Comparing Reinforcement Learning and Imitation Learning in Teaching a Model to Play Games

Angela Liu
University of Maryland, College Park
Department of Computer Science
College Park, United States
aliu1237@umd.edu

Laura Zheng
University of Maryland, College Park
Department of Computer Science
College Park, United States
lyzheng@umd.edu

Ming Lin
University of Maryland, College Park
Department of Computer Science
College Park, United States
lin@umd.edu

***Abstract*—Behavioral cloning (BC) for first-person shooter (FPS) games has shown promising results in contexts of limited information. However, BC models are no exception to issues with compounding errors, stochasticity, and temporal dependencies commonly observed in other applications like autonomous driving. In this paper, we build upon foundational previous work to address failure states in fine-tuning. Through preliminary analysis of existing behavioral cloning models in Counter-Strike: Global Offensive (CS:GO), we find four general categories of failure states. Then, we propose a data aggregation method for adjusting the agent from a failure state, collecting the recovery action data, and fine-tuning the behavioral cloning model. However, these manual adjustments do not significantly affect the base performance, which still struggles to achieve human-like performance. We then looked into applying DQfD, a reinforcement learning algorithm, to train a new model. We demonstrate the potential of this new approach and propose a new methodology to improve our results.**

***Index Terms*—first-person shooter, game AI, data analysis**

I. Introduction

Recent work in behavioral cloning (BC) for first-person shooter (FPS) games have shown promising results in games without convenient APIs or large-scale simulation, such as Counter-Strike: Global Offensive [1]. Behavioral cloning is a technique where an agent learns an action policy from expert demonstration in a supervised fashion, whereas an agent learns in deep reinforcement learning (DRL) through free exploration of a well-defined environment with a well-defined reward function. DRL is also more commonly used in artificial intelligence (AI) for games that are easily simulated, examples including DeepBlue Chess [2] and DOTA [3]. However, the application of vision-based behavioral cloning from first-person perspectives faces several challenges compared to their easily simulated counterparts and is much less studied in the context of limited information.

Specifically, behavior cloning suffers from three aspects: 1) compounding errors, where errors accumulate over a trajectory and lead the agent into a state unrepresented in the training data, 2) stochastic expert actions, where multiple “correct" expert actions exist and the averaged expert action is not a valid action, and 3) Non-Markovian observations, where observations depend on multiple previous states (e.g. a trajectory) rather than only the current state [4], [5]. While these limitations are well-explored in applications such as autonomous driving, it is a relatively new technique for tactical multi-agent game AI, where agents operate under a competitive zero-sum objective.

Additionally, major changes to the game itself could break a model trained used behavioural cloning, which is very sensitive to game images. Updates to UI or game features could deviate far enough from the outdated demonstration data that the model becomes confused. This is especially true for CS:GO, which recently updated to be Counter-Strike 2 as of September 27, 2023 and introduced many UI changes that broke previous work in fine tuning. However to avoid confusion, we will be referring to the game as CS:GO for the rest of the paper.

Reinforcement learning can be used to mitigate these limitations. Unlike behavioural cloning, models learned through rl are more adaptable to changing environments and can recover better from errors. However, the amount of data required to train an agent to become proficient in a complex game such as first-person-shooters is usually too large to be practical. Instead, we can use a modified algorithm Deep Q-learning from Demonstrations(DQfD) [6] that utilizes existing demonstration data to speed up the training process.

In this paper, we present two different methods for training a model to play CS:GO - one using behavioural cloning with fine tuning and another using the DQfD algorithm. In the beginning, we will explore the application of behavioral cloning methods on FPS games and propose a simple method for failure state recovery based on mini-map and player localization cues. A visual summary of our method can be found in Figure 2. Later, we created a custom gym environment for running the CS:GO game and demonstrate the process that allows our model to converge to near zero loss and increase training score over time. However, further improvements to how we do reward engineering is needed.

