Universidad Autónoma de Barcelona

Instituto de Investigación en Inteligencia Artificial CSIC

Improvements in Generative Adversarial Networks with hard constraints

Author
Carlos Mougan Navarro

Coordinators:
Filippo Bistaffa
Juan Antonio Rodriguez Aguilar

Contents

List of Figures

Acknowledgements

First of all thanks to my supervisors Filippo Bistaffa and Juan Antonio Rodriguez for showing me what Artificial Intelligence and Research means.

Thanks to my fellow classmates that have contribute in the knowledge process during this year that has culminated with this master thesis.

Abstract

Deep Learning is a trend in Machine Learning and Artificial Intelligence research. Deep learning techniques have brought revolutionary advances in several fields as machine learning and computer vision. Every now and then, a new deep learning technique borns and outperforms state-of-the-art machine learning and even deep learning techniques. In recent years, the world has seen many major breakthroughs in this field. From the Deep Learning field, there is a technique that has already achieve spectacular results: Generative Adversarial Networks (GAN). In this Master Thesis, we improve GANs with prior knowledge, in order to generate feasible candidates. These candidates are produced so that they could be lately selected from a narrower space for an optimization problem to achieve the best solutions. We focus on the feasibility of solutions, so the proposed candidates satisfy all the hard-constraints.

We decide to tackle a new constraint satisfaction problem, the MoonBoard, a climbing/bouldering utility that can be found in many gymnasiums. We provide an encoding of this problem in which we apply hard-coded constraints with prior knowledge.

We find Deep Convolutional Wasserstein Generative Adversarial Network with Gradient Penalty to be the one that achieves a higher percentage of feasible solutions. After producing a good set of valid candidates (90%), our research leads us further. Could we integrate prior knowledge in order to achieve a better performance? For this, we tried different architectures in order to improve the percentage of valid solutions and we achieved a 7% improvement when applying hard-coded constraints as penalty input for the GANs training.

1 Introduction

The objective of this introductory chapter is to present the research topic of this master thesis, which revolves around Generative Adversarial Networks and its improvements with prior knowledge. In the first section, we will give an introduction to the problem and we will talk about the motivation (1.1) that has led us into this research topic and provide an example of an optimization problem. Then we present our specific work and chosen task (1.2). To conclude (1.3) we provide a brief summary of the contributions made.

1.1 Motivation

Combinatorial Optimization (CO) is a fundamental problem in computer science. A classical example of this kind of problems is the "Knapsack Problem" where one has to maximize the benefit of objects in a knapsack without exceeding its capacity. The problem can be computationally expensive to solve (NP-comlete) depending on the inputs. The Knapsack Problem is a Combinatorial Optimization problem that seeks for the best solution among all the feasible solutions. There is many similar problems where research is being conducted such as ride sharing [6]. In Section 2 we talk about our specific problem.

In recent times Deep Learning has attracted a lot of attention, achieving superhuman performance for some tasks. With superhuman we mean that it has been able to beat human performance, accomplishing astonishing results. Furthermore, from the Deep Learning field, there is a branch that has received special consideration, Generative Adversarial Networks. First introduced by Ian Goodfellow in 2014 [8], this kind of training has been applied in different domains such us speech enhancement [16], image generation [7], attention prediction [5], anomaly detection [19]... Even in art, where a painting was sold for $432,500$ New York Times [14]. Even if this Deep Learning technique seems to be really promising, there is still many field where it can be applied. In this Master Thesis we apply Generative Adversarial Networks in order to produce feasible candidates and then we integrate prior knowledge in order to improve the quality and the number of feasible solutions generated by our model. Furthermore we will treat a bouldering trainning facility that can be seen as a constraint satisfaction problem, the MoonBoard.

Machine learning can improve Combinatorial Optimization by replacing some of the heavy computations with a fast approximation by proposing feasible candidates to Constraint Satisfaction Problems. Most Machine Learning algorithms fall into the category of Supervised Machine Learning. In Combinatorial Optimization we don't have any label, so we have to use Unsupervised Learning. In this master thesis we apply Generative Adversarial Networks to propose new possible solutions to the problem.

Even though most Machine Learning algorithms generalize well to the properties of the data, finding feasible solutions is not an easy problem [25], it is even more challenging in Deep Learning Learning, especially when using Generative Adversarial Networks.

1.2 Our Work

The goal of this master thesis is the improvement of the solutions proposed by Generative Adversarial Networks for a given task. For Combinatorial Optimizaton problems we can require two main components to the solutions proposed by Generative Adversarial Networks:

- The solutions proposed are **feasible**, meaning that they satisfy all of the hard-constraints of the task we are trying to solve. For example in the Knapsack Problem, one constraint is the weight limit.
- The solutions proposed are **optimal**, meaning that somehow they are close to a good solution. For example in the Knapsack Problem, optimal solutions will be those one that have a high number of items.

We will focus on the feasibility requirement since GAN have problems generalizing the instrinsic constraints of the task. The solutions proposed by the Generator agent do not always satisfy the set of constraints that define the task, this makes the solutions proposed not feasible.

In order to improve the percentage of valid solutions proposed by the Generator agent, we evaluated different training architectures, loss functions, and encodings to try to improve percentage of valid solutions. We will add **prior knowledge that can be expressed as hard-coded constraints** to the problem so Generative Adversarial Networks definetely produce feasible candidates.

In order to tackle this problem, we will choose a problem where hard constraints are clear and size is managable. In this way it is easy to check if the solutions satisfy the constraints. For this, we have chosen a problem where we can focus in producing feasible candidates for **constraint satisfaction problems**. The selected problem is a climbing utility that is normally found in bouldering facilities "The MoonBoard" (2).

1.3 Contributions

With this master thesis we have made the following contributions to the field:

Contribution to improve GANs exploiting hard constraints

We improved the percentage of feasible solutions generated by the Generator agent in the Generative Adversarial Networks training architecture by encoding a penalty function (4) that indicated how far the generated candidate is from being a real one. This allows us to include prior knowledge as hard constraints in the Generative Adversarial Networks as an input.

Contribution to the encoding of the Moonboard

We modeled the definition and the formalization of the MoonBoard problem, by developing an encoding that allows us to formally express its features and execute computational tasks. Modelling the MoonBoard has never been tackled from a Deep Learning point of view. The encoding of bouldering task could be later used for more general climbing tasks as normal bouldering.

Extensive empirical analysis of GAN comparison

We provided with a thorough analysis of GANs observations. We have conducted several experiments: varying the structure of the training and the structure of Generator/Discriminator, applying different hyperparameters, and modelling various loss functions. We have selected the best performing GAN by means of valid proposed candidates; after this, we compared the proposed solutions with the ones generated by the GAN with hard constraints, and we measured the quality of the candidates proposed by both.

2 Problem Statement

Our goal for this master thesis is to improve Generative Adversarial Networks [8] to generate candidates that satisfy a given set of hard constraints. In other words, using the Generative Adversarial Networks paradigm as a heuristic that will help us reduce the solution space of the problem. We will demand to the solutions provided by the Generative Adversarial Network one main thing:

- We will require that the solutions are **feasible**. Meaning that they satisfy all the hard constraints that define the problem. Sometimes Machine Learning algorithms are not able to generalize to specific properties of the problem. In order to improve this and for this specific purpose we added the possibility to integrate hard coded constraints to the problem.

So in this Master Thesis, we are focusing on improving the feasible solutions proposed by the Generative Adversarial Network. For that, we will choose an easy to manage problem (in a computational way) and try to improve the percentage of valid solutions that the GAN is generating.

With prior knowledge of the problem we are able to define hard constraints that determine the feasibility of the solutions. We will also introduce this value as an input to one of the agents in the Generative Adversarial Networks, so the generated solutions have a higher percentage of feasibility.

2.1 Case Study

To improve the percentage of valid solutions proposed by the Generative Adversarial Network, we will choose an understandable and managable problem, so that it is easy to decide which are the satisfying constraints.

The task chosen for our Generative Adversarial Network to learn, is a climbing/bouldering problem, the MoonBoard, that is a facility that can be found in gymnasiums and bouldering facilities. In the official web page [url] of the MoonBoard more information and boulder problems can be found.

