First Person Shooter level generation using Generative Adversarial Networks

Edoardo Giacomello

20 febbraio 2018

Abstract

Estratto in lingua Italiana

To someone...

Acknowledgments

Contents

Abstract **3**

Estratto in lingua Italiana **5**

Acknowledgments **9**

1 Introduction **17**
1.1 Background: Level Design . . . 17
1.2 Related Work . . . 17
1.2.1 Procedurally Generated Content . . . 17
1.2.2 Procedural Content Generation via Machine Learning (PCGML) . . . 17
1.2.3 Video Game Level Corpus . . . 17
1.2.4 Generative Adversarial Networks . . . 17
1.3 Scope . . . 17
1.4 Thesis Structure . . . 17
1.5 Summary . . . 17

2 Towards learn-based level generation **19**
2.1 Generative Adversarial Networks . . . 19
2.1.1 Overview . . . 19
2.1.2 Deep Convolutional GAN . . . 20
2.1.3 Wesserstein GAN . . . 20
2.1.4 Wesserstein GAN with Gradient Penalty . . . 20
2.2 Game of choice: DOOM . . . 20
2.2.1 Description . . . 20
2.2.2 Motivation . . . 20
2.3 General Considerations . . . 20
2.4 Summary . . . 20

3 Dataset and Data Representation **21**
3.1 Data Sources . . . 21
3.2 Source Data Format: WAD Files . . . 22
3.2.1 Overview . . . 22
3.2.2 Doom Level Format . . . 23
3.2.3 Conversion issues . . . 24
3.3 Target Data Format: Feature Maps and Vectors . . . 26
3.3.1 Level Description and Motivation . . . 26
3.3.2 Feature Maps . . . 26
3.3.3 Graph Representation . . . 27
3.3.4 Text Representation . . . 28
3.3.5 Scalar Features . . . 28
3.4 Dataset Organization . . . 28
3.4.1 Overview . . . 28

3.4.2 Full Dataset and Filtered Dataset . . . 28
3.4.3 Level Size Statistics . . . 30
3.5 Summary . . . 31

4 System Design and Overview **43**
4.1 Generative Model Structure . . . 43
4.2 Use Cases . . . 44
4.2.1 Use Case: Model Optimization (Training) . . . 45
4.2.2 Use Case: Model Validation and Sample Evaluation . . . 45
4.2.3 Use Case: Sampling or Generation . . . 46
4.3 Data Flow . . . 47
4.4 WAD Editor and Feature Extractor . . . 49
4.4.1 Reading and Writing . . . 49
4.4.2 Feature Extraction . . . 49
4.5 Summary . . . 50

5 Experiment Design and Results **51**
5.1 Neural Network Architecture and Training Algorithm . . . 51
5.1.1 Network Architecture . . . 51
5.2 Input selection, Metrics and Sample Method . . . 53
5.2.1 Input data . . . 53
5.2.2 Evaluation metrics . . . 53
5.2.3 Sampling the network . . . 55
5.3 Results . . . 55
5.3.1 Resulting Model . . . 55
5.4 Generated Samples . . . 55
5.5 In-Game Demonstration . . . 55
5.6 Summary . . . 55

6 Results Evaluation and Conclusions **57**
6.1 Results Evaluation . . . 57
6.1.1 Samples Evaluation . . . 57
6.1.2 Loss of accuracy . . . 57
6.2 Summary . . . 57

7 Future Work **59**
7.1 Open Problems . . . 59
7.2 Possible Applications and future develops . . . 59

8 Appendix **61**

Glossary **65**

List of Figures

3.1 A simple level showing sectors and linedefs 25
3.2 DoomDataset Size distribution 30

4.1 System Overview: Generative Model Structure 44
4.2 Use Case: Model Optimization 45
4.3 Use Case: Validation and Sample Evaluation 46
4.4 Use Case: Sampling or Generation 47
4.5 System Overview: Data Flow Diagram 48

List of Tables

3.1 WAD File structure 32
3.2 ThingsMap Encoding (1 of 3) 33
3.3 ThingsMap Encoding (2 of 3) 34
3.4 ThingsMap Encoding (3 of 3) 35
3.5 TriggerMap Encoding 36
3.6 Textual Representation Encoding 36
3.7 Features: IDArchive Metadata 37
3.8 Features: WAD-extracted 38
3.9 Features: PNG-extracted (1 of 2) 39
3.10 Features: PNG-extracted (2 of 2) 40
3.11 Features: Graph 41
3.12 Percentiles of level width and height distributions 41