II. Background

A. Behavioral Cloning for FPS Games

Behavioral cloning (BC) is a common and well-explored method for replicating human behavior in applications other than video games, such as autonomous driving and robotics [7]–[9]. This method extends naturally to video games since video games can be considered simulation-only and

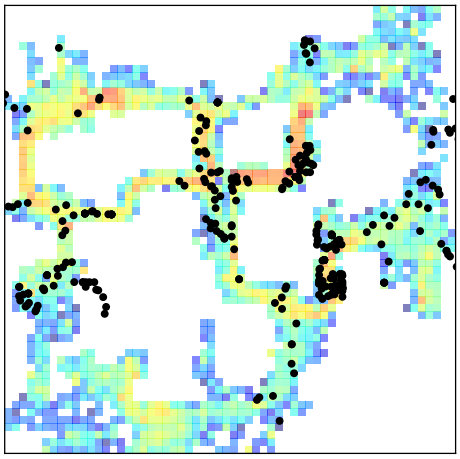


Fig. 1: Visualization of encountered failure states over a 10 minute time frame. Here, we plot the positions of failure states (black) in relation to the density of map coverage from the fine-tuned DM agent of [1]. Failure states often occurred in sparse areas adjacent to popular areas.

noiseless robotics problems [10], [11]. Recently, Pearce and Zhu provide foundational work in behavioral cloning [1] for first-person shooter games, where large-scale simulation is difficult for successful deep reinforcement learning, which is what most state-of-the-art game artificial intelligence (AI) methods employ [12]–[14]. This foundational work is the first to employ behavioral cloning in first-person shooter settings, where action and observation spaces are high-dimensional.

Behavioral cloning is a form of "imitation learning" where an agent learns to mimic the action of a demonstrator, typically a human expert. This technique can be applied to both FPS games and autonomous robotics, albeit with a few differences. Video games have a much lower risk factor than autonomous driving, so incomplete datasets are acceptable and environment exploration is cost-efficient. FPS game agents can also incorporate tuning into systems since they operate in a more restricted environment than autonomous vehicles do.

For convenience, we reiterate the objective of behavioral cloning from [1]. Given offline observation data pairs $\mathcal{D} = \{\{a_1, o_1\}, ...\{a_N, o_N\}\}$ collected from an expert control policy $\pi^*(a|o)$ representing a probability distribution of actions $a \in \mathbb{R}^M$ for M degrees of freedom, the objective of behavioral cloning is to learn a policy $\pi_\theta(a|o)$, parameterized by θ, which optimizes the following objective:

$$\theta = \underset{\theta}{\operatorname{argmin}} \sum_{i}^{N} L(a_i, \pi_\theta(\hat{a}|o))$$

where $L : A \times A \to \mathbb{R}$ represents a loss function.

B. Reinforcement Learning

Reinforcement learning, in contrast to behavioural cloning, is a training method whereby the model learns through trial and error. The agent receives a reward or punishment as it interacts with its environment and gradually learns to maximize its reward. Through process, rl algorithms can discover the optimal strategy without outside assistance, making it a powerful tool for adapting to new environments. This feature can be especially useful in the context of popular games that receive regular updates. Reinforcement learning has commonly been applied to games such as Atari, Chess, and Go with great success.

III. Methodology

A. Failure State Recovery

We define a failure state as a situation where the player policy is unable to move in any degree of freedom for more than five seconds. We set this threshold of five seconds in the context of a competitive tactical shooter format, since opponents will be other players, and movement in the game, in general, should be constant.

In general, trained player policies will fall into a failure state due to compounding errors commonly present in imitation learning problems for robotics. A common behavior bias found in datasets collected from human users is that the user almost never faces the wall while also being close to the wall; in other words, the expert player which the imitation learning model learns from is almost always facing away from the walls of map corridors.

A popular technique addressing compounding errors is Data Aggregation (DAgger) [15], [16]. Intuitively, DAgger address out-of-distribution observation states by incorporating new offline data in each iteration from the expert policy π^*. As the student policy π experiences new states, the expert generates the corresponding "correct" action and adds it to the data pool for policy π to learn from.

Implementing this for FPS games, however, is difficult due to the lack of an existing expert policy. While there are large amounts of human-generated data possible, coupling new observations with expert actions requires a human to be present during DAgger iterations, making the naive application infeasible. Alternatively, we propose a modified technique for handling failure states without the need for an expert policy. This method is further outlined in Algorithm 1.

