Extended Abstract

Motivation Super Smash Bros. Melee presents a structured, fast-paced environment with hybrid discrete–continuous controls, and opportunities for a variety of different rewards. Prior RL work has largely focused on simple reward domains, or limited action/state spaces lacking access to projectile and fine-grained animation data. We evaluate the spectrum of offline and online RL paradigms including: Behavioral Cloning (BC), Implicit Q-Learning (IQL), Proximal Policy Optimization (PPO), Deep Q-Learning via DQN, and Double DQN in Melee. We also explore the effects of large vs. small state representations, continuous vs. discretized controls, and diverse reward-shaping schemes for the game. We show the efficacy of each architecture on limited compute and data. We achieve this by leveraging LibMelee's full game-state API (including projectile info) sample and have our agent play the game against CPUs.

Method We explored multiple offline and online algorithms, along with various reward functions in an attempt to teach the agent to play smash. We only train on 2 characters: Fox and Falco because they both have ranged attacks. The offline approaches we tried includes BC and IQL. For Online learning, we implemented PPO, DQN, and Double DQN. In addition, we experimented with 3 different rewards to improve learning. We use large (all data) and small (x, y pos) state spaces, large and small, discrete and continuous action spaces. The offline agents train on downloaded files and the online agents train directly off of the emulator.

Implementation We ran all experiments on the same Dolphin+Slippi emulator setup instrumented by LibMelee, which gives us frame-level access (60 Hz) to both the game state and control inputs. To study how state and action complexity affects learning, we defined two state representations a large 70-dim vector (full character, projectile, and stage features) and a small 4-dim vector (just each fighter's x, y positions). The action sets include: Small discrete (left/neutral/right × jump/no-jump), Small continuous $[0, 1]^2$, Large discrete all 17 controller inputs quantized to 0,1 and -1,0,1, Large continuous all 17 floats in 0,1 and [0,1]. We also compared three reward schemes of increasing sparsity and difficulty: Minimize Distance: dense feedback of distance from CPU. Stay Alive: penalties for falling offstage (–100) and taking damage. Win the Game: sparse combination of stock changes (deaths/kills) and damage taken/dealt We evaluated on the emulator against Level 9 CPUs and tracked how long they survived, the their average distance to the agent, time of first death, etc depending on the corresponding reward function.

Results Value-based agents (DQN and Double DQN) rapidly master the dense Min Dist task achieving optimal proximity within a 20 epochs but fail under the sparse Stay Alive and Win the Game objectives, in the small state/action setting. By contrast, PPO while slower to converge on Min Dist demonstrates markedly greater robustness to reward sparsity, delayed penalties, and large state/action spaces: it outlives Double DQN by 600 frames under Stay Alive, by 2000 frames under Win the Game, deals 0.165 more normalized damage, and secures 1 extra stock per match. Offline methods (BC and IQL) efficiently imitate CPU play and inject reward guidance via IQL's expectile weighted critics, but alone they fall short of competitive Level 9 performance without subsequent online fine-tuning.

Discussion Our results show that no single algorithm suffices across all reward densities and task complexities: DQN shine on dense, immediate rewards, and falter on sparse, delayed objectives and large state/action spaces. On the other hand, PPO's on-policy gradient updates, despite slower early convergence, prove far more robust to the delayed reward challenges. Moreover, offline pretraining via BC and IQL accelerates initial learning but struggles to close the gap to Level-9 CPUs without online fine-tuning.

Conclusion Our findings show the importance of matching algorithm choice to reward structure, enriching state representations (e.g., projectile data), and employing careful reward shaping. Looking ahead, integrating offline imitation with online self-play, gradually increasing reward sparsity, and human-in-the-loop feedback may be key to improve our agent enough to beat level 9 CPUs consistantly and even human competition. Future works should work on using more compute, RLHF, multi-agent self play, and use IQL as a baseline and using IQL as a prior to fine tune via PPO online.

Analysis of RL Architectures for Delayed Rewards in Super Smash Brothers Melee

Danica Xiong
Department of Computer Science
Stanford University
daxiong@stanford.edu

Tony Xia
Department of Computer Science
Stanford University
tonyx717@stanford.edu

Abstract

We present a comprehensive study of Reinforcement Learning frameworks for Super Smash Bros. Melee, a fast-paced platform fighter video game with hybrid discrete–continuous controls and sparse, delayed win/lose signals, and a very large state space. Leveraging the Dolphin+Slippi emulator and LibMelee API for frame-level (60 Hz) access to character, projectile, and stage data, we compare five training BC, IQL, DQN, Double DQN, and PPO. We use compare large (70-dim) versus small (4-dim) state representations and continuous versus discretized action spaces on the gamecube controller. We evaluate three reward schemes of increasing sparsity (distance minimization, survival penalties, and win-game objective). Our results against Level 9 CPUs are measured by damage dealt, stocks retained, training rate, among other metrics. Our results show value-based agents (DQN/Double DQN) rapidly master dense proximity rewards with small state and action spaces. PPO, though slower and less efficient on dense tasks, exhibits greater robustness to delayed and sparse rewards, surviving hundreds to thousands of frames longer, dealing more damage, and securing additional stocks. We also show that offline methods (BC/IQL) provide useful initialization but require online fine-tuning. These findings highlight the need to align algorithmic choices with reward, state, and action structure, and point to hybrid pipelines—combining imitation, curriculum learning, and self play as promising paths toward human-competitive Smash agents.

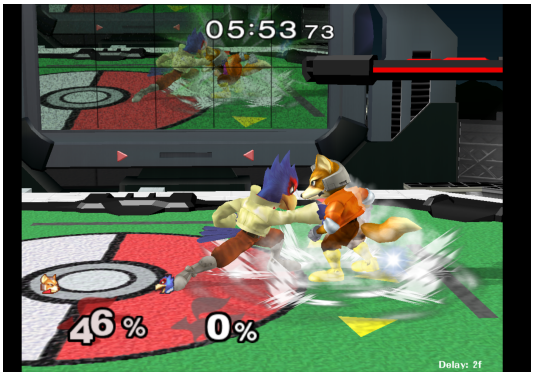

Figure 1: Fox Agent vs Falco CPU, a close game!

