Learning Game Design

Roger Liu
rogerliu@andrew.cmu.edu

Stephen Chen
stephen1@andrew.cmu.edu

Abstract

Action video games feature many game variables such as monster health and damage which control the difficulty of the game. To help automate the search for parameters which provide reasonable challenge to players, we present "Dungeon Master," a Reinforcement Learning agent which uses Continuous Actor Critic to find a game parameters that meet some high level, designer specified difficulty. We implemented a minimal, turn-based action game featuring rooms of monsters whose game statistics are controlled by the Dungeon Master. We created a novel variable update method, which allows our agent to step game variables towards a configuration that meets a difficulty specification, measured in player deaths per room. For a 6 room "dungeon" in our game, Dungeon Master performs as well as a human designer in finding game parameters that meet a specified difficulty for AI player model.

1 Introduction

In the process of creating video games, a designer's intentions are not always reflected in the resulting gameplay. A boss monster in an action game might be frustratingly hard, or a particular strategy might allow the player to breeze through the game. To address this, game developers undergo multiple cycles of "playtesting" in order to tune the game's difficulty and resulting player experience. This process is expensive and time consuming, as developers must be on hand to observe the playtesting sessions and identify where the game's design fails to create the appropriate amount of challenge. This time investment is greatly exacerbated in Role-Playing Games (RPGs), as the interactions between various character statistics (like health, strength, defense) and gameplay mechanisms can become insurmountably complex, making it difficult for a designer to identify and correct the root of the problem [5]. In addition, RPGs usually allow the player to customize their character, specializing their skillsets to deal with different situations as they progress through the game. This is especially difficult to design around, as designers cannot know what tools a player has going into a challenge.

While designers now have many more tools and systems available to make playtesting and data collection easier, there are few systems for automating the difficulty adjustment process. Existing literature mostly focuses on a concept called Dynamic Difficulty Adjustment (DDA). In these systems, the difficulty of the game changes dynamically while a player is playing. For instance, Hunicke and Chapman proposed making reactive (immediate) and proactive (long term) changes to the environment of a first-person shooter game based on the probability of the player's health dropping to dangerous levels [2]. Andrade and Corruble dynamically adjusted the difficulty of a fighting game AI by using reinforcement learning to learn optimal state-action values, but then had the AI use suboptimal actions based on a "challenge function" which scored how well the player was doing [1]. There are other approaches with adjustment techniques other than RL [6, 4], and different player feedback functions [3, 8].

Though these systems are able to change a game's difficulty to provide an appropriate level of challenge to the player, they would not perform well in RPGs. This is because existing DDA algorithms determine difficulty based on a simplified model of game dynamics. For example, in a first-person shooter, the difficulty can be determined just based on how much damage a player is expected to deal vs. how much they are expected to take. These models are acceptable for games where the character statistics are generally simple. In a first-person shooter and fighting game, damage dealt is usually fixed and there are no tools that greatly upset the expected outcome of a situation. However, in an RPG, the player character is expected to grow in strength and gain more tools to work with. Their statistics are no longer fixed, and the interaction between various items and gameplay mechanisms makes it difficult for a DDA with a simple dynamics model to understand when a situation is actually difficult for the player. If such a complex model is known to begin with, then DDA will not even be needed, as developers themselves will understand how to tune the difficulty of a game. In addition, because of the transparent nature of stats in most RPGs, changes made by DDA will be very apparent to the players, which may make them feel cheated out of overcoming the game's challenge. As a result, the difficulty of an RPG is best tuned offline, where the full gameplay model can be simulated many times without having the changes be made obvious to the player.

This paper outlines our implementation of a reinforcement learning agent which learns to tweak a game environment so that the resulting gameplay difficulty closely matches the intentions of the designers, thereby automating the long iterative process of playtesting. To do this, we designed an agent (dubbed the "Dungeon Master" or DM) to adjust various parameters of an environment (known as the "Dungeon") that players navigated. The DM was rewarded when it tuned the environment such that the game difficulty matched the designer's intent (i.e. player dies around x times). This problem was tackled through the lens of reinforcement learning. The MDP states were the statistics of the enemies in the dungeon, and the actions were adjustments of those statistics.