Algorithm 1 Modified DAgger for Behavioral Cloning

Result: Trained policy $\hat{\pi}^*$
Dataset $\mathcal{D} \leftarrow \varnothing$ $\hat{\pi}_0 \leftarrow$ Random Initialization $o \leftarrow$ Initial observation
for $n \leftarrow 1...N$ **do**
  $\pi_i \leftarrow Recovery(\pi_i)$; `// Modify candidate`
  $D_i = \{(s, \pi_i(s)\}$; `// Sample trajectories`
  $D \leftarrow D \cup D_i$; `// Combine datasets`
  Train $\hat{\pi}_{i+1}$
end

We found in existing behavioral cloning models that failure states fell into three categories: overshot up, overshot down, and facing walls. Occurrences occurred the most in areas of

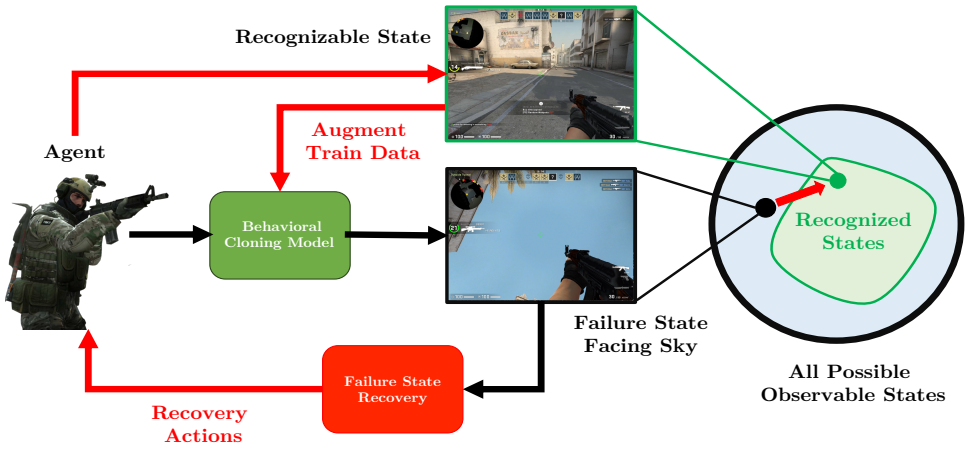
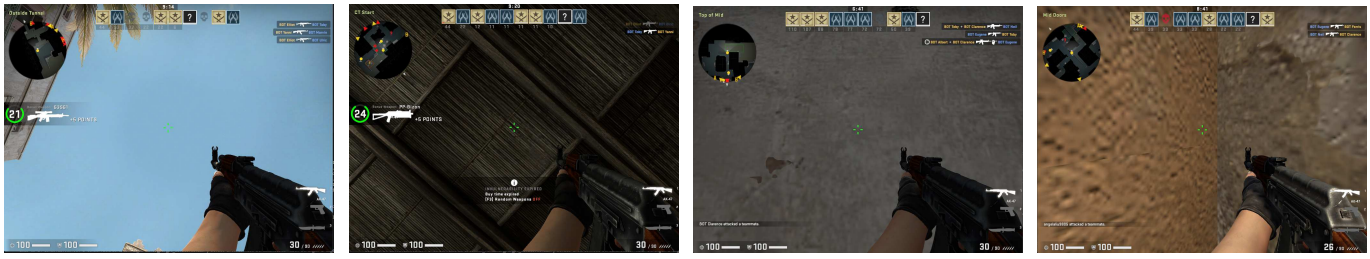


Fig. 2: Overview of our method. Our method, denoted by red arrows, depicts the failure state recovery routine for a baseline behavioral cloning model in green. The result of our routine is thus nudging the agent's observed state into a recognizable distribution, then recording the recovery actions for future fine-tuning.



(a) Facing sky (b) Facing ceiling (c) Facing wall (d) Facing corner

Fig. 3: Four examples of failure states.

