RL-Augmented Action Spaces in MsPacman

Timothy Kostolansky
timkosto@mit.edu

Julian Yocum
juliany@mit.edu

Abstract

Since the inception of video games, artificial intelligence (AI) has played a critical role. As far back as the 1950s, AIs were used as player adversaries. Then, dynamic difficulty adjustment made possible hyper-personalized player experiences. While AI has had a long history in video games, few games have innovated on these paradigms. We propose exploring novel approaches for the application of RL to game mechanics. In particular, we describe an RL-augmented player action space mechanic in Atari 2600 Ms Pacman for a novel and engaging gameplay experience.

1 Introduction

We develop a pygame application based on Atari 2600 Ms Pacman to demonstrate novel gameplay dynamics made possible uniquely through deep learning. We augment the action space of the player using RL, creating a novel and engaging experience for the player. Specifically, we create four distinct policies, each trained with a different objective/reward function, and allow the player to dynamically select between these policies during gameplay. The player is able to upgrade and learn how to use each of the "skills" that each policy offers, providing a fun and novel way to play a classic game.

To test our hypothesis we ran two experiments: 1) Training a variety of RL policies (with different objectives/reward functions) on the MsPacman Atari video game; 2) Augmenting the video game with an RL-augmented player action space using the trained policies.

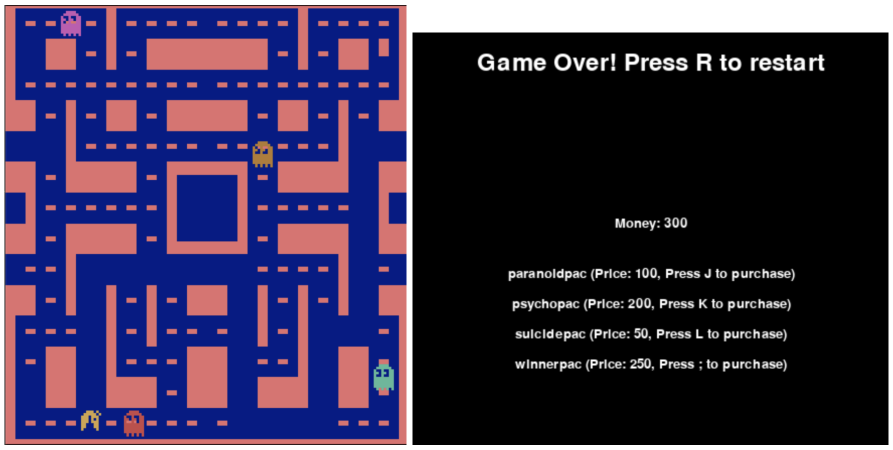
A successful demonstration of novel gameplay mechanics would create a new and challenging twist to an already-known setting in an Atari video game. This demonstrates how deep learning can be used to enhance the video game experience and open the door to other novel applications of deep learning to gameplay mechanics.

2 Related Work

We take inspiration from the creator of *Into the Breach* [1], a turn-based strategy game that successfully employed an AI which randomizes the game setting in ways that players found challenging and rewarding. We take note of this creator's decision to create an AI that is as simple as possible in order to achieve the desired effects.

[2] approaches the problem of creating a dynamic environment to train an agent by using Unsupervised Environment Design (UED). This method approaches environment generation by training an agent (the protagonist) within a learning environment (the adversary) alongside another learning agent (the antagonist). The adversary's goal is to generate environments that maximize the antagonist's reward while minimizing the protagonist's reward.

The primary method used in our game is Asynchronous Proximal Policy Optimization (APPO) [4], a policy-gradient actor-critic method to learn a policy for the augmented player action space.



Game Over! Press R to restart

Money: 300

paranoidpac (Price: 100, Press J to purchase)

psychopac (Price: 200, Press K to purchase)

suicidepac (Price: 50, Press L to purchase)

winnerpac (Price: 250, Press ; to purchase)

Figure 1: Game screenshots from pygame application. Left: Atari 2600 MsPacman during gameplay. Right: Management screen for the display of money earned from accumulated score, and RL policies for purchase. Players can build up their money and upgrades over multiple rounds of gameplay.

3 Methods

We develop a pygame application based on Atari 2600 MsPacman with an RL-augmented player action space. The application allows users to play Ms Pacman with AI assistance, and upgrade their policies using their accumulated money on a management screen displayed in Figure 1. We create policies that a human player can choose between dynamically while playing the game. Each policy is trained on a distinct reward function in order that each policy qualitatively "behaves" differently from the others.

3.1 State and Action Space

The state space of the human player and each policy is the state space of the chosen Atari game. For MsPacman, this is the RGB image representing the locations of Ms. Pacman, the ghosts, the food pellets, and the maze.

