SMWLevelGenerator
Generating Super Mario World Levels
Using Deep Neural Networks

Master's Thesis on Combining Sequence Prediction,
Generative Methods, and Image Processing

Jan Ebert

Intelligent Systems

Faculty of Technology

November 27, 2019

Declaration of Authorship

I hereby declare that this master's thesis was authored solely by myself, Jan Ebert. All figures and other illustrations were created by myself unless noted otherwise. All sources and any parts thereof – including program source code, data, tables and figures – used in this work have been properly cited and referenced.

Signature	*Place and date*

Abstract

We present *SMWLevelGenerator*, a simple and extensible preprocessing, training and execution pipeline for Super Mario World level generation. We start by generating parts of a level via generative adversarial networks, predict metadata via image processing-based methods and “generate” the rest of the level via sequence prediction. We implement a multitude of models, a pipeline handling different kinds of training data and easily extensible interfaces to handle both more data and more models. The training pipeline is optimized for both CPU as well as GPU usage and our preprocessing pipeline keeps database sizes to reasonable limits.

Our sequence prediction models learn to generate levels based on the most general way possible, meaning they are unplayable and bland. While our trained networks do not achieve the desired results, we hope to provide an environment for future research to intuitively use, improve and extend our collected knowledge.

A repository containing this text and the source code is available at `https://github.com/janEbert/SMWLevelGenerator`.

CONTENTS

1 INTRODUCTION

Currently, AI is being used to reshape our lives. However, a common discussion in recent times is malevolent usage of AI technology. Generative methods especially have evolved rapidly, becoming much more sophisticated than some years ago. With the advent of GANs being able to be used for movie production [1] or quality enhancements [2] and more refined language models that may produce texts passing the Turing test [3, 4], generative methods seem to be in a golden age.

This thesis provides a fun, artistic view into generative possibilities. Ultimately, this project provides research perspectives into combining different fields of machine learning, namely natural language processing, generative methods and image processing. With a combination of techniques used in these three fields, we generate most of our level by prediction, avoiding the problems usually faced during training of generative adversarial networks.
We present *SMWLevelGenerator*, a pipeline to train models capable of generating Super Mario World [5] levels from scratch. We encourage readers to become users and try out the provided functionality, maybe even extending the pipeline with new models or ideas.

While game level generation is not a particularly new field of research, Super Mario World provides a more complex environment than any related work has attempted to generate for. Also, this work uses both more and more recent models, focusing especially on the transformer architecture [6], originally introduced as a encoder-decoder model for neural translation tasks. Due to its functional similarity to the long short term memory architecture [7] and its faster training time, we believe it is a good fit for more challenging generation tasks such as provided by Super Mario World.

1.1 *Games and Optimization Problems*

Humans have been playing games for a variety of reasons including entertainment, socialization and education for over thousands of years [8]. They are an integral part to both human culture, society and problem specific understanding. In modern, recent, times, games and their theory have had an impact on other fields of research as well. John Nash proved [9] that finite games with multiple players have an optimal strategy, the

Nash equilibrium. It is now an essential part of economics and politics as well [9]; most problems with strategic interaction between several agents can theoretically be solved so that every agent gets the maximum reward based on every other agent's optimal decision.

The video game industry in particular has a large impact on the economy around the world. In 2018, the US video game industry even matched the US film industry, a much older and more mainstream industry, in revenue [10]. Possibly in part due to its economic impact, the video game industry has become responsible for many advances in technology, improving and developing techniques and hardware [10] which have now become the norm in many other fields as well due to their effectiveness. A very practical and relevant example is the use of graphical processing units (GPUs) to speed up the training of machine learning models.

We have established that games are playing an important part in society, furthering research and innovation in various areas and giving people the ability to learn and socialize in a comfortable environment. Due to the infinite possibilities in game environments, an enjoyable game exists for almost everyone – similar to how almost every person finds at least one music genre they enjoy.
However, while playing a game to optimize its underlying mathematical problem is fun and important, in this work, we will focus on the second part, research and innovation in the hopes of leveraging the new knowledge in other fields. While we also optimize on a problem, that problem is not related to a game itself but rather it is related on a meta level. We try to generate platforming levels, optimizing them to be similar to existing levels known to the system. Our system will analyze a dataset of levels and try to generate new levels based on the seen patterns. It will work without any prior knowledge or feature engineering done by a human.

2 FUNDAMENTALS

We will now lay out the more theoretical groundworks of this thesis. First, we answer what the problems we encounter in games are and how our work relates to them. We will then introduce our environment – first in a high-level view, then diving lower into a more abstract representation of what we are working with. Finally, we describe the theoretical aspects of the methods we will be using in this thesis.

2.1 *Super Mario World*

We are going to focus on a single game for level generation: *Super Mario World* (SMW). Super Mario World was released in 1990 by Nintendo for the Super Nintendo Entertainment System [5]. It is a 2-dimensional platforming game with the goal of reaching each level's exit in a limited amount of time. Games in the platforming genre confront the player with dangers like pits and enemies which have to be avoided by carefully jumping and maneuvering across them in a host of levels. The original game features 75 levels with 96 different exits (each level may have more than one goal, especially secret goals).
The player has access to a moveset including but not limited to running, jumping, picking up and throwing objects, and even shooting fireballs, flying and riding a dinosaur extending the amount of abilities even further. To access some abilities, Mario has to pick up power-ups usually contained in "question mark blocks" ("? blocks"). Blocks are commonly activated by hitting them from below. During a level, the player may collect various items like the power-ups mentioned above (which also let Mario take one extra hit before he is defeated) as well as coins and one-up mushrooms which increase the amount of lives (tries) available to the player.
Usually, when Mario lands on top of an enemy, the enemy is defeated. There are some enemies which Mario cannot safely land on or cannot defeat by jumping on top of them. Those enemies require a different trick like throwing something at them or hitting a block they are standing on from below.
Sometimes, the player may even have to use an enemy to complete a level. For example, the turtle-like "Koopas" get knocked out of their shell before being defeated. The shell has a variety of uses like activating an unreachable block, defeating enemies

and even giving access to mid-air jumps by throwing and subsequently landing on the shell.

The next two sections will give an in-depth look into why Super Mario World was chosen as a machine learning problem and just how complex the levels are. We will then give a more low-level overview into how the levels work and what one has to look out for when creating a database suitable for feeding into a machine learning model.

2.1.1 *Hacking Scene*

At the time of writing, Super Mario World is almost 30 years old. Due to both its age and success, it has been able to gain a large following of people not only interested in playing the game but also extending it. As with many other – especially retro – games, there is a read-only memory (ROM) hacking scene around Super Mario World. In this thesis, "ROM" will refer to the digital form of the Super Mario World ROM cartridge that would usually be inserted into the Super Nintendo Entertainment System; like the data that is burned onto a CD. A lot of tools have been created to manipulate Super Mario World; even in ways that the original creators hadn't programmed. People are able to create their own levels and share them with the masses by uploading a patch file that has to be applied to the original game ROM[1], thereby avoiding copyright infringement as sharing the ROM, Nintendo's protected property, is illegal. The program we use for applying patches is *Floating IPS* [11, 12], or *Flips*.

Due to the refined tooling available, users do not need to program in 65C816 assembly[2] to modify the game. Instead, even non-technical users are able to use one of the oldest tools to modify Super Mario World: *Lunar Magic* [13]. It features a GUI (that also enabled some great visualizations) and has optional features that extend Super Mario World with new functionality such as larger levels, custom coded tiles, and more. The creator of Lunar Magic, *FuSoYa*, modified the existing software to support both dumping and reconstructing levels, making this work possible due to the reduced workload of parsing the data from the binary ROM. The features are currently only available in a private build (based on Lunar Magic version 3.04) that will be used throughout the thesis to (1) parse level data from existing

[1] Usually the American version; CRC32 checksum `a31bead4`.
[2] The Super Nintendo Entertainment System's 16 bit microprocessor is based on the 65C816 microprocessor.

ROMs and to (2) write newly generated levels back into the original Super Mario World ROM.

With Lunar Magic as a user friendly tool granting creators all the freedom they could ever desire to design their own Mario levels, communities around ROM hacking emerged. One of those communities is *SMW Central* [14], hosting most of the available Super Mario World ROM hacks. Due to both SMW Central's filtering capabilities and its quality of content, it was the source for all levels used in this thesis (excluding the ones in the original ROM). All the hacks used were obtained from SMW Central where their respective authors (who are not necessarily related to SMW Central) uploaded them.

Being able to filter hacks was important for two reasons. One being that programmed custom behavior was out of the scope of this work, requiring both parsing the binary 65C816 assembly code and then understanding its every behavior. So we used the "vanilla" tag to exclude hacks using behavior not in the original game (also called the "vanilla" game; we will use these terms interchangeably). Also, to assert some form of quality in the dataset, only hacks with a rating greater than or equal to 3.0 were downloaded (our rating is the mean of all individual ratings, ranging from 1.0 to 5.0 and including *None* for no individual ratings). Sadly, some hacks are protected via an encryption (possibly offered by Lunar Magic) and do not allow reading their data; we had to omit all of these. All in all, over 17 000 unique levels in over 300 hacks were obtained thanks to the community keeping a masterpiece alive and fresh for not only all the years since the game's release but also for many years to come. While this amount of data is not a lot relative to what deep learning models usually train on in 2019 (with data points in the millions, even approaching billions), it may be enough for our models to get a grasp on what makes a level enjoyable. A list of the authors and a table containing more detailed statistics of the hacks is included in the source code repository.

Sadly, the "vanilla" tag is user-assigned, meaning users may label hacks with some non-vanilla behavior as "vanilla" while others may forget to assign the label at all, leaving us with both unclean and less data than we would hope for. We include scripts to check for some non-vanilla behavior and remove levels implementing those from our dataset. A list of removed patches, ROMs and levels can be found in the `stats` directory in the source code repository.

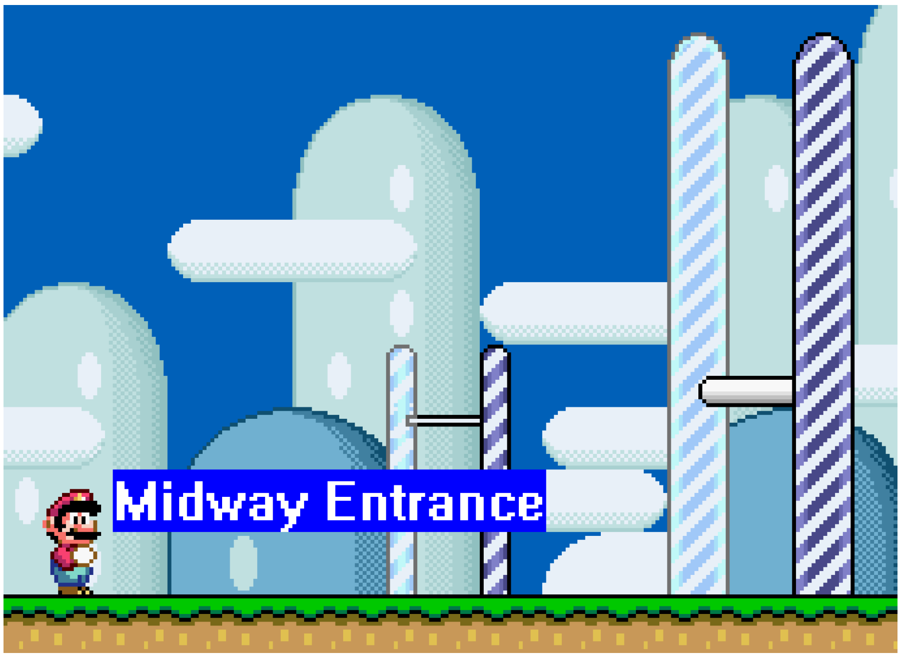


Figure 1: Aside from the midway and goal point, also depicted is that the midway entrance is separate from the midway point. While the midway point consists purely of tiles, the moving goal point tape is actually a sprite.

Finally, as mentioned, hacks operate on the real ROM as patches. Most of the levels of the original ROM are usually untouched and remain the same even after applying the patch. As we dump all levels contained in a ROM, we remove these duplicated levels afterwards. If any one of the dumped files is changed, we do not remove the level as it may be a bonus level or a level solely containing the goal post (often used for ghost houses).
In the original ROM, there are several duplicated "test" levels which are also removed. Hack authors may also leave unfinished levels or test levels in the ROMs which we cannot reliably detect. This will "pollute" our dataset a little but will hopefully not have a great effect during training.

2.1.2 *Levels*

We already assessed that Super Mario World is a complex game with many possibilities both in play and creative freedom in level design due to the vast number of interactions in the virtual world. We will now fill in some missing gaps of what makes up a level.

To give the player an easier or less frustrating time, most levels contain a *midway point* (shown in figure 1). When activated (achieved by touching the midway point tape), Mario will spawn at that point instead of the usual level entry point in case he is defeated. Most levels end when Mario reaches the goal post (pictured in figure 1; touching the tape is not required). Some levels end when Mario touches an orb while others end when a key is transported to and inserted into a keyhole. Finally, there are boss levels that are completed by defeating a boss. We will, however, not consider these.

Each level may be connected to a number of other levels. The game achieves this through two systems which will be explained in the paragraph on layers on page 8. The high-level connection between two levels is given by tiles with special behavior that can be entered by Mario. These tiles are called "exit-enabled" tiles. The most common are doors and pipes (only some pipes are exit-enabled; those cannot be visually distinguished from those which aren't).

Levels can have underwater regions through which Mario will have to swim, removing his ability to jump on enemies to defeat them. Some levels are vertical, growing in height rather than width. Other levels are covered in darkness, where light is only emitted in a small radius around Mario while others automatically scroll the level forward, forcing the player to react more quickly and not get stuck on a wall (when the level scrolls beyond Mario, he is defeated). Finally, a level may contain "blue P switches" that temporarily turn coins into blocks and certain blocks into coins, granting new paths to the player that are possibly required to reach the goal. A very similar tile is the "gray P switch" which spawns coins and special doors.

All these different tiles can be combined in any way imaginable. Most of those combinations in the high-dimensional level space are non-playable nonsense. However, as described in section 2.1.1, due to the large amount of possible – and, more importantly, enjoyable – interactions between all the different pieces contained in the original game, a seemingly infinite number of unique, fun levels can be created.