Algorithm 2 Recovery Policy

Data: Candidate policy $\hat{\pi}$, Horizontal turning interval θ_Z, Vertical turning interval θ_Y, Max turning angle β
Result: Modified Policy $\hat{\pi}_R$ and chosen action a
$a \leftarrow \hat{\pi}(o)$ $a \leftarrow clip(a, -\beta, \beta)$ $s \leftarrow Environment(a)$
if *Player facing wall in* s_t **then**
  $a \leftarrow \theta_Z + a_{RotX}$
end
if *Player facing sky* **then**
  $a \leftarrow -\theta_Y + a_{RotY}$
end
$s \leftarrow Environment(a)$

the game map closest to edges. Examples of failure states are pictured in Figure 3.

To address these challenges, we use RGB pixel values on specific parts of the game window and used this information to make slight adjustments. This is feasible since video games are noise-less simulation environments. For example, we can expect that map textures and HUD locations and colors will stay constant over multiple iterations of a game.

We conduct checks for vertical adjustment scenarios by focusing on the values closest to the center of the screen, where the crosshair is. If this value happens to be close to the RGB value for blue, we iteratively nudge the mouse down until the color changes shade. We make the assumption that sky colors are the closest to the RGB value $(0, 0, 1)$ than any texture on the map. Note that this does not handle areas with ceilings; we leave this to future work, as it is nontrivial to implement without a public API.

We use a similar method for horizontal adjustments, except now focusing on the minimap on the top left of the screen. The minimap is a representation of the player's local 2-dimensional position, with the player always at the origin and facing forward. The minimap rotates along with the player, allowing us to check for objects or walls in front of the player without having to perceive a complex 3D scene. Flat, black values in front of the player on the minimap are indicative of the player facing a wall. The agent's rotation is nudged again in small intervals until the pixel value on the minimap changes.

B. Adapting DQfD for CSGO

The weaknesses of our previous fine tuning approach was its lack of flexibility. The hardcoded values for pixels and turns meant even slight deviations to the game ui and commands could break our implementation. This problem was especially highlighted when CS:GO updated to Counter-Strike 2 and a couple features were not ported over. Differences included a transparent radar, removal of bot difficulty options, and changes to memory addresses, all of which made continuing our original work difficult. By apply Deep Q-Learning onto a

gym environment that plays CS:GO, we could leverage one of reinforcement learning's greatest strengths - its adaptability.

Our first major challenge was creating a new CS:GO gym environment to run the algorithm on. We found existing gym environments that ran CS:GO on steam for linux but not for windows. Thus, we implemented our environment from scratch, using the linux code as a reference. Gameplay data was gathered through a combination of screenshots, gsi requests, and dumped offsets. In the process, we encountered several other challenges. Since the implementation we used passed in integers for actions, we needed to figure out how to map discrete data to all necessary action states. Considering that people often input multiple key presses at once when playing fps games and that players can look anywhere on screen, this was not a straightforward conversion. Camera position could be addressed by binding the arrow keys to player rotation, which adds 4 new action states. The next major decision was how to incorporate combined actions. Players commonly hold down multiple keys at once and limiting key inputs hinders gameplay. One approach was to create new actions that corresponded to multiple keys but after extensive testing, there was no major difference between mapping exclusively to single keys and mapping to multiple keys. To keep total action states low, we only considered single keys commonly used in our own play. Our action space included 11 total actions: fire, reload, jump, camera up/left/down/right, and move forward/left/back/right. These actions are also listed in the order of precedence our data collection process used to extract the key press per frame. For instance, the forward key is often held down but the player may want tap the right arrow key a couple times to make a right turn. In this case, turn right ranks higher than move forward so the recorded action will be a series of forward moves punctuated by right turns.

Before running the model, we collected our own demonstration data for 16 different 10 minute rounds. Each run collected screenshots and key pressed, which would be mapped to a number 0-10. To save space on our machine, we converted each image to grayscale and shrunk them in size. We paired the screenshots folder with the actions file for each run and use it for our pretraining.

Lastly, we conducted extensive reward engineering on the model get the agent to shoot as many enemies as possible. This was done by running the model and adjusting the reward function through several iterations. The specific steps of the modified DQfD code can be seen in Algorithm 3.