1 Introduction

Super Smash Brothers Melee is a fast-paced, fully observable platform-fighting game in which each player uses a combination of binary button presses (e.g., A, B, L, R, D-pad directions) and analog joystick inputs to control a character's movement, attacks, and defensive maneuvers. Matches are won by knocking the opponent off the stage (depleting their stock count) 4 times. Damage percentage and remaining lives ("stocks") serving as key metrics of survivability.

Super Smash Bros. presents a challenging reinforcement learning problem characterized by high-dimensionality: large states spaces, various continuous and discrete action spaces, and a single goal to beat your opponent. In this work, we develop an RL agent capable of outperforming Nintendo's built-in CPU opponents (Levels 1–9) and ultimately human players, using a variety of RL algorithms. Specifically, we compare five approaches: behavioral cloning (BC), Implicit Q-Learning (IQL), Proximal Policy Optimization (PPO), Deep Q-Networks (DQN), and Double DQN under a variety of state and action space configurations, reward definitions, and data-efficiency trade-offs.

At each timestep, the agent observes a vector of game features—positional data, velocities, animation states, projectile information, damage percentages, character data, stage data, and stocks. We select actions from a high-dimensional action space that can be represented either continuously: joystick between [0,1] or discretely -1,0,1, and regular buttons (pressed, unpressed) on the Gamecube controller.

We explore the impact of (a) using "large" versus "small" state representations, and (b) discretized versus continuous action encodings (c) offline vs online architectures, and explore (d) incorporating historical window states. To address the challenge of sparse and delayed game rewards, we introduce a tailored reward-shaping mechanism for Melee's dynamics. Through experiments and analysis, we compare each architecture's sample efficiency, stability, and final performance, and compare them among various metrics, offering insights and practical improvements for future RL research in complex, real-time fighting games.

Our simplified rewards include: Minimize Distance, which rewards minimizing the Euclidean distance to the opponent, and Stay Alive, which rewards maximizing remaining stocks and minimizing damage taken. We also have a cumulative reward function that uses stocks and percentage as metrics to judge how close the agent is to winning the game. Ultimately, training the large architecture with all actions, state spaces, and culminative reward was our goal.

2 Related Work

1. **Beating the World's Best at Super Smash Bros. with Deep Reinforcement Learning Firoiu et al. (2017)**
   This paper demonstrates that deep reinforcement learning methods could be applied to Smash Bros. Melee. The authors trained agents on Dolphin Emulator. Their method uses the emulator's game state including position, velocity, and frame. They used Q-learning and policy-gradient approach which we would also like to experiment with and potentially improve. Their agent was able to learn complex, partially observable, multi-player environment and surpassed professional players. Our goal is to build a similar pipeline and test with other reward functions. Also we want to apply imitation learning to try to speed up their process. A main challenge will be to parallelize all of our code, as they were able to parallelize it and run 50 emulators in parallel to speed up training time.
2. **Nintendo Super Smash Bros. Melee: An "Untouchable" Agent Parr et al. (2017)**
   This paper shows the nuances that comes with tuning a game with a large action space and much larger state space like Smash Bros. In this paper, the authors tune the reward function such that their agent prioritizes it's own health more than everything else. The authors also use Dolphin, but they have a survival-only reward. They use a terminal flag such that as soon as the agent is it, the flag is set and the match is reset. This is essentially the same as having negative infinity reward. They use double DQN and "Dueling DQN" which we could potentially try as well.
3. **Learning to Play Super Smash Bros. Melee with Delayed Actions Sharma (2017)**
   This paper is very cool. The authors point out that a lot of the agents have unfair advantages because they get the frame and state output immediately (unlike humans who have slow

reaction time). The authors delay the neural network and uses an RNN to buffer past observations. This way, their neural network is able to perform well with significant input lag. We don't plan on implementing this, but it's a cool read and if we have time, we can consider it.

4. **Transformers in Reinforcement Learning: A Survey Agarwal et al. (2023)**
This paper is a survey on transformer architecture used in RL. Using transformers helps with various RL problems, but the core one that relates to our paper is that it helps tackle long range credit assignment. They talk about real-world adoption of transformers in RL and the hurdles surrounding compute, data efficiency, and robustness.

3 Methods

We explored multiple offline and online algorithms, along with various reward functions in an attempt to teach the agent to play smash. We only train on 2 characters: Fox and Falco because they both have ranged attacks for projectile data. The offline approaches we tried includes behavior cloning (BC) and implicit Q learning (IQL). For Online learning, we implemented proximal policy optimization (PPO), deep Q learning (DQN), and double deep Q learning (double-DQN). In addition, we experimented with reward shaping to improve learning. We use large and small state spaces, large and small, discrete and continuous action spaces, and different rewards. Its important to note, we only use large reward spaces with large state spaces and large action spaces. This is because it doesn't make sense to measure "total damage dealt" if the agent doesn't have access to attack abilities or damage data, etc. The offline agents train on downloaded files and the online agents train directly off of the emulator.

3.1 Offline Learning

Behavior Cloning: As a baseline, we implemented the vanilla behavior cloning algorithm. Given a dataset $D = \{s_1, a_1, \cdots s_m, a_m\}_{i \in n}$ with n trajectories taken by an expert player, behavior cloning trains a generative model π_θ to minimize the average negative log likelihood of the trajectories:

$$\min_\theta -\mathbb{E}_{(s,a)\in D}[\log(\pi_\theta(a|s))]$$

The loss encourages the model to high likelihood to the actions that are similar to the actions taken by the expert data. The architecture we chose was a feedforward network with two heads, predicting the mean and standard deviation of the underlying distribution respectively.

