CIS 6800 Final Project Report
MELEEGENT: VISION-BASED RL AGENT FOR *SUPER SMASH BROS MELEE*

Roberto Ligeralde, Ethan Yu, Kevin Liu

Abstract

Nintendo's *Super Smash Bros* franchise is a series of fighting games known for their unique form of combat. Compared to most fighting games (e.x *Street Fighter*, *Mortal Kombat*, and *Guilty Gear*), Smash Bros, particularly Melee, places a much greater emphasis on rapid movement and stage positioning, making it a dynamic environment for RL agents. While previous work has had success in using DQN and other approaches to play Melee at a competitive level, these models "cheat" in the sense that they rely on information not accessible to human players. We aim to use a visual backbone in place of this data, while maintaining superhuman performance.

1. PROBLEM STATEMENT

With its continuous action space and complex physics interactions, Super Smash Bros Melee provides a challenging environment for reinforcement RL agents. Unlike traditional fighting games with discrete inputs and well-defined state transitions, Melee's mechanics demand precise control over not only attacks and movement, but also spacing-the position of one's opponent relative to one's own attack range.

In previous research, RL agents have achieved notable success by leveraging game-state information such as character positions, velocities, and damage percentages. However, this reliance on internal game data presents a limitation for real-world applications, as it does not generalize well to situations where such direct access to the game state is unavailable. A more robust approach would involve training agents using only visual input, mimicking the way human players perceive and react to the game.

In this work, we aim to develop a vision-based RL model that learns to play Super Smash Bros Melee by interpreting raw pixel data from the game screen. This presents several challenges. First, the agent must not only learn to recognize key elements of the game, such as characters, hit points, and different projectiles, but also understand the intricate relationships between these elements as the game goes on. Additionally, the model must operate in real time, making decisions at a frame-by-frame level while contending with the inherent complexity of the game's physics and combat system.

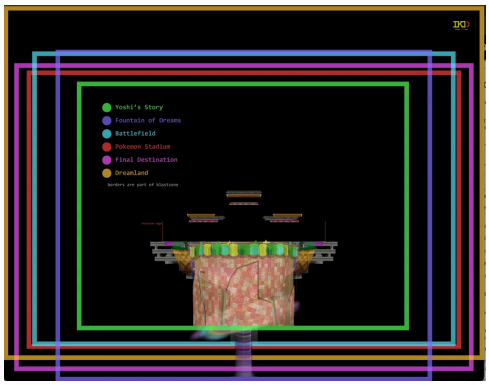

Figure 1: Overlay of the tournament legal Melee stages with visualized blast zones.

2. RELATED WORK

2.1 Phillip

Firoiu et al's "Beating the World′s Best at Super Smash Bros. Melee with Deep Reinforcement Learning" (2017) proposes an actor-critic agent now know by the Melee community as Phillip. While it achieved impressive results in defeating several Top 100-ranked players, its dependence on reading from game state on reading from game state gives it an arguably unfair advantage over human players and prevents it from playing with and against projectiles, which are not directly observable from game state. Furthermore, training was limited by the poor emulation technology of the time, with efficient emulators like Ishiiruka and associated tools like LibMelee (both discussed later on) not yet released. The author's claim that the same actor-critic approach used with a visual observation space would likely achieve similar results was the inspiration for this work.

2.2 Project Nabla

Chen's "Project Nabla: Towards human-like agents for Super Smash Bros. Melee" (2022) proposes using data from matches between skilled human players for behavioral cloning. In particular, they use game-state data saved from matches played via the aforementioned Project Slippi to predict what inputs the domain experts will take in a particular game-state, and fine-tuning this model via PPO (described more in detail below).

3. BACKGROUND

3.1 Foundation Models

Foundation models have emerged as a transformative paradigm in machine learning, leveraging massive datasets and architectures to learn general-purpose representations. These models, such as GPT for language and DINO for

vision, are pretrained on diverse data and fine-tuned for specific downstream tasks. In our project, we utilized DINOv2, a state-of-the-art self-supervised vision model, to extract frame-level features from gameplay footage of Super Smash Bros. Melee. The DINOv2 model is trained with a self-distillation loss, which encourages the network to produce consistent features under different augmentations of the same input. This property makes it particularly suitable for our application, as it captures semantically meaningful features invariant to visual transformations.

