Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Cybernetics

Automatic Game Level Generation

Bachelor thesis

Vladyslav Yazykov

Study program: Open Informatics
Branch of study: Artificial Intelligence and Computer Science
Supervisor: Ing. Jaromír Janisch

Prague, August 2021

Thesis Supervisor:
Ing. Jaromír Janisch
Department of Computer Science
Faculty of Electrical Engineering
Czech Technical University in Prague
Karlovo náměstí 13
121 35 Praha 2
Czech Republic

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Yazykov Vladyslav**	Personal ID number:	**452777**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Cybernetics**		
Study program:	**Open Informatics**		
Specialisation:	**Artificial Intelligence and Computer Science**		

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Automatic Game Level Generation

Bachelor’s thesis title in Czech:

Automatické generování herních úrovní

Guidelines:

The common task in computer games is a level generation of the right difficulty. This thesis will explore the ways of automatic level generation with difficulty that should accommodate to the current player, using advanced machine learning methods.
Objectives:
- review the current research on automatic level generation in computer games
- implement a simple variable single-player computer game as a playground environment
- based on the research, design an automatic level generation technique with varying difficulty
- implement an machine learning based agent to solve different environments
- evaluate the level generation approach with agents of different skill level

Bibliography / sources:

[1] Summerville, Adam, et al. "Procedural content generation via machine learning (PCGML)." IEEE Transactions on Games 10.3 (2018): 257-270.
[2] Di Liello, Luca, et al. "Efficient Generation of Structured Objects with Constrained Adversarial Networks." Advances in Neural Information Processing Systems 33 (2020).
[3] Dennis, Michael, et al. "Emergent Complexity and Zero-shot Transfer via Unsupervised Environment Design." Advances in Neural Information Processing Systems 33 (2020).
[4] Wang, Rui, et al. "POET: open-ended coevolution of environments and their optimized solutions." Proceedings of the Genetic and Evolutionary Computation Conference. 2019.

Name and workplace of bachelor’s thesis supervisor:

Ing. Jaromír Janisch, Artificial Intelligence Center, FEE

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **10.01.2021** Deadline for bachelor thesis submission: **13.08.2021**

Assignment valid until: **30.09.2022**

Ing. Jaromír Janisch	prof. Ing. Tomáš Svoboda, Ph.D.	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student's signature

Author statement for undergraduate thesis:

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, date 13.8.2021. ....
Vladyslav Yazykov

Prohlášení autora práce

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 13.8.2021. ....
Vladyslav Yazykov

Abstract

This work explores the novel (as to our knowledge) formulation of procedural level generation (PLG) as a task of learning a mapping from the difficulty to the level embedding space. Our level generator takes a difficulty input in a $[0, 1]$ range and outputs a level embedding of that difficulty. Further on, we extend the task to generate a variety of different levels within the same difficulty. We solve the problem using an N-dimensional seeded generator, that searches for the correct (having the required difficulty) embedding in a close proximity to the specified (seed) embedding.

We explore the meaning of difficulty, possible definitions and interpretations. We then introduce several possible ways of evaluating the difficulty of a level using agents that sample trajectories in a resettable environment. For simplicity, in this work we use Q-learning agents and policy gradient agents.

The work presents a general framework that is simple to understand and use. Different parts of the framework (e.g. difficulty evaluation) can be extended by more sophisticated algorithms, that could increase the overall quality of the generated levels, as well as improve the computational efficiency. Thus, this work represents a foundation for further research in game level generation, difficulty evaluation, comparison of different difficulty evaluation methods, and how does evaluation influence level generation.

Keywords: Procedural level generation, game level difficulty estimation.

Tato práce zkoumá novou (podle naších znalostí) formulaci procedurální generování úrovní ve smyslu mapování z prostoru obtížnosti do prostoru úrovně. Náš generátor na vstupu dostává obtížnost v rozsahu $[0, 1]$ a vydává embedding této obtížnosti. Dále rozšíříme úkol tak, aby generator generoval různé úrovně v rámci stejné obtížnosti. Problém vyřešíme pomocí N-dimenzionálního generátoru, který hledá správný (s požadovanou obtížností) embedding v těsné blízkosti zadaného embeddingu.

Zkoumáme význam obtížnosti, možné definice a interpretace. Poté představíme několik možných způsobů vyhodnocení obtížnosti úrovně pomocí agentů, kteří vyhodnocují trajektorie v resetovatelném prostředí. Pro jednoduchost v této práci používáme Q-learning a policy gradient agenty.

Práce představuje obecný framework, který je snadno pochopitelný a použitelný. Různé části rámce (např. vyhodnocení obtížnosti) lze rozšířit o sofistikovanější algoritmy, které by mohly zvýšit celkovou kvalitu generovaných úrovní a také zlepšit výpočetní efektivitu. Tato práce tedy představuje základ pro další výzkum generování herních úrovně, vyhodnocení obtížnosti, srovnání různých metod vyhodnocení obtížnosti a jak vyhodnocení ovlivňuje generování úrovně.

Keywords: Procedurální generování úrovně, odhad obtížnosti herní úrovně.

Acknowledgements

I am grateful to my supervisor Jaromír Janisch who has guided me through the completion of this work, shared his expertise, which has often helped me to get out of the ”local minima” of writing this thesis. Our regular meetings and discussions were invaluable for me to complete this work.

I would like to thank my friends and family for their support, critique, and patience for listening to my explanations of this thesis over and over again, challenging my ideas, helping to refine them, and understand them better.

I would also like to express my gratitude to all the educational resources, that simply exist and have allowed me to learn many machine learning-related topics with fun and ease, as well as getting inspired to create something beautiful.

List of Figures

List of Acronyms

A2C Advantage Actor Critic. 7

CAN Constrained GAN. 7

CESAGAN Conditional Embedding Self-Attention Generative Adversarial Network. 4

DDA dynamic difficulty adjustment. 37

GAN generative adversarial network. 4, 16, 17, 28

GPT-3 Generative Pre-trained Transformer 3. 1

LReLU Leaky ReLU. 30

LSTM long short-term memory. 4

MCC Minimal Criterion Coevolution. 5

MCTS Monte Carlo tree search. 3, 4

ML machine learning. 3

NN neural network. x, 1, 2, 11, 13, 15, 17, 18, 26, 28, 30, 31, 33

PCG procedural content generation. 1, 3

PLG procedural level generation. vi, 1, 11

POET Paired Open-Ended Trailblazer. 5

RNN recurrent neural network. 14

SGD stochastic gradient descent. 13

SL semantic loss. 7

SMB Super Mario Bros.. 3, 4

Contents

Chapter 1

Introduction

In the last years, procedural content generation (PCG) has been receiving more and more attention as a means to reduce the production cost, make games re-playable, and create a possibly infinite amount of playable content. In this work we focus on the specific branch of the PCG, that is concerned with the level/environment generation - PLG.

PLG is mostly used in Rogue-like [1] games: Bindings Of Isaac [2], Spelunky [3], Dead Cells [4], etc., to create a possibly infinite amount of playable game levels, without having to spend hours designing each one individually.

Usually, the generation is guided by a grammar: the program is given a set of rules and a set of starting building blocks, the levels are then generated by applying those rules (usually in a random or pseudo-random way). Sometimes, a mix of hand-crafted and procedurally generation elements is used.

The power of PLG comes at a cost of authenticity when the generated levels look very similar to each other. We hypothesize that using a deep learning generator as an approximation to the human designer will result in a human-like level of creativity while keeping a low production cost - generating a level would mean executing a single forward pass on the NN.

Ideally, we would like our network to have an understanding of the level's difficulty, level diversity, and game style - to match the overall style of the game. In addition, in more complex games, the narrative may be taken into account - for example, the generator would read a script, from which it would create an interactive environment for the player to act in [5]. However, combining all of these tasks into a complex level-generative AI seems a bit unrealistic at the moment. Thus, in this thesis, we focus on two small aspects:

[1]https://en.wikipedia.org/wiki/Rogue_(video_game)

[2]https://en.wikipedia.org/wiki/The_Binding_of_Isaac_(video_game)

[3]https://en.wikipedia.org/wiki/Spelunky

[4]https://en.wikipedia.org/wiki/Dead_Cells

[5]This idea is inspired by the Generative Pre-trained Transformer 3 (GPT-3) [1] being able to generate web-pages, being provided with their textual description

teaching our NN to learn and understand the level's difficulty, as well as requiring it to generate a diversity of different ones.

For simplicity, we will focus on action games with simple mechanics (e.g. Flappy Bird [6]), where the challenge is generally to perform the correct action at the correct moment. These games manage to entertain the player through the challenge by gradually increasing the difficulty by changing the level configuration: e.g., placement of various in-game elements (blocks, enemies, switches, etc.). The game is generally thought to be designed successfully if the difficulty of the game matches the player's skill, so that they feel challenged, but not overwhelmed [2]. This is done in two steps: evaluating the player's skill, and generating the level of the appropriate difficulty. This paper focuses on the latter.

[6]https://en.wikipedia.org/wiki/Flappy_Bird

Chapter 2

Related work

A lot of research and progress has been recently been made in the field of PCG using machine learning (ML) [3], [4]. Here we review the ones most relative to our work, as well as the ones that we drew the most inspiration from.

2.1 With prior data

We start by reviewing supervised learning approaches to game level generation. All of these require previous data of some sort, and, thus, are not suitable for generating a completely new game from scratch. However, we review these techniques as nevertheless being related to the game level generation and the ones that can be used to potentially enrichen the set of generated levels, once the one has been produced.

[5] discussed how to combine various design patterns and procedural content generation to dynamically create game levels for Super Mario Bros. (SMB). Authors explore the usage of patterns as training objectives in [6], utilizing a simple evolutionary algorithm with a fitness function rewarding more interesting (and playable) patterns. Authors continue their work in [7], where single-column micro-patterns are combined into several-columns meso-patterns (requiring or affording some player action - e.g. jumping). Meso-patterns are then combined into bigger macro-patterns - complete level sections of connected meso-patterns. Later, in [8], authors use n-grams of vertical slices the authors construct a table of n-grams and their occurrences. Each n-gram is a sequence of vertical slices of a length n. For each n-gram, the number of occurrences is tracked and then turned into a probability that is used for sampling new levels.

The approach using Markov Chains alongside the Monte Carlo tree search (MCTS) was explored in [9]. The authors used a reward function containing level solvability (playability), the number of gaps, enemies, and collectibles (coins or power-ups) to calculate the score of a node in a tree and apply it in a backpropagation manner. The Markov chain is supplying transmission probabilities that guide the search in more probable directions. MCTS is then pruning those selections, removing the ones that lead to unplayable or otherwise undesirable levels.

[10] uses long short-term memorys (LSTMs) to generate SMB levels. The authors use the individual tiles as the string characters, opposed to the whole vertical slices. The used snaking representation of the level (meaning that the tiles first go from bottom to the top, and the next column has the tiles going from the top to the bottom - or vice versa). The tracking of the player path (using A*) was used to ensure solvability, and tracking the column depth (how far are into a level are we) made LSTM understand the macro-level progression.

An approach of generating game levels from gameplay videos has been explored in [11]. The authors claim that the approach may theoretically learn a genre of games and can be extended to different genres. This needs some footage of the game to be trained from, although once the inner representation is learned, new games may theoretically be created from scratch.

[12] are using the Video Game Level Corpus to generate SMB levels using generative adversarial network (GAN) with Latent Variable Evolution, using Covariance Matrix Adaptation Evolutionary Strategy to evolve the latent vector, obtaining interesting results that have succeeded to capture the level detail from the corpus. While being domain-agnostic, this approach also requires some level data to learn from (thus making it unusable for creating game levels from scratch), and is quality-dependent on the quality of the corpus used. Moreover, the fitness function must make sense and should be carefully designed (the work assumes the correlation between the number of jumps and difficulty).

[13] generates a set of environments using a Conditional Embedding Self-Attention Generative Adversarial Network (CESAGAN). The generated levels that pass the playability check are added back into the training set to generate new playable levels. This allows to achieve good results on a small (~ 5) number of pre-generated levels, and also significantly reduces the number of duplicated levels.

2.2 No prior data

2.2.1 Open-ended approaches

These approaches do not optimize a reward function. Instead, they are aimed at maximizing diversity - generating new, different environments. The agents are usually intrinsically motivated, meaning that they are not given a direct reward from the environment, but instead develop a new skill while trying to explore their environment [14].

Minimal Criterion Coevolution (MCC) [15] simultaneously evolves a population of mazes and a population of agents that try to solve these mazes. The goal of this open-ended search is to produce a diverse set of different mazes and agents that would solve them. The process of maze evolution may be viewed as a level generation process, but its unclear how to evaluate the difficulty of the given level, as the algorithm sidesteps the usage of a fitness function, which is an advantage in the open-ended search, but a disadvantage in our case.