Algorithm 3 Modified Deep Q-learning from Demonstrations

Data: Demonstration data set D^{replay}, random initial behaviour weights θ, random target weights θ', update freq τ, number of pretraining gradient updates k

for *steps* $t \in \{1, 2, ...k\}$ **do**
  Sample a mini-batch of transitions from D^{replay} with prioritization
  Calculate loss $J(Q)$
  Perform a gradient descent step to update θ
  if $t \mod \tau = 0$ **then**
    $\theta' \leftarrow \theta$
  end
end
for *steps* $t \in \{1, 2, ...\}$ **do**
  Sample action a from behaviour policy
  Play action a
  Observe s'
  $r \leftarrow (\Delta\text{score} + 2 * \Delta\text{kills} + \Delta\text{assists} - \Delta\text{deaths} + 0.01\lfloor\text{time}/30\rfloor)$
  Store (s, a, r, s') into D^{replay}
  Sample mini-batch of n transitions from D^{replay} with prioritization
  Calculate loss $J(Q)$ using target network
  Perform gradient descent step to update θ
  if $t \mod \tau = 0$ **then**
    $\theta' \leftarrow \theta$
  end
  $s \leftarrow s'$
end

IV. Results

A. Fine Tuning

We show results comparing the properties of the behavioral cloning model with the vanilla routine versus our modified recovery routine. In terms of hardware, all results were produced on Intel Xeon Gold 5218 CPUs (2 cores), 64 GB of memory, and an NVIDIA RTX 2080 Ti graphics card. Game settings are replicated from [1].

Overall, we find that our routine diversifies the horizontal and vertical mouse movement labels compared to the baseline dataset. A visualization comparing the two distributions can be found in Figures 4 and 5.

To evaluate how our recovery policy may influence the performance of the baseline agent directly, we also quantify the difference in deathmatch metrics. Deathmatch is a free-for-all game mode in CS:GO where every other player is an enemy of the agent. The goal is to get as many eliminations as possible. The metrics used are Kills Per Minute (KPM) and Kill-Death Ratio (KD). Detailed comparisons for this can be found in Table I. While kill per minute and kill death ratio are worse than the original stats for each category, our routine performs better consistently for both values compared to an untouched routine that did not implement any tuning. We also quantify the occurrence of different failure state scenarios in Table II for each training model. Over a five-minute session, we find that horizontal turning happens less frequently with the best model, ak47_sub_55k_drop_d4_dmexpert_28, and that all models do not engage in vertical turning. We also find that most horizontal turning involves scenarios facing narrow corridors and sharp turns. As we see in Figure 1, most of the failure states are concentrated in sparsely explored areas directly adjacent to areas with high map coverage.

B. DQfD Implementation

Our final model training statistics show promising results. We find through our loss and training score that performance

TABLE I: Evaluation comparison for Kills-Per-Minute (KPM) and Kill-Death Ratio (K/D) between the baseline behavioral cloning model and our method. Without any additional training, our method enhances the performance of the baseline model by up to 10 times.

	Easy		Medium	
Model	KPM↑	K/D↑	KPM↑	K/D↑
Baseline [1]	0.29	0.40	0.03	0.05
Ours + Baseline [1]	**0.33**	**0.44**	**0.44**	**0.32**
Expert Baselines				
Built-in Bot (easy)	2.11	1.00	–	–
Built-in Bot (medium)	–	–	2.41	1.00
Human (Non-gamer)	4.25	1.80	2.38	0.90
Human (Casual gamer)	4.20	4.20	3.51	2.48
Human (Strong CSGO player)	14.00	11.67	7.80	4.33

TABLE II: Average rate of failure states per minute for baseline fine-tuned behavioral cloning model gameplay.