5.1 Training Algorithm Operations 53

Chapter 1

Introduction

1.1 Background: Level Design

1.2 Related Work

1.2.1 Procedurally Generated Content

1.2.2 Procedural Content Generation via Machine Learning (PCGML)

1.2.3 Video Game Level Corpus

1.2.4 Generative Adversarial Networks

1.3 Scope

1.4 Thesis Structure

1.5 Summary

Chapter 2

Towards learn-based level generation

Overview The scope of this chapter is to further describe the setting of our work and highlight the main differences between the work that have been currently done in the literature and the one we introduce in the following chapters. In particular, section 2.1 first provides the theoretical background necessary to understand the setting in which our work take place. Then, section 2.2 describes and motivates the choice of the game we work with. Lastly, section 2.3 reveals some important considerations about the setting just described and the possible difficulties in this type of research.

2.1 Generative Adversarial Networks

2.1.1 Overview

GAN A model that in the latest years gained increasingly more interest in the field of Machine Learning research is the one of the Generative Adversarial Network, proposed by Goodfellow et al. in "Generative Adversarial Networks" (2014) [16]. 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 [10, p. 276]. In particular, a generative network called 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.

Conditional GAN In our work we make use of a modification of the GAN model introduced by Mirza and Osindero [24], which allows to condition the data generation to class labels. This structure is summarized in section 4.1 where the logical structure of the generative model we adopt is presented.

Additional Architectures Due to the amount of interest that GANs have raised, many researches have been conducted to improve the quality performances of GANs, leading to a variety of proposed new architectures. They can be classified as modifications to the underlying neural network architectures or even to the conceptual 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 5.1.1.

2.1.2 Deep Convolutional GAN

2.1.3 Wesserstein GAN

2.1.4 Wesserstein GAN with Gradient Penalty

2.2 Game of choice: DOOM

2.2.1 Description

2.2.2 Motivation

2.3 General Considerations

2.4 Summary

Chapter 3

Dataset and Data Representation

Overview This chapter aims to be a description of the dataset structure the model is trained and evaluated with, as well as an overview of the processes that led to the creation of the dataset themselves. In section 3.1 a reference to the data sources is given, then the focus of section 3.2 is on how data is natively encoded for the game engine in order to give some hints on what are the difficulties to face in converting to and from that format in an automatic way. Section 3.3 describes in detail what data is provided with the dataset, how levels are converted from the native format and what features are extracted in order to provide an input for the neural network. Lastly, section 3.4 gives an overview of how the data is provided and organized in the resulting dataset.

3.1 Data Sources

Archive All data used to train and validate the model comes from the *Idgames Archive* founded in 1994 by Barry Bloom [3] and mirrored on various FTP sources. The mirror we used for collecting levels is Doomworld.com [7], which is one of the oldest and currently most active community about the DOOM video-game series [6].

Level Selection 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 all 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 possibly different kinds 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 levels contained in it are intended to work with modifications of the game engine code, and it would have led to problems in managing every particular exceptional behaviour.

Source Data Organization 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 [6], data may not always be accurate, especially that concerning download and view counts; they are still proposed as a starting point for further research.

3.2 Source Data Format: WAD Files

Introduction The Doom Game Engine [21] 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* [11] 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

Overview 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.

Essential Subset and Optimizations 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 [13] 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*" [5] 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 [9], 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.

