UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO DE CIÊNCIAS, TECNOLOGIAS E SAÚDE DO CAMPUS ARARANGUÁ
CURSO DE GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO

Patrick Hoeckler

An Analysis of Techniques for Building Generative Adversarial Networks

Araranguá
2021

Patrick Hoeckler

An Analysis of Techniques for Building Generative Adversarial Networks

Trabalho de Conclusão de Curso do Curso de Graduação em Engenharia de Computação do Centro de Ciências, Tecnologias e Saúde do Campus Araranguá da Universidade Federal de Santa Catarina para a obtenção do título de Bacharel em Engenharia de Computação.
Orientador: Prof. Fabrício de Oliveira Ourique, Dr.

Araranguá
2021

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Hoeckler, Patrick
An analysis of techniques for building generative adversarial networks / Patrick Hoeckler ; orientador, Fabrício de Oliveira Ourique, 2021.
117 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Campus Araranguá, Graduação em Engenharia de Computação, Araranguá, 2021.

Inclui referências.

1. Engenharia de Computação. 2. Generative Adversarial Networks. 3. Generative Models. 4. Deep Learning. 5. Neural Networks. I. Ourique, Fabrício de Oliveira. II. Universidade Federal de Santa Catarina. Graduação em Engenharia de Computação. III. Título.

Patrick Hoeckler

An Analysis of Techniques for Building Generative Adversarial Networks

Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do Título de Bacharel em Engenharia de Computação e aprovado em sua forma final pelo Curso de Graduação em Engenharia de Computação.

Araranguá, 05 de Maio de 2021.

Documento assinado digitalmente
Fabricio de Oliveira Ourique
Data: 17/05/2021 09:04:05-0300
CPF: 916.167.860-00
Verifique as assinaturas em https://v.ufsc.br

Prof. Fabrício de Oliveira Ourique, Dr.
Coordenador do Curso

Banca Examinadora:

Documento assinado digitalmente
Fabricio de Oliveira Ourique
Data: 17/05/2021 09:04:38-0300
CPF: 916.167.860-00
Verifique as assinaturas em https://v.ufsc.br

Prof. Fabrício de Oliveira Ourique, Dr.
Orientador

Documento assinado digitalmente
Antonio Carlos Sobieranski
Data: 17/05/2021 10:18:35-0300
CPF: 005.305.809-77
Verifique as assinaturas em https://v.ufsc.br

Prof. Antônio Carlos Sobieranski, Dr.
Avaliador
Universidade Federal de Santa Catarina

Documento assinado digitalmente
Alexandre Leopoldo Goncalves
Data: 17/05/2021 09:43:27-0300
CPF: 786.740.349-34
Verifique as assinaturas em https://v.ufsc.br

Prof. Alexandre Leopoldo Gonçalves, Dr.
Avaliador
Universidade Federal de Santa Catarina

For CEP

ACKNOWLEDGEMENTS

Eu gostaria muito que essa parte estivesse em branco, que eu pudesse dizer que fiz tudo sozinho e não tive nenhuma ajuda, mas não foi bem assim que aconteceu, eu não teria chegado até aqui se eu tivesse vindo sozinho. Tem algumas pessoas que eu gostaria de agradecer.

Mantendo o clichê, eu primeiro tenho que agradecer a minha mãe. Ela não teve muita influência direta nessa minha formação, eu até estava me questionando se fazia sentido incluir ela aqui já que não parecia que ela teve muito a ver com essa parte da minha vida. Mas aí eu percebi que eu estava sendo maluco, eu não consigo pensar em nenhuma pessoa que tenha feito mais por mim do que minha mãe, se olhar para as influências indiretas vai dar de perceber claramente que elas são bem maiores do que qualquer influência que os outros possam ter tido.

Eu tenho quase certeza de que se tirassem qualquer outra pessoa da minha vida eu ainda teria conseguido chegar aqui, seria bem mais sofrido, mas eu teria chegado. Mas se tirasse a minha mãe eu nem sei se eu teria conseguido começar a faculdade. O clichê é clichê por um motivo, obrigado mãe.

Mas partindo pra influências mais diretas, meu pai, fazer a faculdade sem a ajuda que meu pai me deu teria sido muito mais sofrido. Ele foi quem me manteve alimentado e com um teto em cima da cabeça, eu não sei como eu teria feito sem tua ajuda pai.

Seguindo o baile para os meus colegas de apartamento, Gustavo (o Gino) e Caio. O Gino foi quem me apresentou esse curso e dividiu o apartamento comigo durante todo o curso, o Caio entro lá pelo meio do tempo pra trazer mais alegria e principalmente pra baratear mais o aluguel, sempre é bom. Bons tempos, boas lembraças, algumas não tão boas, mas ainda estamos no lucro.

A meus bons amigos que fiz no curso, em especial o Ramom, o Ricardo e o Felipe, grandes camaradas. Também tive o prazer de trabalhar com alguns professores aqui da UFSC que me fizeram aprender muita coisa e conseguir uma renda extra (sempre é bom), em especial gostaria de agradecer ao professor Marcelo Zannin da Rosa, ótima pessoa e com um ótimo gosto para camisetas. Espero que a bondade que vocês mostraram para mim possa voltar para vocês.

Também gostaria de agradecer meus avaliadores Alexandre L. Gonçalves e Antônio C. Sobieranski junto com meu orientador Fabrício de Oliveira Ourique, minhas aulas com vocês estão entre as melhores e as quais eu aprendi mais. O Fabrício também foi um ótimo orientador, além de me ajudar com o TCC me ajudou com várias dúvidas que eu tinha sobre o estágio.

Por último eu gostaria de agradecer a todos aqueles que estão lutando pela educação gratuita, com certeza essas pessoas tiveram um enorme impacto em tudo o que eu aprendi aqui na faculdade. Tudo o que eu fiz nesse TCC eu aprendi gratuitamente, eu acho incrível

que eu possa sentar na minha casa e assistir cursos completos do MIT, de Harvard, de Stanford e todos os outros. Eu não teria aprendido tanto se não fosse por todas essas pessoas e eu torço que isso consiga chegar pra todo mundo um dia.

ABSTRACT

Generative Adversarial Networks (GANs) are a subcategory of Artificial Neural Networks where the objective is the generation of new data, they do that by modeling the probability distribution of real data, usually coming from a dataset, and sampling from the modeled distribution in order to produce original data that is similar, and optimally indistinguishable, from what was used in training. The principle behind GANs is based on a competition between two different networks, a discriminator who tries to distinguish real from fake data, and a generator who tries to fool the discriminator by producing data that is as close to the real one as possible. However, the competition between the networks makes training GANs be something notoriously difficult, instability and non-convergence are a common occurrence and many techniques have been proposed to improve not only the learning process, but also the quality of the generated results. The goal for this document was to analyse a number of the most common approaches and make an empirical evaluation of those, trying to apply the techniques in different datasets and seeing which configuration produces the best results. In the end there should be a roadmap that can be used to help guide the initial decisions about what method to use when constructing GANs for new and unknown situations.

Keywords: Deep Learning. Neural Networks. Generative models. Generative Adversarial Networks. GAN.

RESUMO

Generative Adversarial Networks (GANs) são uma subcategoria de Rede Neurais Artificiais onde o objetivo é a geração de novos dados, elas fazem isso tentando modelar a distribuição de probabilidades de dados reais, geralmente vindos de um dataset, e amostrando da distribuição modelada de modo a produzir dados originais que são similares, e idealmente indistinguíveis do que foi usado durante o treino. O princípio por trás de GANs é baseado em uma competição entre duas redes distintas, um discriminador que tenta distinguir entre dados reais e falsos, e um gerador que tenta enganar o discriminador produzindo dados que são o mais perto possível dos dados reais. Entretanto, a competição entre as duas redes faz do treinamento de GANs algo que é notoriamente difícil, instabilidade e não-convergência são ocorrências comuns e muitas técnicas foram propostas para melhorar não apenas o processo de aprendizado, mas também a qualidade dos resultados gerados. O objetivo deste documento foi de analisar um número de abordagens mais comuns e realizar uma avaliação empírica destas, tentando aplicar as técnicas em diferentes datasets e observando qual configuração produz os melhores resultados. Ao fim deve haver um roteiro que pode ser usado para ajudar a guiar as decisões iniciais sobre qual método utilizar ao construir GANs para novas situações desconhecidas.

Palavras-chave: Deep Learning. Neural Networks. Modelos generativos. Generative Adversarial Networks. GAN.

LIST OF FIGURES

LIST OF ABBREVIATIONS AND ACRONYMS

Adam	Adaptive Moment Estimation
BN	Batch Normalization
BS	Batch Size
CGAN	Conditional GAN
CIFAR	Canadian Institute for Advanced Research
CLIP	Clipping value for the parameters of the critic
CNN	Convolutional Neural Network
CS	Classifier Score
DCGAN	Deep Convolutional GAN
DL	Deep Learning
EM	Earth-Mover
FCD	Fréchet Classifier Distance
FID	Fréchet Inception Distance
GAN	Generative Adversarial Network
IS	Inception Score
KNN	K-Nearest Neighbours
LDIM	Dimensions of latent space
LSTM	Long Short-Term Memory
MNIST	Modified National Institute of Standards and Technology database
MSE	Mean Squared Error
NCRIT	Number of updates to critic before update to generator
OPT	Optimizer
ReLU	Rectified Linear Unit
ResNet	Residual Networks
RNN	Recurrent Neural Network
SGD	Stochastic Gradient Descent
SMOOTH	Value of one-sided label smoothing
SVM	Support Vector Machine
tanh	hyperbolic tangent
UP	Upscaling Method
WGAN	Wassertein GAN
WGAN-GP	WGAN with Gradient Penalization

LIST OF SYMBOLS

σ	Sigmoid function
ϵ	Computational stability term
η	Learning Rate
λ	Regularization strength
$\mathcal{Z}$	Latent space of the generator
$\mathcal{X}$	Input space
$\mathbb{E}$	Expected value
β_1	Momentum hyperparameter of Adam

CONTENTS

1 INTRODUCTION

There is a subfield in machine learning called generative modeling which is concerned with the task of generating new data from what already exists. The goal is to see a great amount of data in order to understand how it is structured, more formally, try to represent its probability distribution.

Generative Adversarial Networks (GANs) are a relatively new approach to generative modeling that were proposed by Goodfellow, Pouget-Abadie, et al. (2014). The main idea behind them is a competitive game between a generator and a discriminator, usually implemented as neural networks. The generator creates fake data while the discriminator tries do distinguish it from the real data. The goal of the generator is to produce data as realistic as possible in order to fool the discriminator, which in turn tries its best not to be fooled. The idea is that, in the end the generator will be so good at its job that it will be impossible for the discriminator to see any difference between real and fake data.

1.1 JUSTIFICATION

Generative models may seem superfluous at first sight, since the main idea behind them is generating something similar to the already numerous data used to train the models in the first place. Creating more of what there is already plenty of is really not that useful in many cases, but generative modeling goes much further than that. The idea is to create an understanding of how the data is structured, allowing for going beyond than simple generation, including transformation, combination, re-imagination, and more.

Examples include automatic colorization of black and white photos (NAZERI; NG, 2018), upscaling images to higher qualities (LEDIG et al., 2016), filling missing details in images (YU et al., 2018), automatic artistic renditions of photos (KARRAS; LAINE; AITTALA, et al., 2020), and simulating possible futures for training Reinforcement Learning models (GOODFELLOW, 2017). But simple generation can also be desirable, as for the case of generating or continuing pieces of music (DHARIWAL et al., 2020).

GANs are a big part of generative modeling, since their introduction they have become increasingly popular, initially being used only for image generation, now they are employed over many other scenarios. One of their advantages over other models is the different way that they approximate the data distribution, which is often more useful for practical situations (ARJOVSKY; CHINTALA; BOTTOU, 2017), and also the fact that they can learn to see a problem as having multiple possible solutions and being able to pick a single one instead of averaging out all of them (GOODFELLOW, 2017).

1.2 PROBLEM

GANs however, are infamous for being particularly difficult to train, the adversarial game that is used in training them can result in an infinite loop around the optimal solution (ARJOVSKY; CHINTALA; BOTTOU, 2017). There are also many proposed improvements to the original GAN architecture, making it hard to decide which one is the right choice for a particular situation, or which one would generally be a good option as a starting point to build from.

Even after deciding the type of GAN, the process of building it, usually entails a long search of good hyperparameters that can make learning possible. When training GANs a difficult situation can happen quite frequently, where the results faced are completely unusable and there is no clear direction as to what went wrong.

1.3 OBJECTIVES

The goal of this document is to explore the theory behind GANs, how they work, what problems they have, what are some solutions to these problems, and in the end, compile all this information and run several experiments that will be used to empirically validate the effectiveness of different approaches in the particular and general cases.

By the end of the experiments there should be enough information to build a roadmap to help guide the construction of a GAN, detailing which methods have a good chance of producing good results, what are the common problems that may impede progress and their corresponding solutions, and what should be avoided in most cases. It is important that the techniques analysed have a high chance of applying generally to many situations and not just be confined to a single problem.

1.4 METHODOLOGY

The process of creating this document consisted first of research in the area, from the concept of GANs to the many proposed improvements to them. Following the research there was a selection for the different techniques that could be implemented and for some good datasets to train the models. Lastly the experiments were made, observing different hyperparameters and how they can influence the overall performance of the model. The results of the experiments are all described in Chapter 5 of this document.

For building the neural networks and running the experiments, the Python programming language was chosen and the open-source, machine-learning library TensorFlow (ABADI et al., 2015) was used for building and training all the models. The libraries TensorFlow-GAN, NumPy and Matplotlib were also extensively used to respectively: evaluate the models, handle general numerical calculations, and generate the data visualizations.

All code used for this project is free and open-source, being found on the GitHub repository at the link `https://github.com/PatrickHoeckler/tcc_gan`. The code is written mainly in Jupyter notebooks, this was chosen so that it could contain additional information that adds more clarity, so if the code is explored by someone who is not familiar with how it works, it is still possible to transmit better the idea behind it.

1.5 NOTATION

This document will follow the notation proposed by (GOODFELLOW; BENGIO; COURVILLE, 2016, p. xiii-xvi), in particular the following:

- Simple lowercase symbol: a - single dimension scalar value
- Bold lowercase symbol: $\boldsymbol{a}$ - vector
- Bold uppercase letter: $\boldsymbol{A}$ - matrix
- Simple superscript: a^b - normal exponentiation
- Parenthesis superscript: $a^{(b)}$ - situation specific index

Other notation will be explained as it appears throughout the document.

2 DATASETS

To understand the concepts explored in this document it is sometimes helpful to bring real world examples in order to represent the theoretical ideas in more familiar terms. This chapter will introduce the datasets relevant to this document, used for explaining the concepts, but mostly for performing the experiments that will be described in Chapter 5.

For any machine learning problem there is the desire to model something, some practical examples could be: how likely a person is to have a disease given a set of medical conditions; what type of animal an image represents; or what is the best move to make given a board position in chess. Whatever the underlying situation being modeled, it is necessary to have some data to build the model around.

This data can be obtained through self play (e.g. in Reinforcement Learning problems), but in the majority of cases it is given by a dataset. A dataset is simply a collection of samples from the situation being modeled, it does not contain all the possible values but, if sufficiently expansive, it should have enough samples to be a good representation of the distributions and particularities of the modeled situation. The goal of a dataset is to contain enough data, so that a machine learning algorithm trained on it can generalize well to data outside of it.

For neural networks a dataset is commonly divided into three groups: training, validation and test data. The training data is used in the learning process, it is what the network will see and will try to model, given this importance it is usually the largest portion of a dataset. The validation data on the other hand is used to decide how to build the network and how to train the model, another way of saying this is that the training data is used to tune the network's parameters, while the validation data is used to tune the hyperparameters (see section 3.3).

The validation process consists of training several models on the usually smaller validation data and seeing which set of hyperparameters produced the better results. One might wonder why would there be a need for this data and why not just use the training data instead? The main benefit of using a different set for validation is that validating on the training data has the risk of finding a set of hyperparameters that is particularly good on this data but that does not generalize well, using a separate validation data is a way to not overfit the hyperparameters to the training data and achieve better generalization.

The last part of a dataset, the test data, is used to validate the quality of the model and its hability to generalize. This data should never be used to update either the parameters or hyperparameters, it should instead only be used as an evaluation tool, a way to estimate how well the model will perform on unseen data.

The next sections will explore the datasets relevant to the experiments made for this document. It is usual for datasets to already come separated into train and test data (the validation data is usually taken from a subset of the training data only if needed). This

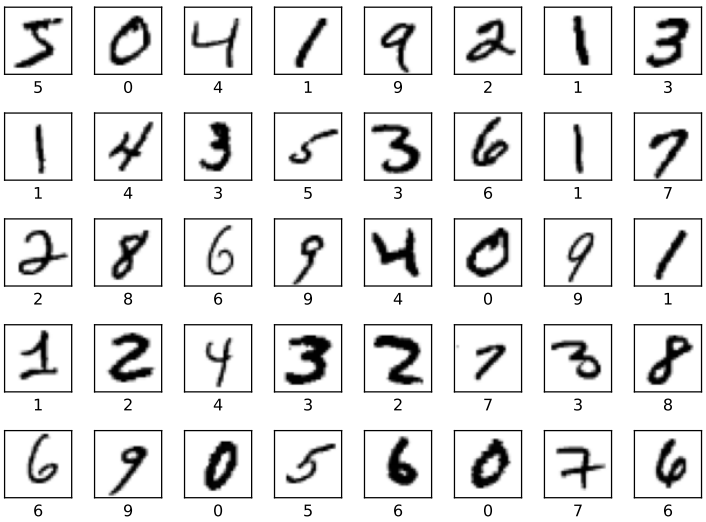
division will be mentioned for the described datasets, but it is relevant to note that any other divisions could also be obtained by combining and redistributing the data differently.

2.1 MNIST

Introduced in 1998 by Yann LeCun et al. (1998), the Modified National Institute of Standards and Technology database (MNIST) is one of the most popular datasets in the field of machine learning, its simplicity has made it a perfect choice as an introduction to deep learning and classification problems (NIELSEN, 2015), but also as a benchmark for new techniques in serious research – some examples include (HINTON et al., 2012), (GOODFELLOW; POUGET-ABADIE, et al., 2014), (MIRZA; OSINDERO, 2014) and (KINGMA; BA, 2017).

This dataset consists of 70,000 (60,000 training and 10,000 test) gray-scale images of handwritten digits, all images are of size 28×28 pixels and are labeled with the corresponding digit. The pixel values are inverted, this means that the strength of the strokes are represented with white pixels (values close to 255) against a black background (pixel value 0), this is however just how the data is represented numerically, for visualization purposes it is better to invert the colors as seen on Figure 1 – This figure shows some samples from this dataset along with the corresponding label.

Figure 1 – Labeled samples from the MNIST dataset



Source – From the author (2021)

2.2 FASHION MNIST

The simplicity of the MNIST dataset makes it a very natural choice for benchmarking a Neural Network, however the data that it represents is also very simplistic – Wan et al. (2013) were able to achieve a classification error lower than 0.3% on the test set.

The fact that MNIST can be too easy has raised some questions about the usefulness of this dataset in benchmarking methods that scale to more complex tasks.

In response to these questions Xiao, Rasul, and Vollgraf (2017) proposed the Fashion MNIST dataset, arguing that MNIST is too easy and cannot represent modern computer vision problems. Their goal was to replace MNIST with a more robust dataset, without losing the simplicity of use that made the original so popular in the first place.

The Fashion MNIST dataset has all the same properties of MNIST, it consists of 70,000 (60,000 training and 10,000 testing) 28×28 gray-scale images labelled from 0 to 9. The images however do not represent handwritten digits, they are instead preprocessed pictures of clothing items from the Zalando fashion company (XIAO; RASUL; VOLLGRAF, 2017), the labels directly map to the type of clothing represented. Just like in MNIST, the pixel values for the images are also inverted, the authors have made an effort to make the change of datasets as simple as just changing the link to get the files.

Figure 2 shows some labeled samples from this dataset, the pixel values are inverted for better visualization.

Figure 2 – Labeled samples from the Fashion MNIST dataset

Source – From the author (2021)

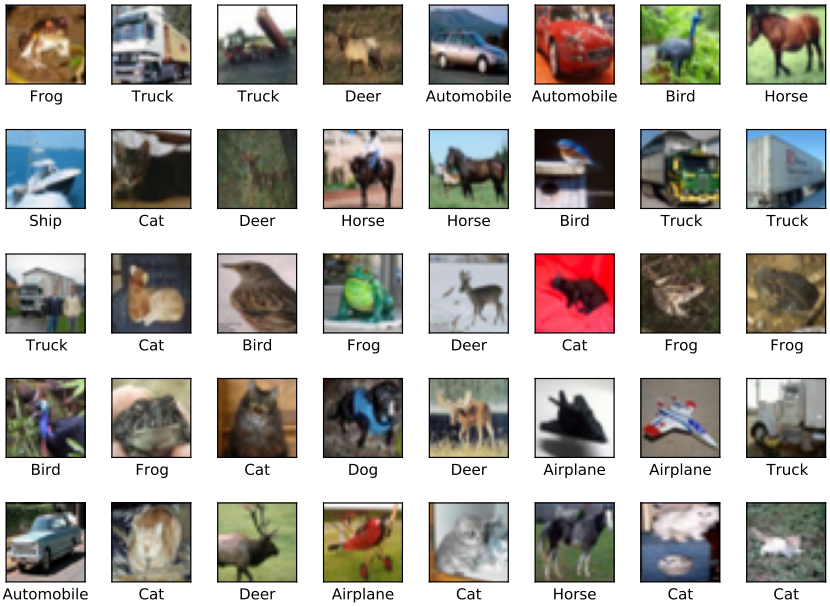
2.3 CIFAR-10

The Canadian Institute for Advanced Research (CIFAR) datasets, CIFAR-10 and CIFAR-100, are two different subsets of the much larger 80 Million Tiny Images dataset, both are made of 60,000 (50,000 training and 10,000 testing) colored natural images of size 32×32 that were labeled by paid students to fit in a set of classes.

The images from CIFAR-10 are divided into 10 classes with 6,000 images each, while CIFAR-100 has 100 classes with 600 images each (KRIZHEVSKY; HINTON, et al., 2009). For this document, only the CIFAR-10 dataset was chosen for the experiments.

The CIFAR datasets are another very popular choice for benchmarking neural networks, but given that they consist of colored images with increased resolution and more complex classes they offer considerably more challenge when compared to the MNIST dataset. Figure 3 shows examples of labeled samples taken from the CIFAR-10 dataset.

Figure 3 – Labeled samples from CIFAR10 dataset



Source – From the author (2021)

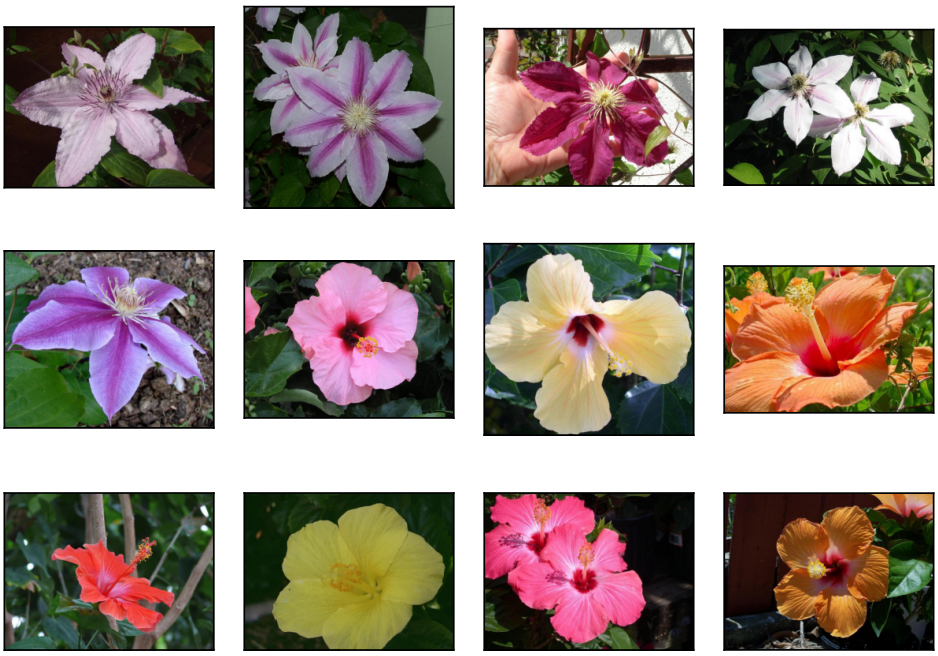
2.4 FLOWERS

The flowers dataset consists of $8,189$ high resolution images of 102 different categories of flowers, each category has from 40 to 250 different images (NILSBACK; ZISSERMAN, 2008). Samples from this dataset can be seen on Figure 4.

2.5 CELEBA

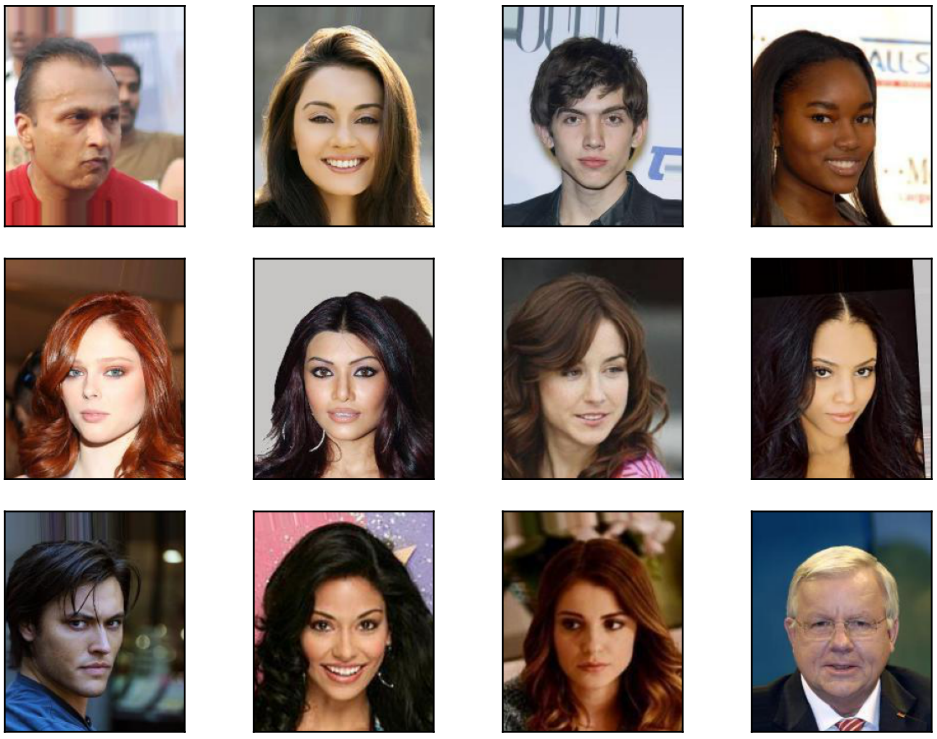
This is the largest dataset used in this document in terms of number of elements, it consists of $202,599$ pictures of faces of celebrities, all rescaled to size 178×218. All images are heavily annotated, having 40 binary features (e.g. blonde hair, eyeglasses, wearing hat, young) and the positions of eyes, nose and mouth all labeled (LIU et al., 2015). However, for the purposes of this document the annotations will not be relevant. Figure 5 shows examples of pictures in this dataset.

Figure 4 – Samples from the Flowers dataset



Source – From the author (2021)

Figure 5 – Samples from the CelebA dataset



Source – From the author (2021)

3 MACHINE LEARNING AND NEURAL NETWORKS

Machine learning is the discipline of computer science where the goal, instead of laying down the steps for a machine to produce a result given some inputs, is in fact to make the machine find by itself ("learn") the correct course of action by showing it relevant data. A more rigorous definition is given by Mitchell (1997).

> A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at task T, as measured by P, improves with experience E. (MITCHELL, 1997, p. 2)

In this definition the experience E can be viewed as the data that is used to teach the machine – this could be for example a dataset of images that the computer is expected to classify with the correct label (see subsection 3.1.1), or also repeated self play to become better at games like chess or Go as seen in reinforcement learning (see subsection 3.1.3), where AlphaZero is a notable example (SILVER; HUBERT, et al., 2017). The task T is what the machine is ultimately trying to achieve (e.g. classify images, play chess) and P is the measure of success in the task (e.g. percentage of accurately classified images, proportion of wins against an opponent in chess).

There are multiple approaches that fall unto the category of machine learning, popular techniques include K-Nearest Neighbours (KNN), decision trees, random forest, Support Vector Machine (SVM), linear and logistic regression, and others[1]. Among the existing methods, neural networks have shown very good results in the last years, specially after the resurgence of deep learning thanks to the increased computational power, better use of parallelization, and the development of frameworks like Pytorch and Tensorflow.

Currently, neural networks are at the forefront of many areas like image classification[2], reinforcement learning[3], and generative modeling; where Generative Adversarial Networks (GANs), which are the main focus of this document, are showing very good results.

This chapter will explain what are neural networks and how they can learn by themselves, this learning process is also often called training. First however, it is worth to have a brief description of the different types of learning.

3.1 TYPES OF MACHINE LEARNING

For a machine to learn it is necessary to have something for it to learn, some set of data that can point it to the desired behaviour. Depending on the type of data available

[1] The paper from Xiao, Rasul, and Vollgraf (2017) gives many performance results for different methods applied on the Fashion MNIST dataset

[2] Big breakthrough in 2012 with AlexNet and the resurgence of Convolutional Neural Networks (KRIZHEVSKY; SUTSKEVER; HINTON, 2012)

[3] AlphaGo was the first ever computer to be able to defeat a human professional player at the game of Go (SILVER; HUANG, et al., 2016) and its successor AlphaZero was able to surpass it by learning entirely through self-play (SILVER; HUBERT, et al., 2017)

it is possible to divide learning into 4 different categories: Supervised; Unsupervised; Semi-supervised; and Reinforcement Learning.

3.1.1 Supervised Learning

The simplest way to teach a computer is to train it with both the input data and the expected output. For example, suppose a classifier that is trained on the MNIST dataset, the goal of this classifier is to take a gray-scale image as input and return the corresponding digit as output. Since MNIST also contains the labels, the training would consist of feeding the images to the learning algorithm and comparing the predicted digits with those given by the labels. The difference between the prediction and the real output can then be used to adjust the classifier in order to make future predictions more likely to be correct.

Learning problems where the data contains both the input and the desired output are known as Supervised Learning problems, this is currently the most common form of learning. The advantage of this approach is that it makes very explicit to the computer what is expected from it, resulting in generally easier learning when compared to other forms of learning that have incomplete data.

The biggest problem with supervised learning is producing the datasets in the first place, obtaining the input data is generally simple, however producing the corresponding outputs (e.g. labels for classification problems) can be very difficult. For problems like image colorization (ZHANG; ISOLA; EFROS, 2016) or super resolution (LEDIG et al., 2016) this is trivial (i.e. convert a colorized image to gray-scale, downscale high resolution images), however for a dataset like MNIST it is necessary to have a human label every single one of the $70,000$ images. For bigger datasets like ImageNet (RUSSAKOVSKY et al., 2015) that contain millions of images, classified into thousands of different classes, and that have annotated bounding boxes for a significant number of the images, the process is very slow and expensive. Amazon Mechanical Turk [4]is often used for such tasks.

3.1.2 Unsupervised Learning

In contrast with supervised learning where all the data is often costly labelled, unsupervised learning refers to problems where only the input data is available and the network has no explicit notion of the desired outputs. These types of problems have a huge potential since the amount of unlabeled data is much larger, however, learning without labels is also much more difficult and it is still an active area of research.

Situations where unsupervised learning can be used are more rare and may require some specific prior knowledge of the datasets to work properly. The MNIST dataset can be used again as a simple example to understand how this type of learning works. Suppose

[4] Mechanical Turk page: `https://www.mturk.com`

that it is desired to build a classifier for handwritten digits, but the only dataset available is MNIST stripped of its labels; it may seem impossible to learn anything from this since all the machine sees are just images without any notion of right or wrong classification.

The trick is to exploit the prior knowledge of the problem and the distributions of the dataset in order to make progress. Since the objective is to build a digit classifier, then one prior information is that the number of possible classes will be 10, or 11 if a class "Not a digit" is also included.

Also consider the size of the input space, for MNIST this is a $28 \times 28 = 784$ dimensional space, and with gray-scale images the number of different possible inputs is $256^{28 \cdot 28} = 2^{6,272} \approx 10^{1888}$. In such high dimensional spaces, all of the sensible inputs (e.g. images of digits) are just a tiny fraction of the whole input volume. This holds true for basically all situations, any random sample from an input space (pixel values, letters, audio amplitude) will almost always fail to produce the structured data present in the real world (pictures, words and phrases, human voice) (GOODFELLOW; BENGIO; COURVILLE, 2016, chap. 5) – it can be said that real life data is *sparse* on the input space.