We are now going to leave the high-level view of levels and take a look at how everything works together on a lower level: Levels are made up of *screens*; matrices of the same size. Each screen is made up of two subscreens; in horizontal levels, subscreens are stacked vertically and the whole, combined screens are concatenated horizontally while in vertical levels, subscreen

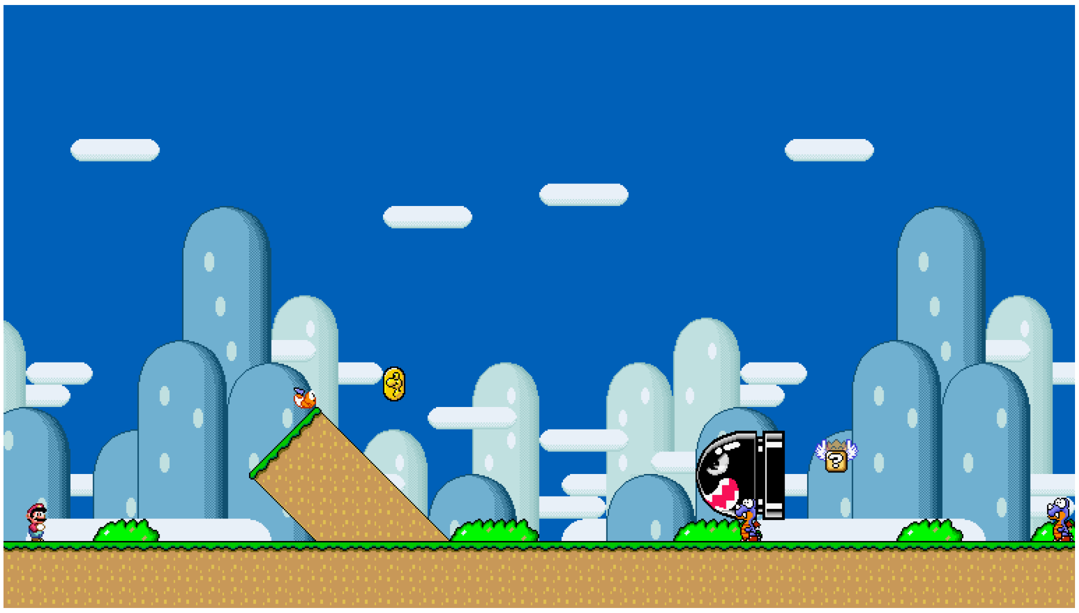


Figure 2: The first three screens of level 261. The main entrance point is depicted as Mario.

are stacked horizontally and screens concatenated vertically. While subscreens are important in data preparation, we will not focus on them in this work and instead always refer only to whole screens. A screen is a 2-dimensional slice of the level consisting of a number of columns and rows. Screens in horizontal and vertical levels have different sizes (even if horizontal screens are transposed), but otherwise, screen sizes are constant over different levels. While Lunar Magic supports non-vanilla behavior in that screens can have other sizes (although still a constant size per level). Like with all non-vanilla modifications that have a direct impact on data, we are going to ignore that feature. For simplicity, we are also only going to train on horizontal levels. While support for vertical levels in database generation exists, models that support both types of levels are more complicated. Also, vertical levels only make up a fraction of total levels, so not a lot of data is lost (circa 2 000 of the over 17 000 levels are vertical).

Calling a screen a matrix is not the whole truth. While screens itself are basically only areas in a level, those areas can contain many different layers of other types of objects.

LAYERS While we will explain "layers" in this section, in the hacking community, the term "layer" refers to both interactive and graphical tilemaps (of which there are three of). When we

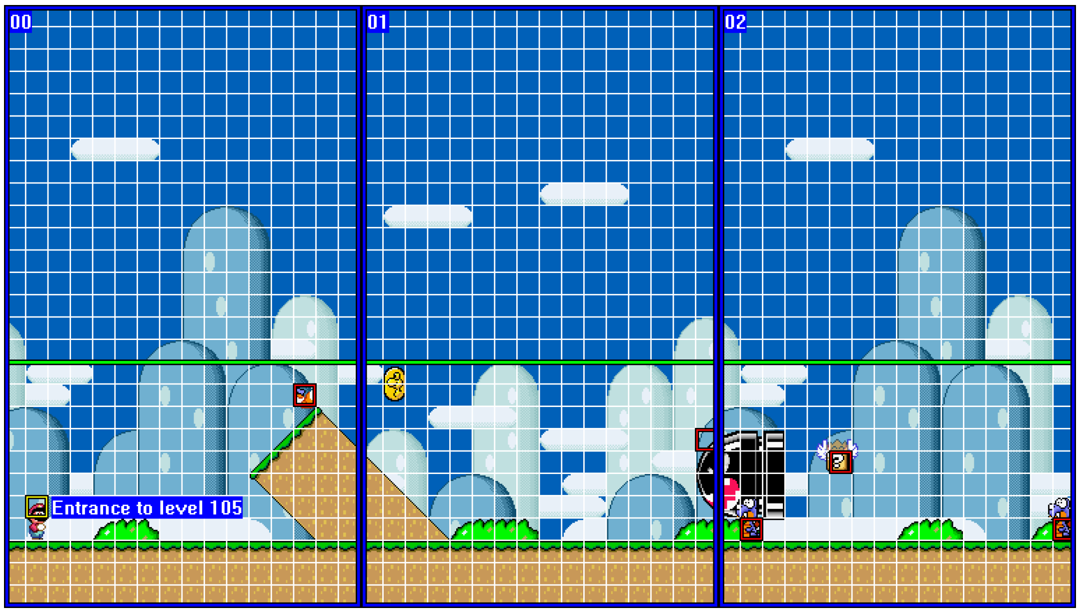


Figure 3: The first three screens of level 261 with detailed information and a white grid distinguishing individual tiles. Screens edges are blue with their hexadecimal number at the top left; the green line in the middle indicates subscreen boundaries. The position of the main entrance has a yellow border. The actual positions of sprites containing information have a red border.

refer to the second meaning, we will explicitly list a number behind the layer (for example "layer 2").

Aside from the previously explained screens, levels consist mainly of five different *object types* which each correspond to a file dumped by Lunar Magic. Also, each of these object types needs many layers that contain all the data. The types and a short description are listed here:

1. *Tiles* are most of the static blocks in a level. They usually make up the ground, platforms, walls and ceiling of a level (if present). Also, most collectibles that are immediately visible in the level are a tile; examples are coins or moons[3]. Some special tiles like doors and some pipe tiles are exit-enabled, meaning they lead out of a level (exits are described below). Even non-interactive background objects like arrow signs or bushes that only exist for aesthetical reasons are tiles. In the hacking community, this layer of tiles is also called "layer 1".
   The original game supports 512 tiles, each with possibly different behavior. Lunar Magic extends the amount of

[3] Moons give the player three lives upon collection.

tiles by allowing users to define new, custom tiles that either (1) reference another tile's behavior (these references may recurse but not loop), so only changing the texture of the tile or (2) point to a custom procedure, so possibly changing both, the texture and the behavior of the tile. Due to only allowing vanilla hacks that promise not to implement custom behavior, we can ignore the second case and simply follow references until finding one of the original tiles.

As mentioned at the beginning of this paragraph, tiles may exist on two layers in the level. Layer 2 may contain interactive objects or just non-interactive graphical background data. A special property of layer 2 tiles is that they can move if desired (how that is achieved will be described in a bit). When layer 2 is used for interactive tiles, it cannot act as a background tilemap. Therefore, a third (strictly non-interactive) layer exists for the sole purpose of a background tilemap which is usually in layer 2. This layer 3 is not related to tiles in any way. For simplicity, we ignore all data in layers 2 and 3.

2. *Metadata* is any data that changes the behavior of a level without belonging to a certain position in the level. For example, which tileset (that is, the graphics) or which background music is used in a level is decided by the metadata. A level's metadata also contains its entry points. Finally, metadata may even change the behavior of certain tiles. We will describe metadata in detail in its paragraph on page 13.

3. *Entrance points* are the positions of the main and midway entrances.

4. *Sprites* are usually moving or movable objects that interact with Mario and each other in a certain way. For example, enemies and even goal objects like the orb, key and keyhole (described in section 2.1.2) are sprites (while the goal post is a tile, the moving tape Mario can cross through is also a sprite).

   Sprites may also have other effects on the level; for example, sprites control auto-scroll behavior in a level or the movement of layer 2 tiles. Sprites may also affect a level's lighting or background objects. When Mario reaches a screen with one of those special sprites, the behavior will activate.

5. *Exits* connect one level to another. An exit is screen-based, meaning it will be used by all exit-enabled tiles in the screen the exit is contained in. There are two types of exits: *Screen exits* send Mario to an *entrance point* (described above) of another level while *secondary exits* can send Mario to a designated point in the level to be entered, described below.

6. *Secondary entrances* are the destinations of secondary exits. These destinations cannot lie anywhere; they have certain pre-determined positions based on a table of positional values and based on whether they are in the first or second subscreen of the screen the secondary entrance lies in. Any number of secondary exits may point to the same secondary entrance.

Now that we have described the different kinds of object types we will be dealing with, we will take a quick look at how they expand in the abstract space our model is going to work with. It should be noted that each object type contains individual *type instances* (for example, each unique type of tile like "ground ledge block" or "gray cement block" is a type instance of the object type "tile"). Each of these type instances regards its own layer to get a binary encoding of the level. Figure 4 may be helpful to understand what we mean. Prior to constructing our data, we need to know the amount of layers. We already know that there are 512 unique tiles in the vanilla game. For the sprites, we need the amount of unique sprites used over all hacks. To represent all possible exits, we need the maximum amount of screen exits and connected secondary exits and secondary entrances. In each hack, we count the amount of screen exits and connected secondary exits and secondary entrances and take the maximum of each of these over all hacks. We also need 2 layers to represent the level's main and midway entrance points. If we sum up the total amount of all unique tiles and sprites, the maximum amount of screen exits and (connected) secondary exits and secondary entrances plus the two entrance points, we get a total of $512 + 241 + 511 + 449 + 449 + 2 = 2164$ layers. A horizontal level consists of 27 rows and at maximum $16 \cdot 32 = 512$ columns (16 columns per screen, 32 screens at maximum). We may therefore get a maximum amount of

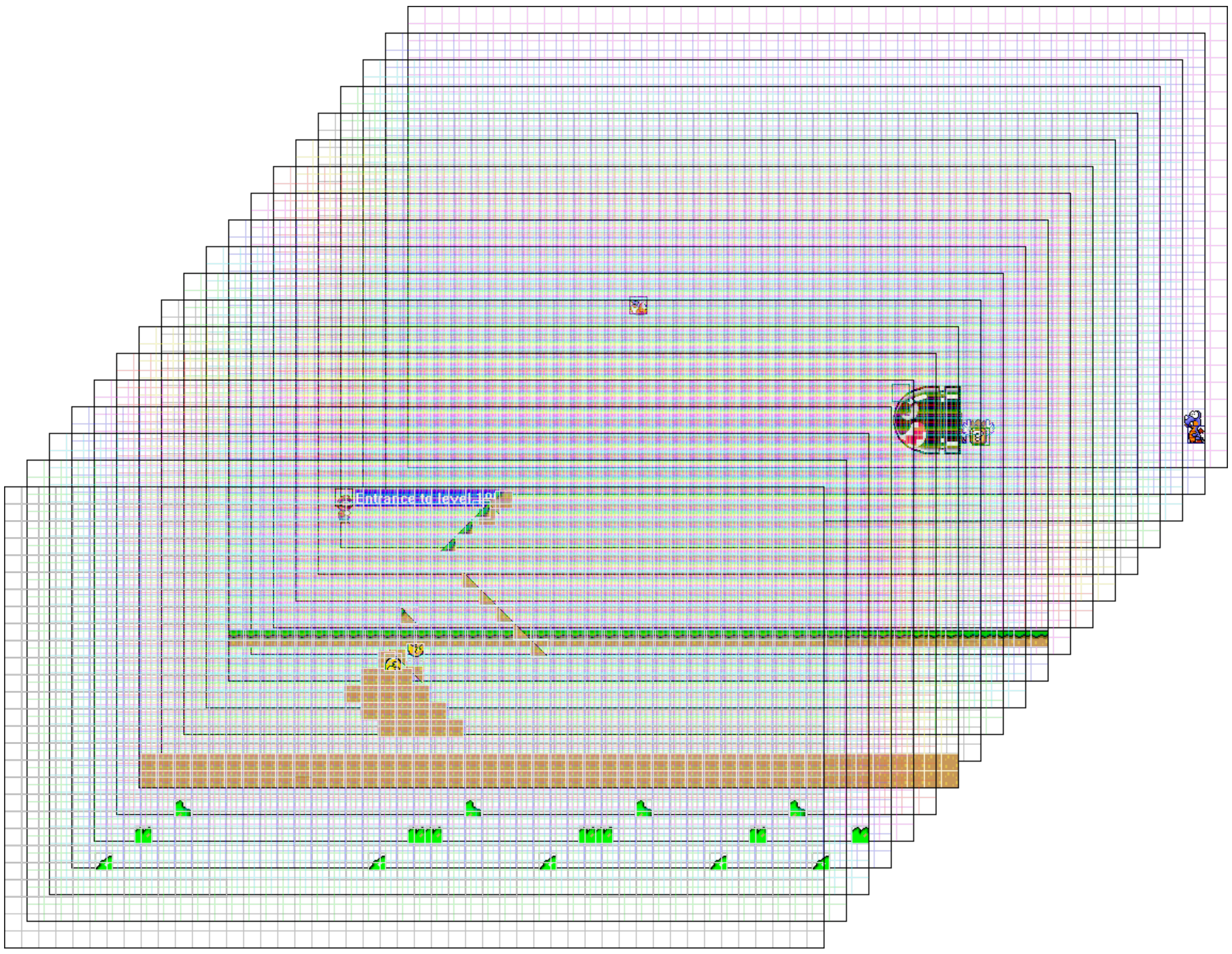


Figure 4: The first three screens of level 261 shown as individually sliced layers in order of our abstract representation. The layers contain in order (an asterisk marks a non-interactive tile): (tiles) empty tile*, bush left*, bush middle*, bush right*, Yoshi coin top, Yoshi coin bottom, ground background*, slope right corner, slope left edge*, slope right edge*, ground, slope top edge, slope top underside*, slope left corner, (entrance points) main entrance, (sprites) flying ? block, Bullet Bill, Rex, sliding Koopa without shell.