The action space of each of the policies is the standard action space of the MsPacman game: moving up, down, left, and right. On the other hand, the action space of the human player will be the standard action space augmented with a set of trained policies for a specific Atari game.

$$A_{\pi_i} = \{\texttt{up}, \texttt{down}, \texttt{left}, \texttt{right}\} \tag{1}$$

We created a set of trained policies for a human to choose between: a policy trained to eat lots of food pellets, a policy trained to do towards the ghosts, a policy trained to eat as many ghosts as possible, and a policy with the original MsPacman rewards. The human player will then be able to dynamically switch between these four policies in such a way to play MsPacman with the original objective.

$$A_{\text{human}} = A_{\pi_i} \cap \{\pi_{winner}, \pi_{psycho}, \pi_{paranoid}, \pi_{anti}\} \tag{2}$$

3.2 Objective Function

The objective function for each of the trained policies will be distinct in order that the policy captures a distinct behavior. The novelty of this project will be in the designing of objective functions that encode unique behaviors in the policies. We developed a four-policy MsPacman human action space (refer to 3.1) in our work. We describe each of the four policies and provide the reward functions used in Figure 2.

1. **Winner-Pac** (π_{winner}) is trained on the vanilla environment reward function, which is the per-step score difference capped to 1, i.e. eating a pellet, or a ghost or big pellet all receive reward 1 and all other actions reward 0.
2. **Psycho-Pac** (π_{psycho}) is trained with eat as many ghosts as possible. The agent receives a reward of 1 on steps where a ghost is eaten and 0 otherwise.

3. **Paranoid-Pac** ($\pi_{paranoid}$) is trained to stay alive for as long as possible. The agent receives a reward of 1 at every step, except -1 if `terminated` is true.
4. **Anti-Pac** (π_{anti}) is trained to minimize the time spent alive. The reward received is the negation of Survivor-Pac's reward.

After successful training, these four policies could provide a rich and novel gameplay mechanic for the human player to use.

```
class CustomRewardEnv(gym.Wrapper):
    def __init__(self, env_name):
        super(CustomRewardEnv, self).__init__(gym.make(env_name))

    def step(self, action):
        obs, reward, terminated, truncated, info = self.env.step(actions)

        if policy == "winnerpac":
            reward = np.clip(reward, -1, 1)
        elif policy == "psychopac":
            reward = np.where(reward >= 100, 1, 0)
        elif policy == "paranoidpac":
            reward = (1.0 - terminated) - terminated
        elif policy == "antipac":
            reward = - ((1.0 - terminated) - terminated)
```

Figure 2: A custom gym wrapper modifies the reward function for four different policies with qualitatively different behavior: a policy with the original MsPacman rewards (π_{winner}), policy trained to eat as many ghosts as possible (π_{psycho}), a policy trained to stay alive for as long as possible ($\pi_{paranoid}$), and a policy trained to die as quickly as possible (π_{anti}).

3.3 Training

We trained our policies with Asynchronous Proximal Policy Optimization (APPO) [4] using Sample Factory[1] with default hyperparameters for two billion steps. Unlike vanilla PPO, APPO allows collection of rollouts and training of models at the same time. This results in quicker training, as training is able to be completed while workers collect rollouts. We modified the Sample Factory gym environment with two wrappers. The first is a sticky action wrapper for introducing stochasticity into the environment following [3]. The second is a reward wrapper to modify the reward function for the four polices described. We provide the full reward wrapper as displayed in Figure 2.

4 Results

Each of the four policies learns to act successfully by itself, but they also all suffer from being quite brittle along human trajectories. 3 depicts reward and lives curves for each of the policies and we describe each of their behaviors.

Psycho-Pac learned to eat the big pellets (a prerequisite to killing ghosts) and kill ghosts. This is reflected in the large jumps that can be seen in the reward curve. The policy highly overfit to killing as many ghosts as possible in the first level, but it failed to "generalize" and learn to regenerate the big pellets (necessary for its goal of killing ghosts) by eating all the small pellets (necessary to start a new level in which there are more big pellets).

Paranoid-Pac learned to retain all three of its lives while accruing no original in-game rewards. This resulted in staying alive for a long period of time without moving on to subsequent levels. When placed along human trajectories, it failed to generalize and stay alive in the novel settings that it was placed in.