Along with sparsity, real data is also not evenly spaced on these high dimensional spaces, but it is instead concentrated around a relatively small number of clusters. Goodfellow, Bengio, and Courville (2016, chap. 5) argue this point by noting that it is possible to take similar images and apply some transformations, like moving objects or changing the light, in order to move from one image to another in the cluster. For example, one could take an image of the number 0 in MNIST, add some slight rotations or change the width of the strokes, and end up with a different image that still represented the number 0. However a transformation from the number 0 to the number 3 does not seem to be so simple, so at least in an intuitive sense this example gives an idea of why the sparse data is also clustered around some points.

Using this knowledge of clusters and the number of classes expected in a problem like MNIST, one can build an algorithm like KNN to divide the input space into a desired number of regions, using a measure of similarity between inputs from the dataset as a guide for the divisions. Without using any labels the resulting algorithm is able to tell in which of the regions a given input belongs. For the MNIST case, these regions have strong correlation with the corresponding digit – (XIAO; RASUL; VOLLGRAF, 2017) obtained accuracies of above 95% with this technique.

Of course this approach is very dependent of the dataset and type of problem, it is not always that unsupervised learning will be so straight forward or that the divided regions will correlate well with the desired output.

GANs, the main focus of this document, are also primarily unsupervised learning approaches. In Chapter 4 it will be discussed in detail how they work and learn from the underlying data distribution.

3.1.3 Semi-supervised and Reinforcement Learning

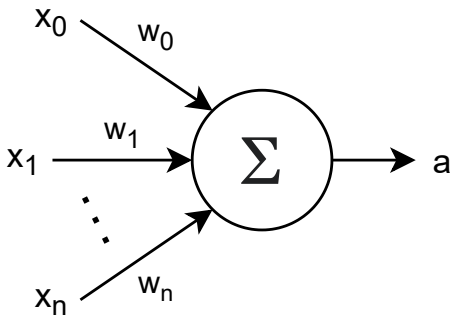
Semi-supervised learning is an intermediate between fully supervised and unsupervised learning, it concerns situations where the dataset is only partially labeled, the idea behind it is to use the vast amounts of unlabelled data to support the training from the relatively small number of labeled data. Since unlabeled data is much more easily available, semi-supervised learning has a lot of potential for building better models by fully leveraging the available data (ZHU, 2005).

Reinforcement learning is particularly different from the other types of learning, its objective is to teach an agent to interact with a changing environment; learning does not occur with a dataset, but is instead achieved through trial-and-error where the agent is rewarded or punished depending on its actions (KAELBLING; LITTMAN; MOORE, 1996). AlphaZero is an example of a reinforcement learning agent, it learned to play chess, shogi, and Go entirely through self-play (SILVER; HUBERT, et al., 2017).

3.2 ARTIFICIAL NEURAL NETWORKS

To understand neural networks, first it is necessary to understand its building blocks, artificial neurons. Commonly called just neurons, nodes, or units (the preferred term used in this document will be unit), these were historically inspired by biological neurons. The idea behind them is that a single unit is a very simple element that receives some inputs and produces an output, the real power comes from connecting many of these together, from thousands, to millions, or even billions of connections[5].

Figure 6 – Single unit representation



Source – From the author (2021)

Figure 6 shows a representation of an artificial unit, it receives a set of inputs $[x_0, x_1, ..., x_n]$, denoted as a vector $\boldsymbol{x}$, and each input has a corresponding scalar value w_i called its weight. Units will also often have a bias term b that is independent of the inputs and that is useful for shifting the output.

[5] As of writing this document GPT-3 is the biggest neural network model of all time, having 175 billion parameters (BROWN et al., 2020).

By changing the set of weights (in vector form $\boldsymbol{w}$) and the bias term, it is possible to achieve different behaviours from the unit based on its inputs. Equation (1) shows how these parameters are used to calculate what is called the *weighted input* (z) of the unit (NIELSEN, 2015, Chapter 2).

$$z = \sum_{i}^{n} w_i x_i + b = \begin{bmatrix} w_0, & w_1, & \dots & w_n \end{bmatrix} \begin{bmatrix} x_0, & x_1, & \dots & x_n \end{bmatrix}^T + b = \boldsymbol{w} \cdot \boldsymbol{x}^T + b \tag{1}$$

The weighted input is usually passed through a non-linear function f, called the *activation function*, to produce the output a, called the unit activation, as seen in Equation (2).

$$a = f(z) = f\left(\boldsymbol{w} \cdot \boldsymbol{x}^T + b\right) \tag{2}$$

A neural network is built by combining multiple units together and the learning process consists of adjusting all the weights and biases in order to produce the expected results given the dataset. Any combination of connections can be considered a neural network, however for the overwhelming majority of cases the networks are divided into layers, and for most of these cases the connections form an acyclic graph. This means that there are no cycles in the network, the input flows from one layer to the next, and there is usually no connection between layers that are not consecutive – exceptions to this are Long Short-Term Memory (LSTM) networks (HOCHREITER; SCHMIDHUBER, 1997) and Residual Networks (HE et al., 2015a), but they are not the focus of this document.

The layers of the network are commonly divided into input layer, hidden layers, and output layer. The input layer represents the input data, usually in diagram representations the inputs are drawn like units, this however is just an stylistic choice since the values do not pass through any calculation in this layer.

The output layer contains the units that will be interpreted as the result produced from the input. For example, in a case where the network is trying to predict the price of apartments given area, number of rooms, and others properties (classical machine learning example), the output would be a single unit whose activation is the predicted price. Problems where the output can assume a range of values are usually called *regression* problems.

On the other hand, problems where the output is better interpreted as a discrete value are called *classification* problems. Predicting the digits of MNIST falls into the category of classification problems, where the output represents which of the 10 possible digits the input image represents.

In the case of MNIST the output layer can be made of 10 units, where each of the activations gives the probability of the input belonging to the corresponding digit. A natural question to raise from this description is: why would there be a need for one unit to represent each class, when the number 10 can be more efficiently represented using only 4 bits (4 units)? Or even, why not use just one unit that outputs the predicted number?

One reason for this is that these simpler representations would lose the property of interpretability of the output as a probability distribution. But the most important reason can be validated empirically, it is easier for a network to classify the inputs into isolated units then it is to try and correlate them with specific bits for each class (NIELSEN, 2015, Chapter 1) or to reduce the output into a single unit.

For most classification problems using a neural network the output layer will contain one unit for each possible class and the desired output will be a vector where all elements are 0's, except for the element corresponding to the true label that will be a 1, indicating the 100% probability for that class. This way of encoding the outputs is called *one-hot encoding*.

The rest of the layers in a neural network, called the hidden layers, are all the layers between the input and output. A neural network does not need to have any hidden layers, but they are fundamental for building more complex relations and robust models.

Hornik, Stinchcombe, and White (1989) showed that, given sufficient hidden units, any neural network with a single hidden layer can be used to approximate any function to any amount of precision, in other words, neural networks with at least a hidden layer are *universal approximators*. But in practice it is observed that more hidden layers usually perform better, they are able to divide the problems into steps and gradually reach the result. For example, an image classifier might use the first layers to distinguish lower level features like edges, while deeper layers recognize shapes, textures, and all the way to complex patterns like faces (GOODFELLOW; BENGIO; COURVILLE, 2016).

More recent years have seen a resurgence of Deep Learning (DL), this is generally understood as learning with networks having at least two hidden layers, but modern models can have much more than that[6].

Figure 7 shows an example of the layered structure of a neural network, for simplicity the diagram shows only fully connected layers (all the units of a layer are connected to all the units of the previous layer, see subsubsection 3.2.1.1) but it could also have different types of layer without losing the general idea of input, hidden, and output.

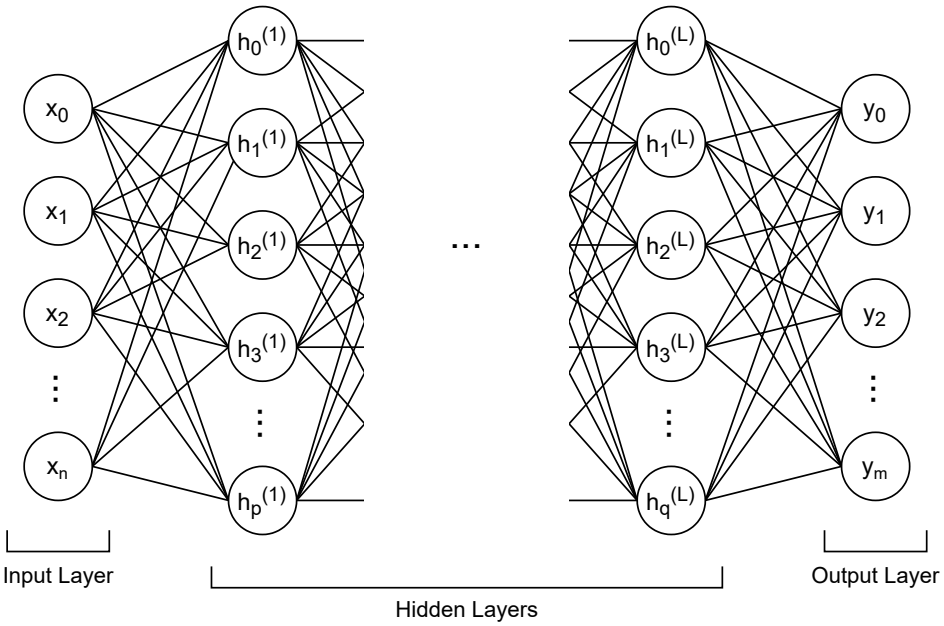
3.2.1 Types of layers

The networks constructed for this document will all use a combination of fully connected and convolutional layers. This section will explain how these work, how they differ, and their advantages in relation to each other.

Before proceeding however, it is important to define the notation that will be used. Since from now on the focus will changes from single units to an entire network, it is necessary to establish a notation that allows for indexing individual units, in any layer, and also their parameters.

[6] Google's Inception v3 image classifier model has 42 layers in total (SZEGEDY; VANHOUCKE, et al., 2015)

Figure 7 – Diagram of a fully connected neural network



Source – From the author (2021)

The bias, weighted input, and activation for unit i in layer l will be written as $b_i^{(l)}$, $z_i^{(l)}$, and $a_i^{(l)}$ respectively. The weight connecting unit j in layer $l-1$, to unit i in layer l will be written as $w_{ij}^{(l)}$.

The set of biases, weighted inputs, and activations in layer l can also be written as the vectors $\boldsymbol{b}^{(l)}$, $\boldsymbol{z}^{(l)}$ and $\boldsymbol{a}^{(l)}$. The weights connecting the units in layer $l-1$ to units in layer l can be written as the weight matrix $\boldsymbol{W}^{(l)}$.

$$\boldsymbol{z}^{(l)} = \begin{bmatrix} z_0^{(l)} \\ z_1^{(l)} \\ \vdots \\ z_n^{(l)} \end{bmatrix} \quad \boldsymbol{a}^{(l)} = \begin{bmatrix} a_0^{(l)} \\ a_1^{(l)} \\ \vdots \\ a_n^{(l)} \end{bmatrix} \quad \boldsymbol{b}^{(l)} = \begin{bmatrix} b_0^{(l)} \\ b_1^{(l)} \\ \vdots \\ b_n^{(l)} \end{bmatrix} \quad \boldsymbol{W}^{(l)} = \begin{bmatrix} w_{00}^{(l)} & w_{01}^{(l)} & \dots & w_{0m}^{(l)} \\ w_{10}^{(l)} & w_{11}^{(l)} & \dots & w_{1m}^{(l)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n0}^{(l)} & w_{n1}^{(l)} & \dots & w_{nm}^{(l)} \end{bmatrix}$$

Recall that the activation of a unit is simply its activation function, denoted here as f, applied to the weighted input. Equation (3) rewrites this relation as shown in Equation (2) using the revised notation.

$$\boldsymbol{a}^{(l)} = f\left(\boldsymbol{z}^{(l)}\right) \tag{3}$$

Observe that in Equation (3) the activation function is applied to the vector $\boldsymbol{z}^{(l)}$, this is a common shorthand notation to represent the element-wise application of the function to the vector. This notation will be used throughout the entirety of this document, all functions applied to vectors will be applied element-wise unless otherwise noted.

3.2.1.1 Fully connected layer

Fully connected, also called Dense layers, are a type of layer where all the inputs are connected to all outputs of the previous layer. Equation (4) shows how the weighted inputs of this layer are calculated from the activations of the previous layer.

$$z_i^{(l)} = \sum_j w_{ij}^{(l)} a_j^{(l-1)} + b_i^{(l)} \tag{4}$$

If the previous layer is the input layer, the values for $\boldsymbol{a}^{(l-1)}$ will be the input of the network. To abstract the type of input it is useful to just replace the notation with a general layer input $\boldsymbol{x}$. The activation can be written more simply by using vector form as shown in Equation (5), the layer superscripts were also omitted for more clarity.

$$\boldsymbol{a} = f\left(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}\right) \tag{5}$$

Dense layers are extremely common, being used in many types of neural networks. They are very useful for mapping their inputs, that can have any number of dimensions, to a vector with any different number of dimensions, this is used in almost all classifiers to reduce the detected features into a one-hot encoded vector of the possible classes. For example, in the 2012 ImageNet challenge the three last layers of the network AlexNet were fully connected, they mapped the $43,264$ features into a 1000 dimensional vector corresponding to all classes of images in the challenge (KRIZHEVSKY; SUTSKEVER; HINTON, 2012).

This type of layer can also be used to apply transformations to features that will be used later in the network, this is used in GANs to map the latent space to a vector that is transformed into the generated image (seeChapter 4). They can even be used for full feature extraction, but this is usually not the best choice since they are quite expensive given the high number of connections and, as will be seen, they lack some useful properties that are present in convolutional layers.

3.2.1.2 Convolutional layer

One drawback of fully connected layers is that they do not leverage the structure of the data when calculating the activations. Consider for example the case of images, when dealing with random values all pixels are uncorrelated with one another, but in real world situations the pixels of an image group together to make edges, shapes and complex figures. The same could be said for audio, video, language and many other situations (DUMOULIN; VISIN, 2018).

Consider the example of a simple image of a digit in the MNIST dataset, by translating the image by a couple of pixels or by slightly warping the strokes the digit represented does not change. But since the fully connected layer has no sense of neighbouring pixels (or temporal coherence for audio and video, etc.), then it has no choice other than learning

weights for all possible slight transformations to the inputs. This not only makes learning more difficult, but also introduces many unnecessary parameters to the network.

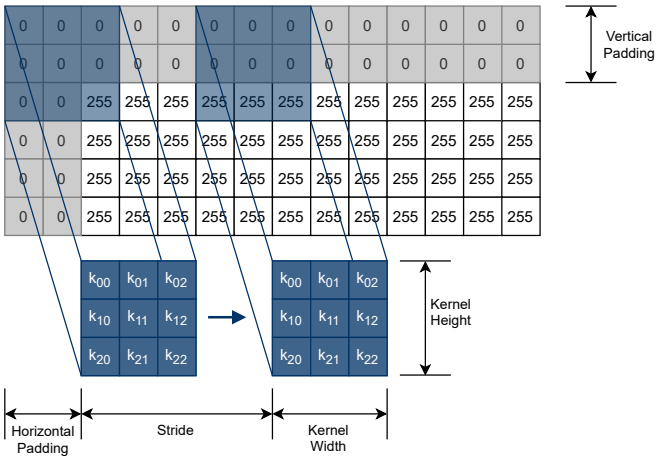
Convolutional layers are a way of dealing with this problem. Initially inspired by the visual cortex of vertebrates (FUKUSHIMA; MIYAKE, 1982), the idea behind them is to detect the presence of features, no matter where they are present or if they are slight perturbed (e.g. an edge should be seen as an edge, no matter where it is located in the image or if it is slightly rotated).

Neural networks that employ convolutional layers are usually called Convolutional Neural Networks (CNNs), their history is very long and they were already used in the 1990's for learning the MNIST dataset when it was introduced (LECUN, Y. et al., 1998). However they were not very popular for larger problems and only grew in popularity after the great breakthrough in the ImageNet challenge achieved by the CNN AlexNet in 2012 (KRIZHEVSKY; SUTSKEVER; HINTON, 2012). Since then they have become very common and are used in a variety of situations, even outside of image recognition.

This section will only explain how they work for the 2-dimensional case, but the idea can be easily generalized to the 1-dimensional or multidimensional cases. The inputs of a convolutional layer share the properties that they are represented as multidimensional arrays, have one or more axis where the ordering matters (for images these are the width and height), and can have an axis representing different views of the data (e.g. the RGB channels for a colored image) (DUMOULIN; VISIN, 2018).

The name convolution is not a coincidence, the layer operation is related to the convolutions seen in signal processing for 1D discrete signals, the image convolutions like gaussian and Sobel filters, or higher dimensions mathematical convolutions. The operation consists of sliding a window of weights, called the kernel, over the entire input and calculating the sum of the weighted inputs covered by this window to produce the resulting values.

Figure 8 – Convolutional layer kernel properties



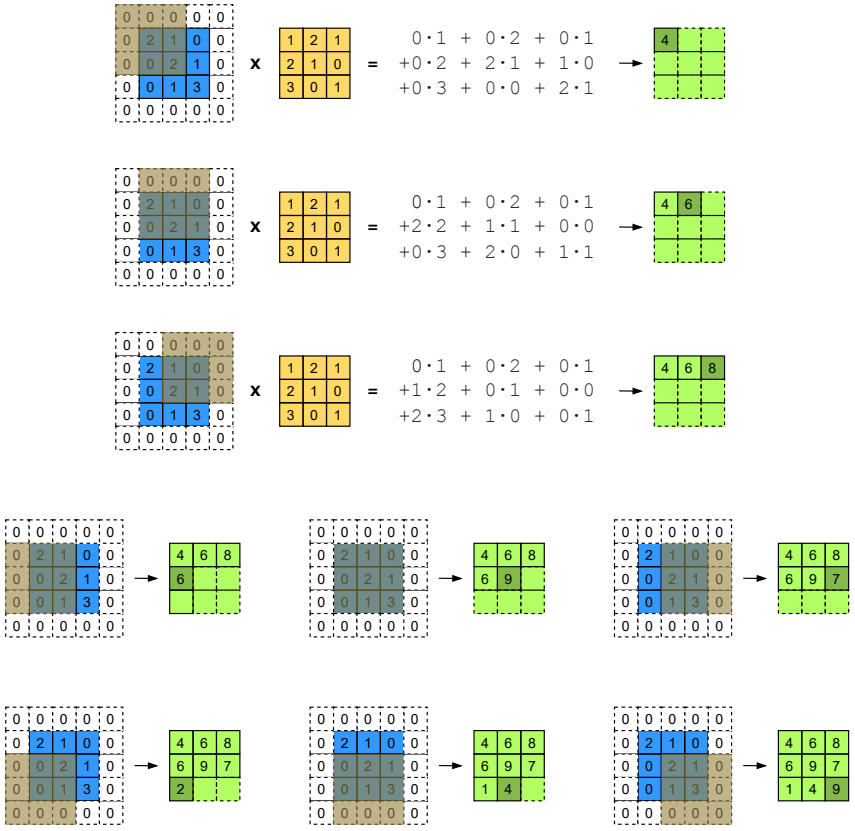
Source – From the author (2021)

It is simpler to understand this by looking at an image representation, Figure 8 shows an example of a kernel being applied at the corner of an image with a single channel, whose values are 255, and that is padded with zeros.

As seen in the figure, the kernel will start at the top left of the image (including some optional zero padding) and calculate a value for that position, then it will step a number of pixels, called the stride, and calculate the next value. When there is no more room to step horizontally, the kernel will start again at the left of the image and step a stride size downward, this is repeated until the whole image is traversed.

The convolution value between the kernel window and the pixels is simply the sum of the element wise products between the pixel values and the corresponding kernel weight. An image can be used again to help visualize this process, Figure 9 shows how a convolution is calculated for a 3×3 kernel, on a 3×3 image, with 1 pixel zero padding, and stride of 1 for both horizontal and vertical directions.

Figure 9 – Simple convolution process



Source – From the author (2021)

Also note that the kernel does not change when calculating the outputs of the convolution, in other words, the weights of the kernel are shared between the units. These weight are learned instead of being predefined, this allows for the network to decide which features the kernels should detect in order to solve the problem; these could be for example edges, textures, shapes, specific colors or others.

One important detail to mention is that the examples shown until now only consist of single channel images. For colored images with 3 channels, and more general cases with n channels, the kernel is not just a 2D matrix but is more accurately represented as a volume (i.e. one 2D kernel for each channel) . This means that in Figure 9 the kernel is a $3 \times 3 \times 1$ volume, if the input were a colored image it would be a $3 \times 3 \times 3$ volume. For the general case of an n channel input, the volume would be $h \times w \times n$ for a kernel with height h and width w.

The convolution for multiple channels is basically the same as for a single channel, each $h \times w$ slice of the volume is convoluted with one of the channels and the results are added together to get the final convolution. These resulting values can be considered as the weighted inputs $\boldsymbol{z}$ of the convolutional layer, so the activation function should be applied in order to get the final layer output.

In summary, for the case of images, a convolutional layer takes as input a 3D volume (i.e. two spatial dimensions to convolve with, and 1 channel dimension to give different views of the image) and operates on it using a kernel volume, the spatial sizes of the kernel are free to choose, but the number of channels must match. The convolution operation with a single kernel will reduce the input volume to a 2D feature map, where each point in the map indicates how much the feature is present at that region of the input covered by the kernel window. In general it is desirable to detect many different features from the input image, this means that convolutional layers will have multiple kernels and all the resulting convolutions will be stacked to produce an output volume.

The output will have as many channels as the number of kernels used in the convolutional layer, but the width and height will depend on the kernel size, stride, and padding. Consider sliding the kernel in the horizontal direction (the same logic applies to any direction and for all input dimensions), the number of values calculated by the convolution will be the number of possible positions that the kernel can be placed in this direction.

Suppose that in the horizontal direction the sizes of input, kernel, zero padding, and stride are i, k, p, and s respectively. Then the output size o in this direction is given by Equation (6) (DUMOULIN; VISIN, 2018), where $\lfloor . \rfloor$ is the floor function.

$$o = \left\lfloor \frac{i + 2p - k}{s} \right\rfloor + 1 \tag{6}$$

Convolutions can be used to enlarge the input image, but are most commonly used to reduce the dimensions while raising the number of features detected. For example, the GoogLeNet Inception architecture reduces the $224 \times 224 \times 3$ input image to a feature map of size $7 \times 7 \times 1024$ (SZEGEDY; LIU, et al., 2014). This reduction is usually not done using only a single convolutional layer, it is most common for the input volume to pass through multiple convolutions that gradually reduce its width and height while increasing its depth. Looking at the Inception model again, the whole network is 22 layers deep with

most of those being convolutions for feature extraction (SZEGEDY; LIU, et al., 2014).

According to Goodfellow, Bengio, and Courville (2016, chap. 9), convolutional layers leverage the ideas of sparse interactions, shared parameters, and equivariant representations to improve a machine learning system. They describe sparse interactions in the sense that kernels are smaller than the input, reducing the number of parameters, the memory used, the processing cost, and improving statistical efficiency. And the idea of equivariant representations is related with the invariance to translation, since at least in principle, the use of a sliding window allows for detecting the same feature no matter where it might appear on the image.

3.2.1.3 Transposed Convolutions

Any convolution operation can be converted to a matrix multiplication where the input is flattened to a 1-dimensional vector and the kernel is converted to a sparse matrix $\boldsymbol{C}$. The forward pass through the network, which is when the convolution is applied, is equivalent to multiplying the flattened input by $\boldsymbol{C}$, and the backward pass is equivalent with multiplying by $\boldsymbol{C}^T$ (DUMOULIN; VISIN, 2018). The multiplication with $\boldsymbol{C}^T$ converts the output volume to the input volume and is commonly called the transposed convolution, or sometimes *deconvolution*.

Any kernel represents both a convolution or transposed convolution, it all depends on how the values are interpreted as a matrix. It is important to note that the transposed convolution is not a way to reverse the convolution step, it is generally not possible to calculate the input of a convolution based on the kernel and output values. But the transpose operation can be considered as a reverse in the sense of transforming the output volume into the input volume.

Given the property of producing the opposite volume in relation to convolutions, transposed convolution layers are commonly used to upscale data to higher dimensions. For example, they are used in GANs to upscale the latent space vector into the corresponding image (seeChapter 4) (GOODFELLOW, 2017). However, upscaling with this method has been shown to produce image artifacts (ODENA; DUMOULIN; OLAH, 2016) and some models, like styleGAN (KARRAS; LAINE; AILA, 2018), already drop the use of transposed convolution layers for different upscaling approaches.

3.2.1.4 Pooling layer

Pooling layers are very similar to convolutional layers in the sense that they both slide an window of some width and height through the image, using some stride value, optional padding, and producing a number for each possible position of the window. But pooling layers do not have any learnable parameters and just execute a predefined function over all the inputs inside the sliding window to produce the output.

The two most common types of pooling are max and average pooling. Max pooling, as the name suggests, simply returns the maximum value present in the sliding window, while average pooling returns the average of those values. Pooling layers are very commonly used together with convolutional layers, Goodfellow, Bengio, and Courville (2016, p. 355-336) describe that a usual convolutional layer in a CNN consists of a convolution operation, followed by the nonlinear activation, and lastly by a pooling layer; they say that this operation helps to make the representation approximately invariant to input translations.

3.2.1.5 Embedding Layer

This layer is an alternative way of representing discrete valued inputs. Recall that one way to represent a discrete value (e.g. the class of a given input) is to use one-hot encoding, this allows for mapping n possible values into a n dimensional vector of all 0's and a single 1 representing the value.

Embedding layers offer a way to map n values into a vector with m dimensions, where m can be any number. The mapping from value to vector in an embedding layer is not something fixed, but is also learned during training. This type of layer is very useful in language models, where encoding thousands of possible words into one-hot vectors is infeasible, embeddings allow for much smaller representations that can be learned by the model to best fit a given problem.

When conditioning GANs with class information for the CGAN variant (see subsection 4.3.2), it is necessary to combine the label of the data together with the input. Embedding layers offer a more robust solution to this when compared to one-hot vectors, because of that they were used for conditioning in the experiments seen inChapter 5.

3.2.2 Activation Functions

Historically one of the first implementations of artificial neural networks used the step function as activation for the units (NIELSEN, 2015, Chapter 1), this means that for positive inputs the step function produces a 1 and for all other values it produces a 0. It also could use the sign function (SINAI, 2017), producing a -1 for non positive inputs.

This type of approach is called a Perceptron, one notable example of its implementation was the MARK I Perceptron, a hardware solution where all weights were regulated by potentiometers and automatically adjusted by motors to train the network (HAY; MURRAY, 1960). However it was later shown that this type of binary activation was very limited and could only solve linear separable problems (SINAI, 2017).

One may question the need for activation functions or why do they need to be nonlinear, what is the reason for introducing nonlinearity to the network? To understand this, first note that not using an activation function is the same as setting the output $y = x$, this is also a linear relationship between the values, so it is only necessary to explain why a linear activation is a problem.

The nonlinearity is introduced to make the network able to learn more complex relationships in the data, since not all real world situations have a linear dependence between input and output. By transforming the input through multiple nonlinear activations it is possible to create very elaborate mappings from input to output. This does not hold true for linear transformations, since applying a sequence of them to some input is equivalent to applying just a single one.

To see this consider an input vector $\boldsymbol{x}$ with two transformations matrices $\boldsymbol{A}_1$, $\boldsymbol{A}_2$ and vectors $\boldsymbol{b}_1$, $\boldsymbol{b}_2$. The output $\boldsymbol{y}_2$ is obtained by applying two linear transformations to the input vector as follows.

$$\boldsymbol{y}_1 = \boldsymbol{A}_1\boldsymbol{x} + \boldsymbol{b}_1 \qquad \text{and} \qquad \boldsymbol{y}_2 = \boldsymbol{A}_2\boldsymbol{y}_1 + \boldsymbol{b}_2 \tag{7}$$

By rearranging the terms it can be confirmed that the two linear transformations are equivalent to a single one, this can be seen by the following derivation.

$$\begin{aligned} \boldsymbol{y}_2 &= \boldsymbol{A}_2(\boldsymbol{A}_1\boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2 \\ &= \boldsymbol{A}_2\boldsymbol{A}_1\boldsymbol{x} + \boldsymbol{A}_2\boldsymbol{b}_1 + \boldsymbol{b}_2 \\ &= \boldsymbol{A}\boldsymbol{x} + \boldsymbol{b} \end{aligned}$$

This logic holds true for any number of linear transformations. Now it should be hopefully clear to see that there is inherently no difference between a multilayered neural network with linear activations and a simple two layered input-output network. It is necessary to introduce nonlinearities in the hidden layers in order to build more complex models – linear transformations can however be used in the output layer to map the values to a more desirable range, regression problems are an example.

Although nonlinear, the binary property of the step function limits the capabilities of a neural network, to work around this limitations it is necessary to introduce a different type of activation. To avoid the problem of jumping values it is best to have a continuous function instead of a discrete one, it is also important that this function be differentiable in its domain and that the derivatives are not zero everywhere (see ReLU activation in subsubsection 3.2.2.3 for more remarks about this restrictions). The derivative requirements are necessary to make possible the use of the learning algorithm Gradient Descent with Backpropagation (see sections 3.3 and 3.4).

This section will briefly introduce some important activation functions that are widely used in multiple machine learning problems and that were used in the experiments found inChapter 5.

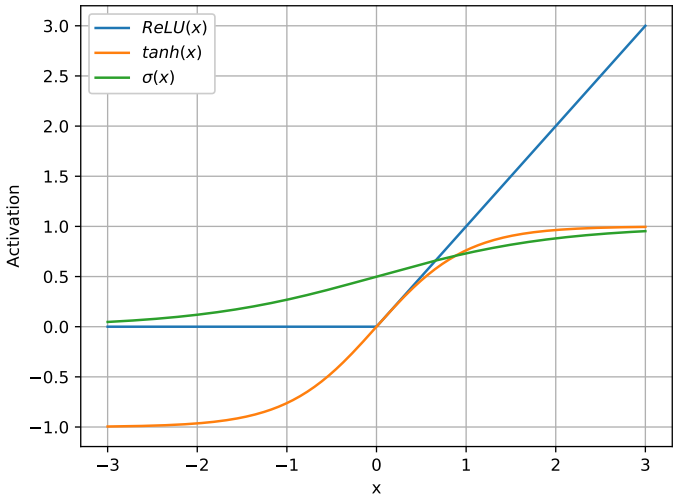
3.2.2.1 Sigmoid

Usually denoted by σ, the sigmoid function maps all real numbers to the interval $(0, 1)$. For a given input x, the value for $\sigma(x)$ is given by Equation (8).

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{8}$$

The sigmoid function can be considered the continuous version of the step function, as the absolute value of x grows, $\sigma(x)$ gets exponentially closer to $\mathsf{step}(x)$, but there is a continuous transition close to 0. This can be seen more clearly inFigure 10.

Figure 10 – Curves of sigmoid, tanh and ReLU activation functions



Source – From the author(2021)

This function can be useful to convert a single output to a probability value or to normalize a set of outputs. However, the fact that the outputs are always positive raises some problems, Yann A LeCun et al. (2012) showed that in these situations the backpropagation algorithm must update all the network parameters in the same direction, this means that the parameters are not free to wander the parameter space in the best direction.

Most of the time the network will benefit more by adding to some parameters while subtracting from others, the restriction to always update in the same direction makes learning more difficult and can greatly reduce the speed of convergence. Yann A LeCun et al. (2012) also argue that any deviation in the average of the outputs will bias the update direction, so it is better to have activations that are zero centered.

Knowing this problem, it only makes sense to consider sigmoid activations in the output layer, since for most cases it is better to use a zero centered alternative like the hyperbolic tangent (tanh) in the hidden layers.

3.2.2.2 Hyperbolic Tangent

The hyperbolic tangent function, as seen in Equation (9), is a zero centered, scaled, and shifted version of the sigmoid function.

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 2\sigma(2x) - 1 \tag{9}$$

This function is almost always better than sigmoid since it has the same shape and properties without having the disadvantage of not being zero centered. It can be used in any layer, including the output. The times were a sigmoid would be preferred are when it is desired that the output be bounded to $(0, 1)$, as is the case for a probability value.

3.2.2.3 Rectified Linear Unit

Both the sigmoid and hyperbolic tangent functions have two characteristics that can raise some problems when training a machine learning model. The first one is that they can only produce very close approximations of the number zero, but not the exact value – the only exception would be when the input of the tanh function is also exactly zero, but this is very unlikely to happen and would only work for very specific inputs.