2 Methods

2.1 Game Environment

We created our own minimalistic, turn-based RPG game for the Dungeon Master to tune. In this game, the player character progressed through multiple "rooms" that composed a "dungeon." The player would fight monsters in each room. Combat proceeded by having the player and opponents take turns performing actions. The player could choose to attack with one of their weapons or drink one of their limited number of potions to recover their health. Damage was calculated based on the player's base strength and a damage value associated with the weapon used. There were two weapons available: a longsword and a greatsword. The player chose which weapon to start with, and could obtain the other randomly upon defeating monsters. The longsword could only hit a single monster, but it did more damage than the greatsword. Comapratively, the greatsword could hit all monsters in the room, but dealt less damage per monster. The player also gained experience per monster defeated, amount varying based on the monster. At given experience point thresholds, the player would earn a level, gaining a randomized amount of maximum health and base strength.

The monsters always performed basic attacks, unless a more powerful special attack was available. Each monster's special attack was available every f turns. When a monster died, they had a chance to randomly drop each weapon and a chance to randomly drop a potion. When all monsters in a room were defeated, the player moves on to the next room. It is important to note that the players health and potions do not recover when going between rooms.

The game parameters that the Dungeon Master was allowed to tune were, for each monster: normal attack damage, special attack damage, special attack frequency f, probability of dropping a potion on death, probability of dropping a longsword on death, probability of dropping a greatsword on death.

If the player was defeated during combat, their death was recorded and they were immediately restored to their maximum health and given the starting number of potions. They kept their levels, experience points, and weapons found. The record of player deaths per room served as our measurement of "difficulty" of the dungeon.

Once we implemented the framework for the game, we needed a player to play through it. While we would ideally want real people playing through the game, doing so would not be time-efficient or accurately capture a consistent style of play. Thus, we implemented a simple greedy player, who attacked every turn and immediately used a potion when its health dropped below a preset threshold.

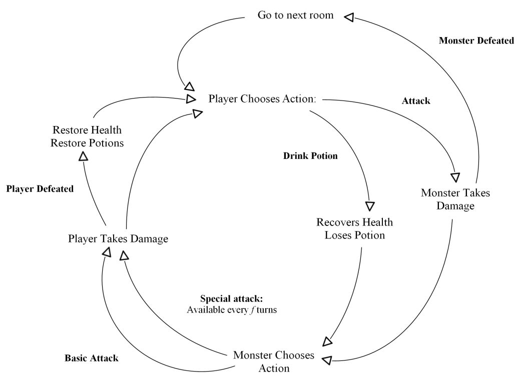

Figure 1: Diagram of Game Flow

This greedy player used the longsword when there was a single enemy, and a greatsword when there was more than one, given that it had the weapon.

We then formulated the problem of adjusting the game as an RL problem.

We designated $x_i^{(k)}$ as the kth monster's ith stat. To be concrete, for monster k:

1. $x_1^{(k)}$ was normal attack damage
2. $x_2^{(k)}$ was special attack damage
3. $x_3^{(k)}$ was special attack frequency f
4. $x_4^{(k)}$ was probability of dropping a potion on death
5. $x_5^{(k)}$ was probability of dropping a longsword on death
6. $x_6^{(k)}$ was probability of dropping a greatsword on death

Each statistic had a minimum and maximum value $x_{i,min}, x_{i,max}$. These were specified by the designers. Examples included clipping probabilities between 0.0 and 1.0.

We normalized these stats to [-1, 1] by doing the following:

$$\widetilde{x_i^{(k)}} = 2\frac{x_i^{(k)} - x_{i,min}}{x_{i,max} - x_{i,min}} - 1$$

A state s was a configuration of the normalized stats of the monsters in the dungeon, represented as a vector.

$$s = \left\langle \widetilde{x_1^{(1)}}, \ldots, \widetilde{x_6^{(1)}}, \widetilde{x_1^{(2)}} \ldots \right\rangle$$

An action was an adjustment of those stats. It was given by:

$$a = \langle m_1^{(1)}, d_1^{(1)}, \ldots, m_6^{(1)}, d_6^{(1)}, m_1^{(2)}, \ldots \rangle$$

To apply an adjustment to a stat, we did the following update:

$$x_i^{(k)} \leftarrow 1.5^{m_i^{(k)}} \cdot x_i^{(k)} + d_i^{(k)}$$

clipping at $x_{i,min}$ or $x_{i,max}$ if necessary. Intuitively, m determined a scaling factor and d was a flat adjustment. We chose this update rule because it centers each action dimension around 0. Positive values increase the statistic, while negative values decrease the statistic.