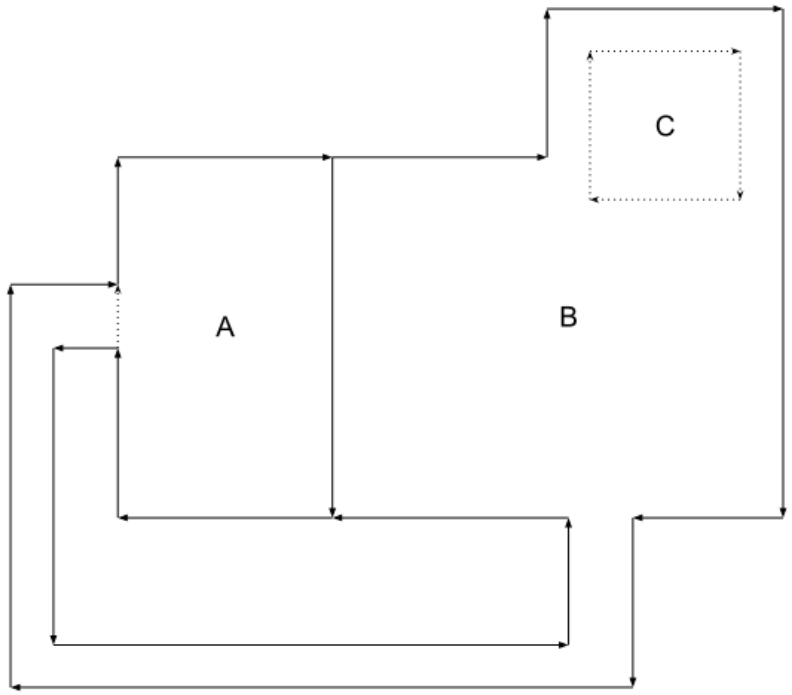

Figure 3.1: A simple level showing three sectors A, B and C and the linedefs defining them, following the positive oriented curve constraint. Sector C can be viewed as a small platform inside the sector B. Solid arrows represent walls, while dashed lines represent invisible linedefs or changes in height 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.

3.3 Target Data Format: Feature Maps and Vectors

Introduction 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 taken from [28], 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 work on. In particular, convolutional neural netoworks are naturally 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 [28], 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 [31].

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"[4]: An euclidean distance transform [30] is first applied to the FloorMap obtaining a map that we call Distance Map or DistMap; then the local maxima are found [27] 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 segementation [25] 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 [33] 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 [28]. 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 [23]. 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

3.4 Dataset Organization

3.4.1 Overview

This section will describe 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 5.2.1 in our work we only used a subset of levels stored as a separated archive.

3.4.2 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 [19] 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 5.2.1, 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 [15] 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.

Provided Methods The code repository [15] 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.

3.4.3 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

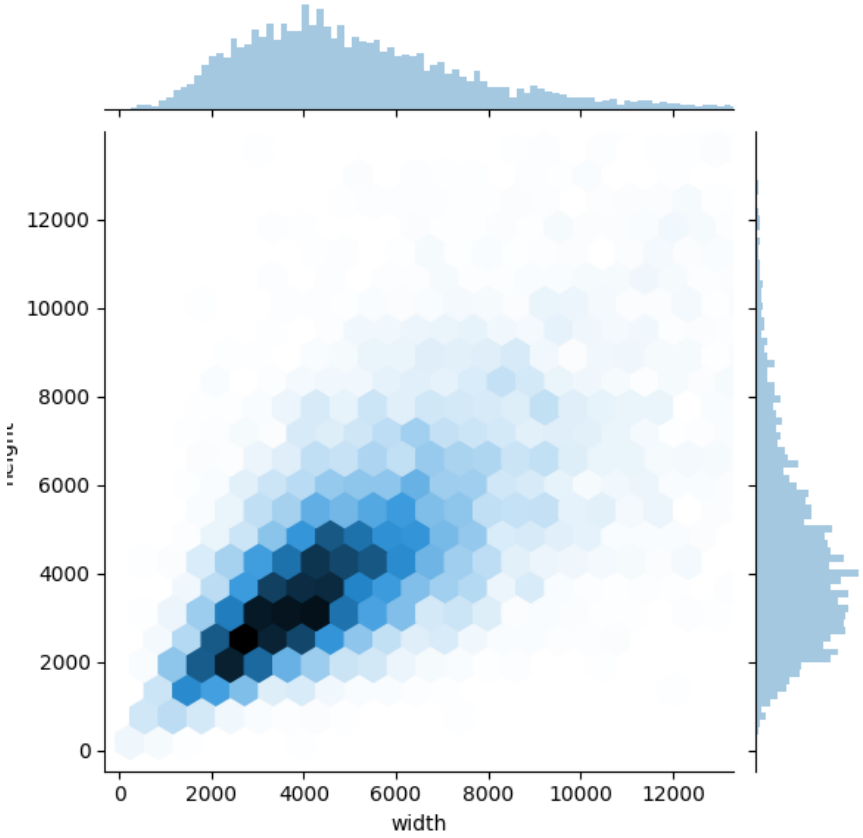

Figure 3.2: Joint distribution for the features "width" (on the x axis) and "height" (on the y axis) expressed in MU for the level contained in the full dataset. It is possible to notice how the size of the majority of the levels are below 7000 Map Units

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 the 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.

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

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

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

Chapter 4

System Design and Overview

Overview This chapter describes the proposed system from an high level perspective. The purpose of this chapter is indeed to give an overview of how the system modules interact from the point of view of the use cases and data flow analysis. 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 for 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 whole system design focusing on processes and data transformation rather then a component view of the system.

4.1 Generative Model Structure

Overview The initial system design was built upon the architecture of Generative Adversarial Networks [16] 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 [24] of the GAN Model, proposed by Mirza and Osindero, applied to a more recent GAN model that is discussed in section 5.1.1.