The lack of a true zero can be undesirable when the goal of the network is to build a sparse representation of the data, that is, a representation that only depends on a small number of inputs that strongly correlate to the output. This is in contrast with a model that depends on many inputs, but most of them have very little impact on the output.

The sparsity argument not only has some biological support, but has been shown to also positively influence the quality of a model (GLOROT; BORDES; BENGIO, 2011). Promoting sparsity is found in many other areas of science (e.g. statistical modeling, image compression) and it is very useful for producing simpler representations of the data (BRUNTON; KUTZ, 2019).

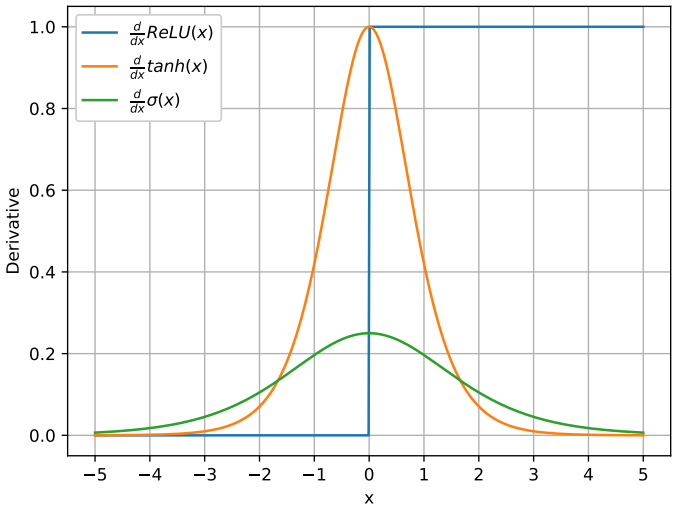
The other problem present in the sigmoid and tanh functions is that they saturate for high absolute values of the input. Most of the variation in these functions occur close to zero, but for larger values there is barely any difference between outputs. For example, the difference between the tanh activation between inputs 1 and 2 is around 0.2, while the difference from 2 to 100 is just 0.036. One other way to say this is that the derivatives of these functions are very close to zero for inputs far from the origin (see Figure 11), this can give rise to the vanishing gradients problem and make learning extremely slow (see subsection 3.5.5).

The Rectified Linear Unit (ReLU) activation function is an alternative that addresses both of the mentioned problems with sigmoid and tanh. It is constructed from combining different linear regions (called a piecewise linear function), this allows for the activation to inherit many desirable properties of linear transformations while still retaining nonlinearity and being able to build complex relations (GOODFELLOW; BENGIO; COURVILLE, 2016).

The ReLU function is simply composed of a constant zero for all negative inputs and the identity function for everything else, this means that for an input x the activation is calculated as shown in Equation (10).

$$\mathrm{ReLU}(x) = \max(0, x) \tag{10}$$

Figure 11 – Derivative of sigmoid, tanh and ReLU activation functions



Source – From the author(2021)

This definition allows for ReLU to produce exact zeros for any negative inputs, promoting sparsity in the model representations. The constant positive derivative (see Figure 11) also removes the problem of vanishing gradients, since the units never saturate. Another big advantage of ReLU is that it is extremely easy to compute, a simple `if` statement is enough to get the result; compared with the need to calculate exponentials in the sigmoid and tanh functions, ReLU performance is much faster.

Since around 2010, with papers like (GLOROT; BORDES; BENGIO, 2011) exploring the ReLU activation, there were many popular methods that showed impressive results using this function (e.g. (KRIZHEVSKY; SUTSKEVER; HINTON, 2012) and (SZEGEDY; VANHOUCKE, et al., 2015)). At the current time ReLU is the standard recommended activation function to be used in most problems (GOODFELLOW; BENGIO; COURVILLE, 2016).

There are however some downsides to this activation. First there is the sharp change in the function behavior at the value 0, making its derivative undefined at that point, this however does not seem to be a problem in practice (GLOROT; BORDES; BENGIO, 2011). ReLU also loses the desirable zero centered property that made tanh a good substitute for the sigmoid, the argument from Yann A LeCun et al. (2012) holds for all activations, this means that all the network updates for units that use ReLU must be made in the same direction, making learning more difficult.

Lastly, for a unit to be able to learn it is necessary that it outputs a value in the range where the activation function has a non-zero derivative for at least one example in the training data. But it is possible that for all the training data in a given problem, there exists some units using ReLU activations that will always output zero, this makes learning impossible in these units and effectively freezes them on their state forever.

There are many other alternatives proposed to address these problems, some relevant examples are Maxout, LeakyReLU, ELU, GELU and PReLU, these have found some success in big machine learning problems[7] and the benchmark by (MISHKIN; SERGIEVSKIY; MATAS, 2017) showed good results for some of these alternatives – although ReLU also fared well in those tests.

Between these alternatives, LeakyReLU is specially relevant for GANs, being an important piece of the DCGAN variant that is the base of many GAN architectures. It consists of a simple change from ReLU that just guarantees that the units will always have at least some positive derivative. This activation is calculated as shown in Equation (11), where α is a hyperparameter that must be a small value (usually 0.3).

$$\text{LeakyReLU}(x) = \max(\alpha x, x) \tag{11}$$

3.2.2.4 Softmax

In classification problems it is very common to have the output layer encode the input class has a n dimensional, one-hot encoded vector, where n is the number of classes. Since in one-hot encoding each element corresponds to the probability of the input belonging to the corresponding class (all zeros and a single 1 for the correct class), it is desirable for the output of a classification model to also be a probability distribution. Besides the fact that it aligns with the encoding representation, it also gives a meaningful representation for the output of the network, by observing the output probabilities it is possible to have an idea of how confident the network is in the results[8].

The softmax activation function offers a way to map all the units weighted inputs to a probability distribution. One difference when compared to the other activation functions is that the softmax does not take a single number as input, instead it uses all values in the layer for calculating the unit activation. This makes sense since the probabilities are dependent on the proportions of each unit weighted input. Equation (12) shows how the activation for the unit i is calculated based on the layer weighted inputs ($z_0 \dots z_n$), the layer superscripts were omitted for clarity.

$$a_i = \frac{e^{z_i}}{\sum_{k=0}^{n} e^{z_k}} \tag{12}$$

3.3 LOSS FUNCTION AND GRADIENT DESCENT

So far only the concept of what a neural network is was explained, but not how it can learn from data, to understand this it is worth to have first a brief summary of what

[7] GELU is used in natural language processing models like GPT-3 (BROWN et al., 2020)

[8] Since the values in a probability distribution should always add to 100% this may not always give good representations, this is the case for adversarial examples that can fool the network to misclassify images with a high degree of confidence (SZEGEDY; ZAREMBA, et al., 2013).

consists a neural network.

A collection of units having nonlinear activation functions form a layer; layers are stacked from the input to the output with some optional hidden layers in-between; the weighted input of a unit is calculated by multiplying its inputs with some weight parameters and adding a bias term; the unit activation is the output of its activation function applied to the weighted input; and by varying the values of the weights and biases it is possible to reach different model behaviours – this section will only consider the case when the inputs of a unit come from the immediate previous layer and the output is calculated by passing the input values from layer to layer until the last layer (this is called a *feedforward* neural network), other configurations like Residual Networkss (ResNets) (HE et al., 2015a) and Recurrent Neural Networks (RNNs) also exist, but it is possible to extend the arguments present here to also explain learning in these contexts.

For all neural networks, learning is the process of changing the weights and biases (and possibly other values) in order to produce the desired result. The set of values that are learned during training are called the *parameters* of the network.

The number of parameters gives the degree of freedom that the network has, in a larger parameter space the network has more possible choices and can construct more complex relations. On the other hand, fewer parameters may make it difficult or impossible to represent the full complexity of the data. However, besides requiring more computational power, very complex models break the idea of sparsity and can more easily fall into the problem of overfitting (see subsection 3.5.1).

The number of parameters in the network is given by its architecture, how many layers it has, how many units in each layer, how the connections are made, and so on. This is a choice made when building the network and is not something that is learned by the algorithm. The set of properties that influence the network but that are not learned are usually called the *hyperparameters*.

Another example of hyperparameter is the activation functions used in the units, they are chosen when constructing the neural network and can't be learned – however some activations like PReLU have internal learnable parameters (HE et al., 2015b).

Learning is then the process of finding optimal parameters for a network given some hyperparameters. But how can a machine automatically learn the parameters? To do this the network must have some way to quantify the results that it produces, a way to distinguish between good and bad outputs, this is what is called the *loss function*.

Also called the cost or objective, the loss function $J(\boldsymbol{x}, \boldsymbol{\theta})$ is a function that returns how good the output of the network is, given an input $\boldsymbol{x}$ and a set of network parameters $\boldsymbol{\theta}$ (represents all trainable parameters: weights, biases, and any others). Learning can then be described as the process of adapting the parameters $\boldsymbol{\theta}$ in order to minimize or maximize the objective function (usually minimize, depends on the function used). The choice of function will depend on the type of problem, but two of the most common methods are

Mean Squared Error (MSE) and Categorical Cross Entropy.

3.3.1 Mean Squared Error

This function is the standard squared euclidean distance between two vectors. For a given input $\boldsymbol{x}$, the network output $\hat{\boldsymbol{y}}$ for this input, and the true desired output $\boldsymbol{y}$ (e.g. the corresponding label), the squared distance is calculated as shown in Equation (13) – note that the subscript 2 indicates the euclidean distance (the ℓ_2 norm).

$$\|\boldsymbol{y} - \hat{\boldsymbol{y}}\|_2^2 = \sum_i (y_j - \hat{y}_i)^2 \tag{13}$$

This distance can also be generalize to matrices or any n dimensional values of $\boldsymbol{y}$ by just calculating the element-wise squared differences between $\boldsymbol{y}$ and $\hat{\boldsymbol{y}}$.

However, simply calculating the distance for one possible input is not good for evaluating how well the network is doing in general. To better evaluate the network's performance it is necessary to see how well it is doing in the entire dataset or some subset of it. For a set $\boldsymbol{X}$ of n inputs, the MSE is the mean of the squared distances as shown in Equation (14).

$$J(\boldsymbol{X}, \boldsymbol{\theta}) = \frac{1}{n} \sum^{n} ||\boldsymbol{y} - \hat{\boldsymbol{y}}||_2^2 = \frac{1}{n} \sum^{n} \sum_i (y_i - \hat{y}_i)^2 \tag{14}$$

This loss function is usually used for regression problems, where the output can have a range of values and is not simply defined as a 0 or a 1. It is also useful to calculate the differences between images in pixel space, like used for autoencoders (KRAMER, 1991).

3.3.2 Categorical and Binary Cross Entropy

Like MSE this loss function is calculated by averaging the error over many different inputs, but in this case the error calculated is the cross entropy. Given two discrete probability distributions $\boldsymbol{y}$ and $\hat{\boldsymbol{y}}$, the cross entropy is calculated as shown in Equation (15).

$$H(\boldsymbol{y}, \hat{\boldsymbol{y}}) = -\sum_i y_i \log(\hat{y}_i + \epsilon) \tag{15}$$

The ϵ value in this equation is not a part of the cross entropy definition, but when using computers to calculate the logarithm, numbers very close or equal to zero can introduce numerical instabilities. For this reason, in almost all practical implementations it is always added a small constant ϵ when calculating the cross entropy and other functions that can have similar issues – in Tensorflow (ABADI et al., 2015) the default value for ϵ is equal to 10^{-7}.

The cross entropy applied over a set of inputs gives the categorical cross entropy loss as shown in Equation (16).

$$J(\boldsymbol{X}, \boldsymbol{\theta}) = \frac{1}{n} \sum^{n} H(\boldsymbol{y}, \hat{\boldsymbol{y}}) \tag{16}$$

This kind of loss function is useful when dealing with probability distributions, since the concept of entropy is deeply related with information and probability. This makes it a better alternative for classification problems when compared with MSE, since the desired output is the class that the input belongs to (one hot encoded) and the network output is a probability distribution over all possible classes.

But for cases where the output can only assume two values (0 or 1), the categorical cross entropy is reduced to a binary cross entropy. For GANs, this is the more useful loss function and it is calculated as shown in Equation (17).

$$J(\boldsymbol{X}, \boldsymbol{\theta}) = -\frac{1}{n} \sum^{n} \sum_{i} \left(y_i \log(\hat{y}_i + \epsilon) + (1 - y_i) \log(1 - \hat{y}_i + \epsilon) \right) \tag{17}$$

3.3.3 Minimizing the loss

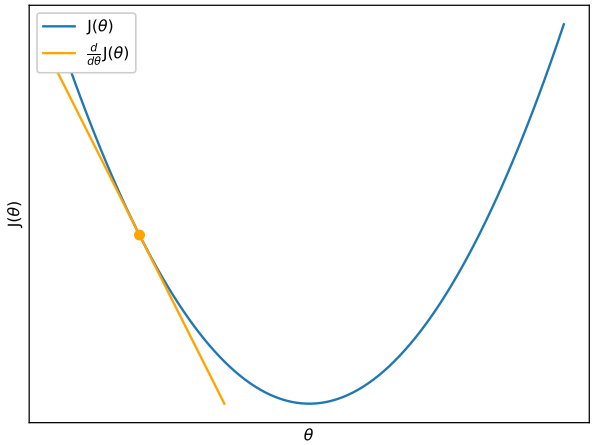
Having an understanding of what is the loss function, the remaining question is: how can the network use the loss in order to update its parameters? Note that since this function effectively measures how well the network is doing, it can directly be used to guide the parameter changes, the goal is to change the parameters in order to minimize the loss (for now on the goal will only be described as minimization, since it is the most common approach and any maximization problem is equivalent to minimizing the negative of what is being maximized).

Note that the inputs $\boldsymbol{x}$ are fixed, given that the dataset is also fixed, and that the goal of using neural networks is that they should learn to model the data and not just choose whatever inputs reduce their loss. So in the eyes of the network $J(\boldsymbol{x}, \boldsymbol{\theta})$ becomes only $J(\boldsymbol{\theta})$. This may seem obvious, but this shows an important intuition that the cost function is a high dimensional surface in N dimensional space, N being the number of parameters of the network. And this demonstrates the importance of having both the network and loss functions be continuous and differentiable functions, since this produces a continuous and differentiable surface that allows for updating the parameters in order to reach a minimum region.

To see how the minimum is reached, it is better to start in a very simplified case where the network has only one parameter θ. Consider a case where the 1-dimensional loss surface assumes the form shown in Figure 12.

Consider that the network initially starts with the parameter θ_0 and loss J_0 at the dot shown in Figure 12. To reduce the loss is to update the parameter θ in the opposite

Figure 12 – Example of a one dimensional loss surface



Source – From the author (2021)

direction of the surface derivative. This can be shown by the following derivation.

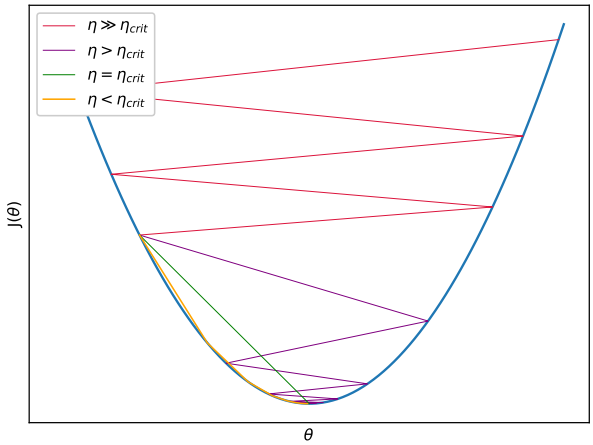
$$\begin{aligned} J_0 + \Delta J &< J_0 \\ J_0 + \Delta\theta \frac{dJ}{d\theta} &< J_0 \\ \Delta\theta \frac{dJ}{d\theta} &< 0 \\ \operatorname{sign}(\Delta\theta) &= -\operatorname{sign}\left(\frac{dJ}{d\theta}\right) \end{aligned} \tag{18}$$

This derivation above only holds true for small values of $\Delta\theta$, since the derivative is calculated on infinitesimal small values, bigger changes run the risk of overshooting the region where the derivative approximation is reasonable. Equation (18) shows that the parameter change must be in the opposite direction of the change in the loss, the parameter updates can then be written as shown in Equation (19).

$$\theta_{i+1} = \theta_i + \Delta\theta = \theta_i - \eta \frac{dJ}{d\theta} \tag{19}$$

The value η is a positive real number that determines how big are the steps taken when updating the parameters. This value is also a hyperparameter and is called the *learning rate* of the network. A small learning rate is more precise but can make the training very slow, while bigger values are faster, but run the risk of overshooting and zigzaging around the target, or even worse, diverging and never reaching the result. Figure 13 shows how an initial parameter value is updated when using different values for η, it can be seen that there is a critical value η_{crit} that is able to minimize the loss in a single step, while smaller and bigger values will converge to the minimum in multiple steps, and much bigger values will diverge.

Figure 13 – Parameter updates for 1D loss surface in function of learning rate



Source – From the author (2021)

For one dimension it is relatively simple to find the critical learning rate, but for very high dimensional problems this is unfeasible or even impossible since a critical value for one parameter is not necessarily the same for all other parameters.

In practice, it is usually necessary to experiment with some values in order to find what best suits the loss surface of the problem, it is also common to employ some sort of dynamic update that slowly reduces the learning rate, allowing for big steps in the beginning while being more precise towards the end.

3.3.4 Gradient Descent

The same derivation used to obtain Equation (18) can be applied to generalize the minimization procedure to higher dimensional parameter spaces, the only difference is to consider the partial derivatives for each parameter. Doing this would reveal that, like in the 1-dimensional case, the change in a parameter $\theta^{(j)}$ must be in the opposite direction of the partial derivative of the loss with respect to this parameter. The parameter update would then be given by Equation (20).

$$\theta_{i+1}^{(j)} = \theta_i^{(j)} - \eta \frac{\partial J}{\partial \theta^{(j)}} \tag{20}$$

There is however another way to look at these updates. Recall that the gradient of any scalar function is a vector field composed of all its partial derivatives, so the gradient of the loss function is the vector field given by Equation (21).

$$\nabla_{\boldsymbol{\theta}} J = \begin{bmatrix} \frac{\partial J}{\partial \theta^{(1)}} \\ \frac{\partial J}{\partial \theta^{(2)}} \\ \vdots \\ \frac{\partial J}{\partial \theta^{(n)}} \end{bmatrix} \tag{21}$$

Also recall that for any function f, its gradient at any given point is a vector that points towards the direction of steepest increase to the function at that point and the negative of this vector points to the steepest decrease (STRANG; HERMAN, 2016, Chapter 4.6). So the negative of the gradient $\nabla_{\boldsymbol{\theta}} J$ gives the direction to update all parameters in order to most reduce the loss function, this is what was being shown element-wise in Equation (20). By grouping the derivatives it is possible to write the parameter updates in a much cleaner way.

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_i) \tag{22}$$

By repeating the process of calculating the gradient of the loss function and then updating all the network's parameters (given a sufficiently small learning rate), then eventually all parameters will converge to values that minimize the loss function. This is what is referred to as *learning* in a neural network and this iterative process is called *Gradient Descent*, since the steps are taken using the gradient in order to decrease the loss function – for maximization, the only difference is to update in the positive direction of the gradient and this is called *Gradient Ascent*.

One important thing that was left unmentioned is the fact that this algorithm will almost surely converge to a local minimum in the loss surface instead of the global minimum. In the example shown for the 1-dimensional case the surface was very simple, with a single minimum value to converge, but even in these very simple cases it is possible to find surfaces much more complex (e.g. the curve $y = x + 2\sin x$ has infinitely many local minima but no global minimum). For the very high dimensional spaces and very complex geometries of the loss surfaces in neural networks, there will be many valleys where gradient descent will converge, but rarely those will be the optimal solution. The local minimum found will depend on the starting point in parameter space, and there is usually no better to way than just randomly selecting a point[9].

So if gradient descent will almost never reach the optimal solution, why would anyone use it? How can someone make sure that the local minimum found is good enough for the problem? Or if someone is trying to build a neural network and the results are not being good, how can this person know if this is caused by a legitimate problem on the network or the dataset, instead of simply being the fact that the training process was unlucky and found a bad local minimum? And perhaps most important, how is it possible that neural networks are achieving so much ground-breaking results while relying on gradient descent to learn?

This is indeed a worry that many researchers in the field have, but there are many options to combat this problem, it can be cited regularization techniques (e.g. ℓ_1 and ℓ_2

[9] This does not mean that the starting values are picked without any rhyme or reason, the starting point can have a huge impact on the network and it is very important to choose good values. However, initialization techniques are concerned with finding a random distribution with the right values of mean and variance to sample from. The point here is explicitly about the random nature of the starting point and not that this nature has no thought behind it.

norms – see subsection 3.5.2) and optimizers (e.g. SGD, momentum, RMSProp, Adam – see subsection 3.5.3) as examples.

Besides that, Choromanska et al. (2014) argue that global minimum are not necessarily the best option, since they have a high chance of overfitting to the data (see subsection 3.5.1), they also empirically verify that for big enough neural networks most local minima are very similar to one another and have high quality. Their conclusion is that in practice it is not worth to strive for the global minimum, given that for large networks local minima have high quality and may even generalize better to unseen data.

3.4 BACKPROPAGATION

Last section showed how learning in a neural network is a minimization problem on the loss function and that it is solved by repeatedly updating the network's parameters using the gradient of the loss. But how exactly is the gradient calculated? The loss function is a surface in very high dimensions, calculated by averaging some distance function between the network's output and the true output for all inputs on the dataset; and the output of the network is a mapping calculated by passing the input through possible thousands, millions, or more units, where each one can apply a nonlinearity to its output. In summary, the loss surface is extremely complex and the same should be expected for its gradient.

To make it simpler to understand how the gradient is calculated, the procedure will be shown only for the weights and biases parameters, since those are present in practically all neural networks (although sometimes the bias is omitted in some units). This section will suppose a neural network with $L + 1$ layers, where layer 0 is the input, layer L is the output and values in between are hidden layers.

Essentially, calculating the gradients consists of a smart application of the chain rule of calculus. Recall that the chain rule is a way of calculating the derivative of composite functions, this means that for a function y that depends on t, and t that depends on x, then the chain rule allows for calculating the derivative of y with respect to x by compounding how x changes t and how t changes y. For the case of single variable functions the chain rule is given by Equation (23) (STRANG; HERMAN, 2016, p. 406).

$$\frac{dy}{dx} = \frac{dy}{dt}\frac{dt}{dx} \tag{23}$$

But for the case of neural networks, the loss function is dependent on all the activations of the output layer, and those are dependent on their weights, biases, and possibly other parameters, besides being dependent on activations of the previous layer. So it is necessary to use the multi-variable generalization of the chain rule shown in Equation (24) (STRANG; HERMAN, 2016, p. 412), here it is necessary to compound the

effect of x for all the variables $(t_0, t_1, \dots t_n)$ that y is dependent on.

$$\frac{\partial y}{\partial x} = \sum_{i}^{n} \frac{\partial y}{\partial t_i} \frac{\partial t_i}{\partial x} \tag{24}$$

Having the chain rule in mind, it is also useful to define an additional term δ that represents the partial derivative of the loss with respect to the weighted input of a unit. For unit i on layer l this term is given by Equation (25).

$$\delta_i^{(l)} = \frac{\partial J}{\partial z_i^{(l)}} \tag{25}$$

And lastly, note that by differentiating Equations 3 and 4, the following relations are obtained.

$$\frac{\partial z_i^{(l+1)}}{\partial a_i^{(l)}} = w_{ij}^{(l+1)} \qquad \frac{\partial z_i^{(l)}}{\partial b_i^{(l)}} = 1 \qquad \frac{\partial z_i^{(l)}}{\partial w_{ij}^{(l)}} = a_j^{(l-1)}$$

The main idea of this algorithm is to derive δ for all layers, then use these values to calculate the gradient terms for all parameters, the first step is to calculate δ in the last layer. For the following derivation, consider $\hat{\boldsymbol{y}}$ as the network output, and notice that it is the same as the activations of the output layer $\boldsymbol{a}^{(L)}$. By using this knowledge, Equation (26) is derived as follows.

$$\begin{aligned}
\frac{\partial J}{\partial z_i^{(L)}} = \delta_i^{(L)} &= \frac{\partial J}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial z_i^{(L)}} \\
&= \frac{\partial J}{\partial \hat{y}_i} \frac{\partial a_i^{(L)}}{\partial z_i^{(L)}} \\
&= \frac{\partial J}{\partial \hat{y}_i} \frac{\partial}{\partial z_i^{(L)}} f\left(z_i^{(L)}\right) \\
\delta_i^{(L)} &= \frac{\partial J}{\partial \hat{y}_i} f'\left(z_i^{(L)}\right)
\end{aligned} \tag{26}$$

Recall that the loss function J and activation function f should both be continuous and differentiable, and since they are chosen when building the network their derivatives are known. The values $\hat{y}_i$ and $z_i^{(L)}$ are also known since they are calculated by the network and can be easily stored during training. This means that Equation (26) can be used to calculate all δ values in the last layer.

By using Equation (26) it is also possible to find an expression to calculate all the other δ values, the following derivation shows how this can be done by writing $\delta^{(l)}$ in terms

of $\delta^{(l+1)}$ as shown in Equation (27).

$$\begin{aligned}
\frac{\partial J}{\partial z_j^{(l)}} = \delta_j^{(l)} &= \sum_i \frac{\partial J}{\partial z_i^{(l+1)}} \frac{\partial z_i^{(l+1)}}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \\
&= \sum_i \delta_i^{(l+1)} \frac{\partial z_i^{(l+1)}}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \\
&= \sum_i \delta_i^{(l+1)} w_{ij}^{(l+1)} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \\
\delta_j^{(l)} &= \sum_i \delta_i^{(l+1)} w_{ij}^{(l+1)} f'\left(z_j^{(l)}\right)
\end{aligned} \tag{27}$$

Notice how the algorithm works, first the input is feedforwarded through the network to obtain the output $\hat{\boldsymbol{y}}$, this value is used to calculate $\delta^{(L)}$, that is then *backpropagated* through the network in order to calculate $\delta^{(l)}$ for all previous layers. This process gives the name *Backpropagation* to the algorithm.

Now for calculating the gradients using δ. Notice that for this case, where only the weights and biases are being considered, the gradient depends on the change ∂J with respect to $\partial b_i^{(l)}$ and $\partial w_{ij}^{(l)}$ for all units and layers.

The following derivation applies the chain rule to obtain the relation in Equation (28) for the partial derivatives of the loss with respect to all the biases parameters.

$$\begin{aligned}
\frac{\partial J}{\partial b_i^{(l)}} &= \sum_k \frac{\partial J}{\partial z_k^{(l)}} \frac{\partial z_k^{(l)}}{\partial b_i^{(l)}} \\
&= \frac{\partial J}{\partial z_i^{(l)}} \frac{\partial z_i^{(l)}}{\partial b_i^{(l)}} + \sum_{k \neq i} \frac{\partial J}{\partial z_k^{(l)}} \frac{\partial z_k^{(l)}}{\partial b_i^{(l)}} \overset{0}{\nearrow} \\
&= \frac{\partial J}{\partial z_i^{(l)}} \\
\frac{\partial J}{\partial b_i^{(l)}} &= \delta_i^{(l)}
\end{aligned} \tag{28}$$

The derivation for Equation (28) first breaks the partial derivative of the cost in terms of the weighted inputs $\boldsymbol{z}^{(l)}$ in the layer where the bias is present. This is enough since the bias can not influence any previous layers and all influences in the next layers are already captured in the change ∂J with respect to the weighted inputs $\partial z_k^{(l)}$. Since it is also known that the bias does not influence any other unit in the layer, the derivation could have been made directly without breaking the derivative into a sum of all the terms in the layer, but the whole process was shown here for completion sake.

A similar rationale can be used for the weight parameters, obtaining the relation seen in Equation (29).

$$\begin{aligned}
\frac{\partial J}{\partial w_{ij}^{(l)}} &= \sum_k \frac{\partial J}{\partial z_k^{(l)}} \frac{\partial z_k^{(l)}}{\partial w_{ij}^{(l)}} \\
&= \frac{\partial J}{\partial z_i^{(l)}} \frac{\partial z_i^{(l)}}{\partial w_{ij}^{(l)}} + \sum_{k \neq i} \frac{\partial J}{\partial z_k^{(l)}} \cancelto{0}{\frac{\partial z_k^{(l)}}{\partial w_{ij}^{(l)}}} \\
&= \frac{\partial J}{\partial z_i^{(l)}} a_j^{(l-1)} \\
\frac{\partial J}{\partial w_{ij}^{(l)}} &= \delta_i^{(l)} a_j^{(l-1)}
\end{aligned} \tag{29}$$

Equation (29) was the last piece of the puzzle, together with Equations 26, 27, and 28, it can be applied to calculate the gradient for all parameters in the network, and this allows for gradient descent to update the parameters and minimize the loss function. From data to model, a complete procedure for a machine to learn by itself.

There are still some more concepts that will be briefly explored in the next section. One further detail to mention about backpropagation is the fact that the derivations in this section were only made for the weights and biases parameters, what about possible others? There are many different types of additional parameters, but the idea with backpropagation is that the δ values are already calculated for all the units in the network, any new parameter θ must simply have a correlation with some of these values in order to obtain ∂J in terms of $\partial\theta$ and apply gradient descent for updates. Modern libraries and frameworks already abstract most of these calculations for the programmer via underlying procedures of automatic differentiation, for example, Tensorflow (ABADI et al., 2015) provides a `GradientTape` object to automatically watch and calculate the gradients for any desired parameter.

3.5 OTHER CONCEPTS

This section will briefly describe some other concepts that are relevant to this document but not warrant a thorough explanation.

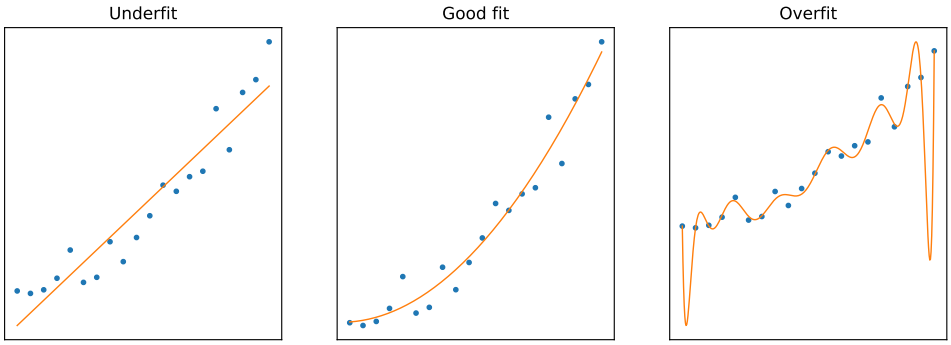
3.5.1 Overfitting

One fundamental principle, not only for neural networks but many other machine learning techniques, is the idea of generalization. An algorithm that learned by training on some given data should be able to produce good results not only on this seen data, but also on similar and never before seen data. In the case of supervising learning problems for example, all the data is already labeled, there is no need to create a complex algorithm

that can find these labels since the answer is already know. When teaching machines to think, the goal is not to make them memorize all the inputs and outputs, it is instead to make them learn how to abstract the data, to capture the fundamental patterns, and to interpret the input in a high level (e.g. seeing shapes, and objects instead of just pixels).

Since in the majority of cases the data available for training is only a fraction of the possible values expected for the problem, a machine learning algorithm must be able to use this relatively small amount of data to project how all the input space is distributed. This is as much a task for the algorithm as is for the dataset itself, the training data should contain enough samples to offer an accurate representation of how the input space is truly distributed. Ideally this data should be diverse enough to represent well the possible inputs (e.g. when training to differentiate between cats and dogs, the data should contain photos of both cats and dogs of multiple colors, in various angles, and with different poses) and it also should contain as much samples as possible[10].

Figure 14 – Visual representation of a learning algorithm fit to noisy data



Source – From the author (2021)

When the dataset is insufficient or the learning algorithm is not capable enough, the final model will only partially represent the data, but will not be able to capture all the details of the representation; this is called *Underfitting*, the model was not able to fully represent the structure of the problem. In the other hand, the algorithm can also go to the opposite extreme and learn the dataset too well, this means that it will pick up particularities of the training data that don't generalize to the whole input distribution; this is called *Overfitting* and it is facilitated by small datasets that don't represent much variety (noise in a sample has more influence in the signal to noise ratio of the whole

[10] Although having too much representation for one region of the input space in relation with the others can introduce bias in the model. For example, (BUOLAMWINI; GEBRU, 2018) found that some gender classification systems had a significant higher error rates for dark skinned women (as high as 30% higher than for white males) and that some facial analysis datasets underrepresented minorities in the samples. A similar problem happened to amazon, where an AI recruiting system was discriminating women for jobs since most of the resumes that it was trained on came from men (DASTIN, 2018). One proposed approach to combat this problem is to train another network to learn how the data is distributed and sample more from underrepresented data (AMINI et al., 2019).

dataset), or by networks that have too many degrees of freedom. Figure 14 visually represents the different fits for a 1-dimensional case.

