POLITECNICO DI MILANO

Dipartimento di Elettronica, Informazione e Bioingegneria

Corso di Laurea Magistrale in
Ingegneria Informatica

DOOM Level Generation using Generative Adversarial Networks

Relatore:
Prof. Pier Luca Lanzi
Correlatore:
Dr. Daniele Loiacono

Tesi di Laurea Magistrale di:
Edoardo Giacomello
Matricola n. 850136

Anno Accademico 2016-2017

Abstract

This work studies the feasibility of level generation for First Person Shooter games using Generative Adversarial Networks in the setting of Procedural Content Generation via Machine Learning (PCGML). As the Procedural Content Generation becomes a widely used technique in developing video-games, many researchers explored new paradigms for generating game content based on generative models that learn from existing content. So far, few work has been done in applying these novel techniques to games that allows a two-dimensional exploration of the levels rather than a one-directional traversal that is typical of platform games. Our study proposes as a starting point in applying Generative Neural Networks to the problem of generating this type of two-dimensional levels, using DOOM as our sample game. In this work, we first expand the existing Video Game Level Corpus proposing a dataset of 9000 DOOM levels collected from the community and we design a system for extracting a set of features and representing the levels as images and tiled representation. We then introduce some of the main issues in applying existing techniques to this particular domain, selecting a set of measures that help in assessing the perceived sample quality in our case. We train two different models which differ from the presence of input features and we design a set of experiments to evaluate the impact of the input features on the generated levels. In addition, we study the possibility of controlling the level generation by acting on the network inputs. Our results show the advantages of adding the level features as a network input from both the point of view of sample quality and learned feature distribution, indicating our method as a viable option for being utilized as an automatic tool for level generation which doesn't require particular domain expertise.

Estratto in lingua Italiana

Questo lavoro analizza la fattibilità della generazione di livelli per videogiochi "Sparatutto" utilizzando le Reti Antagoniste Generative (GAN), nel contesto della Generazione Procedurale di contenuti attraverso l'apprendimento automatico (PCGML). Con la crescente applicazione di tecniche di Generazione Procedurale nell'industra videoludica, molti ricercatori stanno esplorando nuovi paradigmi per generare nuovi contenuti utilizzando modelli che apprendono da contenuto pre-esistente. Fino ad ora, poco è stato realizzato in merito all'applicazione di questo nuovo paradigma a giochi che permettono un'esplorazione bi-dimensionale dei livelli, in contrapposizione al classico attraversamento monodirezionale tipico dei giochi "Platform". Il nostro studio si pone come punto di partenza per l'applicazione delle Reti Antagoniste Generative al problema della generazione di questo tipo di livelli bi-dimensionali, in particolare utilizzando il gioco DOOM come riferimento. Come primo passo, proponiamo un dataset di 9000 livelli di DOOM collezionati dalla comunità di videogiocatori in aggiunta a quelli già forniti dal Video Game Level Corpus, progettando un sistema per estrarre automaticamente un insieme di caratteristiche descrittive dei livelli e convertire gli stessi in immagini. Proseguiamo selezionando un insieme di misure che agevolano la valutazione oggettiva della qualità dei livelli generati, evidenziando le principali difficoltà nell'applicare le tecniche esistenti al nostro dominio applicativo. Ottimiziamo quindi due reti diverse, che differiscono per la presenza in ingresso delle caratteristiche dei livelli e definiamo un insieme di esperimenti per valutare l'impatto che gli input aggiuntivi hanno sulla generazione dei livelli. Inoltre, analizziamo la possibilità di controllare la generazione dei livelli agendo sugli input aggiunti alla rete. I risultati ottenuti mostrano i vantaggi dell'aggiunta delle caratteristiche dei livelli come input alla rete, sia da un punto di vista della qualità dei campioni generati che dalle distribuzioni apprese, indicando che il nostro metodo può essere utilizzato come una opzione valida per generare livelli senza la necessità di particolare esperienza nel dominio applicativo.

A Cristina, per avermi sempre supportato e dato i giusti consigli.
A mio padre, per avermi sempre insegnato ogni cosa.
A mia madre, per tutto ciò che fa.

Acknowledgments

I would like to thank my supervisors Prof. Pier Luca Lanzi and Dr. Daniele Loiacono for their helpfulness in providing both valuable suggestions and technological support for allowing me to develop this work. I also would like to thank Cristina Anacleti for her suggestions on the editing of this thesis, as well as my family that supported me during all my life and in particular in the last months. Lastly, I would express my gratitude to everyone that will read my work and all the authors and designers that in the latest years produced great video-game content for the community, laying the ground for this work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Content creation in video-games is one of the most time consuming and difficult tasks in the process of developing a good quality product and producing interesting content often requires design experts. Content of a video-game belongs to two categories: Functional content is related to the game mechanics, in contrast to Non-Functional content, which usually serves a cosmetic purpose or has marginal effect on the game from a point of view of the player actions. In this work we only consider the problem of level design, which belongs to the first category.
Level design is an essential part in the development of many video-game genres such as Platform Games and First Person Shooters; in many cases a good level design contributed to the enormous success of many video-games.
Besides the costs of level design, other problems arose in the early history of video games: often the memory resources were scarce and the content couldn't be stored in memory. *Procedural content generation* came up as a solution to solve this issue by generating levels and other content by means of an algorithm rather then an human designer.

Many early games used Procedural Content Generation for overcoming the memory limitations on the machines that were used to play [56]. Notable examples in the early history are *Elite*[9], a space simulation in which procedural generation is used to create the game universe and *Rogue*[12], a dungeon-crawling game in which dungeon rooms and the hallways are generated by means of an algorithm.
With the increase of computing capabilities over time, the problem of storage became less severe. Nonetheless, PCG remained as a feature in many video-games, often playing a central role in the design of many games. For example in *Diablo*[7] every map and item is procedurally generated and many other games utilize software for automatizing some processes that would be extremely expensive if done manually, like populating an area with vegetation [64].
Recently, PCG have been often applied to increase *re-playability*: if a game is played many time, the game experience could be always different. An example is given by *Minecraft*[49], in which an initial part of the world is procedurally generated at the start of the game and it is expanded basing on the world seed as the player explores new areas. Other titles that make use of procedurally generated content as a fundamental design tool are *Dwarf Fortress*[2], *Elite: Dangerous*[13] and *No Man's Sky*[24], only to cite some of them.

Thanks to the increasing interest that machine learning topics gained in recent times, it is possible to apply new methodologies to the problem of content generation. In particular, Summerville et al. propose a new practice called PCGML (Procedural Content Generation via Machine Learning) [58] as the generation of game content by machine learning models that have been trained on existing content. This type of approach differs from "classical" procedural content generation because it does not imply a search in the content space, but the model directly generates the content. Summerville et al. propose in their article a survey on the work that has been already done in the field and present many possible applications of PCGML.

1.1 Scope

In this work we study the applicability of Generative Adversarial Networks to the problem of generating new maps for the First Person Shooter game DOOM in the context of Procedural Content Generation via Machine Learning. We propose an alternative model to the classical *procedural content generation*, inheriting the advantages introduced by PCGML. This allows creation of new levels without the need of having an human expert to embed their knowledge during the process, but exploiting patterns in training data instead. Our work takes place in the domain of first person shooter maps for which few work has been done yet using this type of models, since they differ from the commonly used platform maps which exhibit a sequential structure and a linear traversal. To assess the usability of this model in a real generation environment we study the advantages of adding input parameters in the form of level features, which allows to customize the generated levels. For studying the capabilities of the network models we produced we design three experiments that shows how the presence of input features affects level generation.

1.2 Related Work

1.2.1 PCGML in Video-Games

Summerville et al. show that a good amount of work has been done with different use cases, methods and data representation [58]. However, the domain in which the majority of work related to level design is done is the one of platform games. For example, Dahlskog, Togelius, and Nelson [11] uses *n-grams* to generate new levels for *Super Mario*[47], Jain et al. use *autoencoders*, while Snodgrass and Ontañón experiment an approach based on Markov Chains [57]. An exception to this type of domain that is still related to level generation is the work of Summerville and Mateas in [59] where the authors present a method for generating levels of *Legend of Zelda*[48] using Bayes Nets for the high level topological structure and Principal Component Analysis for generating the rooms. Our work proposes instead a method for generating the whole level topology using a single model, with the possibility of easily adding more features or eventually applying the same structure to another dataset.
In their work, Lee et al. [41] use neural networks for predicting resources location in *StarCraft II*[19] maps. Although the data domain is similar to the one used in our work, the problem only focused on resource placement rather then map topology generation and requires the image of an already existing level as input. Moreover, we make use of Generative Adversarial Networks, which is a particular setting in which a generator is able to produce new samples using a vector of noise as input.
In the settings of Generative Adversarial Networks, Beckham and Pal propose a method for generating realistic height level maps for video-games [5], which is more applicable to realistic landscapes rather than fictional indoor environments such those of First Person Shooters. This kind of model have also been applied to non-functional content generation in the work of Horsley and Perez-Liebana, in which GANs are used to generate 2d sprites [31].

1.2.2 Video Game Level Corpus

One of the problems with this type of generative models, as explained in [58] is that they require a large amount of data to be optimized. Unfortunately, the domain of video-games levels does not benefit of large datasets to work with, and generally levels from different video games does not share common data structures. Summerville et al. created the *Video Game Level Corpus (VGLC)*, a collection of game levels represented in multiple formats. Using this format as starting point, we collected about 9000 user-generated Doom levels of different size and designed an extended representation that better fit our needs, while still keeping the dataset compatible with the original VGLC representation. Although VGLC provides the parser they

used for data generation, we wrote a new parser which better integrates with our system and feature representation, and can also be used as a stand-alone parser for future researches.

1.2.3 Generative Adversarial Networks

Generative Adversarial Networks are a recent generative model based on Artificial Neural Networks. This type of model allows to learn the data distribution of a dataset and generate synthetic data that exhibit similar characteristics to the real data. Among all the domains in which GANs have been already applied, that of images is one of the most prominent. For example, generation tasks are commonly applied to the handwritten digits (MNIST [40]) dataset, human faces (CelebA [43]) and bedrooms (LSUN [70]) as in [53], but a large amount of creative work is done with many other datasets such as birds, flower [71], and other type of images. Another task which involves pictures is image-to-image translation: Isola et al. investigates GANs as a general solution to this problem in several settings [34] such as image colourisation, segmented image to realistic scene conversion or the generation of realistic objects starting from hand-drawn input from the user (Figure 1.1). The GAN approach has been also used in many other domains such as frame prediction in videos [45] and sound generation [69], being a research area in rapid expansion.

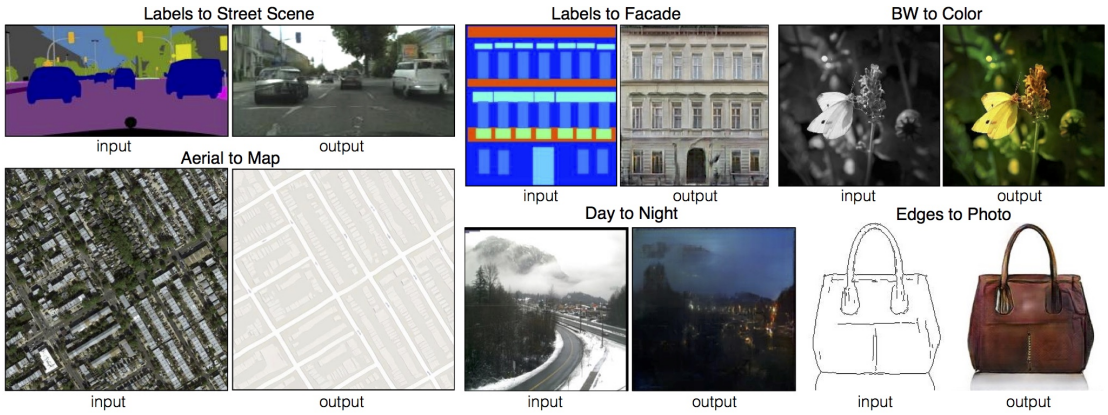

Figure 1.1: Examples from the work of Isola et al. on image-to-image translation problem [34].

1.3 Thesis Structure

Chapter 2 describes the theory which is needed to understand how we designed our system and motivates the choice of DOOM as the game we used in this work. Chapter 3 describes in detail the data we used in our work and how we converted it in order to make it functional for our system. Chapter 4 first describes the system from an high level perspective defining the possible use cases and processes, then detail the neural network model we designed for our system and describes some additional metrics we designed to monitor the training process. Chapter 5 shows what choices are to make for designing an experiment using our system, then defines three experiments which let us study the trained models. Chapter 6 reports and discuss the most relevant results obtained during the training process and by running the experiments. Chapter 7 shows our general considerations about our work, while highlighting the open problems and proposing further work on this topic.

1.4 Summary

In this chapter we introduced the problem of Level Design in video-game industry and the solutions that have been historically applied for approaching the problem. Then we referenced a new area of research in which this work take place and defined its scope. We also described the main differences between our research domain and the domains of the main contributions in each research area we considered, highlighting the differences between our work and other contributions. In the next chapter we give a more in-depth description of the Generative Adversarial Network models and considerations about the choice of the game.

Chapter 2

Theory and Motivation

This chapter introduces the theoretical aspects our work is based upon, while giving the reader some overview of the tools and techniques we applied in our system. Section 2.1 provides the theoretical background necessary to better understand the setting in which our work take place. Section 2.2 describes and motivates the choice of the game we used in our work.

2.1 Theoretical Background

As the main model for generating levels we selected the Generative Adversarial Networks, or *GAN*. This model showed good results in many applications and it's increasingly more adopted and studied in the research community. Due to this, a large variety of different architectures and variants are being designed in order to improve the original model. In this section we present an introduction to GANs and the main variants we considered for selecting the final model.

2.1.1 GAN

Generative Adversarial Network, proposed by Goodfellow et al. [26] in 2014, are a model that gained gained increasingly more interest in the latest years. The main idea of this kind of generative model is to use two neural networks which are posed in an adversarial setting that models a two-player Minimax Game [18, p. 276]. In particular, a generative network G is trained to capture the data distribution while a discriminator network D estimates the probability that each sample comes from the real data distribution rather than the one generated by G. Equation 2.1 shows the loss functions for D and G in the original GAN architecture[26]:

$$\begin{aligned} L_D^{(i)} \leftarrow & \frac{1}{m}\sum_{i=1}^{m}\left[\log D\left(x^{(i)}\right) + \log\left(1 - D\left(G\left(z^{(i)}\right)\right)\right)\right] \\ L_G^{(i)} \leftarrow & \frac{1}{m}\sum_{i=1}^{m}\log\left(1 - D\left(G\left(z^{(i)}\right)\right)\right) \end{aligned} \tag{2.1}$$

where $z^{(i)}$ is a batch of random noise samples and $x^{(i)}$ is a batch of true data samples. The two networks are trained alternately by *Backpropagation*, in this way the generator network can learn to improve the generated sample quality by using the discriminator output as a sort of feedback.

2.1.2 Deep Convolutional GAN (DCGAN)

Typically, training a Generative Neural Network is a difficult task. This is, among other reasons, due to the need of balancing the generator and the discriminator during the minimax optimization. In order to overcome these difficulties, Radford, Metz, and Chintala proposed the DCGAN model [53] which offers some improvements over the standard GAN architecture. The main modifications introduced are the use of convolutional layers instead of fully connected and max-pooling, the use of batch normalization [33] and ReLu activation functions [50]. DCGAN architecture became one of the baselines to build projects involving GANs and to compare new architectures. In our work we used the DCGAN layer structure showed in figure 2.1 as a starting point for our experiments, which differs from ours for the presence of the conditioning input, the stride and the input size. These details on our architecture will be explained in chapters 4 and 5.

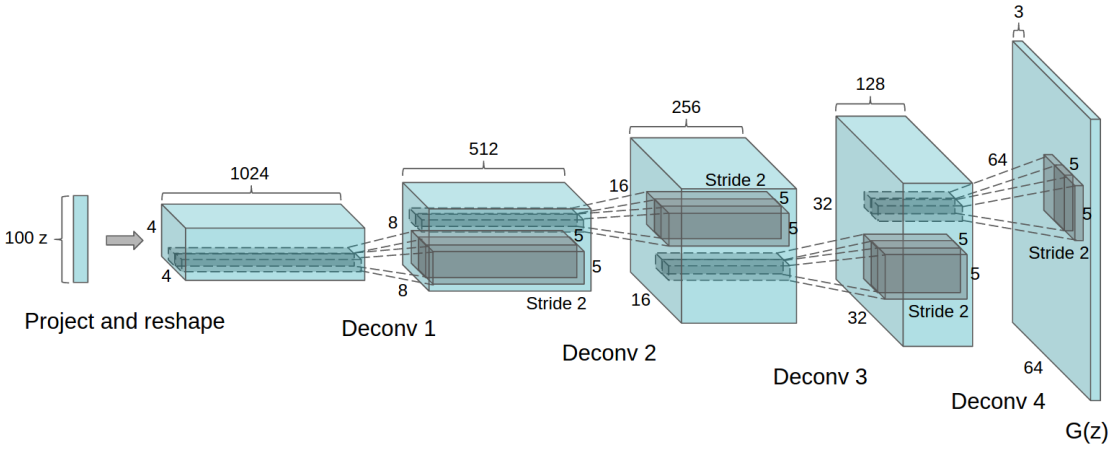

Figure 2.1: DCGAN generator layers from the work of Radford, Metz, and Chintala on the LSUN dataset [53, p.4].

2.1.3 Wasserstein GAN

Despite of the improvements, GANs are still affected by some problems: training stability is dependent on network structure, the training loss rarely converges and does not reflects the actual sample quality. The Wasserstein GAN architecture, introduced by Arjovsky, Chintala, and Bottou [3], aims to mitigate these problems: The authors build a new loss function, basing on the fact that training a GAN can be interpreted as minimizing the *Jensen-Shannon divergence*[42]. They also prove how using Wasserstein (or Earth-Mover) [3, § 3] distance is a more sensible choice since it provides smooth transitions as opposed to discontinuities in JS-Divergence when the supports of the two probability distributions begin to overlap.
The resulting loss function is an approximation for the Wasserstein distance:

$$\begin{aligned} L^{(i)}_{Critic} &\leftarrow \mathbb{E}(D(X_{Gen})) - \mathbb{E}(D(X_{True})) \\ L^{(i)}_{Gen} &\leftarrow -D(X_{Gen}) \end{aligned} \tag{2.2}$$

This approximation is derived from an alternative formulation of the Wasserstein distance, which requires to calculate a supremum over *K-Lipschitz* functions. To force the network to only model K-Lipschitz functions the weights of the critic network are clamped below a fixed value *(weight clipping)* and no output activation functions are applied to the Discriminator, for this reason called *Critic*.

2.1.4 Wasserstein GAN with Gradient Penalty

Gulrajani et al. remark in [28] that the WGAN architecture suffers from some optimization problems related to the weight clipping. Their experiment shows that gradient often vanishes or explodes if the weight clipping is not tuned carefully, and that this technique also biases the network toward too simple functions. To overcome these problems, they propose an alternative way to enforce the *Lipschitz constraint* by keeping the gradient at unitary size. This involves adding a penalty term to the critic loss to enforce the constraint only along straight lines between the real and the generated data distribution, which proves to produce good results and performances. Equation 2.3 shows the calculation of the gradient penalty which is added to the WGAN Critic Loss.

$$\begin{aligned} \hat{X} &\leftarrow \epsilon X_{True} + (1-\epsilon) X_{Gen} \\ G_p &\leftarrow (\|\nabla_{\hat{X}} D(\hat{X})\|_2 - 1)^2 \end{aligned} \tag{2.3}$$

Results obtained from Gulrajani et al. on the LSUN dataset are shown in Figure 2.2.

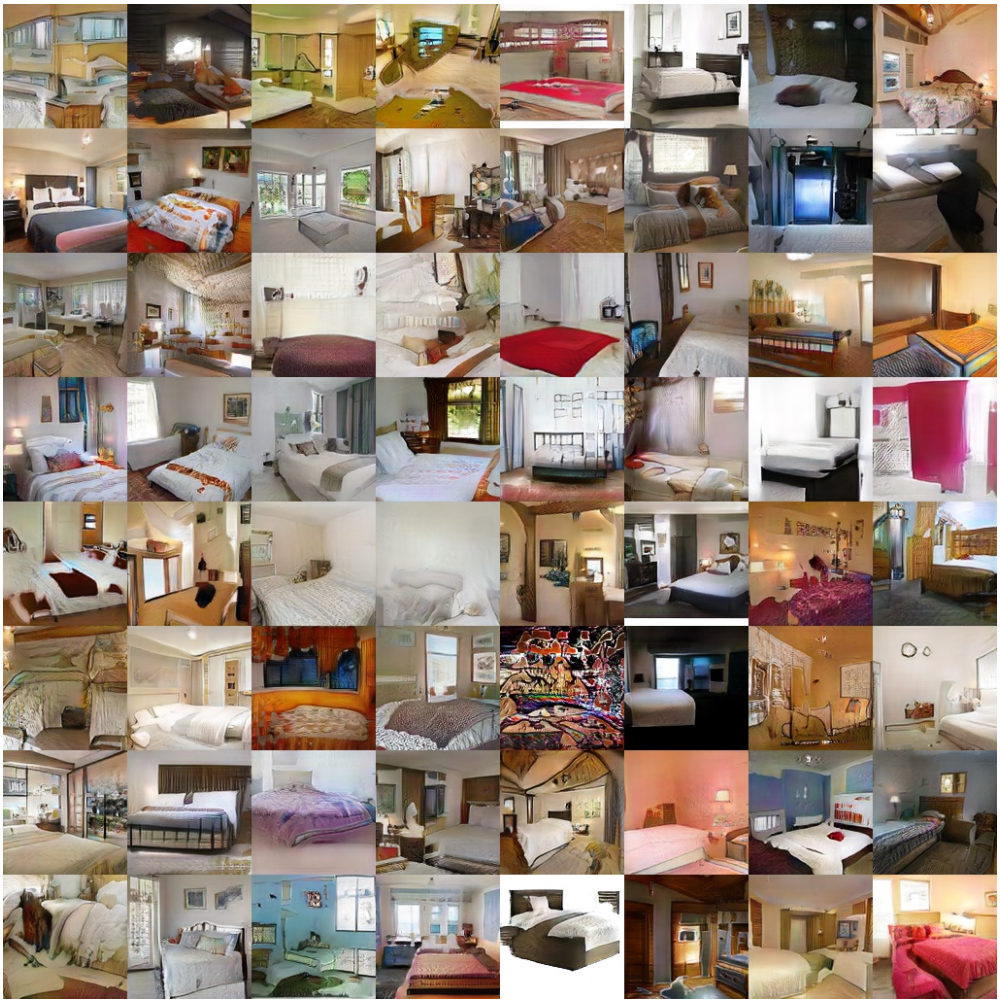

Figure 2.2: Examples from the work of Gulrajani et al. on the LSUN dataset using WGAN-GP. [28, p.8]

2.1.5 Conditional GAN

In our work we apply to WGAN-GP a modification of the GAN model introduced by Mirza and Osindero [46] , which allows to condition the data generation to class labels. The motivation we chose this model is to design a way to control with some extent the generation process, instead of generating purely random samples; this could also allow the network to learn useful information encoded in the feature vector and exploit them to generate better results. The logical structure of the adopted generative model is summarized in section 4.1.

Additional Architectures

Due to the novelty of this model, many researches have been conducted to improve the quality of GANs, leading to a constantly growing variety of proposed new architectures. They can be classified as modifications to the structure of the neural network or even to the theoretical setting of the model itself, by means of changes to the loss function or the training algorithm. Other, less formal but still effective adjustment to the proposed models are the so called "tricks" in machine learning discussion communities, which adoption is often suggested in order to improve the difficult training of the network. The final architecture adopted is described in section 4.4.1.

2.2 Game of choice: DOOM

2.2.1 Description

DOOM is a video-game produced *Id Software* in 1993, and can be considered one of the games that defined the First Person Shooter genre. Gameplay consisted in traversing a series of levels populated by several enemies, and the player had the possibility to collect weapons, ammunition and power-ups in order to reach the exit of the level. Often, in order to reach the exit point, the player had to explore the level in order to find some keys required to open the doors that blocked the path. DOOM levels are divided in episodes of 32 levels each. The first episode was released as shareware and *Id Software* greatly encouraged the diffusion of the game. This contributed to the celebrity of the game in a time in which internet was still not accessible to everyone.

2.2.2 Motivation

The domain we chose for this work is that of First Person Shooter levels, which usually require the player a more explorative approach as opposed to platform games that are typically traversed in a single direction. This different type of exploration usually requires an higher dimensional representation of the levels, which makes working with fps levels difficult due to multiple overlapping height levels. One of the peculiarities of this game engine is that the levels actually develop on a 2-D plane, with the height added separately. In other words, rooms cannot overlap on the height dimension, and often game designers used their skills to produce the illusion of the opposite. This makes DOOM a good game for researching on first person shooters levels while keeping a simple 2d representation.

Another important feature of the Doom Game Engine is that it provided native "mods" support. This fact encouraged many people to develop and exchange their levels for DOOM, and soon this produced a large database which is still expanding nowadays. Thanks to a large active community it is possible to easily find many resources and specifications to work with DOOM levels and understanding how the game engine works.

2.3 Summary

In this chapter we described some of the main Generative Adversarial Network architectures that are currently available in literature, starting from the standard GAN to the most used improvements. Then, we briefly described the motivations we selected DOOM as the game to work with. In the next chapter we give a more detailed explanation of how the Doom Engine works and levels are described, while in chapter 4 we give a logical description of how we implemented the GAN architecture in our system.

Chapter 3

Dataset and Data Representation

This chapter describes the structure of the database we built to train and evaluate our model, as well an overview of the processes required to generate the dataset itself. Section 3.1 gives a reference to the data sources, then section 3.2 shows how data is natively encoded for the game engine in order to introduce what are the difficulties in converting to and from the native format in an automatic way. Section 3.3 details how we encode data in the target format, how levels are converted and what features are extracted in order to provide an input for the neural network. Section 3.4 gives an overview of how the data is organized in the datasets we make available.

3.1 Data Sources

The data used to train and validate the model comes from the *Idgames Archive* founded in 1994 by Barry Bloom [4] and mirrored on various FTP sources. The mirror we used for collecting levels is Doomworld.com [15], which is one of the oldest and currently most active community about the DOOM video-game series [14].

Idgames archive includes levels for multiple games such as DOOM, DOOM 2 or their various modifications. They are divided in hierarchical categories, which classify levels by game, game mode (multi-player "deathmatch" or single-player), and alphabetically. Amongst the categories we selected only those levels that belong to "DOOM" and "DOOM 2", excluding sub-categories named "deathmatch" and "Ports". This choice has been made in order to avoid mixing different types of levels, since a level designed for a Single Player Mode could be structurally different from a level which is designed for multi-player games. Moreover, the "Ports" category has been excluded because it contains levels that are intended to work with modifications of the game engine code, and it would have led to problems in managing every particular exceptional behaviour.

Levels in Idgames archive are stored in zip archives, including a "READ ME" text file containing author notes and the WAD file that contains up to 32 levels. Each zip archive can be downloaded from the respective download page, which presents a variable quantity of information such as: author, a short description, screen shots, user reviews, number of views and downloads, etc. The dataset we present in this work always keeps track of these information about each level for correct attribution. It also offers a "snapshot" of average user review score and the number of views and downloads, when available. It is worth noting that since Doomworld website recently switched to a different download system [14], meta-data concerning downloads and view counts may not always be accurate, but they are still proposed as a starting point for further research.

3.2 Source Data Format: WAD Files

The Doom Game Engine [36] makes use of package files called "WADs" to store every game resource such as Levels, Textures, Sounds, etc. WAD files have been designed in order to make the game more extendible and customizable, and opened the way for a considerably large amount of user-generated content. This section is not meant to be a complete description of how WADs files are structured, but only an overview of which aspects we considered for writing the software that generates the dataset from the WAD files. Every information about WADs files has been taken from the *The Unofficial Doom Specs* [20] and we refer to that document a deeper explanation on every aspect of the file format.

3.2.1 Overview

Type of WADs

There are two types of WADs: the "IWAD", or "Internal WAD", and the "PWAD" or "Patch WAD". The original game files, called "DOOM.WAD" and "DOOM2.WAD", are of the "IWAD" type, as they contain every asset that is needed for the game to run. The WADs containing custom content or modifications to existing content are of the "PWAD" type. The content which is defined in a PWAD is added or replaced to the original IWAD when the WAD is loaded. For example, if a PWAD defines the level "MAP01", which is already defined in "DOOM2.WAD", the PWAD level is loaded instead of the original one, while maintaining all the other content unaltered. Since in our work we deal only with PWADs, we will generally refer to them simply as WADs.

Lumps

Every data inside a WAD is stored as a record called Lump, which has a name up to 8 characters and a structure and size that is different depending on the lump type. Generally there are no restrictions on lump order with the exception of some of them, including those needed for defining a Level.

WAD Structure

Every WAD file is divided in three sections: A header, a set of Lumps and a trailing Directory. The header holds the information about the WAD type, the number of Lumps and the location of the Directory, which is positioned after the last Lump. The directory contains one 12-bytes entry for each Lumps that specifies the Lumps location, size and name. Table 3.1 reports a simplified description of a WAD file.

Coordinate Units

The Doom game engine describes coordinates using integer values between -32768 and +32767, and it is proportional to one pixel of a texture. This unit is called "Map Unit" or "Doom Unit" in this work. Although there's not an unique real world interpretation of one Map Unit, we used an approximation for which each relevant map tile or image pixel is 32x32 MU large; this choice is motivated by the fact that the smallest radius of a functional object in DOOM is 16 MU.