Conditional GAN Structure We present in figure 4.1 the general architecture 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 working principles of a GAN: It is composed by two neural networks, namely a Generator and a Discriminator (or Critic, depending on the underlying architecture that is adopted).
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, we have that:

- $X_{Gen} = G(Z|Y)$: Samples generated from the generator network
- $Logits(X_{Gen}) = D(X_{Gen}|Y)$: Discriminator output when real samples are input to the discriminator.

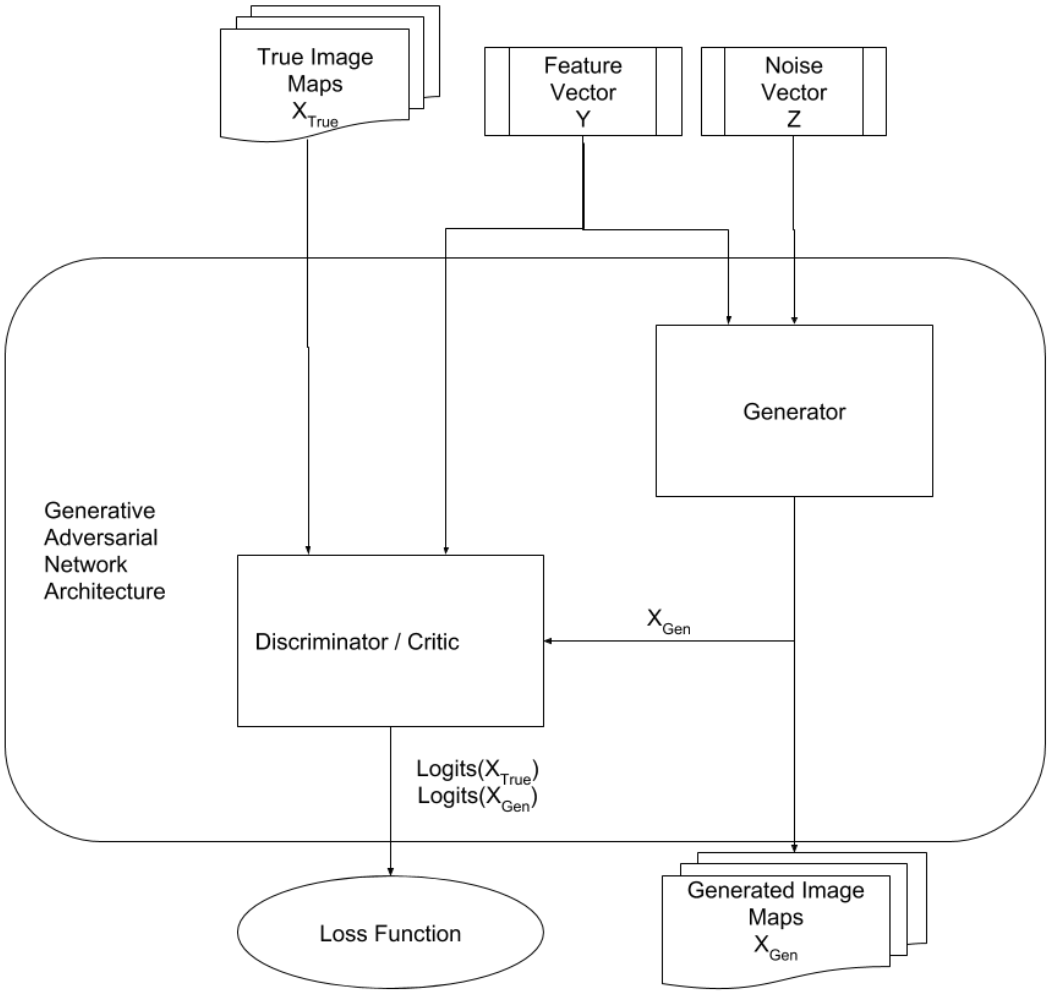

Figure 4.1: System Overview: Generative Model Structure. A Conditional GAN is composed of a generator model and a discriminative model. The discriminative model takes as input either the Images coming from the dataset or the ones generated by the generative model. Both networks are conditioned by the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

- $Logits(X_{True}) = D(X_{True}|Y)$: 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. All the details are given when we'll describe the chosen GAN Architecture and the training process in section 5.1.1.

4.2 Use Cases

Overview 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 3.4.2 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 random 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.