Drawing great inspiration from MCC, Paired Open-Ended Trailblazer (POET) [16] also maintains a set of environments and agents that co-evolve. However, differently from MCC, which only specifies the lower bound for the reproduction of agents and mazes, POET also strives for better performance by optimizing the agents within the corresponding environments. The POET keeps agents and environments in pairs, but regularly performs checks if the skills learned from one agent may prove useful inside a different environment. The paper presents fascinating results, being able to efficiently generate a curriculum that allows training the agents to the high level of performance - unreachable without the introduced transfer technique. From the generative point of view, we can see that, similarly to other approaches, the technique produces a set of different environments of gradually increasing difficulty. Similar to the previous approach, we may store the environments during training and later use them on their own. The pleasant thing about open-ended co-evolving approaches is that they produce diverse environments.

In [17] a single agent with two separate "minds", namely Alice and Bob, plays a game with itself that allows it to learn the rules of the environment. Alice performs a sequence of actions that modify an environment, and Bob is then tasked with either resetting an environment to the starting state (for reversible environments) or repeating Alice's actions as closely as possible. Although the level generation is not the main focus of the paper, we can see how Alice is trained to be a goal generator for Bob - producing goals of appropriate difficulty. By saving Alice's state throughout the training, we can obtain a goal generator that generates goals of gradually increasing difficulty. By taking into account the fact that after her action sequence, Alice effectively generates a new state, we can actually view her as a level generator.

2.2.2 (More like) Self-play

A particularly interesting approach, that is very similar to ours has been introduced by [18]. It discusses a Generative Playing Network that consists of a generator and an agent playing a minimax game, similar to that from GAN [19]. The generator takes a noise input and creates a level, that is evaluated by an actor-critic agent, that returns the utility value U. Generator then minimizes $\mathbb{E}\big[U\big]$, while the agent maximizes $\mathbb{E}\big[U\big]$. In the beginning, the generator will try to generate simpler environments to be solved by the agent, and as the agent learns to solve more difficult environments, the generator will try to generate more difficult environments. We can then stop the training at the state when the environments will become interesting for the human players.

To obtain the target for the generator, the generated state is evaluated by the critic Q, providing the target utility U. The simulation of the generated state is thus not required, and the whole setup is thus fully differentiable. However, the formulation obliges to restrain from the natural environment rewards, constraining them instead to 1 for the won state, -1 for the lost state, and 0 for the unsolvable environment and unfinished state.

The method requires very little data nor domain knowledge, however, it has a high computational cost, and the dependency between the agent and the generator may lead to mutual exploitation (as shown in the experiments, the actual results actually generated complex, unsolvable environments at the beginning, then simple, and finally very easy environments). We hypothesize that an unluckily initialized critic, that produces wrong utility function estimates will also cause the generator to generate incorrect targets, which, in turn, will produce the incorrect training of an agent.

Another great idea, presented in [20] is based on regret. The setup contains two agents acting in the environment: the protagonist and antagonist, as well as an adversary that controls the environment. The algorithm is as follows: first, the adversary generates an environment, in which several trajectories for both the protagonist and the antagonist are collected. *Regret* is then computed as the difference between the maximum total discounted rewards of the antagonist's and the protagonist's trajectories. The protagonist then receives a reinforcement signal $-Regret$, while the antagonist and the adversary both receive *Regret* reinforcement signal. Thus, the adversary is motivated to generate environments beyond the protagonist's abilities, but the regret incentivizes the tasks, on which the protagonist fails but the antagonist succeeds. As in previous methods, we may stop learning once we're satisfied with the difficulty of the generated environments, and use the adversary as a generator.

2.3 Tangent works

Several papers explore level generation rather indirectly:

- For example, [21] trains fractal neural networks to play SimCity, where Advantage Actor Critic (A2C) [22] an agent learns to design cities. Although this does not directly focus on creating an environment for the player to play in, this might be viewed as a step towards game level generation, where the mechanics of the SimCity game itself are used to evaluate the created environment.
- When generating game levels, the question of playability comes naturally. We generally need to deal with this and other possible constraints. [23] comes up with a Constrained GAN (CAN) approach, that uses a semantic loss (SL) [24], evaluated with using the Knowledge Compilation [25] to penalize the network proportionally to the mass it allocated to invalid structures. CAN succeeds in generating natural, valid (in terms of pipes) and playable levels. We can also utilize SL with the knowledge map to induce validness into our generator.

Chapter 3

Proposed method

3.1 Problem setting: levels, embeddings, environments, transition functions

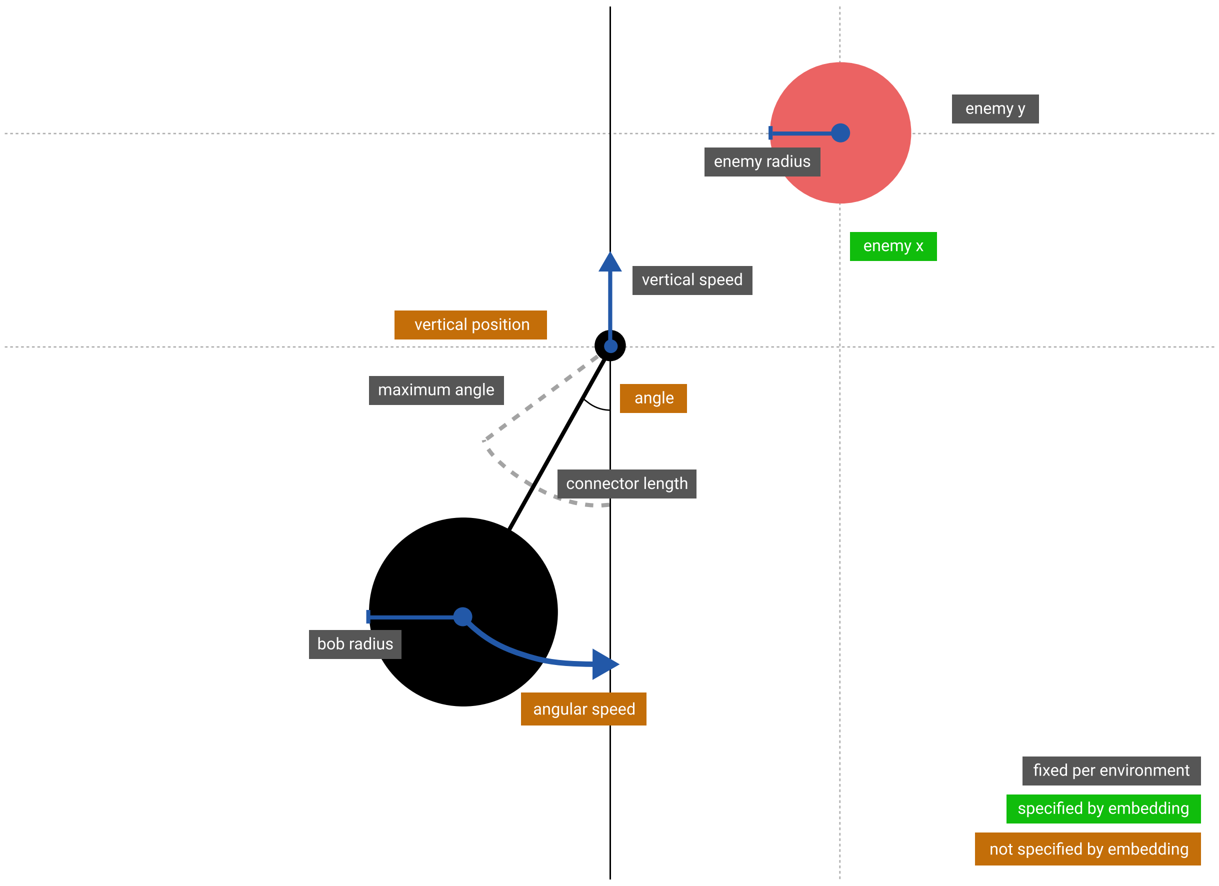


Figure 3.1: Our pendulum environment and its parameters. Colored gray is the parameters that are fixed per environment and essentially "hard-coded" in a transition function. These are not part of a state. Colored green in orange are the components of the state, where green are the components, that are generated (embedding), and orange is the components, that are not a part of embedding.

Environment

Formally, our environment is a tuple (S, A, T, S_0, S_T), where:

- S is a set of states.
- A is a set of actions.
- $T : S \times A \to S \times \mathbb{R}$ is a transition function, which takes a state and an action, and returns the next state and reward.
- $S_0 \in S$ is a set of starting states.
- $S_\mathbf{T} \in S$ is a set of terminal states.

A trajectory τ is a tuple $((s_0, a_0), (s_1, a_1), ..., (s_i, a_i)..., (s_t))$, where:

- $s_i \in S$ are the visited states
- $s_0 \in S_0$ is a starting state
- $s_\mathbf{T} \in S_\mathbf{T}$ is a terminal state
- $a_i \in A$ are the actions taken

At each transition, the next state and reward are calculated using the transition function: $T(s_i, a_i) = (s_{i+1},\ r_i)$

Informally, our environment defines the mechanics of the game (transition function) - defining how the player's actions lead to the next states and whether those states are terminal. Additionally, it specifies the set of starting states.

Game level

We define game level simply as a starting state $s_0 \in S_0$. Although different formulations are possible - for example, one may define a game level as also having a specific transition function - we will use this definition in our work further on. Generating a game level is then interchangeable with generating a starting state.

Level embedding

Level embedding is a vector $\rho \in \mathbb{R}^n$ that specifies some parameters of a starting state, restricting the starting states' space to a subset $S_\rho \subset S_0$. We denote a starting state that has its parameters specified by an embedding ρ as s_ρ.

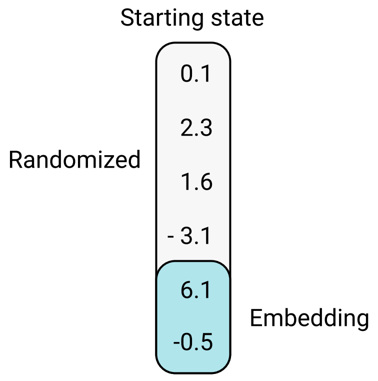


Figure 3.2: An example starting state with the ”blue part” specified by the embedding

For example, in our experiments we use an environment with k different enemies placed in it (Figure 3.1). Thus, our embedding vector is k-dimensional, where each dimension specifies the x coordinate of the k-th enemy. The whole state, however, also includes the enemies' y coordinates, radii, angles, and others. If all of those remaining parameters can be ”hard-coded” into the environment's transition function, then the embedding vector fully specifies a starting state. In other cases, though, those parameters may differ between different starting states $s_\rho \in S_\rho$. This is also the case in our environment, where the starting angle and the angular speed are part of the starting state, but not specified by embedding.

3.2 Goal (broadly): generate game levels automatically

Now we continue with the problem formulation. We want to generate the game levels automatically. As mentioned above, this is interchangeable with generating an embedding vector. Thus, picking any real numbers that form an embedding vector formally completes the task.

However, we may end up with levels, which are either unplayable, present too little or too much challenge to the player, are boring, repetitive, etc. Generally, the embedding vector should conform to a type of constraint. If a game designer could fully specify all of those constraints, then the task becomes trivial. However, it's practically impossible to know how embeddings result in a gameplay experience. In other words, it's almost impossible to specify those constraints perfectly.

When [human] game designers design a level, they cannot know in advance where to place the player, enemies, etc. They cannot tell how difficult or entertaining a level will be, yet they come up with sensible *approximations*, that can later be adjusted during the game testing phase. Formally, game designers have some underlying understanding of those embedding constraints - coming from the "outside" experience (e.g. you may have a good understand of the game even before playing it - perhaps due to its similarity with other games). The constraints may be imperfect, but its a generally a good starting point.

In the conventional PLG, one may specify constraints, for example, via a set of rules, grammar, or something similar. Generated levels thus always conform to the constraints. However, this approach is usually limited in two ways, obvious from the approach itself:

- Levels are usually similar and repetitive - at some time the player usually infers the rules, that were used to define the constraints, and thus the element of novelty disappears as the game becomes predictable.
- This approach is impossible to use for very complex games, where, by definition, there is a large number of complex rules, specifying how does the player interact with the environment, how does it change over time, etc.

Thus, we aim to mimic the process of finding reasonable (closer to the the human-designed) constraints using machine learning methods. We hypothesize that this will keep the levels interesting and creatively designed, while introducing the automation of the process.

