A Reinforcement Learning Model for Playing a Hack-and-Slash Game

Mohammad Arjamand Ali
Department of Computer Science
Eastern Michigan University
Ypsilanti, Michigan
marjaman@emich.edu

***Abstract*—This paper outlines the training and use of a reinforcement learning agent to play a hack-and-slash game. We built out 2 agents, one that utilized vector observations, and one that added camera observations. Different reward structures were explored. Utilizing the ML-agents library and proximal policy optimization, We trained both to play the game with the goal of surviving the longest. We ended up with an actionable model that learned small patterns and was able to survive for a notable time through the game, achieving an average Episode Reward of 2.11.**

***Index Terms*—reinforcement learning, proximal policy optimization, unity, machine learning, video games**

I. Introduction

With these recent advances in machine learning, games have been one of the major areas used to come up with new techniques. Specifically, reinforcement learning models have been exceptionally good for games. Games can be very easily modelled as a reinforcement learning environment, and thus allow us to use RL techniques as benchmarks on them. An example of this is AlphaZero [4] surpassing the skills of most chess veterans.

In this paper, we go over one instance of reinforcement learning in video games. We first review some of the literature and the techniques used for similar types of problems. We go over those that have proven to work, and those that may be relevant to out problem. We then go over the details of our game and the environment the model will interact with.

Then follows an extensive section on the different models used, along with the components that build them, including: the observations included (vector & image), model outputs, technical considerations, architectures and hyperparameters, and reward structures.

We finally evaluate the models on metrics to see how well they performed, if at all. We concluded that one of the models (one with image observations) learnt actionable patterns and performed relatively well.

II. Related Works

The following section contains detail on work related to our problem domain and some of the techniques we end up using.

A. Reinforcement Learning

The work done by [2] highlights some of the key advanced made in reinforcement learning for video games. It categorizes all of the games into different types, with basic ones like arcade games, to image-specific problems like racing and first person shooter games, and multi-agent problems like team sports games. They outline the different types of models and techniques used. They go over how Convolution neural networks are the consistently used approach for most problems due to their versatility and universal applicability. Among the methods used, Q-learning for basic environments and Actor-critic for more complex methods are most used. They also go over some of the challenges still faced in RL, one of them being the issue of sparse rewards. This occurs when rewards are spread out over longer periods/time steps, and the model has difficulty linking it to previous actions. This is one challenge we face in this paper.

B. Proximal Policy Optimization

Amongst the methods of reinforcement learning, Proximal Policy Optimization (PPO), introduced in [3], is one of the SOTA approaches. PPO builds on previous reinforcement learning methods, combining both a value based approach and a policy function. Specifically, it introduces a surrogate function that essentially 'clips' updates to the policy:

$$L^{CLIP}(\theta) = \mathbb{E}\left[\min\left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right)\right]$$

where:

- $r_t(\theta)$ represents the probability ratio of the action given the state at time t, under the new policy compared to the old policy.
- $\hat{A}_t$ is an estimator of the advantage at time t.
- ϵ is a small positive number (hyperparameter) that defines the clipping range.

PPO would clip the new update to within $1-\epsilon$ and $1+\epsilon$ times the previosu value, making sure updates are not too significant from one time step to the other. This ensures that training is smoother, and thus often results in consistent training, faster converging and better performing models.

III. The Game

Our game follows a 2d hack-and-slash style. A hack-and-slash game is essentially a game were you have a player and you attack (hack and slash) to defeat enemies around you, and progress towards some goal.

For out game, the player is given a few abilities. They can freely move in 2d space, and can do 1 of 3 different actions: attack (sword swing in last direction), dash a distance (towards movement direction), and cast (spawn a high damage strike on a random enemy). There are delays for dashing and casting, and no action can be interrupted once started.

The environment consists of a spawner that spawns enemies in increasing speed as the game progresses. There are 3 types of enemies: 2 melee versions, and a ranged enemies that throws a seeking projectile.

A minor important note: Each health item like a player's or enemy's health is managed through a health hitbox. Each damage item like a player's weapon or an enemy's projectile is managed through a attack hitbox.

IV. Methodology

The following outline the methodology used to train and guide the reinforcement learning models. The game was built in Unity, and so the Unity ML-Agents library [1] was used to build and train the models. All models were trained on the Google Colab T4 environment utilizing the GPU.

We went with 2 different models/approaches, *Vector* and *Vector + CNN*, the details of which are explained below. Both of them aim to test out different ideas for the model.

There were multiple other models used as benchmarks for hyperparamaters and testing ideas, but most were not actionable so were not included in this paper.

