Mitigating Cowardice for Reinforcement Learning Agents in Combat Scenarios

Steve Bakos
Ontario Tech University
Oshawa, Canada
steven.bakos@ontariotechu.net

Heidar Davoudi
Ontario Tech University
Oshawa, Canada
heidar.davoudi@ontariotechu.ca

***Abstract*—A common approach in reinforcement learning (RL) is to give the agent a static reward for successfully completing the task or punishing it for failing. However, this approach leads to a behaviour similar to fear in combat scenarios. It learns a sub-optimal policy improving over time while retaining elements of cowardice in updating the policy. Cowardice can be avoided by removing static rewards given to the agent at the terminal state, but this lack of reward can negatively affect performance. This paper presents a novel approach to solve these issues by decaying this reward or punishment based on the agent's performance at the terminal state and evaluates the proposed method across three separate games of varying levels of complexity—The Legend of Zelda, Megaman X, and M.U.G.E.N. All three games are based on combat scenarios where the goal is to defeat the opponent by reducing its health to zero. In all environments, the agents receiving decayed reward and punishment are more stable when training, achieve higher win rates, and require fewer actions per game than their statically rewarded counterparts.**

***Index Terms*—reinforcement learning, fighting game AI, retro game AI, reward shaping**

I. Introduction

The goal of a reinforcement learning (RL) agent in its environment is to maximize its reward. This typically involves giving the agent small amounts of reward or punishment during the episode to encourage a desired behaviour, ultimately leading to a goal. In combat scenarios, the goal is to defeat an enemy in a fight—the agent must reduce its opponent's health to zero by inflicting damage. At the terminal state, when either the agent or its opponent has died, the agent receives a reward or punishment for success or failure.

We define *cowardice* to be a situation where an RL agent avoids its opponent's actions to an exaggerated degree. Running and hiding unnecessarily are examples of cowardice. This differs from the intended behaviour of dealing with the opponent using the fewest actions possible while maximizing reward. Although some environments may require the agent to run and hide, they may benefit from discarding these behaviours as quickly as possible when not needed.

When looking at the average number of timesteps per game for an agent or duration of an episode, cowardice can cause large spikes, as the agent develops a sub-optimal strategy out of fearing the punishment from dying. Depending upon the complexity of the environment, a strategy developed via cowardice can cause effects similar to catastrophic forgetting as the agent's strategy no longer works and it must develop a new one. If the agent learns to lessen the fear of punishment, cowardice doesn't manifest in its play style and it can more rapidly master its environment.

We propose a method that the punishment and reward decays based on a measure of the agent's performance. If the agent was close to winning but still lost, its punishment is less than if it was nowhere near winning. Similarly, reward decays based on the agent's performance. If the agent wins, how "good" was that win? Depending upon our definition of mastery (i.e., the parameter controlling the reward decay), wins may not be equal. An aggressive agent performs well if it ends a match in 5 seconds instead of 20. In the first case, the agent gets a large reward at the terminal state, while it gets a smaller reward in the second. If the agent is defensive, then ending the match with full life yields a larger reward at the terminal state than ending it with half of its life remaining.

In the combat scenarios we test these agents in, the goal is to win. Because this requires the agent depleting its opponent's health, using this as the parameter to decay punishment means not all losses are equal. Regardless of the reward parameter used to define mastery, the agent receives a smaller punishment if the opponent has less health remaining at the terminal state than if it had more health. This decay of punishment reduces cowardice and encourages the agent to fight through its losses.

The common alternative to this approach is to give the agent a static reward. Regardless of the state at which the game ends, the agent receives a reward (e.g., +10) for winning or a reward (e.g., -10) for losing. This lack of distinction between one win from another or one loss from another is what our proposed method addresses. It's in this lack of distinction of one loss to another that cowardice emerges as the agent only learns to fear loss for its punishment, not that it can lessen that punishment by getting closer to achieving a win.