The only limitation is that the embedding space should be bounded from both below and above. Thus, it represents an n-dimensional hypercube, in which we are searching for the tighter constraints, which specify valid levels only. In our particular environment, this corresponds to limiting the enemies' x coordinates to be between x_{min} and x_{max}.

3.2.1 Evaluating the quality of a level

How do we know if the generated level was good or bad? In a real-world scenario, we perform game testing, collect feedback, and adjust if needed. Testers (which would ideally resemble the game's target audience) would play the game and describe their playthrough as either challenging or easy, diverse and interesting, or repetitive and boring. We may use factors, such as fantasy, curiosity, player control, and challenge to evaluate how much *fun* do our players experience in our game. Our ultimate goal is to maximize the experienced fun [2].

It is rather challenging to train our machine-learning-based generator solely from the player's feedback. Perhaps it's possible to fine-tune it once it confidently generates playable and engaging levels. But at the beginning, an untrained NN, will likely produce nothing but garbage.

To evaluate the quality of a level, we are doing something similar to playtesting, but with AI agents instead of humans. These AI agents play the generated levels and report their experience. This produces the feedback for the generator, which then adjusts to produce better levels. This is explored in a greater depth in the subsection 3.5.2.

3.3 Goal (more specific): generate game levels of a specific difficulty

There are multiple components to players' game experience, most of which are tricky to quantify accurately. As mentioned before, our idea of a level's quality is based on the level's difficulty. Generally, when the difficulty reasonably matches the player's skill level, it produces the most engagement and joy for the player. Overshoot and players feel frustrated, have too low of difficulty, and they feel bored. If we know what difficulty is required, as well as being able to accurately produce the level of that difficulty, then, theoretically, we can create the best experience for the player. We focus on the latter part, which leads us to the formulation of the problem as a search of the level of a given difficulty.

While focusing on the difficulty, it is also important to generate a diversity of different levels. If our generator is only able to produce a single level of a given difficulty, we can generate only a single level progression. This would feel repetitive. That is why in the section 3.6 we extend the formulation of the problem to account for level diversity

In this work, we have limited ourselves to a set of arcade games that all usually feature relatively simple gameplay mechanics and have a similar goal of scoring as high as you can. More modern and complex games can focus on different things: experience the game's world, story, characters, etc. These games are effectively beyond the scope of this work. However, they may also benefit from the proposed method of game level generation in some way (e.g. improved enemy AI programming).

However, what is difficulty to begin with? How can we formalize its definition?

3.3.1 Difficulty

We define the difficulty as a real number d, that lies in a closed interval $[0, 1]$. It is a highly arbitrary measure, we only require it to have the following property: the higher the number, the more challenging it is to perform. Zero difficulties would mean the easiest possible environment, while 1 would mean the most difficult one.

3.4 Solution: framework: generator, evaluators, oracles, and differentiability

Generator

Harking back to the search of constraints in the embedding space, we want to find a set of constraints that would limit the space to valid "high-quality" levels. Formally, we can reformulate this as a search of constraints that would limit the space to the embeddings, which result only in the levels of a given difficulty.

Practically, we express this as a function that maps a difficulty to a particular embedding (or *embeddings*, section 3.6). Thus, our generator $\mathcal{G} : [0,1] \to \mathbb{R}^n$ maps a difficulty d_{in} to an n-dimensional embedding vector ρ.

$$\mathcal{G}(d_{in}) = \rho$$

Our generator is parametrized by parameter vector θ and is a simple feed-forward NN, that we have chosen to train via stochastic gradient descent (SGD). Although other optimization methods, such as evolutionary algorithms can be used, back-propagation has become predominant in the last years, and thus it has become our method of choice.

Evaluator

$$\mathcal{D}(\rho) = d_{out}$$

The target for the generator is provided by an evaluator. Evaluator $\mathcal{D} : \mathbb{R}^n \to [0,1]$ is a function, that takes an embedding vector ρ as its input and returns an evaluated difficulty d_{out}. We then compare the input difficulty d_{in} that was used as an input to the generator to generate a level with the evaluated difficulty d_{out} and minimize their difference $|d_{in} - d_{out}|$.

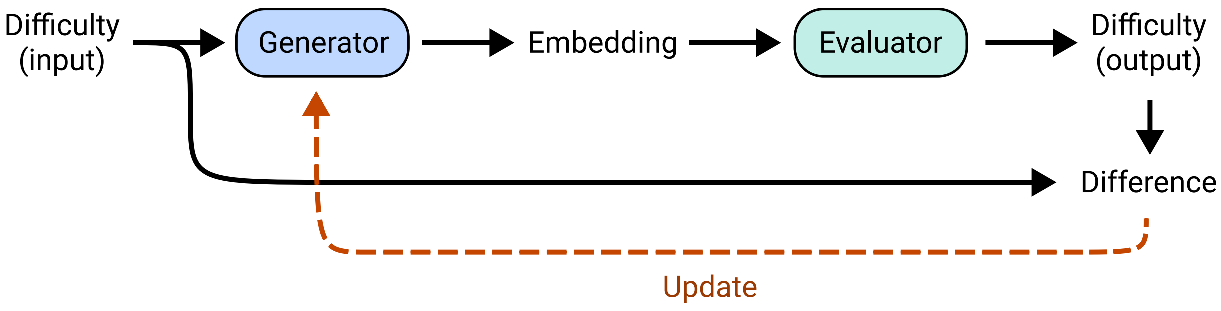


Figure 3.3: Generator/evaluator setup

Note that, at this point, the evaluator is just a function that maps an embedding to a difficulty - not necessarily a NN. For the setup to work, however, the gradients should flow through the evaluator. Depending on the particular choice of one, this may or may not be the case.

3.4.1 Attempt 1: training directly from evaluator

Suppose we have an evaluator $\mathcal{D}$, that can evaluate the difficulty of a state s, and its evaluation is differentiable w.r.t. to the state. Then, the setup looks like the following:

1. The generator generates an embedding ρ_θ, that specifies a subset of starting states $S_\rho \subset S_0$.
2. Starting state $s_{0|\rho_\theta} \in S_{\rho_\theta}$ is sampled.
3. The trajectory is sampled starting from $s_{0|\rho_\theta}$ with transition function, using actions sampled from policy $a_i \sim \pi$. From the trajectory we select states $s_{i|\rho_\theta}$
4. Each state's difficulty $\mathcal{D}_{state}(s_{i|\rho_\theta}) = d_{s_{i|\rho_\theta}}$ is evaluated using a state difficulty evaluator.
5. Individual states' difficulties $d_{s_{i|\rho_\theta}}$ are aggregated (e.g. averaged) into a single measure for the trajectory difficulty, which is used as an approximation for the difficulty of a starting state $d_{s_{0|\rho_\theta}}$, and thus, the difficulty of the embedding d_{ρ_θ}.
6. By sampling many trajectories ($t \to \infty$), we approach a better estimate or a difficulty of the embedding $d_{\rho_\theta} = \mathbb{E}_t\left[\ d^t_{\rho_\theta}\ \right]$.

Differentiability problem

For this to work, however, the gradients $\frac{\partial s_{i+1}}{\partial s_i} = \frac{\partial T(s_i, a_i)}{\partial s_i}$, and $\frac{\partial \mathcal{D}(s))}{\partial s}$ should exist. The latter usually isn't a problem, but the former is. Usually, the transition function contains 'if' statements, that produce non-differentiability. This is also the case with our environment, where the player can either perform an action that would change the direction of oscillation, or do nothing.

It is also worth noting that this back-propagation through the entire trajectory is extremely slow, is likely to cause exploding/vanishing gradients, and introduce instability in training - in a similar fashion as in the recurrent neural networks (RNNs).

3.4.2 Attempt 2: REINFORCE [26]

The idea is to change our generator from outputting an embedding ρ_θ to outputting a multivariate probability distribution P_θ, from which we then sample several $\rho_\theta \sim \mathrm{P}_\theta$. The difficulty evaluation of an embedding can then occur in a non-differentiable manner (e.g., as described in subsection 3.4.1). This also saves plenty of computational resources, as we sidestep the computing of the operations' tree and storing intermediate results.

Each sampled embedding ρ_θ is assigned a score w (characteristic eligibility), that is a difference between the input difficulty d_{in}, and the evaluated difficulty of that embedding d_{ρ_θ}, i.e. $w = |d_{in} - d_{\rho_\theta}|$. And then we maximize the log probability of the embeddings that resulted in a higher w.

$$\theta_{t+1} \leftarrow \theta_t + \alpha \ \mathbb{E}\Big[w \nabla_\theta \log p(\rho_\theta)\Big]$$

This works. However, it comes with a drawback of evaluating many ρ_θ to achieve a reasonable difficulty estimate. This brings the computation cost problem right back, as sampling trajectories also usually cannot be GPU-parallelized.

There is also a flaw in the *meaning* behind the method itself. When our generator assigns a particular probability to an embedding ρ_θ means that the embedding ρ_θ will have a difficulty d_{in} with a probability p_{ρ_θ}. However, this is not what we want our generator to output [1]. We want the generator to output a single embedding, which would always result in the difficulty d_{in}. Thus, our perfect generator would output a Dirac delta function probability distribution, that would have an infinite probability for a single valid embedding, and zero probability everywhere else.

Practically, this causes the training to break the generator at some point, as the log probabilities will approach infinity, and the gradients will either explode or become NaN.

3.4.3 Attempt 3 (and the final solution): NN Evaluator and oracles

These problems led us to search for an alternative solution. The core idea is to use another feed-forward NN to evaluate the difficulty of an embedding ρ_θ. Thus, the gradients $\frac{\partial d_{\rho_\theta}}{\partial \rho_\theta}$ exist, and thus, also the gradients $\frac{\partial d_\rho}{\partial \theta}$.

Oracle

NN evaluator needs to be trained to evaluate the embedding's difficulty $d_{out} = d_\rho$ accurately. For this we will use another difficulty evaluator that does not have to be differentiable. To avoid ambiguity, we will refer to it as an oracle. An oracle takes an embedding ρ, and outputs a difficulty of that embedding d_ρ. We trust it to produce an accurate difficulty estimation, making it perhaps the most important part of the framework.

Evaluating the actual difficulty of the produced level now does not pose a problem. Additionally, the generator now produces an embedding of the required difficulty, that

[1] the agent/environment stochasticity is non-essential here

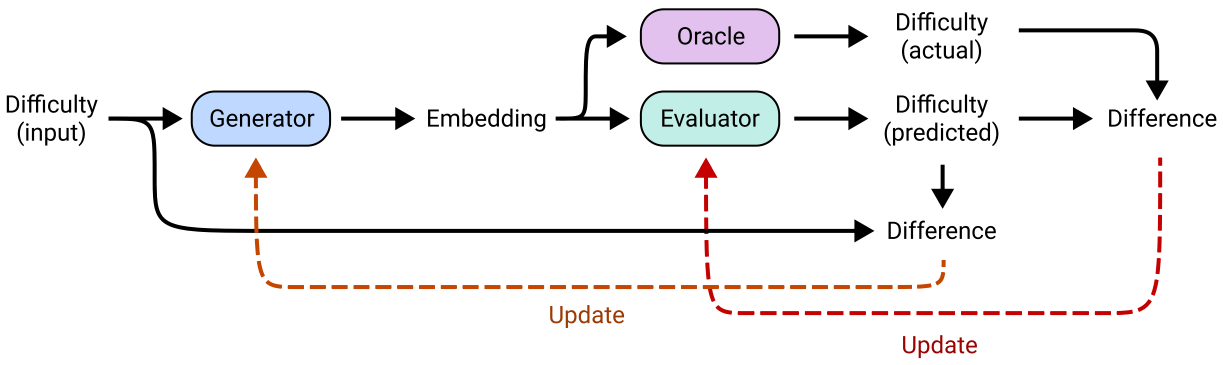


Figure 3.4: Generator/evaluator/oracle setup, that extends the previous setup

makes sense definition-wise.

The structure may resemble that of the GAN [19]. However, here the generator and the evaluator do not compete in a minimax game. Rather, they cooperate. Evaluator's role is to merely map an embedding to its difficulty in a differentiable way.

With this setup, it is simple to use different oracles - there is essentially no constraint on how it should work. The next part will explore different oracles and compare their evaluations. Its worth noting that this setup can be easily extended to examine the multifactorial quality of the levels, however, for simplicity, we will stick to examining the difficulty only.