Overfitting is however so common that it is almost natural to expect its appearance, it is one of the most prevalent problems in machine learning and there are many existing techniques to avoid it. The first step is to have a good quality dataset, data augmentation methods can be used to easily multiply the number of samples without much downsides. In the algorithmic side, next subsection will discuss regularization and how it can combat overfitting among other things.

3.5.2 Regularization

According to (GOODFELLOW; BENGIO; COURVILLE, 2016, p. 117): "*Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error*". This is a general definition that encompasses many different techniques for regularization, but for this document it will suffice to describe only dropout and the ℓ_1 and ℓ_2 norms.

3.5.2.1 L1 and L2 norms

Since one of the reasons overfitting happens is because the models are given a large number of free parameters to adjust to the dataset (worth mentioning again the 175 billion parameters present in GPT-3 (BROWN et al., 2020)), then a common way of reducing overfitting is to limit the choice of parameters by adding a penalization term $\Omega(\boldsymbol{\theta})$ to the loss function as shown in Equation (30) (GOODFELLOW; BENGIO; COURVILLE, 2016, p. 226).

$$\tilde{J}(\boldsymbol{X}, \boldsymbol{\theta}) = J(\boldsymbol{X}, \boldsymbol{\theta}) + \lambda\Omega(\boldsymbol{\theta}) \tag{30}$$

The λ term is a positive hyperparameter that determines how much the penalty is relevant to the overall loss. For ℓ_1 and ℓ_2 norms, $\Omega(\boldsymbol{\theta})$ will be the respective norm of the $\boldsymbol{\theta}$ parameters. The ℓ_2 norm is the standard distance in euclidean space, so for this norm the penalty term is given by Equation (31).

$$\ell_2(\boldsymbol{\theta}) = \|\boldsymbol{\theta}\|_2 = \sqrt{\theta_0^2 + \theta_1^2 + \cdots + \theta_n^2} \tag{31}$$

For the ℓ_1 norm, the value calculated is what is called the Manhattan distance or taxicab distance (BRUNTON; KUTZ, 2019, p. 102) and is calculated as shown is Equation (32)

$$\ell_1(\boldsymbol{\theta}) = \|\boldsymbol{\theta}\|_1 = |\theta_0| + |\theta_1| + \cdots + |\theta_n| \tag{32}$$

Both norms will penalize large parameter values and the modified loss function in Equation (30) will make gradient descent favor solutions that use smaller parameters. The λ hyperparameter then determines how much to favor these simpler solutions.

A simple explanation on why smaller values would reduce overfitting can be given by considering the case of fitting a polynomial to some data. Consider the overfitting case seen before in Figure 14, fitting a polynomial to reduce the square error on that noisy data results in a reasonable good approximation around the data points, but the curve suddenly changes directions in the extremes because of the terms with the highest powers in the polynomial. To make the curve fit the noisy data, a least squares approach will usually produce very high coefficients to balance the higher powers, this reduces the error but will almost never produce the true behaviour behind the data. Penalizing high coefficients with regularization is a way to force the solution to assume simpler values.

This is in no way a rigorous argument for regularization, but it can be used to have an intuition. The goal of this section is just to introduce the concept, regularization is an active area of research and a lot of support for it is based on empirical evidence, there is no complete theory to explain how and why it works so well (NIELSEN, 2015, chap. 3).

One last thing to mention is the fact that ℓ_1 norm promotes sparsity in the parameters, this means that this approach will usually find solutions where multiple parameters will equal zero, larger values of λ will promote more sparse models.

3.5.2.2 Dropout

Since the local minimum reached by a training process will depend on the starting values for the parameters, one common way to increase the accuracy of a model is create an ensemble of different networks trained in the same dataset but with different starting points and/or architectures (NIELSEN, 2015, Chapter 6). The model output can then be a combination of the outputs of the ensemble, the combination could be majority voting, an average between outputs, or any other sensible function. This gives better results because it is not expected that randomly initialized networks would make the same mistakes in all situations (GOODFELLOW; BENGIO; COURVILLE, 2016, p. 253).

Dropout is a technique that can be interpreted as a way to approximate an exponentially large ensemble of networks while using a single network and without needing to train more models (HINTON et al., 2012). The idea behind it is that during training, when feedforwarding the input through the network, some units should have a probability p of being dropped from the calculation, that is, their activations are set to zero.

The idea of removing units during calculation may sound counter-intuitive, but like the ℓ norms it is a way of restricting the learning process so that the network can't build very complex connections. By removing random units in each iteration, the network must be able to learn to represent the data even if a great number of units are removed, it no longer can expect that a combination of units will be present when evaluating the result and so it cannot depend on complex inter-correlations between units (HINTON et al., 2012).

By dropping different units, dropout effectively trains the ensemble of all sub-

networks of the base network. Typical dropout rates are 20% for the input layer and 50% for hidden layers (GOODFELLOW; BENGIO; COURVILLE, 2016, p. 255, 257).

3.5.3 Optimizers

Gradient descent has the theoretical basis for convergence, but it has big problems for use in practice. The first problem is that the whole training set must be averaged when calculating the loss $J(\boldsymbol{\theta})$, even for small datasets this is quite restrictive, for larger datasets containing hundreds of thousands or millions of samples this can make each iteration take an immense amount of time. This type of approach is commonly called Batch Gradient Descent.

Another problem is that, even if calculating the gradient was very fast, the simple parameter update (recall Equation (22)) can become very slow, specially later in training. The point in parameter space can reach a region in the loss surface where the curvature is minimal, making the gradients very small; the updates could also jump around a valley in the loss surface, never reaching the local minimum (GOH, 2017).

For calculating the gradients faster there is the approach called Stochastic Gradient Descent (SGD). There are several different methods for better traversing the loss surface, these are called optimizers. For the purposes of this document only the Momentum, RMSProp, and Adam optimizers will be briefly explained.

3.5.3.1 Stochastic Gradient Descent

Instead of calculating the gradient $\nabla_{\boldsymbol{\theta}} J$ for all inputs in the dataset, SGD instead approximates this value by calculating it for a single sample on the input. This makes the updates much faster, but will cause the updates to fluctuate heavily because of the gross approximation; this however can be beneficial since it can make the updates jump to potentially better local minima and it has been shown that, for small learning rates, SGD will also converge (RUDER, 2016).

Another way to approximate the gradient is by calculating it only for a mini-batch of the dataset, that is, a subset of m samples of the data. This is called Mini-batch Gradient Descent but is also more commonly referred to as SGD as well. Using mini-batches instead of a single sample gives more stable updates while still being very fast, the values for m usually fall between 50 and 256 (RUDER, 2016), but lower values are also common.

Another advantage of the stochastic approach is that the noise in the gradient approximation has an added regularization effect (BENGIO, 2012, p. 5).

3.5.3.2 Momentum, RMSProp, and Adam

The simple parameter update (Equation (22)) for Gradient Descent has some problems that make it difficult to converge in practice. Some of the problems mentioned

by Ruder (2016) are: that it is hard to choose a good value for the learning rate; only using a single learning rate can be insufficient; the parameters can get stuck in difficult regions of the loss surface, especially saddle points.

Between the many different approaches to combat this, one of the most popular that is also recommended when training GANs (GOODFELLOW, 2017, p. 20, 27) is the Adaptive Moment Estimation (Adam) optimizer. This optimizer is essentially a combination of Momentum and RMSProp, two other very popular optimizers.

Momentum, as the name suggests, is a way to give some inertia to the gradient updates, accelerating in the consistent directions while dampening oscillations. This technique has a strong theoretical basis and can give a quadratic speedup on many functions (GOH, 2017). Momentum changes the parameter updates by adding a velocity term controlled by a new hyperparameter β as shown in the following equations.

$$\boldsymbol{m}_i = \beta \boldsymbol{m}_{i-1} + \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{i-1}) \tag{33}$$

$$\boldsymbol{\theta}_i = \boldsymbol{\theta}_{i-1} - \eta \boldsymbol{m}_i \tag{34}$$

The RMSProp algorithm also tries to reinforce movement to the most relevant directions, it does this by keeping a exponentially moving average v of the gradient accumulation (GOODFELLOW; BENGIO; COURVILLE, 2016, p. 303–304). It also introduces another β hyperparameter and changes the updates as follows.

$$\boldsymbol{v}_i = \beta \boldsymbol{v}_{i-1} + (1-\beta) \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{i-1})^2 \tag{35}$$

$$\boldsymbol{\theta}_i = \boldsymbol{\theta}_{i-1} - \frac{\eta}{\sqrt{\boldsymbol{v}_i + \epsilon}} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{i-1}) \tag{36}$$

Recall that although these equations are represented in vector form, all operation are performed element-wise. Finally for the Adam algorithm, it can be seen as a combination of the two previous algorithms. It introduces two new hyperparameters β_1 and β_2 and the parameter updates work as follows (KINGMA; BA, 2017).

$$\boldsymbol{m}_i = \beta_1 \boldsymbol{m}_{i-1} + (1-\beta_1) \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{i-1}) \tag{37}$$

$$\boldsymbol{v}_i = \beta_2 \boldsymbol{v}_{i-1} + (1-\beta_2) \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{i-1})^2 \tag{38}$$

$$\hat{\boldsymbol{m}}_i = \frac{\boldsymbol{m}_i}{1-\beta_1^i} \tag{39}$$

$$\hat{\boldsymbol{v}}_i = \frac{\boldsymbol{v}_i}{1-\beta_2^i} \tag{40}$$

$$\boldsymbol{\theta}_i = \boldsymbol{\theta}_{i-1} - \eta \frac{\hat{\boldsymbol{m}}_i}{\sqrt{\hat{\boldsymbol{v}}_i + \epsilon}} \tag{41}$$

Kingma and Ba (2017) mentions that good default values for the hyperparameters are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. The Tensorflow library (ABADI et al., 2015) follows this default, only changing ϵ to a order of magnitude higher, that is, $\epsilon = 10^{-7}$.

3.5.4 Batch Normalization

One important aspect in training neural networks is the distribution of the inputs for a layer. As mentioned in subsubsection 3.2.2.1 about sigmoid, Yann A LeCun et al. (2012) showed that inputs that are not zero centered will have some bias effect on parameter updates, to avoid this the layer inputs should all be shifted to have a mean of zero; the authors also suggest to normalize all inputs in order to have the same covariance and, if possible, to have their values be uncorrelated. This should significantly speed up the learning process since the network will not have to adapt to a different distribution for every input.

Normalizing inputs is then an easy and effective way for better learning, therefore being a very commonly used technique that will also be used for the experiments proposed in this document.

However Ioffe and Szegedy (2015) note that for any layer in the network, the normalization idea for its inputs also applies; that is, any hidden layer feeding its output to the next layer can be considered as an input layer for a sub-network consisting of all the subsequent layers. Therefore, the same advantages for normalized inputs would be beneficial by normalizing hidden layer outputs.

Looking from the training perspective, each training iteration updates all parameters at the same time, but the gradient change in a layer gives the best change considering that all other layers remain constant (GOODFELLOW; BENGIO; COURVILLE, 2016, p.313-314). In reality, changing the parameters of the previous layers will affect the distribution of inputs for the current layer, this constant change to the inputs of a layer resulting from changes to previous layers is what Ioffe and Szegedy (2015) call *internal covariate shift*.

To address these issues Ioffe and Szegedy (2015) proposed a technique known as Batch Normalization (BN), this method can be applied to any hidden layer and it consists of normalizing the layer inputs by the mean and variance of all the inputs in the current mini-batch. So, for a mini-batch $\mathcal{B}$ of size m and inputs $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m)$, the normalized inputs $\hat{\boldsymbol{x}}_i$ are calculated using the mean $\boldsymbol{\mu}_\mathcal{B}$ and variance $\boldsymbol{\sigma}_\mathcal{B}$ of the batch. Equations 42, 43, and 44 show how the mean, variance, and normalized inputs are calculated.

$$\boldsymbol{\mu}_\mathcal{B} = \frac{1}{m}\sum_{i=1}^{m} \boldsymbol{x}_i \tag{42}$$

$$\boldsymbol{\sigma}_\mathcal{B}^2 = \frac{1}{m}\sum_{i=1}^{m} (\boldsymbol{x}_i - \boldsymbol{\mu}_\mathcal{B})^2 \tag{43}$$

$$\hat{\boldsymbol{x}}_i = \frac{\boldsymbol{x}_i - \boldsymbol{\mu}_\mathcal{B}}{\sqrt{\boldsymbol{\sigma}_\mathcal{B}^2 + \epsilon}} \tag{44}$$

Note that these operations are applied element-wise for each component of the vectors, and again ϵ is used for numerical stability.

These operations result in the vectors $\hat{\boldsymbol{x}}_i$ having a mean of 0 and variance of 1 for each of their dimensions. However, by limiting the distribution to only these values also limits the representation power of the network (IOFFE; SZEGEDY, 2015), so the last step is to scale each one of the normalized activations by two new learnable parameters, γ and β, unique for each activation, to obtain the transformed input $\boldsymbol{y}_i$ that can have any mean and variance. Equation (45) shows how to scale the dimension k of the normalized input i.

$$y_i^{(k)} = \gamma^{(k)} x_i^{(k)} + \beta^{(k)} \tag{45}$$

This method has proven to be very powerful, with the original paper being able to reach the same accuracy as the, at the time state of the art image classification model, in 14 times less training iterations (IOFFE; SZEGEDY, 2015). The authors also cite a regularization effect of BN that can reduce the need for other techniques like dropout.

When the authors originally proposed this method, they recommended applying the batch normalization operation directly to the weighted inputs of the network and only after apply the activation to the normalized values. However, since then Mishkin, Sergievskiy, and Matas (2017) empirically showed that applying BN after the activation produces better results.

3.5.5 Vanishing Gradients

Recall that for Equations 26 and 27 the δ term used to calculate the gradients, was directly dependent on the derivative of the activation function with relation to the weighted inputs $f'(\boldsymbol{z}^{(l)})$. Also note that the gradients are backpropagated through the network, which means that the δ values in a hidden layer are calculated from the δ in the next layer.

Consider now the case for a sigmoid or tanh activation function which is applied to an weighted input far from zero, the activation in this case will be very close to the maximum or minimum value that the function can produce. When a unit outputs this kind of value it is said that it saturated, any slight deviation in the weighted input will barely have any difference in the activation value, in other words the derivative of the activation is close to zero.

Combining the points made in the last paragraphs it is possible to recognize a problem, when a unit saturates its gradient will be very small since the derivative of the activation is close to zero. Even worse than that is the fact that this small gradient will be backpropagated through the network, reducing the gradients of all previous layers. When many units saturate, this effect can compound, making the gradients for deeper layers very small and greatly reducing the training speed; this is the problem known as the *Vanishing Gradients* problem.

One alternate case is when the gradients are high and compound to make the gradients in deeper layers become very high, making the steps too large and not converging

to any local minima. This is called the *Exploding Gradients* problem.

Both situations are undesirable and are specially worrying when trying to train very deep neural networks. Ideally the gradients should all be close to 1 to make training consistent, but for saturating activation functions like sigmoid, there will usually be vanishing gradients (NIELSEN, 2015). That is one of the main reasons to use non-saturating activation functions like ReLU in the hidden layers of the network. Other common ways to combat this problem is to carefully initialize the network parameters and use small learning rates (IOFFE; SZEGEDY, 2015), these authors also suggest that BN can be used on saturating units since it can reduce the chance that the input of the units will fall to the saturating regions.

4 GENERATIVE ADVERSARIAL NETWORKS

GANs fall into a particular subfield of machine learning called *generative modeling*, the goal of this area of study is to be able to generate original data based on a training dataset. As mentioned in subsection 3.1.2, the data found in real life scenarios for any given situation is just a very tiny fraction of all the possible values in the input space, recall the MNIST example given, just a minuscule subset of all possible images can be sensibly interpreted as digits. It can be said that real world data has some structure, and generative models aim to replicate this structure in order to sample from it.

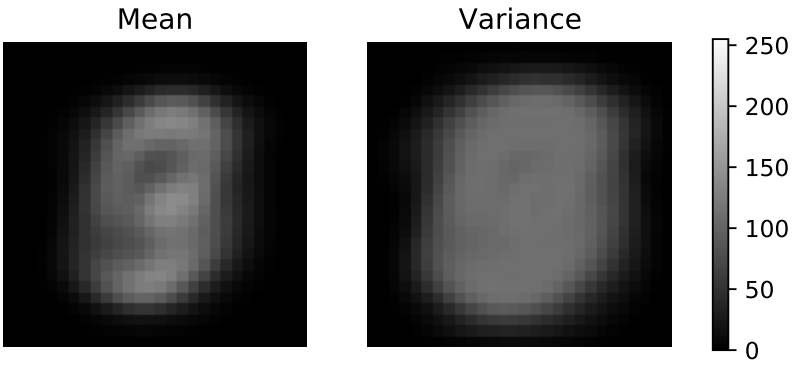
4.1 GENERATIVE MODELS AND DATA DISTRIBUTIONS

Goodfellow (2017) explains that any dataset is made of samples taken from some probability distribution p_{data} that defines the structure of the data, and all the different techniques in generative modeling are trying to produce a p_{model} to replicate as close as possible the underlying distribution of the data.

If p_{data} was known for a given problem, then it could be used by itself to generate original data, but calculating this distribution is basically impossible for all but the simplest datasets, so the most a model can do is try to replicate it. To better understand the complexities in probability distributions it can be helpful to look at some characteristics in the, relatively simple, MNIST dataset.

Recall that the MNIST dataset consists of $70,000$ grayscale images of size 28×28, and these images are color inverted. When introducing the dataset in section 2.1 the images were inverted again to show the original colors, but here the properties will be analyzed without making any preprocessing on the data. First it is helpful to look at what is the mean and variance for each pixel in the MNIST dataset, these values are shown in Figure 15.

Figure 15 – MNIST mean and variance



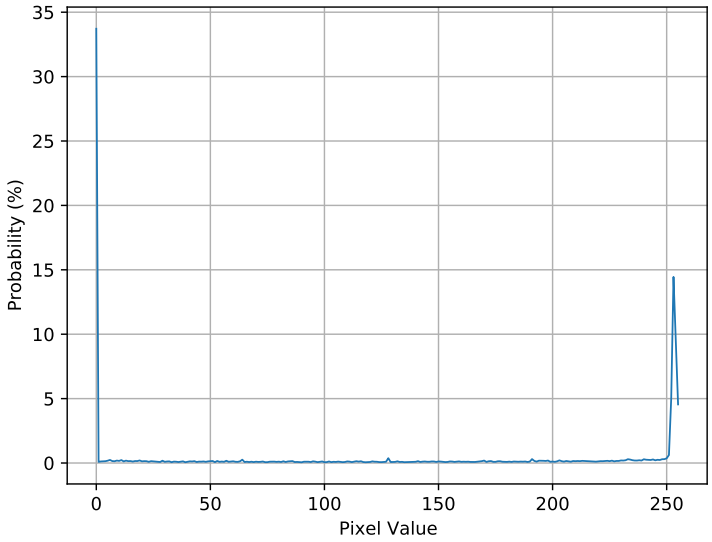
Source – From the author (2021)

The information in Figure 15 already gives some insight into how the data is

structured, the edges of the image practically see no change since all the digits are centered, and it is possible to see dips in brightness in the middle-top and middle-bottom of the digits; these represent the spaces that all the digits are drawn around (i.e. the two holes in the number 8).

Another way to see the distribution is to directly plot the probability of seeing the different values for a pixel, this is done by counting how many times each value has appeared in a selected pixel for all images in the dataset, the probability is then the number of value occurrences divided by the total number of images. Doing this for one of the center pixels, in row 15 and column 15, the end result is the probability distribution seen in Figure 16.

Figure 16 – Probability distribution of values in pixel (15,15) of MNIST

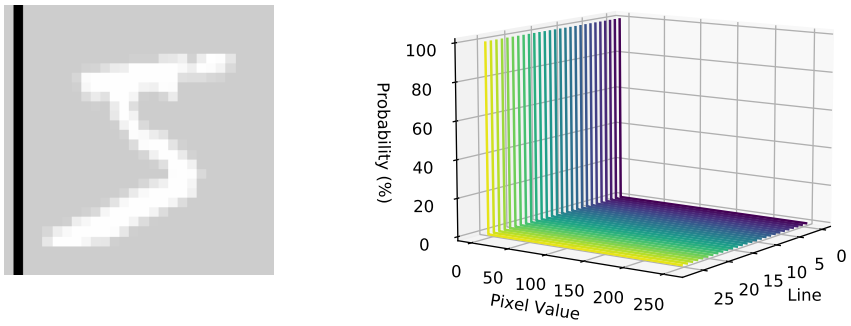


Source – From the author (2021)

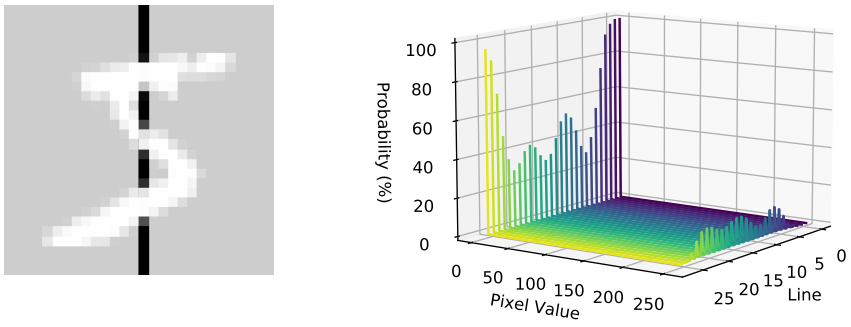
The probability distribution in Figure 16 gives even more information about the dataset, it can be seen that the values are either completely black, or a very bright white. Any grayish values are very unlikely since with handwriting digits the strokes are very sharp and well defined. It is possible to extend Figure 16 by calculating the distribution for a whole column of pixels, the result is a 3D surface, where the new axis represents the corresponding pixel in the column. Figure 17 shows this surface for columns 2 and 15 of the MNIST dataset, note that the digit besides the shape is there only to illustrate the position of the column, the distribution is for the entire dataset.

The probabilities shown in Figure 17 reinforce what was seen for the mean and variance of the images in Figure 15, for the edges the probability is basically 100% that a pixel will be completely black, while for the center column it is possible to see the probabilities being split into very dark or very light pixel values, it even has two slight peaks for the black pixels representing the spaces between the digits (i.e. the holes in the number 8) as also seen for the mean and variance case.

Figure 17 – Probability distribution of values in columns of MNIST



(a) Column 2

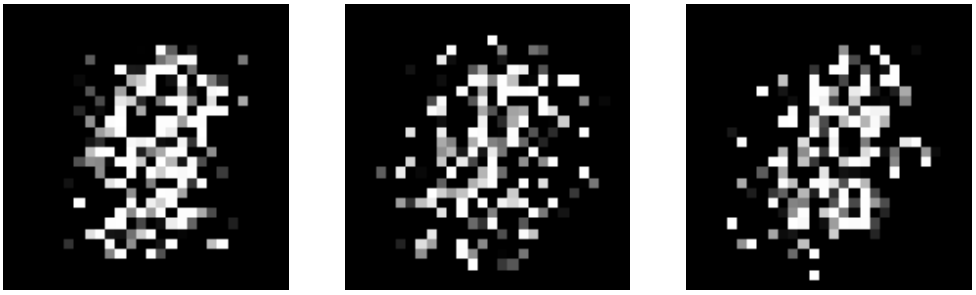


(b) Column 15

Source – From the author (2021)

One may think that this would be enough to represent and generate new data, since the probability distribution for each pixel is already calculated, wouldn't all the image be described? And by sampling the distribution for each pixel, wouldn't it be possible to generate new images of digits? It is certainly possible to try, Figure 18 show three examples of images generated by sampling from the pixel distributions.

Figure 18 – Images generated by sampling from the pixel distributions of MNIST



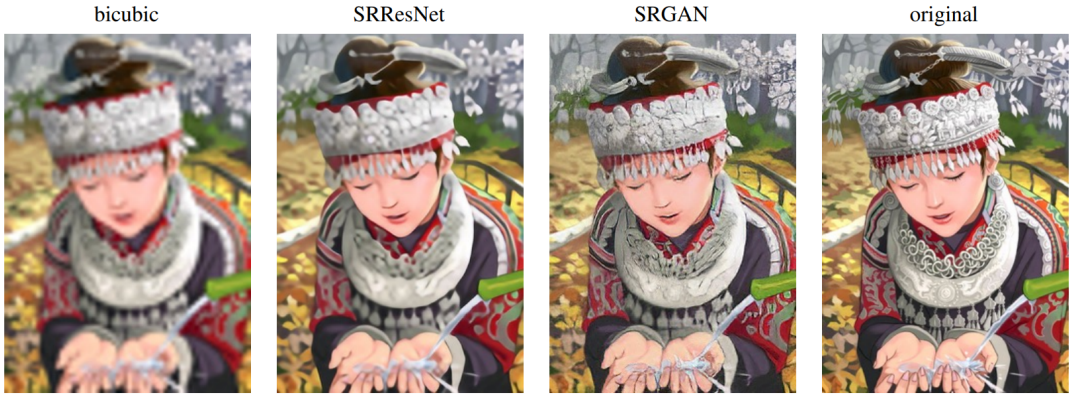
Source – From the author (2021)

The sampled images surely have some similarities, but they are far from being digits. Although the probability distributions calculated are very helpful in giving some insight into the dataset, they consider the pixels as completely independent from one another. In reality, neighbouring pixels influence the values of each other, for handwritten digits a pixel will practically never be white while its neighbours are all black, instead it is much more likely that a white pixel will have some of its neighbours also being white in order to produce a stroke in the image. By averaging out all the influences of every pixel, the resulting distributions are those seen in Figure 16 and Figure 17; they are valid descriptions of the data, but cannot be used to fully represent the underlying structure.

The results in Figure 18 are similar to an effect seen in some neural network models, the data has many possible correct representations, but instead of picking a single one, the model averages out everything and ends up with a blurry mixture that badly represents the data. This can be seen for example in models that colorize black and white images (DAHL, 2021) (e.g. averaging reds, yellows, blues and all other common car colors, results in painting most cars in the same bland sepia tones), and models that aim to increase the resolution of images (LEDIG et al., 2016).

One of the advantages of GANs is that they are more resistant to these kinds of problems, Figure 19 bellow shows an example where the original image was downscaled and different approaches were used to upscale the result back to its original size. Note how the GAN (called SRGAN) produces sharper results when compared with the algorithmic approach of bicubic upscaling and with another neural network that does not use a GAN architecture (SRResNet) – this latter case has some understanding of the underlying structure, but it suffers to pick one good solution and instead ends up averaging all possible answers resulting in a blurry image (GOODFELLOW, 2017).

Figure 19 – Comparison of different upscaling techniques (upscaling $\times 4$)



Source – Adapted from Ledig et al. (2016)

In the next section the reason why GANs are better at this will be explored more deeply. For now these examples show how complex the probability distributions of data can be (even knowing all the probabilities for each pixel still is not enough), given this

complexity it is only possible to approximate the distributions, and generative models are a way of doing that.

One may question the purpose of generative models since the idea behind them is simply to generate more of what there is already a lot of. Indeed for cases like MNIST there is little value in generating a lot more digits, this problem has been solved since the 1990's (LECUN, Y. et al., 1998), and now it only serves as a learning and benchmark tool. However the use of generative models can be extended to more diverse situations, as already seen for upscaling images in Figure 19. Goodfellow (2017) highlights some other uses of generative models, including: different ways to incorporate these models into Reinforcement Learning, leveraging unlabelled data as seen in semi-supervised learning, image-to-image translation, and creation of art.

4.2 THE GAN ARCHITECTURE

The GAN is a type of neural network that was introduced by Goodfellow, Pouget-Abadie, et al. (2014) as an alternative to other generative models at the time. The main idea behind it is the competition of two different networks, a *generator* and a *discriminator*, hence the term *Adversarial* in Generative Adversarial Network. The discriminator is trained with the simple goal of detecting if any given sample belongs or not to the original dataset. The generator on the other hand is trained to make the discriminator fail, its goal is to create samples that the discriminator will consider real.

A common analogy given for this process is that of the police trying to identify counterfeit money (GOODFELLOW, 2017), the generator in this analogy is the criminals, and the discriminator is the police. The criminals will start making bad replicas that are easy for the police to learn to distinguish from real money, this will force the criminals to make better copies and in turn demand more from the police. If this keeps going forever, in the end the criminals would be so good at counterfeiting that the result would be indistinguishable from real money and the police would have no better way than to guess the answer (50% accuracy). This idea stems from game theory and is known as the *Nash Equilibrium* of the system, the result was rigorously proven in the original GAN paper (GOODFELLOW; POUGET-ABADIE, et al., 2014) for the case where a discriminator is trained to the optimum before each step in the generator.

Leaving behind the analogy, the real implementation and training of a GAN consists of creating the generator (G) and discriminator (D) networks, and defining a new loss function ($J^{(G)}, J^{(D)}$) for each one. The networks can be built in any way, using fully connected, convolutional, or any other kind of layer. The discriminator input must be of the same shape as the input data and it should output a single number between 0 and 1 (a sigmoid can be used in the last layer), this number represents the probability of the input being real.

The output of the generator must also be the same shape as the input data since it

will be fed to the discriminator. However, the generator input should be a n dimensional vector, this vector is usually randomly sampled from a random uniform or Gaussian distribution and is called the *latent vector* ($\boldsymbol{z}$), the n dimensional vector space is called the *latent space* ($\mathcal{Z}$). When describing samples from a distribution the common notation is $\boldsymbol{z} \sim p_{\mathcal{Z}}$, this means that the value $\boldsymbol{z}$ assumes the probability distribution $p_{\mathcal{Z}}$ (e.g. random Gaussian).

The process of training the discriminator consists of sampling a batch of real data $\boldsymbol{x} \sim p_{data}$ and a batch of latent vectors $\boldsymbol{z} \sim p_z$, the values of $\boldsymbol{z}$ are fed to the generator to produce a batch of fake data $G(\boldsymbol{z})$; the discriminator is trained to label $\boldsymbol{x}$ as 1 (real) and $G(\boldsymbol{z})$ as 0 (fake). The generator never sees the data, it is trained using only $G(\boldsymbol{z})$. Figure 20 shows a diagram of how these networks use the data for training.

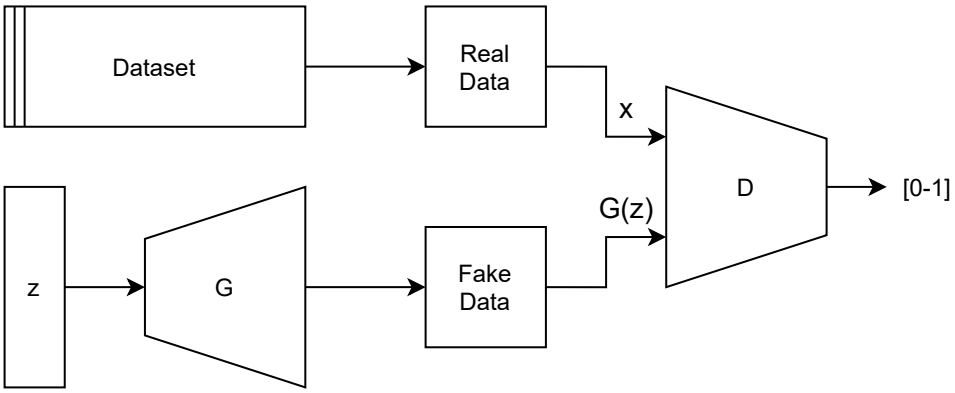


Figure 20 – Diagram of data use in training GANs

To understand how these networks can learn, it is necessary to delve deeper into the theory and analyse how their loss function is defined. In the general sense, both the generator G and discriminator D are just functions of multiple variables. The generator is a map $G : \mathcal{Z} \to \mathcal{X}$, where $\mathcal{Z}$ is the latent space, defined by a probability distribution p_z, and $\mathcal{X}$ is the true data space, defined by a probability distribution p_{data}. The discriminator is a map $D : \mathcal{X} \to \mathbb{R}$ and $0 \leq D(\boldsymbol{x}) \leq 1$ (i.e. the probability of $\boldsymbol{x}$ belonging to the real data). In most practical cases D and G are implemented with neural networks, parameterized by $\boldsymbol{\theta}^{(D)}$ and $\boldsymbol{\theta}^{(G)}$ respectively.

The idea behind GANs stems from Game Theory, where two agents compete against each other in a non-cooperative game (SALIMANS et al., 2016), the solution for this game is called the Nash equilibrium and in this situation both players have achieved their best expected value given the state of their adversary. Training a GAN is a *minimax* game that aims to reach the Nash equilibrium between the generator and the discriminator, the objective of these networks is given by Equation (46) (GOODFELLOW; POUGET-ABADIE, et al., 2014).