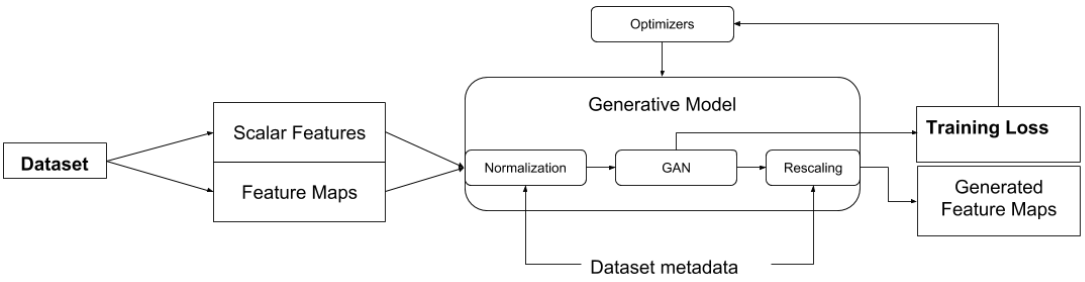

Figure 4.2: Use Case: Model Optimization.

4.2.2 Use Case: Model Validation and Sample Evaluation

This use case is used to assess the ability of the model to generate new samples on previously unseen feature vectors. 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 5.1.1 we select one of the architectures which propose to reduce the severity of this problem, among others.

2. A set of *quality metrics* (5.2.2) 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 grade 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 of more generic metrics are chosen and presented in section (5.2.2). 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.

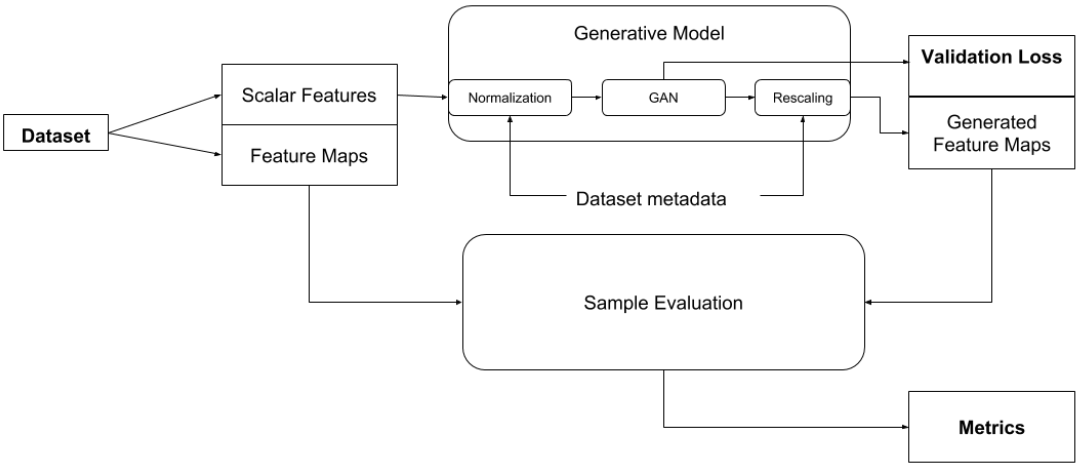

Figure 4.3: Use Case: Validation and Sample Evaluation.

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.
These different sampling methods are motivated in section 5.3.1, while in this section we only highlight the differences from an input/output point of view. The main problem of sampling this network, as highlighted in the work of White[35], 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 are given in section 5.3.1. 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.