3.5 Different oracles

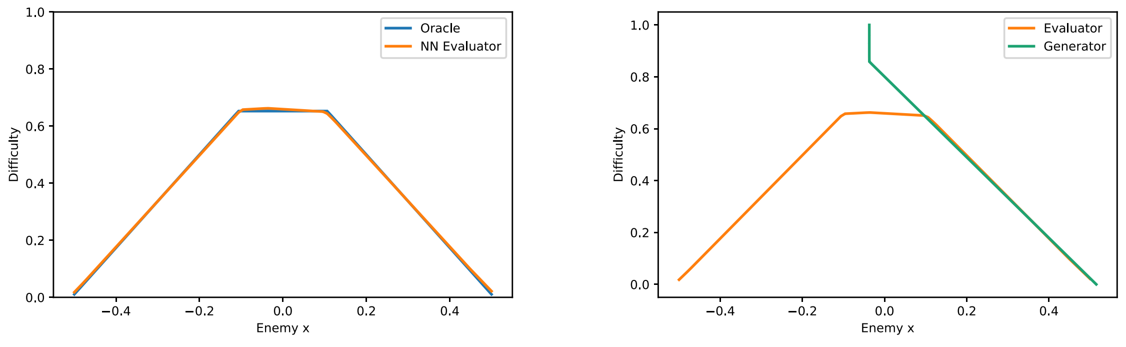
This section compares different oracles. We start with a simple heuristic oracle that uses a hand-crafted function to estimate the difficulty. Then we continue with the more advanced oracles, which use agents. We will start by calculating the difficulty from the trajectory rewards - a purely extrinsic measurement. Then we'll move on to the Q-values and action probabilities, which represent intrinsic measurements. Lastly, we'll briefly discuss the problem of subjectivity/objectivity, and the challenge of training the agents without a reliable level generator.

3.5.1 Warming-up example: heuristic

Firstly, to test the proposed framework, we have created a simple oracle that calculates the difficulty directly from the embedding ρ.

There are several possible ways to train the generator and evaluator. The first one is to update the generator and evaluator both at the same time. Then we can also do one generator/evaluator update per several updates of the other one. One other possibility is to first train the evaluator to a sufficient accuracy, and then omit the actual level evaluation with the oracle completely. Finally, one might use a technique somewhat in between, e.g. pre-train the evaluator and then invoke the oracle once per several training iterations. This, again, can resemble the generator-discriminator training of the GANs.

Here we have chosen to train the NN evaluator first and then use it to train the generator. In our pendulum environment with a single enemy, where embedding ρ is a one-dimensional vector that contains the enemy's x coordinate, the heuristic oracle produces the following difficulty evaluation:



(a) Difficulty evaluation of the single-enemy pendulum environment by a heuristic oracle and the learned approximation by the NN

(b) Generator learning from the evaluator

The x axis represents the enemy x position. Logically, when the enemy is very far off to the side, the difficulty is close to zero. As the enemy gets closer to the center, the amount of free space decreases, and thus it becomes more difficult to perform a successful evasive maneuver. We can see that our heuristic estimation does not cover the full $[0, 1]$ range. This does not contradict the definition of difficulty that we have introduced, as not every environment will have the full range of difficulties.

The yellow curve shows the approximation of the NN evaluator. For such a simple function, training a NN is trivial.

Then the next step is to use the trained evaluator to train the generator. Essentially, we are searching for an inverse of the evaluator function. In a sense, when we have trained an evaluator, we already have a way to generate a required level: just pick whichever embedding corresponds to the desired difficulty by looking at the graph. Practically speaking, we would have a data table, that for each point in an embedding space would store its difficulty. Then we would sort the table by the difficulties and could pick a satisfying embedding. There are two problems, though

1. Embedding space is usually continuous, and we would have to discretize and interpolate between the points. This introduces imprecision, as well as a high spatial complexity - a need to store each embedding and its corresponding difficulty.

2. With a growing number of dimensions, the spatial complexity grows exponentially (curse of dimensionality). Our data table thus grows exponentially large.

That is why we argue that it's reasonable to have another NN approximator (generator), that would perform the inverse mapping. However, from the Figure 3.5b we can see the obvious limitation: the generator cannot find multiple embeddings of the same difficulty. This makes sense, as a function cannot produce two different outputs for the same input. However, we would like the generator to be capable of generating multiple levels within the same difficulty. This would correspond with the generator finding both the x and $-x$, on the graph. We will tackle this problem in the section 3.6. Also, for the part where the difficulty is too high, the generator can essentially produce anything, as the environment (judging from the oracle's evaluation) does not grant the possibility for difficult levels to exist. Depending on what we want, we may simply discard these levels completely, or use them, acknowledging their real difficulty.

Nevertheless, we see that the setup works at least with a simple evaluator and a simple 1-dimensional embedding space environment. Before moving to more sophisticated oracles, we should note the issue of subjectivity.

3.5.2 Oracles that use agents. Objectivity/Subjectivity

Next, we will review more advanced methods of difficulty estimation. All of them resemble the [human] playtesting by having a group of agents which sample trajectories in an environment starting from the embedding-parametrized state.

Subjectivity

Such evaluation would obviously be conditioned on the individual players. Each player is always a little different, has a different skill level, pays attention to different things, and then is also subjected to their own playing experience. The same level may be classified as difficult by an unskilled player and trivial by a skilled one. Thus, the measurement is highly subjective.

One good approach to account for everyone is to adapt the game to each player. This may happen, for example, by tracking the player's performance throughout the game, and then tweaking game parameters to make the game more or less difficult.

Sometimes, however, this is either infeasible or not possible at all (perhaps by the game design, e.g. Dark Souls [2]). Then, the game provides either several difficulty levels to choose from, or a single fixed difficulty. The former relies on the player's ability to report their own skill level accurately, which may often not be the case.

[2]https://en.wikipedia.org/wiki/Dark_Souls

If we reformulate the problem to generating the levels *for a given player*, then we'd have to fine-tune our generator somehow to that specific player (either by some evaluation of the player's performance in the actual game or by conducting some skill tests before the actual game). In both cases we normally want the generator to generate at the very least playable and somewhat challenging levels initially. In other words, we want the generator to be useful for each different player, despite their individual differences. We want our generator to be objective in a sense that produced levels are what the *whole player's population on average* would expect them to be.

Thus, we use multiple agents, where each one returns their own evaluation, and then we average them. Doing so, we obtain a single measurement that we consider objective.

Skill Distribution

However, usually, it makes more sense to assign nonuniform weights to the players' measurements. In our setup, we attribute the weight based on a player's skill level. Extensions to this method may consider, for example, the time spent playing the game (in our experiments all AI agents spend equal time playing the game). Depending on the game we aim to create (casual, challenging, hardcore), we may assign different weights to the measurements obtained by a different skill. For example, we may not care as much about what do low-skilled players think, as what do high-skilled ones.

In our work, to determine the weight, attributed to the measurement of an agent, we use the skill distribution. This distribution should specify how important each skill level is, and should, ideally, reflect the game's target audience's players' skill distribution. We hypothesize that truncated skew-normal distribution [27] is well-suited for this task. The distribution is truncated at $[0, 1]$ interval and is controlled by three parameters: mean μ, standard deviation σ, and skewness α. For example, if we want to design a difficult game, we could specify the mean μ closer to 1, and skewness $\alpha < 0$.

This distribution is arbitrary - proposed by the game designer and should theoretically come from knowing your target audience (e.g., by conducting market research and/or collecting player's data from the other games).

Using trajectory rewards

Our first sophisticated oracle is based on the simple comparison of the trajectory rewards. Here, each agent computes their difficulty based on the rewards that the player/agent is given during their playthrough. For simplicity, although, let us start with the binary formulation, where only two scenarios are possible (similarly to [28]). Let $\mathcal{T}$ denote the space of all the trajectories, which we can split into two disjunctive subsets:

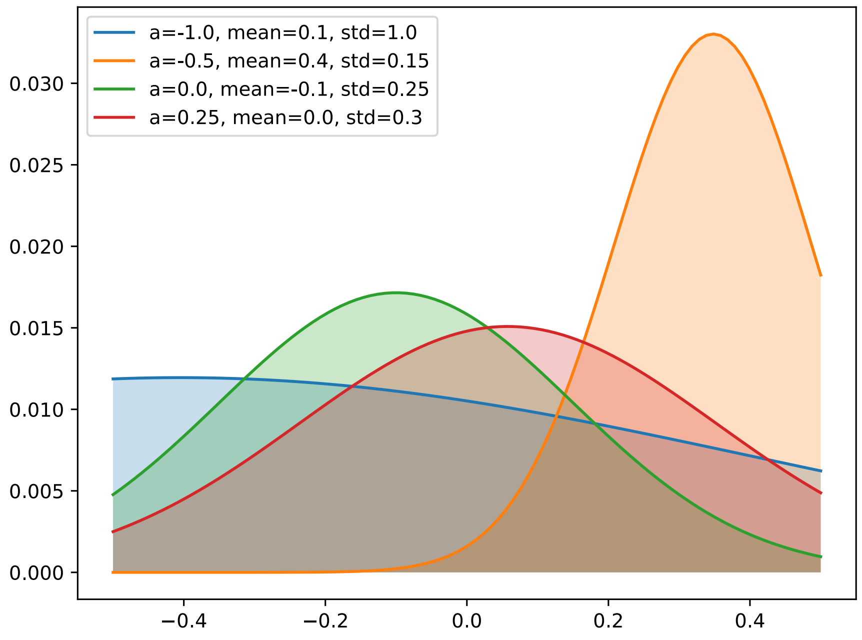


Figure 3.6: Examples of truncated skew-normal distributions

1. Trajectories, which end with a win $\mathcal{T}^W$
2. Trajectories, which end with a loss: $\mathcal{T}^L$

We will examine trajectories, parametrized by some embedding ρ. This determines a subset of starting states S_ρ. We'll denote the set of trajectories, which start from the states $s_\rho \in S_\rho$ as $\mathcal{T}_\rho$. Then if we knew the full space of all possible trajectories $\mathcal{T}_\rho = \mathcal{T}_\rho^W \cup \mathcal{T}_\rho^L$, we could define the difficulty of the embedding ρ as a ratio of the number of the lost trajectories $|\mathcal{T}_\rho^L|$, to the number of all trajectories $|\mathcal{T}_\rho|$:

$$d_\rho = \frac{|\mathcal{T}_\rho^L|}{|\mathcal{T}_\rho|}$$

The problem with this approach is that the number of all possible trajectories in the environment is generally infinitely high. However, we can approximate by sampling the S_ρ and counting the wins and losses of that sample. As our sample size grows, the better will our approximation become. Another obvious limitation to this win/loss approach is a requirement for an environment with a clear definition of win and loss. When there is no such binary definition, but instead a scalar reward is given, we may place an arbitrary threshold and classify the trajectories with the rewards higher than that threshold as winning trajectories, and others as losing.

Generally, though, when step rewards are provided and there is an opportunity to compute a trajectory reward, there is a better way of computing the difficulty. Let's note the sum of step rewards R_t, obtained at each step of the trajectory τ as $R_\tau = \sum\limits_t R_t$. Assuming we know the highest and lowest possible rewards for our environment: $R_{best} =$

$\min_{\tau \in \mathcal{T}} R_\tau$ and $R_{worst} = \max_{\tau \in \mathcal{T}} R_\tau$, respectively, we can define difficulty of a trajectory d_τ as a measure of how close its reward is to the worst reward:

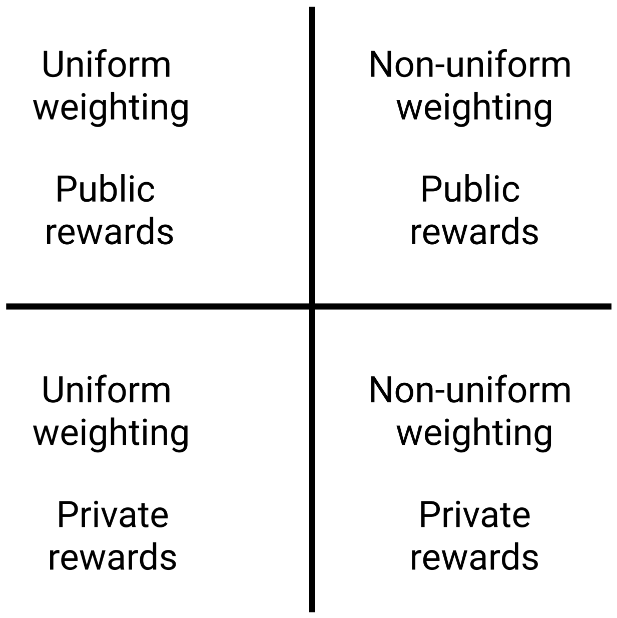
$$d_\tau = \frac{R_{best} - R_\tau}{R_{best} - R_{worst}}$$

Note that when $R_{worst} = R_{best}$, then for any trajectory $\forall \tau$, $R_\tau = R_{worst} = R_{best}$, and the difficulty is undefined $d_\tau = \frac{0}{0}$. This intuitively makes sense, as the difficulty is relative, we can only compare how good or bad are we doing with respect to some other result.