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x} \sim p_{data}}\Big[\log(D(\boldsymbol{x}))\Big] + \mathbb{E}_{\boldsymbol{z} \sim p_z}\Big[\log(1 - D(G(\boldsymbol{z})))\Big] \tag{46}$$

Equation (46) can be quite intimidating at first, but it is more easily understood by breaking it down into parts. First it is important to define what the symbol $\mathbb{E}$ means, this symbols represents the *Expected value* of the operation between the square brackets. The expected value is a concept in statistics that means the average value that a variable will assume given some probability distribution, suppose for example rolling a 6-sided die, the expected value of the dice roll n given the probability distribution $n \sim p_{dice}$ is given by Equation (47).

$$\mathbb{E}_{n \sim p_{dice}}(n) = \sum_{n=1}^{6} n \cdot p_{dice}(n) = 3.5 \tag{47}$$

More generally, the expected value $\mathbb{E}$ of a function $f(x)$, for all values x following a probability distribution p, is represented as $\mathbb{E}_{x \sim p(x)}(f(x))$ (the use of square brackets in Equation (46) is not needed, it was only used to better separate the terms). This value is calculated as the sum of the values $f(x)$ multiplied by their respective probability given the distribution p. The general case for a finite probability distribution is represented in Equation (48).

$$\mathbb{E}_{x \sim p}\left(f(x)\right) = \sum_{i} f(x_i) p(x_i) \tag{48}$$

When dealing with continuous probability distributions the summation in Equation (48) is replaced by an integral, but in the context of machine learning most problems fall into the discrete category and the expected value is commonly approximated by calculating it from a mini-batch of data instead of the whole dataset.

With this understanding of expected value, it is possible to come back to the objective function in Equation (46). Note that this is a minimax game, both the generator and discriminator are dependent on the same value $V(D, G)$, the discriminator tries to maximize the value, while the generator tries to minimize it. So the objective can be broken down into two loss functions $J^{(D)}$ and $J^{(G)}$, where each network is trying to minimize their own loss.

The objective of the discriminator is to maximize $V(D, G)$, this can be re-framed as a minimization problem over a loss function $J^{(D)} = -V(D, G)$ as mentioned by Goodfellow (2017). So the loss for the discriminator is given by:

$$J^{(D)}(\boldsymbol{\theta}^{(D)}, \boldsymbol{\theta}^{(G)}) = -\mathbb{E}_{\boldsymbol{x} \sim p_{data}}\left[\log(D(\boldsymbol{x}))\right] - \mathbb{E}_{\boldsymbol{z} \sim p_z}\left[\log(1 - D(G(\boldsymbol{z})))\right] \tag{49}$$

Observe what Equation (49) is saying, for the first term, $-\log(D(\boldsymbol{x}))$ will tend to infinity as $D(\boldsymbol{x}) \to 0$, the minimum value is 0 for when $D(\boldsymbol{x})$ is exactly 1; in other words, this term will be minimized when the discriminator correctly assigns the real data $\boldsymbol{x}$ as being 100% real. The second term, $\log(1 - D(G(\boldsymbol{z})))$ will be minimized in the opposite way, when $D(G(\boldsymbol{z}))$ is equal to 0; this means that this term encourages the discriminator to correctly assign the fake data as being fake.

The loss for the generator in the minimax game is simply the negative of the loss of the discriminator loss, $J^{(G)} = -J^{(D)} = V(D, G)$. One thing to note in the loss for the

generator is that the first term of $V(D, G)$, as seen in Equation (46), only depends on D, so it can be ignored since the generator can't affect the parameters $\boldsymbol{\theta}^{(D)}$. Then the loss for the generator can be written as the equivalent expression shown in Equation (50).

$$J^{(G)}(\boldsymbol{\theta}^{(D)}, \boldsymbol{\theta}^{(G)}) = \mathbb{E}_{\boldsymbol{z} \sim p_z}\Big[\log(1 - D(G(\boldsymbol{z})))\Big] \tag{50}$$

In this case, $\log(1 - D(G(\boldsymbol{z})))$ will be minimized when $D(G(\boldsymbol{z}))$ is equal to 1; in other words, the loss of the generator is minimized when the discriminator incorrectly classifies the fake data as being real, the generator is encouraged to reduce the discriminator accuracy.

One may wonder how can this game converge to producing a p_{model} similar to p_{data}. To understand that, first it is necessary to understand how the difference between two probability distributions is measured. Much like distances between two points in space, that can be measured in different ways to produce different properties in that space (e.g. ℓ_2 norm for the familiar Euclidean Space), there are multiple definitions of distance between probability distributions. Some definitions produce better properties that can be leveraged to solve particular problems, in the original GAN proposal, Goodfellow, Pouget-Abadie, et al. (2014) proved that the objective given by Equation (46) is equivalent to reducing the distance called the Jensen-Shannon (JS) divergence.

The JS divergence is written in terms of another function, the Kullback-Leibler (KL) divergence. The use of the KL divergence can be found in many areas of machine learning, not just generative models, this is a very powerful metric that is closely tied to the concepts of information entropy and cross-entropy. The KL divergence between two probability distributions p and q is given by Equation (51) (GOODFELLOW; BENGIO; COURVILLE, 2016, p. 71-72).

$$D_{KL}(p \parallel q) = \mathbb{E}_{x \sim p}\left(\log \frac{P(x)}{Q(x)}\right) = \sum_x P(x)\left(\log \frac{P(x)}{Q(x)}\right) \tag{51}$$

One common characteristic between many metrics of distance for probability distributions is that they are not symmetric, that means that the distance between p and q is not the same as the distance between q and p, this asymmetry is also present in the KL divergence as can be seen in Equation (51) – it is for this reason that the name divergence is used instead of distance. Contrary to that, the JS divergence, although being called a divergence, is actually symmetric; it is defined in terms of the KL divergence as shown in Equation (52).

$$D_{JS}(p \parallel q) = \frac{1}{2}D_{KL}(p \parallel m) + \frac{1}{2}D_{KL}(q \parallel m) \qquad \text{where} \qquad m = \frac{p+q}{2} \tag{52}$$

For both the KL and JS divergences, a value of 0 represents that the distributions p and q are the same, and minimizing these divergences brings the two distributions closer. As mentioned before, Goodfellow, Pouget-Abadie, et al. (2014) proved that the GAN

objective function given in Equation (46) minimizes the JS divergence and in turn should converge p_{model} to p_{data}. The authors also proved that this convergence point equates to the optimal discriminator being unable to differentiate between real and fake data, having the same accuracy as a random guess (50%).

The details of the proof are out of the scope of this document, the important information to take is how the networks reproduce the data distribution (by minimaxing the objective function) and why this works (equivalent to reducing the JS divergence). One important detail is that the proof relied on the fact that the updates to G and D were made directly in function space, the same argument does not apply to the situations seen in practice, where the updates to the functions are made on parameter space (i.e. $\boldsymbol{\theta^{(G)}}$, $\boldsymbol{\theta^{(D)}}$) (GOODFELLOW, 2017). This is not a problem in many situations, but there are multiple situations where this approach has been shown to diverge, or to cycle around the equilibrium without convergence; some examples of this are shown by Salimans et al. (2016), Arjovsky, Chintala, and Bottou (2017), Gulrajani et al. (2017) and Mescheder (2018).

One problem of minimizing the loss function of the generator $J^{(G)}$ as seen in Equation (50), is that, in the start of training when the generator is still bad at producing results, the discriminator can become too good and will recognize the fake data with very high certainty; this confidence will saturate the discriminator output and give vanishing gradients for the generator updates, making training extremely slow. Given this problem, the original paper (GOODFELLOW; POUGET-ABADIE, et al., 2014) proposed a change to the loss of the generator; instead of minimizing the probability of the discriminator being correct, the generator should maximize the probability of the discriminator making a mistake. This equates to minimizing the loss function shown in Equation (53).

$$J^{(G)} = -\mathbb{E}_{z\sim p_z}(\log D(G(\boldsymbol{z}))) \tag{53}$$

It is important to mention that this new loss function is an empirical recommendation, the theoretical arguments of convergence do not apply when training the generator with this function (GOODFELLOW; POUGET-ABADIE, et al., 2014). However, this does not seem to be a big problem in practical situations and is usually the preferred loss function between the two.

Another point worth mentioning is that for the theoretical argument of convergence, the generator updates would be made in relation with an optimal discriminator, this would mean that for each generator step, there should be multiple updates in the discriminator in order to have it be optimal for the current generator. At the beginning this was implemented as a new hyperparameter that would define how many updates to the discriminator would be made before updating the generator (GOODFELLOW; POUGET-ABADIE, et al., 2014), but later Arjovsky and Bottou (2017) have shown that the optimal discriminator has gradient 0 almost everywhere, making it impossible to train GANs through gradient descent. This hyperparameter has since fallen out of fashion and usually

GANs are trained one step for each network, the original inventor of GANs would also say: "Many authors recommend running more steps of one player than the other, but as of late 2016, the author's opinion is that the protocol that works the best in practice is simultaneous gradient descent, with one step for each player." (GOODFELLOW, 2017).

One may note how often the theory either fails to apply to practical cases or is replaced by empirical solutions that work better. This is very much the case not only for GANs, but for many other areas of machine learning, "[...] even though these [Artificial Neural Networks] are very useful tools based on well-known mathematical methods, we actually understand surprisingly little of why certain models work and others don't" (MORDVINTSEV; OLAH; TYKA, 2015). In most GAN methods introduced with a theoretical basis behind them, it is very common to see assumptions being made in order for the theorems proposed to apply – see for example the original GAN paper (GOODFELLOW; POUGET-ABADIE, et al., 2014), or (ARJOVSKY; CHINTALA; BOTTOU, 2017) and (HEUSEL et al., 2017).

4.2.1 Mode Collapse

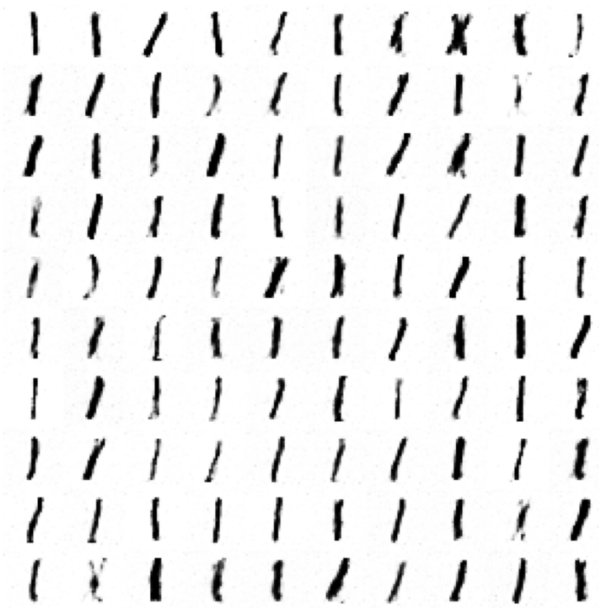
One of the main problems faced when training GANs is when the generator learns to map many, or all possible latent vectors in $\mathcal{Z}$ to the same point $\boldsymbol{x}$ in $\mathcal{X}$, this is called a mode collapse, also known as the "Helvetica Scenario". Goodfellow (2017) says that complete mode collapse is the most common form of harmful non-convergence in GANs and that, although complete collapse is rare, partial collapse is a frequent occurrence.

As an example of mode collapse, consider the case of training a GAN to generate the handwritten digits of MNIST, the generator collapsing could be that it only produces the number 6 for all latent vectors, it may produce convincing results, but it can't represent the full distribution. This in theory could be useful, since new generators could be trained for each separate mode of the data (e.g. one generator for each digit). However this is usually not desirable, still considering the MNIST example, after many iterations the discriminator would learn to be more suspicious of the number 6 and the generator would then try to find a next mode to collapse (e.g. the number 8); thus, both networks would be forever stuck changing modes and never reaching convergence (SALIMANS et al., 2016).

Figure 21 shows an example of extreme mode collapse that can happen when training GANs, in this particular case the generator is a simple neural network of fully connected layers, having a single hidden layer, and being trained on the MNIST dataset[1].

[1] When producing this image, the networks were trained four different times with different sets of hyperparameters, and in all cases the mode collapse happened on the number 1. This probably indicates that this is the easiest pattern for the generator to learn, but most importantly, it gives another counterargument on why it is usually not desirable to train several networks one for each mode, since it is difficult to produce the correct mode.

Figure 21 – Mode collapse on MNIST



Source – From the author (2021)

4.3 PROPOSED IMPROVEMENTS

Since their introduction in 2014, GANs have become very popular for their capabilities but also for their difficult of being trained (GULRAJANI et al., 2017) and evaluated (ARJOVSKY; BOTTOU, 2017). Many improvements have been proposed, of note between them are changes that aim to make training more stable and faster, reduce the effect of mode collapse, produce better results, scale the results to higher resolutions (in case of images), make the latent space have nicer properties, introduce new metrics of quality, and condition the generator on some know input in order to guide the generation process.

This section will explore some of the more popular methods, this is by no means an exhaustive list, the goal is only to introduce the techniques that were part of the empirical experiments in this document (see Chapter 5).

4.3.1 DCGAN

The original GAN suffered from training stability and difficulty of scaling to larger resolutions (in the case of images), Radford, Metz, and Chintala (2015) were able to compile different popular techniques at the time to produce an architecture for the generator and discriminator that would produce better results and be more stable. They called this type of model a Deep Convolutional GAN (DCGAN) since they abandoned the use of fully connected and pooling layers in favor of using only convolutional and transposed convolutional layers.

Different from the other proposed improvements that will be mentioned later, the DCGAN did not introduce any new way of training or using the data differently, it was

simply a clever style of architecture that would produce better results. But the results were indeed very good, so much so that by 2016 most GAN architectures were at least loosely based on the DCGAN (GOODFELLOW, 2017).

The overall architecture can be summarized as follows (RADFORD; METZ; CHINTALA, 2015):

- No fully connected hidden layers and no pooling layers overall, only uses convolution or transposed convolutions to change the dimensions of the input.
- Use of Batch Normalization in all layers of both networks, except the input layer of the discriminator and output layer of the generator.
- Use LeakyReLU for all layers of the discriminator and ReLU for all layers of the generator, use only tanh in the last layer of the generator in order to produce the normalized images (i.e. interval $[-1, 1]$).
- Use of the Adam optimizer

Radford, Metz, and Chintala (2015) were able to have stable training with the DCGAN on a range of different datasets, the architecture was also robust enough to allow for building deeper models and generating higher resolutions. They also showed a surprising property by applying arithmetic on the latent space $\mathcal{Z}$, by taking some random latent vectors that would produce images of mans with glasses and averaging them, the result would be an average vector $\boldsymbol{z}$("man with glasses") that would represent this type of image; by also calculating $\boldsymbol{z}$("man without glasses") and $\boldsymbol{z}$("woman without glasses"), and combining the vectors in the following way $\boldsymbol{z}$("man with glasses") - $\boldsymbol{z}$("man without glasses") + $\boldsymbol{z}$("woman without glasses"), the result would be a latent vector that when fed to the generator would produce an image of a woman with glasses.

4.3.2 Conditional GAN

One of the main advantages of GANs is that they can be trained with unlabeled data, this allows for learning with millions of real samples, such a volume of data is almost always very expensive and time consuming to have labelled. However, one of the earliest proposal for improvement was made by Mirza and Osindero (2014), they argue about the benefits of leveraging the labels when training the GANs.

This approach is conceptually very simple, when training the original GAN the generator is fed some noise vector from $\mathcal{Z}$ and produces a fake sample $G(\boldsymbol{z})$, the discriminator then takes this and another real sample $\boldsymbol{x}$ to produce $D(G(\boldsymbol{z}))$ and $D(\boldsymbol{x})$. For a Conditional GAN (CGAN), everything is the same except for the fact that both the generator and discriminator are also given a label $\boldsymbol{y}$ for the data generated; this means that the generator will produce $G(\boldsymbol{z}|\boldsymbol{y})$ and the output of the discriminator will be $D(G(\boldsymbol{z}|\boldsymbol{y}))$ and $D(\boldsymbol{x}|\boldsymbol{y})$.

The logic behind this approach is that it encourages the generator to learn how to distinguish the data as people do, Goodfellow (2017) comments that this may help the generator in optimizing the solution, but it also could be that the results produced are not necessarily closer to the real distribution, but instead that they favor characteristics that appeal to the human vision.

One advantage of this model is that it allows for more fine control over the result, in the unlabelled GAN the results are random since they come from a sample of the latent space distribution; with CGAN the label input can be used to set the class of the output, while the latent vector can be sampled multiple times to produce variation.

A question about the implementation of CGANs is: how the label can be incorporated into the latent vector and the input image? In the original paper, Mirza and Osindero (2014) had the labels be represented as one-hot encoded vectors. For the generator the input $\boldsymbol{z}$ and the label vector $\boldsymbol{y}$ would be mapped into two layers of 200 and 1000 neurons respectively, these layers would then be combined into another layer that would have the label information imbued. A similar process would happen for the discriminator, it would map both the input and label into two one-dimensional layers and combine them into a new layer.

This way of combining the label with the data has mostly been replaced since then. A better way of doing that is to use embedding layers to represent the class label instead of one hot vectors, this layer should be mapped with a dense layer to a higher dimension and be reshaped into a channel of the input volume to the corresponding network (i.e. generator or discriminator). Figure 22 shows a diagram of how the label is incorporated into the channels using embedding layers.

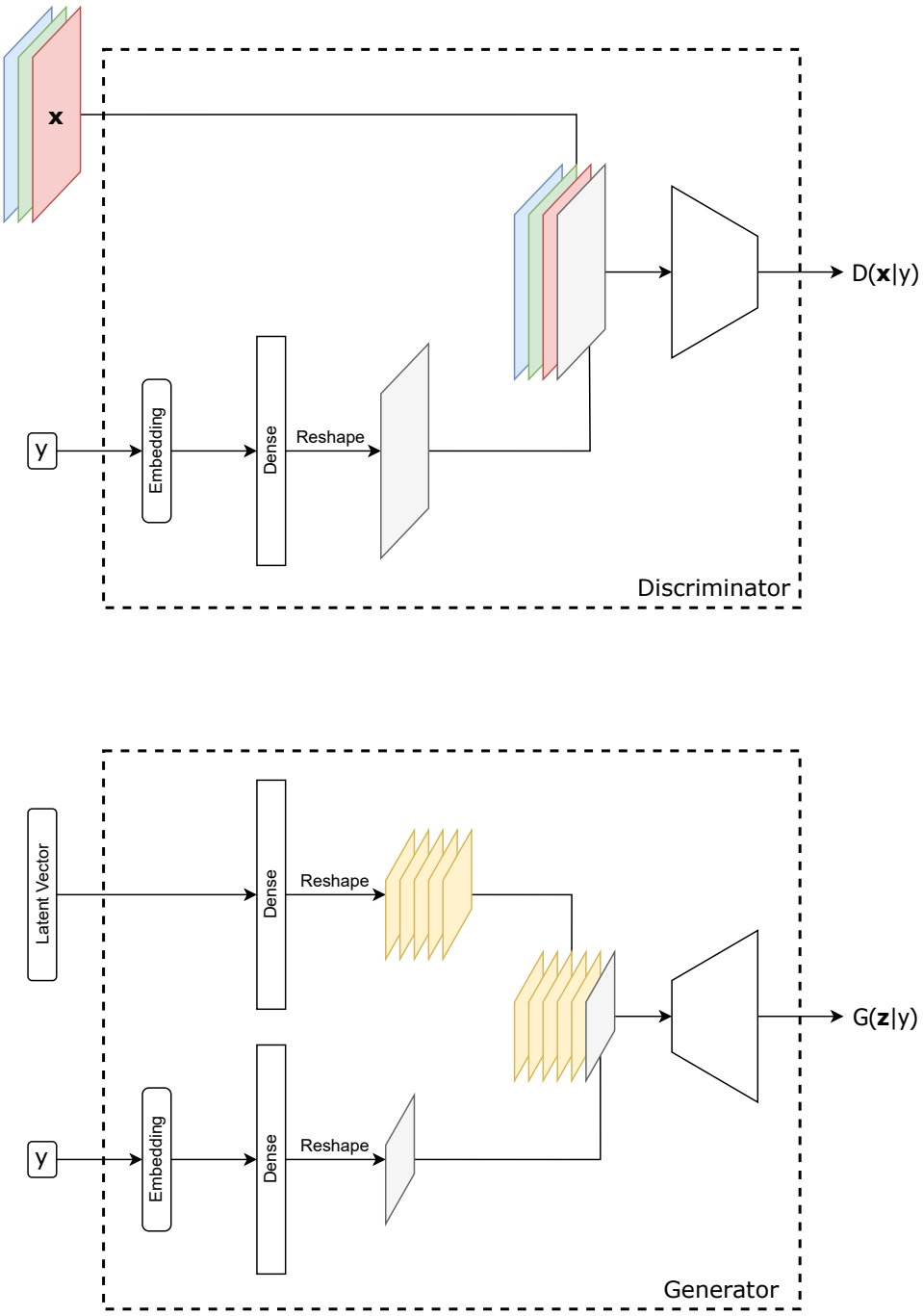
4.3.3 Wasserstein GAN

One of the problems when training GANs by trying to minimize the Jensen-Shannon divergence is that this metric does not produce the best gradients for the generator to learn, recall that Arjovsky and Bottou (2017) have showed that for the optimal discriminator the gradient is equal to 0 almost everywhere. The authors also explain that the Kullbak-Leibler divergence is also problematic since it can produce very high or very low values in different areas of the distributions, making training very difficult.

Arjovsky, Chintala, and Bottou (2017) introduce a simple example of a uniform probability for all points in the line segment $x = 0$, $0 \leq y \leq 1$ on the xy plane. They showed that any modeled distribution $x = \theta$ and $0 \leq y \leq 1$, that is parameterized by a single value θ, will not converge when training with the JS, KL, and reverse KL divergences, or with the total variation distance[2]. They propose the Earth-Mover (EM) or Wasserstein distance as an alternative to these other metrics and showed that in the proposed example,

[2] The details are out of the scope of this document, but the mathematical inclined reader is encouraged to check the original paper.

Figure 22 – Generator and discriminator networks for CGAN



Source – From the author (2021)

this distance produces a continuous value that provides usable gradients everywhere.

The motivation behind this choice of function is heavily inspired by theory. For this document most of the details will be omitted, and only the most relevant information will be quickly cited in order to reach the conclusions and information about the practical implementation. The EM distance between two probability distributions p and q is defined by Equation (54), "where $\Sigma(p, q)$ denotes the set of all joint distributions $\gamma(x, y)$ whose marginals are respectively p and q" (ARJOVSKY; CHINTALA; BOTTOU, 2017).

$$W(p, q) = \inf_{\gamma \in \Sigma(p,q)} \mathbb{E}_{(x,y)\sim\gamma} \|x - y\| \tag{54}$$

In this equation the *infimum* (inf) can be interpreted as the greatest lower bound, so the EM distance is given by the γ distribution that satisfies the greatest lower bound condition for the corresponding expected value. There is an intuitive interpretation of this distance in terms of real world mechanics, consider that p and q describe two mass distributions with equal total mass, then the Wasserstein distance is equivalent to the minimum amount of energy necessary to move the masses around in order to transform the distribution p into q, for this reason that it is also called the Earth Mover distance.

Calculating this distance in this form is extremely difficult, but by using the Kantorovich-Rubinstein duality the original authors rewrote the distance as shown in Equation (55) (ARJOVSKY; CHINTALA; BOTTOU, 2017).

$$W(p, q) = \sup_{\|f\|_L \leq 1} \mathbb{E}_{x\sim p} f(x) - \mathbb{E}_{x\sim q} f(x) \tag{55}$$

The *supremum* (sup) in this equation is also interpreted as the least upper bound and the restriction $\|f\|_L \leq 1$ indicates that f must be 1-Lipschitz continuous. A function f is said to be K-Lipschitz continuous for a constant K when the following inequality holds (ROWLAND; WEISSTEIN, 2021).

$$|f(x) - f(y)| \leq K|x - y| \qquad \forall x, y \tag{56}$$

The inequality above can be interpreted as sliding a cone with inclination K on every point of f and if all the values of the function are outside the cone then the function is K-Lipschitz continuous.

Arjovsky, Chintala, and Bottou (2017) note that the 1-Lipschitz continuity can be replaced by a K-Lipschitz restriction while still maintaining the correct EM distance up to a multiplicative constant $K \cdot W(p, q)$. They also consider solving an easier problem by using a parameterized function $f_{\boldsymbol{\theta}}$, with $\boldsymbol{\theta}$ belonging to the parameter space Θ, and replacing the sup with a max. The simplified problem is given by Equation (57).

$$\max_{\boldsymbol{\theta}\in\Theta} \mathbb{E}_{x\sim p} f_{\boldsymbol{\theta}}(x) - \mathbb{E}_{x\sim q} f_{\boldsymbol{\theta}}(x) \tag{57}$$

This simplified problem relies on the assumption that the supremum can be found for some set of parameters $\boldsymbol{\theta}$, but if this is true, then the EM distance can be calculated

(up to a multiplicative factor) by implementing f as a neural network and using gradient ascent to maximize the value in Equation (57) (ARJOVSKY; CHINTALA; BOTTOU, 2017).

To better understand how this works, the problem in Equation (57) can be written in terms of p_{data}, p_{model} and a generator G as shown in Equation (58).

$$\max_{\boldsymbol{\theta} \in \Theta} \mathbb{E}_{\boldsymbol{x} \sim p_{data}} f_{\boldsymbol{\theta}}(\boldsymbol{x}) - \mathbb{E}_{\boldsymbol{x} \sim p_{model}} f_{\boldsymbol{\theta}}(\boldsymbol{x}) =$$

$$\max_{\boldsymbol{\theta} \in \Theta} \mathbb{E}_{\boldsymbol{x} \sim p_{data}} f_{\boldsymbol{\theta}}(\boldsymbol{x}) - \mathbb{E}_{\boldsymbol{z} \sim p_{\boldsymbol{z}}} f_{\boldsymbol{\theta}}(G(\boldsymbol{z})) \tag{58}$$

One may wonder where is the discriminator in all of this, the astute reader may have already realized that the discriminator is the function f. For the Wassertein GAN (WGAN) however, the output of this function is not a probability but instead it can be any number; the interpretation is that the discriminator is scoring the data that it sees, so it is more commonly referred to as the *critic*.

The goal of the critic is to approximate the EM distance between the data and model distributions $W(p_{data}, p_{model})$, and it does that by maximizing the value in Equation (58). How can the generator use this to learn a good probability distribution?

Since the goal of the generator is to minimize the distance between the real and modeled distributions, its objective should be to minimize the value produced by the critic. Arjovsky, Chintala, and Bottou (2017) proved (theorem 3 of paper) that the gradient of the EM distance for a generator parameterized by $\boldsymbol{\theta}$ is given by Equation (59).

$$\nabla_{\boldsymbol{\theta}} W(p_{data}, p_{model}) = -\mathbb{E}_{\boldsymbol{z} \sim p_{\boldsymbol{z}}}[\nabla_{\boldsymbol{\theta}} f(G(\boldsymbol{z}))] \tag{59}$$

Although the theory is very heavy, the implementations changes are rather simple. The loss for the critic will be just the mean of the fake outputs $f(G(\boldsymbol{z}))$ minus the mean of the real outputs $f(\boldsymbol{x})$; in simpler terms, by minimizing this loss the critic is trying to give high scores for the real data and low scores for fake data. On the other hand, the loss for the generator is simply the negative mean of the fake outputs $f(G(\boldsymbol{z}))$, by minimizing this loss the generator is trying to maximize the score that the critic gives to the data that it produces.

One detail that was left to the side until now is the condition that the critic must be a K-Lipschitz continuous function, this is essential in order for the distance approximation to be valid and it is something that is not restricted by normal implementations of neural networks. The solution proposed by the authors was *weight clipping*, that is, limiting the parameters of the critic to a interval $[-c, c]$ where c is a constant hyperparameter. This is rather an unrefined solution, even the authors recognized that "weight clipping is a clearly terrible way to enforce a Lipschitz constraint" (ARJOVSKY; CHINTALA; BOTTOU, 2017), but it was their best solution at the time and they encouraged further research to find a better way; and this improvement came in the form of a gradient penalty, in the next section this method will be explored further.

4.3.4 WGAN with Gradient Penalty

One drawback about the WGAN method is the hard restriction on the critic's parameters by way of weight clipping, this not only introduces a new hyperparameter c that determines the clipping interval, but also leads to optimization difficulties as shown by Gulrajani et al. (2017); they demonstrated that when training, the gradients can either explode or vanish if c was not carefully chosen, and that the critic is biased to much simpler functions.

Gulrajani et al. (2017) proposed a new method to restrict the critic without needing to resort to hard clipping of the parameters. They base their technique by showing that the 1-Lipschitz function f on Equation (55) has a gradient of norm equal to 1 almost everywhere under the distributions p and q. Using this information, the loss function for the critic can be regularized to penalize for gradients that have norm far from 1, thus the name WGAN with Gradient Penalization (WGAN-GP). Equation (60) shows the regularized loss for the critic.

$$J^{(C)} = \underbrace{\mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}}\Big[f(G(\boldsymbol{z}))\Big] - \mathbb{E}_{\boldsymbol{x}\sim p_{\boldsymbol{data}}}\Big[f(\boldsymbol{x})\Big]}_{\text{WGAN critic loss}} + \underbrace{\lambda\, \mathbb{E}_{\hat{\boldsymbol{x}}\sim p_{\boldsymbol{x}}}\Big[(\|\nabla_{\hat{\boldsymbol{x}}} f(\hat{\boldsymbol{x}})\|_2 - 1)^2\Big]}_{\text{Gradient Penalty}} \tag{60}$$

For this loss function, λ is a new hyperparameter that defines the strength of the penalty (the original paper suggested that $\lambda = 10$ was a good value that applied to many situations), the probability distribution $p_{\boldsymbol{x}}$ is a uniform probability in the line between one real sample $\boldsymbol{x}$ and one fake sample $G(\boldsymbol{z})$, and $\hat{\boldsymbol{x}}$ is a sample from this distribution. What this means is that the gradient penalty is calculated by taking, for each pair of real and fake samples, a random point in the linear interpolation between them, and calculating the gradient of the critic with respect to this interpolation. Since the penalty is calculated per individual pairs, the critic should not use batch normalization since that would interfere with the penalty value, Gulrajani et al. (2017) recommend using layer normalization as an alternative.

4.3.5 Other techniques

The experiments in this document also tried some other methods that will be described more briefly here.

4.3.5.1 One Sided Label Smoothing

Adding noise to labels is an old technique that has been proved useful in different areas of machine learning, in the context of GANs it would change the objective of the discriminator from assigning the values 0 for fake data and 1 for real data, to a smoothed version like 0.1 and 0.9 for fake and real respectively.

However, Salimans et al. (2016) have shown that smoothing the fake labels would cause problems for convergence in some areas of the probability distribution, so they recommending smoothing only the real label.

4.3.5.2 Upsampling Methods

The basic building block for upscaling the image in the generator for the DCGAN is the transposed convolution layer, however, Odena, Dumoulin, and Olah (2016) have shown that this type of upsampling causes checkerboard artifacts in the images and recommend instead using a normal bilinear or nearest neighbour upsampling, followed by a normal convolutional layer. The experiments in this document tried comparing these three types of upsampling for different GANs and datasets.

4.4 EVALUATING GANS

GANs are not only difficult to train, but also to evaluate. Consider for example two different GANs that produce the handwritten digits of MNIST and the desire is to compare them to see which one produces the more realistic or more varied results, how would one achieve such task?

Comparing two classifiers is relatively easy, the most natural way is just to observe their accuracy in the test dataset and see which one is higher. For GANs on the other hand this is not so simple, remember that Goodfellow, Pouget-Abadie, et al. (2014) proved that the optimal discriminator would be unable to distinguish between real and fake data, and thus would have an accuracy of 50%. But simply aiming for this value of accuracy would not work, since a discriminator that tosses a coin to decide would in fact produce the same value.

For many image generation applications, the most important aspect is to produce samples that are appealing to the human eye, so ultimately, a qualitative analysis made by people is very important. However, this is not a reliable measure of quality, Salimans et al. (2016) have observed that workers on Amazon Mechanical Turk would produce varied results and the simple act of giving feedback would influence their accuracy; the authors also noted that they could easily distinguish real from the fake data produced by their models trained on the CIFAR-10 dataset (over 95% accuracy), while the workers would make mistakes much more frequently (78.7% accuracy).