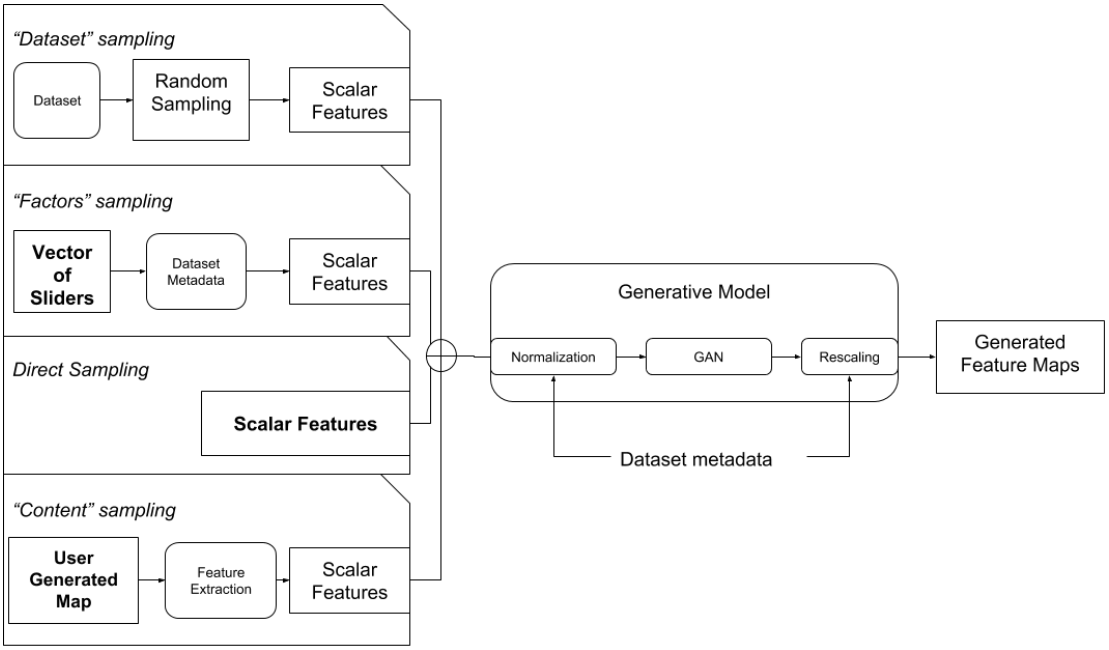

Figure 4.4: Use Case: Sampling or Generation.

4.3 Data Flow

Figure 4.5 represent 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 Model Validation and 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. 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

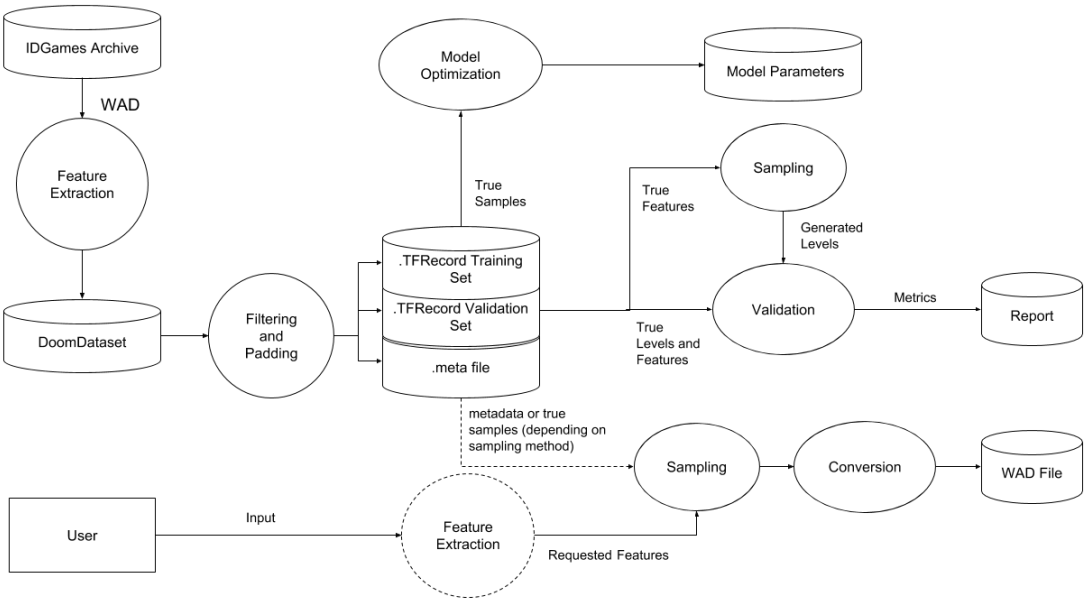

Figure 4.5: System Overview: Data Flow Diagram. Disc-shaped blocks represent data archives, circles represent processes and transformations, while the labels on arrows represents intermediate data. Dashed lines represents objects which presence depends on the use case of reference.

this reason they are indicated with dashed lines.

4.4 WAD Editor and Feature Extractor

Overview 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.

4.4.1 Reading and Writing

Data Collection 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.

Editor 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 or 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.

4.4.2 Feature Extraction

Overview 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.

WAD to Feature Maps 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.

Feature Maps to WAD 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, instead, would be using this editor to write the levels generated from the network back to a WAD file, and inspect 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 [8] or its probabilistic version [14] 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 instead, 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 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.

4.5 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. In order to cope with the difficulties in sampling the network and evaluating the samples, we presented alternative methods for conditioning the network during the generation phase and for evaluating the generated samples. Moreover, we introduced our module for converting WAD levels into a set of Feature Maps and vice-versa.

Chapter 5

Experiment Design and Results

Overview This chapter describes the experiment that we conducted based on the system design showed in chapter 4 and our final results, consisting in the trained model and its evaluation. In particular, in section 5.1 we present the selected architecture among the different possibilities in recent literature. Section 5.2 describes all the necessary details about the input selection, the chosen evaluation metrics and how results are produced. Section 5.3 presents the resulting model, the metrics associated to it and some examples of generated levels.