[1]`https://www.samplefactory.dev/`

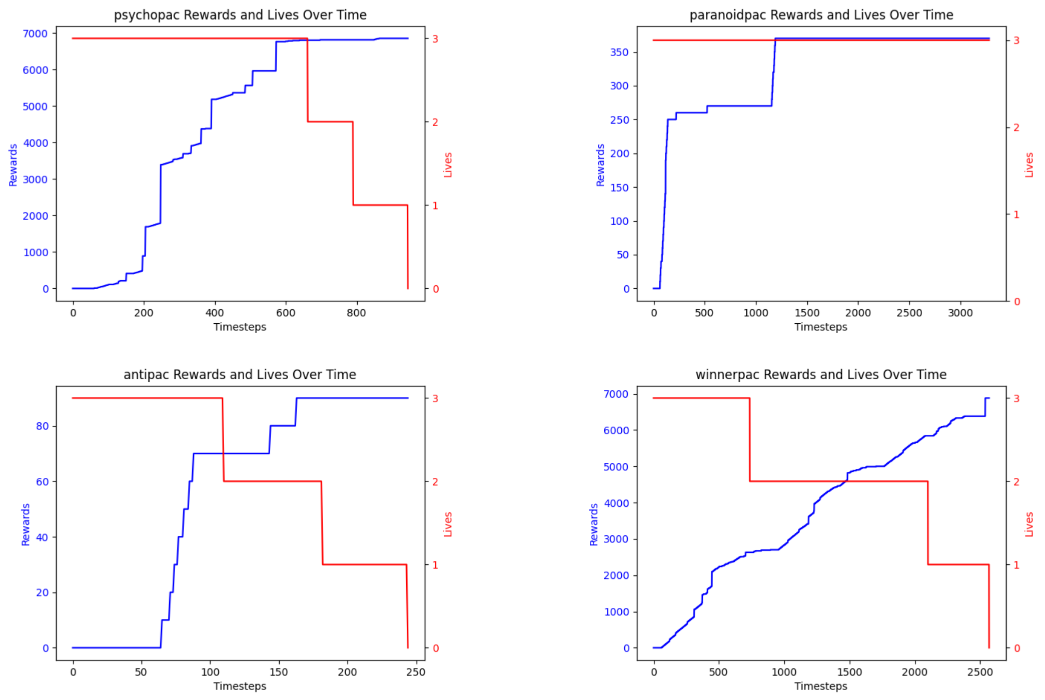


Figure 3: In-episode reward curves for four trained policies. Each policy is trained on a different reward function, and each policy's respective reward/lives curves reflect its desired behavior encoded by its reward function. Despite successful behavior tuning using the reward functions (as seen in this figure), dynamically selecting between the policies (by a human player) resulted in failures to behave as desired when out of the trained distributions. Even small deviations from the training trajectories resulted in failures to behave as desired.

Anti-Pac learned to make a beeline for the ghosts at the start of each life it had. It spends relatively much less time alive and is robust to some stochasticity in the ghosts' movements at the start of each new life, as the policy learns to wait around where the ghosts are spawned in the maze.

Winner-Pac learned to accrue rewards consistently throughout its play, as shown by the approximately linear reward curve. Its performance was not optimal with these original rewards, though, as it eventually loses all three lives in the third level.

The following is a video depicting the Psycho-Pac policy: `https://drive.google.com/file/d/1J6INtewMKj0gSSmNRTs283t_DyGvKEie/view?usp=drive_link`.

The following is a video depicting the Paranoid-Pac policy: `https://drive.google.com/file/d/1OsXo3yQoEa0hFBMdvW_h9LTdyCd0AGSW/view?usp=drive_link`.

The following is the link to a video depicting the brittle nature of initializing the policies along human policies: `https://drive.google.com/file/d/1WRhkmJ3ko_ThNH--cokq-xd3xXvyUeEI/view?usp=drive_link`. Note the policies that are selected by the human player are shown at the bottom of the screen.

5 Discussion

We observed that despite the widespread use of Atari gymnasium environments in RL benchmarks, training a human-level policy was very difficult using standard libraries such as Stable-Baselines3. Stable-Baseline's vanilla DQN and PPO methods plateaued under an average 3000 score and failed to reliably surpass level 1. On the other hand, APPO through the Sample Factory library attained closer to human-level outcomes.

While policies which were continuously from environment reset were found to perform near human-level, polices which were instead initialized along human trajectories showed drastically diminished performance. As the intention of our RL-augmented action space was for a player to rapidly select between different policies, the lack of robustness to off-distribution trajectories meant that human-guided selection of policies had worse performance than just running the best performing policy. This lack of robustness of policies is a well-known failure mode for deterministic environments. Following [3], we attempted to mitigate this through the introduction of stochasticity to the environment via "sticky actions", a wrapper which for each time step repeats the last selected action with probability 0.25. However, this intervention was found to only have mild improvements to policy robustness.

6 Contributions

Julian Yocum applied modifications to the Sample-factory APPO library to include reward wrappers and sticky action wrappers and trained the four policies using the OpenMind cluster.

Timothy Kostolansky designed the reward functions, developed the pygame application, and graphed the reward curves.

References

[1] Matthew Davis. Into the breach ai. `https://docs.google.com/document/d/1OPRNzAVQNgPtP35HzhV2ZzEqE1eCdXbWA7lEZXDdMHQ/mobilebasic`.

[2] Michael Dennis, Natasha Jaques, Eugene Vinitsky, Alexandre Bayen, Stuart Russell, Andrew Critch, and Sergey Levine. Emergent complexity and zero-shot transfer via unsupervised environment design, 2021.

[3] Marlos C. Machado, Marc G. Bellemare, Erik Talvitie, Joel Veness, Matthew J. Hausknecht, and Michael Bowling. Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents. *CoRR*, abs/1709.06009, 2017.

[4] Aleksei Petrenko, Zhehui Huang, Tushar Kumar, Gaurav S. Sukhatme, and Vladlen Koltun. Sample factory: Egocentric 3d control from pixels at 100000 FPS with asynchronous reinforcement learning. *CoRR*, abs/2006.11751, 2020.