Model	Wall	Sky
ak47_sub_55k_drop_d4	30	0
ak47_sub_55k_drop_d4_dmexpert_28	32	0
ak47_sub_55k_drop_d4_aimexpertv2_60	41.6	0
July_remoterun7_g9_4k_n32_recipe_ton96__e14	39	0

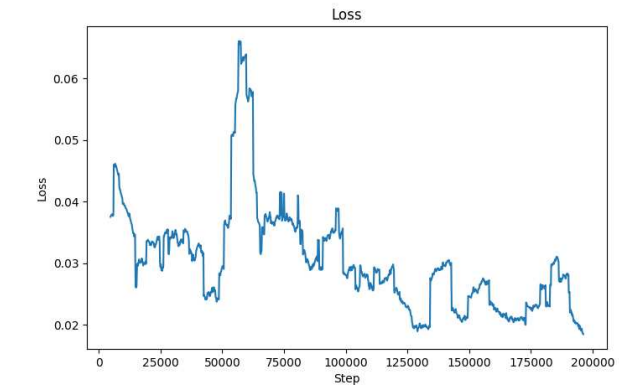
TABLE III: Evaluation comparison for Kills-Per-Minute (KPM) and Kill-Death Ratio (K/D) between the baseline behavioral cloning model and our method. Without any additional training, our method enhances the performance of the baseline model by up to 10 times.

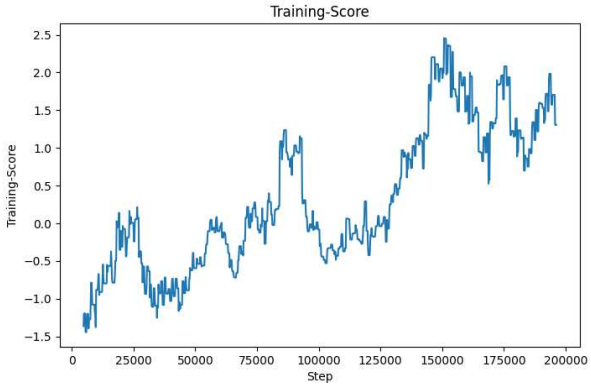
Model	KPM↑	K/D↑
Past Model [1]	**0.44**	**0.32**
Baseline [1]	0.03	0.05

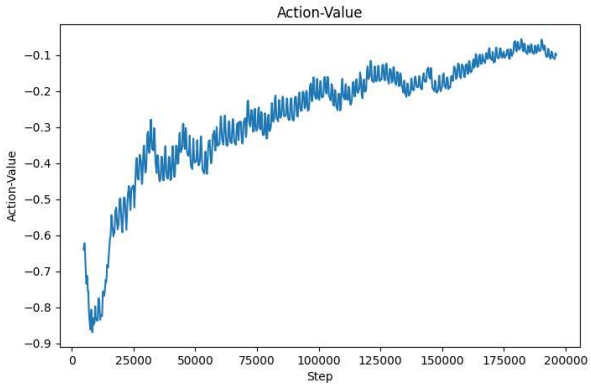
generally goes up. Loss decreases until it plateaus around 0.02 and both training score and action value displays a consistent increase. In sum, the average loss was 0.032, one of the lowest out of all test runs.

However, our model still struggles when put into practice. The agent tends to move around randomly before stopping completely after a couple seconds. The agent stays motionless until it gets killed and begins moving again.

Average Losses: 0.032
target_interval = 10000
warmup_steps = 50000
pretrain_steps = 10000
learning_interval = 4
num_steps = 200000
num_episodes = 900
max_steps_per_episode = 1000
output_freq = 100
save_freq = 50







V. Conclusion

In this paper, we present a simple method for failure state recovery in behavioral cloning for CS:GO without the need for an expert policy. We find through our analysis of baselines that failure states can be easily inferred through pixel checks on the noiseless environment, and implement recovery actions accordingly to augment the training dataset for fine-tuning. We also explored the possibility of using DQfD to train a model to play CS:GO using a combination of imitation learning and reinforcement learning. Although performance did not exceed one trained through behavioural cloning, we shown there is promise on continually expanding on this work.

Our work has several limitations. Firstly, our method is not reflective of human-like recovery and only addresses entirely expert-free scenarios. Future work can take advantage of both limited expert recovery data and our expert-free recovery ac-

tions to produce realistic behavior. Additionally, our results are limited in terms of fine-tuning results. In future work, we seek to evaluate our augmented recovery dataset on a fine-tuned FPS behavioral cloning model and provide additional insight into its value to high-dimensional deep learning policies. Additionally, we recognize that hardware differences between our setup and baseline setups can significantly influence baseline model performance; results for baselines are re-evaluated on our setup for this reason.