A. Observations

2 types of observations were collected from the environment from which models were designed, Vector and Image.

Vector Observations: these are 74 numerical observations from the environment. A detailed explanation of what they are:

- Player position, state (canMove, isAttacking, isDashing, isCasting), health, walking direction
- For the 5 closest health hitboxes: position, health, one-hot encoded hitbox type (enemy1, enemy2, enemy3, none)
- For the 5 closest attack hitboxes: position, hitbox type (enemy1, enemy2, volley, none)

Image Observations: these are captured through the main camera, which is a zoomed in camera that follows the player. It is transformed into an 84x84 px grayscale view.

B. Outputs

The models were designed to give 3 outputs: 2 continuous outputs (x & y values for movement), and 1 discrete choice of 3 labels (attack, dodge, cast).

C. Models

2 models were trained, *Vector* and *Vector + CNN*. Table I includes details on the architecture and designs of the models. The layers & hidden states indicated are for the main free forward neural network of the model.

The CNN encoder used for image observations is the 'simple' encoder type in Unity ML-Agents. All numerical observations are normalized on entry, and there are batch normalization layers after each linear free forward layer.

Fig 1 showcases a detailed version of the architecture for the Vector + CNN model. As contrast, the Vector model can be thought of as the same architecture, just without the convolution layers with the image input on the top left.

TABLE I
Model Architecture

Model	Vector	Vector + CNN
Vector Observations	✓	✓
Image Observations		✓
No. Layers	4	3
No.Hidden States	128	128

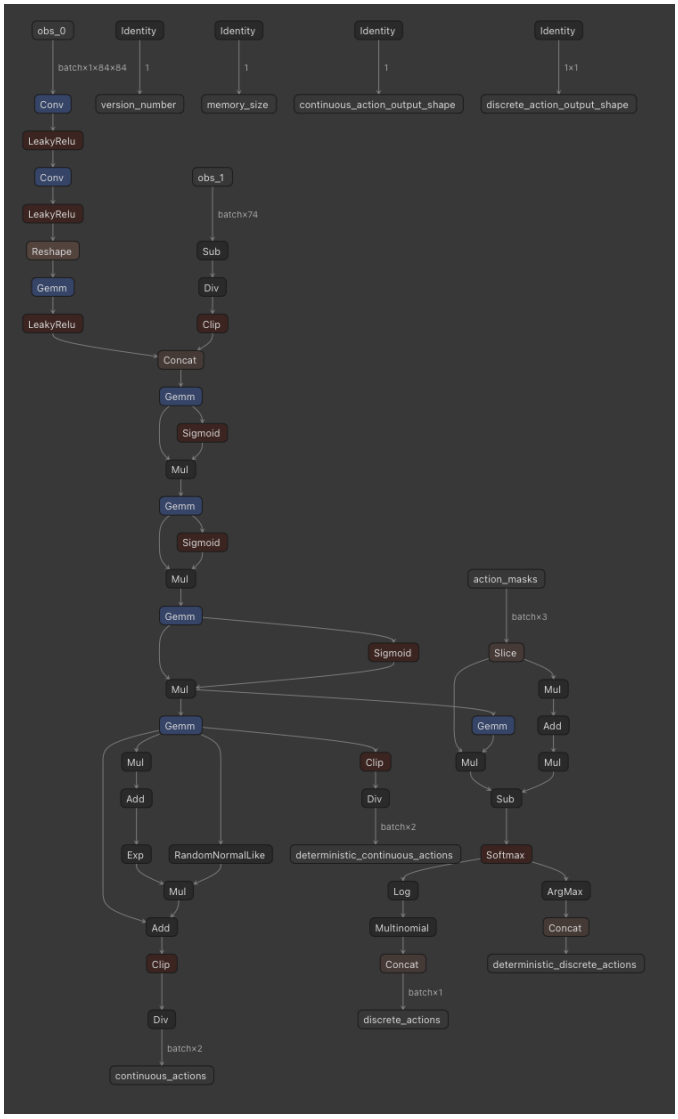

Fig. 1. Detailed model architecture for Vector + CNN model.

D. Implementation Details & Considerations

The model was paired with a Decision Requester set for 4 frames. This component defines the frequency by which the

model called for an action, and is thus incremental during training and inference. A faster request rate was deemed to be inefficient as it caused significant overhead without much improvement in performance, while slower request rates lead to bad performance in general. So a request rate of 4 was chosen.

The model was trained with multiple different instances at the same time. Namely, the models were trained with 3 instances. This allowed the model to train from different experiences at the same time, while also speeding up training time.

Image & Vector stacking was considered. But after a few initial attempts, it only seemed to slow down training due to the significant size overhead without much performance increase. So no stacking was used.