The summary of contributions of the paper is as follows:

- We show when applying the predominate method of rewarding success and punishing failure via static rewards, the agent can struggle to learn and master its environment depending upon its complexity, leading to cowardice.
- We demonstrate that reducing this fear by decaying the punishment the agent receives at the terminal state leads to an increase in the agent's performance, stability, reduction in training time.

- We use three separate environments to compare and evaluate the effects of giving the agent a static reward to decaying this value by some measure of the agent's performance.

This paper's remaining order is section 2 discussing related work for applying RL to retro and fighting games, as well as agents developing cowardice and overcoming it. Section 3 details the proposed method, applying it to each environment, as well as the environment specifics. Section 4 discusses the reasoning for choosing Proximal Policy Optimization (PPO) [1] for training the agents. Section 5 discusses network architectures used. Section 6 evaluates the results and discusses findings, and section 7 draws conclusions from the findings.

II. Related Work

Marwala and Hurwitz [2] trained agents to play Lerpa. Their agent AIden became afraid of punishment, as during the initial stages of its training, it lost a few times and decided it was better not to play at all. They solve this by forcing the agents to play the first 200 hands they receive regardless of fear. In The Legend of Zelda, Megaman X, and M.U.G.E.N—not playing isn't an option. The environments force the agents to fight against their opponents. The opponents within our environments are not as dynamic as [2]'s as they're not learning agents—their scripted behaviour contains varying degrees of randomness, but they do not actively develop new strategies and behaviours.

Fighting games are an application for RL which commonly use static rewards as its terminal states. Oh et al. [3], Kim, Park, and Yang [4], Mendonça, Bernardino, and Neto [5] are examples of using this technique—the agent wins, it receives a static positive reward, and a static negative reward for losing. Agents trained in [4] without a reward at the terminal state are much more volatile than those that have it, leading to instability and frequent fluctuations in performance.

Both [3] and [4] influence their agents' behaviours through interim punishments. A constant ticking punishment in [4] encourages aggression in their agent, which results in the average elapsed time per episode decreasing significantly compared to their other agents which don't use this punishment. In [3], a variation of this idea creates aggressive, balanced, and defensive archetypes. They achieve this via interim punishments for a time penalty, HP ratio reward/penalty depending upon if the agent has more health than its opponent, and a distance penalty. The weights of these features change depending upon the play style they're trying to encourage. While these punishments and rewards are useful in encouraging a desired behaviour, these agents all use static rewards for their terminal state rewards and likely suffered from cowardice at the start of their training and perhaps even still have elements of it in their behaviour.

In [6], agents trained on seven Atari 2600 games. For the sake of generalizability, the punishments and rewards are static and clipped to -1 and +1, respectively. The rewards are for interim results and do not reflect the overall goal of completing the task. As shown in [4], the lack of a terminal state reward for success or failure caused fluctuations in both win rate and the elapsed time in games. These fluctuations are present once again in [6]'s results for Breakout and Seaquest when looking at their average reward graphs. While Breakout has terminal states for winning and losing and would likely benefit from reward given at these states, Seaquest doesn't have one for winning, as the goal is to maximize score and this can go on indefinitely. It's likely that the instability present in their reward graphs can, in part, be attributed to this lack of a terminal state reward.

Another approach to address cowardice exists in Shao et al. [7]. Using StarCraft as the environment, [7] trains a model to handle controlling multiple agents in a variety of skirmishes. They reward agents for doing damage, while punishing agents for taking damage, dying, or moving away from other friendly units or enemies. Fig. 7 in [7]'s work shows, for the first of their experiments, the number of timesteps versus the number of episodes trained. Right away, a large spike in timesteps appears as the agents lose to their opponents and adopt a strategy of cowardice by running. [7] limits the number of actions the agents can take to 1000, which doesn't address the agents' cowardice directly, but limits its impact.