Each layer has a grid, where the colors cycle through black and rainbow colors for each layer. Tiles are highlighted with a brighter color. Elements in the grid other than tiles containing the actual position are highlighted in a darker color. Note that the converted data contains a lot of empty layers not visible here.

In the abstract representation, each highlighted grid element becomes a 1; everything else is 0.

$27 \cdot 512 \cdot 2164 \approx 30 \cdot 10^6$ entries in the abstract, binary representation of a level.
As the dimensionality of levels containing all layers is quite high (and mostly filled with emptiness), we provide filters to only process chosen level data making training both easier and faster. We even provide a filter so only sprites directly relevant to reaching the goal are kept. Due to Super Mario World's complexity, we cannot cover each case. Sometimes jumping on an enemy or throwing a Koopa's shell may be required to finish the level. These are just some examples of the vast majority of possibilities. The filter should include most – if not all – sprites required for reaching the goal in the vanilla game, though. These filters will be described in detail in section 3.2.

We will now take a look at the effects a level's metadata has on it.

METADATA As already explained, metadata contains information about a level that is not related to any position in the level like changing a level's graphics or music. However, even these effects may influence unrelated objects. For example changing a level's fore- and background graphics automatically changes the behavior of some certain tiles. Some tiles may only exits for certain graphics settings. Due to these interactive influences metadata has on a level, we need to take it into account when training our model. Each level has three headers containing metadata. A primary and a secondary level header and a sprite header. While we could feed our model all the different metadata points, we are only interested in the ones that provide information on two issues: (1) changes to interactive parts of the level and (2) changes to a level's size or "vanilla-ness". We need (1) so our model may learn the correct representation of each tile for each level and we need (2) to be able to filter undesired levels. As an example for (1), we already mentioned the graphics setting and its influence on tiles. An example for (2) is how many screens the level contains or whether the level uses Lunar Magic's expanded format or whether it's a horizontal, vertical, or even boss level.
Finally, positional information about the main and midway entrance of each level is also stored in the metadata. We, however, do not treat these as metadata and instead store them in the level matrix like tiles.

Let us now take a look at how Mario levels are different from levels in other games; at what makes them special and how it influences our decisions regarding model design.

COMPARISON WITH LEVELS IN OTHER GAMES While Super Mario World levels mostly follow a linear style of getting from one side of the level (the beginning) to the other (the goal), some levels include non-linearities. Levels may even loop and require the player to solve a puzzle in order to progress or backtrack after acquiring an object. Or after activating a P switch in one level, a door in another may suddenly appear. These influences on both a level's linearity and its interaction with other levels makes it hard to get a general generative solution for a level. We therefore restrict our problem to only generating one independent level each, not a level including the levels its exits point to. This means that we will get incomplete levels or unconnected exits. One way of addressing this would be to generate a more holistic view, meaning concatenating a level to its connected levels and generating all at once. Even then, we would need to interpret the way the level plays out to concatenate the levels in the right order. Due to the immense increases in architectural (software) and model complexity, this is reserved for future work.

In Super Mario World, each level is independent from another (excluding levels connected by screen or secondary exits)[4]. This means we will not have to pay attention to Mario's state in between different levels. For example, in other games, a permanent power-up may be required to progress in a future level. We do not have this issue, enabling us to start fresh with each level we generate. In the same vein, as levels are not connected with each other (again, excluding the connections by screen or secondary exits), our problem size shrinks as we do not need to get a complete view of the whole game world. While the original game separates levels into different biomes or worlds (grassland, cave, forest, . . .) by changing the graphics of sequential levels to the corresponding theme (based on metadata), some hacks may tell a story by tuning the general feel and environment of one level to fit the following one. We can ignore this and still obtain a new, interesting Super Mario World game,

[4] A counter example is Super Metroid [15], also released on the Super Nintendo Entertainment System, in which the whole world is in concept one giant level, also divided into screens.

even though the cosmetic, meta-view of the world whose story is told through the game, is lost.

With that, we have described the most important concepts in Super Mario World and how they influence our decisions regarding model design and data selection. We will now explain the machine learning-related methods used throughout this work in detail.

2.2 *Methods*

In this section, we will introduce the primary ideas we apply to obtain a level generation pipeline. Before we list the methods, we take a look at how to actually achieve our task. In the final subsection, we will talk about Julia, the language the system is programmed in.

By analyzing inputs of large dimensionality (the level's layers and metadata), we want to generate similar, new levels of large dimensionality (also level layers and metadata). This is a very complex problem, requiring careful planning and a lot of computational power to train a model large or refined enough to capture all the essences we need for a new, enjoyable level. A *generative adversarial network* (GAN) [16] would – with the present knowledge in machine learning – be exceptionally hard to train on a task with a dimensionality this large. A lack of proper computational resources makes this even harder as training a GAN requires training two full models in parallel. While a GAN would be perfect in theory, allowing us to learn the dataset without a modified loss function, due to the issues with training, we try another approach: Generating only the first screen by the GAN and generating the rest of the level using sequence prediction methods makes training much easier, allowing us to get a simpler solution to our problem. Although the results will not be as good as with a GAN due to a non-adversarial loss function, with some tuning, good results should be achievable. Also, training of both, the GAN and the sequence prediction model, should be much easier than a single, large GAN. To generate the metadata, we use a simple image processing model that predicts a level's metadata based on its first screen.

The methods will not be listed in pipeline-sequential ordering (the order in which they will be applied to obtain a generated level) but instead we will focus on the amount of data the different methods generate. The sequence prediction task is the most important part of the pipeline, generating a complete level from a minimal amount of input. To obtain the first input of the to be predicted sequence, we use a generative model. Finally, to generate the metadata described on page 13, we use an image processor that predicts the metadata of a level by analyzing its first screen – the output of the generative model and input to the sequence predictor.

Figure 5 gives a visual summary of the pipeline.

2.2.1 *Sequence Prediction*

The function describing our sequence prediction task in the 3-dimensional case for per-column prediction is

$$f : \mathbb{R}^{(k+1+r\cdot l)\times c} \to \mathbb{R}^{(1+r\cdot l)\times c},\ c \in \mathbb{N}^{+},$$

where k is the size of a constant input of metadata (explained in section 3.3), r is the amount of rows in the given level, l is the amount of layers in the given level and c is the amount of columns in the input (this is variable). The added 1 in the first matrix dimensionality is a bit determining whether that column is *not* the last column of the level. To simplify the above, the input is the level, where each column is concatenated with a constant vector of metadata and a 1 except for the last column which has a 0 at that place. The output should be the input (except the constant part of each column) from the second column onwards with a vector of zeros at the end if the level has ended or the predicted next column of the level otherwise. Simplifying even further, given an incomplete level as input, we wish to predict the next column for each column in the input.

Tasked with completing an unfinished sequence, we turn our head to sequence prediction models. The models we are using are also called sequence-to-sequence models. We will closely orient ourselves on natural language processing techniques, specifically natural language generation. The models we evaluate are *long short-term memory* (LSTM) [7] stacks (with a fully connected layer at the end) and *transformer*-based models [6]. More specifically, the LSTM stacks are based on *char-rnns* [17] and the transformer-based models are *GPT-2* [4, 18] models. As a baseline, we implement a non-learning random model that simply outputs sequences of either 1 or 0 based on a user-supplied chance $p \in [0, 1]$.

The reason we use techniques from natural language processing is that levels share many similarities with natural language: References to both prior and future objects, long-term dependencies and some amount of redundant information.

A long short-term memory cell's specialty is a recurrent memory that "remembers" values of a sequence, depending on connection weights. Due to this behavior, it is well suited for both non-linear and long-term dependencies in Mario levels as both can be captured from any prior, relevant sequence element (for example, a keyhole at the beginning of the level implies a key later and the other way around).

Transformer-based architectures rely on a combination of attention functions with multiple "heads" and dense networks. These networks improve upon the LSTM with parallelizability which – with the required computational power for increasingly large neural networks – is a very important factor. Unlike LSTMs, transformers do not keep a hidden state; they analyze the whole input at once, masking future sequence elements appropriately if desired (as we do). Due to not having to keep a state, transformers can analyze multiple different inputs at once, non-sequentially. This enables the parallelizability which in turn enables faster training enabling larger models. With these advantages, we assume that transformers should be better suited to the task due to its high dimensionality. With the GPT-2 architecture [4], transformers' capabilities in natural language generation tasks have been shown. While those models used an amount of parameters that could not possibly be trained with our available resources, we hope that a smaller, with our resources trainable model may still be marginally larger than an LSTM with the same training time.

2.2.2 *Generative Methods*

The underlying problem in the 3-dimensional domain we are trying to solve with our generative models is the following:

$$g : \mathbb{R}^n \to \mathbb{R}^{27 \times 16 \times l},$$

where n is the size of a latent vector of noise our model expects and l is the amount of layers the generated level will contain.

For our only actual generator in our pipeline, we supply three models: a standard *deep convolutional generative adversarial network* (DCGAN) [19] and two *Wasserstein GANs* [20, 21]; one also based on the DCGAN, another based on fully connected layers (we will also call these models MLP-based [22]). Both of these models are specially fitted with stride, padding and dilation values to obtain the correct screen size ($27 \times 16 \times l$, where l is the number of channels or layers).

We decided on using GANs as our generative models due to prior experience and as we imagined their generative capabilities to be above autoencoders [23–25]. Whether this is true should be investigated, for example by adding and testing autoencoder models as well which was sadly not feasible for us due to time constraints.
Another factor in our decision was the hope that the adversarial objective GANs rely on will improve both the quality and the size of the space of generated levels as our models will be less prone to generalization and not have the issues associated with likelihood-based methods that most autoencoders share as well (unlike adversarial autoencoders [26] which solve these, making them a great fit for future experiments).
Finally, the reason we employ both convolutional and MLP-based GANs is primarily due to empirical reasons. While convolutions may suit the edges in Super Mario World levels, maybe enough levels employ either too small or not enough edges to make the convolutions worthwhile.

2.2.3 *Image Processing*

Our image processing models, or metadata predictors, learn the following function in the 3-dimensional case:

$$m : \mathbb{R}^{27\times 16\times l} \rightarrow \mathbb{R}^{k+1},$$

where k is the size of the constant input (briefly introduced in the section on sequence prediction models on page 16 and further explained in section 3.2) and l is the amount of layers. The additional dimension is the bit determining whether the level does not end here (also described in the sections we just mentioned).

Similar to the generative models, we also use both a deep convolutional network with stride, padding and dilation values fitted towards the correct screen size and a model consisting of fully connected layers. Our reasoning for implementing both models was the same as for the GANs, described above.

Our GAN does not generate the level's metadata in addition to its first screen (which we assume would be more correct due to correlations in the network). Even though implementing a generative model for both first screen and metadata generation would have been the best solution, we decided to go the direction of simplicity and decorrelate these two tasks just like we

did with the sequence prediction models. This has a couple of reasons: For one, the added metadata vector would complicate purely convolutional GANs. Another reason is that this greatly expands the space of input data from data in the set 0 to 1 (or interval from 0 to 1 inclusively, depending on if we normalize the data) to real numbers in the interval from 0 to 512 inclusively (assuming our currently supplied metadata; more information in section 3.2).
While we are certain that this influences our generative results negatively, we hope that decoupling these two tasks yields us an easier time during training. As GAN training is already notoriously hard, we estimated that opting for a GAN that implements both tasks would make training even harder. This estimate may be wrong as the metadata could also help as more correlation in the data may be found.

In the end, while both approaches should be experimented with, this exercise will be left to subsequent work. The image processing model may be removed in favor of a GAN-only approach if that proves to be the better choice.

2.2.4 *Julia*

Our source code is written in *Julia* [27]. Julia is a programming language whose version 1.0 was released in August 2018. It presents itself as combining the performance of C, the productivity of Python and the generality of Lisp. With this combination, we were able to quickly assemble a very performant, high-level data processing and low-level machine learning pipeline. It enabled us to – for example – change our array types as desired, enabling incremental optimization from standard arrays to GPU arrays to sparse arrays to sparse GPU arrays. While starting with version 1.1.0, we updated our code to now work with versions 1.2.0 (the latest stable release) and 1.3.0-rc5 (the latest release candidate for the upcoming version). Running on a version 1.3 or later enables multi-threading for the data iterator which may improve training speed by a large magnitude. Our recommended version to run the code is therefore 1.3.0-rc5 (1.3.0 stable was released just before submission; while we certainly recommend the stable version rather than the pre-release and assume there are no breaking functionality changes, we cannot guarantee all of our code to work).

Thanks to community packages, we were able to easily implement functions accelerated by CUDA [28] and cuDNN [29].

While Julia brought great advantages with it, some disadvantages were issues and bugs in packages and the general immaturity of the ecosystem (an example concerning array primitives for sparse arrays on the GPU is given in section 6.1). Another issue were long compilation times which are to be addressed soon.
We list the most important packages for our pipeline with explanations, advantages and disadvantages in section 6.1.

Next, let us take the theoretical knowledge we introduced in this section and make it practical in the next one.