Again, in practice, we do not generally know the maximal and the minimal trajectory rewards, but we use the sampling approximations to compute $\hat{R}_{best}$ and $\hat{R}_{worst}$, which are used to approximate $\hat{d}_\rho$. From now on we will refer to the approximations without the "hat", namely R_{best}, R_{worst}, and d_ρ

Intrinsic/Extrinsic measurement

The question arises that concerns the "visibility" of the R_{best} and R_{worst}. Should they be "private" to each agent, or should they be "publicly" visible to all the agents? The first option makes the measurement more intrinsic and self-reported in its nature, as the agent only observers their own performance and does not know what outcomes may be possible (achieved by other agents). The second option is more objective, as the scores are shared. We may mix this with the agents weighting by the skill distribution, thus giving four possibilities:



Uniform weighting Public rewards	Non-uniform weighting Public rewards
Uniform weighting Private rewards	Non-uniform weighting Private rewards

Figure 3.7: Matrix of possible weighting and normalization of the agent's measurements

Which one should be used? This question goes beyond the scope of the current work, however, we propose some general thoughts about this problem and the problem of selecting the correct oracle in the subsection 3.5.6.

3.5.3 Using Q-values

Using trajectory rewards for estimating difficulties is great in its simplicity. However, in its essence, it's oblivious to the underlying game mechanics. It is somewhat similar to the REINFORCE in a way, that it merely observes the correlation of the rewards with the generated levels. This is analogous to only collecting players' scores during the game testing, and discarding everything else - the self-reported feeling of the game, finer insight into which parts of the level resulted in the overall difficulty, etc.

In Q-learning, the task is to learn a Q-value function, that maps state-action pairs to the expected return of the trajectory, taking the action in a state, and acting optimally further on [29]. The Q-value function gives insight into the differences between the individual actions. This is how we'll determine the difficulty of a state, and then the difficulty of an embedding.

The intuition behind the method is as follows: if all actions have similar Q-values, then we can essentially pick a random action and achieve a similar outcome. However, if there are only a few actions that lead to a good outcome, and the majority of actions lead to a bad outcome, then the difficulty is high. In other words, we have to be accurate and perform the correct action. Mathematically this can be expressed as the difference between the maximum and the average Q-values in a state over the available actions:

$$d'_s = \max_{a_j \sim s} Q(s, a_j) \; - \mathop{\mathbb{E}}_{a_j \sim s} Q(s, a_j)$$

However, d'_s is not normalized to the range $[0, 1]$. To obtain the normalized values, we do something similar to the trajectory rewards method:

$$d_s = \max_{a_j \sim s} \left[\frac{Q(s, a_j) - Q_{min}}{Q_{max} - Q_{min}}\right] - \mathop{\mathbb{E}}_{a_j \sim s} \left[\frac{Q(s, a_j) - Q_{min}}{Q_{max} - Q_{min}}\right] = \frac{d'_s}{Q_{max} - Q_{min}} \tag{3.1}$$

The normalization can occur either before or after calculating the difficulty as shown in Appendix A. We use Q_{min} and Q_{max} publicly to the whole population of agents. Thus our oracle knows the possible outcomes across all agents.

To obtain the difficulty of a trajectory, we aggregate the individual difficulties of the states. There are several possibilities here. The most straightforward one is to compute the average difficulty over the states. Another option might be to use the maximum value - it could make more sense in estimating the difficulty of a trajectory, for example, with an unexpected difficulty spike in the end. Finally, more sophisticated methods can be used, that analyze the difficulty progression over the trajectory more closely.

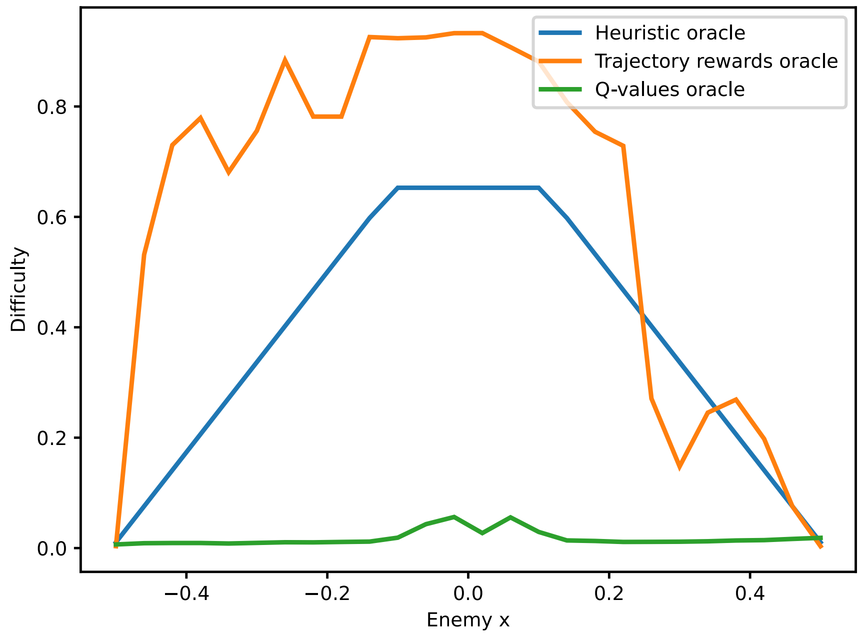


Figure 3.8: Difficulty evaluations of the pendulum environment by different oracles

We can see from the Figure 3.8 that the Q-values oracle produces a very different difficulty estimate from both heuristic and trajectory rewards oracles. We hypothesize that some extension to the proposed algorithm is required to make this oracle useful.

3.5.4 Using action probabilities

Sometimes our AI agent selects actions by a probabilistic policy, i.e., that outputs the probability distribution over the possible actions in a given state $p(a|s)$. This is usually the case in the policy gradient methods, for example, REINFORCE or A2C/A3C [22]. This allows for another way of measuring the difficulty.

The first idea is to treat the action probabilities the same way as the normalized Q-values and compute the state difficulty as a difference between the maximum and the average probability over the actions. Another possibility is to compute the normalized entropy of that distribution, that is:

$$d_s = -\sum_{a \sim s} p(a|s) \log_{N_{actions}} p(a|s)$$

Then the individual difficulties of the states may be aggregated in a similar fashion as the normalized Q-values. We should note that this measurement is purely intrinsic and self-reported in nature, i.e. the action probabilities are not getting normalized w.r.t. to the $\max_s \max_{a \sim s} p(a|s)$ and $\min_s \min_{a \sim s} p(a|s)$.

AI Agents

All oracles that run environment simulations with AI agents obviously need those agents to come from somewhere. In our experiments, we used a simple heuristic agent, that uses a hard-coded policy for trajectory rewards oracle and a DQN agent for Q-values oracle. Trajectory rewards oracle essentially imposes no constraints on the agents whatsoever, so any agent may be used. Q-values oracle requires a way to estimate Q-values. However, the agents may use a different policy, then the greedy (w.r.t. Q-values from the network), or ϵ-greedy policy. One may use, for example, an A2C agent, with a slight modification to output Q-values instead of V-values.

The quality of the oracle depends on the quality of the agents, which are trained in tandem with the generator. This produces a potential problem: poorly trained agents may give incorrect evaluations to the generator, which in turn, makes the generator produce garbage levels, which, in turn, makes the agents converge to even worse policies. A possible solution is to use more simple hard-coded agents from the beginning, let the generator learn to generate decent levels, and then switch to more complex AI agents further on after the process has reached some degree of stability.

3.5.5 The difficulty as a measure of time required to reach a level of performance

A very different approach to measure difficulty is to track the time, that was required to reach a good level of performance in the environment. The most straightforward way to measure performance is by using trajectory rewards. Another one might be to use the intrinsic measurements of the agent's difficulty - as we introduced above - e.g., by Q-values, actions probabilities.

The drawback of this method is its extremely low speed. After each iteration, once the level is generated, we would have to train new agents and observe how much time they needed to master the level. The method is also very demanding to the agents themselves. In simple games, it's rather simple to reach the super-human level of performance. However, in complex games with sparse rewards and unclear reward functions, this may present considerable difficulties.

We hypothesize that this method may be used at the latest stages of the generator training to fine-tune it - perhaps in tandem with another difficulty estimation method.

3.5.6 Choosing the right oracle

Which oracle should we choose and use? If we aim to automate a real game development and maximize players' fun, then we argue that the most important aspect is how close the oracle's evaluation is to the real human players' evaluation. Thus, almost always we'd choose an oracle that uses AI agents to sample trajectories and observe both intrinsic and extrinsic data of their playthrough. In a real-world scenario, many aspects are weighted together in producing a single difficulty measurement. An improvement over this work might also introduce multi-dimensional measurement as a target to the generator, instead of a single difficulty measure. While using the oracles with AI agents, the agents must resemble the human players as closely as possible. [30] focuses on a player's style in addition to the skill to produce more human-like agents. Moreover, creating more human-like agents is explored in [31] where both synthetic and human-like agents were introduced to find defects, examine unintended game transitions, etc. Perhaps, mixing reinforcement learning agents with agents trained in a supervised way from the player's data is required in the later stages of development to fine-tune the generator. Lastly, from the computation cost standpoint, it is likely efficient to begin with the simpler oracles (e.g. hard-coded heuristics), and slowly transition to more sophisticated ones.

3.6 Extension to the task: generate diverse levels

As we have seen from the Figure 3.5b, currently our generator is incapable of generating several different levels with the same difficulty. Instead, we want to be able to generate different, distinct levels, while having them all evaluate to the same difficulty.

Diversity (in terms of embeddings)

A rather straightforward way to compute diversity is to compare embeddings. If our generator produces N different embeddings $\rho_1, \rho_2, ..., \rho_N$, then the diversity $\mathbf{D}$ may be computed as the average norm of all the difference of the embeddings:

$$\mathbf{D} = \frac{1}{N^2} \sum_{i=1}^{N} \sum_{j=1}^{N} ||\rho_i - \rho_j||_p$$

Diversity in terms of trajectories

However, a more sophisticated way to compute diversity is to compare the trajectories that stem from different embeddings [15]. To do this, we would have to compute the differences between the states in the same step in all trajectories. This, however, is extremely costly, as well as conditioned on the agents. Thus, this method may probably be used in the latter stages of the game to refine already decent results.

3.7 Solution: seeds

In our work, we are focusing on the difficulties in terms of embeddings. Thus, we want to generate multiple different embeddings for a single difficulty input, so that their difference is maximized while ensuring that they all evaluate to the same difficulty. With the current setup, there is no possibility to generate multiple embeddings, simply because it is not possible to have a function (generator) that produces two different outputs for a single input. However, it is possible to slightly change the setup and be able to achieve our goal.

Attempt 1: seed as a degree of freedom

The idea is as follows:

1. Extend our generator to accept one more input. We will call that second input the seed. The seed ranges from 0 to 1 and essentially acts as an added degree of freedom: while fixing the difficulty, we can change the seed, and the generator should select one of the embeddings from the possible set of ones, that conform to the input difficulty.

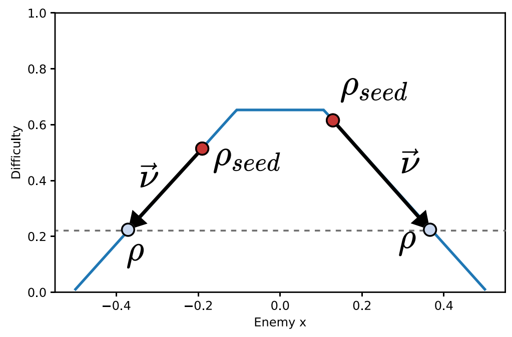
2. We then have to somehow teach the generator to associate different levels with different seeds. To do that we add the negative diversity $\mathbf{D}$ term to the generator's difficulty loss ($|d_{in} - d_{out}|$) so that our generator maximizes the diversity during the and minimization the difficulty loss.

In practice, however, this does not seem to work. Diversity and difficulty losses act in opposite directions: minimizing the difficulty loss makes the generator generate more similar embeddings, while diversity "pushes them away" from each other.