The MoonBoard is an easy to manage problem, smaller than other problems found in the literature [2], [24], this allows us to conduct experiments within a non too expensive computations. The MoonBoard constraints are a small amount of lineal constraints that are easy to formalize. Also the intuitive nature of this

problem allows us to gain prior knowledge that will let us understand and measure what are the restrictions that define the problem.

2.2 What is the MoonBoard?

The MoonBoard is a bouldering training wall with an inclination of 40 degrees. It is designed for climbers that want to optimise their performance in an artificial environment that allows to recreate extremely hard real life climbing moves. Holds are arragned in a grid indexed by letter and numbers coordinates. All of the holds that form the MoonBoard have a specific location and rotation.

It is possible for anyone to do similar routes since the MoonBoard has a set of defined bouldering paths. It also has the option of introducing your own path so you can share problems with fellow MoonBoard users. For a path to be valid it needs to satisfy certain conditions, discussed in the section 2.4.

Figure 2.1: Real Picture of the moonboard, 40° inclination

With thousands of MoonBoards available to climb around the world, the MoonBoard has become one of the most sought-after training systems for home users and commercial climbing gyms alike. [13]

2.3 Data description

The dataset used in our experiments is a JSON file containing 21900 examples of MoonBoard valid configurations. Each configuration comes with its holds, grades and names. The grade of a bouldering path represents the difficulty of the route [22]. Every configuration of the MoonBoard is composed by three different types of holds:

Starters The start of a bouldering problem is marked by these holds. If there is only one hold, the climbing starts with both of the hands on that hold. If two holds are marked, it starts with one hand placed on each hold.The feet start in any of the lower holds of the MoonBoard. [18]. They are coloured in green in the image 2.2

Top The end of a bouldering problem is marked by this hold. Often problems end at the top of the wall, so rather than a specific hold, sometimes it finishes whene grabbing the top part of the wall . The goal is to reach the Top hold, using the Move holds and starting at the Start holds. They are coloured in red in the image 2.2

Moves They are the rest of the holds that make the configuration. In the Figure 2.2, they are the blue holds.

```
  "NameForUrl": "atxurra",
  "Id": 329721,
 },
  "Moves": [
   {
    "Id": 1816963,
    "Description": "G2",
    "IsStart": false,
    "IsEnd": false
   },
   {
    "Id": 1816964,
    "Description": "F5",
    "IsStart": true,
    "IsEnd": false
   }]
```

The above code shows an example of the JSON file that contains all of the data.

2.4 Valid Routes

In this section we will give an illustrative example of what are valid paths in the MoonBoard. In Figure 2.2 we can see a real valid path of the MoonBoard.

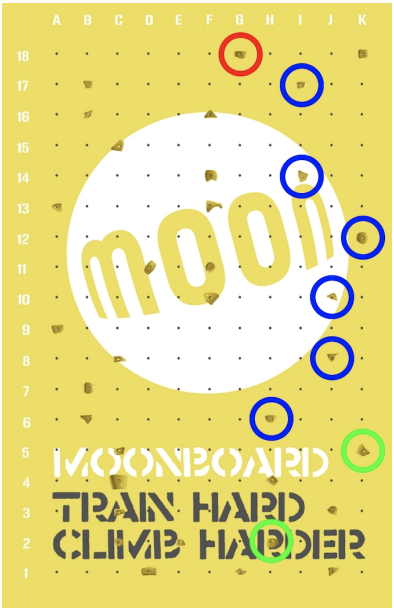


Figure 2.2: Example of a Valid path of the MoonBoard

As we can see with the description 2.3, for every configuration of the MoonBoard there are some requirements that it needs to meet. This requirements are:

- There are 1 or 2 starters
- There is one finish hold

For Figure 2.2 we can see that the green cells are for the starters, the blue for the moves and the green for the finish. This example satisfies all the constraints, since it has two starter holds, some move holds and only a finish hold. We can say that this example satisfies all the hard constraints for the problem and it is a valid candidate. For a more formal definition/encoding see Section 4.

For Figures 2.3 and 2.4 we can see badly generated moonboard configurations:

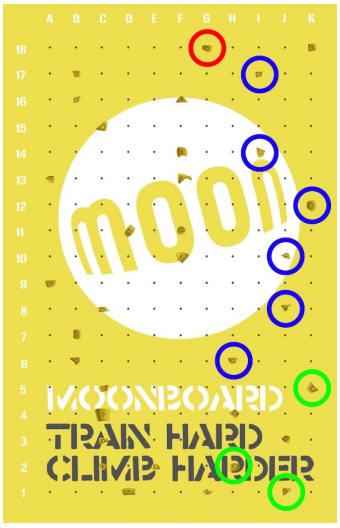


Figure 2.3: MoonBoard with one extra starter

For Figure 2.3 we can see that it has one extra starter hold. This will introduce a penalty for the image since it is one extra start hold further away from satisfying the constraints of two starting holds.

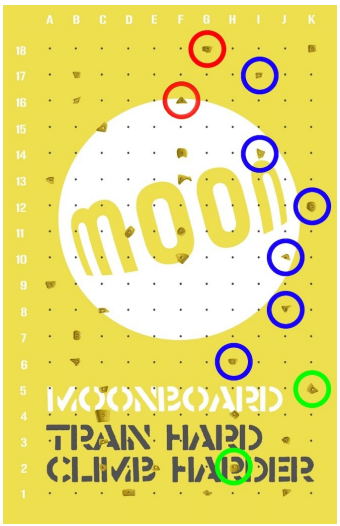


Figure 2.4: MoonBoard with one extra top

For Figure 2.4 we can see that it has one extra top hold. This will introduce a penalty for image as it has one extra finish hold further away from satisfying the constraints of only one finish hold.

3 Background

3.1 Fast introduction to Deep Learning

Neural networks were initially developed to emulate the human brain [21]. There are 15,000 millions of neurons in the human brain, each of them with 10 to 10000 connections with other neurons. The initial goal of the neural networks was to develop a computational model that imitates the human brain and it is able to resolve complex tasks similar to the human brain.

In a Neural Network the most simple computation unit is the neuron, it can also be called a node or unit. This unit receives the input from other nodes, or from any external source and computes an output. Each input has an associated weight (w), which is assigned on the basis of its relative importance to other inputs. The node applies a function to the weighted sum of its inputs as shown in the Figure below:

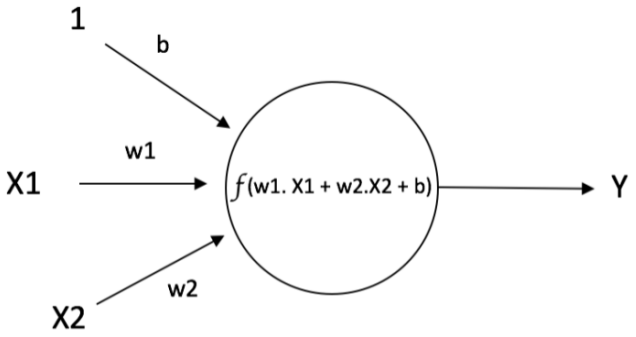


Figure 3.1: Activation and output of a neuron

$$y = f(w_1 \cdot X_1 + w_2 \cdot X_2 + b) \tag{3.1}$$

Where X_1 and X_2 have their correspondent weights w_1 and w_2 associated with those inputs. In addition, there is another input with a value of $b = 1$ called the Bias.