Another problem of human evaluation is the fact that it is not so easy for a person to detect the variation of the model, the generator might produce very good samples while in a mode collapse state. The challenge is to evaluate not only the quality of individual samples, but also the variation over many samples. This is very challenging problem that still is not completely solved, being an active area of research, with new metrics still being proposed – see for example (WANG et al., 2020) and (GUAN; LOEW, 2021). One yet

unmentioned advantage of the WGAN loss is that it can also be used to evaluate the models, the value of the EM distance has been shown to correlate with the quality of the image generated (ARJOVSKY; CHINTALA; BOTTOU, 2017).

Currently, two of the most popular metrics are the Inception Score (IS) and Fréchet Inception Distance (FID), while the IS has largely been replaced by FID, both show good correlation with the image quality and also with image variety. For this document these were the two metrics used to evaluate the models in the experiments.

4.4.1 Inception Score

Just like a classifier can be used to predict the class of the test dataset, it can also be used to predict artificially generated images. If the classifier has a good accuracy, then it would make sense for it to be able to classify generated data with high confidence, given that this data is similar to the real data. This is the idea behind the IS, bad samples would make a classifier be unsure of the right class, while good samples would produce very confident classifications.

In more defined terms, the value of how confident a classifier is that some input $\boldsymbol{x}$ belongs to a class y is the conditional probability of $p(y|\boldsymbol{x})$, the idea is to calculate this over all labels y, and this value ideally should have a clear spike at some y, representing a high confidence classification[3] (SALIMANS et al., 2016). By just measuring this value it would be possible to have an idea of the quality of the generated samples by the confidence of the classifier, but this alone would not be enough to detect the variety of the model, it is desired that the generator create samples in all classes, without giving preference to any particular label. This is equivalent of saying that $p(y)$ should be as uniform as possible[4] (ZHOU, 2021).

By combining these two ideas, the IS is calculated from the KL divergence between these two probability distributions as shown in Equation (61), note that the exponential operator is used only to make the numbers easier to interpret (SALIMANS et al., 2016).

$$\text{IS} = \exp\Big(\mathbb{E}_{\boldsymbol{x}\sim p_{model}}\big[D_{KL}(p(y|\boldsymbol{x}) \parallel p(y))\big]\Big) \tag{61}$$

Since the desired behaviour is for $p(y|\boldsymbol{x})$ to be very well defined at a single point while $p(y)$ should be completely uniform, then the KL divergence between those two should be as high as possible, this means that for the IS, higher values means better results. To calculate the expected value expected value it is necessary to average the results over many generated samples in order to have a close approximation of the true value, the original authors used $50,000$ in their experiments (SALIMANS et al., 2016).

The remaining detail is what classifier to use when calculating the IS, this metric uses the Inception v3 model (SZEGEDY; VANHOUCKE, et al., 2015), hence the name

[3] It is out of the scope of this document, but for those aware of the concept, the desired distribution should have low entropy

[4] Equivalent to high entropy

Inception Score. The Inception model is a very powerful model trained on 1000 classes of the ImageNet dataset of natural images, so it is very sensible as the classifier of choice. There are however some limitations, since the model was trained on ImageNet it may not offer good classification for datasets that are too different from natural images (like MNIST) or from the 1000 classes learned.

Other shortcomings of the IS are that it could give very high scores for models that generate only a single good image for each category (a form of mode collapse) and that it also completely ignores the training dataset (the very thing that the generator is trying to model) (ZHOU, 2021).

4.4.2 Fréchet Inception Distance

The FID metric was introduce as an improvement over the previous IS, it was shown to correlate well with human perception and be more consistent than the IS for different types of disturbances to the images (i.e. the FID consistently gets worse as images get more disturbed, while the IS fluctuates) (HEUSEL et al., 2017).

The Inception v3 model is also used to calculate this metric, but instead of using the class predictions from the output layer like the IS, the FID uses features from the last global pooling layer. To understand why this is a useful choice it is important to know how computer vision models see the data. The idea behind deep learning models is that each layer of the network can capture a different level of abstraction in the data, for example, the layers start detecting edges, followed then by shapes, textures, and finally high level patterns. The Inception v3 model takes as input $299 \times 299 \times 3$ images and reduces them to a 1024 dimensional vector of features, this is a compression of about 262 times the original size; with such aggressive reduction it is imperative for the model to learn only the most relevant features, these are able to describe the data in a much more abstract level without being affected by unimportant things like small random noise.

For a well behaved model like Inception v3, if two images share similar feature vectors, they are probably very similar in an abstract sense as well. Two images of cats will have similar features, even more so if the cats have similar colors, pose or fur. By using the feature layer to measure the FID it is possible to evaluate how well the model represents the structure of the data, and not punish it if it cannot reproduce the exact images of the training set.

The FID uses another metric for the difference between probability distributions called the Fréchet Distance, the difference is calculated between the distributions of the feature vectors for the real and fake images. For practical purposes only the mean and covariance are considered, so it is assumed that the distributions follow a multidimensional Gaussian, making the FID metric be calculated as shown in Equation (62) (HEUSEL et al., 2017).

$$\text{FID} = \|\boldsymbol{\mu}_{data} - \boldsymbol{\mu}_{model}\|_2^2 + \text{Tr}\Big(\boldsymbol{V}_{data} + \boldsymbol{V}_{model} - 2(\boldsymbol{V}_{data} \cdot \boldsymbol{V}_{model})^{\frac{1}{2}}\Big) \tag{62}$$

In this equation $\boldsymbol{\mu}$ and $\boldsymbol{V}$ represent the mean vector and the covariance matrix for the feature vectors, calculated for a big enough sample of the real and fake data (i.e. $50,000$). The Tr operator calculates the trace of the matrix (i.e. the sum of all the elements of the main diagonal), and the matrix square root inside the trace is not calculated element-wise, it is instead the actual square root of the entire matrix. Since this metric is a distance between real and fake data and it is desirable for this distance to be small, then for the FID, lower values means better results.

One of the main advantages of the FID is that it considers the training dataset in the evaluation, so a better model would be the one who can better replicate the structure of the data used to learn. This metric is considered better than the IS and has generally replaced it (ZHOU, 2021), however it still suffer from some of the same shortcomings, notably the fact that it only works for evaluating GANs in the image domain and that it still relies on the specifics of the Inception v3 model.

4.4.3 Using other classifiers

Although the original IS and FID rely on the Inception v3 model, the same metrics could be calculated using another classifier trained specifically for the data used in training the GAN model – for this document this will be called the Classifier Score (CS) and Fréchet Classifier Distance (FCD) respectively – this can produce more accurate results for datasets like MNIST that contain very different images from the ones in ImageNet used to train Inception v3.

One disadvantage of this approach is that it is not useful when comparing models evaluated with different classifiers, however it is always possible to fall back to the original IS and FID as a common ground for comparison, keeping in mind that these metrics may not be very accurate anyway depending on the dataset.

For the experiments in this document it was decided to train individual classifiers for each dataset in order to produce more accurate metrics, the comparisons with other published models was judged less relevant since the idea is to compare all the techniques experimented on, it falls off the main scope of this document to directly compare the results of other works. Also because for some of the older methods, the IS and FID metrics were not even introduced at the time.

5 EXPERIMENTS

This chapter describes the results of the experiments made and contains some remarks about them, the full conclusions will be presented later in Chapter 6. All generated images shown in this chapter are not cherry-picked and will always refer to the generator which produced the lowest FCD metric unless noted otherwise.

When calculating the CS and FCD metrics, the number of samples used was $200 \times 256 = 51,200$ for evaluating generators training with the MNIST and Fashion MNIST datasets, for the CIFAR-10 dataset this number was cut in half to $25,600$ due to memory constraints in the machine available (see Apendix A for specifications).

For all the GANs that use a discriminator instead of a critic (i.e. GAN, DCGAN and CGAN), the loss for the generator was not the one that minimizes the chance of the discriminator being right $\log(1 - D(G(\boldsymbol{z})))$, but the one that maximizes the chances of it being wrong $\log(D(G(\boldsymbol{z})))$. As discussed in section 4.2, this idea existed since the introduction of GANs (GOODFELLOW; POUGET-ABADIE, et al., 2014) and is empirically motivated by the fact that it produces more reliable gradients when the generator has not yet learned to create good results.

The following abbreviations will be used to refer to the values of the hyperparameters used in the experiments:

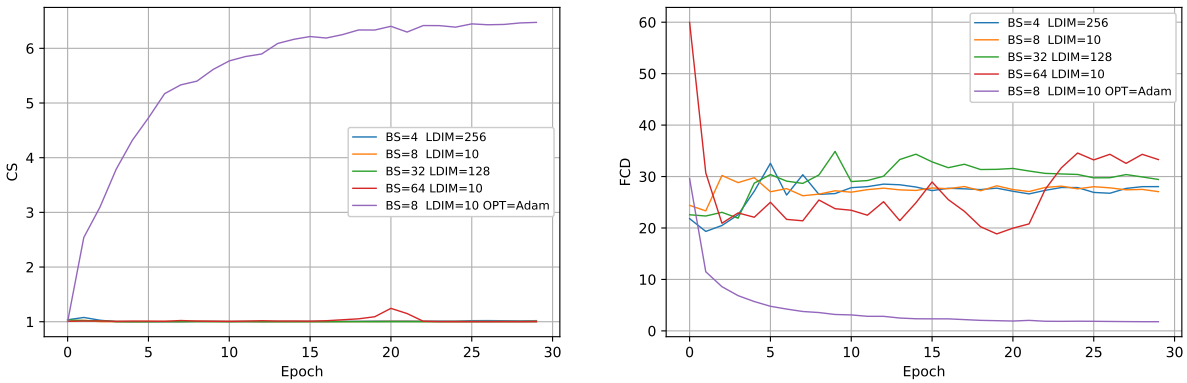
- Batch Size (BS)
- Batch Normalization (BN) – `Yes` if used in any manner; `No` otherwise
- Upscaling Method (UP) – `TrpConv` for transposed convolutions; `Bilinear` for bilinear upsampling; `Nearest` for Nearest Neighbour upsampling.
- Optimizer (OPT)
- Dimensions of latent space (LDIM)
- Clipping value for the parameters of the critic (CLIP)
- Number of updates to critic before update to generator (NCRIT)
- Value of one-sided label smoothing (SMOOTH) – If not present, then no label smoothing was used
- Momentum hyperparameter of Adam (β_1)
- Learning rate (η)

5.1 SIMPLE GAN

The original GAN was able to produce some promising results at the time, but was relatively simple and did not had any of the architectural guidelines that were later introduced by the DCGAN. This experiment uses a simple generator with two fully connected layers that map the latent vector to the resulting image.

For this simple case the network was only trained on the MNIST dataset, since it struggled to produce good results even in this easy scenario. One of the results was already seen in Figure 21, where it was shown how this network had extreme mode collapse producing only the number one. Figure 23 shows how the metrics evolved for the different hyperparameters used.

Figure 23 – Metrics when training a simple GAN on MNIST



Source – From the author (2021)

Recall that for the CS, bigger values means better, and the opposite for FCD. Note how all choices of hyperparameters resulted in terrible values when using the SGD with momentum optimizer, all of these cases suffered the same extreme mode collapse seen on Figure 21, even collapsing to the same digit. However, note how just changing to the Adam optimizer significantly increased the performance, this optimizer generally produces good results (GOODFELLOW, 2017) and is one of the key components proposed in the DCGAN architecture. Given this, all following experiments will be using the Adam optimizer unless otherwise noted. Figure 24 shows samples generated from the GAN trained with Adam.

5.2 DCGAN

The DCGAN architecture was a big step for GANs when it was introduced, it defined a set of recommendations for building networks that would be more stable and scalable, by fully leveraging the power of convolutional layers in both generator and discriminator (RADFORD; METZ; CHINTALA, 2015). The architecture of the DCGAN was used as a basis for all following experiments in this document.

Figure 24 – Samples when training a simple GAN on MNIST

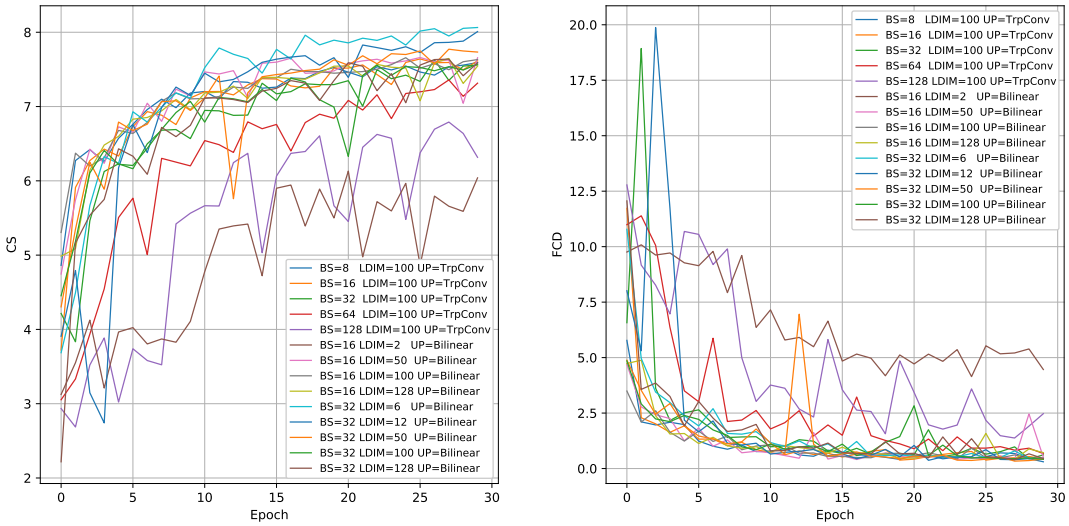
Source – From the author (2021)

The goal of the tests in this section was to try and find what properties of the DCGAN were relevant, what configurations could be used to tackle different situations, and to question the validity of using transposed convolutions has a way of upscaling the latent vector to the full resolution of the image, since it can produce artifacts (ODENA; DUMOULIN; OLAH, 2016) and it is being replaced in recent approaches (e.g. (KARRAS; LAINE; AILA, 2018)) by more conventional upsampling techniques followed by a simple dimension preserving convolution.

5.2.1 MNIST

Training the DCGAN on the MNIST dataset produced the results seen in Figure 25.

Figure 25 – Metrics when training a DCGAN on MNIST

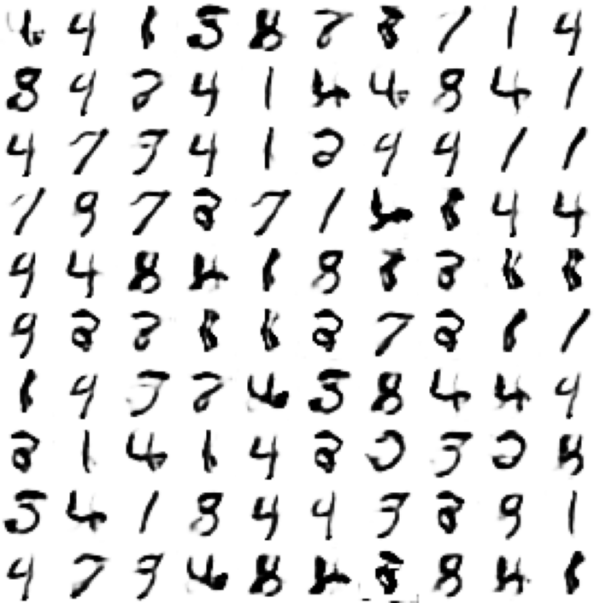


Source – From the author (2021)

There are a couple of things that can be observed in these results. First, observe how the number of dimensions in the latent space does not seem to be very relevant, unless it becomes too small, as seen for the case of `LDIM=2`, which produced the worst results. This makes sense since it is harder for the generator to learn a map from the smaller space to all possible images, in other words, its capacity is too small to produce the complexity in the data.

The effect of this can be seen in Figure 26, these are samples produced by the generator with only two dimensions of latent space. The samples were chosen in a training epoch where it is particularly clear that the generator is producing some relatively good samples, but it suffers to map the small volume of the latent space into all possible digits, resulting in mode collapse.

Figure 26 – Samples taken from a DCGAN with low dimensions of latent space



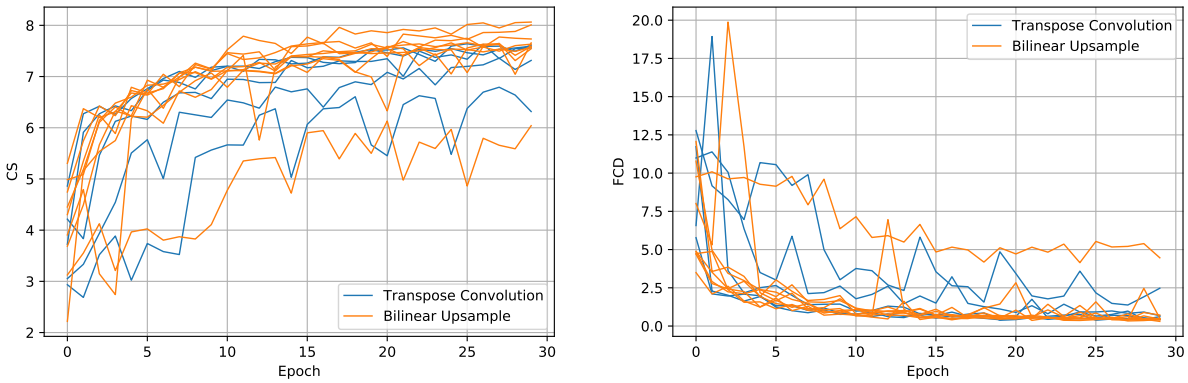
Source – From the author (2021)

But note that by just raising the number of dimensions from 2 to 6 is already enough to fix this problem, after that, the number of increased dimensions has little effect, since the generator already has all the capacity it needs to represent the simple data from the MNIST dataset.

The results in Figure 25 cannot give a definitive answer about the influence of the batch size, although there is some indication that bigger batches may not be ideal. This is seen for the case with batch size of 128 that produced the second least favorable results, and in a lesser extent for the test with batch size of 64.

Last point of note is the effect of the upsampling technique used, to better visualize this it is helpful to group the metrics shown in Figure 25 into the particular sets of interest, this approach will be used repeatedly for the rest of this document. Figure 27 highlights the results per type of upsampling used.

Figure 27 – Effects of upsampling when training a DCGAN on MNIST



Source – From the author (2021)

It can be seen by this figure that the bilinear upsampling technique generally performs better than transposed convolutions in the case of MNIST, as later results will show, this does not imply that it will always be the case.

Figure 28 shows samples produced by the best model experimented in these tests, with configuration (`BS=32 LDIM=12 UP=Bilinear`).

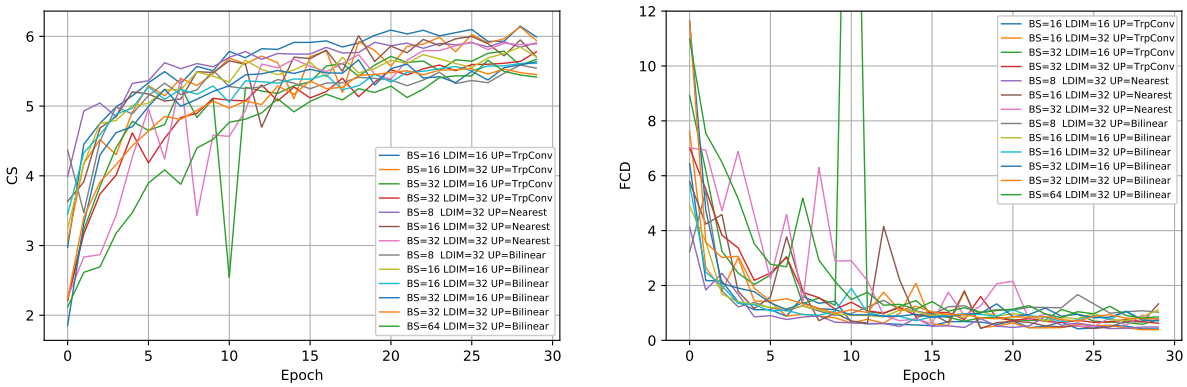
Figure 28 – Samples when training a DCGAN on MNIST

Source – From the author (2021)

5.2.2 Fashion MNIST

Training the DCGAN on the Fashion MNIST dataset produced the results seen in Figure 29.

Figure 29 – Metrics when training a DCGAN on Fashion MNIST



Source – From the author (2021)

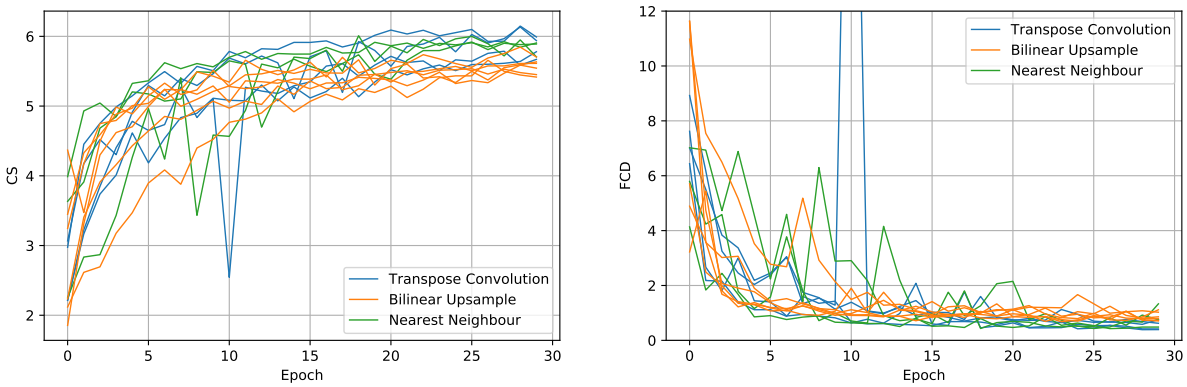
Again the use of bigger batches has produced weaker results, with the case for a batch size of 64 producing the least favorable overall results between all tests. This may be for the fact that smaller batch sizes have more updates per epoch since they divide the dataset into more batches and each batch is a parameter update. However, at least for the case of the batch size of 128 seen in Figure 25, bigger batches seem to have more difficulty to step beyond a certain point. It may be that bigger batches would produce better results given more epochs of training, and they are a small amount faster to calculate per epoch in relation to smaller batches, but for the rest of the tests they were not evaluated more deeply.

One thing to note in Figure 29 is the extreme jump seen on test (`BS=32 LDIM=16 UP=TrpConv`) at epoch number 10, this can be seen to a smaller extent in many cases throughout the following experiments, but this was the most extreme one. It is not clear why this happens, the author's hypothesis is that the generator stepped into a particular steep region of the loss surface, but was quickly pointed in the right direction by the larger gradients of the discriminator in the next updates.

Again, by highlighting the use of the upsampling techniques it is possible to obtain the results seen in Figure 30. In this case, the transposed convolutions showed the best results, while bilinear upsampling had the least favorable overall. The nearest neighbour interpolation was also tested in this case and here it also performed better than bilinear.

Lastly, the samples from the lowest FCD obtained by the test (`BS=16 LDIM=32 UP=TrpConv`) can be seen in Figure 31. Note how the generator does not have a good sense of symmetry, producing shirts with long sleeves in a single arm. This is a very common occurrence in GANs, even modern models will suffer from this; for example, GANs can produce faces where the eyes do not have the same color, or place different earrings in each ear (MCDONALD, 2018).

Figure 30 – Effects of upsampling when training a DCGAN on Fashion MNIST



Source – From the author (2021)

Figure 31 – Samples when training a DCGAN on Fashion MNIST

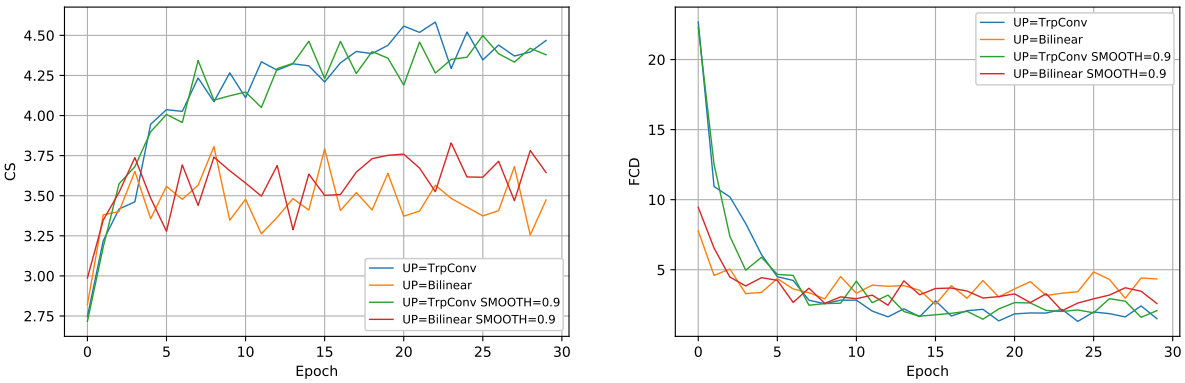
Source – From the author (2021)

5.2.3 CIFAR-10

Figure 32 shows the results of training the DCGAN on the CIFAR-10 dataset. One thing to note here is how both the CS and FCD metrics show weaker results when compared to the previous results.

When making the experiments, it was found that the CIFAR-10 dataset is considerably harder than the other datasets, even when compared with the CelebA and Flowers datasets. The author's hypothesis is that this is due to the relatively low number of images for each class in CIFAR-10, (only 6000 for each), and that the classes are significantly different from each other, making it very hard for the generator to learn the details in all of them.

Figure 32 – Metrics when training a DCGAN on CIFAR-10



Source – From the author (2021)

Recall that the classes in CIFAR-10 are: `airplane`, `automobile`, `bird`, `cat`, `deer`, `dog`, `frog`, `horse`, `ship` and `truck`. These images are significantly more varied than the other datasets used, even the dataset of human faces, CelebA, can later be seen to be relatively homogeneous.

With these results it is possible to see once again a clear difference between upscaling techniques. This time, in accordance with what was seen for Fashion MNIST, the transposed convolution gives the best results. In these tests it was also experimented the use of one sided label smoothing, although the results still do not show any significant difference that allows for making conclusions.

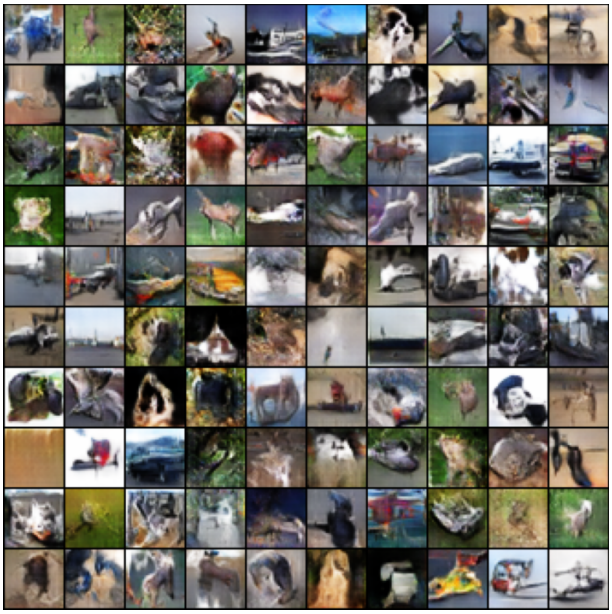
One important thing to note in all the tests on CIFAR made for this document, is the fact that the momentum term of the Adam optimizer (β_1) had a negative impact on the convergence of the models. For MNIST and Fashion MNIST this term was kept at the common value of 0.9 without any problems, however this would result in incomprehensible images and no sort of convergence when training on CIFAR-10, to avoid this, β_1 was kept at zero.

The samples produced for the best overall FCD in the test (`UP=TrpConv`) are shown in Figure 33. Note how it is possible to see a hint of the CIFAR-10 classes in these images, but nothing can be pointed out as a completely sensible real world object.

5.3 CGAN

To implement a CGAN it is only necessary to modify an existing GAN to receive a conditional label. For these experiments, the same base used for the DCGAN was used to construct the CGAN, the only difference was the addition of another input for the label that will pass through an embedding layer and be incorporated into a channel of the other input, as discussed in subsection 4.3.2.

Figure 33 – Samples when training a DCGAN on CIFAR-10

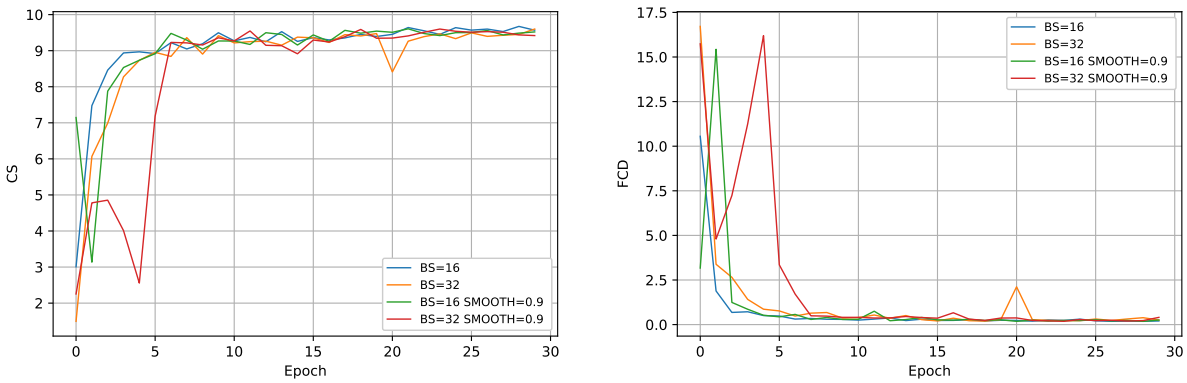


Source – From the author (2021)

5.3.1 MNIST

Training the CGAN on the MNIST dataset produced the results seen in Figure 34.

Figure 34 – Metrics when training a CGAN on MNIST



Source – From the author (2021)

These results show not only that the metrics have improved, but also that the training is much more stable, for all tests the training produced similar good results.

One of the main benefits of the CGAN architecture is the fact that the resulting sample can be controlled by feeding the desired label to the generator. This can be seen in the samples produced by the best model (`BS=16 SMOOTH=0.9`) shown in Figure 35, where each row was conditioned to produce a different set of digits.

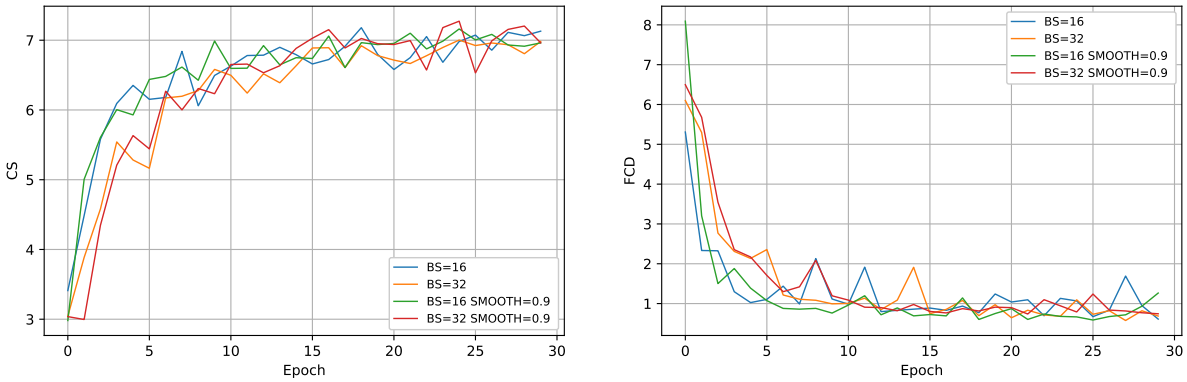
Figure 35 – Samples when training a CGAN on MNIST

Source – From the author (2021)

5.3.2 Fashion MNIST

Training the CGAN on the Fashion MNIST dataset produced the results seen in Figure 36.

Figure 36 – Metrics when training a CGAN on Fashion MNIST



Source – From the author (2021)

The same behaviour seen for MNIST can also be seen here, the overall quality increased, this time however the level of stability is not as strong. The samples produced by the best test (`BS=32`) are shown in Figure 37.