We leveraged these DINOv2 features to predict a predefined game state, providing a structured representation of the environment that the reinforcement learning agent could use for decision-making. This approach combines the strengths of self-supervised learning for representation extraction with task-specific fine-tuning, bridging the gap between raw visual input and actionable state information.

3.2 Reinforcement Learning

Reinforcement learning (RL) is a framework for sequential decision-making, where an agent learns to maximize cumulative rewards by interacting with an environment. More specifically, the aim of RL is to learn a policy π that takes as input a state s_t at timestep t and outputs an action a_t. In this project, we employed two key RL techniques: Behavioral Cloning (BC) and Proximal Policy Optimization (PPO). Below, we outline the theoretical foundations and derivations for these methods.

3.2.1 Actor-Critic Framework

The actor-critic framework is a foundational approach in RL that combines the strengths of policy-based and value-based methods. The policy $\pi(a|s;\theta)$, referred to as the actor, determines the agent's actions, while the value function $V(s;\phi)$, referred to as the critic, estimates the expected cumulative rewards from a given state s.

The objective of the actor is to maximize the expected return:

$$J(\theta) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r_t \right],$$

where $\gamma \in [0,1]$ is the discount factor and r_t is the reward at time step t. The gradient of this objective, known as the policy gradient, is:

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi \left[\nabla_\theta \log \pi(a_t|s_t;\theta) \hat{A}_t \right],$$

where $\hat{A}_t$ is the advantage function, representing how much better or worse the taken action was compared to the expected value.

The critic learns to minimize the temporal difference (TD) error:

$$\delta_t = r_t + \gamma V(s_{t+1};\phi) - V(s_t;\phi),$$

by optimizing the following loss:

$$\mathcal{L}_{critic}(\phi) = \mathbb{E}_\pi \left[\delta_t^2 \right].$$

This interplay between the actor and the critic enables stable learning, where the critic guides the actor's updates by providing feedback on the quality of its actions.

3.2.2 Behavioral Cloning (BC)

Behavioral cloning is a supervised learning approach to imitation learning. Given a dataset of expert trajectories $\mathcal{D} = \{(s_t, a_t)\}$, where s_t represents the state at time t and a_t the corresponding action taken by the expert, the goal is to learn a policy $\pi(a|s;\theta)$ that mimics the expert's behavior. The objective function is the negative log-likelihood of the expert's actions under the learned policy:

$$\mathcal{L}_{BC}(\theta) = -\mathbb{E}_{(s_t,a_t)\sim\mathcal{D}}[\log \pi(a_t|s_t;\theta)].$$

Minimizing this loss trains the policy to imitate the expert's action distribution, providing an effective initialization for reinforcement learning.

3.2.3 Proximal Policy Optimization (PPO)

Proximal Policy Optimization is a widely used policy-gradient method that improves the stability and efficiency of RL training by limiting policy updates within a trust region. Let π_θ denote the policy parameterized by θ and $\pi_{\theta_{old}}$ the policy before the update. The PPO objective is defined as:

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

where $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ is the probability ratio, $\hat{A}_t$ is the advantage estimate, and ϵ is a hyperparameter controlling the size of the trust region.

The clipping operation ensures that policy updates do not deviate excessively, stabilizing training and preventing performance collapse. The advantage function $\hat{A}_t$ is computed as:

$$\hat{A}_t = \sum_{k=0}^{\infty} \gamma^k \delta_{t+k},$$

where $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$ is the temporal difference error and $V(s_t)$ is the value function.

3.3 Context

In our project, we first initialized the policy using BC on expert trajectories to provide a robust starting point. We then fine-tuned the policy using PPO in a self-play setting, where the agent iteratively improved by playing against versions of itself. This combination of imitation learning and reinforcement learning enabled efficient exploration and refinement of the policy, leading to competitive gameplay performance.