The main idea is to obtain linear combinations of explanatory variables by means of functions that are not necessarily linear. Every activation function takes a single number and performs a certain fixed mathematical operation on it. Each neuron has an activation function defining the output. The activation function is used to introduce non-linearity in the modeling capabilities for the network. There are several activation functions we can encounter:

Sigmoid takes a real-valued input and squashes it to range between [0,1]

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{3.2}$$

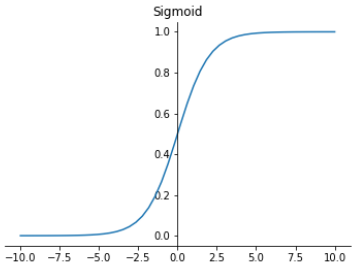


Figure 3.2: Sigmoid Activation Function

Hyperbolic Tangent takes a real-valued input and squashes it to the range [-1, 1]

$$tanh(x) = \frac{e^{x} - e^{-}x}{e^{x} + e^{-}x} \tag{3.3}$$

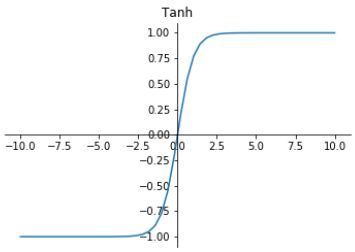


Figure 3.3: Hyperbolic Tangent Activation Function

Relu The activation function rectified linear unit (ReLU) is a interesting transformation that activates a single node if the input is above a certain threshold. The behavior is that, as long as the input has a value below zero, the output will be zero but, when the input rises above a certain value, the output is a linear relationship with the input. The output is a linear relationship with the input variable from the form $f(x) = x$. The ReLU activation function has proven to work in many different situations and is currently widely used.

$$f(x) = max(0, x) \tag{3.4}$$

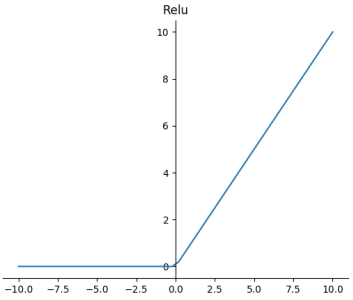


Figure 3.4: ReLU Activation Function

LReLU LReLU stands for Leaky Rectified Linear Unit. Leaky ReLUs allow a small, positive gradient when the unit is not active.

$$f(x) = \begin{cases} x & : x > 0 \\ 0.2 * x & : x < 0 \end{cases} \tag{3.5}$$

3.1.1 Neural Networks

There are different types of neural networks, the feedforward neural network is the first and simplest type. It contains multiple neurons (nodes) arranged in layers. Nodes from adjacent layers have connections or edges between them. All these connections have weights associated with them.

The mathematical model for a neural network with one hidden layer for a continuous output is the following:

$$y_k = \alpha_k + \sum_h w_{hk}\phi_h(\alpha_h + \sum_i w_{ih}x_i) \tag{3.6}$$

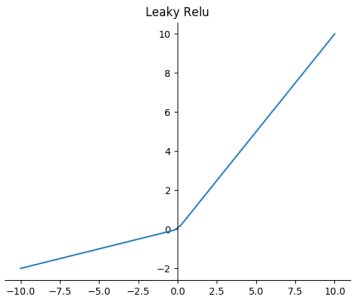


Figure 3.5: Leaky ReLU Activation Function

Where y_k denotes the output, x_i the input, $\alpha + \sum_i w_{ih} x_i$ a linear combination of the inputs, ϕ_h an activation layer (described above), $\alpha_k + \sum_h w_{hk} \phi_h(\alpha_h + \sum_i w_{ih} x_i)$ a linear combination of a hidden layer and w_{ih} y w_{kh} the parameteres to be fitted.

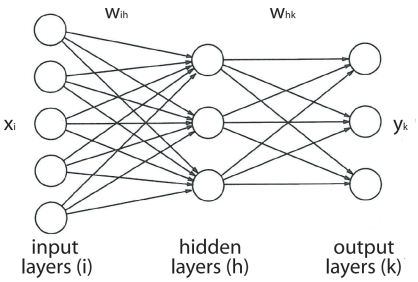


Figure 3.6: One Hidden Layer Neural Network

The first phase "forward propagation" occurs when the network is exposed to the training data and these cross the entire neural network for their predictions (labels) to be calculated. That is, from the input data to the labels through the neurons applying non linear transformations to the information they receive from the neurons of the previous layer and sending it to the neurons of the next layer. When the data has crossed all the layers, and all its neurons have made their calculations, the final layer will be reached with a result of label prediction for those input examples. [20]

3.1.2 Back Propagation Algorithm

In the beginning of the trainning, all the edge weights are randomly assigned. For every data input of the training dataset, the neural network gets activated and the output is observed. This output is compared with the desired output and we

can measure this difference with different metrics depending on the type of task given and what metric we want to optimize: Mean Absolute Error, Mean Squared Error, R^2, Area Under ROC Curve, Logarithmic Loss or custom made function.

Once this comparison metric the error is "transmited" back to the previous layer. Then the error is calculated and the weights are "re-adjusted". This is an iterative process that is repeated until the output error is below a certain threshold.

Once the above algorithm terminates, we consider that the Neural Network has learned and we consider is ready to receive a new set of inputs. We say that the training process is done learning form the labeled data and improved by the error propagation.

3.1.3 Optimization

Optimization is a process of searching for parameters that minimize or maximize our functions. After we have chosen a certain metric like accuracy, Mean Squared Error, R^2, Area Under ROC Curve, or a custom function (J_θ) that indicates how well our model solves a given problem.

Gradient descent is one of the most popular algorithms to perform optimization and by far the most common way to optimize neural networks. Gradient descent is a way to minimize an objective function J_θ parameterized by a model's parameter $\theta \in \mathbb{R}_D$ by updating the parameters in the opposite direction of the gradient of the objective function $\nabla\theta J_{(\theta)}$ with respect to the parameters. The learning rate of η determines the size of the steps we take to reach a minimum.

Gradient descent has the problem of "local minima" where the gradient gets trapped in a position where it can escape and it is not able to achieve optimum performance. Another problem that this algorithm can encounter is the so-called "saddle points". These are plateaus, where the value of the cost function is almost constant. At these points, the gradient is almost zeroed in all directions making it impossible to escape.

Batch Gradient Descent computes the gradient of the cost function with respect to the parameters θ for the entire training dataset.

$$\theta = \theta - \eta \cdot \nabla_\theta J_{(\theta)} \tag{3.7}$$

As we need to calculate the gradients for the whole dataset to perform just one update, batch gradient descent can be very slow and is intractable for datasets that don't fit in memory.

Stochastic Gradient Descent (SGD) instead performs a parameter update for each training example x_i and label y_i

$$\theta = \theta - \eta \cdot \nabla_\theta J_{(\theta);x_i;y_i} \tag{3.8}$$

Batch gradient descent performs redundant computations for large datasets, as it recomputes gradients for similar examples before each parameter update. SGD does away with this redundancy by performing one update at a time. It is therefore usually much faster. SGD performs frequent updates with a high variance that cause the objective function to fluctuate heavily. This algorithm complicates convergence to the exact minimum, as SGD will keep iterating.

Mini-batch Gradient Descent is the middle point between the last two. It performs an update for every mini-batch of n training examples

$$\theta = \theta - \eta \cdot \nabla_\theta J_{(\theta);x_{i:i+n};y_{i:i+n}} \tag{3.9}$$

This way, it reduces the variance of the parameter updates, which can lead to more stable convergence; it can also make use of highly optimized matrix optimization.

Another technique used with Stochastic gradient Descent is the **Momentum**. Instead of using only the gradient of the current step to guide the search, momentum accumulates the gradient of the past steps to determine the direction to go.