5.1 Neural Network Architecture and Training Algorithm

Overview Typically, training a Generative Neural Network is a difficult task. This is, among other reasons, due to the need of balancing the two networks, the generator and the discriminator, during the training process. Another issue is that the loss function is not always correlated with the generated sample quality, added to the problem of evaluating the sample quality. Although we tried some different GANs implementations and techniques, our principal selection criterion has been the effectiveness on our particular dataset of the proposed solution in solving this stability problems.

5.1.1 Network Architecture

Chosen Architecture Among the various GANs implementations that are proposed in the literature, we selected the Wasserstein GAN with Gradient Penalty [18] (WGAN-GP) from Gulrajani et al. This architecture is proposed as an enhanced version of the WGAN [2]. In the WGAN architecture, the discriminator is more properly called "critic", as it is a network that is not trained to discriminate true and generated samples but rather to evaluate them [18, section 2.2, p. 2].

Critic Loss and Generator Loss

Loss Formulation We implement the Critic and Generator losses as in the "Improved Training of Wasserstein GANs" official implementation [17]. Referencing to the notation introduced in 4.1 the losses are defined as follow:

$$\begin{aligned} L^{(i)}_{Critic} &\leftarrow \underbrace{\mathrm{E}(Logits(X_{Gen})) - \mathrm{E}(Logits(X_{True}))}_{\text{WGAN Loss}} + \underbrace{\lambda G_p}_{\text{Gradient Penalty}} \\ L^{(i)}_{Gen} &\leftarrow -Logits(X_{Gen}) \end{aligned} \tag{5.1}$$

where X_{True} and X_{Gen} are batches of levels sampled respectively from the dataset and the generator network and $\lambda = 10$ in our experiments. The difference of the first two addends in eq. 5.1, previously introduced in section 4.1, is an approximation to the Wasserstein Loss [2,

§ 3]. This distance is also called *Earth-Mover Distance* due to its informal interpretation: if two distribution are viewed as two piles of "dirt", then the Wasserstein distance is the minimum cost of turning one pile into the other.
Now that we presented the WGAN-GP architecture we can give an intuitive interpretation of these addends: The critic network is trained, by minimizing 5.1, to assign a "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 logits are not probabilities but unbounded score values.

Gradient Penalty The "Wasserstein GAN" paper uses a technique called "weight clipping" to enforce the *Lipschitz constraint* needed to ensure that the Wasserstein distance is a continuous loss function with respect to the network weights [2, §2]. In order to avoid optimization problems due to the weight clipping used in the original WGAN architecture, "Improved Training of Wasserstein GANs"[18] propose an alternative method called "Gradient Penalty". The gradient of the loss function is kept to unitary norm via a penalty that is added to the loss itself, resulting in:

$$\begin{aligned} \hat{X} &\leftarrow \epsilon X_{True} + (1-\epsilon) X_{Gen} \\ G_p &\leftarrow (\|\nabla_{\hat{X}} Logits(\hat{X})\|_2 - 1)^2 \end{aligned} \tag{5.2}$$

With $\epsilon \sim U[0,1]$. This way, the constraint is enforced only along straight lines between the real and the generated data distribution leading to good results and performances. For more details on the concepts introduced in this section we refer to the original papers [2, 18].

Training Algorithm and Hyper-Parameters

For implementing the training algorithm we followed the algorithm suggested by [18, alg. 1. p. 4], which uses *Adam*[22] 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 also imposed an input rotation of 90 for each sample at each epoch, such that each 4 epochs 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 playability is not affected by its orientation in the space it is represented. Table 5.1 shows in detail the frequency of each computation operation relative to each critic step, with reference to our TensorFlow implementation at [15]. 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 visualizing how the network weights optimization visually impact on the generation of one sample.
The hyperparameters we used in our experiments are $\alpha = 0.0002, \beta_1 = 0, \beta_2, \lambda = 10$.

Network Layers

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

Run Name	Inputs			Outputs	Frequency	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 5.1: Training Algorithm Operations

Motivation

Other details

5.2 Input selection, Metrics and Sample Method

5.2.1 Input data

5.2.2 Evaluation metrics

Overview As discussed in section 2.3, the problem of how evaluating the quality of the samples generated from a neural network remains an open area of research. The architectures that we considered up to now are primarily trained on datasets consisting of images representing real objects, such as faces or bedrooms. For dealing with the problem of evaluating the samples, Salimans et al. propose both a process in which human annotators are asked to assess the perceived quality of the samples [26, p. 4] and the usage of the Inception Model [29] Score for assessing the perceived quality of the samples. Although many authors have success in assessing sample quality using the Inception Score, this method showed poor results 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 (for example using transfer-learning techniques) or applying other proposed solutions based on similarly complex models wouldn't have been possible given our computational and time constraints, we decided to design some heuristics that could correlate with sample quality at least with our dataset.