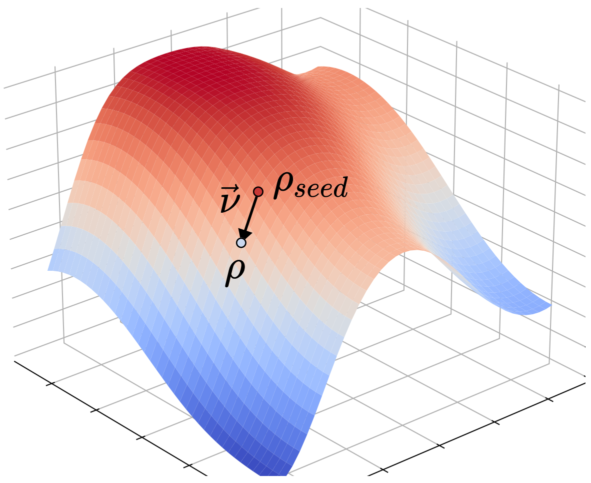
Attempt 2: generating offsets to an N-dimensional seed point

With this approach, we change our seed to be N-dimensional, where the dimensionality is the same as the dimensionality of the embedding space. In other words, our generator will now receive a point in embedding space ρ_{seed} additionally to the difficulty input. Then, the NN will output the offset vector $\vec{\nu}$, which is added to the ρ_{seed} to produce the final output $\rho = \rho_{seed} + \vec{\nu}$. The Figure 3.9a demonstrates how that looks in our environment with a 1-dimensional embedding space. Depending on the chosen ρ_{seed} we thus may end up with different ρ.

The Figure 3.9b demonstrates how that may look in a higher-dimensional space. The vertical axis of the figure is the difficulty of a particular embedding. Other axes specify each dimension of the embedding space. The generated offset $\vec{\nu}$ from the seed point ρ_{seed} specifies the closest point ρ that evaluates to the required difficulty.



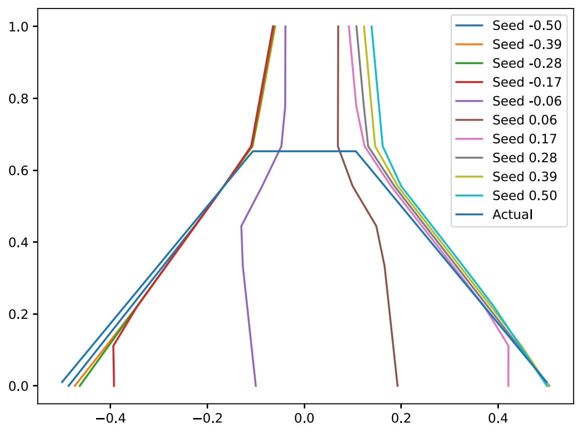
(a) 1-dimensional embedding space



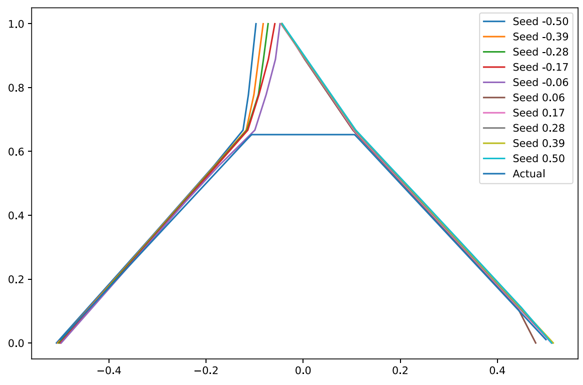
(b) 2-dimensional embedding space

Figure 3.9: Generator generating different offsets depending on the ρ_{seed}

To emphasize the "closest" part we add the $||\vec{\nu}||_2$ regularization term to the generator loss, requiring the generator to produce smaller offsets. In our experiments, we found it crucially important. Otherwise, the generator might as well generate big offsets and converge to output the same ρ for different ρ_{seed}.



(a) Different curves correspond to different seeds, each curve learns a difficulty mapping from it's own seed point ρ_{seed}

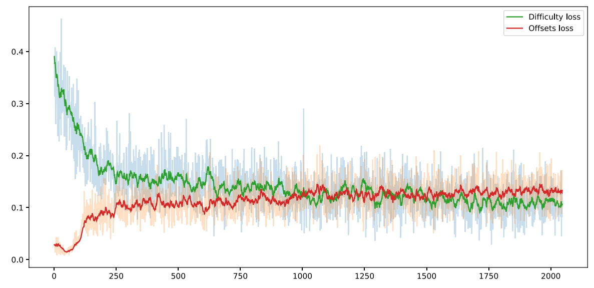


(b) Decreasing the weight of the offsets loss later during the training allows the generator to converge to better results

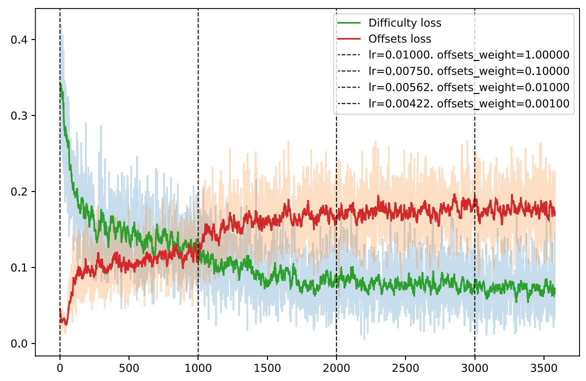
Figure 3.10: Embeddings, generated by the seeded generator

It is a bit tricky to clearly visualize the outputs of the seeded generator. The Figure 3.10a shows the seeded generator in the process of converging. The heuristic oracle is used to produce a target for the generator. Different curves show different seeds. We can see that for some seeds the generator generates the embeddings on one side, and for some - on the other. This essentially completes our task. First, we pick the difficulty, and then by selecting a different seed point we will pick one of the available options.

We have observed that to reach better results, it is important to decrease the weight



(a) Difficulty loss and the offset loss reaching an equilibrium



(b) Decreasing the weight of the offsets loss later during the training allows the difficulty loss to decrease.

Figure 3.11: Difficulty loss and the weight loss progression during the training

of the offsets loss further on during the process of training, as well as the learning rate of the generator. Figure 3.10b shows how different seeds converge to either one or the other side. Figure 3.11b shows how different losses change. We have decreased the learning rate and the weight of the offsets loss by the factors 0.75 and 0.1 respectively on every 1000 steps.

One of the advantages of this method is the potential ability to find all different embeddings of the same difficulty. However, this method is somewhat difficult to use practically. It would be nice to somehow pick only the relevant distinct embeddings, ideally compressing the N-dimensional seed to a single $[0, 1]$ interval.

Alternative approaches

We briefly note other possible solutions to the diversity problem:

1. The generator may output a fixed amount of embeddings ρ_j, and maximize their difference. However, this approach is not as flexible as we would like. Each new output produces a bigger NN. Then there may be a potentially infinite number of different levels of the same difficulties, however, we'd constrain the generator to only generate a fixed number of them.

2. With a probabilistic generator, we might output a multi-modal probability distribution, similarly to that of GAN. In that case, there is a danger of a mode collapse, but we hypothesize that simply rewarding embedding diversity may be enough to combat the problem. However, we have discarded the probabilistic approach as being too costly.

Chapter 4

Practical results

4.1 Pendulum environment

To test our framework, we have implemented a pendulum environment that consists of a swinging pendulum, that constantly moves upwards. At each point in time, there are two possible actions - either do nothing or change the direction of the swing. The goal is to move as far as possible without the bob (grey circle) colliding with the enemy (red circle).

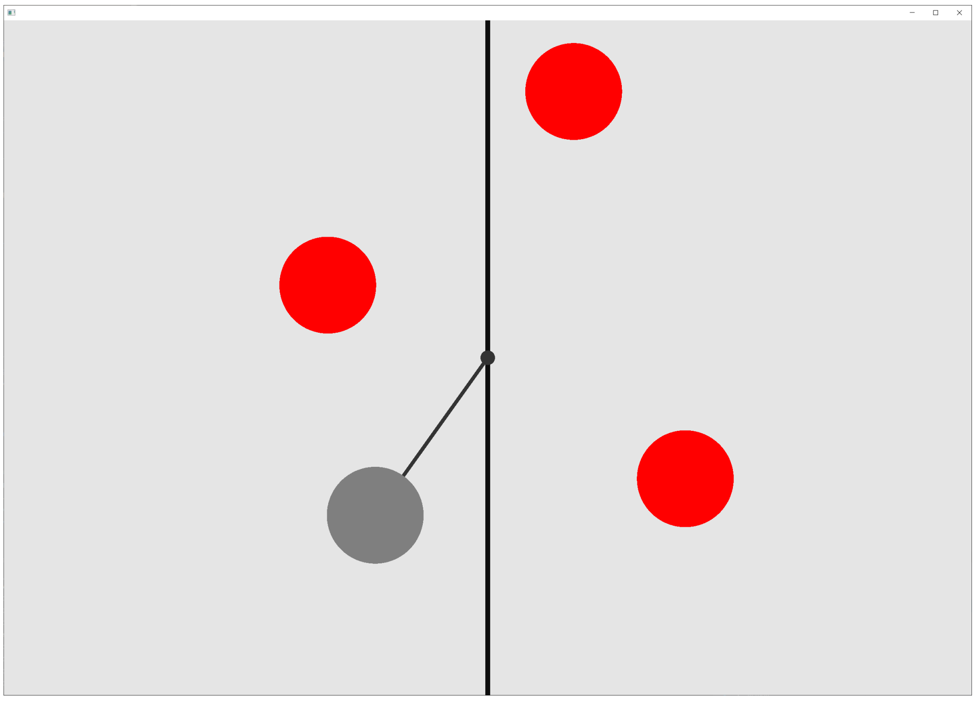


Figure 4.1: Pendulum environment, implemented in python using OpenGL

The number of enemies can be different. In our experiments, we have mostly used a single enemy and then extended the setup to two and three enemies later on. The state is described by $3 + N_{enemies}$ numbers:

1. Vertical position (y_p coordinate of the black circle on the line)
2. Angle α, that the pendulum currently makes with the y axis
3. Angular speed v_α - determines how much the angle α changes over time. Can be either $v_\alpha = \pm\, v_\alpha^{max}$
4. Enemies' x_{e_i} coordinates. It stays constant within the trajectory, however, we include it in the state to be consistent with our problem formulation, but it is of course possible to parametrize the environment and "fix it" onto the transition function

Then, there are environment's parameters that stay constant in all trajectories:

1. Length of the pendulum's rod l
2. The vertical speed with which the vertical position changes v_{y_p}
3. The maximum angular speed of the swing v_α^{max}, that determines the possible values of angular speed
4. Enemies' y_{e_i} coordinates
5. Pendulum's bob radius r_p
6. Enemies' radii r_{e_i}

At each given step, if the collision did not occur, the agent received a small positive reward 1. If the collision did occur, then a large negative reward -100 was given. Also whenever the agent decided to change the swing direction, a tiny negative reward -0.1 was added to whichever reward the agent received. Thus the goal was to evade a collision, as well as to minimize the number of actions taken. This rewards weighting produced agents that behaved as we would expect. At first, however, we did not give a large negative reward for the collision, but instead simply ended the episode. In the limit, this does not make a huge difference, but in practice, this seemed to be one of the main reasons that our DQN algorithm had troubles converging.

4.2 Framework implementation

4.2.1 Generator, Evaluator

In our experiments, for both generator and evaluator, we have used simple feed-forward NN with two hidden layers, 8 neurons. We have used Leaky ReLU (LReLU) [32] to combat the dead neurons problem.