3.2.2 Doom Level Format

Level data in a WAD file follows a precise structure. In particular each level is composed of an ordered sequence of lumps that describes its structure:

(NAME) : Name of the level slot in DOOM or DOOM2 format.

THINGS List of every game object ("Thing") that is placeable inside the level.

LINEDEFS List of every line that connects two vertices.

SIDEDEFS A list of structures describing the sides of every Linedef.

VERTEXES Unordered list of vertices.

SEGS A list of linedef segments that forms sub-sectors.

SSECTORS A list of sub-sectors, which are convex shapes forming sectors.

NODES A binary tree sorting sub-sectors for speeding up the rendering process.

SECTORS A list of Sectors. A Sector is a closed area that has the same floor and ceiling height and textures.

REJECT Optional lump that specifies which sectors are visible from the other. Used to optimize the AI routines.

BLOCKMAP Pre-computed collision detection map.

It is important to notice that although all the lumps above (with the exception of REJECT) are mandatory to build a playable level, some of them can be automatically generated from the remaining ones using external tools. In particular, an editor software or designer has to provide at least the lumps (name), THINGS, LINEDEFS, SIDEDEFS, VERTEXES and SECTORS. The lumps SEGS, SSECTORS, NODES, REJECT and BLOCKMAP serve the purpose of speeding-up the rendering process by avoiding runtime computation. In particular the Doom Engine uses a Binary Space Partitioning Algorithm [22] for pre-computing the Hidden Surface Determination (or occlusion culling), and it is usually done by an external tool. In this work we used the tool "*BSP v5.2*" [10] in the last stage of the pipeline, in order to produce playable DOOM levels from the network output. In the following paragraphs only the lump types that are not generated by the external tool are described.

(Name): The first lump of a level is its slot name. We indicate this lump between parenthesis because, differently from the other lumps, this one has no data associated. The Name field in the Directory is the slot name itself and the size is therefore zero. The level name descriptor has to match either *ExMy* ("Episode x, Map y") or *MAPzz* for DOOM or DOOM2 respectively, where x ranges from 1 to 4, y from 1 to 9 and zz from 1 to 32.

Things: A doom "Thing" is every object included in a level that is not a wall, pavement, or a door. A "Things" lump is a list of entries each one containing five integers specifying the *position* (x,y) in MU, the *angle* the thing is facing, the *Thing type* index, and a set of flags indicating in which difficulty level the thing is present and whether the thing is deaf, if it is an enemy.

Linedefs: A Linedef is any line that connects two vertices. Since DOOM maps are actually bi-dimensional, a single line is needed to define each wall or step. Linedefs do not necessarily have to match with visible entities, as each Linedef can also represent invisible boundaries or triggers, which can be thought as tripwires or switches that make something happen to a sector. If the Linedef acts as a trigger than it references another sector and has a type which defines what would happen to it when the Linedef is activated. For example, a door is implemented as a sector that raises up to the ceiling when a certain Linedef is triggered, but this behaviour is specified in the switch and not in the door as one might think. All the options are defined by a set of flags that specifies what kind of objects the Linedef blocks, if it's two sided or not, the trigger activation condition and so on. Finally, each Linedef has to specify what "vertical plane" (or Sidedef) lies on its right and left side. Only the "left sidedef" field can be left empty, due to the way the sectors are represented.

Sidedefs: A Sidedef is a structure that contains the texture data for each linedef. It corresponds to the side of each wall and it's referenced by the linedefs. Other than the options for texture visualization, it also specifies the sector number that this plane is facing, implicitly defining the sector boundaries.

Vertexes: [1] This is the simplest type of Lump, consisting in an unordered list of map coordinates in MU. Each entry has an x and y position expressed in a 16-bit signed integer. Linedefs references the starting and ending vertex by reporting their index in this list.

Sectors: A sector is defined as any area that have constant floor and ceiling heights and textures. This definition highlights the fact that the doom engine is in fact a 2d engine, since it's not possible to define a sector above or below another one. A sector does not necessarily have to be closed nor a single connected polygon, but non-closed sectors can cause some issues during gameplay. The only constraint to define sectors comes from the fact that linedefs must have a right sidedef, but the left one is optional; the sectors are normally described as (a set of) positively oriented curves [17], with the exception of linedefs that are shared with another sector that may be reversed. The fields of a sector lump includes the floor and ceiling height, the name of the floor and ceiling texture, the light level, the "type" to controls some lighting effects and damaging floor, and the tag number that is referenced by the linedef triggers. Figure 3.1 shows an example of a level with 3 sectors.

3.2.3 Conversion issues

Since 1994 many DOOM players started producing a large amount of levels and increasingly sophisticated editors came to light. This led to a notable variety in conventions, optimizations and use of bad practices that we tried to deal with in developing the Python module for reading and writing wad files. Some of these practices includes unnecessary lumps, random-data instead of null padding for names, and other inconsistencies or arbitrary conventions. However, we paid particular attention during the writing phase in order to precisely follow the specifications and avoid generating low quality WAD files as much as we could. Some other difficulties came with the necessity to render wad files as images and vice-versa. The first one is that neither Linedefs nor vertices came in an any ordered format, along with the fact that sector vertices are not explicitly defined, but only referenced through the linedef/sidedef chain. This means that for finding a sector shape one has to find all sidedefs with the desired sector number, then find all the linedefs referencing those sidedefs and lastly retrieving the vertices, leading to an increase of complexity. Another problem was the fact that although sectors must be positively oriented curves, it is not mandatory to use the left sidedefs for adjacent sectors, leading to duplicated linedefs in the opposite direction. Even some optimizations such as sidedef compression may be possible, that is referencing the same sidedef wherever the same wall texture is used, adding

[1] Although the correct spelling should be "vertices", we keep the original version of the field name.

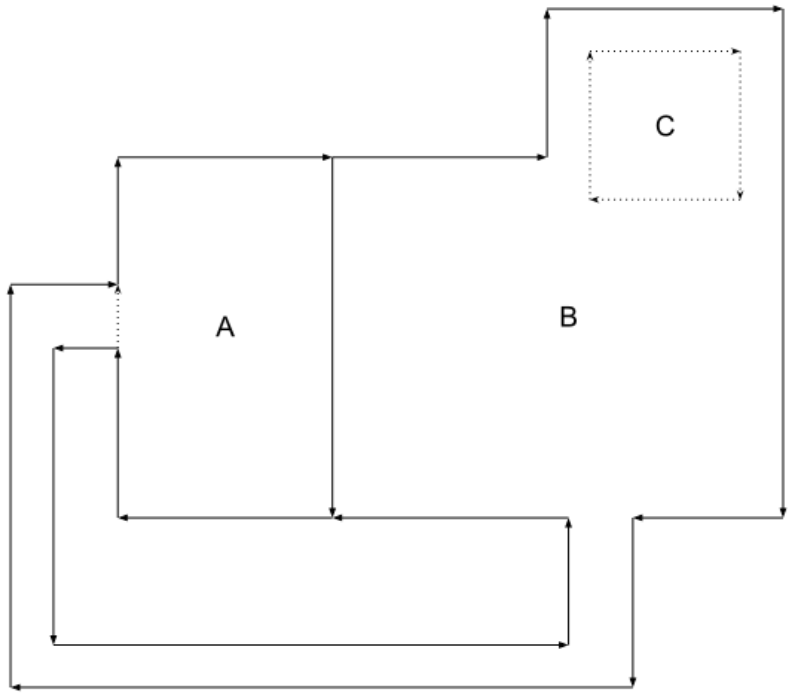

Figure 3.1: A simple level showing three sectors A, B and C and the linedefs defining them, following the positive oriented curve constraint. Sector C can be viewed as a small platform inside the sector B. Solid arrows represent walls, while dashed lines represent invisible linedefs or changes in height between two sectors (steps). In this level, every solid arrow is a linedef that specifies only a right sidedef, with the exception of the one separating sectors A and B that has both a right and a left one.

complexity to the conversion script. Moreover, the concept of sector and the concept of room are not equivalent: even though a sector is defined as an area of constant height, this does not enforce to define a sector only where height changes, so the semantic of a sector actually depends on the designer and the editor used.

Section Length (bytes)	Section Name	Field Size	Field Name	Description
12	Header	4	Identification	ASCII string "PWAD" or "IWAD"
		4	Number of Lumps	The number of Lumps included in the WAD
		4	Table Offset	Integer pointer to the Dictionary
Variable	Lumps	-	Lump Data	Lumps stored as a stream of Bytes
16 * Number of Lumps	Directory	4	Lump Position	Integer holding a point to the lump's data
		4	Lump Size	Size of the lump in bytes
		8	Lump Name	Lump name in ASCII, up to 8 bytes long. Shorter names are null-padded

Table 3.1: WAD File structure

3.3 Target Data Format: Feature Maps and Vectors

This section provides a description of the data format as it is stored in the level dataset.

3.3.1 Level Description and Motivation

Levels are read as a structured object by a module we wrote for the purpose of providing developers and designers a programmatic way to access, analyse and edit DOOM WAD files, instead of using visual authoring software. This allows to automatise some tasks that would be very long to accomplish with standard editors or tools and also offers developers the possibility to write custom editor and scripts.
In order to provide the most complete set of information possible every level is converted to a set of images, called Feature Maps in this work, a tiled representation in textual format that is an extension of the one used in *VGLC* [60], a graph representation and a set of textual and scalar features that contains both the WAD meta-data and the level features and metrics calculated either on the WAD representation (sectors, subsectors, linedefs, etc), the Feature Map representation or the graph representation of the level. A detailed list and explanation of each feature is provided in section 3.3.5.
The choice of these representations have been made primarily for the need of having a data format that the generator model can easily work on. In particular, convolutional neural networks are designed to work well with bi-dimensional or tri-dimensional data such as (multi-channel) images, for this reason the image representation arose naturally. The text representation is provided mainly for consistency with the data format given in [60], even if our representation uses two characters per tile instead of one. Finally, scalar and graph representation have been collected for the need of quantifying and summarising some properties and having a more abstract representation of a level, which can be very helpful in the case this data had to be used in other works.

3.3.2 Feature Maps

Feature Maps are a set of images each of them describing a different aspect of the level. In particular we used for each Feature Map a grayscale 8-bit image in which each pixel can assume values between 0 an 255. This allowed to obtain a good degree of precision while still maintaining a reasonable dataset size.
Because of the motivations explained in section 3.2.1, each pixel in a Feature Map corresponds to a square of 32x32 MU. In the following paragraph we will describe in detail each of them along with the data encoding.

FloorMap: The FloorMap is the most basic form of level representation, since it only represents which part of the space are occupied by the level and which are empty. This kind of map is often used in robotic mapping.
This map also describes approximatively the level area that is possible to traverse, since in DOOM walls have no thickness.
"Floor" pixels have value 255 (white) and "Empty" pixels have value 0 (black).

WallMap: The WallMap represent the impassable walls of the level. They are represented as a one-pixel-wide line and are obtained by directly drawing each Linedef with the impassable flag set on a black image.
Pixel values are 0 for Empty area or floors and 255 for the walls.

HeightMap: The HeightMap is another common map used for visualizing the height of a certain surface. Since height level in DOOM levels are completely arbitrary and virtually unbounded, we normalize each level between its lowest and highest height, assigning the pixel value

0 for empty parts of the level and the remaining are calculated from the formula $c_h = \lfloor h * \frac{255}{|H|} \rfloor$, where H is the ordered set of possible height values for the level and $h \in \{1, 2, ..., |H|\}$ is the index of the height value in H for which we want to calculate the encoded colour.
For example if a level takes the height values $H = \{0, 10, 15, 20\}$ they will be encoded respectively as $c_h = \{63, 127, 191, 255\}$ and 0 for the empty areas. Although this map loses the information about the differences between a height level and another, it has the advantage to represent "higher" and "lower" parts of the level without polarizing the entire map due to extreme levels: while the majority of the levels has a few changes that can be approximated as uniform (such as levels with stairs connecting a few rooms), other had some extreme changes in height but only for a small portion of the map (like a very high elevator leading to a small secret room) that led to scaling problems.

ThingsMap: The ThingsMap represent only data that is contained in the THINGS lump. It features a series of pixels placed at the thing coordinate, with a value that corresponds to a particular "thing". Pixel colours have been grouped by functional purposes so for example weapons occupies values that are close each other. This is for tolerating some output noise during generation without completely changing the functional aspect of an object as would have happened if we kept the original things encoding. Tables 3.2, 3.3 and 3.4 lists the complete encoding for the maps. Descriptions are taken from [63].

TriggerMap: The TriggerMap is used for representing linedef triggers and the sectors which activates. Due to the vast amount of cases the doom engine can handle, only a few types of triggers have been considered. The mapping works by assigning an integer $i \leq 32$ to every trigger object, and subdividing triggers types in 5 groups: local doors (the ones that are activable only if the players directly interact with them), remote doors, lifts, switches and teleports. Local doors can be normal or require a key of a certain colour in order to open, but they are not indexed by the trigger index since they don't require to be linked to other linedefs in order to be opened. Table 3.5 describes the encoding for each possible item i.

RoomMap: The RoomMap represent an enumeration of the rooms obtained with an algorithm that is very similar to the morphological approach used in "Room segmentation: Survey, implementation, and analysis"[8]: An euclidean distance transform [62] is first applied to the FloorMap obtaining a map that we call D̈istance Mapör D̈istMap;, then the local maxima are found [55] such that each maximum has a minimum distance of 3 from the closest one, resulting in the room center coordinates that are used as markers for a Watershed algorithm [51] using the negative distance map as basin. This results in a room segmentation that is good enough for descriptive purposes while maintaining good performances.

3.3.3 Graph Representation

Another way to represent the level is by using a graph. In particular, this graph is a region adjacency graph [66] built upon the RoomMap, where the nodes represent the rooms and the edges are the boundaries they have in common. This graph is built primarily for computing some features about the level and for exploiting its convenient representation of the rooms during the WAD Writing phase: this graph can be annotated with the coordinates of walls belonging to each room, and this information can be used to build a level room-by-room, with the assumption that a room could approximate a sector.

3.3.4 Text Representation

A text representation is also available, following the work of Summerville et al. in [60]. In particular the representation has been extended from one character per tile/pixel to two characters. This way it has been possible to add the information about the sector tag and the damaging

floor. This representation is not currently used by our work but provided for consistence with previous works. Table 3.6 reports all the character used for this encoding.

3.3.5 Scalar Features

Each level is annotated with 176 numerical and textual features which are divided in four categories:

1. **IDGames Archive Metadata** Contain information collected from the database when levels have been downloaded. This information contains the author, the descriptions, download urls, level title, etc. Since a WAD file can contain up to 32 levels, this information is replicated for each level found in the WAD file.
Listed in table 3.7

2. **WAD-extracted features:** This features are low-level features collected directly when processing the WAD file and include the number of lines, things, sectors, vertices, the maximum and minimum coordinates, the level size in MU etc.
Listed in table 3.8

3. **PNG-extracted features** These features are computed starting from the FloorMap using an Image processing library for calculating morphological properties. Each feature is calculated both directly over the whole level and as simple statistics computed over its "floors" taken singularly. A "Floor" is intended as a part of level which is not connected to the rest of the level, thus is reachable only by means of a teleporter.
Listed in tables 3.9, 3.10

4. **Graph Features** Features computed on the room graph, inspired by the work of Luperto and Amigoni in [44]. They are used to provide a higher level representation of the level and an indicative distribution of the different room types.
Listed in table 3.11

Value	Functional Category	Thing Description
0		Empty
1	other	Boss Brain
2	other	Deathmatch start
3	other	Player 1 start
4	other	Player 2 start
5	other	Player 3 start
6	other	Player 4 start
7	other	Spawn shooter
8	other	Spawn spot
9	other	Teleport landing
10	keys	Blue keycard
11	keys	Blue skull key
12	keys	Red keycard
13	keys	Red skull key
14	keys	Yellow keycard
15	keys	Yellow skull key
16	decorations	Bloody mess
17	decorations	Bloody mess
18	decorations	Candle
19	decorations	Dead cacodemon
20	decorations	Dead demon
21	decorations	Dead former human
22	decorations	Dead former sergeant
23	decorations	Dead imp
24	decorations	Dead lost soul (invisible)
25	decorations	Dead player
26	decorations	Hanging leg
27	decorations	Hanging pair of legs
28	decorations	Hanging victim, arms out
29	decorations	Hanging victim, one-legged
30	decorations	Hanging victim, twitching
31	decorations	Pool of blood
32	decorations	Pool of blood
33	decorations	Pool of blood and flesh
34	decorations	Pool of brains

Table 3.2: ThingsMap Encoding (1 of 3)

Value	Functional Category	Thing Description
35	obstacles	Barrel
36	obstacles	Burning barrel
37	obstacles	Burnt tree
38	obstacles	Candelabra
39	obstacles	Evil eye
40	obstacles	Five skulls "shish kebab"
41	obstacles	Floating skull
42	obstacles	Floor lamp
43	obstacles	Hanging leg
44	obstacles	Hanging pair of legs
45	obstacles	Hanging torso, brain removed
46	obstacles	Hanging torso, looking down
47	obstacles	Hanging torso, looking up
48	obstacles	Hanging torso, open skull
49	obstacles	Hanging victim, arms out
50	obstacles	Hanging victim, guts and brain removed
51	obstacles	Hanging victim, guts removed
52	obstacles	Hanging victim, one-legged
53	obstacles	Hanging victim, twitching
54	obstacles	Impaled human
55	obstacles	Large brown tree
56	obstacles	Pile of skulls and candles
57	obstacles	Short blue firestick
58	obstacles	Short green firestick
59	obstacles	Short green pillar
60	obstacles	Short green pillar with beating heart
61	obstacles	Short red firestick
62	obstacles	Short red pillar
63	obstacles	Short red pillar with skull
64	obstacles	Short techno floor lamp
65	obstacles	Skull on a pole
66	obstacles	Stalagmite
67	obstacles	Tall blue firestick
68	obstacles	Tall green firestick
69	obstacles	Tall green pillar
70	obstacles	Tall red firestick
71	obstacles	Tall red pillar
72	obstacles	Tall techno floor lamp
73	obstacles	Tall techno pillar
74	obstacles	Twitching impaled human

Table 3.3: ThingsMap Encoding (2 of 3)

Value	Functional Category	Thing Description
75	monsters	Arachnotron
76	monsters	Arch-Vile
77	monsters	Baron of Hell
78	monsters	Cacodemon
79	monsters	Chaingunner
80	monsters	Commander Keen
81	monsters	Cyberdemon
82	monsters	Demon
83	monsters	Former Human Trooper
84	monsters	Former Human Sergeant
85	monsters	Hell Knight
86	monsters	Imp
87	monsters	Lost Soul
88	monsters	Mancubus
89	monsters	Pain Elemental
90	monsters	Revenant
91	monsters	Spectre
92	monsters	Spider Mastermind
93	monsters	Wolfenstein SS
94	ammunitions	Ammo clip
95	ammunitions	Box of ammo
96	ammunitions	Box of rockets
97	ammunitions	Box of shells
98	ammunitions	Cell charge
99	ammunitions	Cell charge pack
100	ammunitions	Rocket
101	ammunitions	Shotgun shells
102	weapons	BFG 9000
103	weapons	Chaingun
104	weapons	Chainsaw
105	weapons	Plasma rifle
106	weapons	Rocket launcher
107	weapons	Shotgun
108	weapons	Super shotgun
109	powerups	Backpack
110	powerups	Blue armor
111	powerups	Green armor
112	powerups	Medikit
113	powerups	Radiation suit
114	powerups	Stimpack
115	artifacts	Berserk
116	artifacts	Computer map
117	artifacts	Health potion
118	artifacts	Invisibility
119	artifacts	Invulnerability
120	artifacts	Light amplification visor
121	artifacts	Megasphere
122	artifacts	Soul sphere
123	artifacts	Spiritual armor

Table 3.4: ThingsMap Encoding (3 of 3)

Value	Functional Category	Thing Description
0	None	Empty
10	local doors	Blue key local door
12	local doors	Red key local door
14	local doors	Yellow key local door
16	local doors	Local door
32+i	remote doors	Remote door with tag i
64+i	lifts	Lift with tag i
128+i	switch	Linedef that activates the i tag
192+i	teleports	teleport to sector i
255	exit	Level Exit

Table 3.5: TriggerMap Encoding: Each item i is connected to one or more objects. For example: switch (128+1) will open the door (32+1), raise the lift (64+1), etc.

1st character	Description	2nd Character	Description
"-"	["empty","out of bounds"]	"-" [ascii(45)]	Empty, no tag
"X"	["solid","wall"]	"." [ascii(46)]	Tag 1
"."	["floor","walkable"]	"/" [ascii(47)]	Tag 2
","	["floor","walkable","stairs"]	"0" [ascii(48)]	Tag 3
"E"	["enemy","walkable"]	...	...
"W"	["weapon","walkable"]	"m" [ascii(109)]	Tag 64
"A"	["ammo","walkable"]	"~" [ascii(126)]	Damaging floor
"H"	["health","armor","walkable"]		
"B"	["explosive barrel","walkable"]		
"K"	["key","walkable"]		
"<"	["start","walkable"]		
"T"	["teleport","walkable","destination"]		
":"	["decorative","walkable"]		
"L"	["door","locked"]		
"t"	["teleport","source",activatable"]		
"+"	["door","walkable","activatable"]		
">"	["exit","activatable"]		

Table 3.6: Extended Textual Representation Encoding: A second character has been added to the one used by "The VGLC: The Video Game Level Corpus": Each tile is expressed by two characters "XY" where X is the type of object and Y is the tag of the tile. Every tile that has a tag number, activates (or is activated by) the object(s) with the same tag number. So, e.g. "t/" is a teleport that leads to "T/" and "X." is a switch that activates the door "+." and possibly a floor ".."

Feature Name	Description	Type
author	Level Author	string
description	Natural language level information	string
credits	Natural language level information	string
base	Natural language level information	string
editor_used	Natural language level information	string
bugs	Natural language level information	string
build_time	Natural language level information	string
rating_value	doomworld.com level rating value	float
rating_count	doomworld.com vote count	int
page_visits	doomworld.com page visits	int
downloads	doomworld.com download count	int
creation_date	Natural language level information	string
file_url	Download page url	string
game	Doom or DoomII	string
category	doomworld.com category (eg. a-z)	string
title	Full level name	string
name	level .zip filename	string
path	relative path to wad file	string

Table 3.7: Features: IDArchive Metadata

Feature Name	**Description**	**Type**
number_of_lines	absolute number of lines in the level	int
number_of_things	absolute number of objects in the level	int
number_of_sectors	absolute number of sectors (zones with same height) in the level	int
number_of_subsectors	absolute number of subsector (convex subshapes of sectors) in the level	int
number_of_vertices	absolute number of vertices in the level	int
x_max	maximum x coordinate	int
y_max	maximum y coordinate	int
x_min	minimum x coordinate	int
y_min	minimum y coordinate	int
height	level original height in DoomUnits	int
width	level original width in DoomUnits	int
floor_height_[max|min|avg]	[max|min|avg] height for the floor	float
ceiling_height_[max|min|avg]	[max|min|avg] height for the ceiling	float
room_height_[max|min|avg]	[max|min|avg] difference between ceiling and floor height	float
sector_area_[max|min|avg]	[max|min|avg] area of sectors in squadred doom map units	float
lines_per_sector[max|min|avg]	[max|min|avg] count of sector sides	float
aspect_ratio	Ratio between the longest and the shortest dimension, since a rotation of 90 of the level does not alter playability	float
walkable_area	Number of pixels the player can walk on (nonempty_size - walls)	int
walkable_percentage	Percentage of the level that is walkable	float
number_of_<things_type >	Total number of <things_type>in the level. <things_type>: { artifacts, powerups, weapons, ammunitions, keys,monsters , obstacles, decorations }	int
<things_type>_per_walkable_area	Number of <things_type>divided the walkable area in DMU.	float
start_location_[x|y]_px	[x|y] coordinate (in pixels, dataset format) of the start location.	int
slot	Name of the map slot. e.g "E1M1" or "MAP01"	string

Table 3.8: WAD-extracted features

Feature Name	**Description**	**Type**
floors	Number of non-connected sectors (only reachable by a teleport) of the level	int
level_area	Number of pixels composing the level	int
floors_area_[mean|min|max|std]	[mean|min|max|std] number of pixels composing each floor	float
level_bbox_area	Number of pixels of bounding box sorrounding the level	int
level_convex_area	Number of pixels of convex hull for the whole level	int
floors_convex_area_[mean|min|max|std]	[mean|min|max|std] Number of pixels of convex hull for each floor	float |int
level_eccentricity	Eccentricity of the ellipse that has the same second-moments as the level.	float
floors_eccentricity_[mean|min|max|std]	Eccentricity of the ellipse that has the same second-moments as each floor	float
level_equivalent_diameter	The diameter of a circle with the same area as the level	float
floors_equivalent_diameter_[mean|min|max|std]	[mean|min|max|std] equivalent diameter calculated over the floors of the level	float
level_euler_number	Euler characteristic of the level. Computed as number of objects (= 1) subtracted by number of holes (8-connectivity).	int
floors_euler_number_[mean|min|max|std]	[mean|min|max|std] euler number over the floors of this level	float

Table 3.9: PNG-extracted features (1 of 2)

Feature Name	Description	Type
level_extent	Ratio of pixels in the level to pixels in the total bounding box. Non-empty size of the level.	float
floors_extent_[mean|min|max|std]	[mean|min|max|std] extent over the floors of the level	float
level_filled_area	Number of pixels of the level, obtained by filling the holes	int
floors_filled_[mean|min|max|std]_mean	[mean|min|max|std] filled area over the floors of the level	float
level_major_axis_length	The length of the major axis of the ellipse that has the same normalized second central moments as the level.	float
floors_major_axis_length_[mean|min|max|std]	[mean|min|max|std] major axis length over the floors of the level	float
level_minor_axis_length	The length of the minor axis of the ellipse that has the same normalized second central moments as the level	float
floors_minor_axis_length_[mean|min|max|std]	[mean|min|max|std] minor axis length over the floors of the level	float
level_orientation	Angle between the X-axis and the major axis of the ellipse that has the same second-moments as the level. Ranging from -pi/2 to pi/2 in counter-clockwise direction.	float
floors_orientation_[mean|min|max|std]	[mean|min|max|std] orientation over the floors of the level	float
level_perimeter	Perimeter the level which approximates the contour as a line through the centers of border pixels using a 4-connectivity.	float
floors_perimeter_[mean|min|max|std]	[mean|min|max|std] perimeter over the floors of the level	float
level_solidity	Ratio of pixels in the level to pixels of the convex hull image.	float
floors_solidity_[mean|min|max|std]	[mean|min|max|std] solidity over the floors of the level	float
level_hu_moment_[0 ... 6]	Hu moments (translation, scale and rotation invariant).	float
level_centroid_x	Centroid coordinate x	float
level_centroid_y	Centroid coordinate y	float

Table 3.10: PNG-extracted features (2 of 2)

Feature Name	Description	Type
art-points	Number of articulation points in the room adjacency graph. An articulation point is a node which removal would result in a bipartite graph	int
assortativity-mean	Mean assortativity. Assortativity is the tendency of one node to be connected with similar nodes.	float
betw-cen-[min |max |mean |var]	Node centrality statistic calculated with the betweenness method	float
betw-cen-[skew |kurt]	Node centrality statistic calculated with the betweenness method	float
betw-cen-[Q1 |Q2 |Q3]	Node centrality statistic calculated with the betweenness method	float
closn-cen-[min |max |mean |var]	Node centrality statistic calculated with the Closeness method	float
closn-cen-[skew |kurt]	Node centrality statistic calculated with the Closeness method	float
closn-cen-[Q1 |Q2 |Q3]	Node centrality statistic calculated with the Closeness method	float
distmap-[min |max |mean]	Maximum value in the distance map, i.e. the size of the largest room. Background is ignored from computation.	float
distmap-[var |skew |kurt]	Mean value for the distance map, i.e. the mean room size. Background is ignored from computation.	float
distmap-[Q1 |Q2 |Q3]	Skewness of the distnace distribuiton Background is ignored from computation.	float

Table 3.11: Graph features

3.4 Dataset Organization

This section describes how the dataset is stored and how it can be used for future works.
Since the dataset has been created primarily for instructing a neural network to generate new levels, the choices in data formats and data representation have been made to increase reusability and flexibility in data manipulation. In particular the full dataset is stored first as a set of files indexed by a JSON dictionary, but for various reason that will be explained more in depth in chapter 5.1.1 in our work we only used a subset of levels stored as a separated archive.

3.4.1 Full Dataset and Filtered Dataset

In order to keep as much information as possible from the collected levels, we structured the representation of the DOOMDataset as follow:

- A **Full Dataset** containing all the levels we collected from the Idgames Archive.
- A set of **Filtered Datasets** containing a subset of levels that satisfy certain constraints and which are ready to use with TensorFlow [1].

Full Dataset

The full dataset is kept as much portable as possible, in the sense that it shouldn't need particular technologies to be accessed except the capability of parsing JSON files and NetworkX [29] for analysing the graph structure.
It is composed of about 9172 levels organized as a directory structure, while the maps are of different sizes given by the rescaling of the levels in MU to tile/pixel format.
The folder is structured as follow:

- **dataset.json** Contains all the scalar and textual information explained in section 3.3.5 and the relative paths to the files in sub-directories. Acts as a level database.
- **Original:** Contains all the WAD files as extracted by the archives downloaded from the IDGames Archive.
- **Processed:** Contains the Feature Maps, the graph representation, the text representation and the relative set of features.

Data in this folder is named as:

zipname_WADNAME_SLOT.json Is a json file containing all the features (3.3.5) for the level.

zipname_WADNAME_SLOT.networkx Is a gpickle compressed file containing the NetworkX graph for the level.

zipname_WADNAME_SLOT.txt Is the Text Representation (3.3.4) of the level

zipname_WADNAME_SLOT_mapname.txt Is the set of Feature Maps in PNG 8-bit greyscale format.