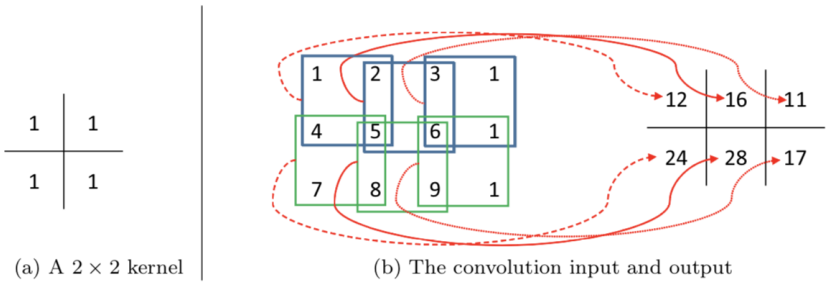
Root Mean Square Propagation or RMSProp is similar to the gradient descent algorithm with momentum. The RMSProp optimizer restricts the oscillations in the vertical direction. [17]

So far RMSProp and Momentum take contrasting approaches. While momentum accelerates our search in direction of minima, RMSProp impedes our search in direction of oscillations. **Adam** or **Adaptive Moment Optimization** lgorithms combines the heuristics of both Momentum and RMSProp. [15]. Adam is an adaptive learning rate method, which means, it computes individual learning rates for different parameters. Its name is derived from adaptive moment estimation, and the reason it's called that is because Adam uses estimations of first and second moments of gradient to adapt the learning rate for each weight of the neural network. It was first published in the ICLR 2015 conference [10]

3.1.4 Convolutions

A convolutional neural network (CNN or ConvNet) is a concrete case of Deep Learning neural networks, which were already used at the end of the 90s but which in recent years have become enormously popular when achieving very impressive results in the recognition of image, deeply impacting the area of computer vision [20]

What is a Convolution? To get an idea of it we will start convolving a matrix with a single convolution kernel. Suppose the input image is 3 4 and the convolution kernel size is 2 2, as illustrated in the Figure



(a) A 2 × 2 kernel (b) The convolution input and output

Figure 3.7: Convolution Operation

When overlaping the convolution kernel, we can compute the product between the numbers at the same location in the kernel and the input, and we get a single number by summing these products together. In the case of the figure will be $1+2+4+5=12$ then we move one right and plug one right, $2+3+5+6=16$ ...

There are two interesting terms in the convolution process. For the problem that the corners and borders get few convolutions, there is the term of *padding* which indicates the number of plain extra sizes that we add to the kernel in order to iterate several times throught the borders and corners. *Stride* is another important concept in convolution. In the Figure of convolution operation, we convolve the kernel with the input at every possible spatial location, which corresponds to the stride s = 1. However, if $s > 1$, every movement of the kernel skip s 1 pixel locations, meaning that between each convolution we make bigger steps.

Why to convolve? A convolution is able to successfully capture the Spatial and Temporal dependencies in an image through the application of relevant filters. We can see that in the Lenna Figure where there are different convolutions applied. [4]

(a) Lenna (b) Horizontal edge (c) Vertical edge

Figure 3.8: The Lenna image and the effect of different convolution kernels

After applying the convolutions, we can learn features that activate for edges with different angles. When we move further down in the deep network, subsequent layers can learn to activate only for specific (but more complex) patterns, e.g., groups of edges that form a particular shape. These more complex patterns will be further assembled by deeper layers to activate for semantically meaningful object parts or even a particular type of object, e.g., dog, cat, tree, beach, etc. [23]

3.1.5 Normalization

The first and most common normalization is the *Batch normalization*. It is a normalization method that normalizes activations in a network across the mini-batch. For each feature, batch normalization computes the mean and variance of that feature in the mini-batch. It then subtracts the mean and divides the feature by its mini-batch standard deviation. Finnally it rescales the normalized activations and adds a constant. This means that the expressiveness of the network does not change.

Algorithm 1 Batch Normalization

procedure INPUT(For values a mini-batch of inputs$\{x_1 \dots m\}$)
  $\mu \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$ //mini-batch mean
  $\rho^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i - \mu$ //mini-bath variance
  $\hat{x_i} \leftarrow \frac{x_i - \mu}{\sqrt{\rho^2 + \epsilon}}$ // normalize

The goal of batch norm is to reduce internal covariate shift by normalizing each mini-batch of data using the mini-batch mean and variance.

Layer Normalization introduced by Jimmy Ley [9] tries to address some of the problems of batch normalization. Instead of normalizing examples across mini-batches, layer normalization normalizes features within each example

Algorithm 2 Instance Normalization

procedure INPUT(For inputs x_i over a dimension D)
  $\mu \leftarrow \frac{1}{D}\sum_{d=1}^{D} x_i^d$
  $\rho^2 \leftarrow \frac{1}{D}\sum_{d=1}^{D} x_d - \mu$
  $\hat{x_d} \leftarrow \frac{x_d - \mu}{\sqrt{\rho^2 + \epsilon}}$

Instance Norm introduced by [3], works in a similar way to previous normalizations but it goes one step further, computing the mean and standard deviation and normalizing for each channel and for each training example.

In Figure 3.9 we can see a visual representation of each of the different normalization across a tensor.

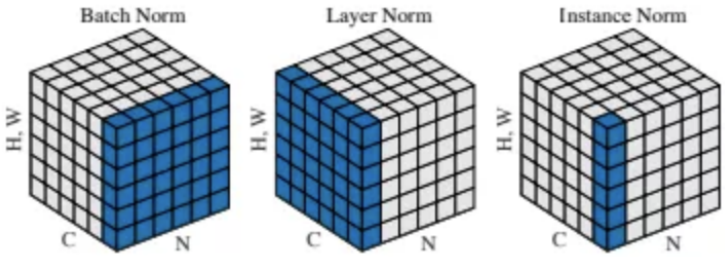


Figure 3.9: Visualization of normalization techniques

3.2 Unsupervised Learning

What is Unsupervised Learning? The classical answer to this is to learn a probability density of distribution. This is often done by defining a parametric family of densities $(P\theta)_{\theta\in\mathbb{R}^d}$ and finding the one that maximized the likelihood on our data: if we have real data examples $\{x^{(i)}\}_{i=1}^m$, we would solve the problem

$$max_{\theta\in\mathbb{R}^d}\frac{1}{m}\sum_{i=1}^{m}\log P_\theta(X^{(i)}) \tag{3.10}$$

If the real data distribution $\mathbb{P}_r$ admits a density and $\mathbb{P}_\theta$ is the distribution of the parametrized density $\mathbb{P}_\theta$, then, asymptotically, this amounts to minimizing the Kullback-Leibler divergence $KL(\mathbb{P}_r \,||\, \mathbb{P}_\theta)$.

For this to make sense, we need the model density $\mathbb{P}_\theta$ to exist. For this reason for this project, we have selected this problem (moonboard) where we can have some intuition about the model density. But in more general problems this is not the normal case. [12]

There are different ways to measure the distance or the divergence $\rho(\mathbb{P}_\theta, \mathbb{P}_r)$ of the two model distributions.

The most fundamental difference between such distances is their impact on the convergence of sequences of probability distributions. A sequence of distributions $(\mathbb{P}_t)_{t\in\mathbb{N}}$ converges if and only if there is a distribution P such that $\rho(\mathbb{P}_t, \mathbb{P}_\infty)$ tends to zero, something that depends on how exactly the distance ρ is defined. Informally, a distance ρ induces a weaker topology when it makes it easier for a sequence of distribution to converge.

We can now define different distances to measure the convergence of the distribution $\rho(\mathbb{P}_\theta, \mathbb{P}_r)$.

- The *Total variation* (TV) distance:

$$\rho(\mathbb{P}_g, \mathbb{P}_r) = \sup_{A\in\sum} \mid \mathbb{P}_r(A) - \mathbb{P}_g(A) \mid \tag{3.11}$$

- The *Kullback Leiber*(KL) divergence:

$$KL(\mathbb{P}_r \mid\mid \mathbb{P}_g) = \int \log \frac{P_r(x)}{P_g(x)} P_r(x) d\mu(x) \tag{3.12}$$

  where both $\mathbb{P}_r$ and $\mathbb{P}_g$ are assumed to be absolutely continuous. The KL divergence is famously assymetric and possibly infinite when there are points such that $P_g(x) = 0$and $P_r(x) > 0$.

- The *Jensen Shanon* (JS) divergence:

$$JS(\mathbb{P}_r, \mathbb{P}_g) = KL(\mathbb{P}_r \mid\mid \mathbb{P}_m) + KL(\mathbb{P}_g \mid\mid \mathbb{P}_m) \tag{3.13}$$

  where $\mathbb{P}_m$ is the mixture $\frac{\mathbb{P}_r + \mathbb{P}_g}{2}$. This divergence is symmetrical and always defined.