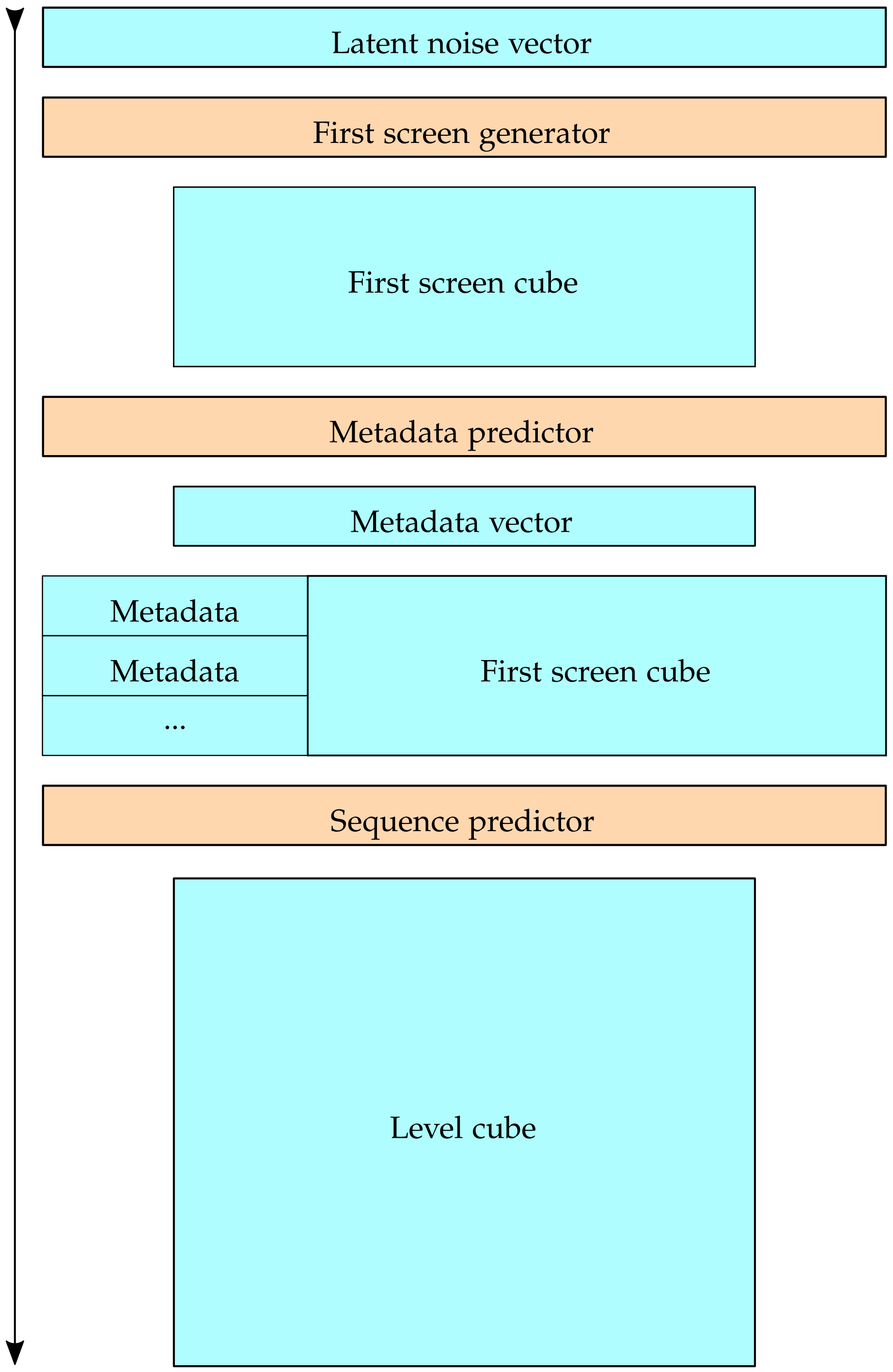


Figure 5: An overview of the generation pipeline for the 3-dimensional case. Blue blocks indicate abstract objects while orange blocks denote models. The metadata vector is concatenated to the first screen as is even though the figure may suggest some kind of shortening. Not shown is the bit indicating whether the level has not ended and the transformation and flattening of the first screen cube to the correctly preprocessed value (these are explained in detail in section 3.2).

3 DESIGN

The previous section introduced all the methods we are going to be using in this work in an abstract way. We will now go into the practical implementation of everything: how we did it and why we did it that way. We will describe the shortcomings or advantages of our approaches and list trade-offs we made to achieve our goals.

3.1 *Setup*

With our code living in a Git [30] repository, we want to achieve – aside from the obvious version control advantages – an easily reproducible but also portable environment (by which we mean no large binary files). We provide several individual scripts for the whole setup pipeline of our environment, including the following:

- downloading the required tools (optionally also additional ones)
- downloading the properly filtered hacks (as we established in section 2.1.1, these are in patch form)
- decompressing them to properly named directories (supporting either `tar` or `7z` as desired)
- applying the patches to the original ROM
- dumping all levels in each ROM

All of these are combined in a single convenience script the user can execute in order to obtain their own environment. We specify a default directory structure our source code is based on so no paths have to be changed if using the convenience script. Note that some patches, ROMs and levels cause issues; we mentioned that these are listed in the `stats` directory in section 2.1.1.

Some time after our download of the over 300 hacks, SMW Central implemented protection against too many requests in a certain time frame. Even though we have set a generous wait time limit between downloads, we understand that troubling the servers over and over for the same files may not be desired. Due to this, we supply scripts that fetch the files (tools and hacks) from their original source, SMW Central, and scripts that download the files collected in a custom location provided by us (the convenience setup script exists in both versions).

While we mentioned portability, the scripts above are currently only implemented in shell script[5]. For cross-platform support, these will be implemented in Julia as well at a later time, eliminating another dependency.
For people either not interested in a manual setup or without access to a shell script interpreter, we provide downloads for pre-computed databases in the dimensions we support out of the box (listed in section 3.2). If, on the other hand, the levels are available, generating those databases is done in a single function call (`generate_default_databases()`).

Our repository contains pre-calculated values for several statistics (these exist mostly to minimize the amount of layers we supply to the models). However, when setting up manually, we recommend running the functions
`SMWLevelGenerator.Sprites.generateall()` and
`SMWLevelGenerator.SecondaryLevelStats.generateall()` to pre-calculate statistics based on the user's dataset. If any of the files containing statistics is not found, that statistic is otherwise calculated on each load of the module. Depending on the size of the dataset, this make take some time, therefore, pre-calculating will speed up precompilation of the module.

Several other scripts allow further processing of the data, including:

- summarizing statistics of the hacks (like authors, rating, ...)
- removing duplicate and test levels based on their CRC32 checksums
- removing levels with non-vanilla behavior

These have already been implemented in Julia.

Detailed setup instructions are found in the project repository's README.

3.2 *Preprocessing*

Our data was preprocessed so we obtain the following: A space-efficient, performant, abstract level matching what we as humans see. To actually get all those traits, we had to apply some

[5] Bash compatibility, aiming for POSIX.

tricks. First, we need to establish the following: Remember that each level may contain $30 \cdot 10^6$ entries at maximum – this requires $30 \cdot 10^6$ bits $\div 8 = 3.75 \cdot 10^6$ bytes $= 3.75$ MB (MB are megabytes) of storage *per level*, assuming we are able to compactly store eight entries per byte (usually you need one byte per entry but as we have binary values, we can store one value in one bit). With approximately 17 000 levels, we would then need $3.75\,\text{MB} * 17\,000 \approx 63.75 \cdot 10^3\,\text{MB} = 63.75\,\text{GB}$ (GB are gigabytes) of storage at maximum. While data is cheap and this calculated number is the absolute maximum amount of storage we may need, we would like our data to be more compact for both efficiency and portability.

Even the data containing only layers of all tiles is so large that keeping the full database in 8 gigabytes of memory is not possible[6]. That means that creating a database is slower – as we would have to partially write results to the file, garbage collect, then restart with another partial result – and that reading, the much more important part, is also slower as we would have to continually swap out the database in memory (this is the effect on efficiency we mentioned). As we want to train deep learning models which may require many epochs over our data, this is not feasible since reading gigabytes of serialized data is very slow.

To improve both space efficiency and performance during training, we started using sparse arrays for larger data. Sparse arrays do not store zeros; since our data is full of zeros (due to the binary encoding which means we also have to store whether something is *not* at a position), we are able to save a lot of space and gain a lot of speed due to improved computational techniques. We will not have to iterate over our whole sparse array but only over the non-zeros when computing a matrix-matrix-multiplication, for example. Also, CUDA supports sparse arrays via the cuSPARSE [31] library, enabling GPU-powered sparse array computations as well (this sadly has not worked out yet, as we explained in section 2.2.4). Julia does not support sparse arrays above two dimensions, so we needed to write our own custom storage for 3-dimensional sparse arrays. While we managed to greatly reduce storage space this way, it was still not enough for storing all possible layers. Our two final optimizations were compressing the indices of the sparse matrices to the smallest type still being able to index

[6] This is the amount of unused RAM the author had available on his laptop while working with about 2 000 browser tabs open.

and – because that was still not small enough – only storing non-empty sparse matrices in an array. We wrote a custom compressed sparse 3D array format for that.

All in all, we now support four different compression levels with two possible extensions based on compact bit arrays where each value fits into one bit. The final size of the 3-dimensional database containing all layers is roughly 430 MB if the highest compression level is used. For sharing purposes, this data can be heavily compressed due to recurring values, resulting in a 28 MB file when compressed with the combination of `tar` and `gzip`. A 7-Zip-compressed file consumes about 16 MB of storage[7].

While compression is great to share data, we are also interested in speed. The usage of sparse arrays already nets us an increase in calculations involving them. When not highly compressed, we store 3-dimensional data sliced along the second dimension, so by each layer from left to right instead of from "front to back". To better explain, remember that we *do* store layers "front to back" for the highest compression level; most layers of object types are empty but most columns (slices from left to right) in the level are not.
As a result, highly compressed data is stored in a way making it more costly to bring it back together as caching suffers a lot when we need each layer column by column (we would need to iterate over all layers to get their first columns which we concatenate to obtain our first input. This process has to be repeated for all columns).

Before we explain further, we should note the following: We create a database for each "type" of training data, meaning models can operate on well separated databases (which obviously comes at a cost of space since data will be duplicated). It would also be possible to work on one database containing all possible values at the cost of speed due to having to convert all the data to the desired training data[8]. We separate the desired training data by both its dimensionality and by the types of layers contained in the data. These distinctions are relevant for database generation, level generation and model creation as preprocessing functions and the models need to know their input sizes ahead of time.

[7] No special settings for `tar`, but the following command line options for 7-Zip: `-t7z -mx9 -m0=lzma2 -mmt2 -md1024m`

[8] This has not been implemented.

The dimensionality of training data is separated into three different types to allow increasing model complexity:

1. 1-dimensional data containing only one row (or column for vertical levels) of one layer of the level. This row is computed in one of two ways (as given by the user): either the row is (1) the row containing the maximum amount of values in the given layer or (2) a combined row obtained by squashing all elements that are equal to one in the given layer vertically. See figure 6 for a simple visualization.

   To determine which layer to use by default (the tile we assume to be the ground tile for the level), we use a heuristic: we find the first non-empty tile below Mario's starting position[9]. If there is no non-empty tile below that position, we return the layer corresponding to tile 256, the default ground tile.
   It must be noted that this is not the correct tile for all levels.

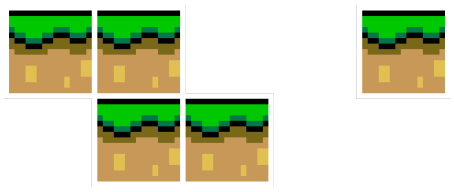
2. 2-dimensional data containing only one layer of the level. The default layer is the same as for 1-dimensional data and suffers the same issues.

3. 3-dimensional data containing whole more than one layer of the level. By default, all layers are used.

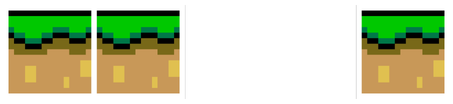
Each of these can work on any combination of object types of data in the level; for example, we can have 2-dimensional data containing only Koopa sprites. Or 3-dimensional data containing only tiles and goal sprites (as mentioned in the paragraph on layers on page 8, section 2.1.2, we defined a subset of the object type "sprite" containing only sprites directly relevant to reaching the goal). With this, we can easily test models on smaller data and adjust towards more complex problems.

For 3-dimensional data it is necessary to provide some information for each of these combinations so that – as mentioned above – both data loaders and models can properly handle different input sizes.
This information is the object types the data was created with and the input size for sequence prediction models and GANs (the input sizes will be calculated automatically in the future). Our system is easily extensible to support any desired types; we

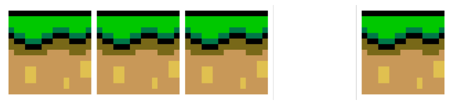
[9] We mention this explicitly as the main entrance is two tiles above Mario's feet. Also, the empty tile is the tile corresponding to tile 37.



(a) Input



(b) `squash=false`



(c) `squash=true`

Figure 6: The two ways to obtain 1-dimensional data. Given (a) as the input for either case, the outcome is either (b) when taking the maximum over all rows or (c) when squashing over all rows.

only implemented the following (as mentioned, only 3D data needs extra implementations):

- `1d`: any 1-dimensional data
- `2d`: any 2-dimensional data
- `3dtiles`: 3-dimensional data containing all tile layers
- `3d`: 3-dimensional data containing all layers

We also provide specific model creation functions for each of these dimensionalities, giving the user simple access to the correct model for each task. We will discuss this in detail in sections 3.3 and 3.4.

In this thesis, we work with data in column form. This means that our models read levels column per column from the left to right. We decided on this order as it captures elements in the past better than reading the level row per row (where you go back and forth between the start and end of the level). We also mode it possible to read the levels tile by tile if desired. The order is the same as for the columns, reading from top to bottom, then from left to right. It is also possible to change this order to read from bottom to top, then from left to right. This may improve results at it may be more important to first analyze the ground before predicting data above.

Now, we will summarize all the simplifications in our data set.

3.2.1 *Simplifications*

We already explained our simplifications in various locations above. This is a comprehensive list of all of them:

- Levels are observed independently, meaning no connected levels are seen.
- We omit a lot of metadata. (We suggest the omitted data is not related to level generation. As they are already parsed, including the omitted data is only a matter of uncommenting and adding a few lines of code, adjusting model inputs and database generation.)
- Test levels or unfinished levels – unless contained in the original game – are in the dataset.
- Any data that is out of bounds is ignored (for example, sprites may "fall down" from a position above level bounds; we ignore those).
- We assume that levels always go from left to right. This is only a problem during training as it arbitrarily makes training harder (only very few levels go from the right to the left). (We could determine whether the main entrance is at the left or right and horizontally flip the level accordingly. Even with this augmentation, handling levels with a starting position in the middle is simply not possible with our architecture.)
- The type of tile that is assumed to be a level's ground (relevant for 1- and 2-dimensional data generation) is determined heuristically (explained above; the first non-empty tile below Mario's starting position or tile 256 if none is found).
- Levels that are (1) vertical, (2) boss levels or (3) levels with layer 2 interaction are excluded.
- Levels with modes deemed unusable by Lunar Magic are excluded.
- Expanded levels (as designated by Lunar Magic) are excluded.

- If the database contains a single sprite, we exclude all levels without a sprite header.

We will now explain our sequence prediction methods in detail.