Implicit Q Learning: An improvement upon behavior cloning is implicit Q learning. IQL allowed us to incorporate a reward function into basic behavior cloning. IQL trains Q_θ to predict $Q(s, a)$, the expected sum of future rewards from taking action a at state s, and V_ϕ to predict $V(s)$, the expected sum of future rewards starting at state s. The Q net and V net were optimized using the following losses

$$L_V = \mathbb{E}_{(s,a)\in D}[L_2^\zeta(Q_\theta(s, a) - V_\phi(s))]$$
$$L_Q = \mathbb{E}_{(s,a,s')\in D}[(r(s, a) + \gamma V_\phi(s') - Q_\theta(s, a))^2]$$

where $L_2^\zeta(\mu) = |\zeta - \mathbf{1}\{\mu \geq 0\}|\mu^2$ is the expectile loss. When the ζ is the hyperparameter ζ is set to a high value (between 0.5 and 1), the Vnet is encouraged to learn the value of taking higher percentile actions.

Finally, the policy is extracted by minimizing the negative log likelihood weighted by the advantages estimated using the Vnet and Qnet trained earlier:

$$L_\pi = -\mathbb{E}_{(s,a)\in D}[\log \pi(a|s) \exp(\frac{1}{\lambda}(Q_\theta(s, a) - V_\phi(s)))]$$

In our implementation, the policy net we used in IQL has the same architecture as the one used in BC.

3.2 Online Learning

Proximal Policy Optimization: PPO is an off-policy actor critic method that uses a surrogate objective to enable multiple gradient updates from each policy rollout. In PPO, the data from each

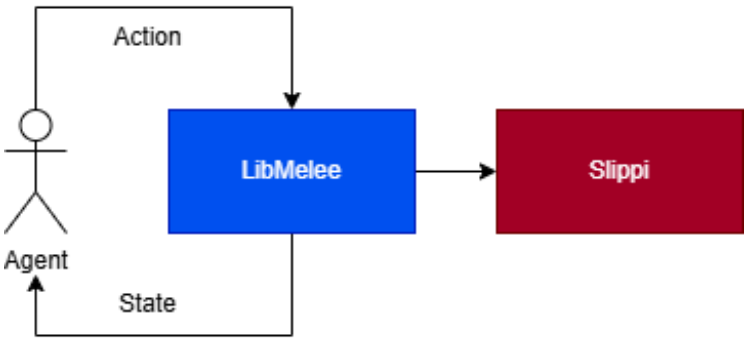

Figure 2: The simulation environment

rollout is first used to train a critic V_ϕ to learn the V function. Then, the actor π_θ is optimized using the following surrogate objective:

$$L(\theta) = \sum_t \min\left(\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)} A(s_t, a_t), \text{clip}(\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)}, 1-\epsilon, 1+\epsilon) A(s_t, a_t)\right)$$

With this surrogate objective, PPO allows for multiple gradient updates in a single rollout, significantly improving data efficiency over on-policy actor critic method.

Deep Q-Learning: DQN is an off-policy method that approximates the optimal Q-Value, which is the action-value function $Q*(s,a)$with a neural network $Q_\theta(s,a)$. Transitions (s_t, a_t, r_t, s_{t+1}) are stored in our replay buffer and sampled for training. We use epsilon greedy to perform exploration and exploitation. The network is trained by minimizing the temporal difference loss:

$$L(\theta) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\left[\left(r + \gamma \max_{a'} Q_{\theta^-}(s', a') - Q_\theta(s,a)\right)^2\right]$$

Double Deep Q-Learning: DQN is prone to overestimation bias due to it taking the maximum Q value for an action. DQN separates the action selection from the action evaluation with 2 DQNs. It uses the online network Q_θ to select the best action and uses the target network Q_ϕ to evaluate the value:

$$L(\theta) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\left[\left(r + \gamma\, Q_\phi\big(s', \arg\max_{a'} Q_\theta(s', a')\big) \;-\; Q_\theta(s,a)\right)^2\right].$$

4 Experimental Setup

4.1 Environment Setup

Super Smash Bros Melee was made for Nintendo GameCube console. In order to emulate the game on PCs, we used Slippi framework with LibMelee python library. Slippi enables us to run the game, and LibMelee provides a python interface for us to interact with the emulator. The library provides us with the states from the emulator, takes in commands from our python code, and executes those actions on the emulator. Fig. 2 shows a simple flowchart of how the environment works.

All the experiments described below were run on the same environment, albeit with different state and action spaces. Over the course of the project, we experimented with two different state spaces and four different action spaces

4.2 State Spaces

- **Large**: This state space includes 17 features for each character, 5 features for up to 5 projectiles, and 1 feature for the stage. Each state in the large state space is a 70 dimensional vector.
- **Small**: This state space includes only the x and y positions of each character. Each state is a 4 dimensional vector.

4.3 Action Spaces

On the game controller, there are 17 inputs: [A, B, D_DOWN, D_LEFT, D_RIGHT, D_UP, L, R, X, Y, Z, main_x, main_y, c_x, c_y, l_shldr, r_shldr]. The first 11 inputs are discrete buttons and the last 6 are continuous actions on a joystick:

- **Small discrete**: Each action is a 2 dimensional vector taken its value from {[0,0],[0,1],[-1,0],[1,0]}. The first entry is the main joystick (left/right) discretized: -1 means left, 1 means right, and 0 means nothing. The 2nd entry is jump: 0 means nothing, and 1 means jump.
- **Small continuous**: Each action $a \in [0, 1]^2$. The first dimension specifies the horizontal movement of the character. The second dimension determines whether to press the jump button.
- **Large discrete**: Each action is a 17 dimensional vector taken from a set of 24 distinct actions. The 24 actions were manually selected to be the 24 most used actions according to a human player. The button inputs were represented with {0, 1}, and the analog inputs were discretized to {-1, 0, 1}.
- **Large continuous**: Each action $a \in [0, 1]^{17}$. This vector contains the value for all the buttons on a standard gamecube controller.