4 . METHODOLOGY

4.1 Workflow Overview

By far the most time-consuming part of this project was setting up the training / inference infrastructure for our model. To begin, we first needed to locally compile

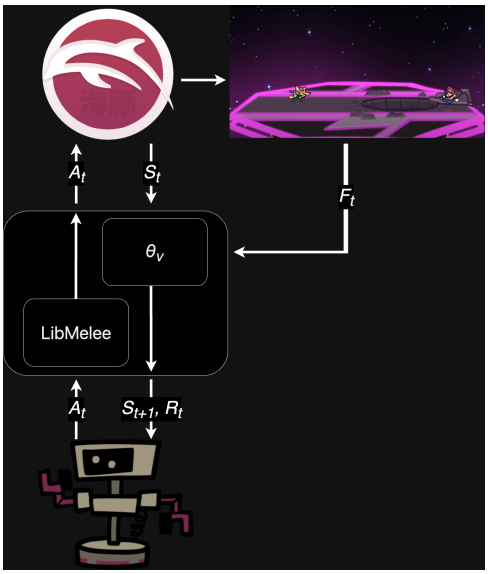

Figure 2: High-level overview of our visual inference pipeline. LibMelee allows our agent to communicate with the emulator, while we use XLib to obtain frames from the Ishiiruka window. Said frame is then passed to our vision network θ_v alongside the previous state, discussed in detail later.

Ishiiruka, a Nintendo GameCube emulator which can be controlled via the Python library LibMelee. Compilation proved surprisingly difficult-documentation was scarce as these projects are hobbyist-maintained, and both Ishiiruka and LibMelee had non-trivial dependencies like Rust and ENet respectively.

With this set up, we then implement an OpenAI Gym environment built on LibMelee. Our observation and action spaces are borrowed from Project Nabla: we define an observation as a `Dict` keyed by `'spatial'`, `'damage'`, and `'status'`. The first two are continuous, normalized arrays of the agent's and opponent's x, y coordinates and character / shield health respectively. Meanwhile, `'status'` contains several categorical variables such as the agent's stock count, their current action (ex: running, falling), and which direction they are facing.

We then define an action as a 10-tuple: the first four indices are discretized x, y coordinates of the two joysticks, the next 5 are indicators for each of a GameCube controller's primary buttons, and the last represents discretized scalar for the shoulder button position. Finally, we borrow the reward function from Phillip:

$$P_t^i = \max\{0, \Delta P_t^i\}$$
$$R_t^i = -\lambda_S \Delta S_t^i + \lambda_P \max\{\Delta P_t^i, 0\}$$
$$R_t = R_t^0 - R_t^1$$

$\Delta S_t^i, \Delta P_t^i$ represent the change in stock count and percentage of player i from timestep $t-1$ to t. We apply the max operator on ΔP_t^i since a player's percentage is reset after taking a stock, so this avoids having negative rewards from taking stocks.

4.2 Storage + Compute

For the duration of this project, we had on loan a machine with 4 NVIDIA TITAN RTX GPUs and 2 AMD Ryzen Threadripper CPUs. All training and compute-intensive pre-processing steps (particularly video reconstruction, see below) are conducted here.

Another bottleneck in our experimentation was that only one group member was allowed SSH access to this machine, and with none of our personal computers suitable to run Ishiiruka (it runs poorly on Windows and not at all on ARM CPUs), only one member could actually run and debug code. We also had to deal with the machine itself having issues-the Ubuntu installation sometimes acted up, and in rare instances we would have to deal with BIOS and power supply problems.

5 . DATA PREPROCESSING

5.1 Video + Controller Reconstruction

As mentioned previously, the most significant pre-processing step is putting our game-state SLP files back into video form. For this we use our Playback binary with Kevin Sung's slp-to-video, a NodeJS project which uses Slippi's"frame-dumping" to create MP4s from SLPs. Running multiple Playback instances in parallel allowed us to efficiently convert 400GB of SLPs into MP4s

Meanwhile, to process the expert trajectories into a usable form, we adapt pre-processing code from Firoiu's later work on Melee imitation learning. More specifically, this code formulates the joystick angles as a regression problem, while all the buttons become a single one-hot vector.

To prepare the video data for training, we downsample each video to a resolution of 224x224 pixels. This step reduces the image size while retaining enough information for the model to learn from, making the data more feasible for processing and training without overwhelming system resources. After cleaning out buggy video conversions and `PKL` files, there are about 250k frame, game state pairs in our vision dataset.

Finally, we convert our MP4s to HDF5 format and use these to define a HuggingFace Dataset class storing observations and expert trajectories. For now we restrict our training to a single match type-both players using Fox, the best character, and playing on Final Destination, the simplest tournament-legal stage as the only one without platforms.