E. Rewards

Table II outlines the reward mechanisms used for the models. Different mechanism were used to test different items. Namely, the rewards for the Vector model emphasize more aggressive play (higher reward for fighting and defeating enemies) as opposed to those for the Vector + CNN model.

TABLE II
REWARD STRUCTURES

Reward Type	Vector	Vector + CNN
Time Survived (each frame)	0.0001	0.0001
Damaging an Enemy	+0.025 * damage	+0.01 * damage
Taking Damage	-0.01 * damage	-0.02 * damage
Defeating an Enemy*	+0.2	+0.06
Dying	-1	-1

Note: The rewards on enemies defeated shown in Table II are approximates. Exact values scale with different enemy types; higher reward for difficult enemies.

F. Hyperparameters

Table III outlines the other major hyperparameters used to train the models, along with a few notes on what they are below.

The buffer size indicates the window which will be used to train the model.

Beta and epsilon are the hyperparameters for the PPO algorithm. Specifically, beta is the co-efficient for the PPO loss function, and epsilon is the clipping parameter for PPO.

The Extrinsic reward is the main reward we define (surviving longer, defeating enemies, taking damage, etc). Additionally, there is a Curiosity reward. This essentially rewards the model if it takes unseen actions even when it knows better options. This encourages exploration, and this results in a better model overall.

Lambda indicates the discount factor for future rewards. The respective gamma values indicate how much future rewards should be valued.

Most of these values were standard values determined to be used after running a few initial training runs, and seeing what works.

TABLE III
HYPERPARAMETERS

Hyperparameter	Value
Batch Size	128
Buffer Size	2048
Learning Rate	2e-4
Beta	1e-3
Epsilon	0.2
Lambda	0.95
Extrinsic Reward Weight	1.0
Extrinsic Reward Gamma	0.99
Curiosity Reward Weight	0.1
Curiosity Reward Gamma	0.99
Max Steps	600000

V. EVALUATION AND RESULTS

The following sections include evaluation results and discussions regarding different aspects on how the models performed, which failed to perform at all, and general patterns the models seemed to pick up.

A. Metric Evaluation

The models were evaluated on Average Episode Reward. The Episode Reward is the cumulative reward an agent gains in one complete run of an episode (until the player dies). 10 inference episodes were run for each model, and their results shown in Fig 2. The average was calculated across these 10 examples, with the results stated in Table IV.

The individual inference recording of the episodes were also analyzed to find patterns in the model behaviour that could be used to infer model performance/actions.

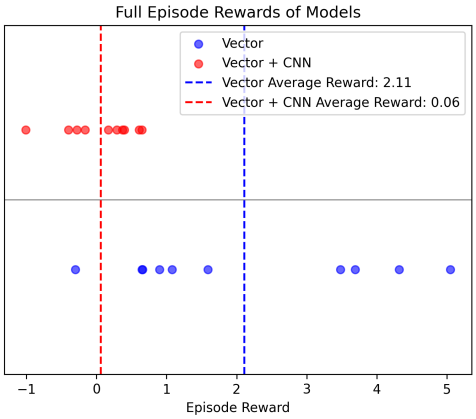

Fig. 2. Episode Rewards, for 10 episodes each, for each Model. Also indicates the average episode reward.

TABLE IV
EVALUATION RESULTS - AVERAGE EPISODE REWARD

Model	Vector	Vector + CNN
Reward	0.06	2.11

Based on Evaluation metric, the Vector + CNN model was significantly better then Vector model. For reference, an model

barely trained performs similarly to the way the Vector only model does, meaning that it dies very quickly, but sometimes manages to survive by sheer luck.

The Vector + CNN, although was not able to perform as well as expected, did learn some patterns and had a decent average Episode Reward was thus consistently better than the Vector model.

B. Loss Evaluation

One of the ways we can get an understanding of the model performance, and more so were it is lacking, is through the loss of the models.

Fig 3 and 4 showcase the respective Policy and Value losses for the Vector model.