The reward of moving to a state was the negative squared euclidean distance between the designer's intended number of deaths per room and the number of deaths that the player model experienced in a playthrough with the new configuration. Intutively, we wanted to capture the difference between the designer's intended difficulty and the measured difficulty of the dungeon.

$$X_s = [\text{room 1 deaths}, \text{room 2 deaths}, ...]$$
$$\mathcal{R}(s, a, s') = -\|X_{s'} - target\|^2$$

We note these states are multi-dimensional continuous variables, and the actions too are multi-dimensional continuous values.

2.2 Dungeon Master

We implemented the DM with the Action-Value Actor-Critic method. Specifically we used a variant called Continuous Actor Critic Learning Automaton (CACLA) [7].

Action-Value Actor-Critic is a model-free reinforcement algorithm which relies on two function approximators. The actor function is a policy gradient method, which learns a policy distribution for each state, and the crtic function learns the value of state-action pairs. At a high level, the actor decides the action to take, and the critic evalutes the result of the action. The evaluation is then used to update the weights of the two approximators.

Let Q_w represent the critic function with parameters w, and $Q_w(s, a)$ represent the approximate value of taking action a at state s. Values of state-action pairs represent the expected discounted total reward earned by the agent by from state s, given some discount factor γ. Temporal difference learning describes a method to learn the weights w. Let δ_t be the TD-error at time step t, defined as:

$$\delta_t = R(s_t, a_t, s_{t+1}) + \gamma Q_w(s_{t+1}, a_{t+1}) - Q_w(s_t, a_t)$$

Then the update rule for the weights w, for some learning rate β is given by:

$$w_{t+1} := w_t + \beta \delta_t \nabla_w Q_w(s_t, a_t)$$

Let $\pi_\theta(s)$ represent the agent's policy distribution for state s using function approximator weights θ. In the original action-value Actor-Critic, the update rule at timestep t for θ, for some learning rate α is given by:

$$\theta_{t+1} := \theta_t + \alpha \delta_t \nabla_{\theta_t} \log \pi_{\theta_t}(s_t, a_t)$$

In the CACLA variant, we only update the policy weights θ when the TD error is positive, that is $\delta_t > 0$.

In addition, our policy weight update is instead:

$$\theta_{t+1} := \theta_t + \alpha n \nabla_{\theta_t} \log \pi_{\theta_t}(s_t, a_t)$$

where $n = \lceil \delta_t / \sqrt{\mathsf{var}_t} \rceil$ and var_t is the running average of the variance of δ.

Intuitively, the first change means that we only update the actor's policy towards actions that give us positive returns, and we never penalize the policy for giving us bad returns. The second change means that we update the actor's policy more than once when the action that yielded a higher than typical TD error.

The actor and critic are backed by simple linear models. The critic is

$$Q_w(s, a) = w^T \phi(s, a)$$

where ϕ just flattens the input vectors together.

The actor is given by

$$\mu_\theta(s) = \theta_w s + \theta_b \qquad a \sim \mathcal{N}(\mu(s), \sigma^2)$$

where θ_w is a matrix of weights, θ_b, μ, a, σ are vectors, and a is the actual action taken. σ here represents "Gaussian exploration." Higher values of σ will cause the actor to explore actions further away than the one given by μ. Values for each component in σ are given by the game, as the game understands what's a reasonable exploration range for each action.

The σ values are scheduled to linearly decay to 0.01 of their original value over the first million steps. This effectively reduces the exploration, and implies a stronger belief in the actor's chosen action. This is partly to mimic a linearly decreasing ϵ in ϵ-greedy, but also to ensure that the actor is taking smaller steps once the critic has a good approximation of the state space.