4.4 Reward Functions

- **Minimize Distance**: The objective is to minimize the distance between the player and the CPU. The reward function has continuous feedback and is simply the division of the distance function. Minimize Distance is the easiest reward.

$$r_t = \frac{1}{\sqrt{(x_{1,t} - x_{2,t})^2 + (y_{1,t} - y_{2,t})^2} + 1}$$

- **Stay Alive**: The function gives a large negative reward for dying, and a smaller negative reward for taking damage. This function has sparse rewards because the reward is only applied when it takes damage/dies, and is 0 the rest of the time. Stay alive is harder than Min Dist.

$$r_t = \begin{cases} -100, & \text{if } \text{off\_stage}_{1,t} = \text{true}, \\ -\big(\text{percent}_{1,t} - \text{percent}_{1,t-1}\big), & \text{if } \text{percent}_{1,t} > \text{percent}_{1,t-1}, \\ 0, & \text{otherwise}. \end{cases}$$

- **Win the Game**: The objective of this reward is to help the agent win the game by (1) killing the enemy, (2) not dying, (3) damaging the enemy, (4) not taking damage. All of these components occur sometimes so this reward is very sparse. Win the Game is the hardest reward. Let S_1 be player 1's Stock, S_2 be player 2's Stock, p_1 be player 1's percentage, p_1 be player 2's percentage

$$\Delta S_1 = S_{1,t} - S_{1,t-1}, \quad \Delta S_2 = S_{2,t} - S_{2,t-1}, \tag{1}$$

$$\Delta p_1 = p_{1,t} - p_{1,t-1}, \quad \Delta p_2 = p_{2,t} - p_{2,t-1}, \tag{2}$$

$$r_{\text{stock}} = \begin{cases} 3\,\Delta S_1, & \Delta S_1 < 0, \\ 0, & \text{otherwise}, \end{cases} + \begin{cases} -3\,\Delta S_2, & \Delta S_2 < 0, \\ 0, & \text{otherwise}, \end{cases} \tag{3}$$

$$r_{\text{hp}} = \begin{cases} -0.001\,\Delta p_1, & \Delta S_1 \geq 0 \text{ and } \Delta p_1 > 0, \\ 0, & \text{otherwise}, \end{cases} + \begin{cases} 0.001\,\Delta p_2, & \Delta S_2 \geq 0 \text{ and } \Delta p_2 > 0, \\ 0, & \text{otherwise}, \end{cases} \tag{4}$$

$$r_t = r_{\text{stock}} + r_{\text{hp}}. \tag{5}$$

4.5 Offline Learning

Data Collection: We collected 60 Slippi replay files from an online source [1]. These replay files contain the state-action pairs from games played by human players. On average, each file contains 10k state-action pairs. 10 replay files were randomly chosen to be the validation set.

Behavior Cloning: We used the large state space and large continuous action space to train the agent. The agent is a feedforward network with an input layer, 2 hidden layers with hidden size 128 and relu activation, and two output heads, representing the mean and log standard deviation of the predicted distribution respectively. Clamping and gradient clipping were used to ensure training stability.

Implicit Q Learning: IQL training used the same state and action spaces as behavior cloning. The policy network has the same structure as the BC agent. The Vnet and the Qnet are both simple 3 layer feedforward network with hidden dimension 128 and relu activation. Min. Dist. reward was used for the training to encourage the agent to move towards the opponent.

Both models were trained on an RTX 3070 for 100 epochs. The training for BC took about 10 minutes, and the training for IQL took about 20 minutes.

4.6 Online Learning

Proximal Policy Optimization: We ran PPO with two different setups:

- small state space and small-and-continuous action space
- large state space and large-and continuous action space

In both setups, we trained PPO with $\gamma = 0.99, \epsilon = 0.2$, 10 updates per rollout, and 1800 frames per rollout. The PPO network consists of 2 hidden layers, each with hidden dimension 128 and ReLU activation function, and two output heads that predict the mean and log-standard-deviation of the action distributions, represented by Gaussians. We trained both setups with the minimize-distance reward and the stay-alive reward over 100 epochs.

Deep Q-Learning: We trained the DQN agent on the large state space and the small state space. Due to the nature of DQN, we can only use discretized action sets for DQN, so we used the large and small discretized action sets. The Q-network is a feed-forward MLP with an input layer matching the state dimension, two hidden layers of 128 units. During learning, we use epsilon-greedy exploration.

Double Deep Q-Learning: Similar to DQN, we tested it on the large and small state spaces, as well as only used the large and small discretized action sets. Besides the addition of the target Q network, everything else, including the replay buffer, episolon-greedy, etc remain the same as the regular DQN. We also tried to run Double DQN with historical states for the final large reward function.

Each online algorithm was trained for 2-5 hours depending on how well the loss was doing on a CPU. Due to the nature of online training and the emulator, no parallelization could be employed without more computers/CPUs.

5 Results

5.1 Quantitative Evaluation

Offline Learning

We trained BC and IQL over 200 epochs, and the training graphs are shown in Fig. 8. The first two graphs show that the trainig converged nicely after 70k batches (with around 350 batches per epoch). We then compared the per episode total minimize-distance reward between the BC and the IQL agent. As seen from the graph, IQL was not able to improve over BC. We believe that this was likely due to the fact that in a human player's strategy, minimizing the distance between the characters is not always desirable. As a result, there might not be an abundance of data where the player actively approaches the opponent.

Online learning

[1] http://slippilab.com/ is an online platform where players upload and share replays.