Our work in implementing DQfD also did not bring about a better improvement. While DQfD worked fine for simple 2D atari games, extrapolating it to a more complex 3D environment is not so simple. One major problem is the high dimensionality of our data. Our agent can move in 27 directions, much more than the 8 directions of a 2D avatar.

Another problem is reward sparseness. While most atari games have simple goals where success and failure can be determined relatively quickly, CS:GO is much different. Whether an agent looks upwards 1 degrees or moves to the right for 5 frames is not strongly correlated to whether it is going in the right direction. Additionally, killing an enemy by chance is extremely rare and there is no guarantee whether the agent can repeat it or learn anything from so few successes. A game like CS:GO needs a complex reward function as well as carefully tuned weights, both of which was unfeasible with manual reward engineering. These problems contribute to the reason that the game was difficult to train by reinforcement learning.

This could be addressed by incorporating smarter reward engineering practices. One method is Direct Behavior Specification via Constrained Reinforcement Learning [17], which automatically weighs each of the behavioral constraints and finetunes the model for desired behavior. Additional work could be done for extracting more information from the game itself as model is training as understanding map layout from screenshots alone is likely not enough. In the future, many modifications can be made to our work for better performance.

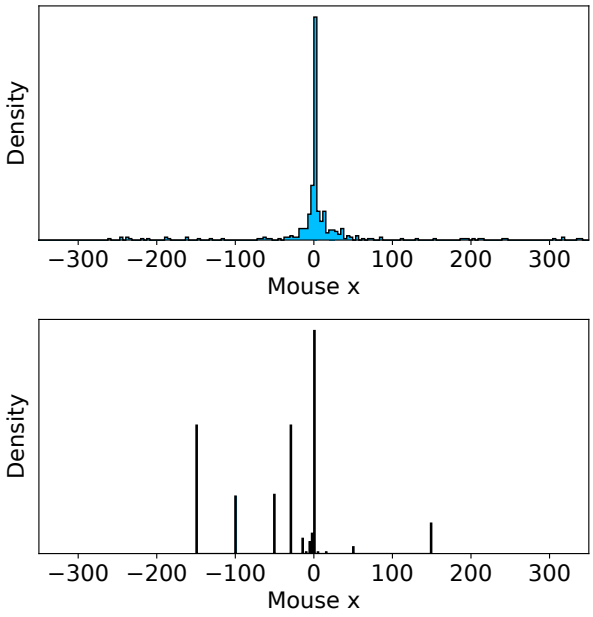


Fig. 4: Horizontal movement distribution comparison between baseline data (top) and 400 minutes of our augmented failure state recovery data (bottom).

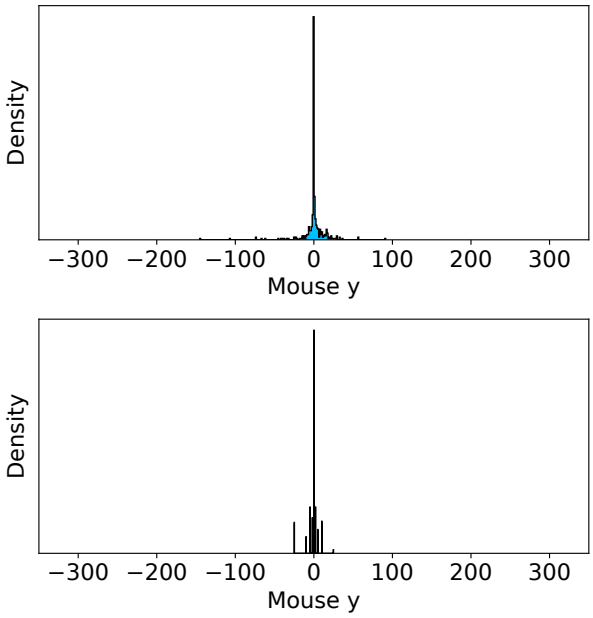


Fig. 5: Vertical movement distribution comparison between baseline (top) and 400 minutes of our augmented failure state recovery data (bottom).