The Dungeon Master is not made aware of each stat's minimum and maximum value when picking its continuous actions. As according to the CACLA paper, this should not be an issue.

3 Results

3.1 Test 6-Room Dungeon

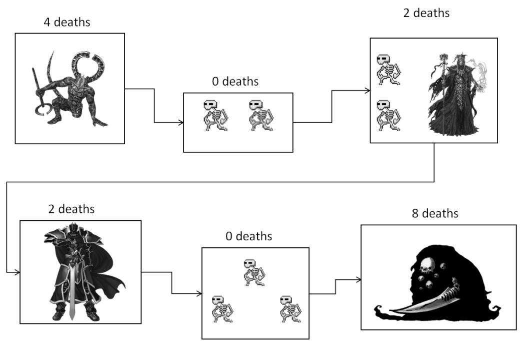

Figure 2: Visual layout of our test 6-room dungeon

We ran the DM on a 6-room "dungeon" with the following monsters and difficulty specification:

1. Room 1 - Hard Difficulty (4 deaths)
    (a) "Demon" - 80 HP
2. Room 2 - Easy Difficulty (0 deaths)
    (a) "Skeleton 2A" - 20 HP
    (b) "Skeleton 2B" - 20 HP
3. Room 3 - Medium Difficulty (2 deaths)
    (a) "Skeleton 3A" - 20 HP
    (b) "Skeleton 3B" - 20 HP
    (c) "Lich" - 40 HP
4. Room 4 - Medium Difficulty (2 deaths)
    (a) "Knight" - 60 HP
5. Room 5 - Easy Difficulty (0 deaths)
    (a) "Skeleton 5A" - 20 HP
    (b) "Skeleton 5B" - 20 HP
    (c) "Skeleton 5C" - 20 HP
6. Room 6 - Very Hard Difficulty (8 deaths)
    (a) "Boss" - 200 HP

After running, the DM concluded with game parameters shown in Table 1. We ran our Greedy Player on this final configuration for 100 test plays and noted the average deaths per room. We then asked human players to play through the configuration. We recorded 11 human samples, and noted the average deaths per room. For a comparison, we asked a colleague to attempt manually tune the dungeon to match our specifications. Like the DM, they were allowed to tune the parameters and then run our Greedy AI Player to check the estimated death count. They took 31 iterations before settling on a final configuration. We ran our Greedy Player 100 times on this configuration, and also asked human players to try our colleague's configuration. The average number of deaths per room for the DM's and our colleague's configuration, as experienced by the Greedy Player and by human players, is given in Table 2. The negative squared euclidean distance of the results from the specification is given in the “Score” row.

Between the Dungeon Master’s configuration and our colleague’s configuration, we found that the two created similar difficulty for the Greedy AI Player. The DM performed slightly better though, as it ensured that only two rooms had average deaths different from specification, whereas our colleague’s results differed from specification in 3 rooms.

When a human player was used to test the configuration, we found that our Dungeon Master performed fairly worse than the human configuration. This was likely because the Greedy Player model did not play as optimally as a human, and thus the DM mistakenly undertuned the difficulty. However, the Dungeon Master still maintained high difficulty in the “Demon Room” and “Boss Room.” Although not perfect, it still created a death distribution which was generally followed our specification.

Table 1: DM Produced Configuration of a 6-Room Dungeon

Monster	Norm Dmg	Spec Dmg	Spec Freq	Potion Drop	L.Sword Drop	G.Sword Drop
Demon	10.2	1.0	12	1.00	0.00	1.00
Skeleton 2A	1.0	8.9	5	0.08	0.89	0.12
Skeleton 2B	3.5	5.0	6	0.17	1.00	0.80
Skeleton 3A	5.0	5.0	2	0.80	0.07	1.00
Skeleton 3B	1.4	10.0	10	0.17	0.42	0.00
Lich	6.1	50.5	8	0.32	1.00	0.44
Knight	7.1	24.3	3	0.70	0.17	0.98
Skeleton 5A	2.2	5.0	8	0.69	0.00	1.00
Skeleton 5B	2.5	5.0	20	1.00	1.00	0.53
Skeleton 5C	2.1	5.0	1	1.00	0.24	0.17
Boss	14.0	100.0	1	1.00	0.86	1.00