Where zipname, wadname, mapname indicate the name of the zip archive the WAD was stored in, the wad file itself and the Feature Map respectively, while SLOT indicates the level slot name in DOOM format (see section 3.2.2, "NAME" lump). The choice of keeping features both in the JSON database and in separated files comes from the need of recomputing the features in an easy way (for example for adding features) and for providing the possibility to manually pick or inspect level properties without the need of accessing a long json file. This, however, comes at the cost of some data redundancy.

Filtered Dataset

Due to technological limitations given by the machines used for training and other reasons explained in chapter 5.1.1, data is filtered according to some criteria in order to make them uniform in term of Feature Map size and removing some level exposing extreme values of the features. Filtered dataset is stored as a set of files described as follor:

• **DATASETNAME-train.TFRecord** Dataset used for training the network using the TFRecord data format, which is a binary data format proposed by TensorFlow for improving data ingestion performances.

• **DATASETNAME-validation.TFRecord** Dataset used for model validation stored in TFRecord data format.

• **DATASETNAME.meta** Metadata and statistics about the dataset contained in train and validation datasets, in JSON format. Since the TFRecord format currently does not natively hold any information about the data contained in its records, it is useful to save data such as the item count, data structure and statistics about the dataset in a separate file, for easing the process of data normalization and other kind of operations.

In our GitHub Repository [25] it is possible to find the files relative to two different data subset:

• **128-many-floors** Dataset of images up to 128x128 pixels (smaller levels are centered and padded) which have any number of floors, consisting of 1933 levels in training set and 829 in validation set.

• **128-one-floor** Dataset of images up to 128x128 pixels (smaller levels are centered and padded) which have just one floor, consisting of 1104 levels in training set and 474 in validation set.

The code repository [25] hosts a Python module that provides all the necessary methods to inspect and analyse and rebuild both the full dataset and the filtered dataset. Further information is provided in the code documentation as it goes beyond the scope of this chapter.

3.4.2 Level Size Statistics

In this section we analyse how the level dimensions in MU are distributed in the Full Dataset. Table 3.12 reports the percentile distributions for both the level height and the level width in Map units and Pixels. This distribution highlights how the choice of scaling 32 Doom units to a single pixel in order to avoid loosing functional information about the levels also leads to reasonable image sizes, since more than 80 percent of the levels are representable in with an image of 256 pixels in both height and width. Joint distribution of width and height is represented in figure 3.2

Percentile	Width in DU	Width in pixels	Height in DU	Height in pixels
10th	2305	72	2065	64
20th	3017	94	2657	83
30th	3633	113	3137	98
40th	4161	130	3606	112
50th	4737	148	4101	128
60th	5393	168	4657	145
70th	6177	193	5313	166
80th	7212	225	6209	194
90th	9040	282	7809	244
100th	31233	976	32743	1023

Table 3.12: Percentiles of level width and height distributions in both Map Units and Pixels

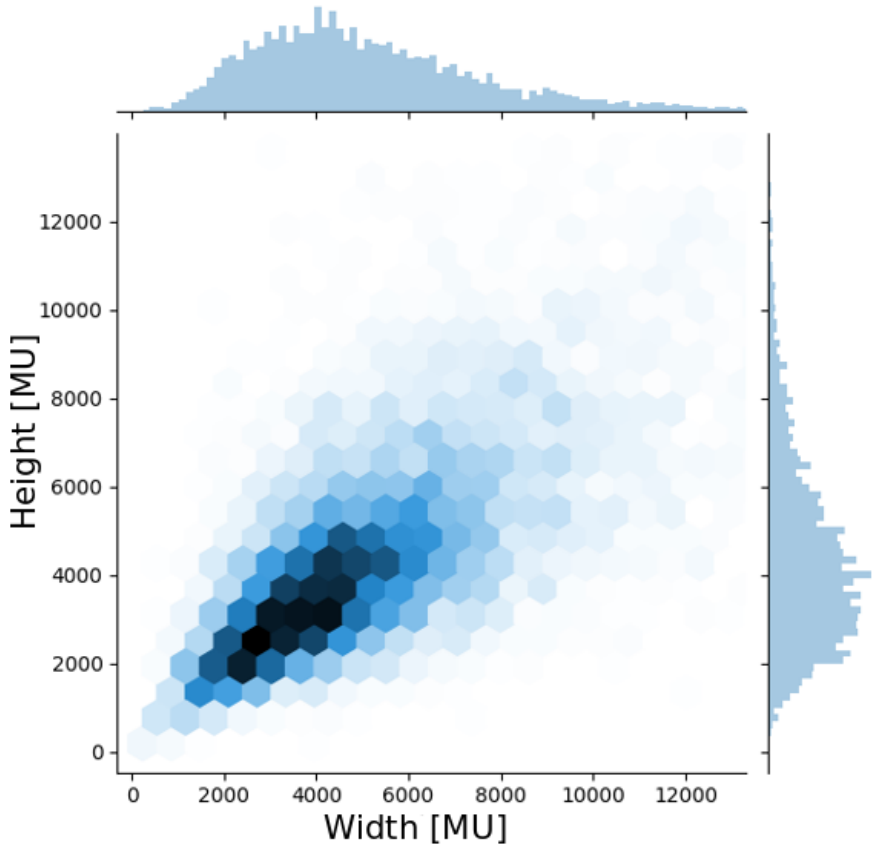

Figure 3.2: Joint distribution for the features "width" (on the x axis) and "height" (on the y axis) expressed in MU for the level contained in the full dataset. It is possible to notice how the size of the majority of the levels are below 7000 Map Units

3.5 Summary

In this chapter we proposed a setting for describing the structure and the features of DOOM levels, analysing the WAD file format and exploiting its properties to extract useful data. Inspired by the work of Summerville et al. in "The VGLC: The Video Game Level Corpus" we produced a dataset of more than 9000 DOOM levels for extending previous work and providing a ready-to-use database for future works on video-game levels. In doing so, we developed a new system for converting WAD files into a data format that is compatible with our needs, while also trying to preserve the textual format provided by the previous work for backward compatibility. In the next chapter we describe our system from an high level perspective, showing how the dataset is used to generate new levels.

Chapter 4

System Design and Overview

The purpose of this chapter is to give an overview of how the system modules interact from the point of view of use cases and data flow analysis. Latest sections of the chapters focuses on the neural network architecture we adopt in our system. Section 4.1 describes the logical structure of the generative model we use, focusing on the input and outputs and providing the notation we use in the remaining part of this work. Section 4.2 illustrates all the main use cases for the system, highlighting how the different inputs are used to produce the expected results. Section 4.3 resumes the system design focusing on processes and data transformation rather then a component view of the system. Section 4.4 details the neural network architecture we used in our system, present a set of metrics we use during the training process and defines the training process itself. Section 4.5 describes the modules we designed to convert DOOM levels to and from images and extract all the necessary features.

4.1 Generative Model Structure

The initial system design was built upon the architecture of Generative Adversarial Networks [26] from Goodfellow et al. Given the problem of generating video-games levels, the need to control to some extent the generation process naturally arose. For this reason, we adopted a conditional version [46] of the GAN Model, proposed by Mirza and Osindero, applied to the WGAN-GP architecture already discussed in section 2.1.4.

We present in figure 4.1 the logical design which defines the inputs and the outputs of the generative model we are using. This structure refers to the Conditional Generative Neural Network we introduced in chapter 2.1.1. In particular, figure 4.1 shows the general interactions of the high-level components of a GAN, namely the Generator and the Discriminator *(or Critic)* networks
The input of this subsystem are defined as follows:

- X: Batch of images having m channels, corresponding to the Feature Maps.
- Y: Batch of vectors having one component for each scalar feature considered.
- Z: Batch of random noise vector, typically sampled from a Uniform or Gaussian distribution.

The discriminator network takes as input a vector of images X and a vector of features Y, while the generator network G takes as input a the vector Y and a vector of random noise Z, which is used to sample different points of the data distribution. We use the subscript "True" or "Gen" to distinguish from the Feature Maps coming from the input dataset and those that are generated by the generator G.
For what concerns the network outputs of our architecture, we have that:

- $X_{Gen} = G(Z|Y)$: Samples generated from the generator network

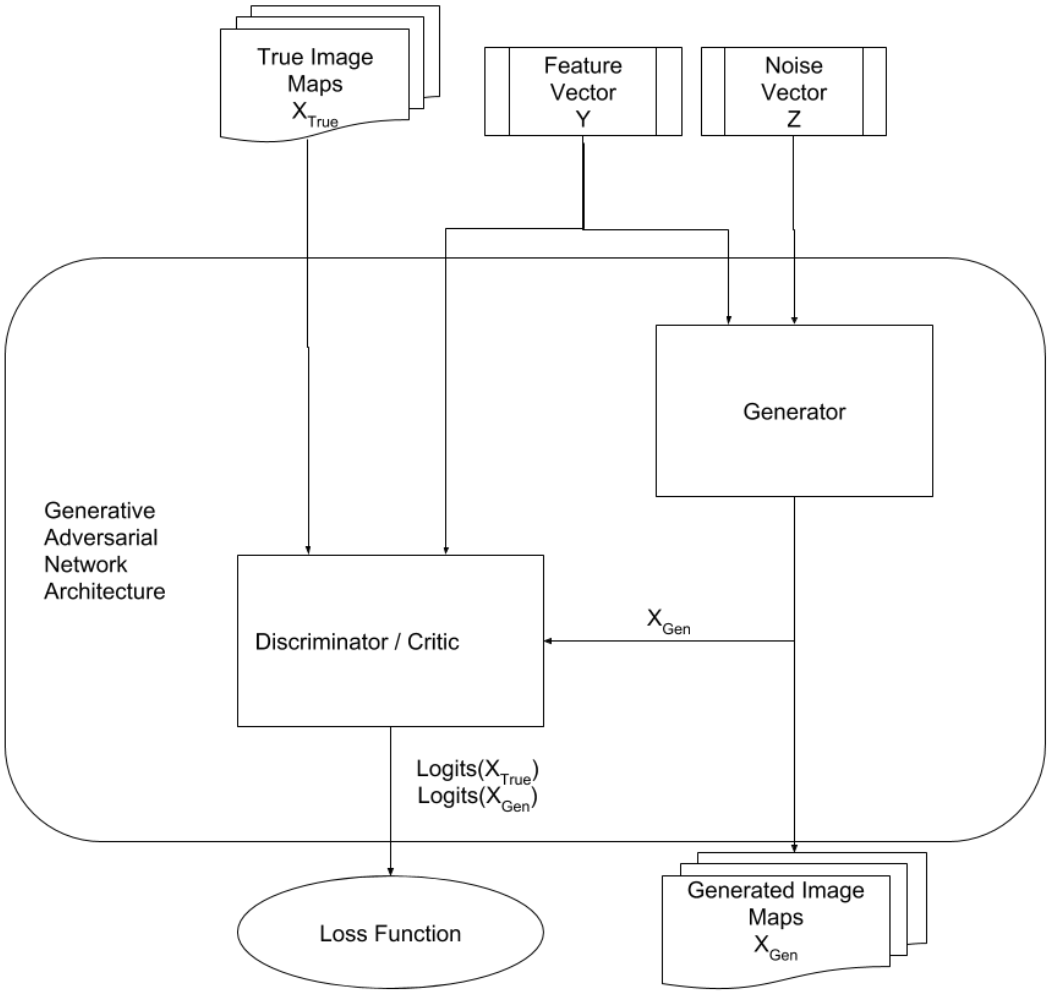

Figure 4.1: System Overview: Generative Model Structure. A Conditional GAN is composed of a generator model and a discriminative model. The discriminative model takes as input either the Images coming from the dataset or the ones generated by the generative model. Both networks are conditioned by the Y feature vector, while the generator also takes an input a noise vector Z to sample from the data distribution it is approximating.

- $Logits(X_{Gen}) = D(X_{Gen}|Y)$: Discriminator output when real samples are input to the discriminator.
- $Logits(X_{True}) = D(X_{True}|Y)$: Discriminator output with generated samples are provided to the discriminator.

Logits are actually the output of the last layer of the discriminator network before the last *activation function*, and they are related to the discriminator assessment of each sample. Loss functions for either the generator and the discriminator are written upon those values and alternately optimized to train the entire network. For simplifying the notation, we will implicitly refer to the output logits simply as D in the case of the Discriminator/Critic, while we refer to the output of the last activation function in the case of the Generator. All the remaining details are given when we'll describe the chosen GAN Architecture and the training process in section 4.4.1.

4.2 Use Cases

This section will describe the main use cases of our system, which are necessary for replicating our results. The emphasis is put on how inputs and outputs are used in each case, while the internal structure of the generative model is not represented in order to simplify the notation. Every figure in this section also describes the function of the dataset metadata introduced in section 3.4.1 while describing the filtered dataset. In particular the neural network inputs and outputs are limited in a certain range, typically between 0 and 1 or -1 and 1. For this reason, dataset statistics are needed to properly rescale input and output data. Blocks which are written in bold type indicate those inputs and outputs that are of interest for the corresponding use case.

4.2.1 Use Case: Model Optimization (Training)

This use case describes the optimization of the model, which is commonly referred as the training phase. This is the phase in which data from the dataset is fed into the model and the loss functions are minimized in order to find the network weights that allow the generation of samples of good quality. In the ideal case the generated samples should come from a distribution which is undistinguishable from the true data distribution. In reality, this is limited by the balance of the discriminator ability in selecting features that allow it to distinguish true data from artificial one, and the generator ability in misleading the discriminator in its task by generating samples that are similar to the real ones. Figure4.2 shows that in this use case both the Scalar Features Y and the Feature Maps X from the dataset are used to train the network. At each epoch the generator network is fed with a noise vector Z along with the conditioning vector Y. This procedure produces a training loss that is provided to an optimizer that acts on the network weights.

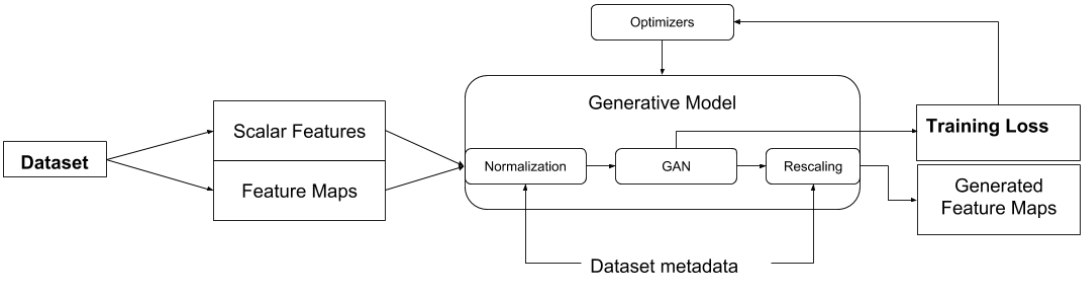

Figure 4.2: Use Case: Model Optimization.

4.2.2 Use Case: Sample Evaluation

This use case is useful to assess the ability of the model to generate new samples on previously unseen feature vectors and monitoring the training process. This is accomplished by running two different procedures in which only the validation set, composed of samples that are left out from the training phase, is used:

1. A *validation loss* is calculated by feeding the discriminator with images $X_{True,Val}$ coming from the validation set and their corresponding feature vector Y_{Val}. This approach is often used classical (i.e. discriminative) neural networks, where it is a good method for detecting over-fitting and assessing network generalization capabilities. In our setting, however, this may not always be a meaningful metric with every proposed underlying architecture. This is usually due to the fact that with many GAN architectures the loss does not correlate well with the quality sample. However, in section 4.4.1 we select one of the architectures which propose to reduce the severity of this problem, among others.

2. A set of *quality metrics* (4.4.2) are computed directly on the true samples $X_{True,Val}$ and the samples $X_{Gen,Val} = G(Z|Y_{Val})$ generated by conditioning the network with the same features of $X_{True,Val}$. This is based on the assumption that if the network actually learns a correlation between the Y vector of features and a certain set of features proper to the corresponding X samples, then the true sample and the generated one might show a certain level of similarity according to the given features. Since this may be a strong assumption, especially in the setting where we are not able to measure what features are actually mapped to each component Y_i due to the high dimensionality of the problem, a set metrics is chosen and presented in section (4.4.2). Selected metrics should be general enough to express, when averaged on batches of samples, a concept of "sample quality" without directly referring to the features encoded in the Y vector.

Figure 4.3 shows how the scalar features from the validation set are used to generate new samples, that are compared to the corresponding true images.

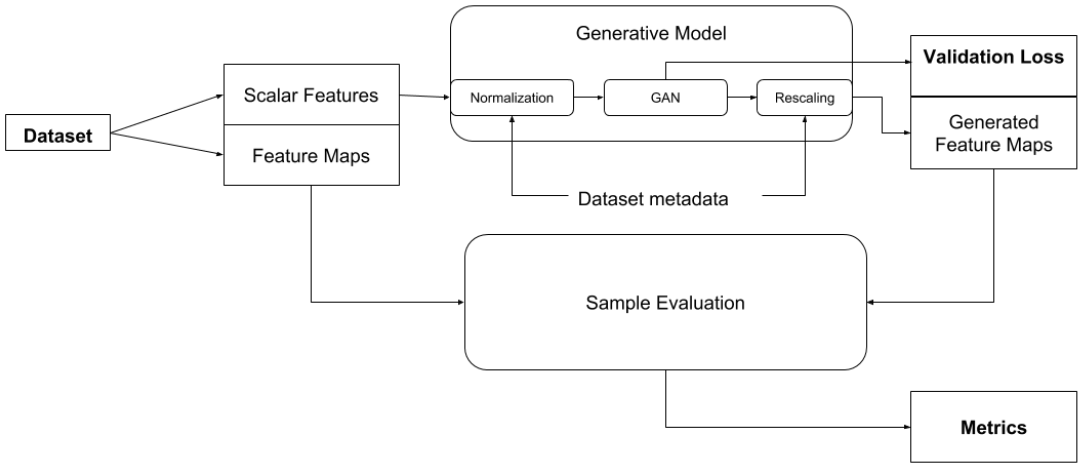

Figure 4.3: Use Case: Validation and Sample Evaluation.

4.2.3 Use Case: Sampling or Generation

This use case is the one that produces new levels and it is run after a model has been trained. In particular, our system supports several methods for sampling the network in order to generate new levels.
The main problem of sampling this network, as highlighted in the work of White[68], is choosing a feature vector that has lies in an area that has enough prior probability. We here present four possible methods for sampling our network, while more details about possible improvements are given in section 7.2.1. These methods differ, other from the sampling method, on the input that is requested from the user. The noise vector Z can be either random generated for each sampling or kept the same for testing how the Y vector impacts on the network generation.

- **Dataset Sampling**: This sampling method does not require any input from the user, since the conditioning vectors Y are sampled from the dataset.
- **"Factors" Sampling**: This method allows the user to specify a set of scalars $y_{feat} = [y_1, ..., y_f], y_i \in [0, 1]$ where f is the number of features. The extrema correspond to particular values of the related feature, based on the dataset metadata. For example they can match $E[Y_i] \pm Std(Y_i)$ such that each Y_i is sampled from a region of the feature space in which it is more likely to have significant probability. This, however, may be not enough because even if the feature components Y_i exists in the dataset distribution when taken singularly, this may not hold for their joint distribution. Moreover, the presence of the Z vector greatly increase

the dimensionality of the problem. That said, this method can still be useful to easily specify small perturbation in one or more features to inspect the network response, but arbitrary feature sampling remains difficult.

- **Direct Sampling**: This kind of sampling requires the user to directly provide a Y vector as it would come from the dataset. It can be used as a starting point to develop more complex sampling methods.
- **"Content" Sampling**: This kind of sampling is the one that is more interesting from the perspective of an end user. If we consider a design tool that could use this network, we would need to provide the user an interface that is the most natural as possible. Rather than inspecting and "guessing" numerical values, a level designer may be interested in sketching a level and possibly obtaining a set of samples whose features reflect the ones of the provided sketch. We thus propose a sampling method that extracts the feature vector directly from a user generated image, in the same way it is extracted from the Feature Maps coming from the Dataset.

Figure 4.4 shows the main differences of the proposed methods in terms of logical transformations and required inputs.

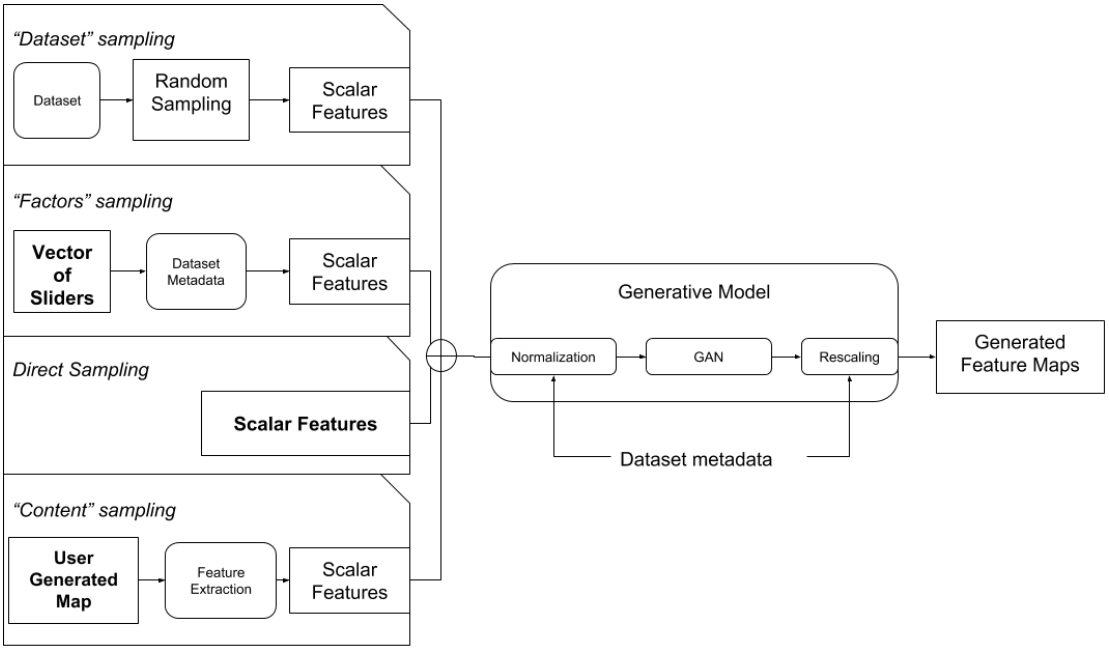

Figure 4.4: Use Case: Sampling or Generation.

4.3 Data Flow

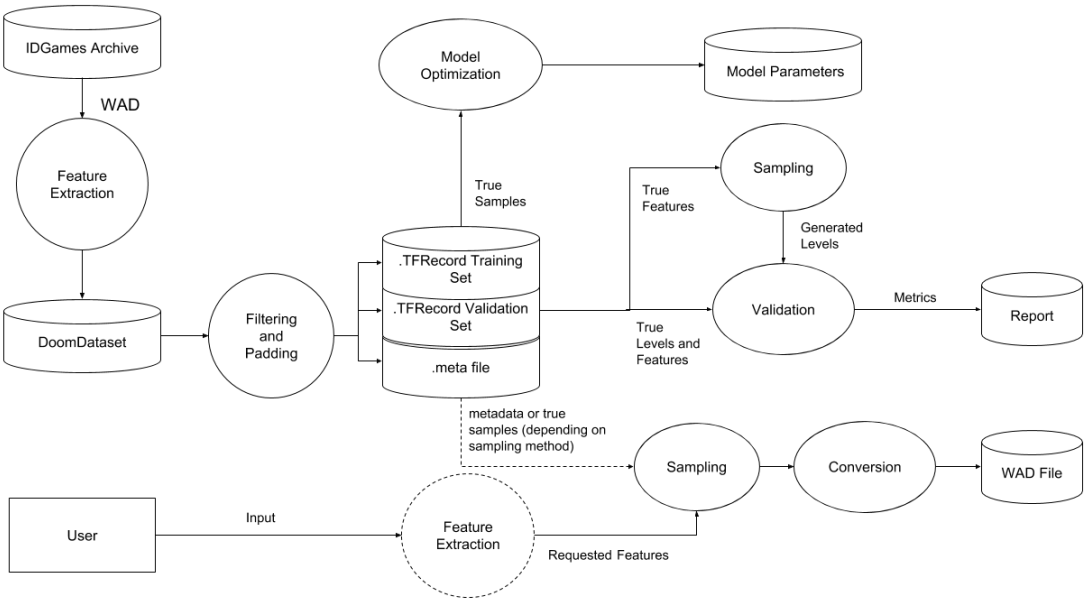

Figure 4.5: System Overview: Data Flow Diagram. Disc-shaped blocks represent data archives, circles represent processes and transformations, while the labels on arrows represents intermediate data. Dashed lines represents objects which presence depends on the use case of reference.

Figure 4.5 represents the conjunction of the use cases explained in section 4.2, from a data flow perspective. The figure is organized as a set of transformations developing from left to right. In particular, blocks on the left represent the inputs to the process and blocks on the right boundary are the produced artefacts. It is possible to identify three main data paths: The first produces the model parameters and it is identified with the use case "Model Optimization", or "Training" (4.2.1).
The second one corresponds to the Sample Evaluation use case (4.2.2) in which levels are generated with a feature vector and then compared with the true ones corresponding to the same feature vector in the dataset. The generated metrics values are stored in a report and showed during the training phase. The third path, which produces WAD Files, corresponds to the Sampling Use Case (4.2.3). In this case the data path elements depend on the sampling method used, for this reason they are indicated with dashed lines.

4.4 Neural Network Architecture and Training Algorithm

Two main issues with GAN models are that the loss function is not always correlated with the generated sample quality, and the sample quality evaluation is still an open field of research. Although we tried some different GANs implementations and techniques for both problems, we focused on the stability problem when selecting the final network architecture, while relying on a set of metrics detailed in section 4.4.2 to qualitatively assess the samples.

4.4.1 Network Architecture

Chosen Architecture

Among the various GANs implementations that are proposed in the literature, we selected the Wasserstein GAN with Gradient Penalty [28] (WGAN-GP) described in section 2.1.4 as it showed better training stability with the DCGAN layer configuration and at least comparable sample quality as opposed to the other models. We considered both the unconditional and conditional versions, by parametrizing the system upon the selected input features. The only difference we introduced to the proposed model is the adoption of the *sigmoid* activation function on the generator output layer in place of *tanh*. This choice is motivated by the fact that tanh have been originally selected for obtaining a better colour coverage in generated rgb images, while we are interested in discrete values that often correspond to the lowest and highest output values. This showed to help the network learning faster the representation of FloorMaps and WallMaps, while not affecting the other maps.

Loss Formulation

We implement the Critic and Generator losses as in the "Improved Training of Wasserstein GANs" official implementation [27], which combines formulas 2.2 and 2.3. Referencing to the notation introduced in 4.1 the losses are defined as follow:

$$\begin{aligned} L^{(i)}_{Critic} &\leftarrow \underbrace{\mathrm{E}(D(X_{Gen})) - \mathrm{E}(D(X_{True}))}_{\text{WGAN Loss}} + \underbrace{\lambda G_p}_{\text{Gradient Penalty}} \\ L^{(i)}_{Gen} &\leftarrow -D(X_{Gen}) \end{aligned} \tag{4.1}$$

where X_{True} and X_{Gen} are batches of levels sampled respectively from the dataset and the generator network, $\lambda = 10$ and $\epsilon \sim U[0, 1]$ (for the gradient penalty). We recall that in this case, D corresponds to the *logits* of the Critic, since the output is considered before the last activation function.

We can now give an intuitive interpretation of the loss we used: The critic network is trained, by minimizing 4.1, to assign an unbounded "score" to real and generated samples. This is one of the reasons that motivate the change in name from "discriminator" to "critic" [1], since the network outputs are in this case not probabilities but unbound values.

Training Algorithm and Hyper-Parameters

For implementing the training algorithm we followed the algorithm suggested by [28, alg. 1. p. 4], which uses *Adam*[39] as the optimizer for both the networks and optimizes $n_{critic} = 5$ times the critic network for each generator update.
In addition to this implementation, we imposed an input rotation of 90 for each sample at each epoch, such that every 4 iterations the network had in input all the possible orientation of a level. This allows us to exploit the rotation invariance in the representation of a level, since its

[1] Discussion on Wasserstein GAN paper with the authors of WGAN and GAN papers, among the others: `https://www.reddit.com/r/MachineLearning/comments/5qxoaz/r_170107875_wasserstein_gan/`

playability is not affected by its orientation in the space it is represented. Table 4.1 shows in detail the periodicity of each computation operation relative to each critic step, with reference to our TensorFlow implementation at [25]. Due to implementation constraints, we calculate the Metrics out of the TensorBoard computation graph, so they appear as inputs since they have to be visualized with the other values. Reference Sample refers to the computation of a sample that is generated by the same Y and Z vectors, sampled at the beginning of the training phase. This helps in understanding how the network weights optimization visually impacts on the generation of the same batch.
The other hyperparameters we used in the training phase are $\alpha = 0.0002, \beta_1 = 0, \beta_2, \lambda = 10$.