5.3.3 CIFAR-10

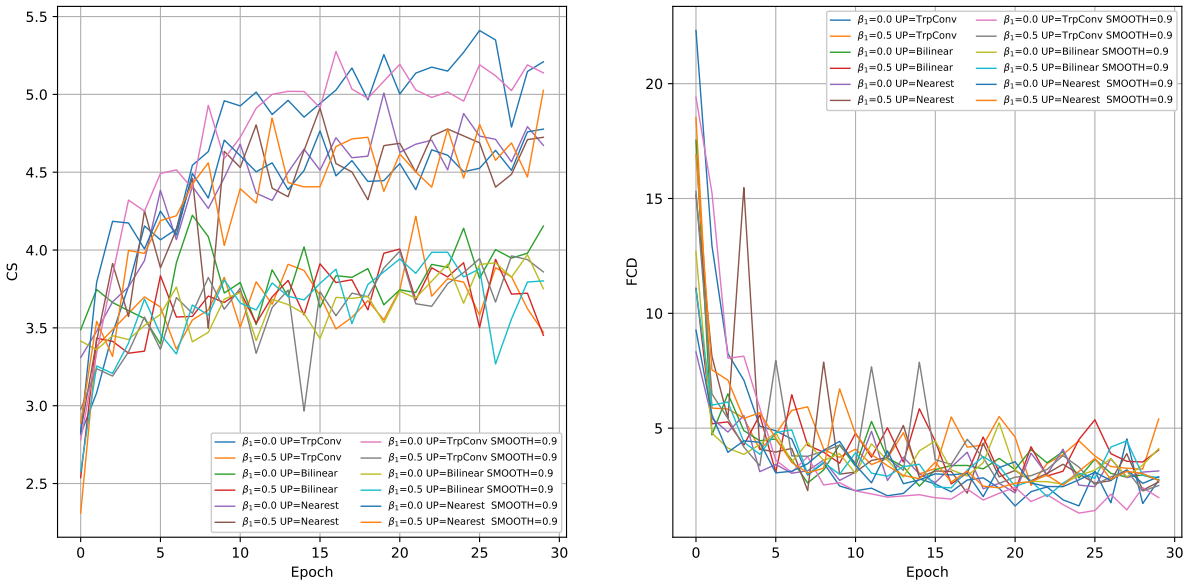
Training the CGAN on the CIFAR-10 dataset produced the results seen in Figure 38. In these experiments it was also tested the β_1 term of Adam to see if some momentum in

Figure 37 – Samples when training a CGAN on Fashion MNIST

Source – From the author (2021)

the updates would be beneficial.

Figure 38 – Metrics when training a CGAN on CIFAR-10

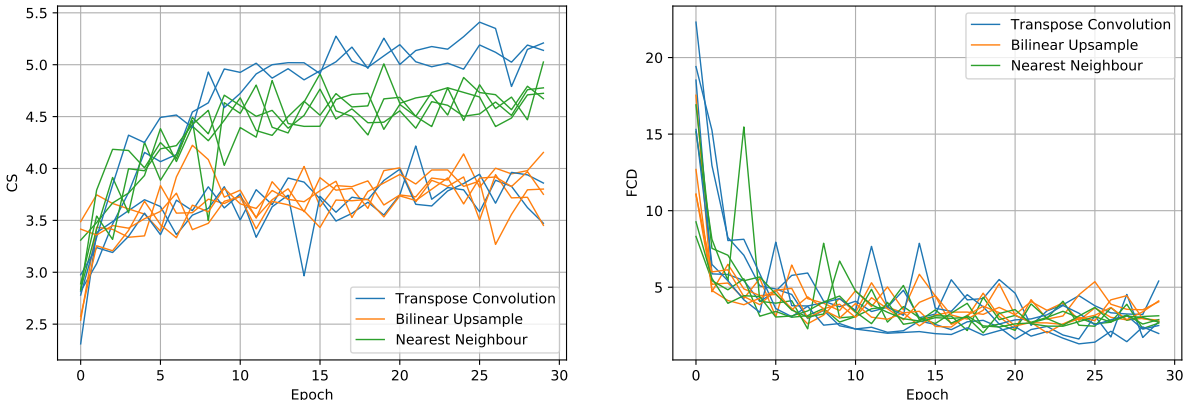


Source – From the author (2021)

To better understand how each component affects the results it is useful to highlight them separately. Figure 39 shows how the metrics evolve for the different upsampling techniques. The same behaviour observed previously is also seen here, still the transposed convolutions perform better, followed by nearest neighbour and bilinear upsampling.

In Figure 40 it is possible to see the effect of the momentum term β_1 in training.

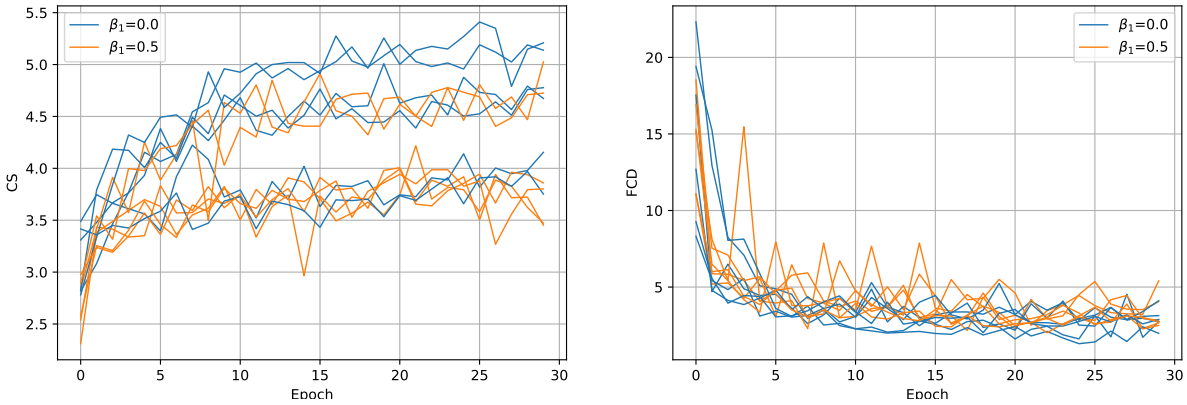
Figure 39 – Effects of upsampling when training a CGAN on CIFAR-10



Source – From the author (2021)

The results support the idea that momentum does not help for CIFAR-10. Also note how the bad performing cases of transposed convolutions seen in Figure 39 are because the use of momentum.

Figure 40 – Effects of momentum when training a CGAN on CIFAR-10



Source – From the author (2021)

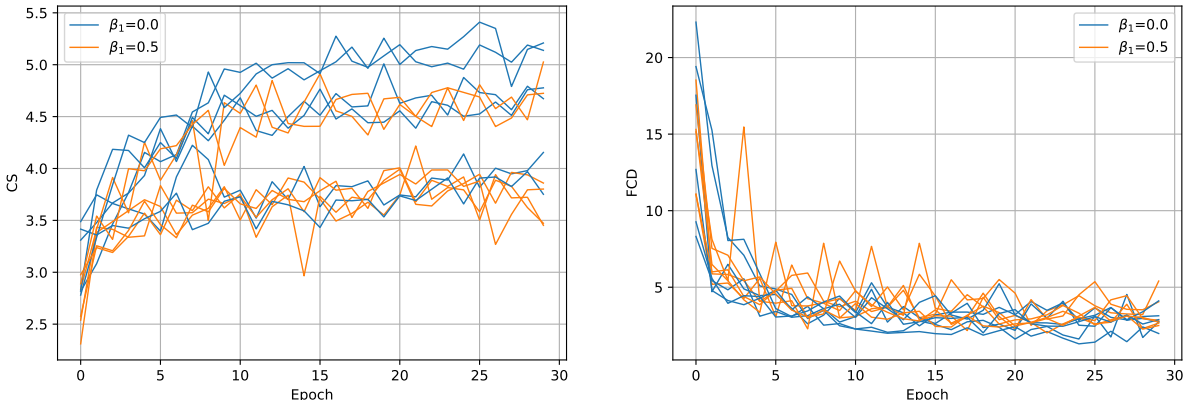
Lastly, the effects of label smoothing can be seen in Figure 41. The impact is still not very significative, although it can be seen a small tendency for better values of the FCD.

The samples produced by the best test (β_1=`0.0 UP=TrpConv SMOOTH=0.9`) are shown in Figure 42.

5.4 WGAN

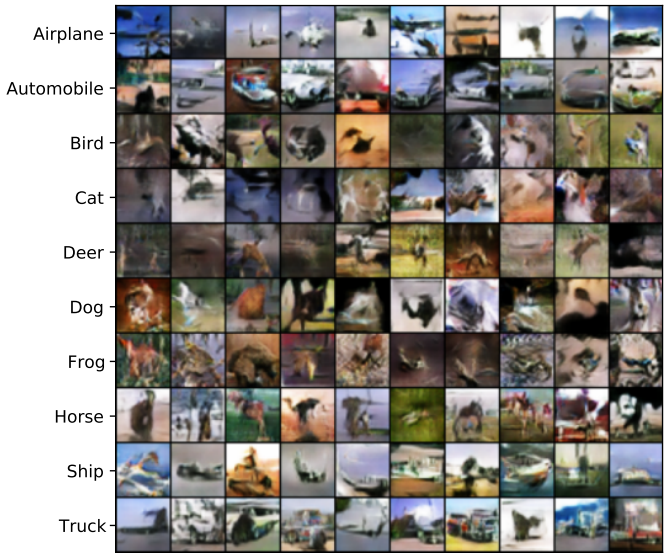
For building the WGAN, the generator architecture can be the same as the DCGAN, the critic can keep the overall structure of the discriminator, but it must have the weight clipping constraints to its parameters. Besides this, the loss function of both the generator and critic must be replaced by the Wasserstein loss as described in subsection 4.3.3.

Figure 41 – Effects of label smoothing when training a CGAN on CIFAR-10



Source – From the author (2021)

Figure 42 – Samples when training a CGAN on CIFAR-10



Source – From the author (2021)

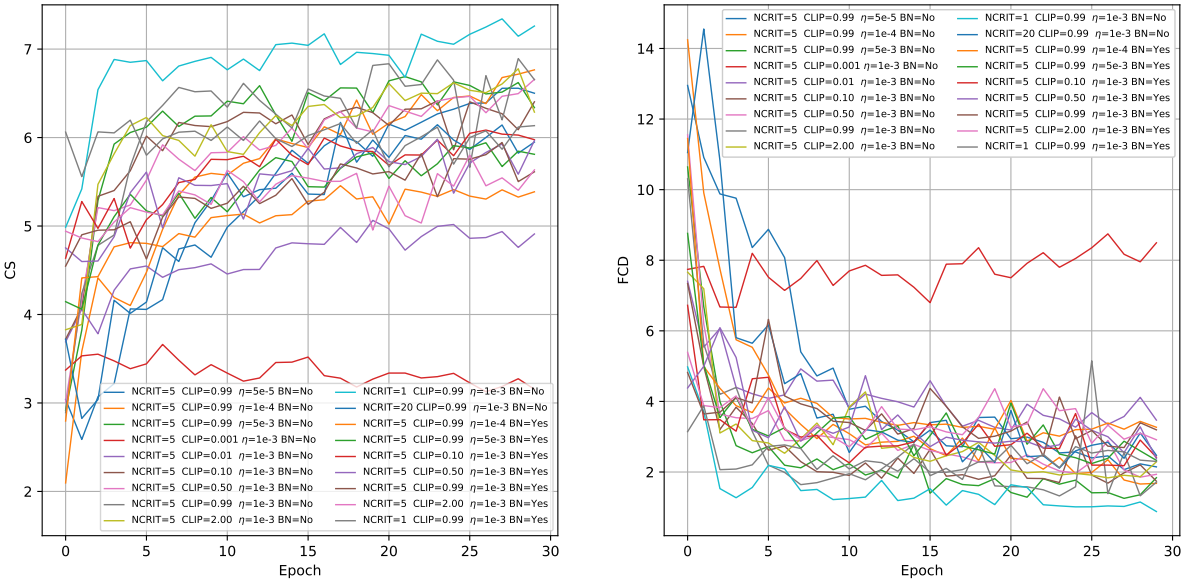
As the results will show, this type of GAN was the hardest to produce good results, many experiments were made in order to find viable hyperparameters, but it still was unable to produce comparable results to the other techniques. Also, per recommendation of the original authors (ARJOVSKY; CHINTALA; BOTTOU, 2017), the RMSProp optimizer was used in these experiments as the momentum of the Adam optimizer may hurt convergence.

5.4.1 MNIST

Training the WGAN on the MNIST dataset produced the results seen in Figure 43. These results show a high variability on training with different hyperparameters, and none of them were able to reach similar results with previous techniques.

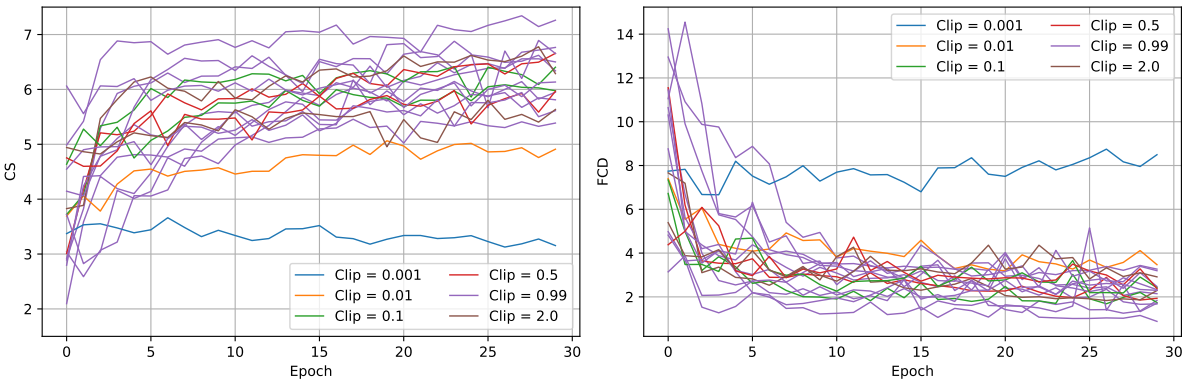
In order to better see the impact of the hyperparameters chosen it is helpful to make different highlights of these results. Figure 44 shows how different clipping values influenced the results of the tests.

Figure 43 – Metrics when training a WGAN on MNIST



Source – From the author (2021)

Figure 44 – Effects of clipping value when training a WGAN on MNIST



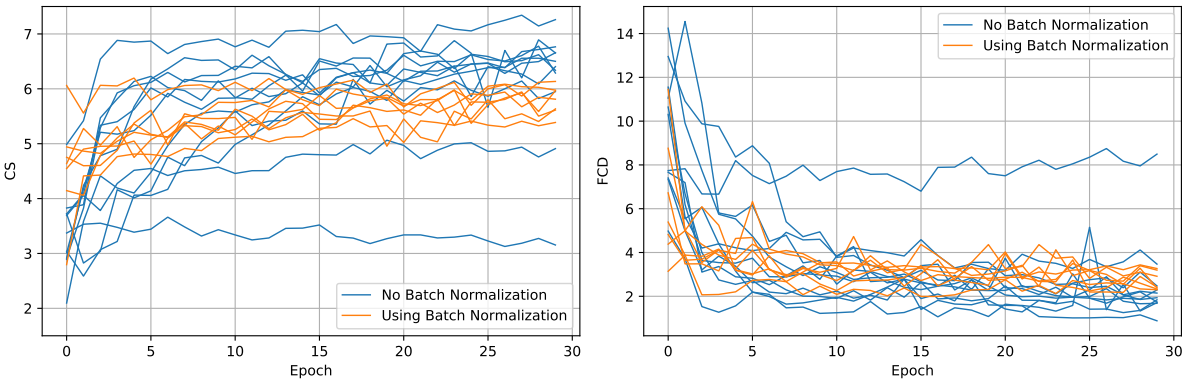
Source – From the author (2021)

In this case, a clipping value of 0.99 showed the best results, but even the relatively large range of 0.1 to 2.0 still produced similar results overall. The performance only started to decrease for smaller clipping values, this is surprising, since in the original paper proposing WGANs it was recommended a value of 0.01 for clipping (ARJOVSKY; CHINTALA; BOTTOU, 2017). The reason for this may be that the authors trained their models on colored images with higher resolutions and that for these cases, smaller clipping

values are beneficial. Whatever the reason, the recommended value produce less favorable results and more experiments would be needed in order to give a more definitive answer.

The use of batch normalization can also be analysed, its effect can be seen on the highlighted results in Figure 45. For this case, not using batch normalization has shown better results.

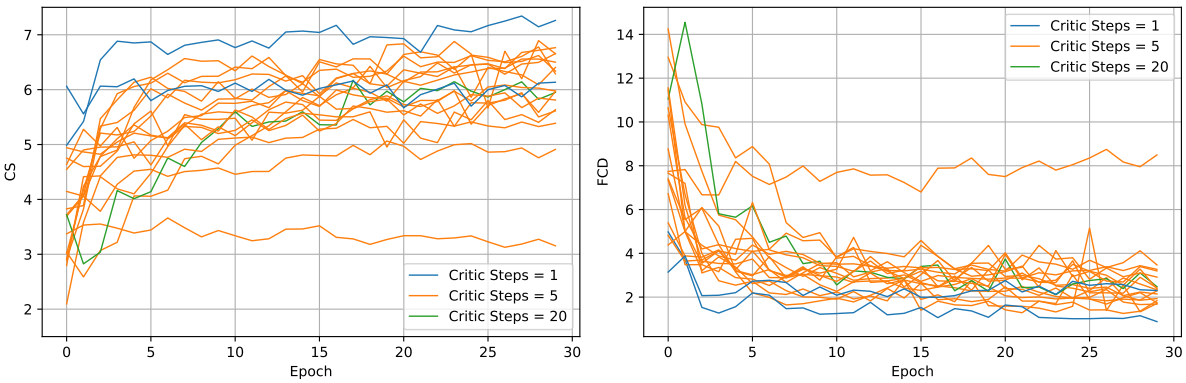
Figure 45 – Effects of batch normalization when training a WGAN on MNIST



Source – From the author (2021)

Lastly, the number of critic updates per generator update can be considered. In their original proposal, Arjovsky, Chintala, and Bottou (2017) argued that training the critic for multiple iterations is something that should be done, since it would only produce more reliable gradients from the critic. However the results shown in Figure 46 question this idea.

Figure 46 – Effects of number of critic iterations when training a WGAN on MNIST



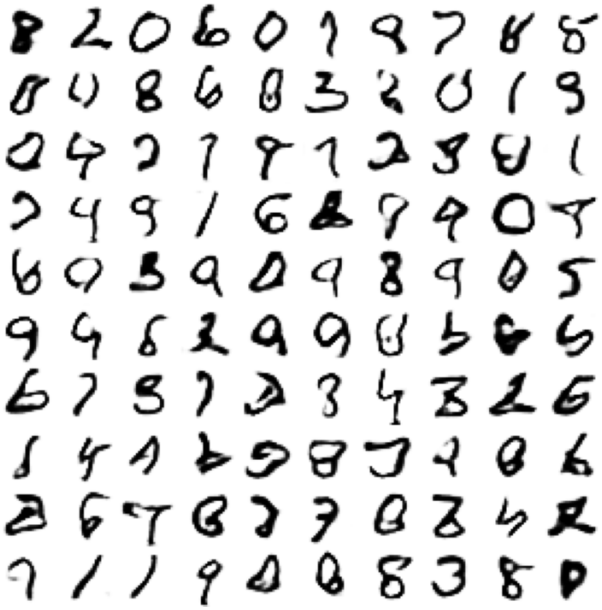
Source – From the author (2021)

For the experiments made, the actual best policy was to train the critic just as much as the generator. In fact, training for more iterations seemed to reduce the performance, as the test for 20 critic iterations had one of the lowest performances, beating mainly the methods that were hurt by the use of batch normalization and low clipping values.

One of the reasons for this is similar to the argument made for the batch sizes, just as bigger batches make fewer updates to the networks per epoch, so do larger number of critic iterations. Since the critic must be updated multiple times in order to update the generator once, then more iterations will mean that the generator is updated less. This may give more reliable gradients, but the cost is that the training becomes significantly slower; one alternative could be using fewer iterations when the training starts and the gradients do not need to be very precise, while gradually increasing them as training progresses.

The samples produced for the best model (`NCRIT=1 CLIP=0.99` η`=1e-3 BN=No`) in the experiments are shown in Figure 47.

Figure 47 – Samples when training a WGAN on MNIST



Source – From the author (2021)

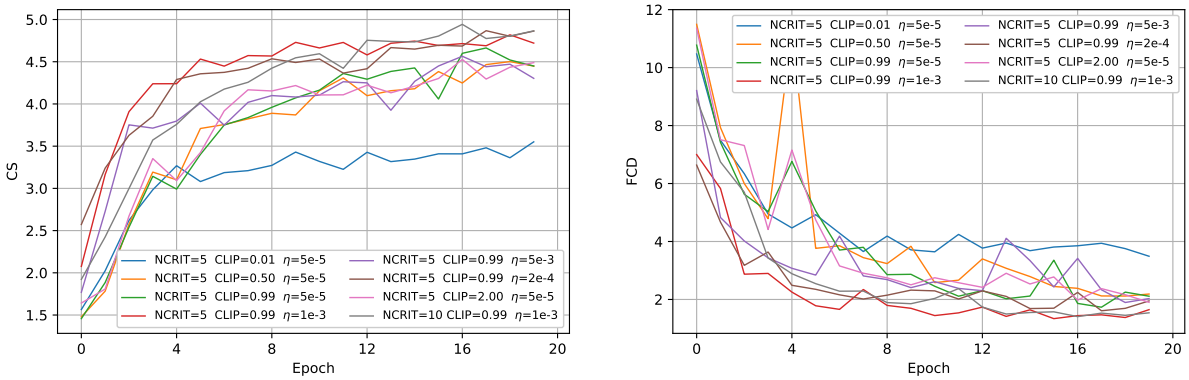
5.4.2 Fashion MNIST

Training the WGAN on the Fashion MNIST dataset produced the results seen in Figure 48.

The clipping values in these results show the same behaviour as seen for the MNIST case. However, the effect of the number of iterations for the critic does not seem too strong in this case, it may be because the values tested are relatively closer together, but more tests would be necessary to draw a conclusion.

The learning rate (η) was also experimented on with these and the previous tests with MNIST, this hyperparameter is usually something that needs some tuning since it can vary significantly in different types of problems, so it is hard to say anything definitive about it. For the MNIST case, highlighting the results by learning rate did not produce

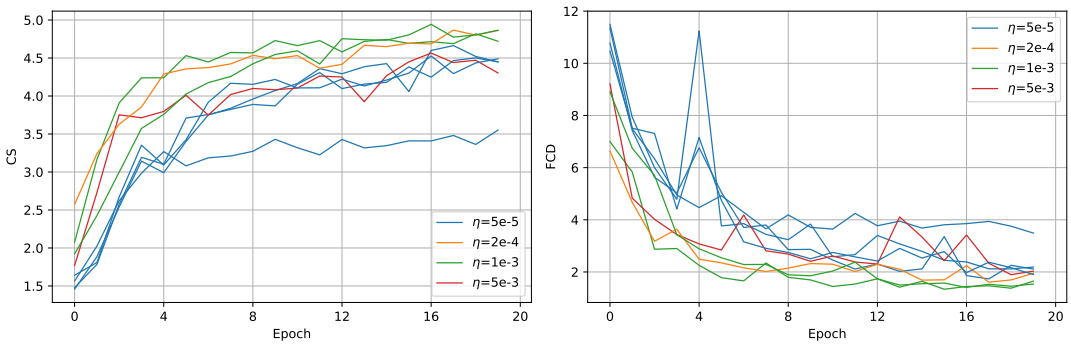
Figure 48 – Metrics when training a WGAN on Fashion MNIST



Source – From the author (2021)

very relevant information. For this case, the results may only provide simple insights, but for completion sake they are shown here in Figure 49.

Figure 49 – Effects of learning rate when training a WGAN on Fashion MNIST



Source – From the author (2021)

The samples generated by the best model for these tests (`NCRIT=5 CLIP=0.99` η`=1e-3`) are shown in Figure 50.

5.5 WGAN-GP

To create a WGAN-GP the architecture of the WGAN can be modified to remove the weight clipping constraints from the critic, and its loss can be changed to have a gradient penalty term as described in subsection 4.3.4. The gradient penalty should offer a better alternative to weight clipping and in turn produce better results.

5.5.1 MNIST

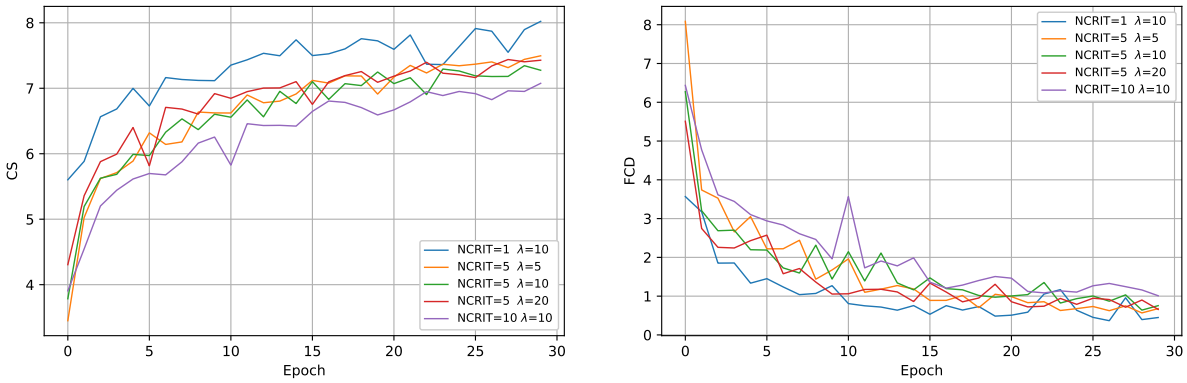
Training the WGAN-GP on the MNIST dataset produced the results seen in Figure 51. Observe how the overall value of the metrics has improved significantly when

Figure 50 – Samples when training a WGAN on Fashion MNIST

Source – From the author (2021)

compared to the WGAN approach.

Figure 51 – Metrics when training a WGAN-GP on MNIST



Source – From the author (2021)

The same behaviour related to the number of iterations for the critic can be seen, that is, more critic iterations tend to perform less well. The λ term, which regulates the strength of the gradient penalty, seems to produce similar results for all values tested. Contrary to the case seen for the WGAN, this actually agrees with the recommended value of 10 given by the authors of the WGAN-GP paper (GULRAJANI et al., 2017).

The samples produced by the overall best model (NCRIT=1 λ=10) are shown in Figure 52.

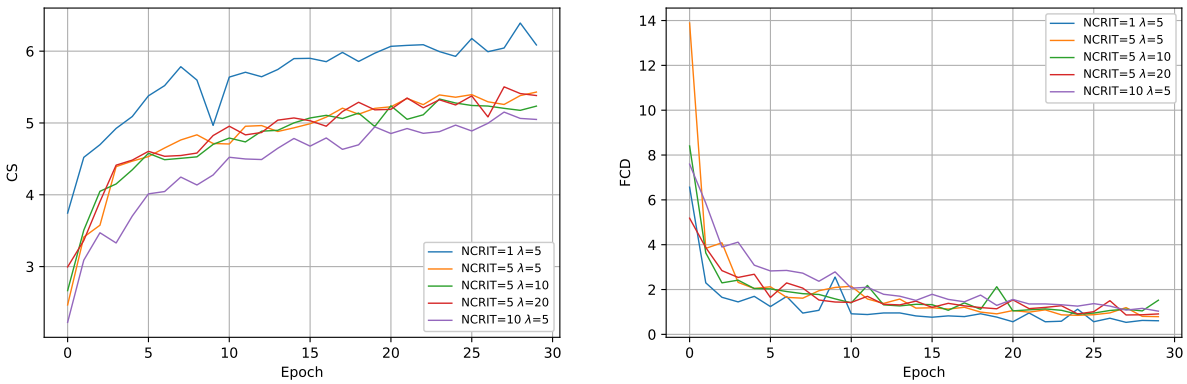
Figure 52 – Samples when training a WGAN-GP on MNIST

Source – From the author (2021)

5.5.2 Fashion MNIST

Training the WGAN-GP on the Fashion MNIST dataset produced the results seen in Figure 53. This gives very similar results and the same conclusions that were seen for the MNIST experiments. Figure 54 shows the samples produced by the best model (`NCRIT=1` λ`=5`).

Figure 53 – Metrics when training a WGAN-GP on Fashion MNIST



Source – From the author (2021)

Figure 54 – Samples when training a WGAN-GP on Fashion MNIST

Source – From the author (2021)

5.5.3 CIFAR-10

For these experiments it was used a value of 10 for λ since it followed the recommendation of the original authors and has also proved to work well in all cases tested. Training the WGAN-GP on the CIFAR-10 dataset produced the results seen in Figure 55.

Once again CIFAR-10 proves itself considerably harder to train, these results show a higher degree of instability and also produced significant less favorable results when compared to DCGAN and CGAN, except for the case which used `NCRIT=1`. This still agrees with the previous results and shows how consistent this technique is, making hyperparameter search easier.

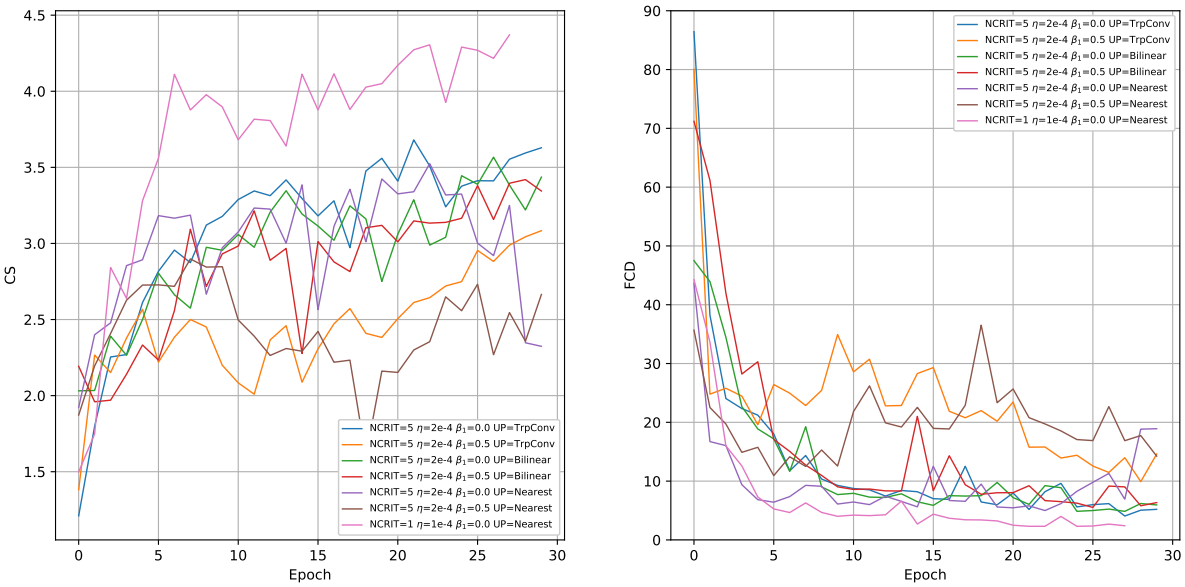
The best resulting model (`NCRIT=1` η`=1e-4` β_1`=0.0 UP=Nearest`) was able to produce the samples shown in Figure 56.

5.6 COMPARISON BETWEEN NETWORKS

Lastly it is possible to see how each architecture compares with one another in the experiments made. This was done here by taking the three best results for each architecture in each dataset and highlighting their metrics accordingly. Figure 57 shows the resulting comparison.

The best model shown here by far is clearly the CGAN, however, as Goodfellow (2017) puts it, this is an unfair comparison. It is important to note that this model has

Figure 55 – Metrics when training a WGAN-GP on CIFAR-10



Source – From the author (2021)

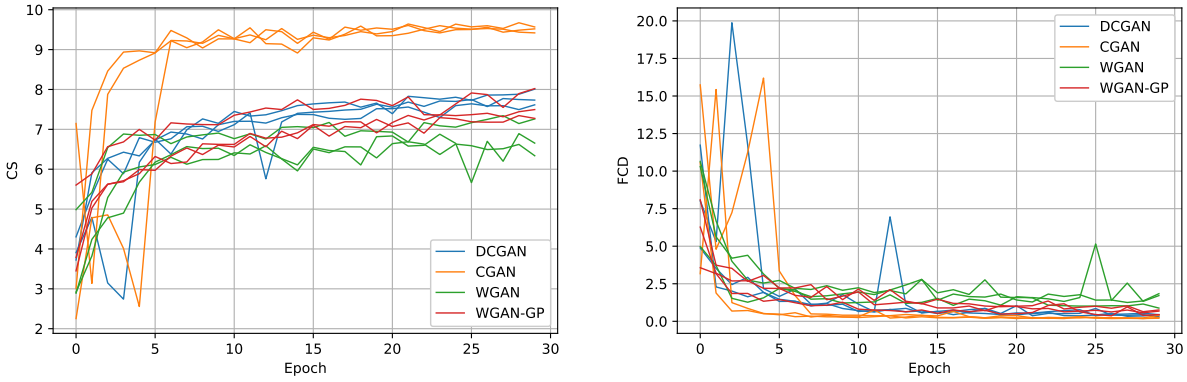
Figure 56 – Samples when training a WGAN-GP on CIFAR-10

Source – From the author (2021)

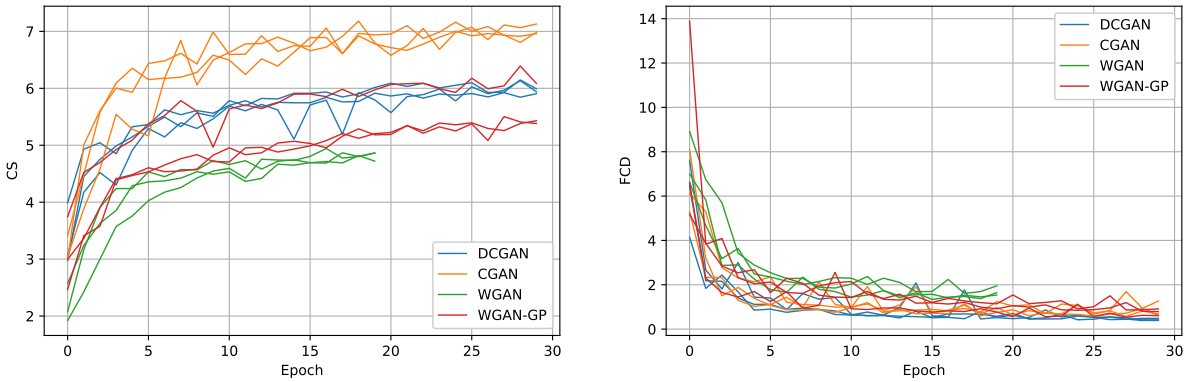
extra information in which to base its output, the quality of the results do not come from a cleverly built network, but by the data itself. This is not a useful way of evaluating the architecture on its own.