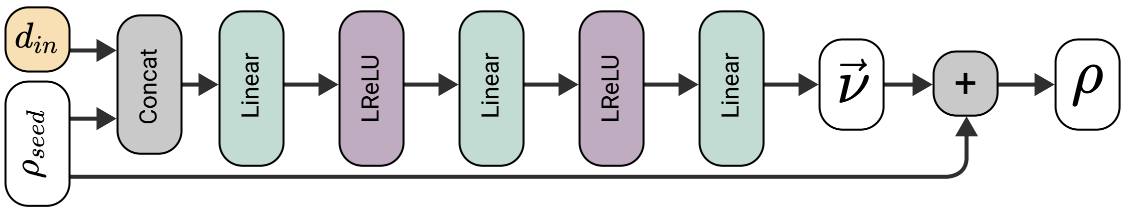


Figure 4.2: Architecture of the seeded generator

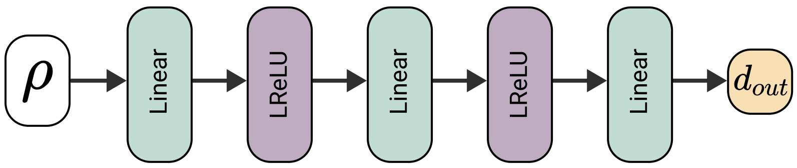
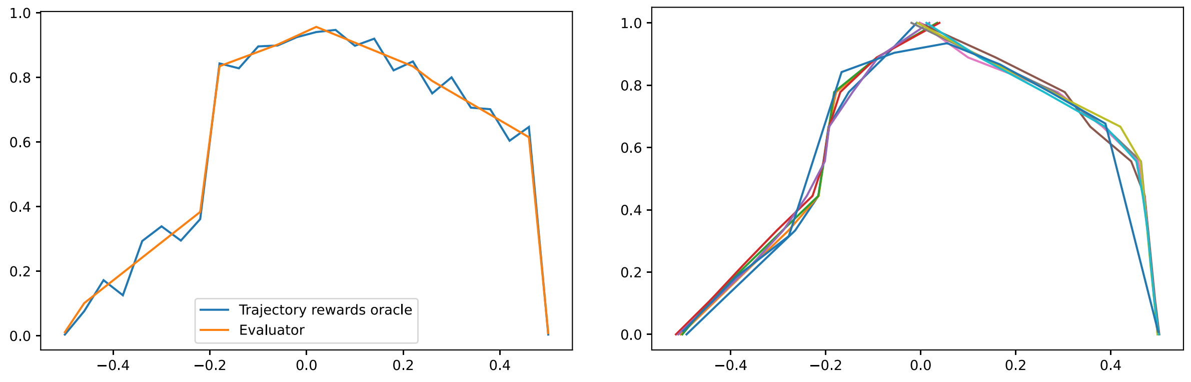


Figure 4.3: Architecture of the evaluator

On each training step, we have generated a batch of random difficulties d_{in} and seed points ρ_{seed}, from which the generator has generated the offsets $\vec{\nu}$, and then the embeddings ρ. Those embeddings were then evaluated by both the oracle, and the evaluator, which output the difficulty d_{out}, and its approximation $\hat{d}_{out}$, respectively. The evaluator was then trained to minimize the difference $|d_{out} - \hat{d}_{out}|$, and the generator was trained to minimize the $|d_{in} - \hat{d}_{out}|$. We used the Adam optimizer [33] with a starting learning rate of 0.01. We then decreased the learning rate on the generator, as well as decreased the weights of the offsets loss every 1000 steps to converge to the better results (Figure 4.4b).



(a) NN Evaluator approximating the trajectory rewards oracle

(b) Generator training form the evaluator, which is trained from the trajectory rewards oracle. Decaying offsets loss and learning rate was used to achieve these results.

4.2.2 Oracles

Heuristic oracle

To have a reasonable baseline, we have implemented a heuristic oracle (which uses a handcrafted function) to evaluate the difficulty of a single enemy environment. This function essentially compares how much space is available on either side of the enemy and whether it is enough for the pendulum to pass past it without colliding. It is simplistic in the sense that it completely disregarded things like the enemy's y coordinate, various speeds, starting angle of the pendulum. However, it has served its purpose well in being the simplest way of difficulty evaluation, to which we could compare the other methods.

Trajectory rewards oracle using heuristic agents

To be used in the trajectory rewards oracle, we implemented a heuristic agent that used a hand-crafted policy. This agent can be used within an environment with multiple enemies. In essence, the agent looked at the distance to the closest enemy:

- If the distance was larger than a threshold, the agent did nothing
- If the distance was below the threshold and would become smaller in the next step (depending on the current angular and vertical speeds), then the agent changed the swing direction
- Otherwise, if the distance would become larger, the agent would do nothing

This simple agent has also proven to be very useful - both as a baseline for comparing RL agents and also as an agent that was later used in the difficulty evaluation by the trajectory rewards. It is also simple to change the skill of the agent by inducing randomness in its actions. With an agent's $skill$ ranging from 0 to 1, the agent's randomness is $1 - skill$. Thus with the probability $1 - skill$, the random action is taken. And otherwise, the action is taken w.r.t. the policy described above. We used 3 different agents - with skills 1.0, 0.5, 0.0, and weighting 0.5, 0.25, 0.25, respectively - in the trajectory rewards oracle. Figure 4.5 shows the generated levels using this technique.

Speeding up

Using the trajectory rewards oracle means sampling a large number of trajectories:

- Several evaluations for the same agent, due to the possible stochasticity in the environment and/or the agent's policy
- Each state had also to be evaluated by several different levels of the different skills
- With the probabilistic generator, we also need to evaluate multiple ($N_{embeddings}$) embeddings

This produces $N_{agents} \cdot N_{evaluations}$ (or $N_{embeddings} \cdot N_{agents} \cdot N_{evaluations}$ with the REINFORCE) number of trajectories to evaluate, where each trajectory evaluation is also relatively costly - even with the usage of heuristic agents. Practically, on our machine, it took about 15 minutes for an epoch to complete, where the sample size and number of evaluations were set as low as possible to still be able to converge.

We used numba [1] to compile our Python code into a high-performant one. With parallelization, it resulted in an x500 speedup in our environment, or 1.8 seconds per epoch.

4.2.3 Q-values oracle and Deep Q-learning

The next step was to use a more sophisticated difficulty evaluation method using Q-values. To do this, we required DQN agents. We have found it crucial to reduce the sparsity of meaningful episodes - essentially by increasing the delta time between the transitions (this resembles the effect of dropping several frames) - to successfully train our agents. The rest of the DQN algorithm was pretty trivial: we have used a memory buffer for experience replay with a capacity for 10000 transitions, and a target network that we have kept in sync using the polyak averaging with a factor of $\lambda = 0.95$ for stability [34]:

$$\theta_{target} = \lambda\ \theta_{target} + (1 - \lambda)\ \theta$$

Then we have plugged the agents into our evaluator, but it seemed to have produced non-comparable results (see Figure 3.8). The normalization of the Q-values squashed all Q-values to be very close to each other, and thus the difficulty was almost always near zero. Sadly, we did not manage to come up with a solution for this problem, however, we hypothesize that perhaps a different normalization of Q-values should be used. Perhaps, for example, the Q-values, obtained from the agents with lower weighting should be discarded or weighted less.

4.3 Increasing the environment's complexity

Lastly, we have examined the scalability of our solution. We have increased the number of enemies in the pendulum environment to make our embedding space 2-, and then 3-dimensional. Figure 4.6 shows the losses during the training for the 2-enemies pendulum environment, where the trajectory rewards oracle was used to train the NN evaluator. We conducted the experiments using the trajectory rewards oracle. Unfortunately, with the higher-dimensional embedding space, it becomes extremely challenging to make sense

[1]https://numba.pydata.org/

of the results, it is difficult to determine the quality from the loss alone. This leads us to the already mentioned problem of searching for a type of meaningful compression of seeds into the $[0, 1]$ interval. We include the images of the generated levels as an appendix to the work. There are essentially a 1000 generated levels of 2-enemies pendulum level, split into 10 images. Each separate image has a fixed 1st dimension of the seed, where an image itself contains images like the Figure 4.5.

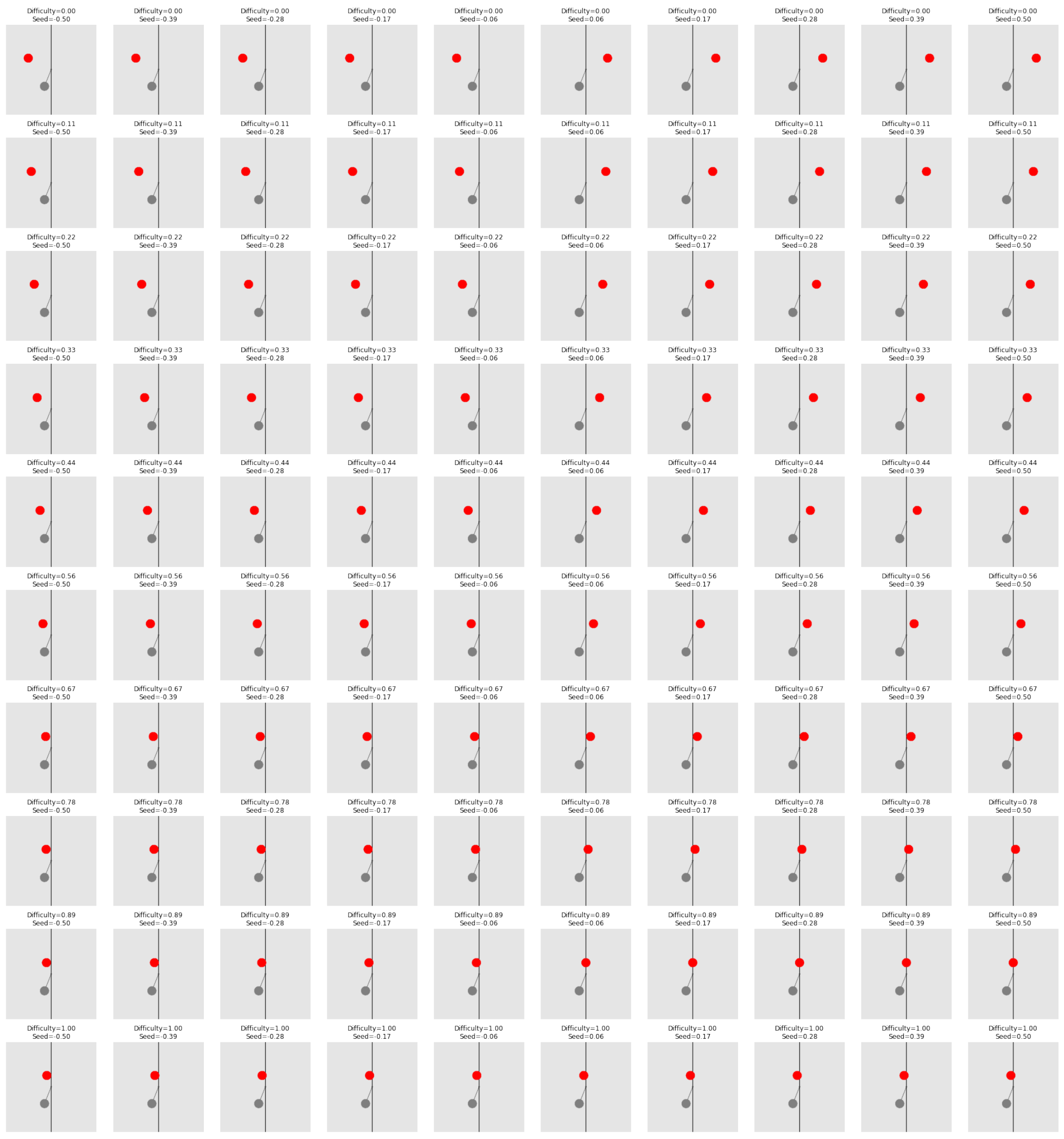


Figure 4.5: Generated levels with a single enemy, using the trajectory rewards oracle. Each row represents levels, generated with the same difficulty, but different seeds. We can see that with the lower difficulty the enemy gets placed more off to the side (top rows), but the seed can determine whether it gets placed on the left, of the right side. As the difficulty increases (bottom rows), the enemy gets placed more in the middle. This corresponds with the difficulty, provided by our oracle.