Run Name	Inputs			Outputs	Periodicity	Evaluated operators
	Critic Input (X_{True})	Scalar Features (Y)	Generator noise (Z)			
Train G	-	Y_{Train}	$U[0,1]$	L_{Gen}	5	G_{optim}, $summary_D$
Train D	X_{Train}	Y_{Train}	$U[0,1]$	L_{Critic}	1	G_{optim}, $summary_D$
Validation	X_{Val}	Y_{Val}	$U[0,1]$	$L_{Gen,Val}$, $L_{Critic,Val}$, X_{Gen}	100	G_{optim}, $summary_D$
Metrics	$Metrics(X_{Val}, X_{Gen})$			-	100	G_{optim}, $summary_D$
Reference Sample	–	Y_{Ref}	Z_{Ref}	X_{Ref}	100	G_{optim}, $summary_D$
Network Checkpoint	–	–	–	checkpoint	100	$save()$

Table 4.1: Training Algorithm Operations

Artefacts Reduction

One of the problems that often affects GAN is the presence of artefacts or regular patterns in the generated samples. This problem can be less noticeable when the network is generating images of real objects such in the majority of other works, but it's an important issue in our setting in which a change in a pixel can have an important impact on the resulting level and its associated metrics. This problem is well explained in [52] and it's due to how the transposed convolution is applied in case the kernel size is not divisible by the stride. In this case the convolution leads to an uneven overlap of outputs in the high-resolution image, thus generating the artefacts. Among the proposed solutions to overcome this issue we chose to simply use a kernel size of 4 and a stride of 2, which showed to reduce the problem in our case without impacting on the throughput or adding new types of layers.

4.4.2 Sample Evaluation metrics

The problem of evaluating the quality of the samples generated from a neural network remains an open area of research [54]. The architectures that we considered are primarily trained on datasets consisting of images representing real objects, such as faces or bedrooms. For dealing with this problem, Salimans et al. propose both a process in which human annotators are asked to assess the perceived quality of the samples [54, p. 4] and the usage of the Inception Model [61] Score for assessing the perceived quality of the samples. Although many authors had success in assessing sample quality using the Inception Score, this method showed to perform poorly on our dataset. The most probable reason is that our dataset is very different from the ImageNet

dataset upon which is trained the Inception Network.
Since tuning the Inception network to work with our dataset or applying other proposed solutions based on similarly complex models wouldn't have been possible given our computational constraints, we decided to design some heuristics that could correlate with sample quality at least with our dataset. These sample evaluation metrics give an additional way for assessing the generated sample quality which is independent from the level features, and can be computed during the training process with a minimal slowdown of the training process. The final models are then studied considering the level features by confronting the true and generated feature distributions (Section 5.1.3).

Instead of asking a neural network to evaluate the generated samples, our process compares true and generated samples that correspond to the same conditioning vector. Since in principle, given a vector of scalar features Y, the network could generate samples which are topologically and visually very different, any metric that used quantities that are strictly related to the level topology rather than a perceived concept of "quality" couldn't lead to reasonable results in this phase. In designing these metrics we inspired to the paper "2D SLAM Quality Evaluation Methods": In their work, Filatov et al. propose their solution to the problem of evaluating the maps generated by a SLAM algorithm running on a mobile agent. Their approach consist in defining three metrics that capture different aspects of the analysed maps, which shows some similarities with the Feature Maps we use, and considering the entire set of metrics as an approximation of the human perceived quality of a map.
Following a similar approach, we defined the following metrics for estimating the perceived quality of a sample generated by our network. These metrics are not meant to be a general solution to the problem of evaluating samples of a GAN nor to improve the work of **slam˙metrics**, but only to be used as a qualitative tool for assessing the samples in our particular case.

Entropy Mean Absolute Error

This metric is defined as the Mean Absolute error between the entropy of two images in their colour space. In particular, since as described in chapter 3.4 we are representing maps as greyscale images whose colour ranges between 0 and 255, we calculate the pixel distribution over each possible colour value c of an image x, $P_c(x)$, then we calculate the entropy as:

$$S(x) = -\sum_{c=0}^{255} P_c(x) * \log P_c(x) \tag{4.2}$$

then, the metric over a batch of true images X_{true} and a batch of generated images X_{gen}, both consisting of N samples, is calculated as:

$$Entropy_{mae}(X_{true}, X_{gen}) = \frac{1}{N} \sum_{i=0}^{N-1} |S(X_{gen}(i)) - S(X_{true}(i))| \tag{4.3}$$

This metric is related to how different the entropy of a generated image is from the corresponding real image, which can also be interpreted as the difference in the quantity of information expressed by the two samples. In general, large values of this metric indicates that the generated sample is close to random noise or the topology of the two levels are greatly different, while small values indicates that the entropies of the two images are on a comparable level.

Mean Structural Similarity Index

This metric is defined as the Structural Similarity (SSIM) Index [67] between two images. This measure is the result of a framework that consider several aspects of an image, such as the luminance, the contrast and the structure, rather than basing only on a single statistic. Moreover, the structural similarity technique is applied locally over the image, for reflecting the fact that

pixels are more correlated to close pixels than distant ones.
For calculating this metric we use the implementation provided by the Scikit-Image Python library, which we leave the formulation to the paper [67, p. 604], and compute the mean of the SSIM index over the images belonging to the true and generated batches. Regarding the interpretation of the metrics, higher values indicate the fact that true and generated samples are often structurally similar: in other words, the network produces samples in which the local structure of the pixels are comparable.

Mean Encoding Error

We define as "Encoding Error" a measure of how far the pixels colour of a generated image are from their closest meaningful value. Specifically, as we introduced the encoding values for each feature map in chapter 3.4 the reader might have noticed that not all values correspond to an actual representation. For example, the FloorMap encodes the pavement as 255 and the empty space as 0, leaving values from 1 to 254 without a real meaning. Since as highlighted in section 4.2 the network only reads and outputs floating point numbers between 0 and 1, this is not actually a problem of choosing an encoding space for the images, rather it is intrinsic to the network definition. Due to the generation process involving noise and non-integer parameters, the network will often output pixel colours that are in-between the possible values the input images can take, even though this behaviour decreases as the training proceeds.
The definition of this function, for a generic pixel colour value x which assumes meaningful values every i colour values, corresponds to a periodic triangular function having base i which assumes the maximum value of 1 wherever x is halfway a meaningful value and another. More formally:

$$Enc_I(x) \equiv \sum_{k \in Z} \Lambda\left(\frac{2(x - kI) - I}{I}\right) \tag{4.4}$$

where $\Lambda(x)$ is the unit-base triangular function such that $\Lambda(0) = 1$.
The metric is then calculated as

$$MEE(X_{gen}) = \frac{1}{N} \sum_{i=0}^{N-1} \frac{1}{|P|} \sum_{p \in P} Enc_I(p) \tag{4.5}$$

where $p \in P$ is the colour of each pixel of the image, $|P|$ is the total number of pixels in an image and N is the batch size of X. In particular, I is 255 for the FloorMap and WallMap while is 1 for the other maps.
Since, by construction, $MEE(X_{true}) = 0$ up to conversion errors or artefacts, this metric is only calculated for the generated samples and it is correlated with the ability of the network to represent precisely the data encoding.

Mean Corner Error

This error is based on the idea introduced with the corner count metric introduced by [21]. We define our implementation of the Corner Error of two images as:

$$C_{err}(n_x, n_y) = \sqrt{\frac{(n_x - n_y)^2}{n_x n_y}} \tag{4.6}$$

where n_x and n_y are respectively the corner count of two binary images, extracted using the Harris corner detector [30]. This formula may seem arbitrary, but it demonstrated to scale well on our dataset, since the corner count is actually limited by the size of our samples. The final metric is computed, as the other cases, averaging the corner error over the images in the batch:

$$MCE(X_{true}, X_{gen}) = \frac{1}{N} \sum_{i=0}^{N-1} C_{err}(n_{true,i}, n_{gen,i}) \tag{4.7}$$

Again, for simplicity, we indicated as $n_{true,i}$ and $n_{gen,i}$ the corner count given by $n_i = count(peak(Harris(X_i)))$, with obvious meaning of the function names. This metric is proportional to the average distance that the true and generated samples have, giving a quantitative measure of relative map complexity. It is worth nothing that it's not always the case, since generation artefacts may dramatically increase this value, producing a great number of corners. For this reason, this quantity reflects both the relative average complexity between batches of images and the presence of noise or artefacts in the generated samples.

4.5 WAD Editor and Feature Extractor

In the previous sections of this chapter we described the generative module of the system. The module that remains to be described is the one that copes with the two endpoints of the system, in particular with the conversion from WAD to Feature Maps (or features) and vice-versa.

4.5.1 Reading and Writing

As explained in section 3.4, data is hosted in an on-line archive. Due to the massive amount of levels a script for automatic download and file extraction has been written. This file also produces a preliminary JSON database, which is further expanded when WAD files are analysed.

The WAD parser that is provided by "The VGLC: The Video Game Level Corpus" didn't allow to extract all the features we needed for our data, while other Python modules that offered WAD file access didn't have enough documentation and support or missed features like the ability to create new files. For this reasons we proceeded to write a more complete editor, which offer the user a structured organization of the data that is contained in a WAD file using a more readable format. In particular each WAD file is read as a structured Python dictionary containing all the data we listed in chapter 3.4. The module has been realized so that a developer could ideally build a new map using only a few lines of code or even a PNG Image.

4.5.2 Feature Extraction

The feature extraction process is built upon our WAD Editor. In particular each WAD is first read as a structured Python dictionary, than it is processed to extract the set of features we provide with the dataset.

The process of generating the Feature Maps is quite straightforward: For each Sector the floor height is drawn as a filled polygon on an image and the sector tags annotated. Then, each linedef is drawn as a straight line, producing the WallMap, thus linedef triggers are matched with the relative sector tags for generating the TriggerMap. FloorMaps are derived by simply flattening the heightmap colour. During the process that generates the Feature Maps, the scalar features are computed on the feature maps themselves or directly from the sector and linedef data, depending on the feature. In this phase, the graph and the textual representation are also produced.

The WAD Editor is also able to produce WAD files from the feature map PNG representation of a level. In principle, it is possible to generate a level by using a bitmap image editor. A more interesting use would be using this editor to write the levels generated from the network back to a WAD file, and explore them directly in game. This is possible, at the cost of a slight loss of information due to the pixel representation of the level. In particular, line detection algorithms such as the Hough transform Line detection algorithm [16] or its probabilistic version [23] did not work well as expected in detecting walls from the levels generated by the network. Another approach we tried was using an edge detection algorithm for drawing sectors as contours, but this revealed to be too complex due to the way sectors are specified in WAD files (3.2). We used an

alternative approach, exploiting the information provided by the Room Adjacency Graph built upon the RoomMap: For each edge in the graph, which correspond to the boundary between a room and another, a set of walls is defined and their coordinates are annotated within the graph. Approximating each sector as an entire room the process of drawing the entire map room-by-room became more straightforward. The work of inserting height changes in parts that are smaller than a room can be done by further segmenting the heightmap when considering each room and it is left as a future work.

4.6 Summary

In this chapter we presented a framework that can be used to train generative models, in particular Generative Neural Networks, to produce new levels from a previously collected dataset. For describing our framework we showed the possible use cases, how they are accomplished by its modules and how the data is transformed from an high level perspective. In order to cope with the difficulties in sampling the network, we also provided possible methods for conditioning the network during the generation phase. We then detailed the neural network architecture we use in our system and the steps accomplished by the training function. For assessing the perceived quality of the samples during the training phase without relying on subjective evaluations, we proposed a set of metrics that can be monitored as the network is optimized. At the end of the chapter we introduced our endpoint modules for converting WAD levels into a set of Feature Maps and vice-versa and for extracting the features. In the next chapter we describe our we set up the experiments and show details on the trained networks.

Chapter 5

Experiment Design

This chapter describes the experiments we conducted using the system described in chapter 4. Section 5.1 shows the preliminary steps for preparing the system to our experiments, such as the selection of the input levels and the features. Section 5.1.4 shows the trained model structure we used in our experiments and 5.2 detail the three experiments we set up in order to study the trained networks.

5.1 Experiment setup

This section explains what choices we made before training the models in order to conduct the experiments. In particular we had to select which data to use given our technological constraints, the network input features and how to evaluate our models.

5.1.1 Input Levels Selection

For preparing our experiments we filtered the DoomDataset by taking only the samples up to 128x128 in size and which had exactly one "floor". This led to a dataset of 1088 samples, which are then augmented by rotation during the training process. This is motivated from the fact that even if the level orientation does not affect playability, using levels with more floors could lead the network to learn a correlation between floors (and how to arrange them inside the map) which could potentially be misleading or be just enforced by the sample size or the way the editor arranged them on the level coordinate space. Moreover, using only one-floor levels helped in reducing artefacts that appeared as very small floors in resulting output.

5.1.2 Feature Selection

In selecting which numerical features Y to use as network input we followed some assumption and criteria:

1. *Reconstructability:* In order to be able to analyse the resulting network, it is possible to use only features that can be reconstructed from the network output images with reasonable loss of information. For example, features what depends on the WAD representation of the level such as the number of sectors are too much dependant both on the editor used to build the level, and the algorithm we use to convert back images to WAD. On the contrary, features based on the level morphology in the generated images are a better choice since they are evaluated using the same algorithm.

2. *Visibility:* In order to be selected as an input, a feature has to have a visual impact on the samples. While in principle neural networks can extract complex and possibly inscrutable structure in the data, we found it reasonable to use only features that can have a visual feedback on the Feature Maps.

3. *Robustness:* Other than having a visual impact on the samples, a good feature must be robust to noise in the feature maps, in other words it must not change greatly for colour variation of a few pixels in the images.

We selected a subset of features that reasonably satisfied these properties by comparing the feature values of visually different levels. An example of level set and the corresponding values for the selected features is shown on figure 5.1.
A summary of the feature we used in our experiment is:

- *level equivalent diameter*: Diameter of the circle having the same area as the level.
- *level major axis length*: Length of the longest axis of the level, independent from the rotation.
- *level minor axis length*: Length of the shortest axis of the level, independent from the rotation.
- *level solidity*: Ratio between the area of the level and its convex hull. It is related on how much the is convex.
- *nodes (number of rooms)*: Count of the rooms in the room adjacency graph.
- *distmap skewness*: Skewness of the distribution of each pixel distance from the closest wall. This metric is visually related to the balance between large and small areas.
- *distmap kurtosis*: Kurtosis of the distribution of each pixel distance from the closest wall. This metric is visually related to the presence of both very large and very small areas, or area size variety.

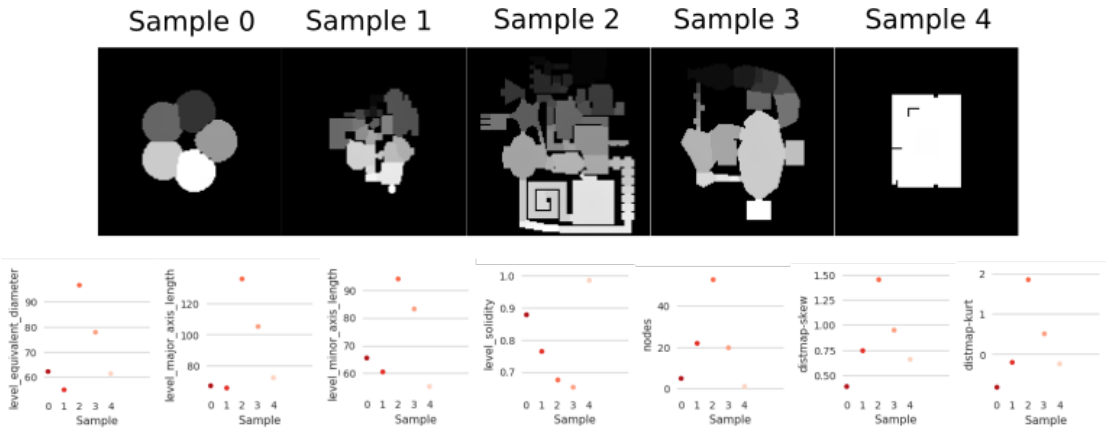

Figure 5.1: Example of feature values on a set of 5 different levels. The first row shows the Room Map of the levels, in which each room is enumerated with a different grayscale colour. The second row shows the feature values for the features *level equivalent diameter*, *level major axis length*, *level minor axis length*, *level solidity*, *nodes (number of rooms)*, *distmap skewness* and *distmap kurtosis*.

5.1.3 Framework Evaluation

In order to evaluate the feasibility of our approach to the problem of level generation, we designed a set of experiments for testing the impact of input features on the generative model. All the experiments involve comparing the distribution of true data and the one generated from the neural network, in particular only features that are informative according to our dataset have been considered. In generating the models, we first trained a neural network without input features so that the generator is only controlled by the noise vector Z. We then added a set of features to our architecture and used it to train a network using the same random initialization. Details on the network we produced are shown in table 5.1.

5.1.4 Trained Architectures

In training the models we kept fixed the WGAN-GP architecture, learning hyperparameters, the number of layers, the kernel size and the stride, while we varied the input features. During the development of the system we tried several different architectures and networks, but we show only the ones we conducted our experiments upon. For example, in our earlier experiments we tried using a single multi-valued map, but the results were affected from too much noise and artefacts. In the hope of obtaining better quality samples we also tried adding more layers to the network: although it showed to learn faster at the beginning, the architecture became unstable and the generator collapsed as soon it reached a quality comparable with the networks we present here. Table 5.1 shows the final models we used in our experiments:

Run Name	**Iterations**	**Features**	**Maps**	**D Layers (filters)**	**G Layers (filters)**
unconditional	36000	No features	floormap heightmap wallmap thingsmap	4 (1024, 512, 256, 128)	4 (128, 256, 512, 1024)
conditional	36000	level equivalent diameter level major axis length level minor axis length level solidity nodes distmap-skew distmap-kurt	floormap heightmap wallmap thingsmap	4 (1024, 512, 256, 128)	4 (128, 256, 512, 1024)

Table 5.1: Trained networks.

5.2 Experiments description

For assessing the capabilities of the networks in relation to the problem of level generation we designed three experiments, which are described in the following paragraphs. Since the second experiment depends on the result of the first one, the results are showed together in chapter 6.

5.2.1 Experiment 1: Unconditional generation

In the first experiment we tested the ability of the "unconditional" network to generate levels that exhibit features similar to the original ones. For each level in the dataset we sampled one level from the unconditional network using random noise as input, we extracted the features from the generated level and we compared the distribution of the dataset with the distribution of the generated features. For comparing the true and generated feature distributions we used the *Two-tailed Kolmogorov-Smirnov Statistical Test* [37] that utilizes a statistic calculated as the distance between the empirical distribution functions of the two samples. The problem we solved with *KS* can be resumed as:

$$\begin{aligned} H_0 &: F_G(x) = F_T(x), \forall x \\ H_1 &: F_G(x) \neq F_T(x), \text{for some x} \end{aligned} \tag{5.1}$$

where $F_G(x)$ and $F_T(x)$ are the empirical distribution functions of the generated and true features, respectively. The tests have been corrected using the Bonferroni correction using a family-wise error rate (significance for the entire experiment) of 0.05; in correcting p-values we also considered the tests of experiment 2. Results are shown alongside the results of experiment 2 in tables 6.1 and 6.2.

5.2.2 Experiment 2: Addition of input features

In the second experiment we tested if the addition of features to the network input can have some effect on the generation of the samples. For doing this, we replicated the experiment one using the conditional network, then we compared the results with those obtained from the unconditional network. For each level in the dataset we sampled one level, using the true feature vector as input and the same noise vectors used in experiment one. We then proceeded as in experiment one in testing the true and generated distributions of features.
For easily comparing the results of the unconditional and conditional networks we clustered the features in four groups, each one corresponding to a possible case:

- **F1:** Features for which the null hypothesis is rejected in both the unconditional and conditional networks.
- **F2:** Features for which the null hypothesis is rejected for the unconditional network (experiment 1) but cannot be rejected for the conditional network.
- **F3:** Features for which the null hypothesis cannot be rejected for both the unconditional and conditional network.
- **F4:** Features for which the null hypothesis cannot be rejected for the unconditional network (experiment 1) but is rejected for the conditional network.

Only for the features in the group F3, we also consider the distance (KS-stats) between the true and generated feature distributions in order to discover which network better reflects the feature distribution. This is reported as an asterisk *(*)* in the results tables 6.1 and 6.2, while values of the statistic are shown in Appendix A.

5.2.3 Experiment 3: Controlling the generation

In the third experiment we studied the impact of the input features on the generated levels from a qualitative point of view. In particular, we selected a set of levels from the dataset according to their feature values: for each input feature we selected the levels that match the 25th, 50th and 75th percentiles of the feature distribution, obtaining a total of $3 * |y_i| = 21$ levels. We then sampled 1000 levels from the conditioned network for each selected input feature vector, obtaining 1000 generated feature vectors for each input level. For each input feature, we plotted the three selected input values and the corresponding generated feature distributions, showing how the generated level distribution changes with respect to a change in the input feature. Results of this experiment are shown in figure 6.17.

5.3 Summary

In this chapter we described the input selection for our trained models, resulting in 1088 levels up to 128 pixels and having one floor. We then described the general principles we used to select the input features for the conditional network, selecting 7 representative features. We then proceeded to define three experiments for testing the two network capabilities and the effects of controlling the conditional network with the input features. In the next chapter we show the results we obtained, while in the following one we discuss the results and present our conclusions.

Chapter 6

Results

This chapter shows the results obtained from the training phase of the two networks and from the execution of the three experiments described in chapter 5. After that, the chapter provides a detailed discussion of the proposed results. In particular, Section 6.1 shows the graphs of the two networks losses and the sample evaluation metrics described in chapter 4. Section 6.2 shows the test outcomes for the three experiments we conducted. Section 6.3 shows a set of levels sampled from both networks. Section 6.4 provides our discussion on the results by considering them in the same order they are presented.

6.1 Training and Sample Metrics

In this section we show the metrics that are calculated during the training phase of each network, including the losses and the Sample Evaluation Metrics described in section 4.4.2. Figures 6.1 and 6.2 show the training and validation loss for the unconditional and conditional networks, respectively. For each feature map, Figures from 6.3 to 6.7 show the Entropy Mean Absolute Error, Figure 6.8 shows the Structural Similarity, Figures from 6.9 to 6.12 show the Encoding Error, Figure 6.13 and 6.14 show the Corner Error for the FloorMap and WallMap, respectively.

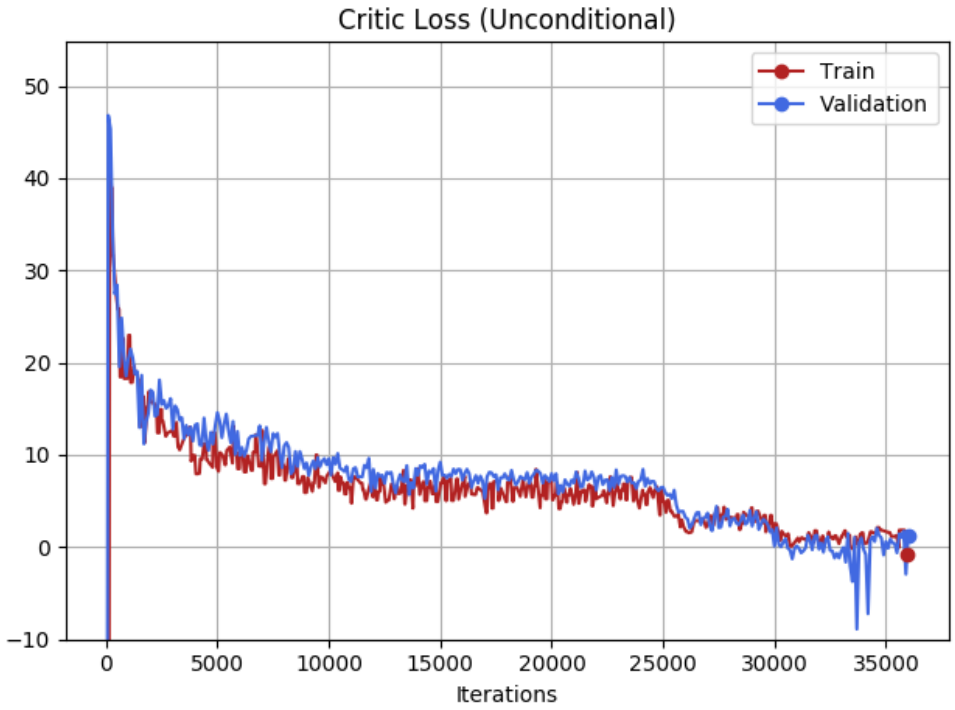

Figure 6.1: Unconditional Network Loss for the Critic: Training loss (red) and Validation Loss (blue) both converge toward zero.

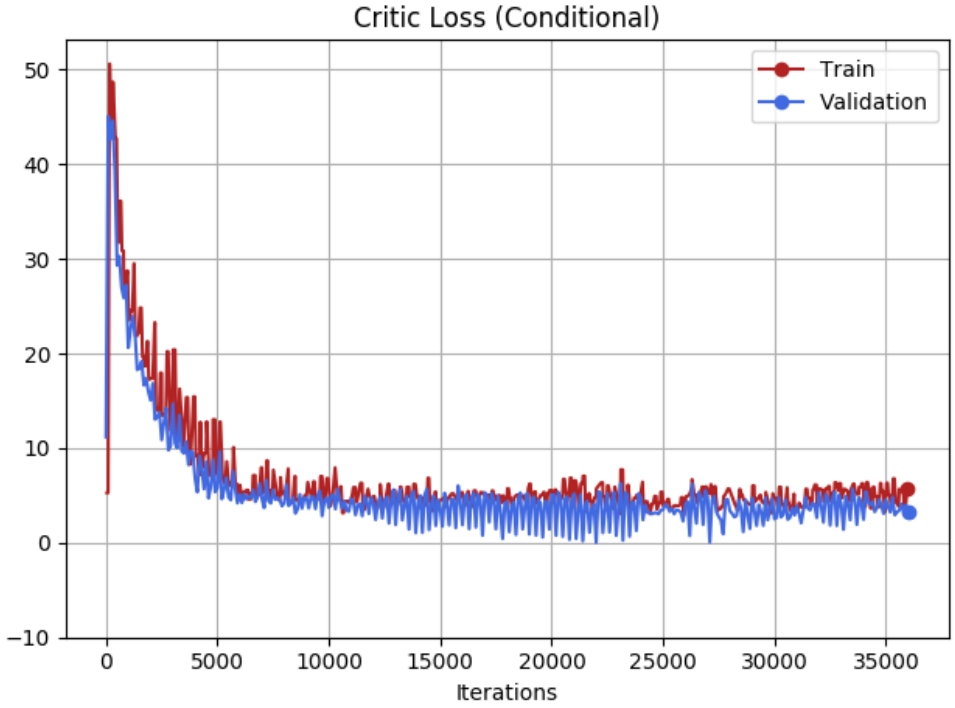

Figure 6.2: Conditional Network Loss for the Critic: Training loss (Red) and Validation Loss (Blue) both converge toward zero.

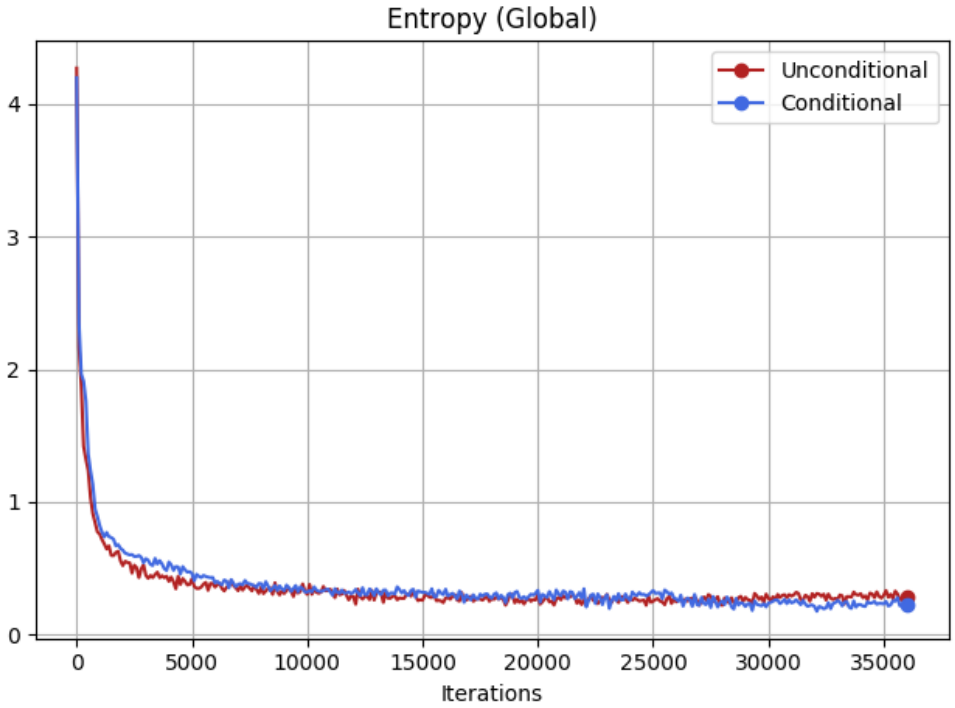

Figure 6.3: Sample Evaluation Metrics: Mean Entropy difference over all the maps (floormap, wallmap, thingsmap, heightmap). Unconditional (Red) and Conditional (Blue) networks generate samples having similar mean entropy.

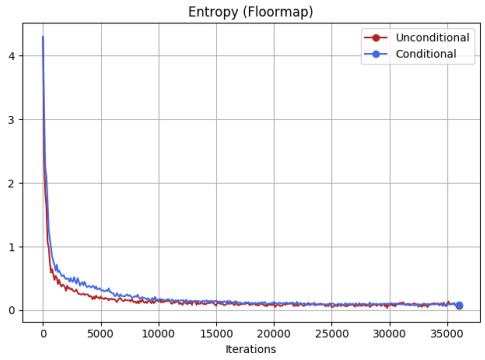

Figure 6.4: Sample Evaluation Metrics: Entropy MAE calculated on FloorMap. Unconditional (Red) and Conditional (Blue) networks generate samples having similar Floormap entropy.