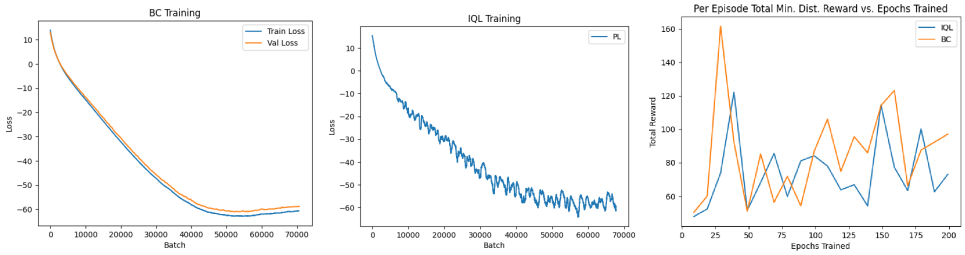

Figure 3: Offline Training

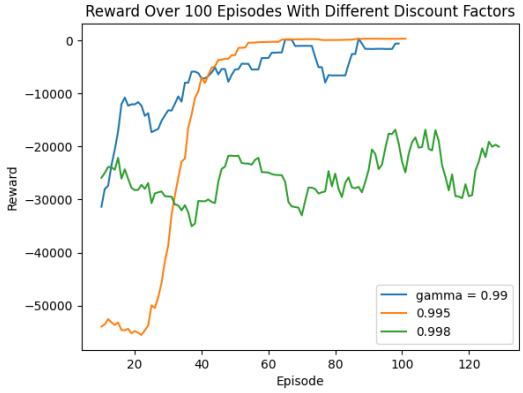

Figure 4: Gamma Parameter Sweep

In order to find the best discount factor to optimize our model's attention to delayed rewards, we trained PPO on three different discount factor: 0.99, 0.995, 0.998. The model converges the fastest with $\gamma = 0.99$. However, the reward dips a few times after convergence due to some mishaps. We notice that with $\gamma = 0.995$, the model achieves the most stable learning. With $\gamma = 0.995$, the model still pays attention to rewards achieved hundreds of frames later. For a game where the consequences of a move usually happens in the span of a few seconds (a few hundred frames), this was ideal. When we set the discount factor to 0.998, learning completely stops, as the model stops trying to optimize for short term rewards. Based on the results, we decided to use $\gamma = 0.99$ in favor of its fast convergence over the stability to enable us to perform fast iterations in experiments. Fig. 4 shows the reward in the training runs.
Comparing Double DQN and DQN for various rewards: For Double DQN with small state space, small action space and the min dist reward, they learn to minimize the distance after 5 iterations. DQN converges about 1-2 iterations faster than Double DQN, likely due to double DQN not allowing overestimation bias which would let the model fit faster.

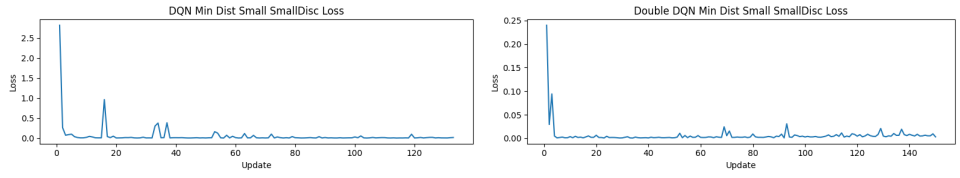

Figure 5: DQN and Double DQN training

For Double DQN with small state space, small action space and the stay alive reward, the spike in frame hit indicates the behaviour that the agent would kill itself and respawn on the podium, then stay on the podium as long as possible to avoid getting it (hence why it dies so quickly but takes such a long time to get hit once). It then learns that getting it is fine as long as it doesn't get hit/fall off the edge of the map, resulting in an increase in first death time and a decrease in first hit time. The

results for DQN look the same.

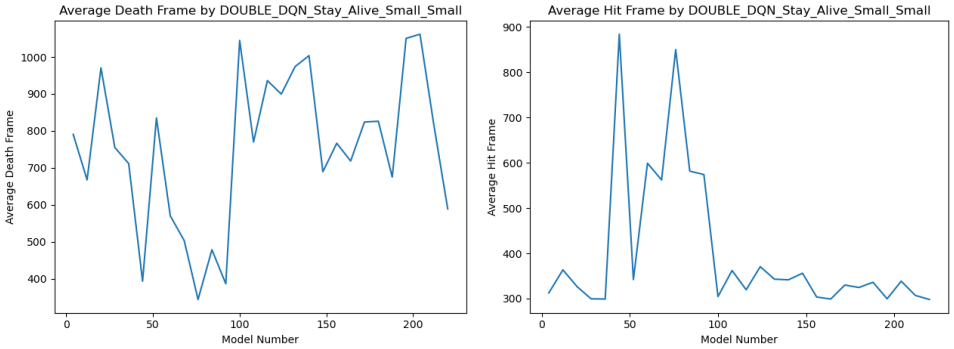

Figure 6: Double DQN Stay Alive results

When we increase the action and state space and use the large reward, Double DQN is no longer to fit anything. The training is noisy and the results indicate it wasn't able to learn much of anything. During evaluation, most of the graphs (avg deaths, time of first death, amount of attacks, etc), all look like noise. It does get hit at earlier times and survives a little longer though. The results for DQN with historical spaces looks the same. This is likely because there is too much data for DQN to train on (30 previous frames * 70 observation = 2100 size state spaces)