Principles Instead of asking a neural network to evaluate the generated samples, our process compares two samples that have been generated by the network using 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 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 ap-

proach consist in defining three metrics that capture different aspects of the analysed maps, which features 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 quantitative 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:

$$S(x) = -\sum_{c=0}^{255} P_c(x) * \log P_c(x) \tag{5.3}$$

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:

$$Entropy_{mae}(X_{true}, X_{gen}) = \frac{1}{N} \sum_{i=0}^{N-1} |S(X_{gen}(i)) - S(X_{true}(i))| \tag{5.4}$$

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 very noisy or the topology of the two levels are greatly different, while small values indicates that the entropy of the two images are on a comparable level.

Mean Structural Similarity Index

This metric is defined as the Structural Similarity (SSIM) Index [34] 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 statistic. Moreover, the structural similarity technique is applied locally over the image, for reflecting the fact that pixels are more dependant on other spatially close pixels than far ones.
For calculating this metric we use the implementation provided by the Scikit-Image Python library, which we leave the formulation to the paper [34, 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, values closer to 1 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 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:

$$Enc_I(x) \equiv \sum_{k \in Z} \Lambda(\frac{2(x - kI) - I}{I}) \tag{5.5}$$

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

$$MEE_(X_{gen}) = \frac{1}{N} \sum_{i=0}^{N-1} \frac{1}{|P|} \sum_{p \in P} Enc_I(p) \tag{5.6}$$

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 [12]. We define our implementation of the Corner Error of two images as:

$$C_{err}(n_x, n_y) = \sqrt{\frac{(n_x - n_y)^2}{n_x n_y}} \tag{5.7}$$

where n_x and n_y are respectively the corner count of two binary images, extracted using the Harris corner detector [20]. 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:

$$MCE(X_{true}, X_{gen}) = \frac{1}{N} \sum_{i=0}^{N-1} C_{err}(n_{true,i}, n_{gen,i}) \tag{5.8}$$

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.

5.2.3 Sampling the network

5.3 Results

5.3.1 Resulting Model

5.4 Generated Samples

5.5 In-Game Demonstration

5.6 Summary

Chapter 6

Results Evaluation and Conclusions

6.1 Results Evaluation

6.1.1 Samples Evaluation

6.1.2 Loss of accuracy

6.2 Summary

Chapter 7

Future Work

7.1 Open Problems

7.2 Possible Applications and future develops

Chapter 8

Appendix

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.. 63

DU Doom Units, see MU. 63

Feature Map Image describing a particular aspect of a level. 26, 29, 43, 47, 49, 50, 53, 63

floor Unconnected piece of level, reachable only by means of a teleporter. 63

FloorMap Image describing which parts of the map are occupied by a floor and which are empty.. 26–28, 49, 54, 55, 63

GAN Generative Adversarial Network. 19, 43, 45, 51, 54, 63

HeightMap Image describing the floor height of a level. 27, 63

Linedef Line that connects two vertices in the WAD file specification. 24, 26, 63

Lump Any section of data within a WAD file.. 22, 24, 63

MU Map Units, Coordinate unit used in Doom Rendering Engine.. 23, 24, 26, 28–30, 63, 65

RoomMap Image describing the room segmentation of the level. 27, 49, 63

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. 23, 49, 63

Sidedef A "vertical plane" in the WAD file specification.. 24, 63

Thing Any deployable asset of a DOOM level, such as Enemies, Power-Ups, Weapons, Decorations, Spawners, Etc. plural. 23, 63

ThingsMap Image describing how the game objects are placed inside the level. 27, 63

TriggerMap Image describing doors, lifts and switches and their correlation. 27, 49, 63

WAD Default format of package files for the DOOM / DOOM II video-games. "WAD" is an acronym for "Where's all the Data?" [32]. 21–23, 26, 29, 63, 65

WallMap Image describing the impassable walls in a level. 26, 49, 55, 63

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] 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.

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

[4] 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.

[5] 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.

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

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

[8] 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.

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

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

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

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

[13] 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.

[14] 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.

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

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

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

[18] 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.

[19] 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.

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

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

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

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

[30] 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.

[31] 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.

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

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

[34] 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.

[35] 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.