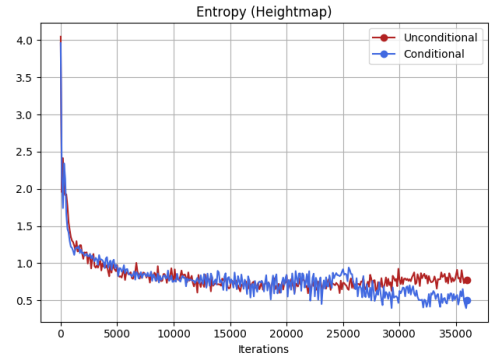

Figure 6.5: Sample Evaluation Metrics: Entropy MAE calculated on HeightMap. Conditional (Blue) network generate samples having slightly better HeightMap entropy than the Unconditional network (Red).

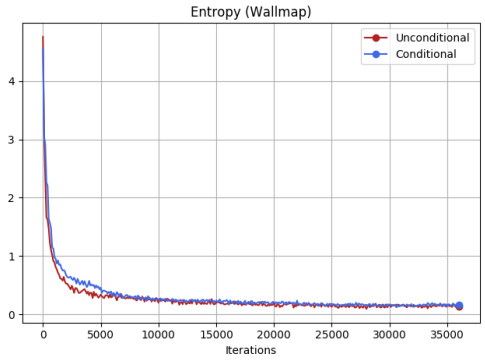

Figure 6.6: Sample Evaluation Metrics: Entropy MAE calculated on WallMap. Unconditional (Red) and Conditional (Blue) networks generate samples having similar WallMap entropy.

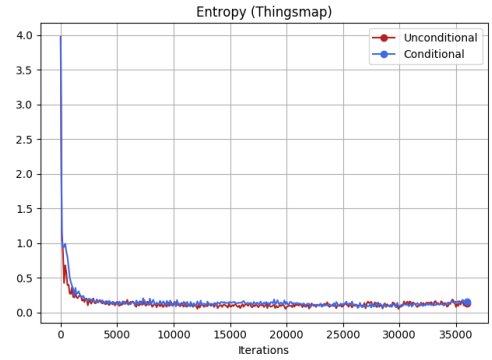

Figure 6.7: Sample Evaluation Metrics: Entropy MAE calculated on ThingsMap. Unconditional (Red) and Conditional (Blue) networks generate samples having similar ThingsMap entropy.

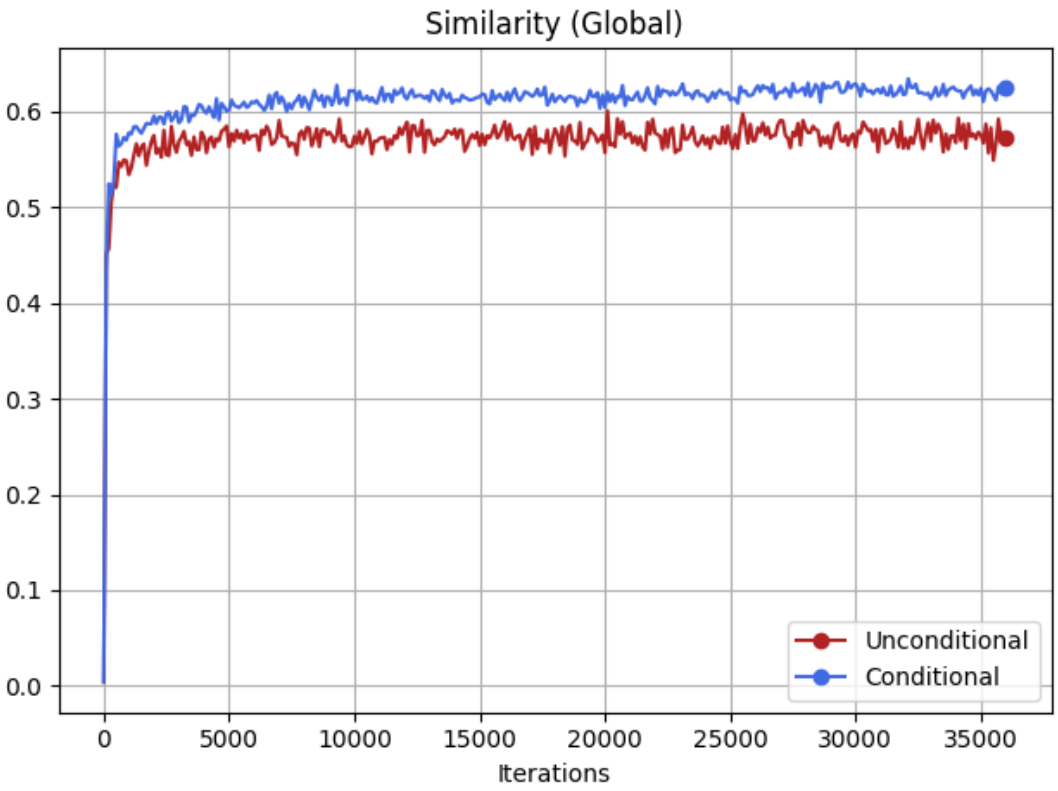

Figure 6.8: Sample Evaluation Metrics: Mean Structural Similarity over the samples. Conditional (Blue) networks generate samples being more similar to true ones than the Unconditional network (Red).

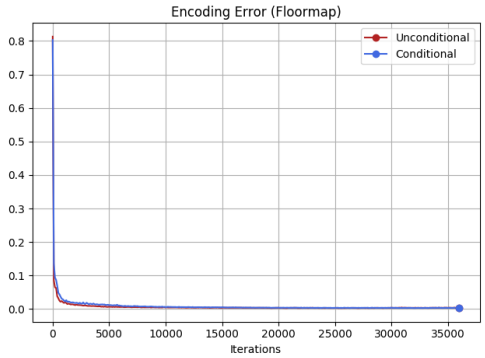

Figure 6.9: Sample Evaluation Metrics: Encoding Error calculated on FloorMap. Both Unconditional (Red) and Conditional (Blue) networks learns the colour scheme for representing the floors.

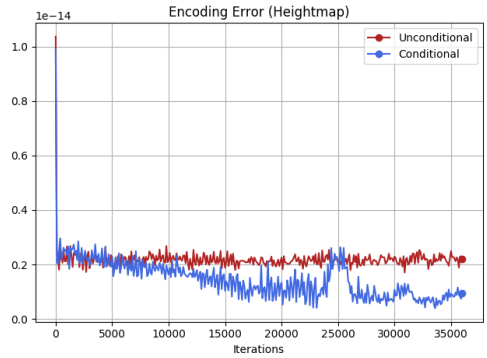

Figure 6.10: Sample Evaluation Metrics: Encoding Error calculated on HeightMap. Conditional (Blue) network is slightly more precise in representing the colour coding for the floor height than the Unconditional network (Red).

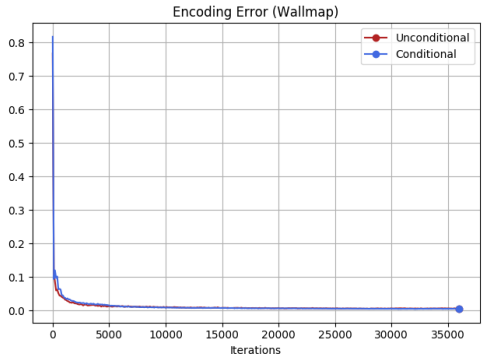

Figure 6.11: Sample Evaluation Metrics: Encoding Error calculated on WallMap. Both Unconditional (Red) and Conditional (Blue) networks learns the colour scheme for representing the walls.

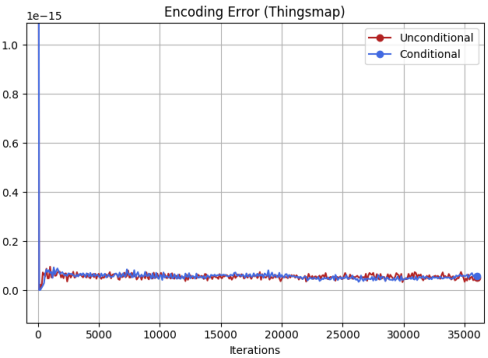

Figure 6.12: Sample Evaluation Metrics: Encoding Error calculated on ThingsMap. Both Unconditional (Red) and Conditional (Blue) networks learns the colour scheme for representing the game objects.

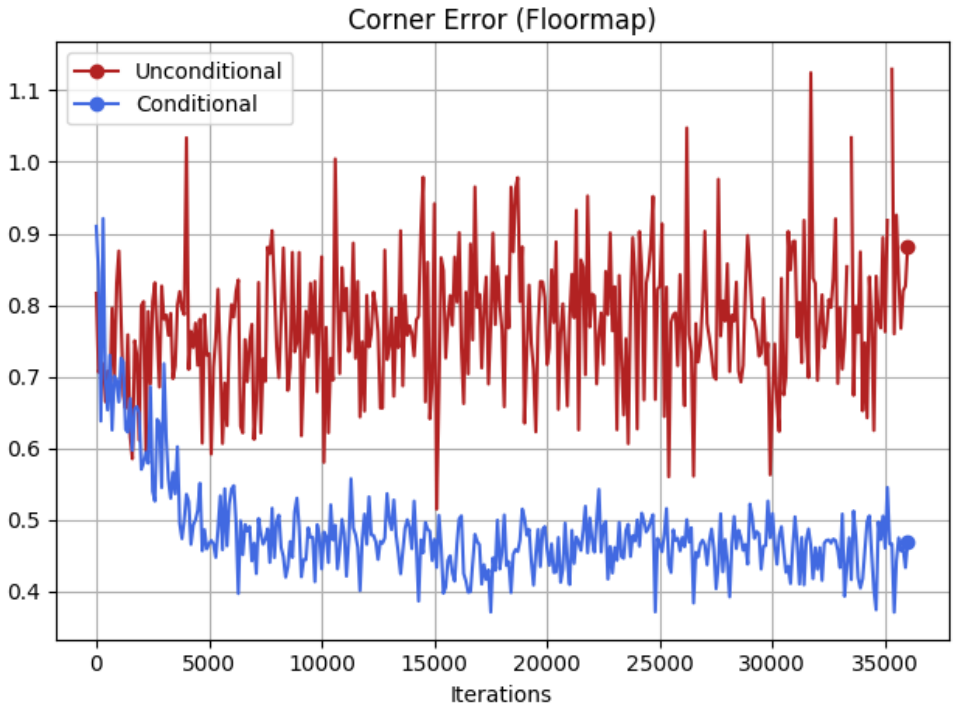

Figure 6.13: Sample Evaluation Metrics: Mean Corner Error over the FloorMap. Conditional (Blue) networks generate samples having a floor corner count closer to true levels than the Unconditional network (Red).

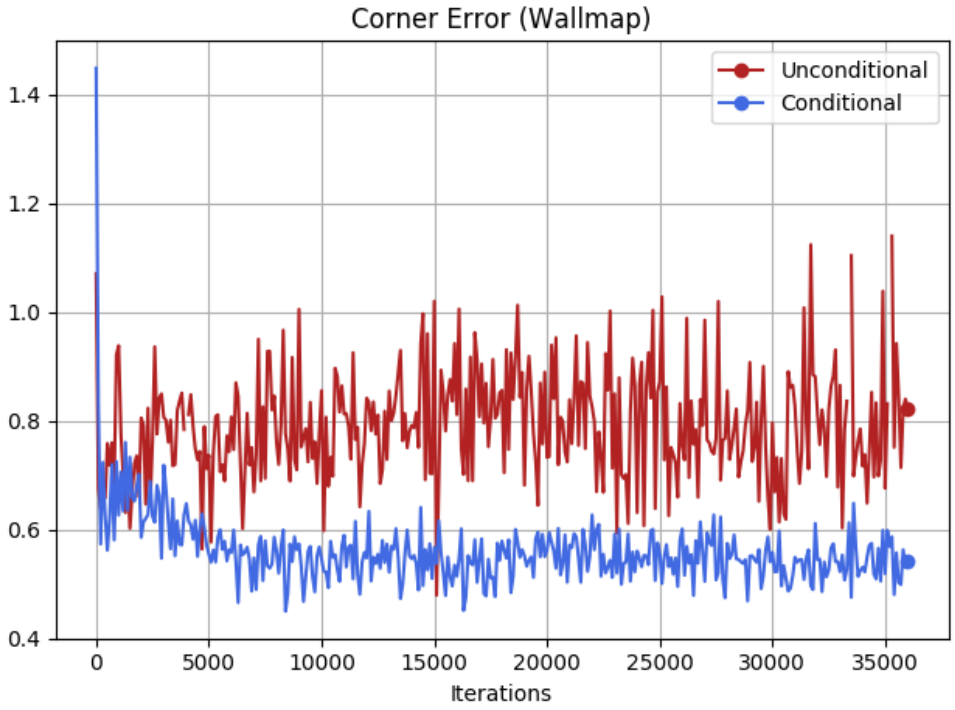

Figure 6.14: Sample Evaluation Metrics: Mean Corner Error over the WallMap. Conditional (Blue) networks generate samples having a wall corner count closer to true levels than the Unconditional network (Red).

6.2 Experiment Results

This section presents the test results for experiment 1 and 2, and the graphs produced by the experiment 3.

6.2.1 Results of experiments 1 and 2

Results for experiments 1 and 2 are shown respectively in the first two columns of Table 6.1 and 6.2. The third column indicates the feature group according to the description made in section 5.2.2. The set of features is divided in input and non-input features, with reference to the conditional network.

Input Features

Table 6.1: KS-test results for input features, using a significance level of 0.05 and the Bonferroni correction method. Results are indicated with R if the null hypothesis can be rejected or with N otherwise. An asterisk indicates the network that performed better (has the minimum KS distance) if the null hypothesis is rejected in every network

feature	uncond	cond	Group
level_equivalent_diameter	N	N	F3
level_major_axis_length	R*	R	F1
level_minor_axis_length	N	N	F3
level_solidity	R	R*	F1
nodes	R	R*	F1
distmap-skew	R	R*	F1
distmap-kurt	R	N	F2

Non-Input Features

Table 6.2: KS-test results for non-input features, using a significance level of 0.05 and the Bonferroni correction method. Results are indicated with R if the null hypothesis can be rejected or with N otherwise. An asterisk indicates the network that performed better (has the minimum KS distance) if the null hypothesis is rejected in every network

feature	uncond	cond	Group
level_area	N	N	F3
level_convex_area	N	N	F3
level_eccentricity	R*	R	F1
level_euler_number	R	R*	F1
level_extent	R	R*	F1
level_filled_area	N	N	F3
level_orientation	N	N	F3
level_perimeter	N	N	F3
level_hu_moment_0	N	R	F4
level_hu_moment_1	R*	R	F1
level_hu_moment_2	R	R*	F1
level_hu_moment_3	R	N	F2

Continued on next page

Table 6.2: KS-test results for non-input features, using a significance level of 0.05 and the Bonferroni correction method. Results are indicated with R if the null hypothesis can be rejected or with N otherwise. An asterisk indicates the network that performed better (has the minimum KS distance) if the null hypothesis is rejected in every network

feature	uncond	cond	Group
level_hu_moment_4	N	R	F4
level_hu_moment_5	N	R	F4
level_hu_moment_6	R	N	F2
level_centroid_x	R*	R	F1
level_centroid_y	R*	R	F1
number_of_artifacts	R	R*	F1
number_of_powerups	R	N	F2
number_of_weapons	R	R*	F1
number_of_ammunitions	R*	R	F1
number_of_keys	R	R*	F1
number_of_monsters	R*	R	F1
number_of_obstacles	R	R*	F1
number_of_decorations	R	R*	F1
walkable_area	N	N	F3
walkable_percentage	N	N	F3
start_location_x_px	R	R*	F1
start_location_y_px	R	R*	F1
artifacts_per_walkable_area	R	R*	F1
powerups_per_walkable_area	R	N	F2
weapons_per_walkable_area	R	R*	F1
ammunitions_per_walkable_area	R*	R	F1
keys_per_walkable_area	R	R*	F1
monsters_per_walkable_area	R	R*	F1
obstacles_per_walkable_area	R	R*	F1
decorations_per_walkable_area	R	R*	F1
avg-path-length	R	R*	F1
diameter-mean	R	R*	F1
art-points	R*	R	F1
assortativity-mean	R*	R	F1
betw-cen-min	N	N	F3
betw-cen-max	R	R*	F1
betw-cen-mean	R*	R	F1
betw-cen-var	R*	R	F1
betw-cen-skew	R	R*	F1
betw-cen-kurt	R	R*	F1
betw-cen-Q1	R*	R	F1
betw-cen-Q2	R*	R	F1
betw-cen-Q3	R*	R	F1
closn-cen-min	R*	R	F1
closn-cen-max	R	R*	F1
closn-cen-mean	R*	R	F1
closn-cen-var	R	R*	F1
closn-cen-skew	R*	R	F1
closn-cen-kurt	R*	R	F1
closn-cen-Q1	R*	R	F1

Continued on next page

Table 6.2: KS-test results for non-input features, using a significance level of 0.05 and the Bonferroni correction method. Results are indicated with R if the null hypothesis can be rejected or with N otherwise. An asterisk indicates the network that performed better (has the minimum KS distance) if the null hypothesis is rejected in every network

feature	uncond	cond	Group
closn-cen-Q2	R	R*	F1
closn-cen-Q3	R	R*	F1
distmap-max	R*	R	F1
distmap-mean	R*	R	F1
distmap-var	R*	R	F1
distmap-Q1	R*	R	F1
distmap-Q2	R	R*	F1
distmap-Q3	R	R*	F1

6.2.2 Graphical results for Experiments 1 and 2

Figure 6.15 shows the cumulative distribution functions relative to the features considered in table 6.1. Another view of the same data is proposed in figure 6.16 in which the probability densities for the input features are shown.

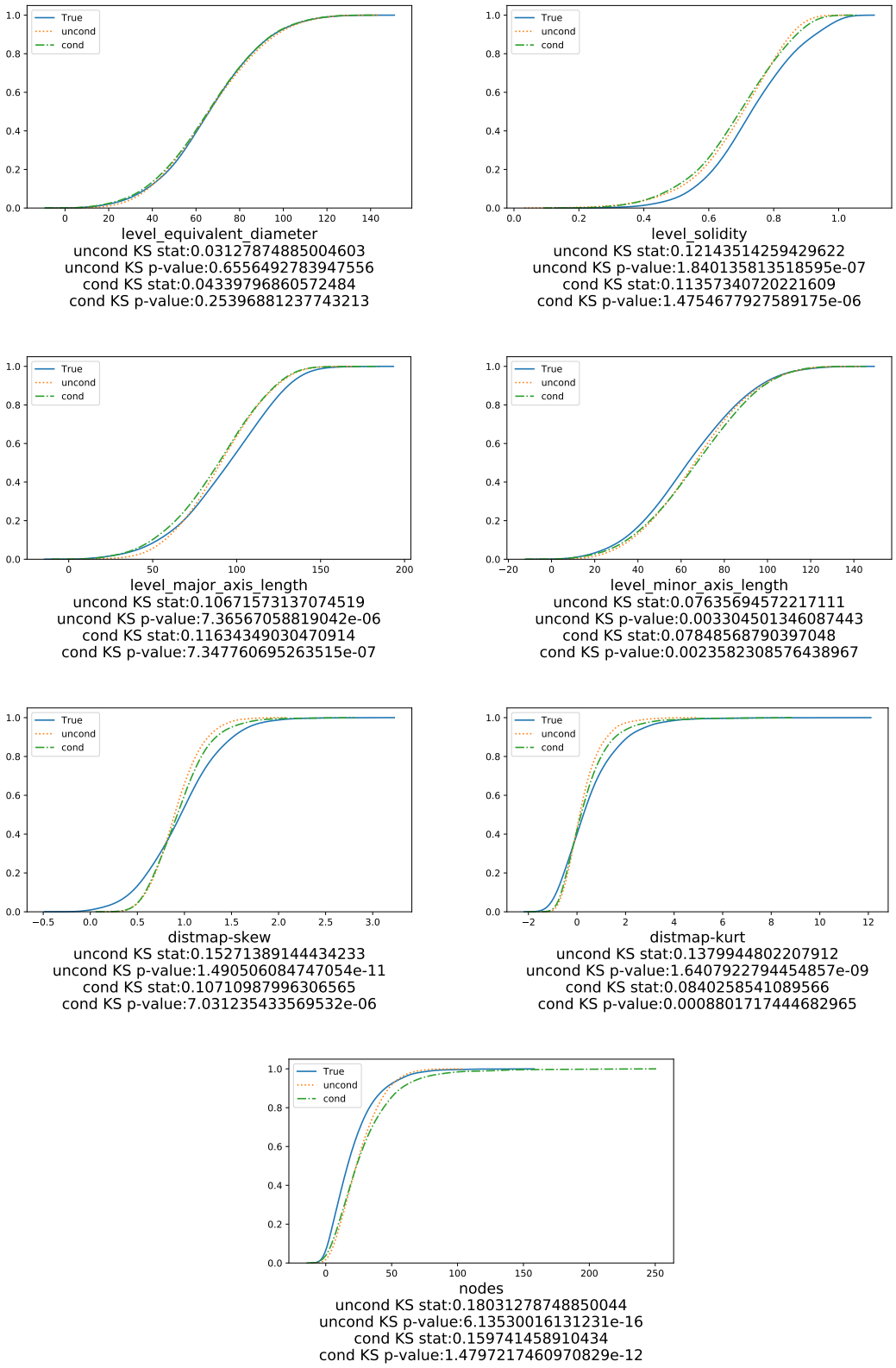

Figure 6.15: Experiments 1 and 2: Cumulative distribution functions for true data, unconditional network and conditional network for each input feature.

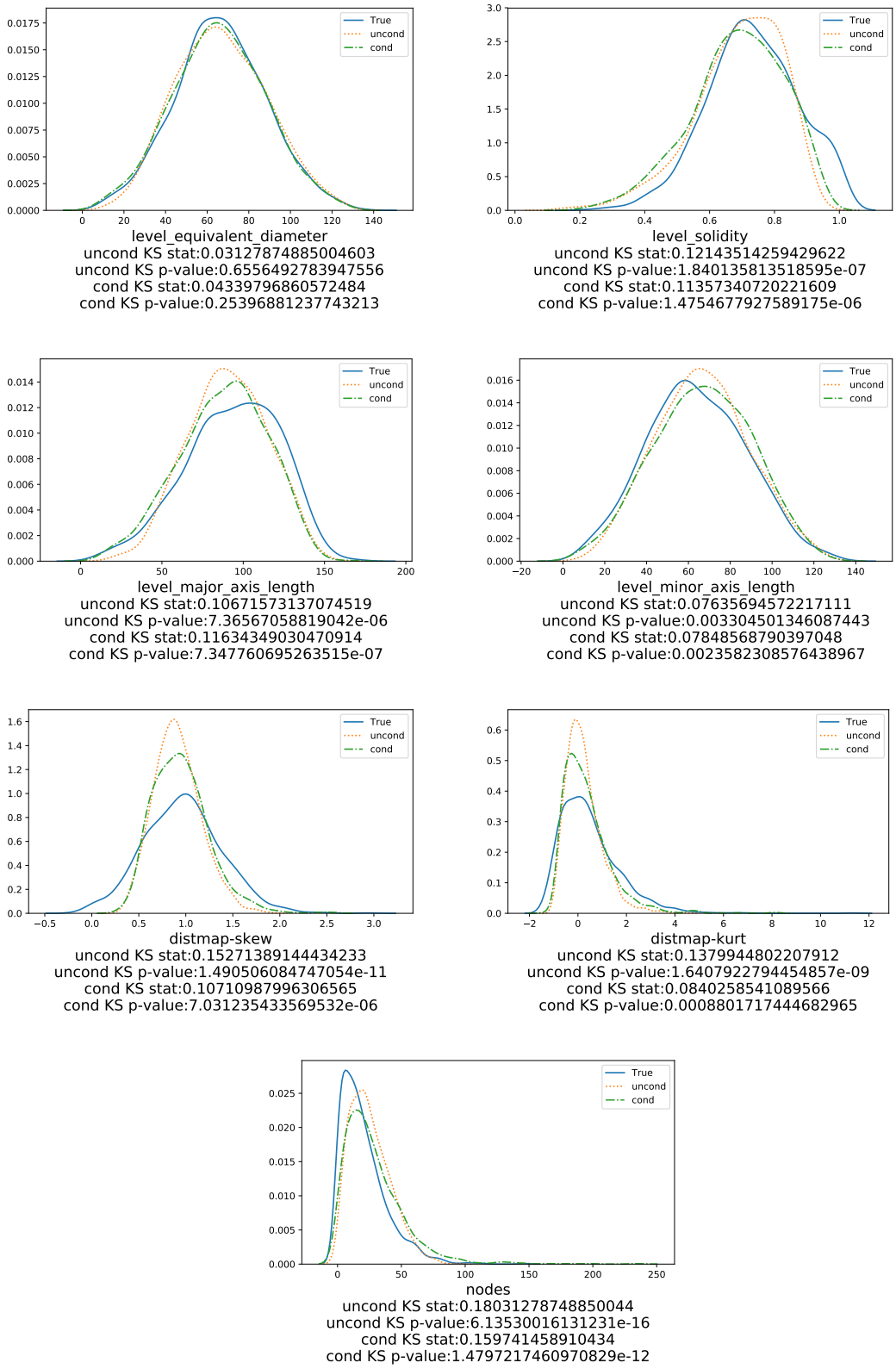

Figure 6.16: Experiments 1 and 2: Estimated probability density functions for true data, unconditional network and conditional network for each input feature.

6.2.3 Results of Experiment 3

Results of Experiment 3 are shown in figure 6.17. Each figure shows a different input feature. The three vertical lines correspond to the values of the three quartiles that have been used to generate the three distributions of 1000 levels each.

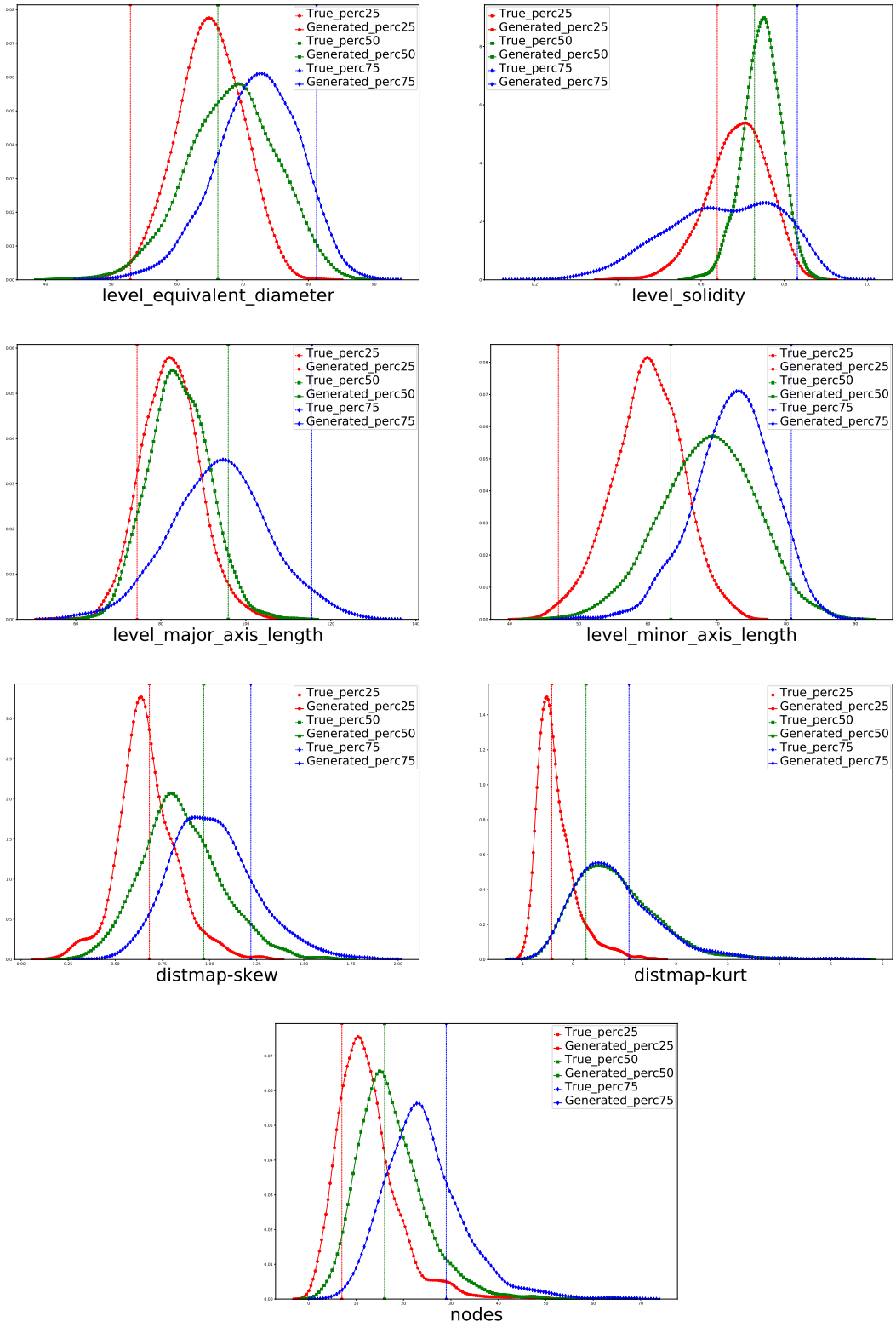

Figure 6.17: Experiments 3: Generated distributions for each true feature value in the cases of 25th (red, circles), 50th (green, squares) and 75th (blue, diamonds) percentiles.

6.3 Generated Samples

In this section we show a small set of samples that have been generated by the two networks using the "dataset" sampling approach introduced in section 4.2.3. Figures 6.18 and 6.19 show samples from the unconditional network, one level per row. Figures 6.20 and 6.21 show samples from the conditional network and the true levels that correspond to the input feature vector used to sample the network.