3.3 *Generation via Prediction*

We now focus on implementation details and design decisions regarding our sequence prediction models. In the following explanations, we will assume the level is read column per column. The only difference when reading per tile is the input size. While each of our models may have different input preferences for a sequence (vector of vectors, matrix, . . .), we will focus our explanation on sequence elements as these all follow the same principles. Each sequence element is a vector consisting of

- a constant input part containing metadata,
- a bit that indicates whether the level has not ended yet (so it is 1 if the level has not ended yet and 0 if this is the last column or beyond), and
- a column (or single tile) of the level.

The constant input part supplies our model with the following metadata:

- level number
- amount of screens
- level mode
- level graphics (remember that these influence interactivity as well)
- main and midway entrance action (these determine whether the level is a water level)
- sprite buoyancy (how sprites interact with water)
- disable layer 2 buoyancy (see above but for interaction with layer 2 data)
- sprite mode

As mentioned, this list is easily extensible if desired as our foundation contains and parses all relevant information.

To support many different models, we implement an abstract model class (also called the "model interface") that requires extending only a few methods for new models (if the provided defaults are not already correct). One of these methods solves the problem of the desired input for the model. Our data iterator will automatically shape the input correctly for each model. The currently supported input types are vectors of vectors and matrices. Single, long vectors containing the whole sequence are implemented but some questions about how their data is ordered must be answered individually (for example, is the constant input provided only once or for each column like for the other input types).

We implement a second loss function, the "soft" loss, that assigns a lower penalty (or none) to sequence elements that were predicted incorrectly if the prior two sequence elements were the same. With this, we hope to de-regularize our models so they are less heavily influenced by sudden changes in the level. The user may choose to use either one during training (see section 3.7.2).

By default, our LSTMs use the rectified linear unit activation function [32, 33] for hidden layers. Leaning on the original implementation [18], we use the Gaussian error linear unit activation function [34] for the transformers.
It is important to note that our sequence prediction models do not train on their own generated data which may improve results (this could be done with a multitude of different steps into the future). This is a large simplification heavily influencing any prediction results on predicted data. As all the errors for each prediction will accumulate, results will suffer accordingly.

The transformer models suffer from an issue in Transformers.jl [35] due to the way batching works for these models (fixing this was out of the scope of this work): We obtain a complete level matrix, a singular input, by concatenating the preprocessed columns in the second dimension; we then concatenate multiple of these matrices in the third dimension to get a batch of inputs. As neither Julia nor cuSPARSE provides support for sparse arrays in the third dimension, we are not able to obtain perfect performance for the transformer models (as we mentioned, cuSPARSE could not be used as of yet so this and the following only affects data on the CPU at the moment). This batching issue also has a large effect on memory usage as the non-sparse data takes up a lot of space in high dimensions. For 3-dimensional data, a batch of 32 matrices of maximum length (512 columns) would take $58438 \cdot 32 \cdot 512 \cdot 2$ bytes $\approx 2 \cdot 10^9 = 2$ GB of data (the first summand is the size of each sequence element) (we need 2 bytes per value due to the constant metadata inputs containing 16-bit values). Mapping this amount of data to a GPU that already contains a large model is only possible in rare cases, so reducing batch size, thereby decreasing efficiency, is the only option. In the future, we will implement a wrapper around sparse matrices to act as a 3-dimensional array during computation for transformer models. Even with these issues, our transformer models still mostly kept up with or even outperformed the LSTM stacks, though this may be due to inexperience on our side in choosing the correct amount of parameters.

We will now list the configuration options for our sequence prediction models. As already mentioned, we provide convenience functions for model creation for each training data dimensionality. As these are only for convenience, they have some limitations[10].

LSTMs are built with the following parameters:

- amount of hidden layers
- amount of hidden neurons per hidden layer
- optional skip (or "shortcut") connections [36] from each hidden layer directly to the final activation function
- probability of dropout between every two hidden layers starting after the first one
- final activation function

Our transformer creation functions take the following parameters:

- number of attention heads
- hidden size of each attention head
- hidden size of each fully connected layer
- amount of transformer blocks

[10] Obviously, any unlisted parameters can be changed manually by modifying the source code as well. This also goes for the models in the following sections.

- final activation function of each block
- probability of dropout after attention layers and after each block
- probability of dropout after fully connected layers
- final activation function

The random model has a single parameter – its activation chance. By default, it is set to the mean over the dataset for the appropriate dimensionality (meaning the loss is optimal).
The input and output sizes are also adjustable for each model.

Having described the implementation details for our sequence prediction models, we will now do the same for our GANs.

3.4 *First Screen Generation*

In the following, we will describe implementation details of our GANs, which each include a separate discriminator and generator model. At first, we will only focus on DCGAN-based models.

First off, as mentioned in section 2.2.4, we defined convolutional layers without bias which were not included in Flux.jl [37] at the time of writing.
Our 1-dimensional GANs automatically adjust their layers to the input size. The GANs for training data in 2D and above are implemented with manually chosen stride, padding and dilation so the output is the same size as the first screen of the level. The input to the discriminators of our GANs is the first screen of the level (a vector of 16 elements in the 1-dimensional case, a 27×16 matrix in the 2-dimensional case and a $27 \times 16 \times l$ cube in the 3-dimensional case, where l is the amount of layers). Unlike our sequence models, neither the discriminators nor the generators receive any constant inputs of metadata (as the metadata is generated from the GANs' outputs).

We use the same abstract model class for our GANs that was used for the sequence prediction models, but extend it with two new classes for discriminator and generator models. With this setup, we hope to make extending the GAN pipeline with new models as simple as the sequence prediction pipeline. Unlike the sequence predictors, discriminator models do not allow inputs other than those of the form $r \times c \times l \times b$ or $c \times l \times b$

(for the 1-dimensional case), where r is the amount of rows, c the amount of columns, l the amount of layers (channels in image processing) and b the batch size. We provide layers to easily convert from this format to a matrix and back (the matrix format is required for fully connected layers). Generator models only take a vector of latent noise of a size determined during model creation. This vector is only given as a matrix due to batching (the model handles this automatically).

The first convolutional layer of the discriminator is not followed by a normalization layer; we follow the example of [38] and [21]. By default, the discriminators use the leaky rectified linear unit activation function [33, 39] for hidden layers while the generators use the standard rectified linear unit activation [32, 33].

We will now describe the differences of the fully connected GAN to the DCGANs. The fully connected GANs automatically convert the inputs from the batched form explained above to a matrix and back internally. They also do not use batch normalization but instead have a dropout layer every second layer starting from the first hidden layer onwards (just like for the LSTMs but with fully connected layers instead of LSTM layers; see section 3.3 for a more detailed description).

An improvement we implemented common to all Wasserstein GANs is the usage of custom CUDA GPU kernels to greatly speed up clamping of the parameters of the discriminators.

Our DCGANs have the following parameters for their convenience constructors (these are once again the same for the DCGAN and Wasserstein DCGAN but with different defaults). It should also be noted that while kernel size is adjustable, this requires building new models as the expected sizes will be different. For the discriminators:

- amount of features in each convolutional layer's kernel; the features per successive layer are then multiplied by successive powers of two (starting from 2^0 up to 2^{k-2}, where k is the amount of convolutional layers[11])
- normalization layer after each convolutional layer
- activation function after each normalization layer
- final activation function

[11] We get $k-2$ because we start counting powers from zero and because the final convolutional layer shrinks the number of features to one.

- kernel window size

For the generators:

- amount of features in each convolutional layer's kernel; the features sizes follow the same rules as the discriminator but in reverse (starting from 2^{k-2} and shrinking towards 2^0, where k is the amount of convolutional layers)
- size of latent noise vector (the input size)
- normalization layer after each convolutional layer
- activation function after each normalization layer
- final activation function
- kernel window size

Finally, our fully connected GANs have the same parameters as the LSTMs in section 3.3 except that the final activation function can also be specified. Furthermore, the generator accepts the size of the latent noise vector as well.
The input and output sizes are also adjustable for all model types.

After the GANs, the metadata predictor follows in our pipeline. That model will be described in the next section.

3.5 *Predicting Metadata*

This section lists implementation details for the final type of model in this thesis: the metadata predictor. As mentioned in section 2.2.3, just like for GANs, there are convolutional and fully connected models. These models receive the same inputs as the discriminators described in section 3.4. Similarly, the MLP-based model correctly reshapes its inputs.
The convolutional metadata predictors were all manually created due to the mix of convolutional and pooling layers. We use maximal pooling layers, each followed by a dropout layer, throughout the network but opt for average pooling as the last pooling operation before the fully connected output layer.

Our models give as output the constant part of the inputs supplied to the sequence predictors. Additionally, they output the bit which indicates whether the level ends at this screen (also an input to the sequence predictor; explained in section 3.3).

The convolutional metadata predictors have the following parameters:

- starting amount of features in each convolutional layer (these are multiplied by powers of two, first increasing, then decreasing. Due the models being hand-crafted, there is no clear formula; we therefore suggest a curious reader to take a look at the source code.)
- probability of dropout
- final activation function
- kernel window size

The parameters of the MLP-based metadata prediction models are the same as those for the MLP-based discriminators in section 3.4. As for all models, both the input and output size may be adjusted as well.

We have now not only described our methods in theory but also laid them out practically, showing which options a casual user has and how they affect the models internally. Next, we will combine the methods to get our level generation pipeline.

3.6 *Generating Levels*

With the GAN outputs and the following metadata, the sequence predictors have all the information they would get from a "real" level. Applying the models in sequence is one step; then we need to reverse the whole preprocessing pipeline, converting our abstract representation back to the correct formats that Lunar Magic can write into the ROM. In this section, we will step through this pipeline one by one. Once again, remember the high-level overview shown in figure 5 on page 21.

We start with random input of the correct size for our generator, apply it and get our first screen. At this point, we could also apply our discriminator and retry generating the first screen if the emitted value is too low. We have not implemented this feature.

The first screen is then fed into the metadata predictor, giving us the generated level's metadata. This constant input is then combined with the level in the correct specification, depending on whether the sequence predictor accepts columns or singular tiles, and all but the last sequence elements have their "sequence has not ended yet"-bit set to 1.

The data is now in the same format as a preprocessed real level and we apply the sequence prediction model sequentially until we either find that the "sequence has not ended yet"-bit is 0 or until the maximum level length is reached. This complete output is then postprocessed. Note that we decided against rounding during all of these steps, making generated data unlike any training data. With this, we hope to enable an even larger space for generation. Rounding in-between each step will be made optional in the future as the current behaviour may not be desired (as rounding is an intuitive simplification during generation).

As we skipped some metadata when preparing our data, we need to fill the omissions with default values. For all unknown data, we chose either the defaults of level 261, the first level Lunar Magic presents to the user, or give "empty" values if possible. Before we give an example, remember that Lunar Magic dumps sprite data in a ".sp" file containing the sprite header and then data of individual sprites in the level until a designator for the ending is encountered. If our model does not generate sprites, we don't write back the sprites in level 261 but instead set the sprite header to that of level 261 but do not give any sprites themselves. While the sprite header is required, sprites are not.

Generation functions include generating a single level or multiple levels. A function to generate a whole hack of levels (meaning 512 different levels, each with a different number) is planned. We also implement model-specific generation functions. For the sequence prediction model, functions may generate levels like the above functions or based on predicting on the original game's levels. The GANs may also generate only the first screen for a level independently. This function also exists for the metadata predictor but we have not implemented writing the predicted data back to the level as of yet.

Note that during all of this, the user does not have to worry about dimensionality in any way; all of this is handled automatically if the model was supplied the correct parameters (such as its dimensionality). We allow writing back 1-dimensional outputs by hardcoding the default row the generated output should be placed in and by filling the rest of the level with empty tiles.

We will now dive into implementation details and design decisions regarding our training pipeline, the most important part of our program that first enables our models to generate meaningful outputs.

3.7 *Training Pipeline*

Our training pipeline for each type of model consists of two major parts: the data iterator which reads and preprocesses our data and the actual training loop acting on the data we obtain from the iterator.
Let us first focus on the data iterators and after that on the training loops.

3.7.1 *Data Iterators*

Since the data iterator is responsible for shaping our data to the form expected by the model, we still need to apply some more preprocessing to the already preprocessed data contained in the database. As this may be a large bottleneck, we focused on optimizing data iteration (on any Julia version 1.3 or later, this bottleneck can mostly be avoided by simply using enough data iterator threads). As we explained in section 3.2, our data is stored differently if using the highest compression level. We also explained that storing our 3-dimensional data in "columnar" layers (sliced along the second dimension) is more performant. With the columnar format, we can simply concatenate the individual matrices and obtain our level in a cache-efficient way.
All data iterators use thread-safe channels which the user can take values out of without having to worry about race conditions and synchronicity.

For optimization purposes related to actual training on the data, we minimize the amount of data per batch for models that require padded data. For example, batches for transformer models only contain columns up to the maximum level length per batch instead of up to the maximum level length over all levels (512 columns for horizontal levels).

We do not normalize our data in any way. Normalizing the mean would mean losing the sparsity patterns which improve computational efficiency by a large margin. We decided against normalizing the covariance due to the differences in the way we shape our data for each model and the subsequent influences on unrelated results. First off, we cannot calculate the covariance of the unmodified dataset due to the differences in sequence lengths. While we could easily compute the covariance of our dataset by padding all sequences to the same length, using this result to normalize data for *all* models would affect – for example – sequence models that allow sequences of different lengths in an undesired way. This is because we normalize their input data based on data that is nonexistent in their view (the padded zeros).

With the added randomness of the order the data iterator threads output their data (due to operating system scheduling and other factors – this randomness is not affected by a random number generator seed), we cannot guarantee reproducible results if using threads. While we could adjust the iterators to synchronize so results are perfectly reproducible, we decided to not implement this feature as we assume this would largely hinder efficiency. In the future, an option to enable perfect reproducibility would be a good improvement.
To minimize overhead, we keep each thread running in an endless loop, meaning the user needs to keep track of how many training examples they need for each epoch. In an earlier implementation, an iterable data iterator was used but we decided this minor convenience was not worth having to start multiple threads on each training and test epoch.

The data iterator for first screen data works similar to the other but has fewer options as we do not need to iterate over the first screen but supply it as a whole. With this, we do not need to handle possibly iterating over each tile and – with that option activated – in which direction to iterate (each "columnar" slice from top to bottom or bottom to top). We also have not found the need to worry about padding and supporting different inputs for different models (as of yet). Obviously, we still need to support the different dimensionalities of data.
This data iterator also supplies a tuple of values for simplicity: the first screen and the constant input of each level. This enables us to reuse the iterator for both GAN and metadata predictor training.