Table 2: Average Dungeon Deaths Per Room

		Greedy Player		Human Player	
Room Name	Specification	Dungeon Master	Human Config	Dungeon Master	Human Config
Demon Room	4	3.50	4.45	3.36	4.20
Skeleton Room	0	0.00	0.00	0.00	0.00
Necromancer Room	2	1.27	2.00	0.55	2.60
Knight Room	2	2.00	2.52	1.00	1.60
Skeleton Room	0	0.00	0.00	0.45	0.00
Boss Room	8	8.00	8.52	8.55	8.60
Score	0.0	-0.78	-0.74	-4.01	-0.92

3.2 Analysis of DM Chosen Parameters

Closer inspection of each monster’s parameters shows that the DM made coherent decisions. In the 1st room, the Demon’s attack values properly match up with the ”hard” difficulty that was specified. In addition, the Demon also is guaranteed to drop a greatsword, something which will aid the player in the next rooms where there are multiple enemies. This falls in line with the idea that the later rooms are supposed to be easier. The weak skeleton monsters all have appropriately low damage values as the rooms that they are in are designated as ”easy”. In addition, the necromancer in the 3rd room of ”medium” difficulty has a very powerful special attack, even though the player is supposed to die very few times in that room. This setup matches what one would expect from this room, where there is a powerful ”leader” monster accompanied by weaker minion monsters. Lastly, the damage values of the Knight and the Boss appropriately match the desired difficulty of those rooms, with the Boss having extremely powerful and frequent special attack to ensure that the player dies 8 times as specified by the designer.

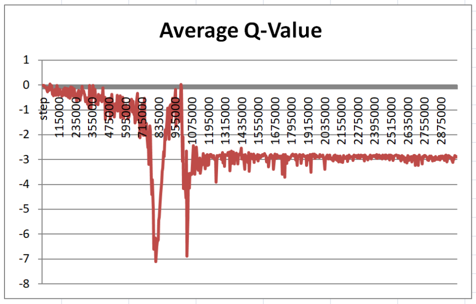

Figure 3: Q-Value

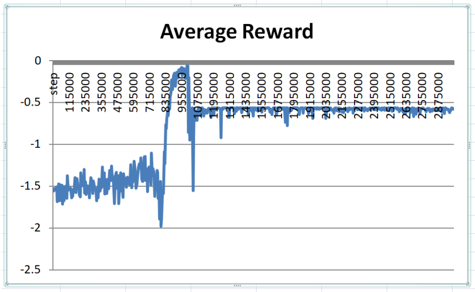

Figure 4: Average Reward

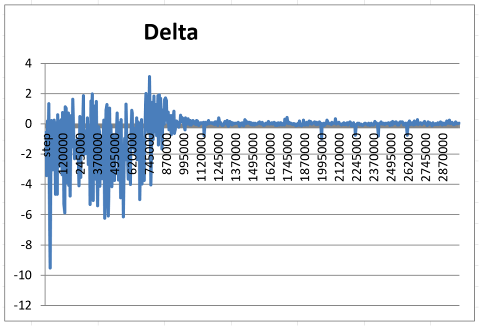

Figure 5: TD Error δ

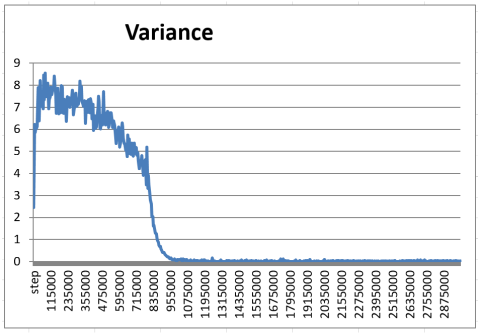

Figure 6: Variance of TD Error δ

3.3 Running the DM to Convergence

Figures 3-6 show the DM tuning a simpler 2-room dungeon. From this smaller example, we found that the agent eventually fit to stable values after running for many iterations. This shows that the agent knows how to navigate the state space and find better values. However, we noted in this case that the stable values the DM settled on did not meet designer specifications exactly (reward was not 0). We were able to circumvent this by having tweaking the agent to stop its tuning early when it hit the target goal.