6.3.1 Unconditional Network

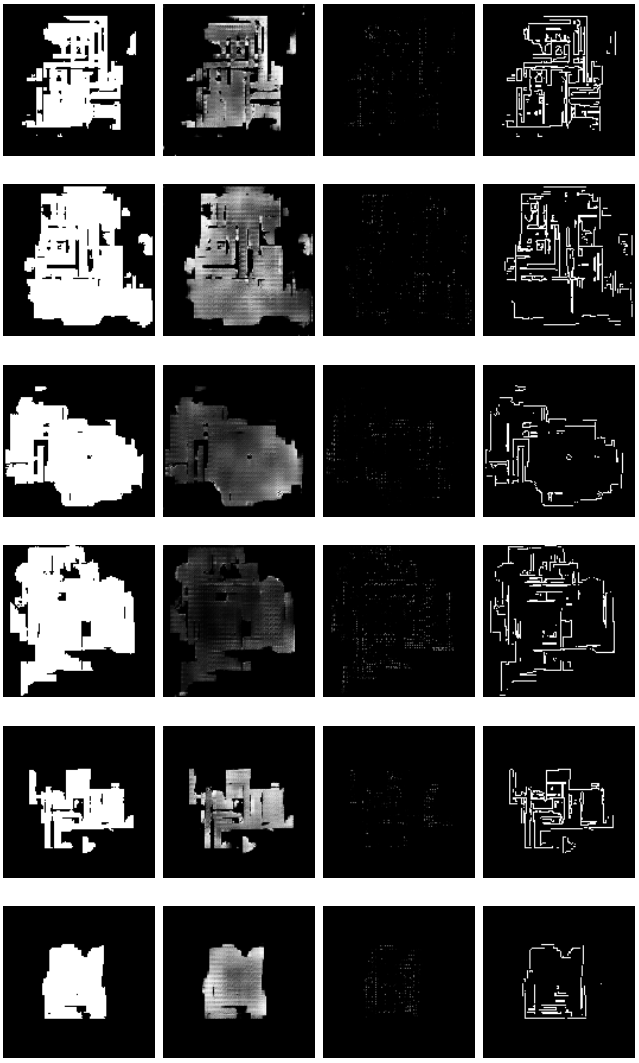

Figure 6.18: Samples generated by the unconditional network (1 of 2). From left to right: Floormap, Heightmap, Thingsmap, Wallmap

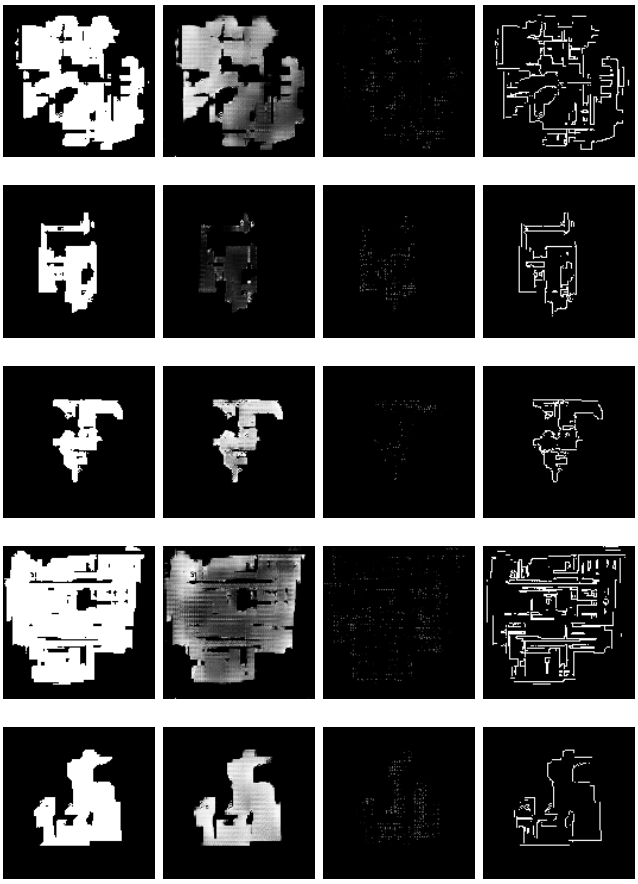

Figure 6.19: Samples generated by the unconditional network (2 of 2). From left to right: Floormap, Heightmap, Thingsmap, Wallmap

6.3.2 Conditional Network

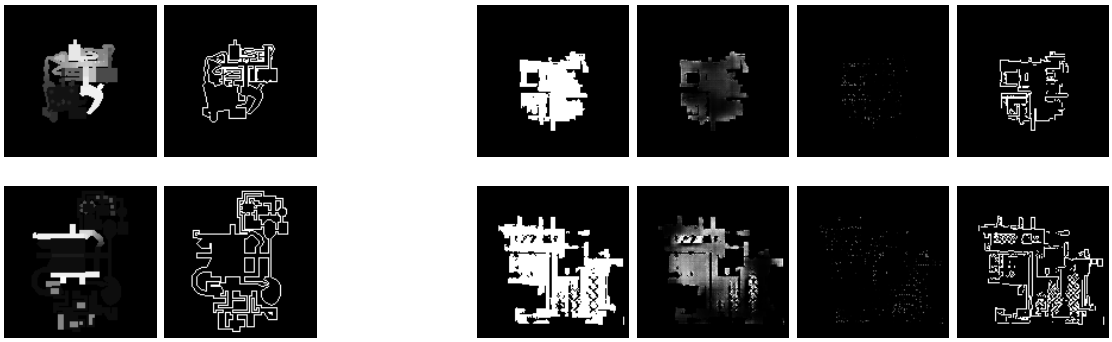

Figure 6.20: Samples generated by the Conditional network (1 of 2). In the left column, the heightmap and the wallmap of the *true* level that corresponds to the feature vector used to generate the corresponding level. On the right column, the corresponding generated Floormap, Heightmap, Thingsmap and Wallmap

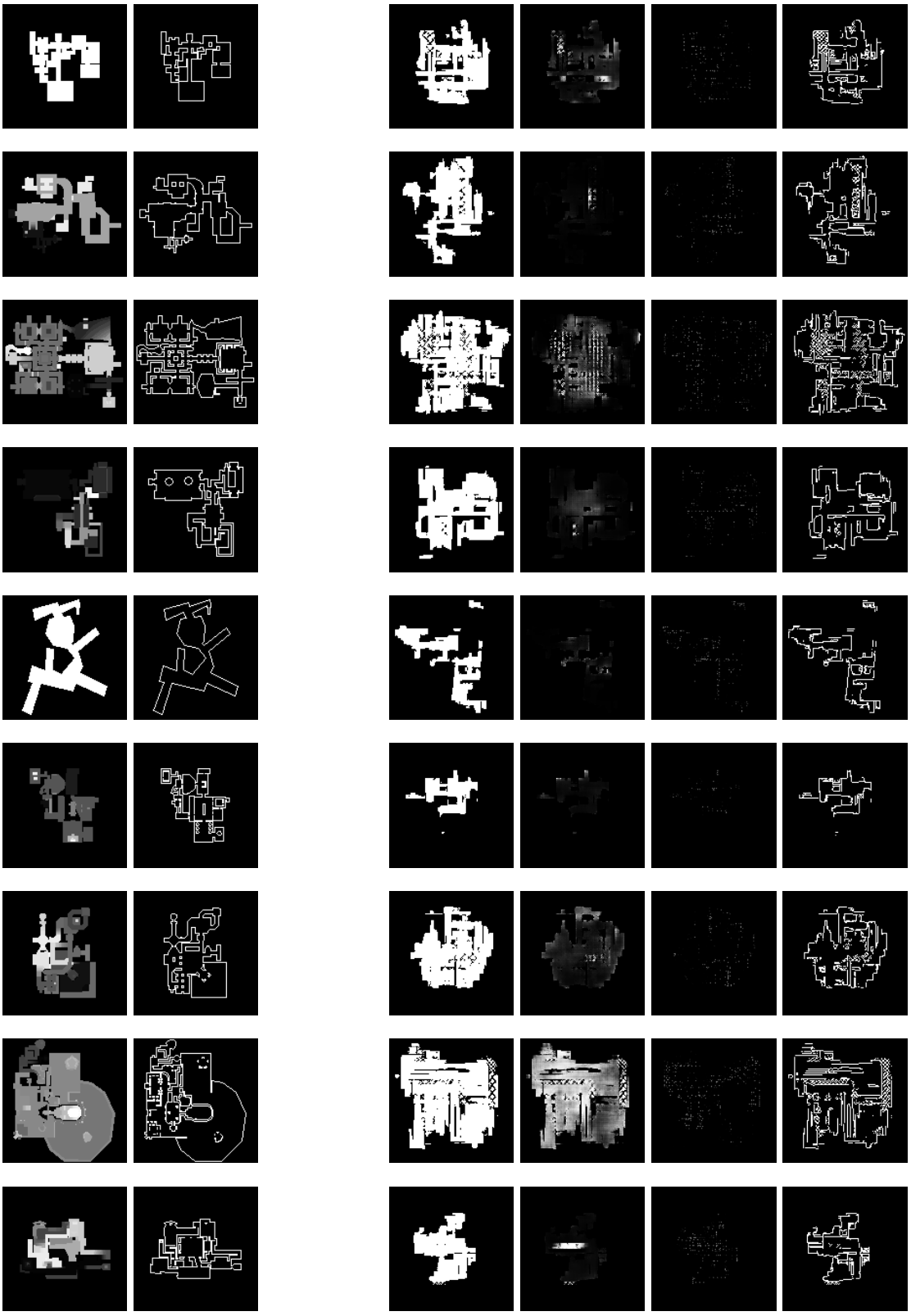

Figure 6.21: Samples generated by the Conditional network (2 of 2). In the left column, the heightmap and the wallmap of the *true* level that corresponds to the feature vector used to generate the corresponding level. On the right column, the corresponding generated Floormap, Heightmap, Thingsmap and Wallmap

6.4 Results Evaluation

We conduct the analysis of the results in the same order as they are presented in the previous sections, eventually making reference to content which is included in the Appendix. The first part of our discussion considers the metrics that have been monitored during the training phase, explaining the results. After that, we consider the results of the experiments described in section 5.2.

6.4.1 Sample Evaluation Metrics

Training losses in figures 6.2 and 6.1 confirm the advantages of the WGAN model over earlier proposals, it is indeed possible to appreciate a converging behaviour similar to that of classical Neural Networks in both the conditional and unconditional case. This proved to be useful for understanding when to stop the training phase, since a manual inspection of the samples wouldn't be informative of the actual network capabilities. The fact that the validation loss follows the behaviour of the training loss also suggests a good generalization capability of the critic in assigning level scores even for previously unseen samples.

Entropy Mean Absolute Errors in figure 6.3 converge toward a small value in the same way the loss does, confirming that the WGAN loss is actually correlated to the sample quality in our case. In particular this metric is related to the difference in information content, or noise, between true and generated samples. While a small difference is still detectable, both networks showed to reduce this difference as the training proceeded.

Mean Structural Similarity (Figure 6.8) shows that the conditional network generates samples that are structurally more similar to true levels than the unconditional network does. This is also true if we consider the networks in their early training stage. It's worth nothing that the structural similarity is a metric for evaluating the perceived sample quality, so it reaches unitary values only if the two samples are the same. In our setting we compare different levels so we cannot treat SSIM as an absolute value but only as an indicator of the "relative quality" of the samples generated by the two networks.

Encoding Error in figures 6.9 to 6.12 shows that the network is capable to easily learn and reproduce the colour coding used in each separated map. This does not mean that the network is able, for example, to only output either the value 0 or 255 when generating a floormap, but this metric shows that the average error is small. For this reason, the only post-processing we apply to generated samples in order to compute the generated features is to threshold the colour value toward the closest meaningful value (eg. a 244 in a floormap is interpreted as a 255).

Corner Error in figures 6.13 and 6.14 shows that while the unconditional network generates levels that don't significantly improve their corner count with respect to the true ones, the conditional network actually learns to generate more accurate levels. This value could hardly reach low values as two levels having the same features (with reference to the ones we have selected as inputs) could naturally differ in their topology, raising the value of this metric. Another aspect that is worth nothing is the lower variance in the conditional case, possibly reflecting a lower amount of noise or artefacts in generated levels.

6.4.2 Experiments Discussion

The first columns of tables 6.1 and 6.2 show for what features it is not possible to reject the hypothesis that the true distribution and the distribution generated by the unconditional network are equal. If we analyse these features we can notice that many features that are "well learned" by the network in their output depend on the level area or the perimeter. In particular, figure 6.15 and 6.16 show how the two networks are able to reproduce the level area distribution (level_equivalent_diameter) and the length of the minor axis of the levels.

The second column of the tables shows the same concept for the conditional network. In particular, table 6.1 shows substantial improvements on the input features: In this case, the feature *distmap-kurt* (associated to the "variety" in room dimensions), is now distributed enough closely to the real distribution that the null hypothesis cannot be rejected any more. The input features that have been learned by the unconditional network are also learned by the conditional network. For the features *solidity, nodes (number of rooms)* and *distmap skewness (balance between big and small areas)* the conditional network cannot reproduce a distribution that is close enough to the real one to change the outcome of the test, however the statistic values in table A indicate that the distributions of the conditional network are the closest to the real one [1].

If we analyse the results in table 6.2 and graphs in Appendix B we can confirm in many case the behaviour stated in the previous paragraph: features which are better represented by both networks (group F3) are related with the levels area. Features expressing locations of particular points of the maps such as the topological centroid or even explicitly represented as the player starting point, are unsurprisingly not well represented by the networks. The features regarding the number of items in the level, which would be useful from a design point of view, are still not well represented probably due to the still too high presence of checker board artefacts in the *Thingsmap*. However, the conditional proved again to be the best of the two networks in describing this set of features, learning particularly well with the distribution of "power-ups". The last set of listed features is, as expected, not well learned by the networks since it is composed by features which are graph-metrics or other too complex features, for which this kind of model is not suitable.

Graphs of figure 6.17 show how the unconditional network response varies by changing each input features to the values that correspond to the three quartiles of the true distribution. Even if in no cases the input feature can deterministically control the output feature of the value, for the majority of features it is possible to alter the output distribution by acting on the input value. This fact is represented by the ordering of the curves, in many case reflecting the ordering of requested values. In particular we can notice how all the features except the *solidity* and *distmap-kurt* react well to values changes. The particular shape of some curves such in *distmap-kurt* and *major axis* may suggest that the location of the output features does not linearly follow the translation of the requested feature or it could work as intended only in some areas of feature space. The particular undesired behaviour of the 75th percentile of *level solidity* can be due to the network failing to learn the representation of such high *solidity* values, or the presence of many artefacts in the corresponding generated levels leading to a incorrect calculation of the feature. It is worth noting that due to the sampling issues we considered in the previous chapters, this experiment has been conducted by selecting different feature vectors that exhibited the desired value on a particular feature. The possible drawback of this experiment is that the visualized distributions could depend on other factors other than the single requested feature value, however we observed better or similar results in earlier networks we trained with different settings, suggesting that the issue we just highlighted could have marginal impact on our results.

[1] This fact is also indicated with an asterisk in table 6.1

6.4.3 Visual Acknowledgement

The results we discussed in the earlier sections indicate that the conditional network has some advantages over the unconditional version, which can be briefly resumed as a better overall sample quality (Sample Evaluation Metrics), a better learning of input and other topological features (experiments 1 and 2) and the possibility to control the level generation up to some extent (Experiment 3). In a last consideration about these results we want to highlight that our experiments focused on proving the ability of the networks to reproduce the true feature distributions of the original dataset. This means that if the null hypothesis for one feature cannot be rejected then we can assume that its distribution may be "close enough" to the true one to generate levels which are similar to the existing ones. The contrary, however, does not necessarily prevents the model to be effectively used as a tool for generating levels, but only indicates that the generated levels somehow differs from the true ones. This fact can be appreciated by a visual comparison of the true and generated samples we proposed in section 6.3: in many cases it is possible to notice that the global shape and size of the generated samples vaguely resembles that of the true levels, indicating that the network actually considers some of the requested features up to a certain point. On the same figure it is possible to notice some struggle of the network in representing smaller features such level borders, probably due to generation noise. This can alter the calculated features and lead to inaccurate distributions, for this reason we recommend to apply some morphological processing or noise reduction techniques before converting the generated images to playable WAD files in order to reduce the generation noise.

6.5 Summary

In this chapter we presented our results by first showing the quality difference in samples generated by the two networks, according to the evaluation metrics defined in section 4.4.2 and then showing experimental results for testing the relation between the true and generated features and the impact of the input features in the conditional network. For providing a visual comparison of the generated levels, we also showed a set of generated samples for each network. In the last section we discussed in detail our results by first analysing the sample evaluation using evaluation metrics, then by considering the results of the experiments that analyse the networks behaviour with respect to the input features. In the next chapter, we provide more general considerations about our work and highlight the open problems and the possible future works.

Chapter 7

Conclusions and Future Work

In this chapter we make general conclusions about our work, while providing a set of open problems and the works that still have to be made. Section 7.1 reports our final conclusions, while section 7.2 highlights the possible future develops for enhancing our results.

7.1 Conclusions

In chapter 6 we showed how the addition of features to the network inputs increases the quality of generated samples and leads to better learning of many features, while also providing a method to influence the network output during the sampling process. One of the most common concerns is that the network could overfit the training set, learning to reproduce samples from the dataset. While we cannot prove it formally, we refer to the results of Huang et al. in [32, Appendix C], which claim that overfit is difficult to occur in the type of model we used, even for a small number of training samples, de facto demonstrating that the behaviour of GANs is different from that of classical deep neural networks used for classification.

Although the model we designed is far for being perfect in solving the level generation problem for 2d non-linear environments such DOOM maps, we think it's still a good starting point for future improvements and could represent a viable alternative to classical Procedural Generation. In particular, most levels generated from the networks have proved to be interesting to explore and play due to the presence of particular features typical of doom maps, such as narrow tunnels and large rooms. This suggests that one of the advantages of our method with respect to Procedural Generation is that in our case there is no need of an expert designer to embed their knowledge in the generation process; still, this method allows the designer to focus on the selection of more high-level features as those we selected as network inputs. This generative method, which is commonly used to produce visually appealing images, proved to be applicable to a topological setting like ours, even if not without issues: While in creating a picture of a face, a small variation in colour intensity is quite unnoticeable and can be tolerated, in our domain even a pixel-sized difference in a level map could alter drastically the level topology itself, for example creating a new access between two areas. This issue is emphasized by the output noise which is quite common in samples generated by a GAN.

7.2 Future Work

In this section we first present the open problems we encountered in our setting, then we propose some specific works that can be made to improve our system.

7.2.1 Open Problems

Data Availability

The amount of data available is a problem that affects in general every deep learning setting. As discussed in earlier chapters, the context of video-games is one of the fields in which data is less available and uniform. In our work we used a dataset of 1088 levels which have been augmented by rotation due to memory size constraints, but we envision that a larger amount of levels could make the network more accurate in generating levels. We propose a possible improvement for our system in section 7.2.2.

Samples Evaluation

As we explained in section 4.4.2, the problem of evaluating the samples generated by a GAN is still a recent field of research and a general prevailing model still have to be proposed. Moreover, the particular domain of our work makes even more difficult to apply the commonly used methods to assess sample quality. In section 4.4.2 we proposed a qualitative method for assessing the generated sample quality during the training process that works with our data. While the method we applied succeeds in indicating that the network is actually learning the level structures, the metrics we proposed have the drawback that they need to be calculated on each map differently in order to benefit of their informational power, while considered altogether for assessing the general sample quality.

Loss of accuracy

The system we designed assumes that one pixel is equivalent to 32 Doom Map Units since it's the diameter of the smallest object in DOOM. While this ensures that objects cannot overlap on the image representation, it introduces an unavoidable loss of accuracy in object positioning. Moreover, using a single pixel for representing objects such as in "Thing Maps" makes the task of distinguish generated object from noise and artefacts more difficult. While we weren't able to detect checker board artefacts in the structural maps such as the floormap, wallmap and heightmaps, the generated ThingsMaps often show an object placement that is too regular, resembling the issue discussed in section 4.4.1. We envision that an increase of the dataset resolution, for example setting one pixel size to 16 MU instead of 32, would help in reducing this problem. This comes at the cost of choosing between an high resolution in representing the maps and an higher resolution in network input/output, which also allows to use more levels from the full dataset.

Improving Network Sampling

Due to the high dimensionality of the feature space, sampling the conditional network using arbitrary feature values often resulted in low quality samples, for this reason we sample the network around points in the feature space for which the network have been trained. However, in real applications it could be better to have the designer specify his own desired input features, such the area, the balance between large halls and corridors, etc. We envision that this issue could be reduced using more training data and possibly using soft labels as proposed by [54], at the cost of a possible increase of noise in the output. An alternative design could be the implementation of the "Content Sampling" introduced in section 4.2.3.

7.2.2 Possible Applications and future develops

Augmenting input data

In our work we only used levels having a single "floor", or connected figure, in order to prevent the network for learning unnecessary patterns like the position or rotation of the various floors that compose the level when they actually are irrelevant from a gameplay point of view. One method for increasing the number of training samples is re-generating the dataset considering each floor as a separated level and calculating the features on a floor basis. An additional method for using a larger part of the data we collected is to simply increase the input/output sample size, although this require more computational capabilities.

Improving samples to WAD conversion

In our work we focused on the generation process, while we also provided a simple method for playing the generated levels in DOOM. However, this method is still not perfect: only preliminary work have been done on applying the height differences in levels, and no visual improvements such as automatic texture selection have been implemented yet. Moreover, work have still to be done for decorating the level with doors, elevators and switches.

Sample interpolation and Style Transfer

An interesting practice for generating diverse samples and assessing the generalization capabilities of the network is the interpolation of samples in the feature space [68]. We provided a method for sampling interpolated batches of levels by providing a start and end point in feature space, while keeping the noise vector fixed. If we consider adding levels from other games to the dataset by converting them to the same domain we used, this technique can be easily used for interpolating levels from different games and study what different features they exhibit.

Tweaking the model

For increasing the model capabilities a lot of work can still be done. One first attempt could be tweaking the network hyperparameters in order to find a setting that allows the network to learn a better representation. Our attempts didn't find any improvements over the default settings, although we cannot prove we are using optimal values.

Different architectures

Due to the constant research in generative models, there is an always increasing number of different new models to try and experiment in order to improve the generation of samples. In our work we limited our search only to the mostly used pure-GAN models which could easily run on our machines, but a large number of models can still be applied. For example, one method showing good results on faces generation that mixes GANs and Autoencoders is the BEGAN model proposed by Berthelot, Schumm, and Metz in [6]. Another approach could be using high-resolution models such as in *"Progressive Growing of GANs for Improved Quality, Stability, and Variation"* [38], showing interesting improvements in visual quality by growing the generator and the discriminator progressively.

Possible Applications

The most obvious application of our work would be the case of off-line video-game level generation. In particular our work finds application whenever it is needed to obtain 3d maps which doesn't overlap on the height axis. Since this is the only particular requirement for levels to work with our framework, it is possible to extend this system to other environments and domains by just changing the feature extraction modules. In other words, the system is independent of the technology of the particular game we used for training, and it could be virtually used in

any type of 3d game or simulation in which the user can move in two dimensions, eventually with changes of floor height. Possible applications in another fields could be the generation of environments for training or testing the behaviour of AI agents, and the use of the trained discriminator network for classification tasks by adding a last layer and fine tuning the resulting network. However, due to the set-up requirements currently needed for running the project, a most probable application that we envision of our work is as a starting point for the development of more complex systems or more advanced studies on Procedural Content Generation via Machine Learning applied to this type of environments.

Appendix A

Appendix: Test Statistics Values and Corrected p-values

Table A.1: KS statistic values for the tests. The value is correlated with the distance of the cumulative distributions of the true and generated data

feature	uncond-s	cond-s
level_area	0.031279	0.043398
level_convex_area	0.047838	0.052632
level_eccentricity	0.144434	0.223453
level_equivalent_diameter	0.031279	0.043398
level_euler_number	0.226311	0.180055
level_extent	0.127875	0.104340
level_filled_area	0.035879	0.042475
level_major_axis_length	0.106716	0.116343
level_minor_axis_length	0.076357	0.078486
level_orientation	0.058878	0.068329
level_perimeter	0.080957	0.060942
level_solidity	0.121435	0.113573
level_hu_moment_0	0.077277	0.098800
level_hu_moment_1	0.125115	0.190212
level_hu_moment_2	0.121435	0.104340
level_hu_moment_3	0.106716	0.076639
level_hu_moment_4	0.070837	0.121884
level_hu_moment_5	0.079117	0.094183
level_hu_moment_6	0.091076	0.059095
level_centroid_x	0.726771	0.777470
level_centroid_y	0.812328	0.815328
number_of_artifacts	0.467341	0.369344
number_of_powerups	0.128795	0.083102
number_of_weapons	0.383625	0.241921
number_of_ammunitions	0.359706	0.422899
number_of_keys	0.963201	0.943675
number_of_monsters	0.514259	0.574331
number_of_obstacles	0.519779	0.438596
number_of_decorations	0.823367	0.784857

Continued on next page

Table A.1: KS statistic values for the tests. The value is correlated with the distance of the cumulative distributions of the true and generated data

feature	uncond-s	cond-s
walkable_area	0.034959	0.046168
walkable_percentage	0.051518	0.081256
start_location_x_px	0.232751	0.166205
start_location_y_px	0.529899	0.382271
artifacts_per_walkable_area	0.553818	0.390582
powerups_per_walkable_area	0.181233	0.088643
weapons_per_walkable_area	0.397424	0.242844
ammunitions_per_walkable_area	0.399264	0.444137
keys_per_walkable_area	0.959522	0.938135
monsters_per_walkable_area	0.597056	0.591874
obstacles_per_walkable_area	0.665133	0.518006
decorations_per_walkable_area	0.897884	0.845799
nodes	0.180313	0.159741
avg-path-length	0.174793	0.156048
diameter-mean	0.155474	0.141274
art-points	0.115915	0.139428
assortativity-mean	0.427032	0.456802
betw-cen-min	0.006440	0.006464
betw-cen-max	0.376265	0.347184
betw-cen-mean	0.371665	0.384118
betw-cen-var	0.362069	0.375894
betw-cen-skew	0.250230	0.238227
betw-cen-kurt	0.215271	0.207756
betw-cen-Q1	0.218951	0.258541
betw-cen-Q2	0.279669	0.287165
betw-cen-Q3	0.301748	0.318560
closn-cen-min	0.624655	0.655586
closn-cen-max	0.173873	0.133887
closn-cen-mean	0.207912	0.213296
closn-cen-var	0.314402	0.278856
closn-cen-skew	0.632935	0.644506
closn-cen-kurt	0.378105	0.406279
closn-cen-Q1	0.201472	0.208680
closn-cen-Q2	0.176633	0.169898
closn-cen-Q3	0.171113	0.163435
distmap-max	0.214351	0.238227
distmap-mean	0.159154	0.202216
distmap-var	0.192272	0.214220
distmap-skew	0.152714	0.107110
distmap-kurt	0.137994	0.084026
distmap-Q1	0.201472	0.208680
distmap-Q2	0.176633	0.169898
distmap-Q3	0.171113	0.163435

Table A.2: Corrected p-values using Bonferroni method

feature	uncond	cond
level_area	1.000000e+00	1.000000e+00
level_convex_area	1.000000e+00	1.000000e+00
level_eccentricity	3.203350e-08	5.274459e-22
level_equivalent_diameter	1.000000e+00	1.000000e+00
level_euler_number	1.051357e-22	1.114460e-13
level_extent	4.541127e-06	1.922426e-03
level_filled_area	1.000000e+00	1.000000e+00
level_major_axis_length	1.060657e-03	1.058078e-04
level_minor_axis_length	4.758482e-01	3.395852e-01
level_orientation	1.000000e+00	1.000000e+00
level_perimeter	2.148826e-01	1.000000e+00
level_solidity	2.649796e-05	2.124674e-04
level_hu_moment_0	4.074087e-01	6.589815e-03
level_hu_moment_1	9.779487e-06	1.816326e-15
level_hu_moment_2	2.649796e-05	1.922426e-03
level_hu_moment_3	1.060657e-03	4.646614e-01
level_hu_moment_4	1.000000e+00	2.495058e-05
level_hu_moment_5	2.969827e-01	1.747566e-02
level_hu_moment_6	3.173520e-02	1.000000e+00
level_centroid_x	2.720185e-250	1.263132e-285
level_centroid_y	4.022221e-313	2.723494e-314
number_of_artifacts	1.768827e-102	4.055502e-63
number_of_powerups	3.503463e-06	1.500697e-01
number_of_weapons	1.735132e-68	4.330549e-26
number_of_ammunitions	5.282114e-60	2.739392e-83
number_of_keys	0.000000e+00	0.000000e+00
number_of_monsters	1.874772e-124	4.438253e-155
number_of_obstacles	3.544692e-127	1.020608e-89
number_of_decorations	1.422909e-321	4.121210e-291
walkable_area	1.000000e+00	1.000000e+00
walkable_percentage	1.000000e+00	2.092123e-01
start_location_x_px	4.085103e-24	2.123218e-11
start_location_y_px	3.028440e-132	9.750062e-68
artifacts_per_walkable_area	1.297132e-144	8.624320e-71
powerups_per_walkable_area	6.130510e-14	5.296084e-02
weapons_per_walkable_area	1.249309e-73	2.653075e-26
ammunitions_per_walkable_area	2.496041e-74	4.829900e-92
keys_per_walkable_area	0.000000e+00	0.000000e+00
monsters_per_walkable_area	2.341589e-168	8.331551e-165
obstacles_per_walkable_area	2.346143e-209	7.877455e-126
decorations_per_walkable_area	0.000000e+00	0.000000e+00
nodes	8.834832e-14	2.130799e-10
avg-path-length	7.611113e-13	7.638737e-10
diameter-mean	8.430551e-10	9.359119e-08
art-points	1.117706e-04	1.650707e-07
assortativity-mean	5.053350e-80	1.523040e-90
betw-cen-min	1.000000e+00	1.000000e+00

Continued on next page

Table A.2: Corrected p-values using Bonferroni method

feature	uncond	cond
betw-cen-max	8.086915e-66	1.434635e-55
betw-cen-mean	3.542549e-64	2.070974e-68
betw-cen-var	4.891902e-55	5.050308e-59
betw-cen-skew	3.829013e-28	3.017244e-25
betw-cen-kurt	2.227206e-20	8.710051e-19
betw-cen-Q1	3.848983e-21	4.810963e-30
betw-cen-Q2	1.379923e-35	1.804461e-37
betw-cen-Q3	1.034416e-41	1.644635e-46
closn-cen-min	1.911063e-184	1.371244e-202
closn-cen-max	1.082671e-12	8.660043e-07
closn-cen-mean	6.820595e-19	6.777393e-20
closn-cen-var	4.501303e-41	1.054396e-31
closn-cen-skew	2.056190e-189	9.669788e-196
closn-cen-kurt	1.760014e-66	9.755770e-77
closn-cen-Q1	1.235166e-17	5.717695e-19
closn-cen-Q2	3.740492e-13	5.455660e-12
closn-cen-Q3	3.081734e-12	5.768903e-11
distmap-max	3.438308e-20	3.017244e-25
distmap-mean	2.362809e-10	1.046636e-17
distmap-var	6.608149e-16	4.399457e-20
distmap-skew	2.146329e-09	1.012498e-03
distmap-kurt	2.362741e-07	1.267447e-01
distmap-Q1	1.235166e-17	5.717695e-19
distmap-Q2	3.740492e-13	5.455660e-12
distmap-Q3	3.081734e-12	5.768903e-11

Appendix B

Appendix: Graphical results for non-input features

We omit results for features that are not informative in our specific context such as features calculated on floor basis, since the dataset we used only consisted on one-floor levels and they would match the corresponding level based features.