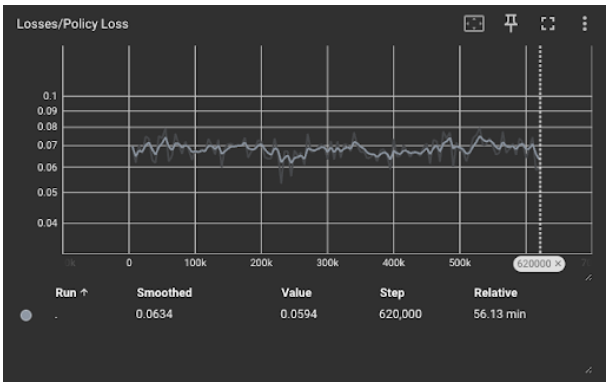

Fig. 3. Policy Loss for Vector model

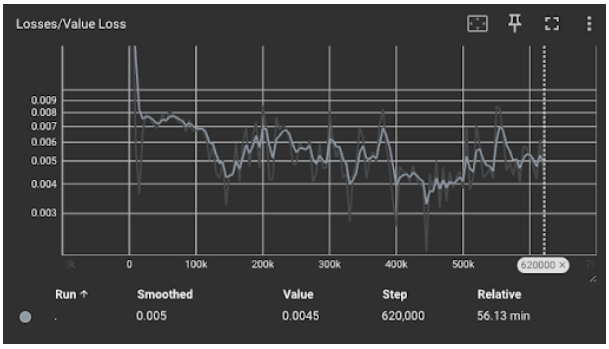

Fig. 4. Value Loss for Vector model

Fig 5 and 6 showcase the respective Policy and Value losses for the Vector + CNN model.

Through these figures, we can gain a semblance of where the model is struggling. We can see that the model value loss does go down, especially for the Vector + CNN model. We also can clearly see that the model's are learning little to none in the Policy department. This indicates that we need to adjust the hyperparameters, especially those relating to PPO to solve this issue.

C. Episode Recording Evaluation

Based on the visual analysis of the recordings, both models adapted towards the same pattern: dash, followed by 2 attacks,

Fig. 5. Policy Loss for Vector + CNN model

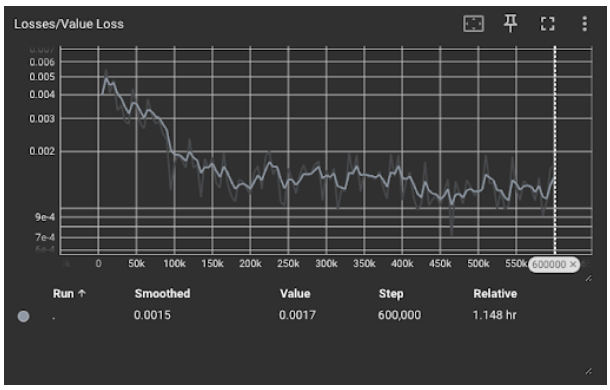

Fig. 6. Value Loss for Vector + CNN model

and then repeat. The actions of the Vector model seemed random though, as it did not change its actions when enemies spawned.

The Vector + CNN model however, seemed to show some learning. It consistently changed directions to hit enemies properly when it came near them, something the Vector model failed to do. This caused it to successfully defeat more enemies, and this survive longer. This model's dashed were also more deliberate, dashing away from danger much clearly, as opposed to the Vector model's more random dashes.

One surprising observation was that neither of the models learnt to use the cast ability. The cast is a low effort attack that causes significant damage, and was assumed to be the strategy the model would adapt to. But none of the models utilized the cast at all. This may be due to the fact that there are no enemies spawned for the first few seconds, and thus the cast is useless for that time period.

VI. Conclusion

We saw that the model struggled to learn, with only one version of the model performing relatively well. The model with image observations performed well. The models learned a consistent patter, with only the better ones learning improvements on it (facing the enemy, dashing away). We also saw that policy loss was constant throughout, which may be the cause of most issues we encounter. This, paired with the issues

of the sparse reward nature seem to be the biggest causes of lack of performance of the models.

This paper contributes to the field of reinforcement learning, especially in video games, by showcasing the challenges and exploring how different observations affect the performance of models. It also adds by providing a decent performing model for the hack and slash game problem.

For future work, a curriculum learning approach may be the best solution. Although the survival model was designed to start easy and ramp up with time, it unfortunately wasn't enough. An environment that first emphasizes basic actions like uniform movement, followed by fighting one enemy, then 2, and so on, may prove to be excellent for a problem like this.

Note: The code for the game, and the model training can be found at: https://github.com/Mohammad4844/rl-model-for-hack-and-slash-game/

REFERENCES

[1] Arthur Juliani, Vincent-Pierre Berges, Ervin Teng, Andrew Cohen, Jonathan Harper, Chris Elion, Chris Goy, Yuan Gao, Hunter Henry, Marwan Mattar, and Danny Lange. Unity: A general platform for intelligent agents. *arXiv preprint arXiv:1809.02627*, 2020.

[2] Niels Justesen, Philip Bontrager, Julian Togelius, and Sebastian Risi. Deep learning for video game playing. *CoRR*, abs/1708.07902, 2017.

[3] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[4] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815, 2017.