3.4 Stopping at the Goal

Once the DM configured a dungeon layout to produce the designer specified number of deaths, we verified the DMs solution by running 100 playthroughs with our automated greedy player. If the greedy player consistently died the same number of times as the specification, we considered the DM's configuration as accurate and terminate. Otherwise, we let the DM continue to run.

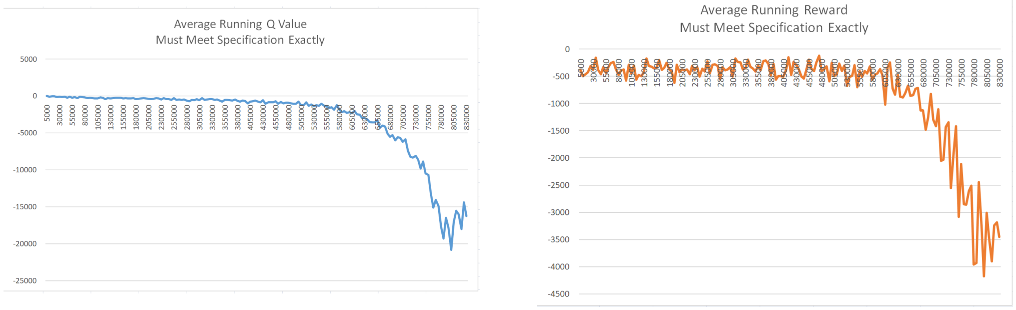

Figure 7: Average Running Reward and Q Value when searching for an exact match

When we required the DM to produce a configuration that exactly matched the designer specified number of deaths, we found that after many iterations, the Dungeon Master would fall into a local

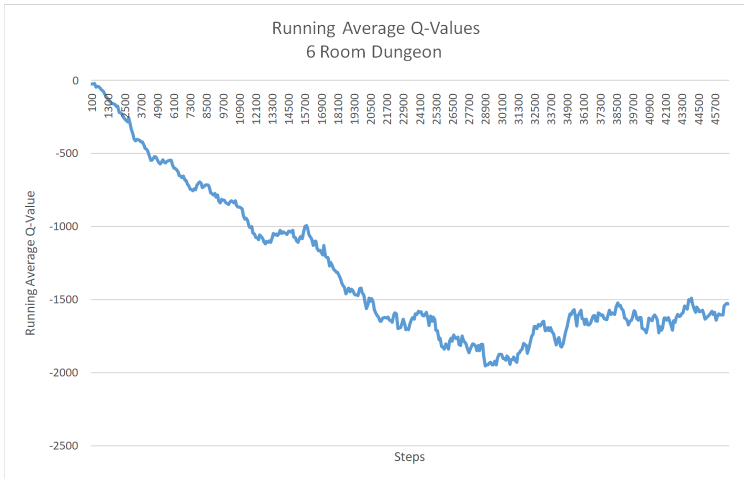

Figure 8: Average Q-Value when training with relaxed requirements

minima and be unable to produce the desired results. The diverging Q-Values are shown in Figure 7.

We then relaxed the constraints on the final dungeon configuration. The DM's final configuartion was allowed to differ from the designer specification by 1 death in at most 2 of the rooms. Under these relaxed constraints, the DM was able to find a solution in significantly fewer steps and its Q-Values graph suggested that it would converge. Figure 8 shows the average Q-values when running under these relaxed constraint. These Q-Values are associated with the 6-room dungeon configuration presented in the earlier sections.

4 Discussion

This project has shown that reinforcement learning can be used to translate high level designer specifications into a tangible implementation. Specifically, reinforcement learning techniques can be used to tune the parameters of a video game to match a designer's intended experience. In addition, this project has contributed a novel update function for automatically adjusting parameters. Specifically:

$$x_i^{(k)} \leftarrow 1.5^{m_i^{(k)}} \cdot x_i^{(k)} + d_i^{(k)}$$

where $m_i^{(k)}$ and $d_i^{(k)}$ are actions taken by the DM. This function proved to be more effective than a linear update like $x := mx + b$, under which, the agent tended to jump between the maximum and minimum values for each parameter. Our proposed function also makes intuitive sense, as the actions of multiplying by 2 and dividing by 2 are equidistant from 0.