It is also important to mention that the CGAN implemented for this document is just a DCGAN with label conditioning, the same could be made for the WGAN and WGAN-GP. Figure 57 includes the CGAN only to give an idea of how much labels can help, but the important comparisons are between the other three models.

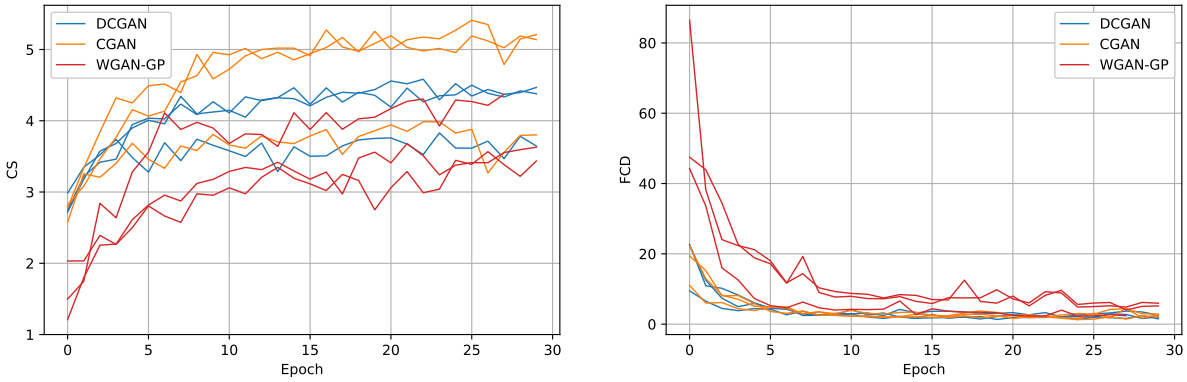
Figure 57 – Comparison of different GAN architectures performances



(a) MNIST



(b) Fashion MNIST



(c) CIFAR-10

Source – From the author (2021)

From the remaining models the WGAN showed the weakest results, this may not necessarily be the case for all tests, since Arjovsky, Chintala, and Bottou (2017) showed good results in the high resolution LSUN Bedrooms dataset with this architecture. More tests would be ideal in order to make more definitive statements, but the results seen in these experiments weren't able to reproduce performances similar to the ones for DCGAN or WGAN-GP, and they also showed a moderate level of instability in the choice of

hyperparameters.

The problem of WGAN is mostly due to the implementation of weight clipping, the WGAN-GP offers a better way of maintaining the Lipschitz continuity without losing the good properties of the Wasserstein loss. It showed similar results to the DCGAN for all datasets, but it has the added benefit of being more stable, the hyperparameter search did not need to be so expansive.

5.7 OTHER EXPERIMENTS

After obtaining a better understanding of how GANs work from the previous experiments, the next goal was to try and apply them to the Flowers and CelebA datasets in order to see if the knowledge could be transferred to these situations and if it would be possible to produce good results.

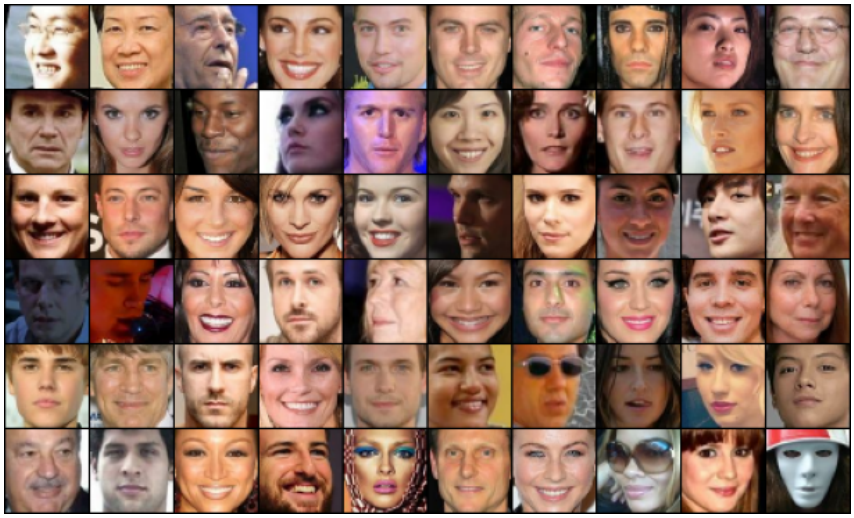
However, the images in these datasets were of too high resolutions for processing using the available machine, knowing this, they both were reduced to the more manageable resolutions of 48×48. The Flowers dataset was created in such a way that the shorter dimension has always 500 pixels, using this fact, the longer dimensions were center cropped to be of the same size and the resulting images were scaled down to the desired lower resolution. Figure 58 shows some samples in this reduced dataset.

Figure 58 – Reduced Flowers dataset

Source – From the author (2021)

The CelebA dataset has all images of height 218 and width 178, besides this, all images have the faces centered in the approximately same spot. Using this fact, the central part of the face was cropped, the horizontal pixels ranged from 41 to 137 and the vertical ones ranged from 85 to 181 (count starts from 0). Theses values were selected to best match the face position and the cropping resulted in 96×96 images, which were then downscaled to the desired resolution. Figure 59 shows samples from this reduced dataset.

Figure 59 – Reduced CelebA dataset



Source – From the author (2021)

These reduced datasets were use to test another three models, however, the CS and FCD metrics were not calculated for these experiments. Calculating the metrics would often cause memory problems even when evaluating models trained on CIFAR-10, evaluating the CelebA models would be impossible given the size of the dataset.

There was an attempt to use the Inception v3 model to calculate the IS and FID for the CelebA models, although it was possible to find a solution that would not cause memory issues, it was significantly slower and the results were completely senseless. Maybe a deeper familiarity with Tensorflow would produce a viable solution, but for this case the metrics were ignored and the results are only to be evaluated qualitatively.

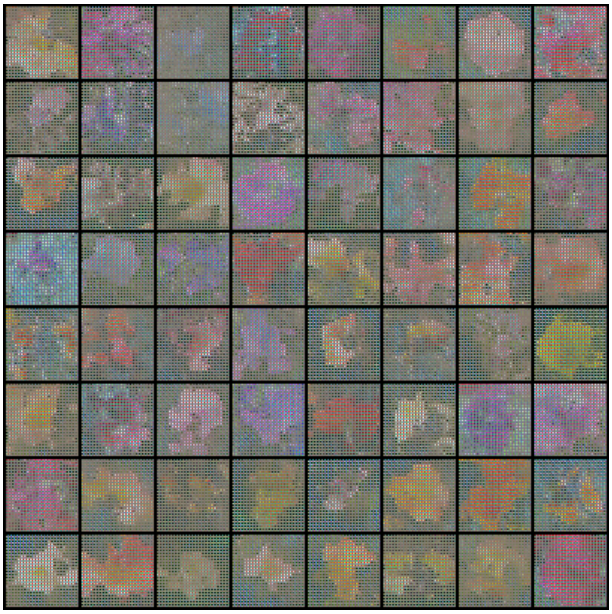
5.7.1 Flowers with DCGAN

Since the Flowers dataset contained only a small number of images, it was relatively quick to test different configurations and see if the results were promising. It was found that batch normalization did not produce good results, and that only nearest neighbour upsampling was a valid upscaling technique.

This experiment showed the most extreme cases of checkerboard artifacts that can occur when using transposed convolutions, no set of hyperparameters that could avoid this was found, even following the recommendations given by Odena, Dumoulin, and Olah (2016) did not proved fruitful. Figure 60 shows what the generator has produced after 20 epochs of training, it is even possible to see a hint of flowers behind the grid artifact.

Bilinear upsampling would create another pattern, the images generated would be oddly smooth everywhere. The generator was able to create some basic representation, by looking at a distance it is even possible to see a hint of the images being generated, but looking closely only very circular shapes are seen. Figure 61 shows samples from the

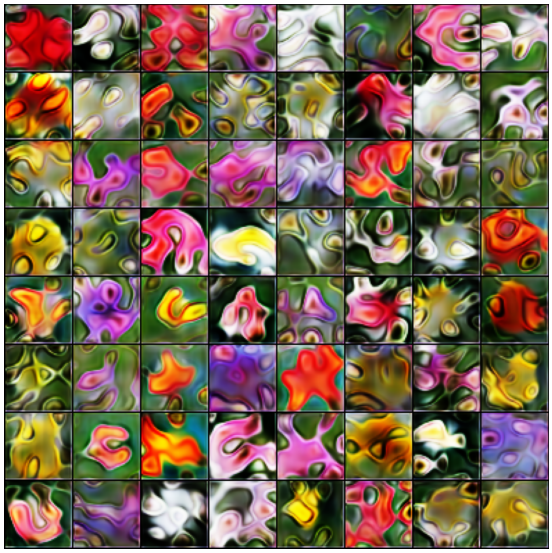
Figure 60 – DCGAN with transposed convolutions trained on Flowers



Source – From the author (2021)

bilinear upsampling generator after 20 epochs of training.

Figure 61 – DCGAN with bilinear upsampling trained on Flowers



Source – From the author (2021)

This was not mentioned until yet, but this behaviour of the bilinear upsampling was universally seen in all tests made on CIFAR-10, although to a lesser extent. This is the author's hypothesis as to why the MNIST tests showed good results for this type of upsampling, since the digits already have some smoothness to their design, this technique fits particularly well to this case. Transposed convolutions and nearest neighbour upsampling usually produce more rough and realistic looking images, explaining the fact of why

they performed better on the other datasets.

Lastly, the nearest neighbour upsampling is the only one who does not produce checkerboard artifacts while still upscaling the images to a more realistic look, this configuration showed the best results and was trained the longest until 100 epochs. The flowers generated by this model after the end of training are shown in Figure 62.

Figure 62 – DCGAN with nearest neighbour upsampling trained on Flowers

Source – From the author (2021)

5.7.2 Faces with DCGAN

This model was considerably slower to train, so most of the decisions over its design were made before training, based on the previous results. Of particular importance, the generator used nearest neighbour upsampling and batch normalization, the latent space had 256 dimensions, and the labels were smoothed to the value 0.9. Due to the time needed for training, this model was not trained to near convergence, but only for 20 epochs.

Since the results tend to fluctuate, it is common to see previous epochs producing better results, for this reason Figure 63 shows samples from the last three epochs of training. Given their small size, there are some particularly realistic looking images produced by the generator in this set, but it still produces many deformed figures and incomplete eyeglasses.

5.7.3 Faces with WGAN-GP

The authors of WGAN-GP mention that "For equivalent architectures, our method achieves comparable sample quality to the standard GAN objective" (GULRAJANI et al., 2017) and follow by mentioning the increased stability of the model as an advantage. The previous experiments have supported both of these claims, since the models have shown

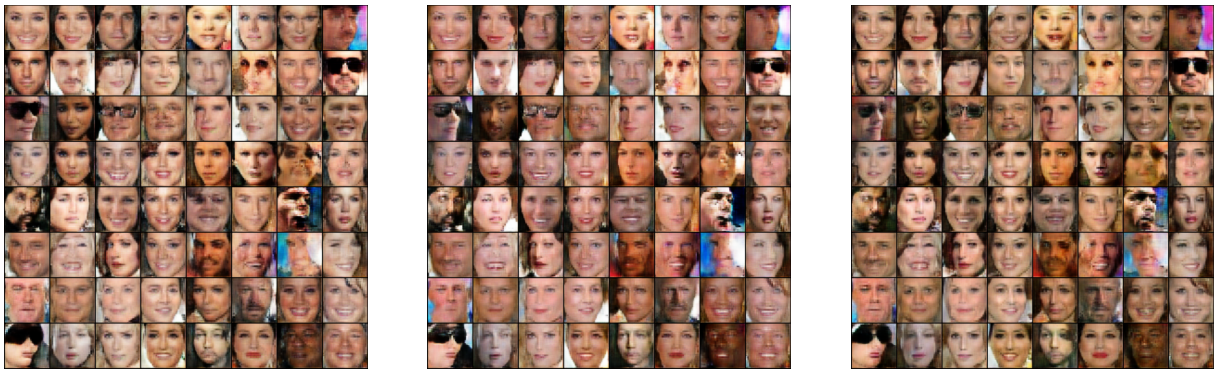
Figure 63 – DCGAN trained on CelebA

(a) Epoch 18 (b) Epoch 19 (c) Epoch 20

Source – From the author (2021)

similar results with the DCGAN for all datasets tested and the hyperparameter choices showed consistent quality in different situations. Given this knowledge, the goal for this experiment was to see if the WGAN-GP could scale better than the DCGAN for this face generation problem.

Figure 64 – WGAN-GP trained on CelebA



(a) Epoch 18 (b) Epoch 19 (c) Epoch 20

Source – From the author (2021)

This model is very much alike the previous one, only the discriminator was changed for a similar critic and the loss was changed as required by the WGAN-GP architecture. The batch normalization was also removed from the generator since it was found to produce some artifacts that reduced the image quality. The results from this experiment are shown in Figure 64.

By a qualitative analysis it is the opinion of the author that the results from the WGAN-GP are generally better, the images seem more realistic and there is less deformations overall.

6 CONCLUSION

After all the experiments the one thing that can be confirmed is that GANs have lived up to their infamy of being hard to train. But with the theoretical basis and experimental evidence arranged in this document it is possible to make some guidelines of what to consider when building a GAN for generating images.

First, and most important, if the dataset contains labels they should be used in the model. This not only allows for more control when using the generator later, but it also significantly increases the performance more than any other technique tested.

About the architecture to use, the standard GAN should be avoided, there is no reason to prefer this method over any other. Particularly, the DCGAN is a good starting point, it is relatively simple to create and it produced good results in all tests, although the selection of hyperparameters may be the most difficult part of the process. But the most promising type is the WGAN-GP, it has the desirable properties of the original WGAN without having the downside of clipping parameters. The WGAN-GP produced better or similar results to the DCGAN in all experiments, it also seemed to be more resistant to hyperparameter changes, making the process of search easier, and lastly it produced visually more pleasing results in the more complex CelebA dataset.

If training a WGAN-GP, the best number for iterations in the discriminator seems to be 1, however the tests cannot say this with a high degree of confidence, it may be that this accelerates training in the beginning but makes convergence harder later in training. However, the results suggest that at the start of training low values of iterations work better.

When upsampling the latent vector, a good technique to choose is nearest neighbour upsampling followed by a normal dimension preserving convolution, this avoids the problem with artifacts produced by the transposed convolution, and it also showed similar results in the experiments, although lagging somewhat behind in some cases. Bilinear convolutions have shown to produce odd smoothing of the images, initially this method should be avoided, unless the dataset also shares this property.

The Adam optimizer should be the default option, maybe replaced by RMSProp when the momentum is not desired, but the β_1 hyperparameter could be set to 0 for this as well. And this term is something to look for in the case of divergence, a default value of 0.9 alone can break convergence.

Batch normalization is also something that can stop the model from converging, the experiments made were inconclusive of when it does or does not work. It can speed up training, but it is a likely candidate to look for if the model is not producing good results.

Label smoothing was also left inconclusive, although some tests showed some slight advantages of using it, it was not significative to be able to confirm their effectiveness. However, they never showed any sign of reducing the performance of the models, so it is

something worth considering to add in order to check if it benefits a determined situation. One disadvantage of this method is that it only works for the DCGAN case, so it may not be worth giving up the benefits of the WGAN-GP for it.

Other than that, the number of dimensions does not need to be very high, only enough as to not reduce the capacity of the generator. The highest value used in the experiments was 256 for the CelebA model, and this is probably enough for the majority of cases. Small batch sizes seems like a good choice, in the order of 16 to 32 produced generally good results in the experiments, if the model is not working correctly the batches should probably not be the main culprit.

These are also not an exhaustive list of all that was proposed as improvements for GANs, many models build on some of the ideas explored in this document. To build better models it is important to understand the concepts and know how to expand them by incorporating new techniques. Rather than the end, the information contained here should be used as a starting point before diving deeper into more advanced territory.

REFERÊNCIAS

ABADI, Martín et al. **TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems**. [S.l.: s.n.], 2015. Software available from tensorflow.org. Disponível em: https://www.tensorflow.org/.

AMINI, Alexander et al. Uncovering and Mitigating Algorithmic Bias through Learned Latent Structure. In: PROCEEDINGS of the 2019 AAAI/ACM Conference on AI, Ethics, and Society. Honolulu, HI, USA: Association for Computing Machinery, 2019. (AIES '19), p. 289–295. DOI: 10.1145/3306618.3314243. Disponível em: https://doi.org/10.1145/3306618.3314243.

ARJOVSKY, Martin; BOTTOU, Léon. **Towards Principled Methods for Training Generative Adversarial Networks**. [S.l.: s.n.], 2017. arXiv: 1701.04862 [stat.ML].

ARJOVSKY, Martin; CHINTALA, Soumith; BOTTOU, Léon. **Wasserstein GAN**. [S.l.: s.n.], 2017. arXiv: 1701.07875 [stat.ML].

BENGIO, Yoshua. Practical recommendations for gradient-based training of deep architectures. **CoRR**, abs/1206.5533, 2012. arXiv: 1206.5533. Disponível em: http://arxiv.org/abs/1206.5533.

BROWN, Tom B. et al. **Language Models are Few-Shot Learners**. [S.l.: s.n.], 2020. arXiv: 2005.14165 [cs.CL].

BRUNTON, S.L.; KUTZ, J.N. **Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control**. [S.l.]: Cambridge University Press, 2019. ISBN 9781108422093. Disponível em: https://books.google.com.br/books?id=CYaEDwAAQBAJ.

BUOLAMWINI, Joy; GEBRU, Timnit. Gender Shades: Intersectional Accuracy Disparities in Commercial Gender Classification. In: _____. **Proceedings of the 1st Conference on Fairness, Accountability and Transparency**. New York, NY, USA: PMLR, Feb. 2018. (Proceedings of Machine Learning Research), p. 77–91. Disponível em: http://proceedings.mlr.press/v81/buolamwini18a.html.

CHOROMANSKA, Anna et al. The Loss Surface of Multilayer Networks. **CoRR**, abs/1412.0233, 2014. arXiv: 1412.0233. Disponível em: http://arxiv.org/abs/1412.0233.

DAHL, Ryan. **Automatic Colorization**. Disponível em: https://tinyclouds.org/colorize/. Visited on: 31 Mar. 2021.

DASTIN, Jeffrey. **Amazon scraps secret AI recruiting tool that showed bias against women**. Oct. 2018. Disponível em: https://www.reuters.com/article/us-

amazon-com-jobs-automation-insight/amazon-scraps-secret-ai-recruiting-tool-that-showed-bias-against-women-idUSKCN1MK08G. Visited on: 22 Mar. 2021.

DHARIWAL, Prafulla et al. **Jukebox: A Generative Model for Music**. [S.l.: s.n.], 2020. arXiv: 2005.00341 [eess.AS].

DUMOULIN, Vincent; VISIN, Francesco. **A guide to convolution arithmetic for deep learning**. [S.l.: s.n.], 2018. arXiv: 1603.07285 [stat.ML].

FUKUSHIMA, Kunihiko; MIYAKE, Sei. Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition. In: COMPETITION and cooperation in neural nets. [S.l.]: Springer, 1982. P. 267–285.

GLOROT, Xavier; BORDES, Antoine; BENGIO, Yoshua. Deep sparse rectifier neural networks. In: JMLR WORKSHOP and CONFERENCE PROCEEDINGS. PROCEEDINGS of the fourteenth international conference on artificial intelligence and statistics. [S.l.: s.n.], 2011. P. 315–323.

GOH, Gabriel. Why Momentum Really Works. **Distill**, 2017. DOI: 10.23915/distill.00006. Disponível em: http://distill.pub/2017/momentum. Visited on: 23 Mar. 2021.

GOODFELLOW, Ian. **NIPS 2016 Tutorial: Generative Adversarial Networks**. [S.l.: s.n.], 2017. arXiv: 1701.00160 [cs.LG].

GOODFELLOW, Ian; BENGIO, Yoshua; COURVILLE, Aaron. **Deep Learning**. [S.l.]: MIT Press, 2016. http://www.deeplearningbook.org.

GOODFELLOW, Ian J.; POUGET-ABADIE, Jean, et al. **Generative Adversarial Networks**. [S.l.: s.n.], 2014. arXiv: 1406.2661 [stat.ML].

GUAN, Shuyue; LOEW, Murray. **A Novel Measure to Evaluate Generative Adversarial Networks Based on Direct Analysis of Generated Images**. [S.l.: s.n.], 2021. arXiv: 2002.12345 [cs.CV].

GULRAJANI, Ishaan et al. Improved Training of Wasserstein GANs. **CoRR**, abs/1704.00028, 2017. arXiv: 1704.00028. Disponível em: http://arxiv.org/abs/1704.00028.

HAY, John C.; MURRAY, Albert E. **MARK I Perceptron Operators' Manual**. Buffalo 21, New York, Feb. 1960. Disponível em: https://apps.dtic.mil/dtic/tr/fulltext/u2/236965.pdf.

HE, Kaiming et al. Deep Residual Learning for Image Recognition. **CoRR**, abs/1512.03385, 2015. arXiv: 1512.03385. Disponível em: http://arxiv.org/abs/1512.03385. Visited on: 17 Mar. 2021.

HE, Kaiming et al. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. **CoRR**, abs/1502.01852, 2015. arXiv: 1502.01852. Disponível em: http://arxiv.org/abs/1502.01852.

HEUSEL, Martin et al. GANs Trained by a Two Time-Scale Update Rule Converge to a Nash Equilibrium. **CoRR**, abs/1706.08500, 2017. arXiv: 1706.08500. Disponível em: http://arxiv.org/abs/1706.08500.

HINTON, Geoffrey E. et al. **Improving neural networks by preventing co-adaptation of feature detectors**. [S.l.: s.n.], 2012. arXiv: 1207.0580 [cs.NE].

HOCHREITER, Sepp; SCHMIDHUBER, Jürgen. Long short-term memory. **Neural computation**, MIT Press, v. 9, n. 8, p. 1735–1780, 1997.

HORNIK, Kurt; STINCHCOMBE, Maxwell; WHITE, Halbert. Multilayer feedforward networks are universal approximators. **Neural networks**, Elsevier, v. 2, n. 5, p. 359–366, 1989.

IOFFE, Sergey; SZEGEDY, Christian. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. **CoRR**, abs/1502.03167, 2015. arXiv: 1502.03167. Disponível em: http://arxiv.org/abs/1502.03167.

KAELBLING, Leslie Pack; LITTMAN, Michael L.; MOORE, Andrew W. Reinforcement Learning: A Survey. **CoRR**, cs.AI/9605103, 1996. Disponível em: https://arxiv.org/abs/cs/9605103.

KARRAS, Tero; LAINE, Samuli; AILA, Timo. A Style-Based Generator Architecture for Generative Adversarial Networks. **CoRR**, abs/1812.04948, 2018. arXiv: 1812.04948. Disponível em: http://arxiv.org/abs/1812.04948.

KARRAS, Tero; LAINE, Samuli; AITTALA, Miika, et al. Analyzing and Improving the Image Quality of StyleGAN. In: PROC. CVPR. [S.l.: s.n.], 2020.

KINGMA, Diederik P.; BA, Jimmy. **Adam: A Method for Stochastic Optimization**. [S.l.: s.n.], 2017. arXiv: 1412.6980 [cs.LG].

KRAMER, Mark A. Nonlinear principal component analysis using autoassociative neural networks. **AIChE journal**, Wiley Online Library, v. 37, n. 2, p. 233–243, 1991.

KRIZHEVSKY, Alex; HINTON, Geoffrey, et al. Learning multiple layers of features from tiny images. Citeseer, 2009.

KRIZHEVSKY, Alex; SUTSKEVER, Ilya; HINTON, Geoffrey E. Imagenet classification with deep convolutional neural networks. **Advances in neural information processing systems**, v. 25, p. 1097–1105, 2012.

LECUN, Yann et al. Gradient-based learning applied to document recognition. **Proceedings of the IEEE**, Ieee, v. 86, n. 11, p. 2278–2324, 1998.

LECUN, Yann A et al. Efficient backprop. In: NEURAL networks: Tricks of the trade. [S.l.]: Springer, 2012. P. 9–48.

LEDIG, Christian et al. Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network. **CoRR**, abs/1609.04802, 2016. arXiv: 1609.04802. Disponível em: http://arxiv.org/abs/1609.04802.

LIU, Ziwei et al. Deep Learning Face Attributes in the Wild. In: PROCEEDINGS of International Conference on Computer Vision (ICCV). [S.l.: s.n.], Dec. 2015.

MCDONALD, Kyle. **How to recognize fake AI-generated images**. Dec. 2018. Disponível em: https://kcimc.medium.com/how-to-recognize-fake-ai-generated-images-4d1f6f9a2842. Visited on: 22 Apr. 2021.

MESCHEDER, Lars M. Which Training Methods for GANs do actually Converge? **CoRR**, abs/1801.04406, 2018. arXiv: 1801.04406. Disponível em: http://arxiv.org/abs/1801.04406.

MIRZA, Mehdi; OSINDERO, Simon. **Conditional Generative Adversarial Nets**. [S.l.: s.n.], 2014. arXiv: 1411.1784 [cs.LG].

MISHKIN, Dmytro; SERGIEVSKIY, Nikolay; MATAS, Jiri. Systematic evaluation of convolution neural network advances on the Imagenet. **Computer Vision and Image Understanding**, 2017. ISSN 1077-3142. DOI: https://doi.org/10.1016/j.cviu.2017.05.007. Disponível em: http://www.sciencedirect.com/science/article/pii/S1077314217300814.

MITCHELL, Tom M. **Machine Learning**. New York: McGraw-Hill, 1997. ISBN 978-0-07-042807-2.

MORDVINTSEV, Alexander; OLAH, Christopher; TYKA, Mike. **Inceptionism: Going Deeper into Neural Networks**. June 2015. Disponível em: https://ai.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html. Visited on: 21 Mar. 2021.

NAZERI, Kamyar; NG, Eric. Image Colorization with Generative Adversarial Networks. **CoRR**, abs/1803.05400, 2018. arXiv: 1803.05400. Disponível em: http://arxiv.org/abs/1803.05400.

NIELSEN, Michael A. **Neural Networks and Deep Learning**. [S.l.]: Determination Press, 2015.

NILSBACK, Maria-Elena; ZISSERMAN, Andrew. Automated Flower Classification over a Large Number of Classes. In: INDIAN Conference on Computer Vision, Graphics and Image Processing. [S.l.: s.n.], Dec. 2008.

ODENA, Augustus; DUMOULIN, Vincent; OLAH, Chris. Deconvolution and Checkerboard Artifacts. **Distill**, 2016. DOI: 10.23915/distill.00003. Disponível em: http://distill.pub/2016/deconv-checkerboard. Visited on: 28 Mar. 2021.

RADFORD, Alec; METZ, Luke; CHINTALA, Soumith. Unsupervised representation learning with deep convolutional generative adversarial networks. **arXiv preprint arXiv:1511.06434**, 2015.

ROWLAND, Todd; WEISSTEIN, Eric W. **Lipschitz Function**. Disponível em: https://mathworld.wolfram.com/LipschitzFunction.html. Visited on: 15 Apr. 2021.

RUDER, Sebastian. An overview of gradient descent optimization algorithms. **CoRR**, abs/1609.04747, 2016. arXiv: 1609.04747. Disponível em: http://arxiv.org/abs/1609.04747.

RUSSAKOVSKY, Olga et al. ImageNet Large Scale Visual Recognition Challenge. **International Journal of Computer Vision (IJCV)**, v. 115, n. 3, p. 211–252, 2015. DOI: 10.1007/s11263-015-0816-y.

SALIMANS, Tim et al. Improved Techniques for Training GANs. **CoRR**, abs/1606.03498, 2016. arXiv: 1606.03498. Disponível em: http://arxiv.org/abs/1606.03498.

SILVER, David; HUANG, Aja, et al. Mastering the game of Go with deep neural networks and tree search. **nature**, Nature Publishing Group, v. 529, n. 7587, p. 484–489, 2016.

SILVER, David; HUBERT, Thomas, et al. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. **CoRR**, abs/1712.01815, 2017. arXiv: 1712.01815. Disponível em: http://arxiv.org/abs/1712.01815.

SINAI, Jonty. **The Perceptron**. Nov. 2017. Disponível em: https://jontysinai.github.io/jekyll/update/2017/11/11/the-perceptron.html. Visited on: 16 Mar. 2021.

STRANG, Gilbert; HERMAN, Edwin "Jed". **Calculus Volume 3**. [S.l.]: OpenStax, Mar. 2016. Disponível em: https://openstax.org/books/calculus-volume-3/pages/1-introduction. Visited on: 21 Mar. 2021.

SZEGEDY, Christian; LIU, Wei, et al. Going Deeper with Convolutions. **CoRR**, abs/1409.4842, 2014. arXiv: 1409.4842. Disponível em: http://arxiv.org/abs/1409.4842.

SZEGEDY, Christian; VANHOUCKE, Vincent, et al. Rethinking the Inception Architecture for Computer Vision. **CoRR**, abs/1512.00567, 2015. arXiv: 1512.00567. Disponível em: http://arxiv.org/abs/1512.00567.

SZEGEDY, Christian; ZAREMBA, Wojciech, et al. Intriguing properties of neural networks. **arXiv preprint arXiv:1312.6199**, 2013.

WAN, Li et al. Regularization of Neural Networks using DropConnect. In: _____. 3. **Proceedings of the 30th International Conference on Machine Learning**. Atlanta, Georgia, USA: PMLR, June 2013. (Proceedings of Machine Learning Research, 3), p. 1058–1066. Disponível em: http://proceedings.mlr.press/v28/wan13.html.

WANG, Zhengwei et al. Synthetic-Neuroscore: Using a neuro-AI interface for evaluating generative adversarial networks. **Neurocomputing**, v. 405, p. 26–36, 2020. ISSN 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2020.04.069. Disponível em: https://www.sciencedirect.com/science/article/pii/S0925231220306548.

XIAO, Han; RASUL, Kashif; VOLLGRAF, Roland. **Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms**. 28 Aug. 2017. arXiv: cs.LG/1708.07747 [cs.LG].

YU, Jiahui et al. Generative image inpainting with contextual attention. In: PROCEEDINGS of the IEEE conference on computer vision and pattern recognition. [S.l.: s.n.], 2018. P. 5505–5514.

ZHANG, Richard; ISOLA, Phillip; EFROS, Alexei A. Colorful Image Colorization. **CoRR**, abs/1603.08511, 2016. arXiv: 1603.08511. Disponível em: http://arxiv.org/abs/1603.08511.

ZHOU, Sharon. **Inception Score**. [S.l.]: Coursera. Disponível em: https://www.coursera.org/lecture/build-better-generative-adversarial-networks-gans/inception-score-HxtYM. Visited on: 16 Apr. 2021.

ZHU, Xiaojin Jerry. Semi-supervised learning literature survey. University of Wisconsin-Madison Department of Computer Sciences, 2005.

APPENDIX A – SPECIFICATIONS OF THE MACHINE

The main components of the machine which ran the experiments consist of:

- **Processor** AMD FX(tm)-6100 Six-Core 3.3 GHz
- **RAM** 4.0 GB
- **Video Card** NVIDIA GeForce GTX 1050, 2.0GB Dedicated Memory, 4.0GB Total Memory
- **Tensorflow** version 2.3.0