- The *Earth-Mover* (EM) distance or Wasserstein-1.

$$W(\mathbb{P}_r, \mathbb{P}_g) = \inf_{\gamma \in \prod(\mathbb{P}_r, \mathbb{P}_g)} \mathbb{E}_{(x,y)\sim\gamma}[\mid\mid x - y \mid\mid] \tag{3.14}$$

  where $\prod(\mathbb{P}_r, \mathbb{P}_g)$ denotes the set of all joint distributions $\gamma(x, y)$whose marginals are respectively $\mathbb{P}_r$ and $\mathbb{P}_g$. Intuitively, $\gamma(x, y)$ indicates how much "mass" must be transported from x to y in order to transform the distributions $\mathbb{P}_r$ into the distribution $\mathbb{P}_g$. The EM distance then is the "cost" of the optimal transport plan.

3.2.1 Genetarive Adversarial Networks

Generative adversarial networks (GANs) were first introduced by Ian Goodfellow [8] as a novel approach to generative modelling, a task whose goal it is to learn a distribution of real data points.

The term adversarial refers to the use of two opposing neural networks in GANs. The *discriminator* network receives either a generated sample or a true data sample and must distinguish between the two[1]. The *generator* is trained to fool the discriminator. Formally, the game between the generator G and the discriminator D is the min-max objective:[11]

$$\min_G \max_D \mathbb{E}_{x\sim\mathbb{P}_r} \log D(x) + \mathbb{E}_{x\sim\mathbb{P}_g} \log 1 - D(G(z)) \tag{3.15}$$

where $\mathbb{P}_r$ is the data distribution and $\mathbb{P}_g$ is the model distribution implicitly defined by $G(z)$. The input z to the generator is sampled from some simple noise distribution.

3.2.2 GANs limitations

When training Generative Adversarial Networks we find different problems. Generative adversarial networks are based on the min-max non-cooperative game. In short, if one wins the other loses. Your opponent wants to maximize its actions and your actions are to minimize them. In game theory, the GAN model converges when the discriminator and the generator reach a Nash equilibrium. Since both sides want to undermine the others, a Nash equilibrium happens when one player will not change its action regardless of what the opponent may do. The discriminator networks learn easier the distribution of the data blocking the generator to learn anything. When this happens we have a *saturated discriminator*. Another common error is *mode collapse*, that its when the generator produces limited variety of samples.

3.2.3 WGAN

We have seen that GAN has several problems such as saturated discriminator, mode collapse and vanishing gradient. Martin Arjovsky [12] proposes the Wasseirstein-GAN. This variation of GAN uses the Wasseirstein distance as a cost function since it has a smoother gradient everywhere. WGAN learns no matter if the generator is performing or not. The fact that this loss function is continuos and differentiable means that we should train the critic till optimality.

This new model represents an alternative to traditional GAN trainning, Martin Arjovsky [12] shows that this type of trainning improves stability of learning, gets rid of problems like mode collapse and provide meaningful learning curves useful for debugging and hyperparameter searches.

In Improved Trainning of GANs, Ishaan Gulrajani1[11] proposes using a gradient penalty **WGAN-GP** in order to make the loss function differentiable ("1-Lipschitz"). A differentiable function f is 1-Lipschitz if and only if it has gradients with norm at most 1 everywhere. So the gradient penalty penalizes the model if the gradient norm moves away from its target norm value 1. The Loss function for the Discriminator and the generator will be:

$$L_D = \nabla \frac{1}{m} \sum_{i=1}^{m} \mathbb{D}(R) - \mathbb{D}(\mathbb{G}(z) + \lambda_2(|| \mathbb{D}(\mathbb{G}(z) ||_2 - 1)^2 \tag{3.16}$$

$$L_G = \nabla \frac{1}{m} \sum_{i=1}^{m} \mathbb{D}(\mathbb{G}(z)) \tag{3.17}$$

4 Formalization

As defined in previous sections (1.3), the main objective of this master thesis is to find a way to improve the percentage of valid solutions that the generator is proposing. In order to achieve this, we will provide to the discriminator information about the value of the distance between the real image and the generated image. We will call this the penalty function.

Our encoding discussed in section (2.3), is based on three matrices of shape [18,11]. Each matrix contains the moves for one type of hold. In Figure 4.1 we can see a valid path and after this, we can see how the encoding of the problem should look in Figures 4.2, 4.3, 4.4.

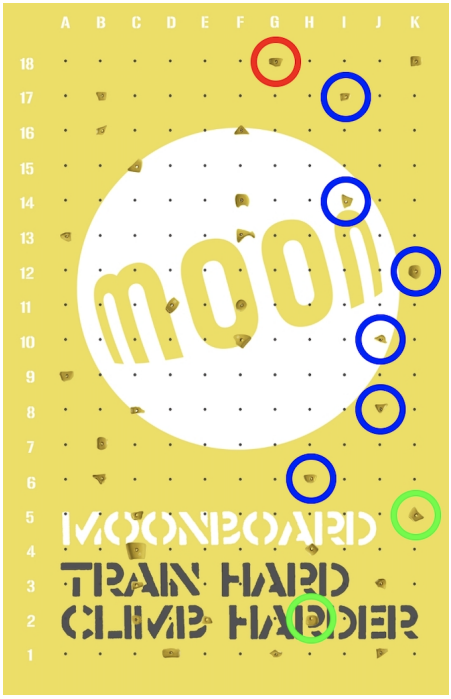


Figure 4.1: Example of a Valid path of the MoonBoard

In Figures 4.2,4.3 and 4.4 we plotted the matrix representation of the start,moves and top holds of the real Figure 4.1 Our tensor will be stacking of the three types of holds.

$$M_S = \begin{bmatrix}
0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&\mathbf{1}\\
0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&\mathbf{1}&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0
\end{bmatrix}$$

Figure 4.2: Start hold matrix

$$M_M = \begin{bmatrix}
0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&\mathbf{1}&0&0\\
0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&\mathbf{1}&0&0\\
0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&\mathbf{1}\\
0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&\mathbf{1}&0\\
0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&\mathbf{1}&0\\
0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&\mathbf{1}&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0
\end{bmatrix}$$

Figure 4.3: Move holds matrix

$$M_F = \begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}$$

Figure 4.4: Top hold matrix

$$M = [M_S, M_M, M_F] \tag{4.1}$$

Where the sub-index represents S for starters, M for moves and F for final holds. And all the elements from all the matrix are either [0,1]: $M_{[S,M,F]}(i,j) \in [0,1]$.Now we will provide an example of the penalty constraint that we are applying in order to improve the percentage of feasible constraints that the GAN is generating. Suppose that there is the following configurations of starter holds:

$$M_S = \begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{1} & 0
\end{bmatrix}$$

Figure 4.5: Wrong *Start* hold matrix

As we can see there is four starter holds, that is two too many for our problem. Then we can define our penalty function for matrix 4.5 like:

$$M_p = \lambda(\sum_{i,j} M_S(i,j) - Start) \tag{4.2}$$

$$M_p = \lambda(4-2) = 2 \cdot \lambda \tag{4.3}$$

Start will be a variable indicating how many starting holds we want. In this case we selected $Start = 2$. In the end, after this process has been applied we will have another matrix of size [18,11] in which we will have all data equal to 2. Even if this is seems redundant the deep learning algorithm should be able to find patterns with the rest of the matrix and extract the relevant information. We can already notice that the parameter λ is going to play a crucial role in determining what is the order of magnitude of this added information. We discuss the effect of this parameter in the section 5.3.2 of results

So as we have seen the data can come either from the discriminator or the generator, they both come as in equation 4.1. Then we applied penalty function that transforms the structure:

$$pen(M) = [M, M_P] = [M_S, M_M, M_F, M_P] \tag{4.4}$$

If the data comes from the real distribution we will call it $M_{\mathbb{R}}$ and if it comes from the generator $M_{\mathbb{G}}$. In our training we will have two different set of matrices:

$$M^*_{\mathbb{R}} = pen(M_{\mathbb{R}}) \tag{4.5}$$

$$M^*_{\mathbb{G}} = pen(M_{\mathbb{G}}) \tag{4.6}$$

As we can see we can make this process iterative with as many constraints as we want to include. This will allow us to add a given set of constraints to the task.