As shown in our results, the Dungeon Master can find game parameters that work as well as those manually picked by a human designer, especially when trying to meet some death specification for our Greedy AI player. The DM's chosen parameters tended to not kill human players as often, because it had optimized difficulty for a less compentent AI player. We believe however, that even if the parameters do not exactly meet specification, they provide a good baseline for human designers to work off. Considering that the Dungeon Master found its game parameters much faster than our colleague, the Dungeon Master is likely to help save tremendous time when developers are tuning the difficulty of their games.

We believe that this current approach is still incomplete. For one, it relies on having an AI player go through a simulation of the game. This is not always feasible, as many games cannot yet be played by a competent AI. In addition, games which incorporate strategies involving timing and positioning cannot be adequately represented with fast, simple simulations. This significantly increases the time needed for the DM to achieve decent results, as the DM requires many iterations to locate good parameters. Though this could be remedied by making the simulation a simplified version of the game, the designer would have to worry about the values between two effectively different games. Lastly, the system is untested on the complex systems found in industry RPGs. We do not expect the current linear actor and critic models to perform well on those environments, though we think that deep models will be able to handle such complexity. Regardless, we believe that at the very least, our current iteration of Dungeon Master can provide a useful baseline configuration of parameters for a desired difficulty level.

As a final topic of discussion, there was a suggestion of tackling this problem through the lens of supervised learning. In this setting, the DM would be trained using pairs of dungeon configurations and the resulting measured difficulty. After training, the designer would then specify a desired difficulty and the DM would produce an appropriate configuration. There are some notable drawbacks to this approach. For one multiple dungeon configurations correspond to the same difficulty, making it hard to determine the "right" configuration for a given difficulty. In addition, it would take a significant amount of time to generate the test data, as it would have perform many simulations over the spectrum of possible difficulties. In comparison, reinforcement learning finds the dungeon configuration through essentially a bounded forward search. Though we acknowledge that this approach could potentially work, we feel that it will take longer to produce results of similar quality and will be more likely to produce bad configurations.

References

[1] Gustavo Andrade et al. "Challenge-sensitive action selection: an application to game balancing". In: *Intelligent Agent Technology, IEEE/WIC/ACM International Conference on*. IEEE. 2005, pp. 194–200.

[2] Robin Hunicke and Vernell Chapman. "AI for Dynamic Difficulty Adjustment in Games". In: *Proceedings of the Challenges in Game AI Workshop, Nineteenth National Conference on Artificial Intelligence (AAAI '04)*. 2004.

[3] Changchun Liu et al. "Dynamic difficulty adjustment in computer games through real-time anxiety-based affective feedback". In: *International Journal of Human-Computer Interaction* 25.6 (2009), pp. 506–529.

[4] Olana Missura and Thomas Gärtner. "Player modeling for intelligent difficulty adjustment". In: *International Conference on Discovery Science*. Springer. 2009, pp. 197–211.

[5] Josh Sawyer. *How To Balance An RPG*. URL: `http://kotaku.com/how-to-balance-an-rpg-1625516832`.

[6] Pieter Spronck, Ida Sprinkhuizen-Kuyper, and Eric Postma. "Online adaptation of game opponent AI in simulation and in practice". In: *Proceedings of the 4th International Conference on Intelligent Games and Simulation*. 2003, pp. 93–100.

[7] Hado Van Hasselt and Marco A Wiering. "Reinforcement learning in continuous action spaces". In: *Approximate Dynamic Programming and Reinforcement Learning, 2007. ADPRL 2007. IEEE International Symposium on*. IEEE. 2007, pp. 272–279.

[8] Alexander Zook and Mark O Riedl. "A Temporal Data-Driven Player Model for Dynamic Difficulty Adjustment." In: *AIIDE*. Citeseer. 2012.