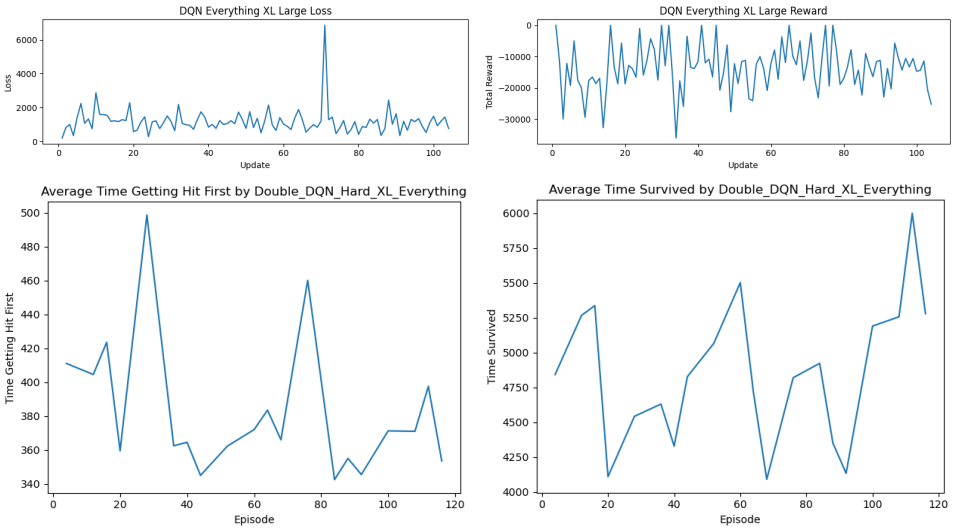

Figure 7: Double DQN XL Results

PPO struggles on min dist likely because PPO is a good general purpose algorithm, but in the midst of dense rewards with small states and actions, DQN is more efficient since there is an explicitly best move. As you can see, Double DQN is able to achieve significantly higher rewards at 225 compared to PPO at 72.5

PPO does slightly better on Stay Alive compared to Double DQN. As you can see, PPO consistantly survives 600 more frames than double DQN.

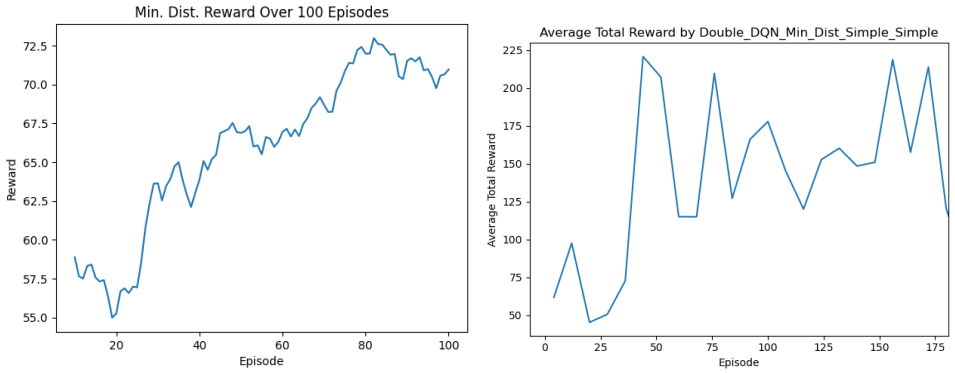

Figure 8: PPO vs Double DQN Min Dist

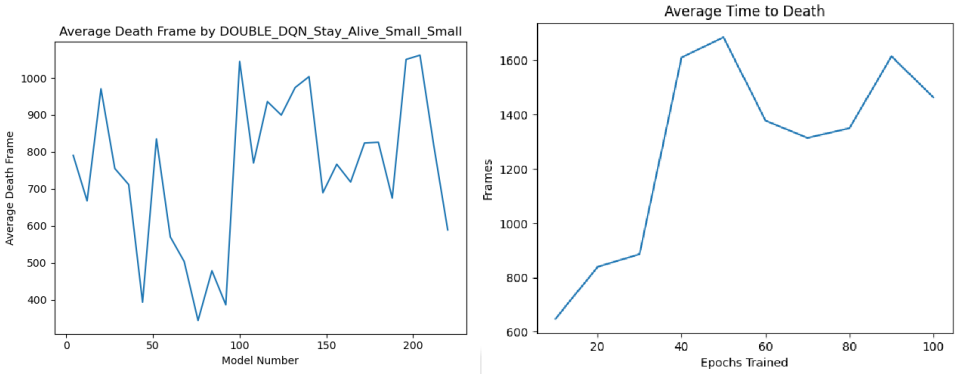

Figure 9: PPO vs Double DQN Stay Alive Average Death Time

Now when comparing PPO to DQN on the XL dataset, we are able to see PPO shine. PPO on average does 0.165 more damage than DQN per game which translates to 165 more points. PPO also on average survives 2000 grames more than DQN. PPO also averages 1 more kill than DQN per game