6 . VISION TO STATE

6.1 State Regression

We train a vision model with a DINOv2 backbone to regress the game state variables at time t given two inputs: the frame at time t and the state at time $t-1$. The visual feature vector from DINO is concatenend with the prior state vector, which includes 3 components: *damage* (4-dimensional), *spatial* (4-dimensional), and *status* (12-dimensional), giving a total dimensionality of 20 for the prior state. All vision experiments are conducted with batch size 32.

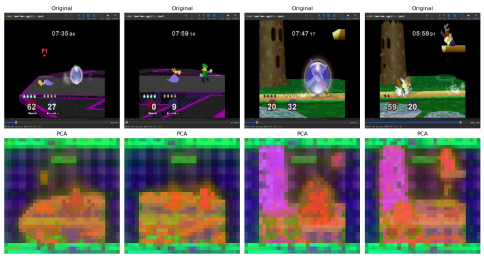

Figure 3: 3-PCA of DINOv2 embeddings on Melee gameplay

6.2 Architecture

The concatenated feature vector is passed through a two-layer fully connected fusion network, where each layer consists of a `Linear` transformation, `LayerNorm`, and `ReLU` activation. This network reduces the combined dimensionality into 512 hidden dimension space. The fused representation is then passed to three separate prediction heads: (1) a `damage` head that outputs a 4-dimensional vector, (2) a `spatial` head that outputs a 4-dimensional vector, and (3) a series of 12 independent `status` heads, where each head predicts specific categorical aspects of the 12-dimensional status vector like jumps remaining, etc.

6.3 Training Objectives

Our objective over training is to minimize loss between model predictions and expected state.

The loss computation is modular:

- **Damage Loss:** Computed using Mean Squared Error (MSE) to evaluate the predicted *percent* and *shield health*.
- **Spatial Loss:** Also computed using MSE to measure the accuracy of predicted x and y positions.
- **Status Loss:** A composite loss calculated for each subcomponent of the status vector. Each subcomponent prediction is passed through a softmax activation, and the loss is computed using Cross Entropy, weighted equally across 12 status components.

The composite loss is merely the weighted sum of the 3 loss functions.

This loss was trained over 10 epochs with the Adam Optimizer at a learning rate of $10e-4$. While the damage and spatial losses showed signs of improvements, the status loss oscillated with mean 20.

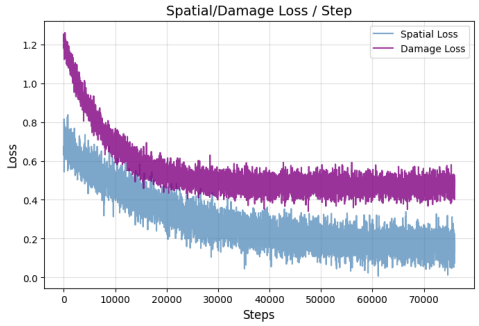

Figure 4: Spatial + Damage Loss/ Step

Figure 5: Status Loss / Step

6.4 Training Experiment v2

As the previous approach of an aggregate loss function failed, we next experimented with an iterative training paradigm where we train spatial, damage, and status loss consecutively and "independently". First, we train only the spatial loss. The loss converges for the x, y positions of the characters very quickly with learning rate $10e-4$ Adam, and 1 epoch.

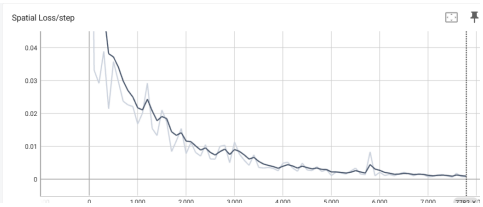

Figure 6: Spatial Loss/Weight Update Steps (Sorry for the bad screenshot)

Next, we train the combined loss function, which includes both damage and spatial components, for one epoch using the Adam optimizer with a learning rate of $10e{-}4$. Once again, both loss terms converge successfully.