REFERENCES

[1] T. Pearce and J. Zhu, "Counter-strike deathmatch with large-scale behavioural cloning," in *2022 IEEE Conference on Games (CoG)*, pp. 104–111, 2022.

[2] M. Campbell, A. J. Hoane Jr, and F.-h. Hsu, "Deep blue," *Artificial intelligence*, vol. 134, no. 1-2, pp. 57–83, 2002.

[3] C. Berner, G. Brockman, B. Chan, V. Cheung, P. Dębiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, *et al.*, "Dota 2 with large scale deep reinforcement learning," *arXiv preprint arXiv:1912.06680*, 2019.

[4] K. Fragkiadaki, "Deep reinforcement learning and control: Imitation learning." https://www.andrew.cmu.edu/course/10-403/slides/S19_lecture2_behaviorcloning.pdf, 2019. Accessed: 2023–05-17.

[5] J. A. Bagnell, "An invitation to imitation," tech. rep., Carnegie-Mellon Univ Pittsburgh Pa Robotics Inst, 2015.

[6] T. Hester, M. Vecerik, O. Pietquin, M. Lanctot, T. Schaul, B. Piot, D. Horgan, J. Quan, A. Sendonaris, I. Osband, *et al.*, "Deep q-learning from demonstrations," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 32, 2018.

[7] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, *et al.*, "End to end learning for self-driving cars," *arXiv preprint arXiv:1604.07316*, 2016.

[8] A. Tampuu, T. Matiisen, M. Semikin, D. Fishman, and N. Muhammad, "A survey of end-to-end driving: Architectures and training methods," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, no. 4, pp. 1364–1384, 2020.

[9] J. Xu, H. Guo, and S. Wu, "Indoor multi-sensory self-supervised autonomous mobile robotic navigation," in *2018 IEEE international conference on industrial internet (ICII)*, pp. 119–128, IEEE, 2018.

[10] Y. Ying, "Mastering first-person shooter game with imitation learning,"

in *2022 2nd International Conference on Networking, Communications and Information Technology (NetCIT)*, pp. 587–591, 2022.

[11] C. Thurau, C. Bauckhage, and G. Sagerer, "Imitation learning at all levels of game-ai," in *Proceedings of the international conference on computer games, artificial intelligence, design and education*, vol. 5, 2004.

[12] OpenAI, :, C. Berner, G. Brockman, B. Chan, V. Cheung, P. Dębiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, R. Józefowicz, S. Gray, C. Olsson, J. Pachocki, M. Petrov, H. P. d. O. Pinto, J. Raiman, T. Salimans, J. Schlatter, J. Schneider, S. Sidor, I. Sutskever, J. Tang, F. Wolski, and S. Zhang, "Dota 2 with large scale deep reinforcement learning," 2019.

[13] Z. Lin, J. Li, J. Shi, D. Ye, Q. Fu, and W. Yang, "Juewu-mc: Playing minecraft with sample-efficient hierarchical reinforcement learning," in *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22* (L. D. Raedt, ed.), pp. 3257–3263, International Joint Conferences on Artificial Intelligence Organization, 7 2022. Main Track.

[14] Y. Gao, B. Shi, X. Du, L. Wang, G. Chen, Z. Lian, F. Qiu, G. Han, W. Wang, D. Ye, *et al.*, "Learning diverse policies in moba games via macro-goals," *Advances in Neural Information Processing Systems*, vol. 34, pp. 16171–16182, 2021.

[15] W. Davies and P. Edwards, "Dagger: A new approach to combining multiple models learned from disjoint subsets," *machine Learning*, vol. 2000, pp. 1–16, 2000.

[16] M. Kelly, C. Sidrane, K. Driggs-Campbell, and M. J. Kochenderfer, "Hg-dagger: Interactive imitation learning with human experts," 2019.

[17] J. Roy, R. Girgis, J. Romoff, P.-L. Bacon, and C. Pal, "Direct behavior specification via constrained reinforcement learning," *arXiv preprint arXiv:2112.12228*, 2021.