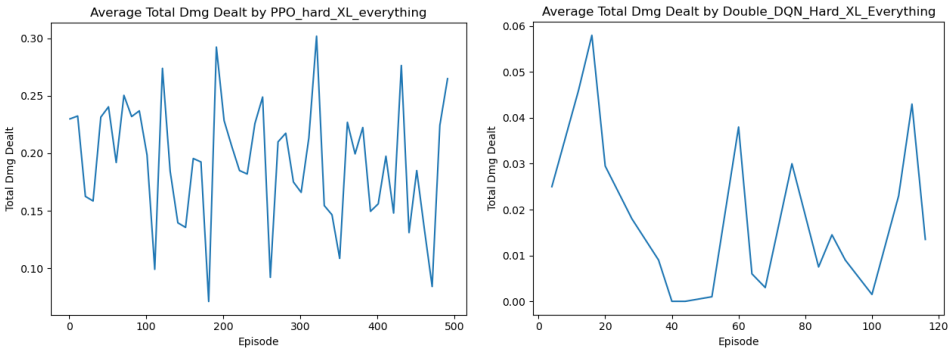

Figure 10: PPO vs Double DQN Total Damage Dealt

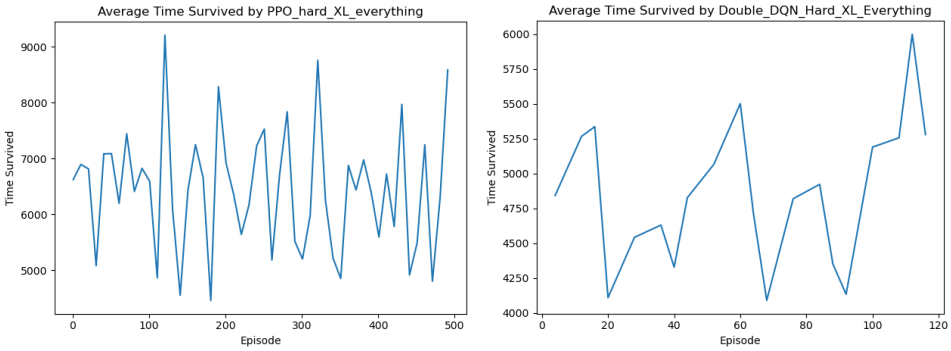

Figure 11: PPO vs Double DQN Survival Time

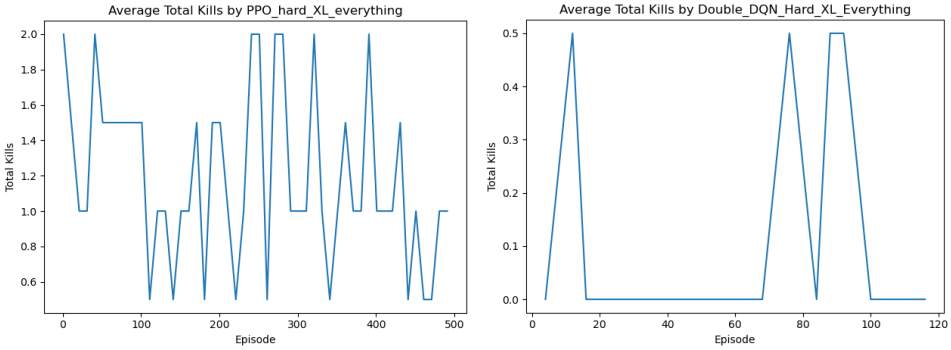

Figure 12: PPO vs Double DQN Kills

5.2 Qualitative Analysis

5.3 Offline Learning

Both BC and IQL were able to learn to perform natural-looking actions such as short-hop-aerial-attacks, and were even able to perform combination attacks such as chaining an up-throw into an aerial attack. However, due to the lack of training data, the agents were not able to perform the aforementioned moves precisely enough to beat a level 9 CPU. For example, it would perform an attack too far away from the opponent, resulting in a miss, and as a result, giving the CPU a chance for counter attack.

We evaluated both models by letting them play against a level 9 CPU for 10 games. On average, the BC agent was able to take off 1 out of 4 lives from the opponent, while the IQL agent was able to take off 2 out of 4 lives. In one of the extremely closed match, the IQL agent was able to stay even with the opponent all the way until the last life. Fig. 1 captured this exciting moment.

5.4 Online Learning

DQN and Double DQN perform approximately the same in game. They minimize the distance by running directly to the CPU and jump on to the platform, then stay there for the rest of the game. DQN and Double DQN both struggle an equal amount on the Stay Alive and overall Win the Game reward.

On the other hand PPO struggles on min dist as it never explores jumping on the ledge and constantly runs underneath the CPU for min dist. It does better on stay alive by 600 as its able to maximize the distance between itself an the CPU, consequently avoiding it's attacks. Lastly, PPO actually is able to learn to attack in the Win the Game reward and attacks the CPU constantly. It also doesn't kill itself by jumping off the edge of the map.

6 Discussion

6.1 Offline Learning

In both BC and IQL, the models were able to learn to do complex moves. However, the model was not able to perform them at a high enough accuracy to win any games. We believe this is due to a lack of data. With more diverse dataset and a larger model, the model should be able to learn to play better.

Because of the lack of data, we were also not able to see significant difference between IQL and BC agents. Moreover, IQL did not perform better even when trained on optimizing for the minimize-distance reward. As human players do not prioritize approaching the opponent, there aren't enough data points for the model to learn to minimize the distance.

6.2 Online Learning

DQN and Double DQN learn Min Dist very quickly. This is likely because Q-learning is very data efficient for rewards that are not sparse.

DQN and Double DQN struggle for Stay Alive due to the sparse delayed rewards. Both models struggle to delay the time of earliest death and delay the time of getting hit (we wanted to use reward shaping to achieve these behaviours, and did not include death time/hit time explicitly in our reward calculation). However DQN does slightly improve its performance on delaying earliest death, likely because it was able to sample climbing up from the ledge of the map. We noticed that for DQN, if an action combination isn't explored (ie. climbing the ledge, or doing an attack on the enemy) while epsilon-greedy was still large, it would be harder to learn this behaviour later on which was likely what happened to our Double-DQN network. Due to the reward for hitting the enemy being 100x smaller than dying, the reward for getting hit was likely getting overlooked. However when experimenting with larger magnitudes for the getting hit reward, it would just start falling off the map. Also 100x smaller makes sense as usually the character will die immediately if it's percentage is at 150

We then move on to Double DQN for the maximum reward, maximum state spaces, and all actions available. Needless to say.. it struggled a lot. Average deaths stayed consistent at 4 (maximum) number of deaths every round. However the average time survived did increase slightly after 100 epochs, although that could be due to noise. The average death time does seem to be delayed as well which is a decent sign that its fitting to avoid dying by instantly running off of the stage. However it never really learned to avoid attacks/do attacks, likely because each of these attacks cancelled each other out. Ie. hitting the enemy for 5 hp and then getting hit for 5 hp on the same frame would result in 0 reward. So having both rewards for attacking and defending might have prevented the model from learning to favour one or the other.

PPO does significantly worse than DQN and Double DQN for Min Dist. It struggles to discover the optimal strategy in Min Dist which is to move to the right (towards the opponent) and jump onto their platform, then do nothing. This is likely due to the data inefficiency in PPO since it is on-policy. Moreover, you can see that it only finds the 75 award rather than 225. PPO just runs towards the enemy, but does not explore jumping on to the platform, so it just hovers underneath the objective.