3.7.2 *Training Loops*

Our training loops for all model types follow the same schematic:

1. Initialize parameters, data iterators, loggers, optionally load model, . . .
2. Store parameters
3. Initial test
4. Loop over all epochs:

    (a) Train model(s)
    (b) Periodically test model and check for early stopping
    (c) Periodically save model checkpoint

5. Save model checkpoint
6. Clean up resources

All training loops accept either a model or the path to a model checkpoint, correctly initializing everything from the checkpoint and starting training from the exact point it left off (although this may not always be the exact point due to the randomness in threaded data iterators; we skip as many steps as are saved in the checkpoint). The parameters that are the same over all training loops contain values like the number of epochs, early stopping-related values, which checkpointing method to use (as discussed in section 2.2.4, we support saving as BSON or JLD2), after how many training steps to log and save the model, the amount of data iterator threads, the random seed and others.
The parameters are collectively combined with parameters stored in the model (an optional specification of our abstract model interface) and saved in a JSON file to combine both human readability and parsing.

To make debugging model prototypes easier, we implement the `overfit_on_batch` parameter for all training loops, allowing the user to overfit the model on a small batch of a default or chosen size, thereby catching most early errors.

We already mentioned in section 3.3 that our sequence prediction models implement a "soft" loss function; these are also selected as a training parameter.
Due to the iterator allowing it, the GAN training loop allows training both a GAN and a metadata predictor at the same time.

4 EXPERIMENTS

For our experiments, we used cluster computers available to us thanks to Bielefeld University; therefore we cannot make exact claims as to which CPUs or GPUs we used.
Due to time and resource constraints, we were not able to run experiments over all dimensions. We also decided against training the models for longer than 10 hours on a single experiment. The time limit was lifted to 24 hours for experiments in the 3-dimensional case. With these restrictions, sequence prediction models may not have finished training or converged (we set a maximum of 1 000 epochs for all models; non-sequence predictors trained much faster and never met the time limit). In the case of 3D data with only all tile layers (`3dtiles`), the sequence prediction models were not able to train 4 complete epochs in the 24 hour time frame. The transformer in the same dimensionality was even slower as we had to reduce the batch size to 1 due to the large memory usage we explained in section 3.3[12]. From the trained models, we take the checkpoint with the lowest test loss except for the GANs for which we take the checkpoint with the highest discriminator test loss. The databases we used all consisted of tile layers only.

We are also not going to take a look at every model in higher dimensions; internal tests have shown which models work and which don't. While we *will* contrast LSTMs and transformers, we are going to only present select models for the other cases. Another reason is that we do not wish to waste precious public time on the compute cluster and electricity on models that are very unlikely to achieve good results. The GANs we use throughout are dense Wasserstein GAN and the metadata predictors are dense models.

We used the default parameters of each model over all experiments. The most important parameters are listed in tables 1, 2, 3 and 4. For the other values, we point an interested reader to the source code.
The same goes for the training pipelines' parameters, although we will list these without a table: as mentioned, all pipelines train up to 1 000 epochs and seed their global random number

[12] We could have implemented batch sizes of 1 to not be batched but instead remain as a singular input (keeping sparsity) to improve performance for this special case. However, we decided it was not worth it; we mentioned our plans for 3-dimensional sparse arrays fixing most batch-related issues all at once in section 3.3.

	lstm1d	lstm2d	lstm3dtiles
hiddensize	32	32	64
num_hiddenlayers	1	2	2
p_dropout	0.05f0	0.05f0	0.10f0
skipconnections	false	false	false

Table 1: Selected model parameters of the LSTMs; parameter names on the y-axis versus model constructors on the x-axis.

generator with 0. The sequence prediction pipeline used a learning rate of 0.0002 and batch size of 32 with "hard" loss and a mean square error [40] criterion. The GAN pipeline used a non-default learning rate of $5 \cdot 10^{-5}$ and the RMSProp [41] optimizer for both models as well as a batch size of 32 (remember that the Wasserstein GAN uses no criterion for its loss calculation). The discriminator has no prior warm-up and trains for one step for each steps of the generator (so the same amount). The image processing pipeline also used a non-default learning of $5 \cdot 10^{-5}$, the standard batch size 32 and a mean square error criterion. For generation, we also set the random seed to 0. In the next sections, we first look at levels generated by prediction only. After that, we present the combined pipeline. The levels we will predict on stay fixed for all experiments; they are levels 257 up to 263 from the original game. Figure 7 presents the first two screens of each of these (for level 261, refer to figure 3 on page 9). We make an exception for level 260 and only give its first column as input. For the complete pipeline generation, for each experiment, we are going to generate 16 levels and then analyze the first ten levels with unique numbers that we generate (we never generated fewer than ten unique numbers).

Earlier tests showed that a larger amount of parameters increased loss in general and did not lead to decreasing it even after many epochs. Therefore, with regards to the amount of parameters, we are mostly confident in our choices. However, due to inexperience with the transformer models, we may have chosen bad values.

	transformer1d	transformer2d	transformer3dtiles
num_heads	8	8	8
attnhiddensize	4	8	16
ffhiddensize	8	16	32
num_layers	2	2	3
p_dropout_attn	0.05f0	0.05f0	0.10f0
p_dropout_ff	0.05f0	0.05f0	0.05f0

Table 2: Selected model parameters of the transformers; parameter names on the y-axis versus model constructors on the x-axis. "attn" is short for attention, "ff" for feedforward.

4.1 *1-Dimensional*

We trained on a database that did *not* contain squashed data but where we instead took the row containing the maximum amount of values over all rows (remember figure 6 on page 27).

The LSTM constantly predicts the same tile it saw last (especially noticeable for level 257 which ends with an empty tile). The exception is level 260 for which the model predicts empty tiles when the first screen ends (remember we only gave the first column for this one).

While only changing its generations a bit, the transformer learned to sometimes first generate empty tiles and then start the same constant prediction we saw for the LSTM case. The transformer also did not start predicting emptiness for level 260. What we mentioned above happened for levels 257 and 258. We show the first two generated screens of level 258 in figure 8 (for level 257, only one column after the first screen was incorrectly predicted to be zero).

The 1-dimensional pipeline with the LSTM yielded mostly unusable levels but interesting insights: The GAN created a completely empty level and others that only contained one ground tile. The metadata predictor constantly gave the same metadata for these different generations. Finally, one level was generated that seems to show signs of progress: in figure 8 you see a difficult first screen for which the LSTM did not constantly predict the same column. Also, everything seen is the whole level – the LSTM correctly ended the level itself.
With the transformer, the standalone generations were not as successful; it constantly predicted ones with no exceptions.

	densewsgan1d	densewsgan2d	densewsgan3dtiles
hiddensize	32	64	128
num_hiddenlayers	3	4	4
clamp_value	0.01f0	0.01f0	0.01f0
generator_inputsize	32	96	256
p_dropout	0.10f0	0.10f0	0.10f0

Table 3: Selected model parameters of the GANs; parameter names on the y-axis versus combined constructors on the x-axis (these constructors do not actually exist; they are actually separate and substitute `gan` with `discriminator` or `generator` as appropriate). With the exception mentioned below, values apply to both constructors. Clamp values are unique to discriminators and generator input sizes are unique to generators, therefore these only apply to the appropriate model.

	densemeta1d	densemeta2d	densemeta3dtiles
hiddensize	32	32	64
num_hiddenlayers	1	2	2
p_dropout	0.05f0	0.10f0	0.10f0

Table 4: Selected model parameters of the metadata predictors; parameter names on the y-axis versus model constructors on the x-axis (we omitted `predictor` in the constructor names for conciseness).

4.2 *2-Dimensional*

In the 2-dimensional case, the LSTM made a solid line of whatever the last column of the input screen was; just like in the 1-dimensional predictive case.
The transformer started adding (jumpable, although sometimes difficult) holes after some time for most level. This was sometimes followed by empty predictions. Only level 102 was a failed generation as solely empty columns were predicted. We show the predictions for level 260 in figure 9.

Sadly, the generator in the 2-dimensional case only generated empty first screens; the models then also only predicted emptiness. We then used another checkpoint, the latest checkpoint of the generator. As this resulted in segmentation faults with Lunar Magic for the first level we generated, we then used another checkpoint with high discriminator test loss. Still, we only got empty first screens with no predictions. So the only results we have are empty levels.

4.3 *3-Dimensional, Only Tiles*

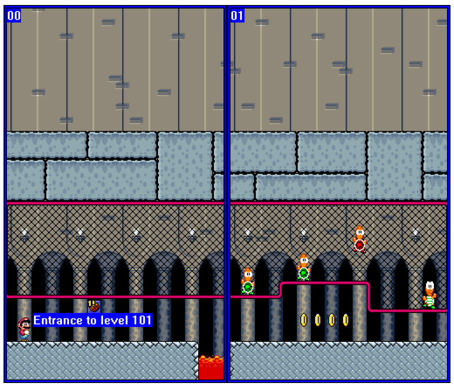
The 3-dimensional models that trained on layers of tiles only were already taking very long (too long) to train. In the 24 hours of training, the latest checkpoint of our LSTM model only had 3 000 training steps. The transformer model managed 27 000 steps, but remember that its batch size was 1 whereas the LSTM trained on batches of size 32. This means that it actually trained on significantly less data than the LSTM. For the sake of presenting results, we will use the LSTM's latest checkpoint. Therefore, note that these results are not comparable as the transformer was hardly trained compared to the LSTM.

Mostly due to its low amount of training, the LSTM we trained generalizes even further than the one from the 2-dimensional case. It mostly generates tiles one ID above empty tiles (with more training, this will most likely converge towards empty tiles – assuming the model continues generalizing). These outputs stayed constant except for level 258 where the model at some points starts to act like the models presented above and outputs columns of only empty tiles.

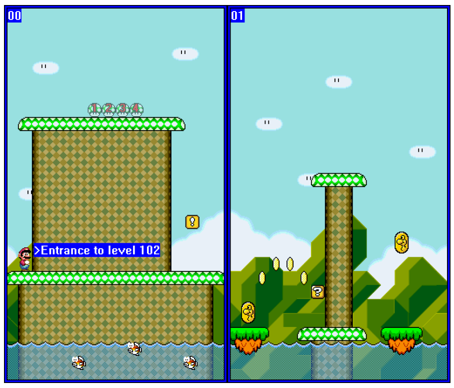
The transformer model converges further towards empty tiles than the LSTM but constantly outputs the same column consisting of a mix of empty and tiles one ID above empty tiles. Just like with the LSTM, except for level 258, these outputs stayed constant. However, the mix clearly shows preference towards the non-empty tiles towards the bottom of the level suggesting the model learned that these tiles were more often "closer" (in terms of the layers) to tiles with higher IDs (like ground tiles) than those further above. Even though and because the transformer only had a fraction of the amount of training the LSTM enjoyed, we would say that its outputs still had a better representation of level data.

Interestingly, for both models, the column from which they start generating empty tiles is at exactly 256 suggesting learning that levels with a similar first screen are usually only half as long as others. In the original game, this level has fewer columns.

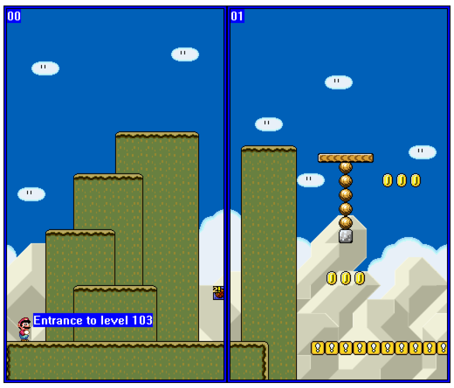
The complete pipeline does not change a lot – the GAN generates first screens that are full of tiles one ID above empty ones. The models then follow the respective behavior we explained above, although we have not seen the exceptional behavior of *not* constantly generating the same column. The metadata predictor has learned to predict the same tileset for all the levels we analyzed and varied Mario's entrance position (we saw "running right", "running left", "vertical pipe up"). The level numbers predicted were mostly different but remained below 256. This makes sense given that the first screens were all very similar (as mentioned in section 3.6, the metadata predictor does not see the same data we see written back as the generated first screen is not rounded).