The loss of the Discriminator and the Generator for the WGAN-GP [11] will be:

$$L_D = \nabla \frac{1}{m} \sum_{i=1}^{m} \mathbb{D}(M^*_{\mathbb{R}}) - \mathbb{D}(M^*_{\mathbb{G}}) + \lambda_2(|| \mathbb{D}(M^*_{\mathbb{G}}) ||_2 - 1)^2 \tag{4.7}$$

$$L_G = \nabla \frac{1}{m} \sum_{i=1}^{m} \mathbb{D}(M^*_{\mathbb{G}}) \tag{4.8}$$

We can compare the modified equations 4.8 and 4.7 with the original equations from the background 3.16 and 3.17. We have modified the original loss by plugging the matrix with the penaly (M^*). This modification not only implies a change of the loss function but a change in the architecture of the GANs.

$$L_D = \nabla \frac{1}{m} \sum_{i=1}^{m} \mathbb{D}(R) - \mathbb{D}(\mathbb{G}(z) + \lambda_2(|| \mathbb{D}(\mathbb{G}(z) ||_2 - 1)^2 \tag{4.9}$$

$$L_G = \nabla \frac{1}{m} \sum_{i=1}^{m} \mathbb{D}(\mathbb{G}(z)) \tag{4.10}$$

5 Evaluation

This section is structured in the following parts: In the first section 5.1 we explain which were our hypotheses prior to the experiments. Then in section 5.2 we give the training methodology used, explain the architecture of our model and we give the hyperparameters used. Finally, in section 5.3 we show which are the results of the evaluation.

5.1 Hypothesis

In this master thesis, we had the following goals:

- Identifying the best achieving Generative Adversarial Networks in terms of percentage of valid solutions. There are several types of GANs in the literature and we perform experiments with them in order to find the one that achieves the best results for our problem.
- Our main goal for this master thesis is introducing prior knowledge as a hard constraint to GANs. Once we had the best achieving GAN, we added the hard constraints penalty and notice the relevance of the λ scaling parameter. For this reason, we focused on finding the best hyperparameter that produced the highest percentage of valid solutions.
- After finding the best λ parameter and GAN architecture, our goal was to quantify the increase of benefit in the percentage of valid solutions produced by the generator.

  This improvement will be measured comparing the results produced by the GAN architecture with the penalty and the more classical approach.

5.2 Methodology

This section is devoted to explaining the methods we use to test our hypotheses. In the first section 5.2.1, we explain the architecture used for the Generative Adversarial Networks with and without constraints. In the second section, we give a high-level overview of some of the hyperparameters that we used in our experiments.

5.2.1 Experimental settings

After trying different configurations in order to find the best achieving one that enables us to improve the percentage of valid solutions, we found the following architecture to achieve the best results.

In Figure 5.1 we can see the architecture of the training of a normal Generative Adversarial Network. There is a sample of the real images and a sample of the generated. The generated images come from a noise vector in order to produce each time different solutions. The discriminator agent receives samples from both sides and determines whether if it comes from the real distribution or the generated distribution.

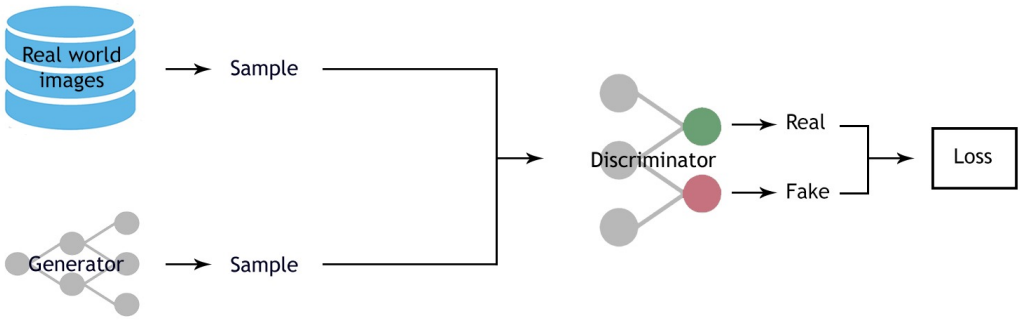


Figure 5.1: Architecture of the normal GAN trainning

In Figure 5.2, we can see that the new architecture of training that we propose for integrating hard constraints. The major change is that the penalty function is applied after choosing a sample from the real distribution and after generating a sample.

The penalty function is the one we cited before ($pen(M) = [M, M_P] = [M_S, M_M, M_F, M_P]$ 4.4) in the section of formalization 4. It concatenates another matrix to the incoming one, adding the penalty. At the end the information that the discriminator is receiving is the one from equations 4.5 ($M^*_{\mathbb{R}}$) and 4.6 ($M^*_{\mathbb{G}}$).

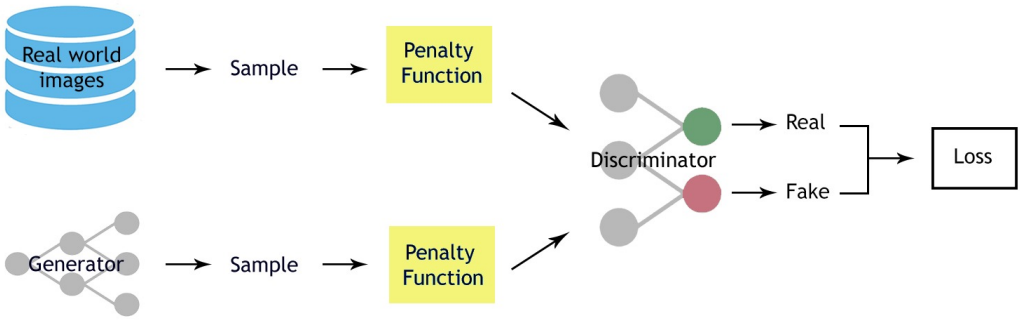


Figure 5.2: Architecture of the GAN with hard constraints

The application of the penalty provokes that the *discriminator* is able easily to distinguish between the data from the real and the data from the generated distribution, since there is a matrix that initially is filled with information (M_P) when it comes from the generated distribution and it is empty when it comes from the real distribution. We have tried in many different ways to address this problem of comparing the two data distributions. The one that achieves the best results is again the Wasserstein GAN with gradient penalty.

One of the reasons why WGANs with gradient penalty have better performance is because of the **vanishing gradient** or also called **saturation problem**. This due to the fact that the Discriminator learns easily at the beginning whether the image is fake or not just by looking at the M_P matrix. In this case, the gradient in which the Generator has to train is diminished to such a point that the Generator cannot usefully learn from it. Ian Goodfellow [8] claims that this problem is caused by the discriminator successfully rejecting generator samples with high confidence so that the generator's gradient vanishes. With Wasserstein Generative Adversarial Network with Gradient Penalty, the saturation problem does not take place and allows the generator to keep learning even if the Discriminator is already trained.