$$\mathcal{L} = 0.2\ell_{\text{spatial}} + \ell_{\text{damage}}$$

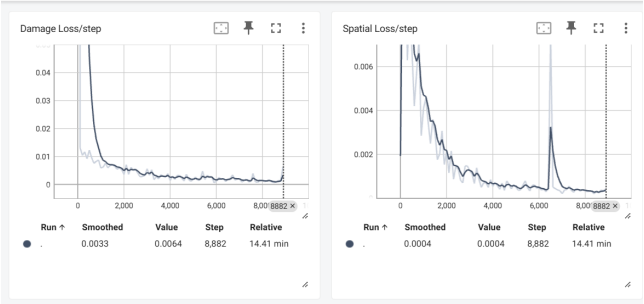

Figure 7: Spatial + Damage Loss/Weight Update Steps

Next, we train the final combined loss function with all 3 sublosses. We train over 2 epochs using a learning rate schedule of $1e-4$ for the first 2 epochs, $2e-5$ for the next 2 epochs, and finally $1e-6$ for the last epoch using the Adam optimizer.

$$\mathcal{L} = 0.1\ell_{\text{spatial}} + 0.2\ell_{\text{damage}} + \ell_{\text{status}}$$

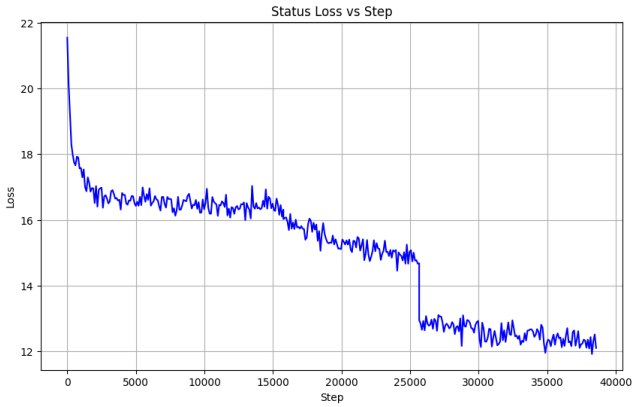

Figure 8: Status Loss/Weight Update Steps

The status loss converges to approximately 12. The challenging aspect lies in learning the coefficients for the 410 possible actions a character can perform. To enhance this objective, we may require a more powerful architecture or a more effective strategy to address class imbalances, as some actions in the space are highly specific to particular situations.

7 . REINFORCEMENT LEARNING

7.1 Setup

Our RL training utilizes Imitation, a library built on top of Stable Baselines 3 with pre-built trainers for behavioral cloning and PPO. We train the `MultiInputActorCriticPolicy` from SB3, which consists of of 2 MLP's (we initialized to 2 layers and 64 hidden units) and a feature extractor designed for dictionary observation spaces.

Running our policy against a benchmark is key for both evaluating BC quality and selecting the latest policy in PPO. To this end, we set the opponent policy in our environment as Melee's in-game Fox CPU and take average reward received by our policy over T frames.

7.2 Behavioral Cloning

We train behavioral cloning for one epoch on 334 expert games, amounting to roughly 3 million (state, action) pairs. We employ a batch size of 32 rollouts and use the Adam optimizer with a constant learning rate of $\eta = 10^{-4}$

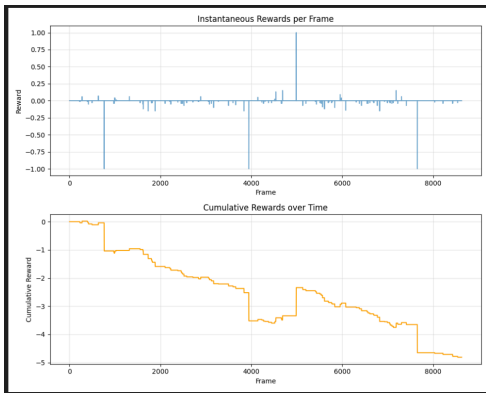

Figure 9: BC Agent Versus Level 9 CPU Reward Plot

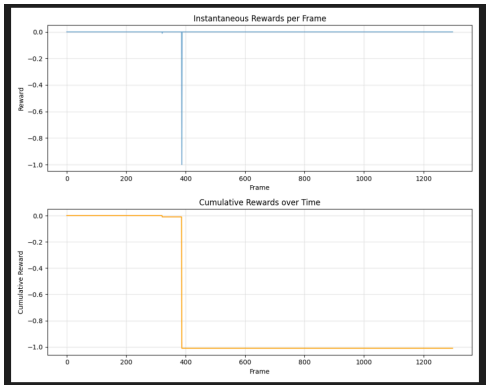

Figure 10: BC Agent w/ State Predictor Versus Level 9 CPU Reward Plot

As expected, the pure BC agent does not perform well against the in-game CPU with and without the state predictor model attached. Reading from gamestate it loses 4 stocks to 1, and reading from the vision model it lost 1 stock before time ran out.