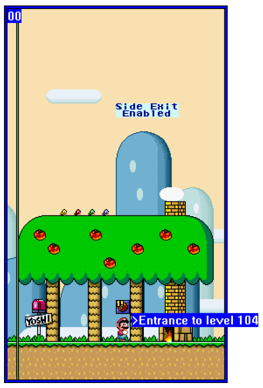
(a) Level 257



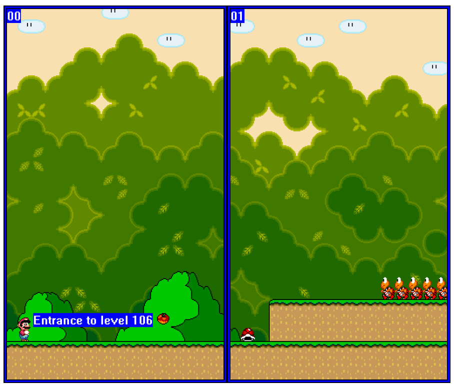
(b) Level 258



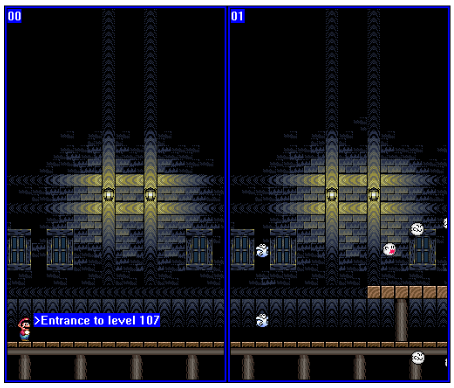
(c) Level 259



(d) Level 260

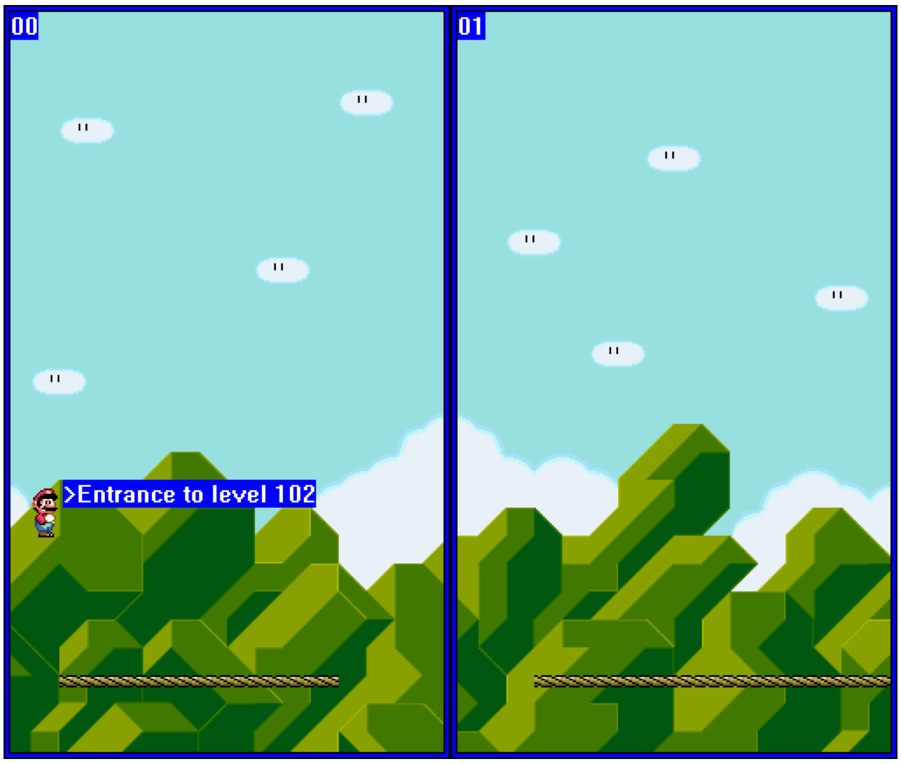


(e) Level 262

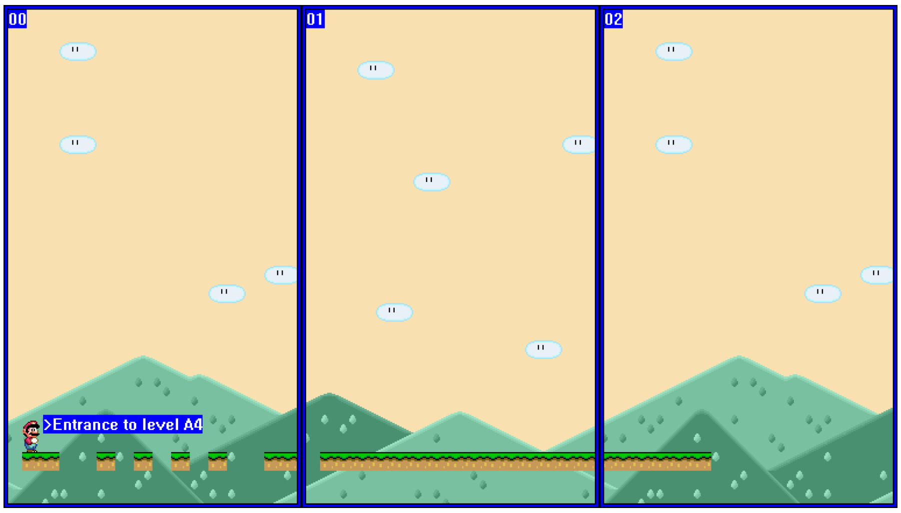


(f) Level 263

Figure 7: The levels we test our sequence prediction models on. For level 260, we only use the first column as input, indicated by the vertical green line.



(a)



(b)

Figure 8: Results in the 1-dimensional case. (a) shows the first two screens of level 258 where the second one was predicted by the transformer model; (b) shows a complete level generated by the pipeline using the LSTM.

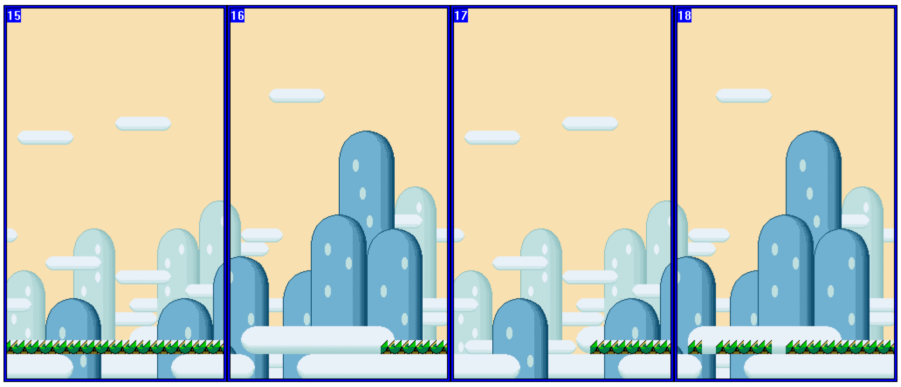


Figure 9: Results in the 2-dimensional case: shown are selected screens near the end of level 260 as predicted by the transformer model. Note that holes were already being generated previously.

5 DISCUSSION

Just as we implemented our pipeline to be as general as possible, our models learned to be as general as possible. The experiments once again show just how well neural networks generalize. In our case, this generalization was a bit too much and we should look for ways to further de-regularize. Formulating the problem as a regression had a large influence on this, though we will discuss this in section 5.1. We also assume that choosing mean square error as the criterion did not penalize layers close to each other enough (leading to models generating "almost empty" tiles).

Due to most of the work relying on the sequence prediction models, these are the weakest link of our pipeline. What we found out is that the models were not able to memorize the data (overfit on it) but instead minimized error by heavily generalizing. While we tried giving the models more layers, this resulted in much higher, oscillating loss that ultimately was not decreasing even after many epochs.

5.1 *Related Work*

Most related work worked with the simpler environment given by Super Mario Bros. [42]. With fewer layers, interactions and general size, this is – while still fundamentally hard – a much simpler problem in relation to the high dimensions we are faced with in Super Mario World. Also, we deliberately chose to include non-functional level parts (non-interactive, cosmetic tiles) in our dataset in the hopes of making generations more like manually created levels. This should have been a later feature instead when a pipeline generating better levels was found; none of the related work included non-functional parts.

We also noticed that a lot of the papers we list here use datasets much smaller than ours (often even using subsets of the 32 levels in Super Mario Bros.), most likely for simplification purposes but possibly also to increase the odds of overfitting which we assume may be desired. Another possibility may be disregard for or unawareness of community-created levels. With the training loop parameter `overfit_on_batch`, originally designed for debugging purposes, we can emulate this behavior. However, we chose not to go this way as we wanted to create diversity. In hindsight, this was probably a bad idea.

A related paper using a column-per-column encoding by Dahlskog et al. [43] uses *n*-grams [44] to predict levels based on the same "style" found in the training dataset, citing the simplicity of *n*-grams in modeling surface-level statistics. While capturing the structure of the training levels well, these models would fail to work with or "invent" never-before seen columns. We hoped to enable this kind of "creative generalization" with our deep learning models.

Geitgey [45] successfully predicts Super Mario Bros. levels by using a char-rnn. His approach encoded the level as a string (with one-hot class encodings, predicting the next tile), reading it out column per column from top to bottom (like we do when reading per tile in default order); his model was also responsible for setting the length of the columns, making his problem space significantly larger. Due to the combinatorial explosion of classes when predicting per column[13] and for efficiency reasons, we had foregone the approach of class encoding initially. However, with the already implemented option of reading per tile and the new-found knowledge of our experiments, we should revisit and implement this approach for future experiments.

Summerville and Mateas [46] analyze several methods of encoding the level, ultimately also deciding that per-tile encodings are easier to learn than per-column encodings. They also propose using "snaking", alternating reading each column from bottom to top, then from top to bottom. This benefits locality in the sequence and artificially doubles the dataset enabled by starting the snaking from either bottom to top or top to bottom. Another proposition is adding another value to the input – a continually increasing value to enable a better understanding of level progression to the model. They also use A* search [47] to enable better playability of the level.
This was not relevant to our case for two reasons: (1) contrary to Super Mario Bros., Super Mario Word allows backtracking, making it impossible to only consider linear paths and (2) due to the largely increased amount of interactions, we cannot assume that tracking a path over tiles leads to the completion of a level. However, the path enabled controlling the difficulty of the level by widening or shrinking the possible paths and lead

[13] With $d = 2^{l \cdot r} \cdot 512 \cdot 32 \cdot 7 \cdot 14 \cdot 8 \cdot 8 \cdot 2 \cdot 2 \cdot 18$, where fixed numbers are the amount of possible values of the constant metadata vector, l is the number of layers, r the number of rows and d the resulting number of classes, for 3-dimensional input columns containing all layers ($l = 2164$, $r = 27$), d has 17 599 *digits* compared to $d = 662$ for $r = 1$ (per-tile input).

to better results; pursuing an approach similar to path search may be worthwhile.

Volz et al. [48, 49] used a Wasserstein DCGAN with gradient penalty together with the covariance matrix adaptation evolution strategy (CMA-ES) [50] to do an evolutionary search over the latent space of the GAN's inputs after training. With this, they can control certain features such as the number of ground tiles and enemies. This in turn enables controlling the difficulty of a generated level by – for example – generating less ground and more enemies to increase the difficulty; this is even applicable gradually. They also used agent-based testing to ensure playability of the levels. Finally, even though their dataset consisted of only a single level, the results show more variability than expected. This variability was a byproduct of using an actual generative approach, making moving through the latent space for individual chunks of the level possible to obtain gradual mutations. While evolutionarily searching the latent space was a great idea, implementing this was out of the scope of this work.

Giacomello et al. successfully used both a unconditional and conditional Wasserstein GAN with gradient penalty [51] to generate DOOM [52] levels. Both GANs take as input image representations of the level (representing platforms, walls, . . .) while the conditional version additionally receives other important information about the level such as metadata (title, author, number of downloads, . . .) or heavily engineered features (size, number of rooms, equivalent diameter of the level volume, . . .). While DOOM is a 3-dimensional first-person shooter, the authors in turn base their work on papers related to platform level generation for Super Mario Bros., completing the cycle. The authors show that a GAN-only approach works well in a complex domain; considering conditional information (by this we mean both GANs receiving prior level information; this is not unique to the conditional GAN) was not in our interest but may be desired to improve results.

5.2 *Future Improvements*

While the experiments have shown that our approach to generating levels via sequence prediction have not worked out yet, we have many suggestions and possible solutions that ought to be tried out in the future in addition to the ones already mentioned in the previous section. We also have several improvements planned for the project which are out of the scope of this thesis. However, we do want to mention that a distributed version would make a lot of sense for larger models; then, training models for the 3-dimensional case with all layers may be feasible.

As is usual in deep learning, for all our models, tuning the hyperparameters further may lead to surprising results. While we do not believe – looking at the experiments and our prior discussion above – that only changing hyperparameters will lead to a good level generator, with deep learning you never know if the combined changes over many parameters may suddenly lead to a good result. In particular, reducing or removing dropout may help.
We also believe the training pipeline in general would benefit from further features such as learning rate regulation (especially for warm-up).

While a lack of features may be influencing our results, a lot of features we implemented could sadly not be tested well due to time constraints. These include the per-tile encoding, "soft" loss and the LSTMs' skip connections.

In terms of the sequence prediction models, first off, we should try training on generated data that goes back some amount steps into the past instead of only on "perfectly" predicted data – the actual, ground truth data (we mentioned this in section 3.3). This would allow minimizing errors for future predictions. As we see, we cannot rely on the model to predict the next column perfectly, meaning an error for each predicted column will accumulate and result in worse and worse predictions.

The high-dimensional experiments showed how the transformer with less training had a – in our opinion – better representation of the data than the LSTM. This is also a form of de-regularization and might be worth exploring.

We could try implementing more loss functions akin to the "soft" loss we introduced in section 3.3 as well. Another promising improvement would be implementing adversarial sequence generators [53, 54] which we had initially foregone due to considerations regarding the high dimensionality of our data. With those, the GANs and – possibly – the metadata predictor become redundant, leading us to a single model capable of generating complete levels from scratch.

6 APPENDIX

Here, we will list some more details that were not out of the scope of the thesis but more intended for a reader interested in further programming and design aspects. We will first go more in detail on the Julia packages used throughout the thesis and then describe the model interface.

6.1 *Packages*

Let us now list the packages (excluding Julia's large standard library) most important for our preprocessing and training pipeline. While this list deliberately does not cover all packages used, we hope to give an overview of what the ecosystem made or will make possible, why these packages were important for us and what issues we still had to solve manually.

We encountered several issues with model checkpoints. We first switched from an unmerged pull request branch of BSON.jl [55, 56] (which contains an issue only later addressed in the master branch) to the HDF5-based JLD.jl [57], then to the more unstable JLD2.jl [58] due to not being able to save functions in JLD.jl. With JLD2.jl, there were still errors which led us to also check out a pull request for this package [59] (this has been merged but for stability reasons, we are not ready to switch to the master branch of JLD2.jl just yet). Finally, we settled with JLD2.jl for our model checkpoints as the most stable package. While the files are not as convenient as those of BSON.jl, we are sure the stability is worth it. In the end, we support saving to either BSON.jl or JLD2.jl with JLD2.jl as the default.

For the databases containing our training data, we used JuliaDB.jl [60] as it supports parallel and out-of-core processing of the database. While these features have not been used yet, they should future-proof our program for cluster computing and databases that do not fit into memory on a host PC. Smaller statistics (especially pre-computed ones that we load to reduce startup time) were saved as CSV files which CSV.jl [61] was a great fit for.

For our machine learning needs, we used Flux.jl [37]. While Flux.jl is one of the most mature deep learning frameworks in Julia, it still missed some basic functionality. For example, we had to define our own convolutional layer without bias as it had not been included at the time of writing. With Zygote.jl [62],

a framework supporting source-to-source automatic differentiation that is to be included in Flux.jl, we hope to gain a large speed boost in the future with compiled derivatives as they will not have to be tracked at runtime.
Due to Flux.jl's GPU support enabled by CuArrays.jl [63], we were able to achieve great performance even while writing specialized loss functions. We were sadly not able to support the CUDA library for sparse arrays, cuSPARSE; due to the CuArrays.jl package not supporting all the features necessary for us (for example simple primitives like array slicing) for cuSPARSE arrays, we had to comment out the few lines pertaining to sparse CUDA arrays at the moment. These issues were not in the scope of this thesis but will be addressed at a later point, surely granting another large speedup to any GPU-run training pipeline that uses sparse matrices (the training pipeline is explained in section 3.7). Another, more low-level GPU package, CUDAnative.jl [64], allowed us to write our own CUDA kernels with ease. This enabled faster clamping of the parameters of our Wasserstein GANs.
The Transformers.jl [35] package was essential in getting the GPT-2 transformer model to run. While the model is not included in the package as of the writing of this thesis, the given building blocks were a large help.
Finally, TensorBoardLogger.jl [65] enabled a great tool for monitoring training progress: TensorBoard [66].