5.2.2 Parameter setup

One of the most important things to do is choosing the hyperparameters. Hyperparameters are the variables that we set before applying any learning algorithm to a given dataset. There is not a principled and general way of choosing hyperparameters, whose values are selected depending on the specific domain. For the MoonBoard we selected the following hyperparameters:

lr = 0.0002 The Learning Rate is the most important parameter to tune, is the step that the learning algorithm does at every iteration.

batch_size = 128 Refers to the number of samples used in each iteration.

betas = (0.5, 0.999) Control the decay rates of the moving averages of the ADAM optimization algorithm.

n_epochs = 300 The number times that the learning algorithm will work through the entire training dataset.

n_critic = 5 The number of iterations of the critic per generator iteration

latent_size = 64 The size of the generator random input.

start_max = 2 Number of maximum start holds

start_min = 1 Number of minimum start holds

finish_holds = 1 Number of possible finish holds

gp_param = 10 Gradient penalty hyperparamter for the WGAN-GP

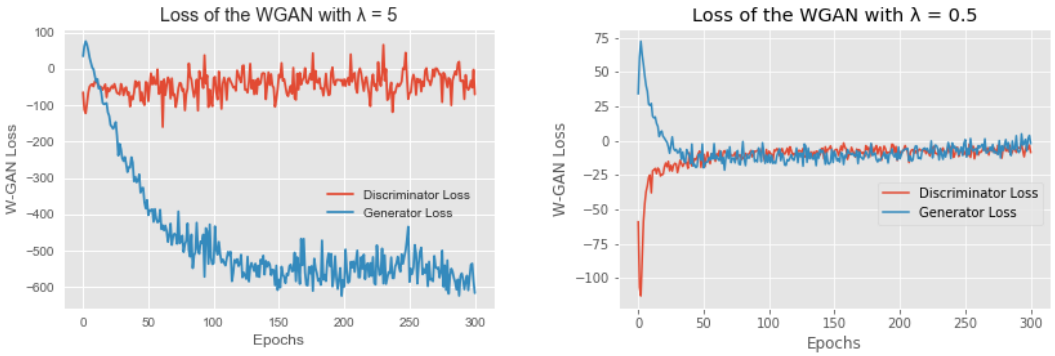
As we can see there a lot of hyperparameters, so choosing exactly the one that gives the best performance is a difficult task. We empirically tried many of them and ended up selecting the best-performing ones by hand.

5.3 Results

Along this section, we show the obtained results from the conducted experiments that we have explained in the previous sections.

5.3.1 GAN Collapse

While performing experiments with the λ parameter we found that if we exceed $\lambda > 1$, the training of the Discriminator and the Generator diverged.



(a) (b)

Figure 5.3: Figure (a)and Figure(b) shows the evolution of the loss of the WGAN-GP with different penalty parameters

In Figure 5.3, we can see two graphs. They represent the evolution of the loss of the Discriminator (4.7) and Generator (4.8) agents over 300 epochs. The scaling parameter for the hard constraints is $\lambda = 5$ for the left one and $\lambda = 0.5$ in the right one. We can see that in the left one the training diverges, while in the right one the training of the Discriminator and the Generator converge. Obviously, when training Generative Adversarial Networks we want that the loss of both agents converges.

We empirically found that for $\lambda > 1$ the loss diverges independently of the architecture we are using. We can say that we have found an upper bound for our λ parameter. For this reason, we consider that all values for $\lambda > 1$ are not worth further consideration.

This result is in accordance with the range of values of the rest of the matrix. The rest of the matrix is within the range of [0,1] if we add a λ parameter $\lambda > 1$ the penalty matrix (M_P) has bigger values than the rest of the matrix and this effect blocks the neural network for learning.

5.3.2 Sensitivity Measurements for λ

The penalty function adds an extra matrix to the image, as explained in the Formalization section (4). We added a scaling parameter (λ) in order to help us scale the range of values in which the penalty is contained. We found that the training behaviour was extremely sensitive to the scaling parameter λ of the penalty.

In order to evaluate the sensibility of this parameter, we trained the generator (300 epochs) and gather a set of solutions. From this set of solutions produced with the trained generator, we measured what is the percentage of solutions that satisfy the constraint.

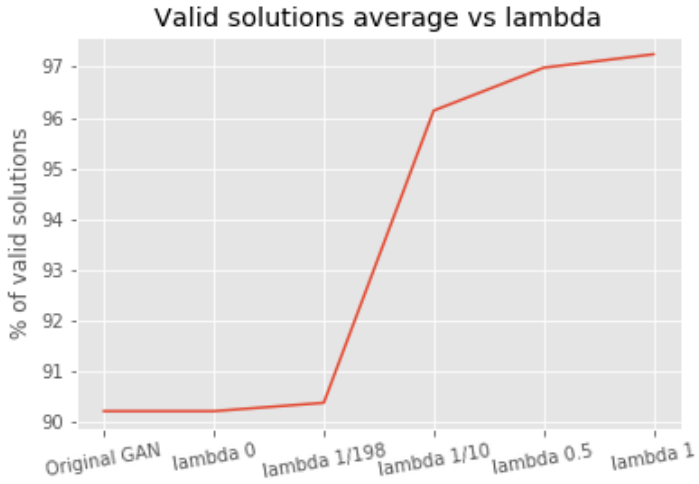


Figure 5.4: Percentage of valid solutions for different lambda parameters

With small lambdas $\lambda < 1/50$ we obtain almost the same value than with no channel penalty. While hyperparameter tunning we found that if we used $\lambda > 1$ for the channel penalty, the loss of the discriminator and the generator does not converge. We can see that in Figure 5.3a while for $\lambda < 1$ the loss of both agents (discriminator and generator) converges while achieving optimal performance of valid solutions.

From the results for $\lambda > 1$, we understand that the GAN has problems learning the properties of the data when one of the matrices has another dimension range of value than the others. Ans from the results for smaller λ, we can say that the optimal is above $\lambda > 0.5$. After this hyperparameter search, we can conclude that the best λ parameter for our problem is within the threshold of $0.5 < \lambda < 1$.

5.3.3 Validity

One of the main goals of this master thesis was to improve the percentage of solutions proposed by a GAN that satisfy a set of constraint. For that, we first selected the right architecture configuration (5.2.1) and the bests hyperparameters (5.2.2) for the Generative Adversarial Networks and then we encoded hard constraints as an input. The improvement of this encoding can be seen in Figure 5.5. Where we are comparing the percentage of valid solutions proposed by the GANs with and without hard-constraints penalty.

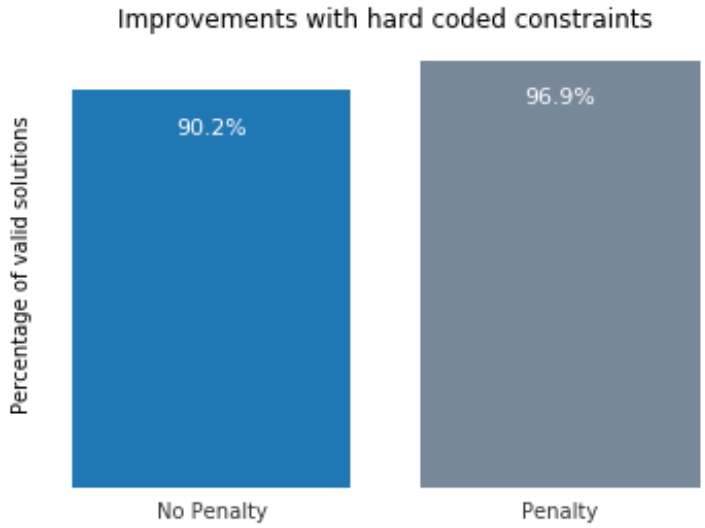


Figure 5.5: Improvements of hard coded constraints in Generative Adversial Networks

In order to compare, we trained both GAN architectures for 300 epochs and generated over 10,000 solutions. From this sets of solutions we measured the ones that satisfy the hard constraints and calculated the percentage of valid solutions between each.

We were able to achieve an improvement of 7% when the Generator agent is already trained. In Figure 5.5 we made a bar plot where we compared the original W-GAN GP with no penalty and the modified W-GAN GP with hard constraint penalty.

In Figure 5.6, we can see the evolution of the percentage of valid solutions over the epochs. We can notice that already in the early stages of the training there is an improvement of the percentage of valid solutions. In less than five epochs the GAN with penalty is able to achieve a +80% of valid solutions, where the normal

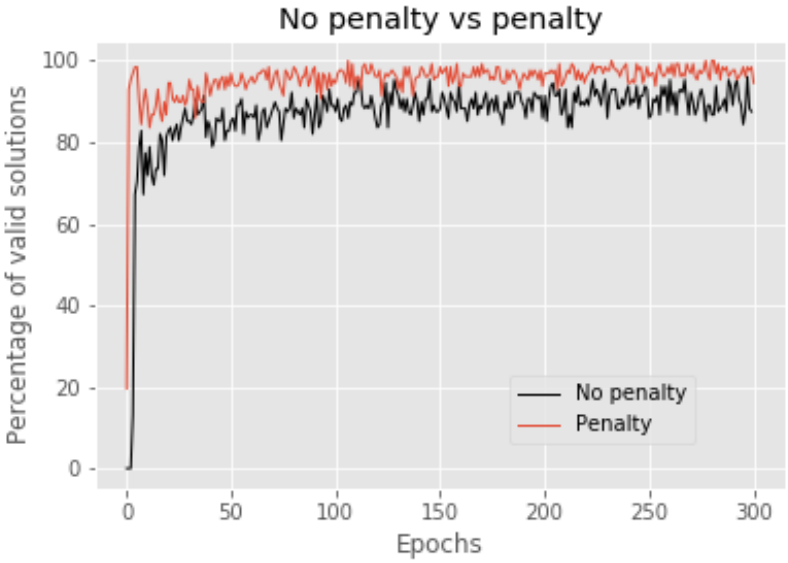


Figure 5.6: Percentage of valid solutions of the GAN with hard constraints penalty and without

GAN needs around 25 epochs. Also the percentage of feasible candidates end up converging at a higher rate than the GAN without penalty.