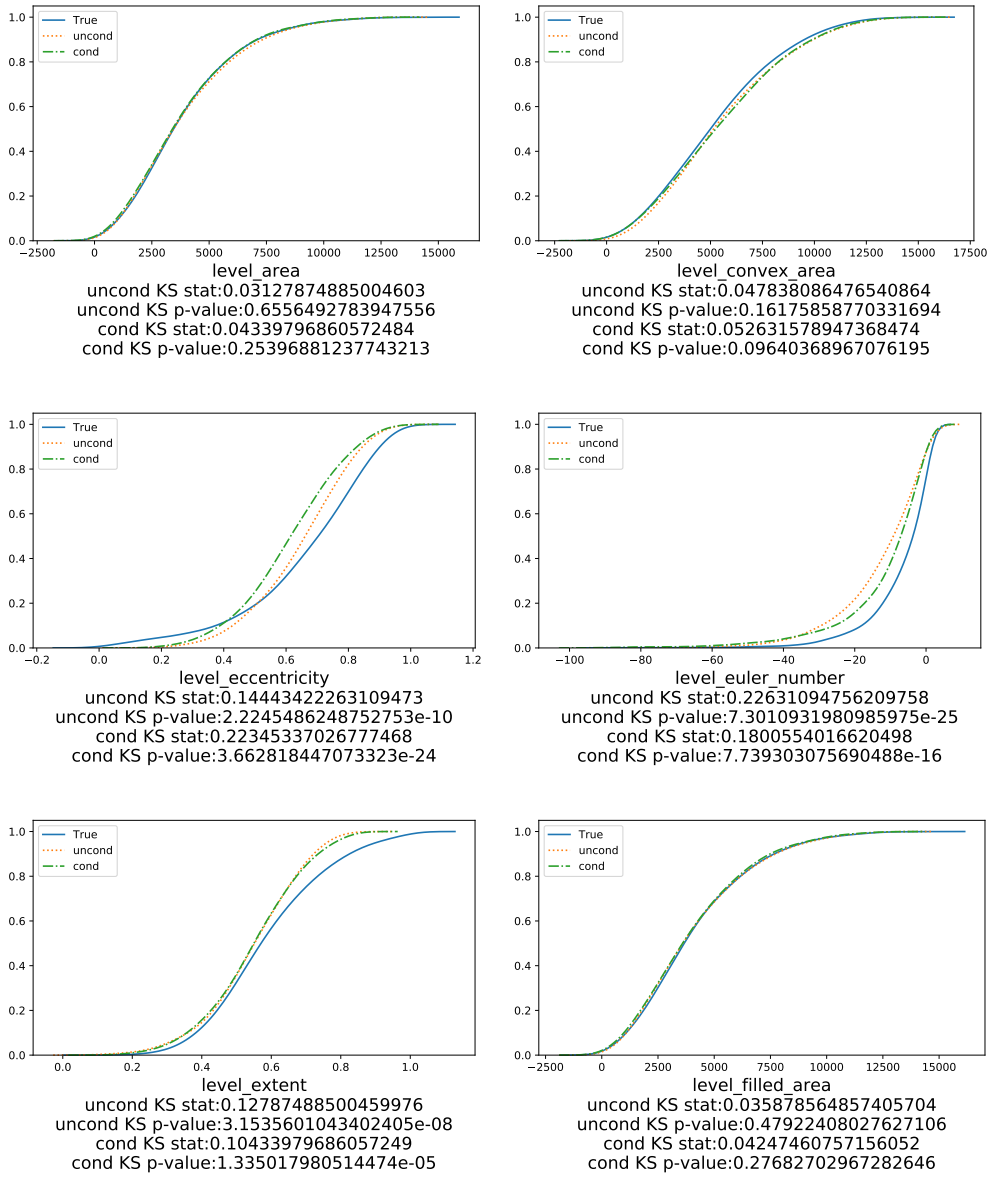

Figure B.1: Experiments 1 and 2: Cumulative distribution functions for true data, unconditional network and conditional network for each non-input feature.

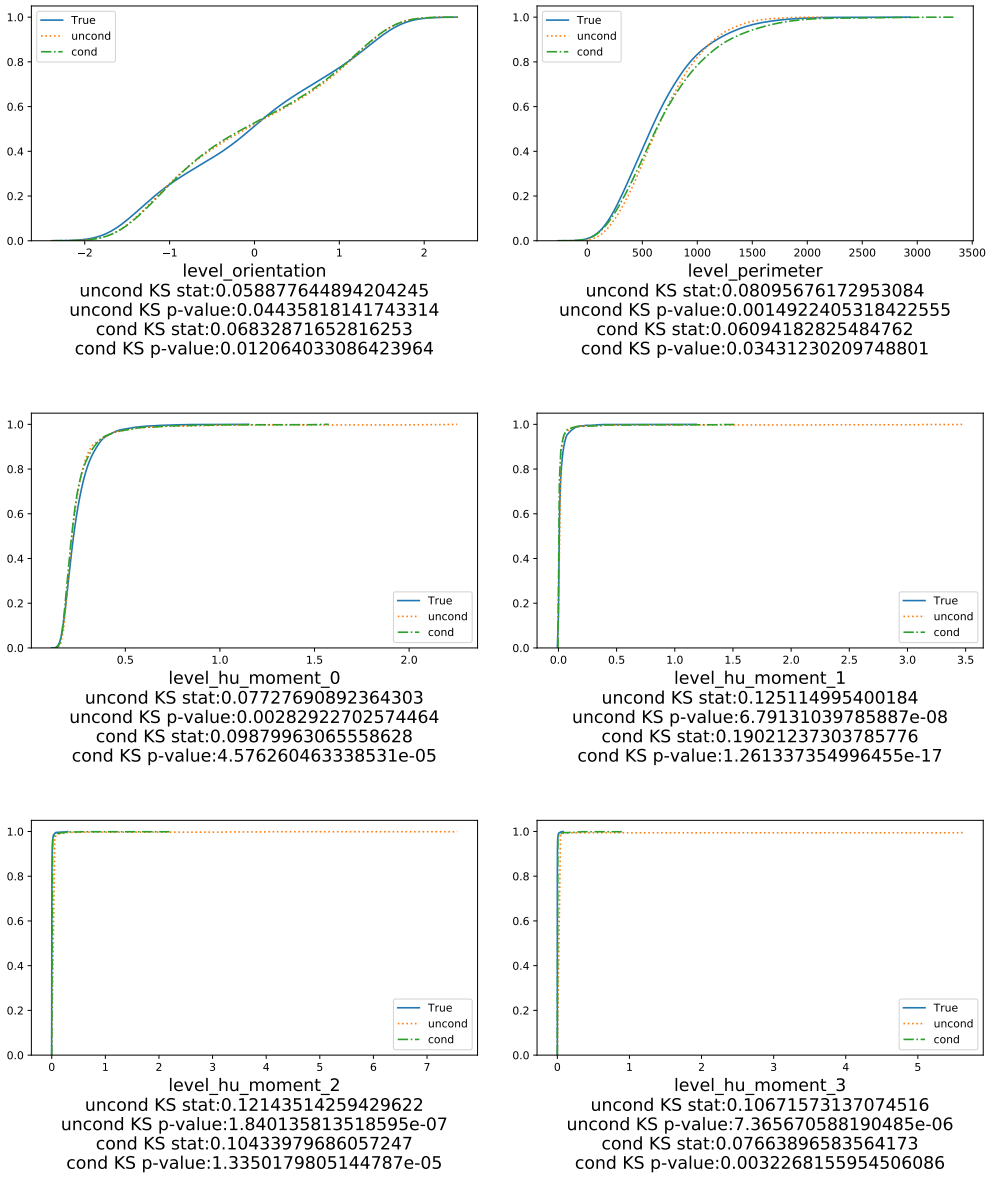

Figure B.2: Experiments 1 and 2: Cumulative distribution functions for true data, unconditional network and conditional network for each non-input feature.

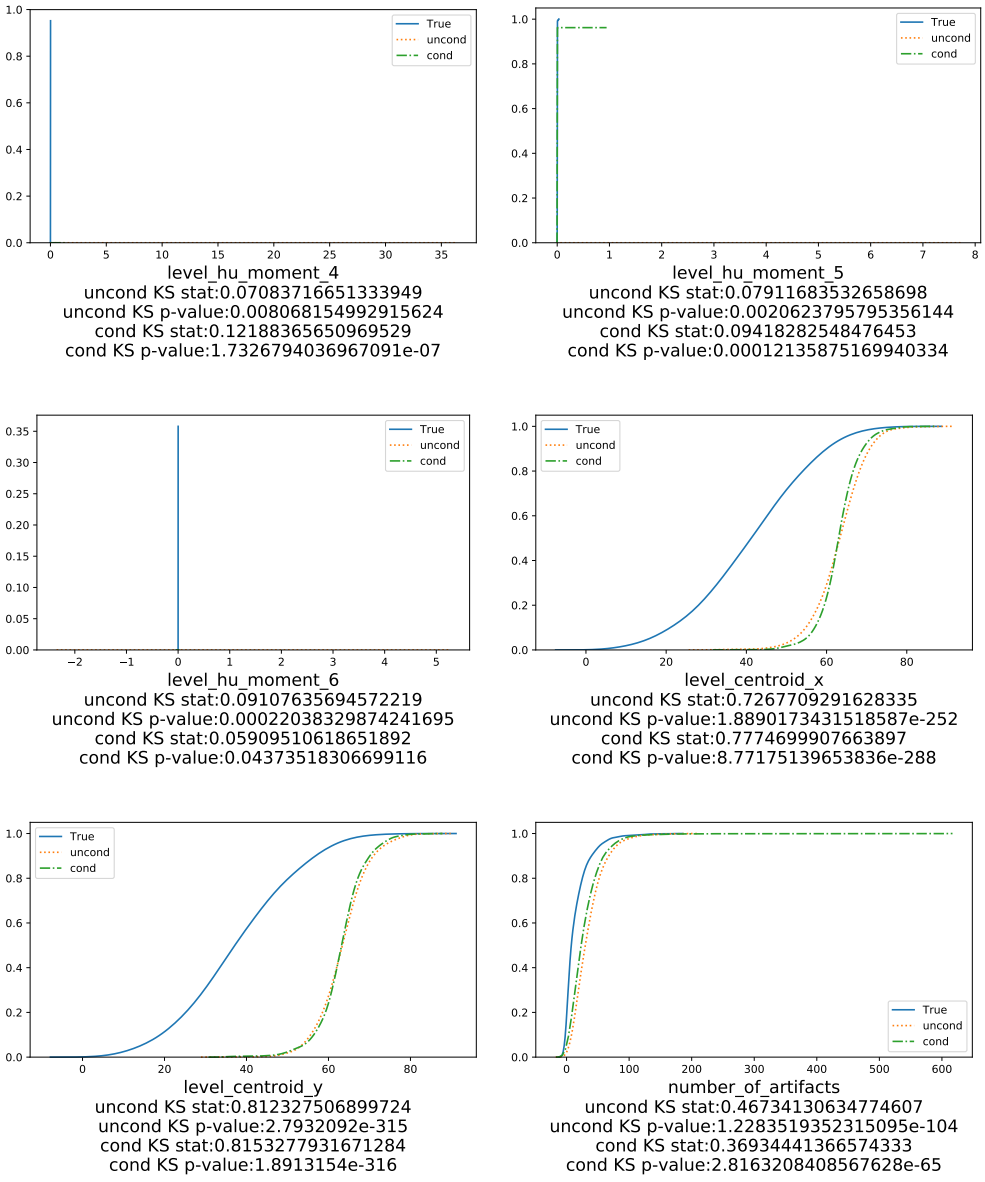

Figure B.3: Experiments 1 and 2: Cumulative distribution functions for true data, unconditional network and conditional network for each non-input feature.

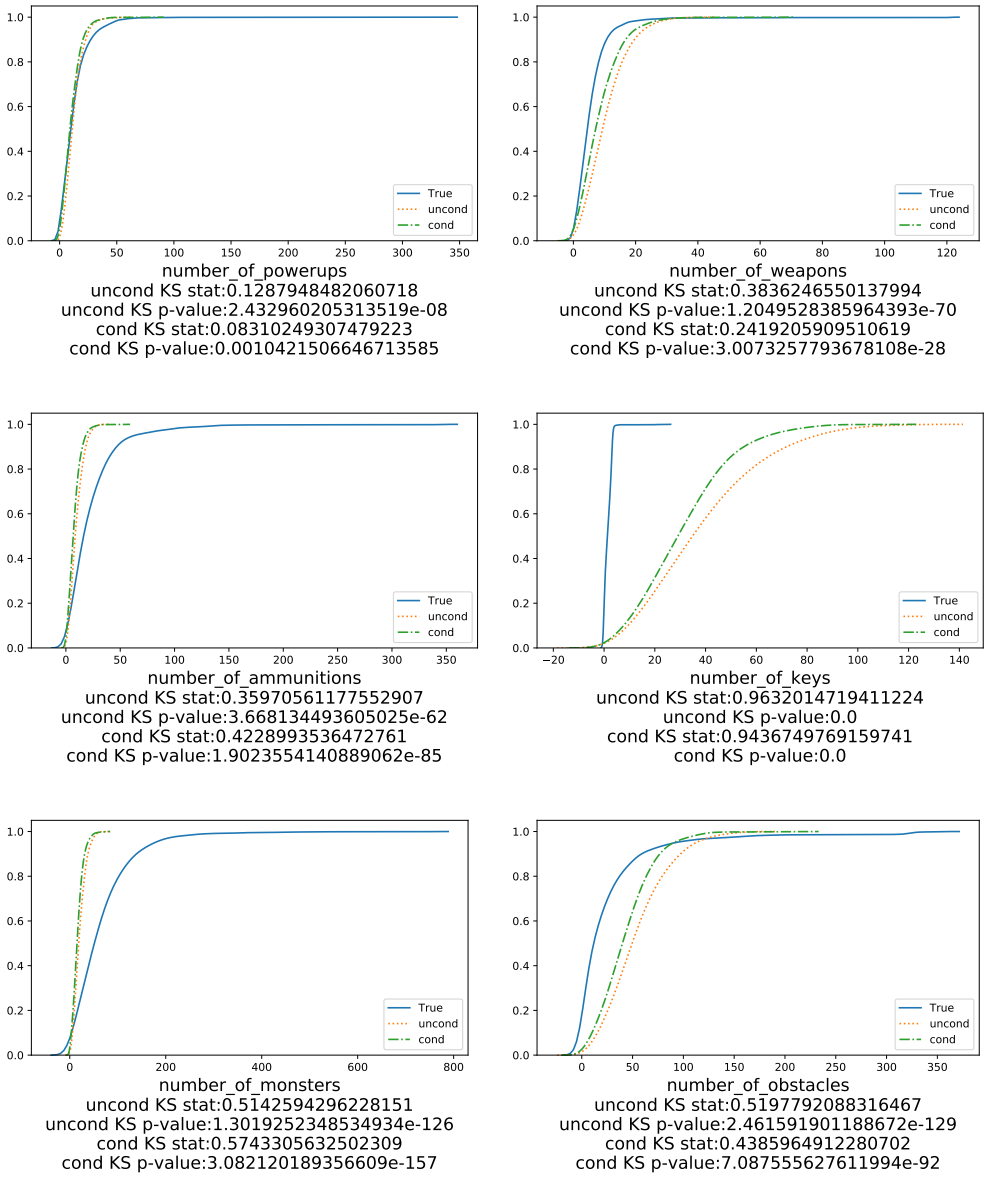

Figure B.4: Experiments 1 and 2: Cumulative distribution functions for true data, unconditional network and conditional network for each non-input feature.

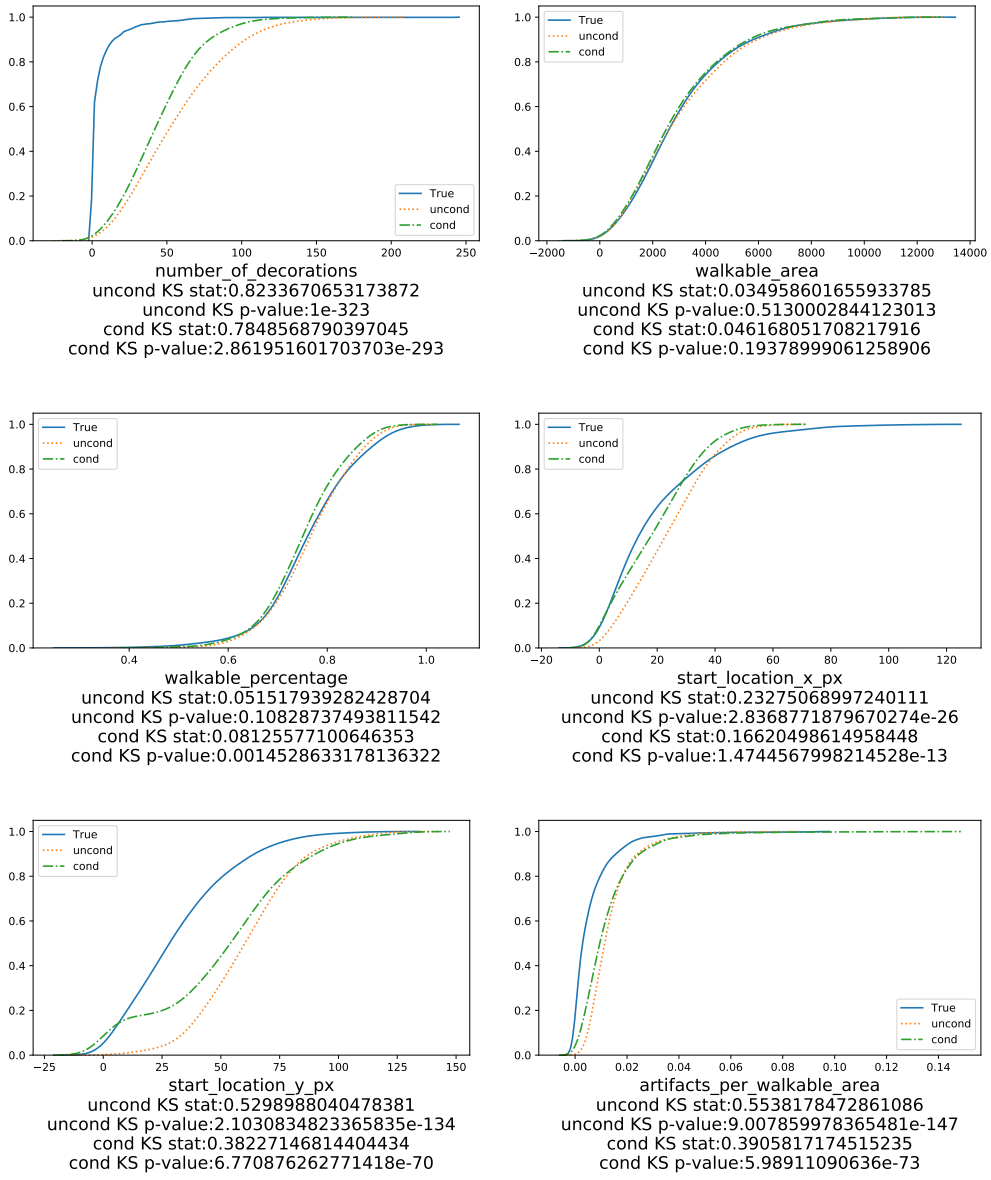

Figure B.5: Experiments 1 and 2: Cumulative distribution functions for true data, unconditional network and conditional network for each non-input feature.

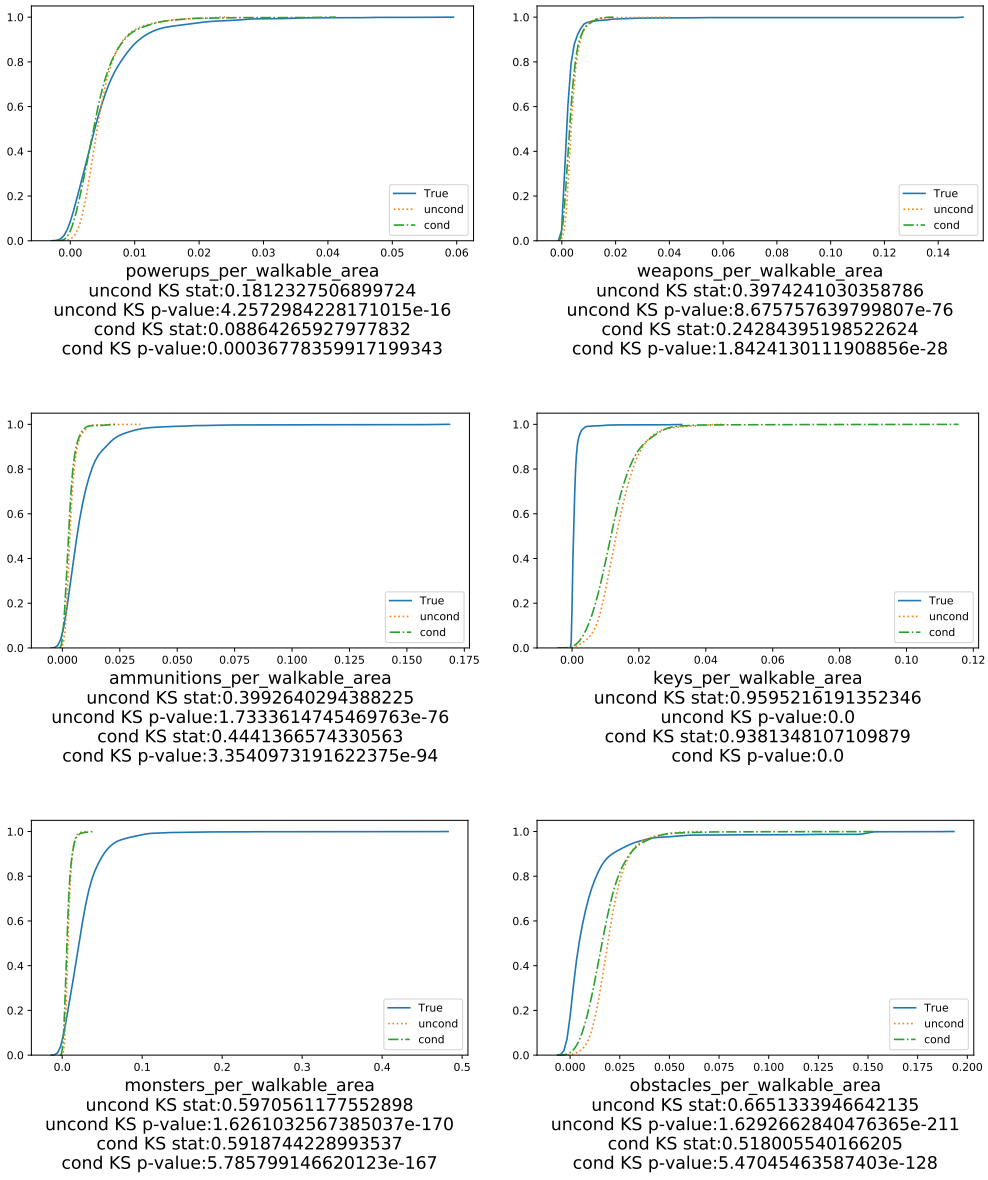

Figure B.6: Experiments 1 and 2: Cumulative distribution functions for true data, unconditional network and conditional network for each non-input feature.

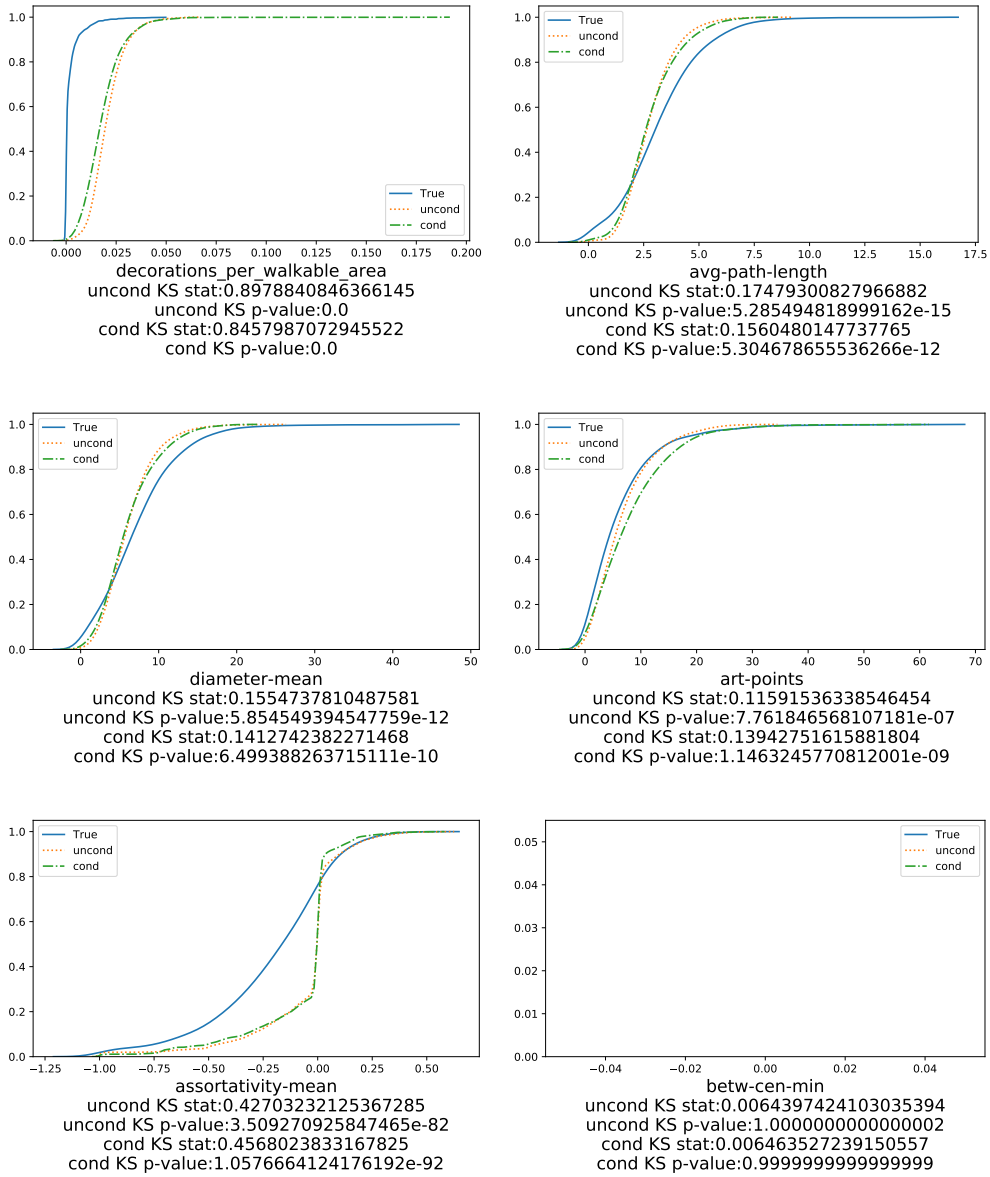

Figure B.7: Experiments 1 and 2: Cumulative distribution functions for true data, unconditional network and conditional network for each non-input feature.