6.2 *Model Interface*

While the source code already gives a good documentation of the model interface, for completeness sake, we will give a brief overview here as well. Our model interface `LearningModel` is an abstract class that implements several methods that may be overwritten. For a class `YourModel` to be a `LearningModel`, the model needs to (1) be defined as a `Flux.@treelike` (in future versions as a `Flux.@functor`; this is simply a line consisting of `Flux.@treelike YourModel`) to enable parameter tracking and several convenience methods. It needs to (2) implement a field `hyperparams` of type `Dict{Symbol, Any}` which may hold any parameters that discern the model from a model of the same type (these are optional). (3) The `hyperparams` dictionary has one required key, the `:dimensionality` of the model as a `Symbol` so data may be correctly shaped for the model (this is any of the types of training data we listed in section 3.2).

(4) A `LearningModel` model for sequence prediction has to implement the following functions:

- `(model::YourModel)(input)`: the application function to enable calling the model on input data.
- `makeloss(model::YourModel, criterion)`: return a 2-argument loss function by applying the given criterion (for example, a criterion is `Flux.mse`).
- `dataiteratorparams(model::YourModel)`: return parameters for the data iterator (does the model expect data as a vector of vectors, in matrix form, as a joined vector, . . .).
- `calculate_loss(model::YourModel, loss, input, target)`: return the loss obtained by applying the loss function to the given input with the given target (this is mostly the function returned by `makeloss`, but with optional preparation of the model).
- `step!(model::YourModel, parameters, optimizer, loss, input, target)`: update the given model with a single training step on the given input with the given target and return the loss; a single training step.
- (optionally) `makesoftloss(model::YourModel, criterion)`: like `makeloss` but return a "soft" loss function penalizing wrong predictions less (this can of course just be used as any other alternative loss function).

GANs and metadata predictors do not need to implement `dataiteratorparams` and the optional `makesoftloss` as they are not used. However, some functions expect different arguments. For these, we refer a curious reader to the source code.
Finally, the generator models of GANs are required to save the input size of their latent vectors under the key `:inputsize`.

REFERENCES

1. *Face2Face: Real-Time Face Capture and Reenactment of RGB Videos (CVPR 2016 Oral)* `https://www.youtube.com/watch?v=ohmajJTcpNk` (2019).
2. Wang, X. *et al.* ESRGAN: Enhanced Super-Resolution Generative Adversarial Networks. arXiv: `1809.00219 [cs]`. `https://arxiv.org/abs/1809.00219` (2019) (Sept. 17, 2018).
3. in. *Wikipedia* Page Version ID: 925939233 (Nov. 13, 2019). `https://en.wikipedia.org/w/index.php?title=Turing_test&oldid=925939233` (2019).
4. Radford, A. *et al.* Language Models Are Unsupervised Multitask Learners.
5. in. *Wikipedia* Page Version ID: 922683724 (Oct. 23, 2019). `https://en.wikipedia.org/w/index.php?title=Super_Mario_World&oldid=922683724` (2019).
6. Vaswani, A. *et al.* Attention Is All You Need. arXiv: `1706.03762 [cs]`. `https://arxiv.org/abs/1706.03762` (2019) (June 12, 2017).
7. Hochreiter, S. & Schmidhuber, J. Long Short-Term Memory. *Neural computation* **9,** 1735–80 (Dec. 1, 1997).
8. in. *Wikipedia* Page Version ID: 917083876 (Sept. 22, 2019). `https://en.wikipedia.org/w/index.php?title=History_of_games&oldid=917083876` (2019).
9. in. *Wikipedia* Page Version ID: 920159762 (Oct. 8, 2019). `https://en.wikipedia.org/w/index.php?title=Nash_equilibrium&oldid=920159762` (2019).
10. in. *Wikipedia* Page Version ID: 920950695 (Oct. 12, 2019). `https://en.wikipedia.org/w/index.php?title=Video_game_industry&oldid=920950695` (2019).
11. Alcaro. *Alcaro/Flips* Nov. 18, 2019. `https://github.com/Alcaro/Flips` (2019).
12. *Floating IPS (Flips) v1.31 - Tools - SMW Central* `https://www.smwcentral.net/?p=section&a=details&id=11474` (2019).
13. *FuSoYa's Niche - Lunar Magic SMW Editor Introduction* `https://fusoya.eludevisibility.org/lm/index.html` (2019).

14. *SMW Central - Your Primary Super Mario World Hacking Resource* https://www.smwcentral.net/ (2019).

15. in. *Wikipedia* Page Version ID: 919682616 (Oct. 5, 2019). https://en.wikipedia.org/w/index.php?title=Super_Metroid&oldid=919682616 (2019).

16. Goodfellow, I. J. *et al.* Generative Adversarial Networks. arXiv: 1406.2661 [cs, stat]. https://arxiv.org/abs/1406.2661 (2019) (June 10, 2014).

17. Andrej. *Karpathy/Char-Rnn* Nov. 5, 2019. https://github.com/karpathy/char-rnn (2019).

18. *Openai/Gpt-2* OpenAI, Nov. 5, 2019. https://github.com/openai/gpt-2 (2019).

19. Radford, A., Metz, L. & Chintala, S. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. arXiv: 1511.06434 [cs]. https://arxiv.org/abs/1511.06434 (2019) (Jan. 7, 2016).

20. Arjovsky, M., Chintala, S. & Bottou, L. Wasserstein GAN. arXiv: 1701.07875 [cs, stat]. https://arxiv.org/abs/1701.07875 (2019) (Dec. 6, 2017).

21. martinarjovsky. *Martinarjovsky/WassersteinGAN* Nov. 14, 2019. https://github.com/martinarjovsky/WassersteinGAN (2019).

22. in. *Wikipedia* Page Version ID: 923044913 (Oct. 26, 2019). https://en.wikipedia.org/w/index.php?title=Multilayer_perceptron&oldid=923044913 (2019).

23. Kramer, M. A. Nonlinear Principal Component Analysis Using Autoassociative Neural Networks. *AIChE Journal* **37,** 233–243. ISSN: 1547-5905. https://aiche.onlinelibrary.wiley.com/doi/10.1002/aic.690370209 (2019) (Feb. 1, 1991).

24. Kingma, D. P. & Welling, M. Auto-Encoding Variational Bayes. arXiv: 1312.6114 [cs, stat]. https://arxiv.org/abs/1312.6114 (2019) (May 1, 2014).

25. in. *Wikipedia* Page Version ID: 924850099 (Nov. 6, 2019). https://en.wikipedia.org/w/index.php?title=Autoencoder&oldid=924850099 (2019).

26. Makhzani, A., Shlens, J., Jaitly, N., Goodfellow, I. & Frey, B. Adversarial Autoencoders. arXiv: 1511.05644 [cs]. https://arxiv.org/abs/1511.05644 (2019) (May 24, 2016).

27. Bezanson, J., Edelman, A., Karpinski, S. & Shah, V. B. Julia: A Fresh Approach to Numerical Computing. *SIAM Review* **59,** 65–98. ISSN: 0036-1445. https://epubs.siam.org/doi/10.1137/141000671 (2019) (Jan. 1, 2017).

28. *CUDA Toolkit* https://developer.nvidia.com/cuda-toolkit (2019).

29. *NVIDIA cuDNN* https://developer.nvidia.com/cudnn (2019).

30. *Git* https://git-scm.com/ (2019).

31. *cuSPARSE* https://developer.nvidia.com/cusparse (2019).

32. Nair, V. & Hinton, G. E. Rectified Linear Units Improve Restricted Boltzmann Machines.

33. in. *Wikipedia* Page Version ID: 923288576 (Oct. 27, 2019). https://en.wikipedia.org/w/index.php?title=Rectifier_(neural_networks)&oldid=923288576 (2019).

34. Hendrycks, D. & Gimpel, K. Gaussian Error Linear Units (GELUs). arXiv: 1606.08415 [cs]. https://arxiv.org/abs/1606.08415 (2019) (Nov. 11, 2018).

35. Peter. *Chengchingwen/Transformers.Jl* Nov. 14, 2019. https://github.com/chengchingwen/Transformers.jl (2019).

36. He, K., Zhang, X., Ren, S. & Sun, J. Deep Residual Learning for Image Recognition. arXiv: 1512.03385 [cs]. https://arxiv.org/abs/1512.03385 (2019) (Dec. 10, 2015).

37. *FluxML/Flux.Jl* Flux, Nov. 14, 2019. https://github.com/FluxML/Flux.jl (2019).

38. *Pytorch/Examples* pytorch, Nov. 14, 2019. https://github.com/pytorch/examples (2019).

39. Maas, A. L., Hannun, A. Y. & Ng, A. Y. Rectifier Nonlinearities Improve Neural Network Acoustic Models.

40. in. *Wikipedia* Page Version ID: 927514373 (Nov. 22, 2019). https://en.wikipedia.org/w/index.php?title=Mean_squared_error&oldid=927514373 (2019).

41. Tielemann, T. & Hinton, G. *Neural Networks for Machine Learning - Lecture 6.5-Rmsprop: Divide the Gradient by a Running Average of Its Recent Magnitude.* https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf (2019).

42. in. *Wikipedia* Page Version ID: 927554124 (Nov. 23, 2019). https://en.wikipedia.org/w/index.php?title=Super_Mario_Bros.&oldid=927554124 (2019).

43. Dahlskog, S., Togelius, J. & Nelson, M. J. *Linear Levels through N-Grams* in *Proceedings of the 18th International Academic MindTrek Conference on Media Business, Management, Content & Services - AcademicMindTrek '14* The 18th International Academic MindTrek Conference (ACM Press, Tampere, Finland, 2014), 200–206. ISBN: 978-1-4503-3006-0. https://dl.acm.org/citation.cfm?doid=2676467.2676506 (2019).

44. in. *Wikipedia* Page Version ID: 927473508 (Nov. 22, 2019). https://en.wikipedia.org/w/index.php?title=N-gram&oldid=927473508 (2019).

45. Geitgey, A. *Machine Learning Is Fun! Part 2* https://medium.com/@ageitgey/machine-learning-is-fun-part-2-a26a10b68df3 (2019).

46. Summerville, A. & Mateas, M. Super Mario as a String: Platformer Level Generation Via LSTMs. arXiv: 1603.00930 [cs]. https://arxiv.org/abs/1603.00930 (2019) (Mar. 2, 2016).

47. in. *Wikipedia* Page Version ID: 925009518 (Nov. 7, 2019). https://en.wikipedia.org/w/index.php?title=A*_search_algorithm&oldid=925009518 (2019).

48. Volz, V. *et al.* Evolving Mario Levels in the Latent Space of a Deep Convolutional Generative Adversarial Network. arXiv: 1805.00728 [cs]. https://arxiv.org/abs/1805.00728 (2019) (May 2, 2018).

49. TheHedgeify. *TheHedgeify/DagstuhlGAN* Aug. 26, 2019. https://github.com/TheHedgeify/DagstuhlGAN (2019).

50. Hansen, N. & Ostermeier, A. *Completely Derandomized Self-Adaptation in Evolution Strategies* https://www.mitpressjournals.org/doix/abs/10.1162/106365601750190398 (2019).

51. Gulrajani, I., Ahmed, F., Arjovsky, M., Dumoulin, V. & Courville, A. Improved Training of Wasserstein GANs. arXiv: 1704.00028 [cs, stat]. https://arxiv.org/abs/1704.00028 (2019) (Dec. 25, 2017).

52. in. *Wikipedia* Page Version ID: 927702319 (Nov. 24, 2019). https://en.wikipedia.org/w/index.php?title=Doom_(1993_video_game)&oldid=927702319 (2019).

53. Yu, L., Zhang, W., Wang, J. & Yu, Y. SeqGAN: Sequence Generative Adversarial Nets with Policy Gradient. arXiv: 1609.05473 [cs]. https://arxiv.org/abs/1609.05473 (2019) (Aug. 25, 2017).

54. Li, Z. *et al.* Adversarial Discrete Sequence Generation without Explicit Neural Networks as Discriminators.

55. *JuliaIO/BSON.Jl* JuliaIO, Nov. 13, 2019. https://github.com/JuliaIO/BSON.jl (2019).

56. Kesler, C. *Saving Array Length for Undef Array Issues by Codyk12 · Pull Request #47 · JuliaIO/BSON.Jl* https://github.com/JuliaIO/BSON.jl/pull/47 (2019).

57. *JuliaIO/JLD.Jl* JuliaIO, Nov. 5, 2019. https://github.com/JuliaIO/JLD.jl (2019).

58. *JuliaIO/JLD2.Jl* JuliaIO, Nov. 10, 2019. https://github.com/JuliaIO/JLD2.jl (2019).

59. Palethorpe, R. *Maybe Fix Typename for Nested Modules by Richiejp · Pull Request #126 · JuliaIO/JLD2.Jl* https://github.com/JuliaIO/JLD2.jl/pull/126 (2019).

60. *JuliaComputing/JuliaDB.Jl* Julia Computing, Inc., Nov. 12, 2019. https://github.com/JuliaComputing/JuliaDB.jl (2019).

61. *JuliaData/CSV.Jl* Julia Data, Nov. 13, 2019. https://github.com/JuliaData/CSV.jl (2019).

62. *FluxML/Zygote.Jl* Flux, Nov. 14, 2019. https://github.com/FluxML/Zygote.jl (2019).

63. *JuliaGPU/CuArrays.Jl* JuliaGPU, Nov. 14, 2019. https://github.com/JuliaGPU/CuArrays.jl (2019).

64. *JuliaGPU/CUDAnative.Jl* JuliaGPU, Nov. 14, 2019. `https://github.com/JuliaGPU/CUDAnative.jl` (2019).

65. Vicentini, F. *PhilipVinc/TensorBoardLogger.Jl* Oct. 24, 2019. `https://github.com/PhilipVinc/TensorBoardLogger.jl` (2019).

66. *TensorBoard* `https://www.tensorflow.org/tensorboard` (2019).