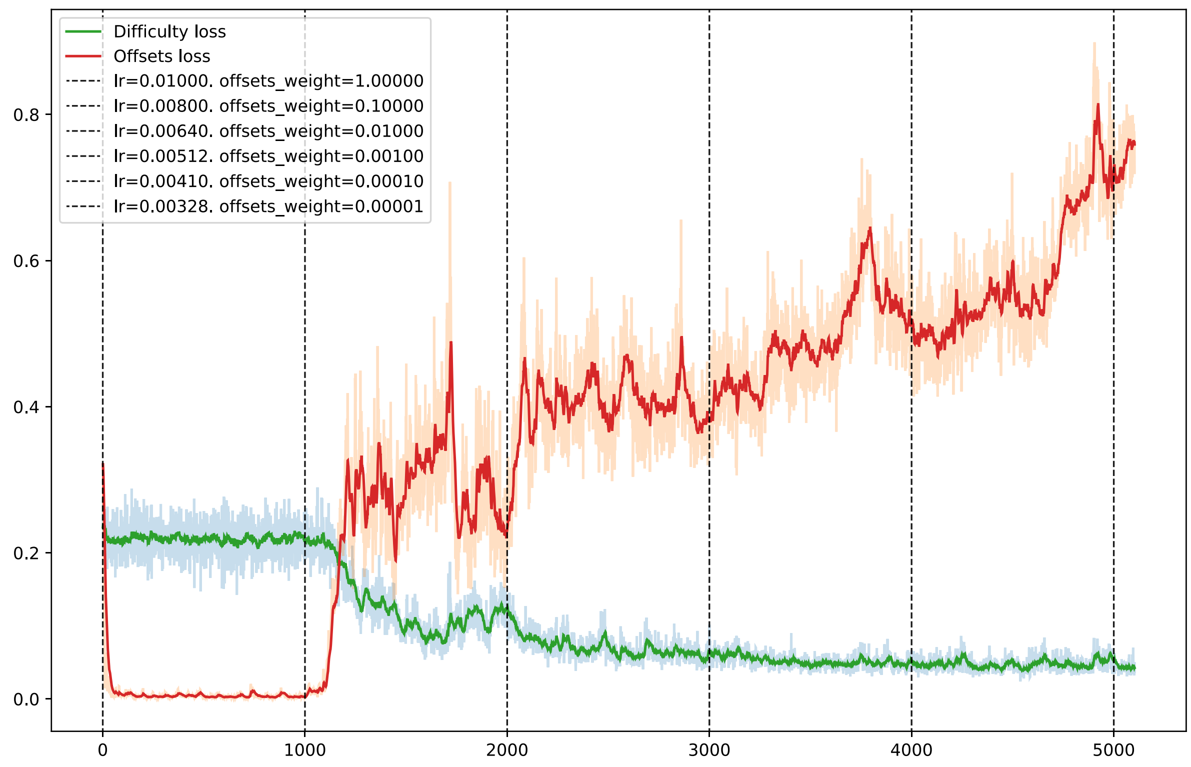


Figure 4.6: Decaying difficulty and offsets loss while training the generator to produce multi-level pendulum levels

Chapter 5

Conclusions

5.0.1 Theoretically

We introduced the novel formulation of generating game levels as a search of a mapping from difficulty to a game level. We proposed a general framework to solve the formulated task, consisting of a generator network, an evaluator network, and an oracle. We explored the meaning of difficulty, we have proposed several ways of domain-free difficulty evaluation methods using AI agents: using trajectory rewards, Q-values, action probabilities, and entropy. We have introduced a problem of diversity - searching for many different levels with the same difficulty. We have proposed an (N-dimensional) seeded generator as a solution to the diversity problem. By integrating the level generation into a system with a dynamic difficulty evaluation of the player's skill, and thus, required difficulty, we can potentially generate much more entertaining games via dynamic difficulty adjustment (DDA) [2], [35].

5.0.2 Practically

We have shown the proposed framework in action on our test environment using the heuristic oracle, and trajectory rewards oracle. We were also able to accomplish the goal of diversity - generating several different levels of the same difficulty. This introduces a baseline for future work and improvements.

5.0.3 Further work

Our experiments were conducted in a simple testing environment. Thus, it's essential to explore how the method works on more complex ones. As we have slightly increased the complexity of our environment by increasing the number of enemies (increasing the embedding space dimensionality from 1 to 2, and later, 3), several problems have emerged:

1. The number of trajectories, required to be evaluated by the oracle's AI agents grows exponentially. This leads to the problem of efficient embedding space exploration to decrease the time of training.

2. Results visualization - in 2-dimensional embedding space it is already not clear how to evaluate the generator's results, aside from comparing the loss and evaluating the levels empirically using the real human players.

Another important task is to train the AI agents correctly and efficiently during the phase when the generator is not yet capable of producing valid levels. Then, we have also observed that the seeded generator approach has difficulties producing valid results where the generated offsets have to be large. Thus, further work in that direction is also required. Lastly, the N-dimensional seeded generator is difficult to use in practice - we should try to compress the important information into a $[0, 1]$ interval that is easy to sample from.

Appendix A

Extra 1

Here we provide short proof for the Equation 3.1. We have claimed that normalization can occur after calculating the difficulty of the state. Let N_s be the number of actions $a_1, a_2, ..., a_{N_s}$ in the state s, then:

$$
\begin{aligned}
d_s &= \max_{a_j \sim s}\left[\frac{Q(s,a_j) - Q_{min}}{Q_{max} - Q_{min}}\right] - \mathop{\mathbb{E}}_{a_j \sim s}\left[\frac{Q(s,a_j) - Q_{min}}{Q_{max} - Q_{min}}\right] \\
&= \frac{\max\limits_{a_j \sim s} Q(s,a_j) - Q_{min}}{Q_{max} - Q_{min}} - \frac{1}{N_s}\sum_{j=1}^{N_s}\left[\frac{Q(s,a_j) - Q_{min}}{Q_{max} - Q_{min}}\right] \\
&= \frac{\max\limits_{a_j \sim s} Q(s,a_j) - Q_{min}}{Q_{max} - Q_{min}} - \frac{\frac{1}{N_s}\Big(\sum\limits_{j=1}^{N_s} Q(s,a_j) - N_s Q_{min}\Big)}{Q_{max} - Q_{min}} \\
&= \frac{\max\limits_{a_j \sim s} Q(s,a_j) - Q_{min} - \frac{1}{N_s}\Big(\sum\limits_{j=1}^{N_s}\Big[Q(s,a_j)\Big] - N_s Q_{min}\Big)}{Q_{max} - Q_{min}} \\
&= \frac{\max\limits_{a_j \sim s} Q(s,a_j) - \cancel{Q_{min}} - \frac{1}{N_s}\sum\limits_{j=1}^{N_s}\Big[Q(s,a_j)\Big] + \cancel{Q_{min}}}{Q_{max} - Q_{min}} \\
&= \frac{\max\limits_{a_j \sim s} Q(s,a_j) - \mathbb{E}_{a_j \sim s}\, Q(s,a_j)}{Q_{max} - Q_{min}} = \frac{d'_s}{Q_{max} - Q_{min}}
\end{aligned}
$$

Bibliography

[1] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, *et al.*, "Language models are few-shot learners", *arXiv preprint arXiv:2005.14165*, 2020.

[2] G. K. Sepulveda, F. Besoain, and N. A. Barriga, "Exploring dynamic difficulty adjustment in videogames", *2019 IEEE CHILEAN Conference on Electrical, Electronics Engineering, Information and Communication Technologies (CHILECON)*, 2019. DOI: `10.1109/chilecon47746.2019.8988068`. [Online]. Available: `http://dx.doi.org/10.1109/CHILECON47746.2019.8988068`.

[3] A. Summerville, S. Snodgrass, M. Guzdial, C. Holmgård, A. K. Hoover, A. Isaksen, A. Nealen, and J. Togelius, *Procedural content generation via machine learning (pcgml)*, 2018. arXiv: `1702.00539 [cs.AI]`.

[4] S. Risi and J. Togelius, *Increasing generality in machine learning through procedural content generation*, 2020. arXiv: `1911.13071 [cs.AI]`.

[5] S. Dahlskog and J. Togelius, "Patterns and procedural content generation: Revisiting mario in world 1 level 1", May 2012. DOI: `10.1145/2427116.2427117`.

[6] S. Dahlskog and J. Togelius, "Patterns as objectives for level generation", 2013.

[7] S. Dahlskog and J. Togelius, "A multi-level level generator", Aug. 2014. DOI: `10.1109/CIG.2014.6932909`.

[8] S. Dahlskog, J. Togelius, and M. Nelson, "Linear levels through n-grams", Nov. 2014. DOI: `10.1145/2676467.2676506`.

[9] A. Summerville, S. Philip, and M. Mateas, "Mcmcts pcg 4 smb : Monte carlo tree search to guide platformer level generation", 2015.

[10] A. Summerville and M. Mateas, *Super mario as a string: Platformer level generation via lstms*, 2016. arXiv: `1603.00930 [cs.NE]`.

[11] M. Guzdial and M. Riedl, *Toward game level generation from gameplay videos*, 2016. arXiv: `1602.07721 [cs.AI]`.

[12] V. Volz, J. Schrum, J. Liu, S. M. Lucas, A. Smith, and S. Risi, *Evolving mario levels in the latent space of a deep convolutional generative adversarial network*, 2018. arXiv: `1805.00728 [cs.AI]`.

[13] R. R. Torrado, A. Khalifa, M. C. Green, N. Justesen, S. Risi, and J. Togelius, *Bootstrapping conditional gans for video game level generation*, 2019. arXiv: `1910.01603 [cs.NE]`.

[14] A. Aubret, L. Matignon, and S. Hassas, *A survey on intrinsic motivation in reinforcement learning*, 2019. arXiv: `1908.06976 [cs.LG]`.

[15] J. Brant and K. Stanley, "Minimal criterion coevolution: A new approach to open-ended search", Jul. 2017, pp. 67–74. DOI: `10.1145/3071178.3071186`.

[16] R. Wang, J. Lehman, J. Clune, and K. O. Stanley, *Paired open-ended trailblazer (poet): Endlessly generating increasingly complex and diverse learning environments and their solutions*, 2019. arXiv: `1901.01753 [cs.NE]`.

[17] S. Sukhbaatar, Z. Lin, I. Kostrikov, G. Synnaeve, A. Szlam, and R. Fergus, *Intrinsic motivation and automatic curricula via asymmetric self-play*, 2018. arXiv: `1703.05407 [cs.LG]`.

[18] P. Bontrager and J. Togelius, *Fully differentiable procedural content generation through generative playing networks*, 2020. arXiv: `2002.05259 [cs.AI]`.

[19] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, *Generative adversarial networks*, 2014. arXiv: `1406.2661 [stat.ML]`.

[20] M. Dennis, N. Jaques, E. Vinitsky, A. Bayen, S. Russell, A. Critch, and S. Levine, *Emergent complexity and zero-shot transfer via unsupervised environment design*, 2021. arXiv: `2012.02096 [cs.LG]`.

[21] S. Earle, *Using fractal neural networks to play simcity 1 and conway's game of life at variable scales*, 2020. arXiv: `2002.03896 [cs.LG]`.

[22] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, *Asynchronous methods for deep reinforcement learning*, 2016. arXiv: `1602.01783 [cs.LG]`.

[23] L. D. Liello, P. Ardino, J. Gobbi, P. Morettin, S. Teso, and A. Passerini, *Efficient generation of structured objects with constrained adversarial networks*, 2020. arXiv: `2007.13197 [cs.LG]`.

[24] J. Xu, Z. Zhang, T. Friedman, Y. Liang, and G. V. den Broeck, *A semantic loss function for deep learning with symbolic knowledge*, 2018. arXiv: `1711.11157 [cs.AI]`.

[25] A. Darwiche and P. Marquis, "A knowledge compilation map", *Journal of Artificial Intelligence Research*, vol. 17, 229–264, 2002, ISSN: 1076-9757. DOI: `10.1613/jair.989`. [Online]. Available: `http://dx.doi.org/10.1613/jair.989`.

[26] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning", *Machine Learning*, vol. 8, pp. 229–256, 1992.

[27] A. O'Hagan and T. Leonard, "Bayes estimation subject to uncertainty about parameter constraints", *Biometrika*, vol. 63, pp. 201–203, 1976.

[28] S. Roohi, A. Relas, J. Takatalo, H. Heiskanen, and P. Hämäläinen, "Predicting game difficulty and churn without players", *Proceedings of the Annual Symposium on Computer-Human Interaction in Play*, 2020. DOI: `10.1145/3410404.3414235`. [Online]. Available: `http://dx.doi.org/10.1145/3410404.3414235`.

[29] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, *Playing atari with deep reinforcement learning*, 2013. arXiv: `1312.5602 [cs.LG]`.

[30] Y. Zhao, I. Borovikov, F. de Mesentier Silva, A. Beirami, J. Rupert, C. Somers, J. Harder, J. Kolen, J. Pinto, R. Pourabolghasem, J. Pestrak, H. Chaput, M. Sardari, L. Lin, S. Narravula, N. Aghdaie, and K. Zaman, *Winning isn't everything: Enhancing game development with intelligent agents*, 2020. arXiv: `1903.10545 [cs.AI]`.

[31] S. Ariyurek, A. Betin-Can, and E. Surer, *Automated video game testing using synthetic and human-like agents*, 2019. arXiv: `1906.00317 [cs.AI]`.

[32] B. Xu, N. Wang, T. Chen, and M. Li, *Empirical evaluation of rectified activations in convolutional network*, 2015. arXiv: `1505.00853 [cs.LG]`.

[33] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2017. arXiv: `1412.6980 [cs.LG]`.

[34] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms", in *Proceedings of the 31st International Conference on Machine Learning*, E. P. Xing and T. Jebara, Eds., ser. Proceedings of Machine Learning Research, vol. 32, Bejing, China: PMLR, 2014, pp. 387–395. [Online]. Available: `http://proceedings.mlr.press/v32/silver14.html`.

[35] H.-S. Moon and J. Seo, "Dynamic difficulty adjustment via fast user adaptation", *Adjunct Publication of the 33rd Annual ACM Symposium on User Interface Software and Technology*, 2020.