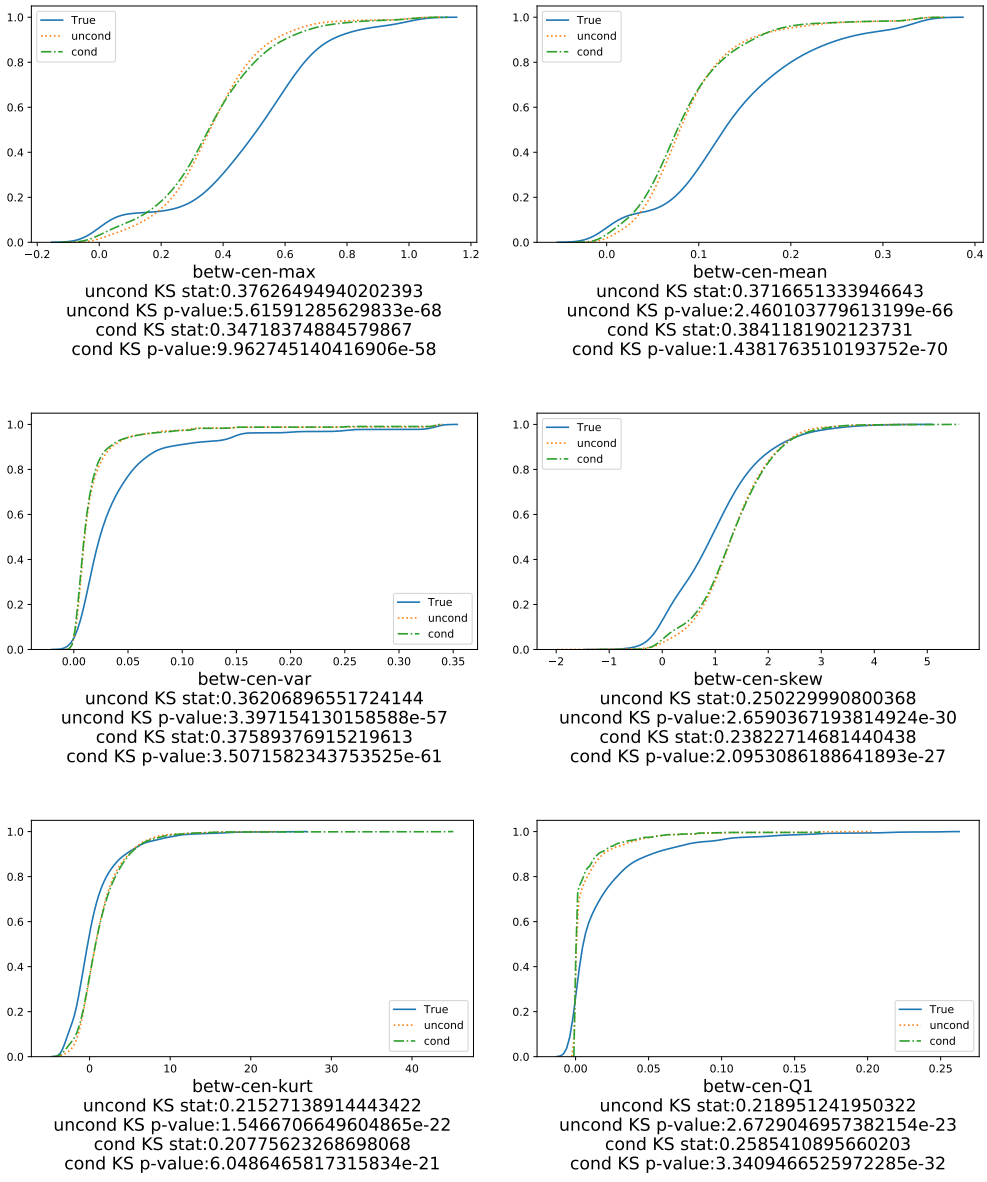

Figure B.8: Experiments 1 and 2: Cumulative distribution functions for true data, unconditional network and conditional network for each non-input feature.

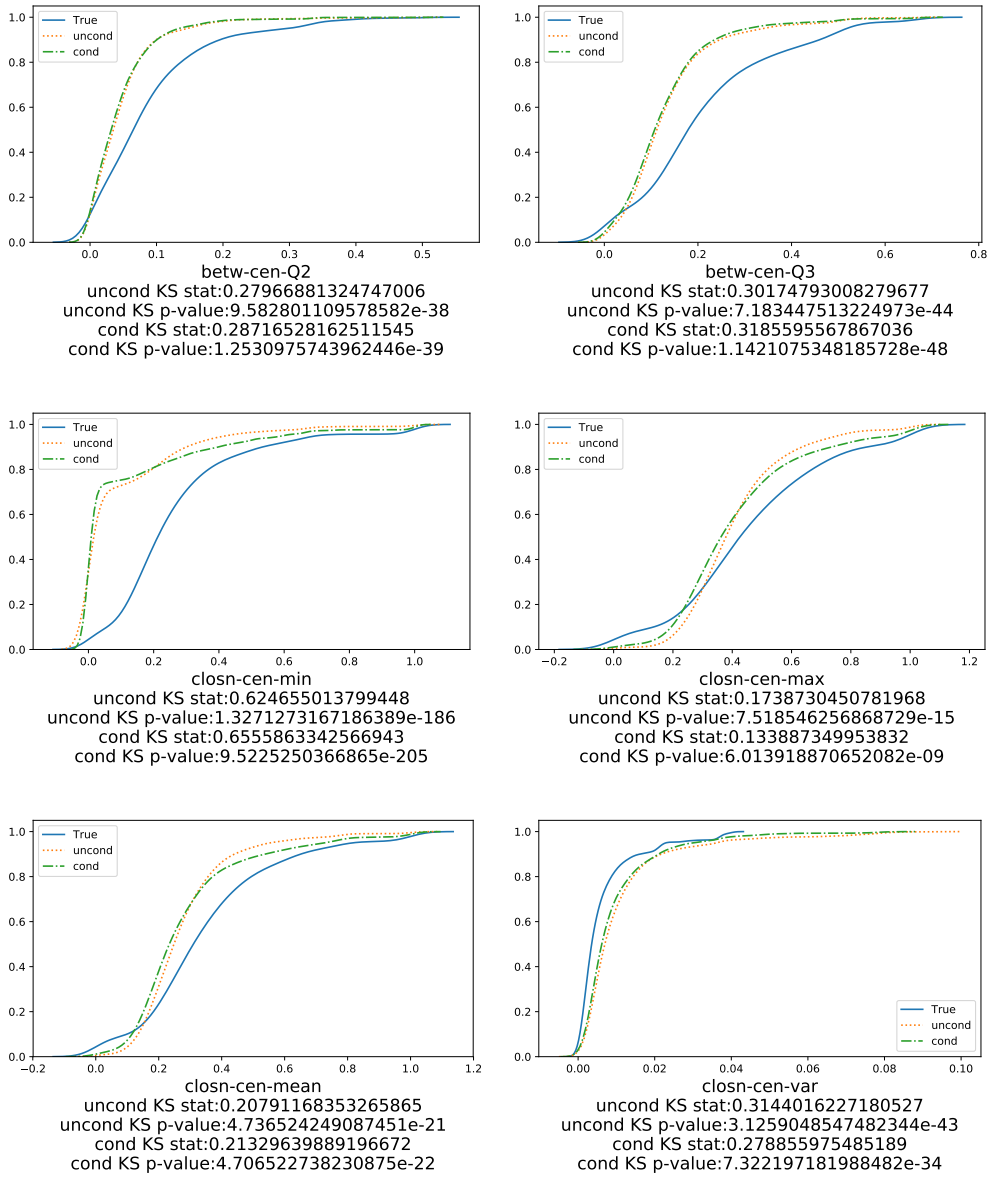

Figure B.9: Experiments 1 and 2: Cumulative distribution functions for true data, unconditional network and conditional network for each non-input feature.

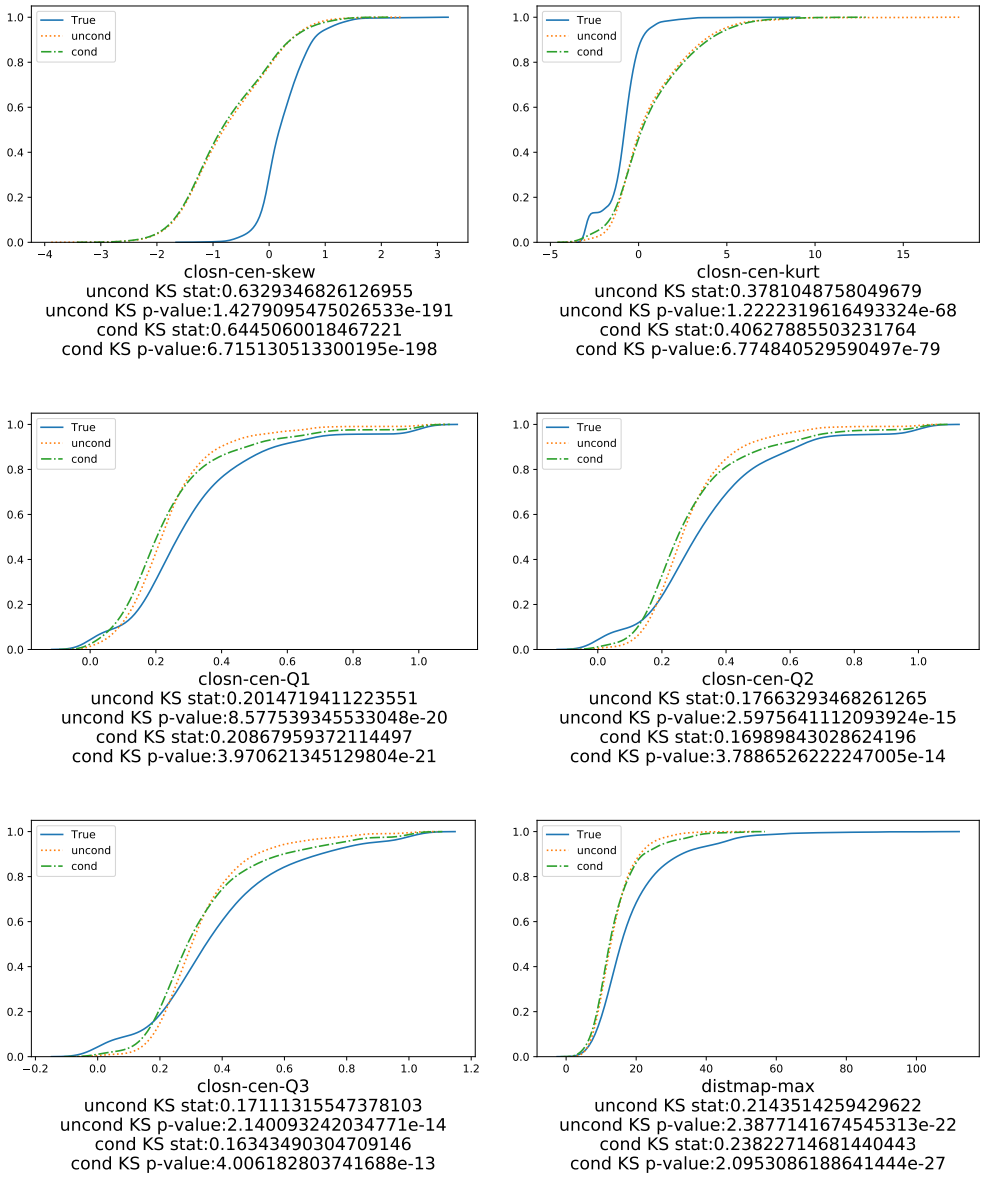

Figure B.10: Experiments 1 and 2: Cumulative distribution functions for true data, unconditional network and conditional network for each non-input feature.

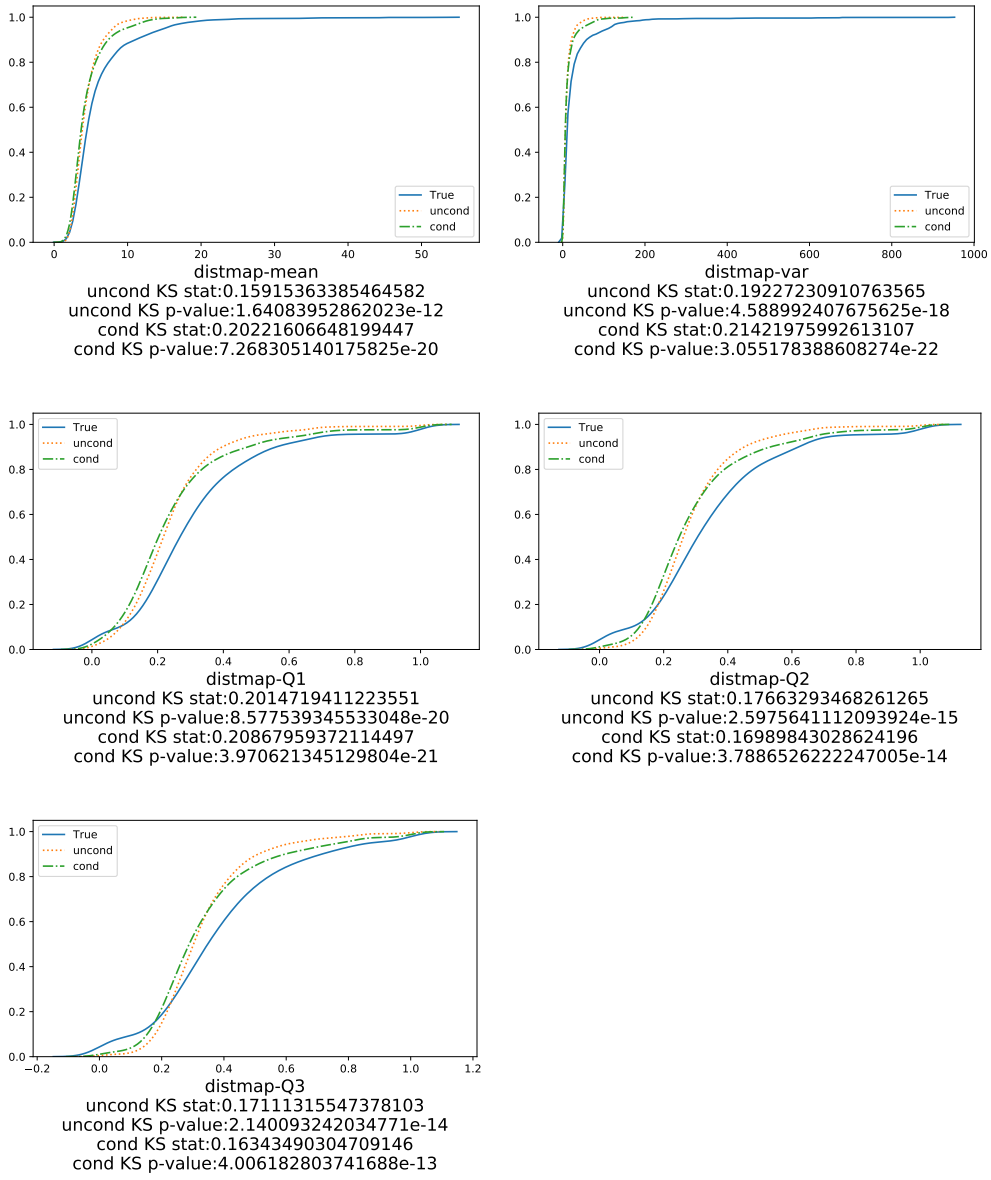

Figure B.11: Experiments 1 and 2: Cumulative distribution functions for true data, unconditional network and conditional network for each non-input feature.

Glossary

deathmatch Game mode in which two or more players compete against each other in achieving the highest number of kills before a timeout or a player reaches a predetermined score.. 109

DU Doom Units, see MU. 109

Feature Map Image describing a particular aspect of a level. 30, 33, 47, 51, 54, 56, 57, 109

floor Unconnected piece of level, reachable only by means of a teleporter. 109

FloorMap Image describing which parts of the map are occupied by a floor and which are empty.. 30–32, 52, 55–57, 109

GAN Generative Adversarial Network. 21–24, 47, 49, 52, 54, 109

HeightMap Image describing the floor height of a level. 31, 109

Linedef Line that connects two vertices in the WAD file specification. 28, 30, 109

Lump Any section of data within a WAD file.. 26, 28, 109

MU Map Units, Coordinate unit used in Doom Rendering Engine.. 27, 28, 30, 32–34, 109, 111

RoomMap Image describing the room segmentation of the level. 31, 57, 109

Sector A Sector in a DOOM level is any closed area (with possibly invisible walls) that has a constant floor and ceiling height and texture. plural. 27, 57, 109

Sidedef A "vertical plane" in the WAD file specification.. 28, 109

Thing Any deployable asset of a DOOM level, such as Enemies, Power-Ups, Weapons, Decorations, Spawners, Etc. plural. 27, 109

ThingsMap Image describing how the game objects are placed inside the level. 31, 109

TriggerMap Image describing doors, lifts and switches and their correlation. 31, 57, 109

WAD Default format of package files for the DOOM / DOOM II video-games. "WAD" is an acronym for "Where's all the Data?" [65]. 25–27, 30, 33, 109, 111

WallMap Image describing the impassable walls in a level. 30, 52, 56, 57, 109

Bibliography

[1] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[2] Tarn Adams. *Dwarf Fortress*. 2006.

[3] Martín Arjovsky, Soumith Chintala, and Léon Bottou. "Wasserstein GAN". In: *CoRR* abs/1701.07875 (2017). arXiv: 1701.07875. URL: http://arxiv.org/abs/1701.07875.

[4] Barry Bloom. *Barry Bloom post on OFC.UNT.EDU News*. URL: https://groups.google.com/d/msg/alt.games.doom/Wjp0gf-zBf8/YIYsTQyD9F4J.

[5] Christopher Beckham and Christopher Joseph Pal. "A step towards procedural terrain generation with GANs". In: *CoRR* abs/1707.03383 (2017). arXiv: 1707.03383. URL: http://arxiv.org/abs/1707.03383.

[6] David Berthelot, Tom Schumm, and Luke Metz. "BEGAN: Boundary Equilibrium Generative Adversarial Networks". In: *CoRR* abs/1703.10717 (2017). arXiv: 1703.10717. URL: http://arxiv.org/abs/1703.10717.

[7] Blizzard North Blizzard Entertainment. *Diablo*. 1996.

[8] R. Bormann et al. "Room segmentation: Survey, implementation, and analysis". In: *2016 IEEE International Conference on Robotics and Automation (ICRA)*. 2016, pp. 1019–1026. DOI: 10.1109/ICRA.2016.7487234.

[9] David Braben and Ian Bell. *Elite*. 1984.

[10] Colin Phipps and Simon Howard and Colin Reed and Lee Killough. *BSP v5.2*. [BSP - The Doom node builder software, accessed November 2017]. 1994 - 2006.

[11] Steve Dahlskog, Julian Togelius, and Mark J Nelson. "Linear levels through n-grams". In: *Proceedings of the 18th International Academic MindTrek Conference: Media Business, Management, Content & Services*. ACM. 2014, pp. 200–206.

[12] A.I. Design. *Rogue*. 1980.

[13] Frontier Developments. *Elite: Dangerous*. 2014.

[14] Doomworld on DoomWiki. *Doomworld - The Doom Wiki at DoomWiki.org*. [Online; accessed 06/02/2018]. 2005. URL: https://doomwiki.org/wiki/Doomworld.

[15] Doomworld.com Website. *Doomworld*. [Online; accessed 16/10/2017]. 1998. URL: https://www.doomworld.com/.

[16] Richard O. Duda and Peter E. Hart. "Use of the Hough Transformation to Detect Lines and Curves in Pictures". In: *Commun. ACM* 15.1 (Jan. 1972), pp. 11–15. ISSN: 0001-0782. DOI: 10.1145/361237.361242. URL: http://doi.acm.org/10.1145/361237.361242.

[17] Encyclopedia of Mathematics. *Orientation*. URL: http://www.encyclopediaofmath.org/index.php?title=Orientation&oldid=39859.

[18] *Encyclopedia of Optimization*. Kluwer, 2001. ISBN: 9780792369325. URL: https://books.google.com.mx/books?id=gtoTkL7heS0C.

[19] Blizzard Entertainment. *StarCraft II*. 2010.

[20] Matthew S Fell. *The Unofficial Doom Specs*. Online. 1994. URL: http://www.gamers.org/dhs/helpdocs/dmsp1666.html.

[21] Anton Filatov et al. "2D SLAM Quality Evaluation Methods". In: (Aug. 2017).

[22] Fuchs, Henry and Kedem, Zvi M. and Naylor, Bruce F. "On Visible Surface Generation by a Priori Tree Structures". In: *SIGGRAPH Comput. Graph.* 14.3 (July 1980), pp. 124–133. ISSN: 0097-8930. DOI: 10.1145/965105.807481. URL: http://doi.acm.org/10.1145/965105.807481.

[23] C. Galamhos, J. Matas, and J. Kittler. "Progressive probabilistic Hough transform for line detection". In: *Proceedings. 1999 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (Cat. No PR00149)*. Vol. 1. 1999, 560 Vol. 1. DOI: 10.1109/CVPR.1999.786993.

[24] Hello Games. *No Man's Sky*. 2016.

[25] Edoardo Giacomello. *DoomPCGML*. https://github.com/edoardogiacomello/DoomPCGML. 2017-2018.

[26] Ian J. Goodfellow et al. "Generative Adversarial Networks". In: *CoRR* abs/1406.2661 (2014). arXiv: 1406.2661. URL: http://arxiv.org/abs/1406.2661.

[27] Ishaan Gulrajani. *improved wgan training*. https://github.com/igul222/improved_wgan_training. 2017.

[28] Ishaan Gulrajani et al. "Improved Training of Wasserstein GANs". In: *CoRR* abs/1704.00028 (2017). arXiv: 1704.00028. URL: http://arxiv.org/abs/1704.00028.

[29] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. "Exploring network structure, dynamics, and function using NetworkX". In: *Proceedings of the 7th Python in Science Conference (SciPy2008)*. Pasadena, CA USA, Aug. 2008, pp. 11–15.

[30] Chris Harris and Mike Stephens. "A combined corner and edge detector". In: *In Proc. of Fourth Alvey Vision Conference*. 1988, pp. 147–151.

[31] Lewis Horsley and Diego Perez-Liebana. "Building an automatic sprite generator with deep convolutional generative adversarial networks". In: *Computational Intelligence and Games (CIG), 2017 IEEE Conference on*. IEEE. 2017, pp. 134–141.

[32] Gao Huang et al. *An empirical study on evaluation metrics of generative adversarial networks*. 2018. URL: https://openreview.net/forum?id=Sy1f0e-R-.

[33] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *CoRR* abs/1502.03167 (2015). arXiv: 1502.03167. URL: http://arxiv.org/abs/1502.03167.

[34] Phillip Isola et al. "Image-to-Image Translation with Conditional Adversarial Networks". In: *arxiv* (2016).

[35] Rishabh Jain et al. "Autoencoders for level generation, repair, and recognition". In: *Proceedings of the ICCC Workshop on Computational Creativity and Games*. 2016.

[36] John Carmack. *Doom Engine Source code on GitHub.com*. [Online; accessed 06/02/2018]. 1997. URL: https://github.com/id-Software/DOOM.

[37] Frank J. Massey Jr. "The Kolmogorov-Smirnov Test for Goodness of Fit". In: *Journal of the American Statistical Association* 46.253 (1951), pp. 68–78. DOI: 10.1080/01621459.1951.10500769. eprint: https://www.tandfonline.com/doi/pdf/10.1080/01621459.1951.10500769. URL: https://www.tandfonline.com/doi/abs/10.1080/01621459.1951.10500769.

[38] Tero Karras et al. "Progressive Growing of GANs for Improved Quality, Stability, and Variation". In: *CoRR* abs/1710.10196 (2017). arXiv: 1710.10196. URL: http://arxiv.org/abs/1710.10196.

[39] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *CoRR* abs/1412.6980 (2014). arXiv: 1412.6980. URL: http://arxiv.org/abs/1412.6980.

[40] Y. Lecun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. ISSN: 0018-9219. DOI: 10.1109/5.726791.

[41] Scott Lee et al. "Predicting Resource Locations in Game Maps Using Deep Convolutional Neural Networks". In: *The Twelfth Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment. AAAI*. 2016.

[42] J. Lin. "Divergence measures based on the Shannon entropy". In: *IEEE Transactions on Information Theory* 37.1 (1991), pp. 145–151. ISSN: 0018-9448. DOI: 10.1109/18.61115.

[43] Ziwei Liu et al. "Deep Learning Face Attributes in the Wild". In: *Proceedings of International Conference on Computer Vision (ICCV)*. 2015.

[44] Matteo Luperto and Francesco Amigoni. "Predicting the Global Structure of Indoor Environments: A Constructive Machine Learning Approach". unpublished article at the moment of writing. 2018.

[45] Michaël Mathieu, Camille Couprie, and Yann LeCun. "Deep multi-scale video prediction beyond mean square error". In: *CoRR* abs/1511.05440 (2015). arXiv: 1511.05440. URL: http://arxiv.org/abs/1511.05440.

[46] Mehdi Mirza and Simon Osindero. "Conditional Generative Adversarial Nets". In: *CoRR* abs/1411.1784 (2014). arXiv: 1411.1784. URL: http://arxiv.org/abs/1411.1784.

[47] Shigeru Miyamoto and Takashi Tezuka. *Super Mario Bros.* 1985.

[48] Shigeru Miyamoto and Takashi Tezuka. *The legend of Zelda*. 1986.

[49] Mojang. *Minecraft*. 2011.

[50] Vinod Nair and Geoffrey E. Hinton. "Rectified Linear Units Improve Restricted Boltzmann Machines". In: *Proceedings of the 27th International Conference on International Conference on Machine Learning*. ICML'10. Haifa, Israel: Omnipress, 2010, pp. 807–814. ISBN: 978-1-60558-907-7. URL: http://dl.acm.org/citation.cfm?id=3104322.3104425.

[51] P. Neubert and P. Protzel. "Compact Watershed and Preemptive SLIC: On Improving Trade-offs of Superpixel Segmentation Algorithms". In: *2014 22nd International Conference on Pattern Recognition*. 2014, pp. 996–1001. DOI: 10.1109/ICPR.2014.181.

[52] Augustus Odena, Vincent Dumoulin, and Chris Olah. "Deconvolution and Checkerboard Artifacts". In: *Distill* (2016). DOI: 10.23915/distill.00003. URL: http://distill.pub/2016/deconv-checkerboard.

[53] Alec Radford, Luke Metz, and Soumith Chintala. "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks". In: *CoRR* abs/1511.06434 (2015). arXiv: 1511.06434. URL: http://arxiv.org/abs/1511.06434.

[54] Tim Salimans et al. "Improved Techniques for Training GANs". In: *CoRR* abs/1606.03498 (2016). arXiv: 1606.03498. URL: http://arxiv.org/abs/1606.03498.

[55] scikit-image development team. *Scikit Image Documentation - peak_local_max Documentation*. URL: http://scikit-image.org/docs/0.12.x/api/skimage.feature.html#skimage.feature.peak_local_max.

[56] Noor Shaker, Julian Togelius, and Mark J Nelson. *Procedural content generation in games*. Springer, 2016.

[57] Sam Snodgrass and Santiago Ontañón. "Experiments in map generation using Markov chains." In: *FDG*. 2014.

[58] Adam Summerville et al. "Procedural Content Generation via Machine Learning (PCGML)". In: *CoRR* abs/1702.00539 (2017). arXiv: 1702.00539. URL: http://arxiv.org/abs/1702.00539.

[59] Adam James Summerville and Michael Mateas. "Sampling hyrule: Multi-technique probabilistic level generation for action role playing games". In: *Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference*. 2015.

[60] Adam James Summerville et al. "The VGLC: The Video Game Level Corpus". In: *Proceedings of the 7th Workshop on Procedural Content Generation* (2016).

[61] Christian Szegedy et al. "Going Deeper with Convolutions". In: *CoRR* abs/1409.4842 (2014). arXiv: 1409.4842. URL: http://arxiv.org/abs/1409.4842.

[62] The Scipy community. *SciPy Documentation - distance_transform_edt*. 2008-2009. URL: https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.ndimage.morphology.distance_transform_edt.html.

[63] Thing Types on DoomWiki. *Thing Types - The Doom Wiki at DoomWiki.org*. [Online; accessed November 2017]. 2005. URL: http://doom.wikia.com/wiki/Thing_types.

[64] Julian Togelius et al. "Search-Based Procedural Content Generation". In: *Applications of Evolutionary Computation*. Ed. by Cecilia Di Chio et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 141–150. ISBN: 978-3-642-12239-2.

[65] Tom Hall. *DOOM Bible, Appendix A: Glossary*. [Online; Doomworld Mirror, accessed 06/02/2018]. 1992. URL: http://5years.doomworld.com/doombible/appendices.shtml.

[66] Alain Trmeau and Philippe Colantoni. "Regions Adjacency Graph Applied To Color Image Segmentation". In: *IEEE Transactions on Image Processing* 9 (2000), pp. 735–744.

[67] Zhou Wang et al. "Image quality assessment: from error visibility to structural similarity". In: *IEEE Transactions on Image Processing* 13.4 (2004), pp. 600–612. ISSN: 1057-7149. DOI: 10.1109/TIP.2003.819861.

[68] Tom White. "Sampling Generative Networks: Notes on a Few Effective Techniques". In: *CoRR* abs/1609.04468 (2016). arXiv: 1609.04468. URL: http://arxiv.org/abs/1609.04468.

[69] Li-Chia Yang, Szu-Yu Chou, and Yi-Hsuan Yang. "MidiNet: A Convolutional Generative Adversarial Network for Symbolic-domain Music Generation using 1D and 2D Conditions". In: *CoRR* abs/1703.10847 (2017). arXiv: 1703.10847. URL: http://arxiv.org/abs/1703.10847.

[70] Fisher Yu et al. "LSUN: Construction of a Large-scale Image Dataset using Deep Learning with Humans in the Loop". In: *arXiv preprint arXiv:1506.03365* (2015).

[71] Han Zhang et al. "StackGAN: Text to Photo-realistic Image Synthesis with Stacked Generative Adversarial Networks". In: *CoRR* abs/1612.03242 (2016). arXiv: 1612.03242. URL: http://arxiv.org/abs/1612.03242.