We note that far more effective BC is possible via recurrent policies such as `RecurrentMultiInputActorCriticPolicy` from `sb3-contrib`. This is because high-level Smash players think in terms of long sequences of movements and attacks: individual button presses are not particularly meaningful. However `Imitation's` BC trainer did not support such policies and we did not have enough time to implement our own.

7.3 PPO Self-Play

After training an inital "coarse" model with BC, we fine-tune it with self-play. For each epoch, we run PPO on the learned policy for 1000 iterations, with 5000 timesteps per iteration. The agent is initialized with the policy from (6.1), while the initial opponent simply samples the action space randomly. We update the opponent every 10 timesteps, to the best performing policy so far (by our benchmark).

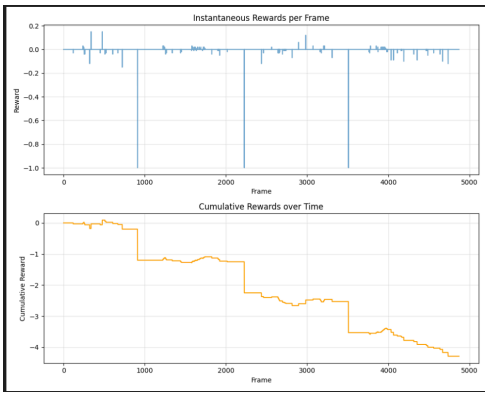

Figure 11: PPO Agent vs Level 9 CPU Reward Plot

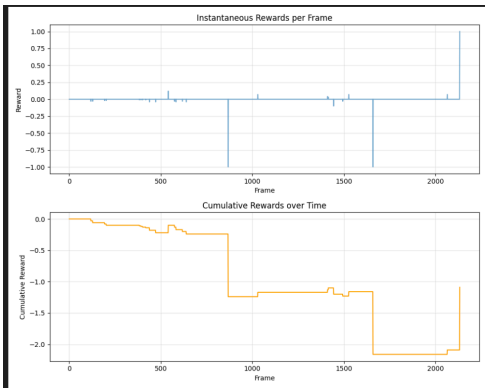

Figure 12: PPO Agent w/ State Predictor vs Level 9 CPU Reward Plot

Our PPO results were highly disappointing, as it only managed to take one stock across both gamestate and vision model outputs. To improve, we would not only pursue a stronger BC model via recurrence, but also simply train longer: Phillip and Nabla were both trained for weeks, while the above models were only trained for less than a full day. Recurrence would most likely be helpful here as well, as the model could pick up on Melee nuances like the stale move queue: moves doing less percentage and knockback the more they are used repeatedly.

8 . CONCLUSION

8.1 Future Work

Beyond simply training on more characters and stages, as well as training longer (Phillip and Nabla were trained on the order of weeks using superior hardware), we see two clear paths for further experimentation. Firstly, we would like to try stepping away from game-state regression entirely, instead using a learned representation from the vision network to predict next actions directly.

Furthermore, there is potential in imposing a reaction time constraint on our model. While MeleeGent is more fair than previous work in that its observation space is equal to that of a human player, our current setup allows it to select and execute a new action every frame-even the most skilled humans will face a roughly 20 frame delay in parsing the current scene and pressing the appropriate buttons. It would be interesting to see if our current training setup is robust to such a constraint.

Lastly, employing both these concepts would prove useful-a non-regressive, reaction time-constrained model would be on a level playing field with humans, meaning human players could directly borrow from its playstyle. Perhaps MeleeGent could be for Smash Bros what Stockfish has been for chess.

9 . SPECIAL THANKS

We would like to thank a few individuals outside of our group who were integral to the completion of this project:

1. Huzheng Yang: for allowing us to borrow (and for helping run tech support on) the aforementioned machine. Having access to the Ishiiruka GUI was critical for debugging the environment especially-we most likely would not have even completed that step without it.
2. Vlad Firoiu: for regularly answering questions about Ishiiruka and Phillip on Discord. As mentioned previously, documentation for our most crucial tools was extremely slim, so having someone already familiar with them (especially with respect to our use case) was essential.