When it comes to staying alive, PPO does slightly better than Double DQN in terms of delaying time to first death. This is because deaths and attacks are slightly delayed rewards that are easier for PPO to learn. It takes about 3 seconds, 3*60 frames to fall off the stage, and 60 frames for attacks to land. PPO is able to slightly learn these delayed rewards. DQN also learns to prevent dying because we apply a negative reward the moment the character leaves the map.So the DQN architectures heavily favour staying at the center of the stage rather than dying. PPO explores more and learns about staying away from the enemy CPU to avoid getting hit.

When it comes to the final objective of winning the game, PPO does much better than DQN. PPO actually survives for a much longer time (600 frames) longer than DQN. Both models actually show a decrease in time before getting hit, likely because its trying to survive more than dying. Both models also show an increase in time before their first death, however PPO's is more consistent and the growth is slower but less jittery than DQNs. The total damage dealt by PPO also steadily increases, whereas DQN decreases, likely because DQN chooses to focus on staying alive rather than hitting the enemy. Also hitting the enemy requires a few frames before the reward is given so DQN would struggle learning this correspondence. PPO survives much longer and overall does amazing on the overall reward. It could likely improve with more training.

7 Conclusion

We conducted an evaluation of five RL paradigms: Behavioral Cloning (BC), Implicit Q Learning (IQL), Proximal Policy Optimization (PPO), Deep Q-Learning (DQN) and Double DQN across multiple state and action representations and three reward schemes in the challenging domain of Super Smash Bros. Melee. Our experiments reveal that value-based methods (DQN and Double DQN) rapidly master the dense Minimize Distance objective, but struggle when rewards become sparse and delayed (e.g., Stay Alive and"Win the Game).

By contrast, PPO, despite its lower sample efficiency on dense tasks demonstrates greater robustness to reward sparsity and delayed penalties. PPO consistently outperforms DQN variants on the Stay Alive objective (surviving 600 frames longer) and the full Win the Game task (surviving 2000 frames longer, dealing 0.165 more damage, and securing 1 additional kill per match). These results underscore PPO's ability to balance exploration and exploitation when direct value maximization becomes noisy or ambiguous.

Offline algorithms (BC and IQL) provide a useful bootstrap: BC rapidly imitates CPU behavior, and IQL leverages reward feedback to refine policy extraction, but neither alone achieves competitive play against Level-9 CPUs. Integrating offline pretraining with online fine-tuning remains a promising direction to accelerate convergence and overcome local optima.

Overall, our study highlights that (1) algorithmic choice must align with reward density, more complex algorithms like PPO arent always best if the state and action space is small and rewards are dense (2) richer state information (e.g. projectile data) and careful reward shaping require more expressive architecture (PPO) especially to deal with delayed rewards (3) hybrid training pipelines combining imitation and RL hold potential for scaling toward human-level play.

Future work would include: using more compute, multi-agent self-play, and human-feedback (RLHF) to ultimately create the "unbeatable" Smash Bros. agent. Also using IQL as a prior and finetuning online via PPO could use data super efficiently and result in very good behaviour.

8 Team Contributions

- **Danica Xiong:**
  - PPO Implementation/Training/Evaluation/Graph/Paramete tuning
  - DQN Implementation/Training/Evaluation/Graph/Paramete tuning
  - Double DQN Implementation/Training/Evaluation/Graph/Paramete tuning
  - Transformer PPO Implementation
  - Online training pipeline
  - Replay file training pipeline
  - Evaluation pipeline
  - Rewards testing
  - Writeups
- **Tony Xia:**
  - PPO Implementation/Training/Evaluation/Graph/Paramete tuning
  - BC Implementation/Training/Evaluation/Graph/Paramete tuning
  - IQL Implementation/Training/Evaluation/Graph/Paramete tuning
  - Data collection
  - Evaluation pipeline
  - Offline training pipeline
  - Data loading pipeline
  - Rewards testing
  - Writeups

Changes from Proposal We originally wanted to create an agent that is capable of beating real people and pro players. Unfortunately due to compute limitations, we were not able to do that as we require parallelized emulators to train online, or a ton more data for offline training. Thus, we scaled down our hopes and tested our frameworks on more, simpler environments and rewards. We still evaluated on the large rewards/hard CPUs, but we would likely see much better results with 50x more compute and data.

References

Pranav Agarwal, Aamer Abdul Rahman, Pierre-Luc St-Charles, Simon J. D. Prince, and Samira Ebrahimi Kahou. 2023. Transformers in Reinforcement Learning: A Survey. arXiv:2307.05979 [cs.LG] https://arxiv.org/abs/2307.05979

Vlad Firoiu, William F. Whitney, and Joshua B. Tenenbaum. 2017. Beating the World's Best at Super Smash Bros. with Deep Reinforcement Learning. arXiv:1702.06230 [cs.LG] https://arxiv.org/abs/1702.06230

Ben Parr, Deepak Dilipkumar, and Yuan Liu. 2017. Nintendo Super Smash Bros. Melee: An "Untouchable" Agent. arXiv:1712.03280 [cs.AI] https://arxiv.org/abs/1712.03280

Yash Sharma. 2017. Learning to Play Super Smash Bros. Melee with Delayed Actions.

A Additional Experiments

We also built a transformer multi-state PPO architecture. We tried to run it but the loss and update is way too slow to run on CPU and my computer doesn't have a strong enough GPU to run the update in time for the emulator to update :(But future works would include more compute time, parallelized compute, multi-agent self play, and RLHF.

B Implementation Details

Check out our github for code: https://github.com/danicax/smashRLagent
Check out our google doc for videos of the gameplay: https://drive.google.com/drive/folders/1n6TlWAceC3E1C6SHIbeQADvqvzrenzgB?usp=drive$_link$