts to perform well we can start to update these layers as well to fine tune the model.

B Information theory

In this section we introduce basic concepts from information theory which we will need to explain the theory. Information theory is a theoretical framework for understanding information content and processing in a system mostly relying on the concept of entropy. A general reference to the subject is the book *Elements of information theory* by Cover and Thomas (2006). Entropy is a measure of the information content in a random variable or similarly its distribution, say $X \sim P(x)$, and can be related to the expected number of bits it would require to represent it

$$H(X) = -\mathbb{E}_X[\log(P(X))] \geq 0. \tag{1}$$

Entropy can be considered as an alternative view of probability which ranges from 0 to ∞. As with probability there is a conditional entropy

$$H(X|Y) = -\mathbb{E}_X[\log(P(X|Y))|Y]. \tag{2}$$

Then there is the cross entropy where we examine the description of one random variable over the distribution of another

$$H(X,Y) = -\mathbb{E}_X[\log(P(Y))]. \tag{3}$$

Mutual information is a measure of the information a given random $X \sim P$ variable has about another variable $Y \sim Q$.

$$I(X;Y) = H(X) - H(X|Y) \tag{4}$$

We have the Kullback-Leibler (KL) divergence also called the relative entropy where

$$D_{KL}(P\|Q) = H(P,Q) - H(P) = \mathbb{E}_{X\sim P}\left[\log\left(\frac{P(X)}{Q(X)}\right)\right] \geq 0 \tag{5}$$

which measures the difference of a distribution Q to a reference distribution P. As you may note the KL-divergence is not symmetrical. There are several ways to make this divergence symmetrical and one of them is the Jensen-Shannon (JS) divergence. It sums over the KL-divergence for each distribution to a mean distribution

$$D_{JS}(P\|Q) = \frac{1}{2}D_{KL}(P\|M) + \frac{1}{2}D_{KL}(Q\|M). \tag{6}$$

The mean distribution is defined as $M(X) = \frac{P(X)+Q(X)}{2}$.

Note that the JS-divergence and KL-divergence is a special case of the f-divergence

$$D_f(P\|Q) = \int f\left(\frac{p(x)}{q(x)}\right) q(x)dx \tag{7}$$

where p and q are the probability densities of P and Q respectively. The function f can be any convex function where $f(1) = 0$. The JS and KL-divergence corresponds to a choice of f. For KL-divergence $f(t) = t\log t$ and JS-divergence $f(t) = t\log t - (t+1)\log(t+1)$.