With the Figures 5.5 and 5.5 we can prove the validity of our experiments. The change in the training architecture and in the loss function allows us to incorporate prior knowledge as a hard constraint.

6 Conclusions

This chapter dissects the conclusions derived from this master thesis. It summarizes the contributions that this thesis presents.

6.1 To improve GAN exploiting hard constraints

Our hypothesis that we could add prior knowledge as a hard constraint in order to improve the percentage of valid solutions proposed by the Generator agent was confirmed. We were able to improve the number of solutions proposed by the Generator by a 7%, in a problem that already has a high performance (+90%).

One of the bigger problems of neural networks is the hyperparameter search and tuning. It is difficult to know before hand which are the most suitable hyperparameters. Most of these parameters are selected in an empirical way of try and error. For our case study, it was difficult to find a scaling parameter for the added penalty since it is not in the right range the neural networks either they do not learn anything or they saturate the training. So choosing the right scaling hyperparameter (mostly by empirical trials) it's one of the biggest throwdowns/achievements for experiments.

6.2 To the encoding of the MoonbBoard

This problem has never been treated before by the optimization community and we were able to encode it in order to give it a computational and mathematical approach. This achievement allows everyone to treat the problem and use it for computational purposes.

6.3 Empirical analysis of GAN comparison

There are multiple training architectures for Generative Adversarial Models in the literature. We tested the most promising ones and selected the one that achieves the best results for our problem. Our approach was able to achieve the best results with a percentage of valid solutions up to the 90%.

7 Future Work

While this thesis has aimed to improve generative adversarial networks with prior knowledge via hard constraints, its results also open new unexplored paths for future work. In this section, we present two promising paths for future research.

Extend the algorithm for general satisfaction constraint problems

We have implemented a training for Generative Adversarial Networks that allows us to implement a hard constraint and improve the solutions generated. A next research topic that we could pursue is to generalize the code so any satisfaction constraint problem could be applied.

An example of this could be training an algorithm is, for example the 8 queens, and adding a set of constraints in the penalty function. The idea is to increase the percentage of valid solutions proposed by the generator.

Incorporate our generative approach into state-of-the-art constraint optimization algorithms

As we explained in section 1.2, we can demand two main components for candidates in an optimization problem: being feasible (they satisfy all hard constraints) and being optimal (they are close to a good solution)

We could apply traditional combinatorial optimization for the candidates proposed by the Generator that is able to pick the optimal one from a set of already feasible candidates. As in the example of Knapsack problem, explained in the introductory chapter (1.1), once we generate feasible solutions (below a certain weight) we select the ones that have more items (a better performance).

Bibliography

[1] S. A. Barnett. Convergence problems with generative adversarial networks (gans). https://arxiv.org/abs/1806.11382, 2018.

[2] Pranav Dar. 25 open datasets for deep learning. https://www.analyticsvidhya.com/blog/2018/03/comprehensive-collection-deep-learning-datasets/, 2017.

[3] Victor Lempitsky Dmitry Ulyanov, Andrea Vedaldi. Instance normalization: The missing ingredient for fast stylization, 2016.

[4] Tang X Dong C., Loy C.C. Accelerating the super-resolution convolutional neural network., 2016.

[5] Jinoh Kim Sang C. Suh Ikkyun Kim Kuinam J. Kim Donghwoon Kwon, Hyunjoo Kim2. Salgan: visual saliency prediction with adversarial networks. https://arxiv.org/pdf/1701.01081.pdf, 2017.

[6] Georgios Chalkiadakis Sarvapali D Ramchurn Filippo Bistaffa, Alessandro Farinelli. A cooperative game-theoretic approach to the social ridesharing problem. https://doi.org/10.1016/j.artint.2017.02.004, 2017.

[7] Hongsheng Li Shaoting Zhang Xiaogang Wang Xiaolei Huang Dimitris Metaxas Han Zhang, Tao Xu. Stackgan: Text to photo-realistic image synthesis with stacked generative adversarial networks. http://openaccess.thecvf.com/content_iccv_2017/html/Zhang_StackGAN_Text_to_ICCV_2017_paper.html, 2017.

[8] Mehdi Mirza Bing Xu David Warde-Farley Sherjil Ozair Aaron Courville Yoshua Bengio Ian J. Goodfellow, Jean Pouget-Abadie. Generative adversarial nets, 2014.

[9] Geoffrey E. Hinton Jimmy Lei Ba, Jamie Ryan Kiros. Layer normalization. https://arxiv.org/abs/1607.06450, 2016.

[10] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014. cite arxiv:1412.6980Comment: Published as a conference paper

at the 3rd International Conference for Learning Representations, San Diego, 2015.

[11] Soumith Chintala Martin Arjovsky and Leon Bottou. Improved training of wasserstein gans, 2017.

[12] Soumith Chintala Martin Arjovsky and Leon Bottou. Wasserstein gan. https://arxiv.org/abs/1701.07875, 2017.

[13] MoonBoard. what is the MoonBoard? https://www.moonboard.com/, 2019. [Online; accessed 20-May-2019].

[14] NewYorkTimes. The new york times: Ai art at christie's sells for 432,500. https://www.nytimes.com/2018/10/25/arts/design/ai-art-sold-christies.html, 2018.

[15] PaperSpace. Intro to optimization in deep learning: Momentum, rmsprop and adam. https://blog.paperspace.com/intro-to-optimization-momentum-rmsprop-adam/, 2019. [Online; accessed 22-May-2019].

[16] Joan Serra Santiago Pascual, Antonio Bonafonte. Segan: Speech enhancement generative adversarial network. https://arxiv.org/pdf/1703.09452.pdf, 2017.

[17] Towards Data Science. A look at gradient descent and rmsprop optimizers. https://towardsdatascience.com/a-look-at-gradient-descent-and-rmsprop-optimizers-f77d483ef08b/, 2019. [Online; accessed 25-May-2019].

[18] Sierra. How to boulder in a gym: Indoor bouldering basics. https://www.sierra.com/blog/climbing/rock-your-gyms-climbing-wall/, 2019. [Online; accessed 20-May-2019].

[19] A survey of deep learning-based network anomaly detection. Salgan: visual saliency prediction with adversarial networks. https://link.springer.com/content/pdf/10.1007/s10586-017-1117-8.pdf, 2017.

[20] Jordi Torres. First contact with deep learning. https://torres.ai/first-contact-deep-learning-practical-introduction-keras/, 2018.

[21] Walter H. Pitts Warren S. McCulloh. A logical calculus of the immanent in nervous activity. http://www.cse.chalmers.se/~coquand/AUTOMATA/mcp.pdf, 1943.

[22] Wikipedia. Grade (bouldering). https://en.wikipedia.org/wiki/Grade_(bouldering, 2017.

[23] Jianxin Wu. Introduction to convolutional neural networks. https://pdfs.semanticscholar.org/450c/a19932fcef1ca6d0442cbf52fec38fb9d1e5.pdf, 2017.

[24] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. https://arxiv.org/pdf/1708.07747v1.pdf, 2017.

[25] Andrea Lodil Yoshua Bengio and Antoine Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. https://arxiv.org/abs/1811.06128, 2018.