An alternative reward function in [7] shows the results of training these agents without punishing the agents for moving away from each other or their opponents. [7]'s Fig. 6 shows these results and shows that neither group of agents wins until roughly 1400 episodes. Cowardice is on the decline at this point as the number of timesteps per game is settling around the 400 mark. It's likely that a contributing factor to this behaviour is each agent receives a static punishment of -10 when it dies, which doesn't decay based on a result of the agent's performance. We address this issue via decaying the punishment the agent receives when it loses via a metric of its performance.

III. Proposed Method

Within the observation space of environments, the agent observes parameters at each timestep to decide its next action. These parameters can be things such as the agent's health, the opponent's health, match time remaining, and many more. These parameters play a vital role in defining failure, success, and mastery. When the agent reaches zero health, it's failed. The opponent reaching zero health means the agent succeeded. Mastery occurs when the agent succeeds and receives as little punishment as possible.

In defeat, the agent receives punishment for its failure to encourage a better choice of actions at the various states of gameplay it saw during the episode. Over time, its decision making improves and it can eventually learn to master its environment. However, when defeated, the agent often made correct decisions, but not enough of them to secure victory. Instead of simply punishing the agent for failing, regardless of how many correct decisions it made, we propose decaying this punishment by a measure of its performance.

In The Legend of Zelda, Megaman X, and M.U.G.E.N, the parameter in the observation space that is the difference be-

tween failure and success is the opponent's remaining health. At the agent's defeat, the opponent's remaining health reflects the agent's correct decision making. If the opponent has full health, the agent made no correct decisions. However, if the opponent's health is not full, the agent made at least one correct decision. Because the opponent's health is finite and known beforehand, we can use this as metric to judge how correct the agent's decision making and decay its punishment accordingly while it's attempting to transition from failing its episodes to succeeding. This lets the agent understand one loss from another and determine which is better as the punishment at the end of the episode now adjusts instead of being constant. With this understanding of one loss being better from another based on the agent's performance, it now becomes less afraid to engage with the opponent.

To transition from success to mastery, we follow the same approach. At the agent's victory, the value of a parameter within the observation space we'll choose defines how well the agent understands not only defeating its opponent, but doing so convincingly, showing mastery. If we choose the agent's remaining health as the parameter, we can now say that an episode that ends in victory with the agent at full health is better than an episode where the agent ends with half of its health remaining. Choosing the time remaining in the match as this parameter, we now can say that an episode that ends quickly is better than one that doesn't. With this metric, we can now decay the reward the agent receives at the end of the episode to encourage a particular behaviour based on the parameter chosen. Choosing the agent's remaining health as this parameter, the agent's terminal reward now depends upon its health and will become influenced to avoid damage. Using the time remaining in the match, speed becomes important and the agent will race to end the match as fast as possible. These are examples of a defensive and aggressive behaviour.

To achieve this decay, we use (1). P represents the chosen parameter from the observation space (e.g., agent health remaining, opponent health remaining, match time remaining, etc.). P_{obs} is the value this parameter has at the terminal state and P_{max} is the maximum value this parameter can take. V_{ter} is the static reward given at the terminal state, resulting in V_{net} given to the agent. Fig 1 shows this graphically. The punishment curve uses the negative branch of (1) while reward uses the positive branch. P_{obs} depends upon if the agent won or lost. If the agent loses, P_{obs} is the value at the time of defeat of the opponent's remaining health. If the agent wins, P_{obs} is the value at the time of victory of the chosen parameter to define mastery.

$$V_{net} = \pm V_{ter}^{\frac{P_{obs}}{P_{max}}} \tag{1}$$

A. Legend of Zelda

The Legend of Zelda, Fig. 2, is a 2D adventure game, which uses Open AI's Gym Retro [8] for its implementation, where the player controls Link in a grid world. The goal of the game is to explore the world and defeat enemies. Link mainly uses a sword, which, if he has full health, is a projectile. If he has sustained any damage, his sword loses this projectile functionality and becomes a close-range weapon. The agent controls Link and trains against the boss of the first dungeon. Cowardice is Link hiding in the doorway or behind blocks to avoid fighting the boss.

1) State: In The Legend of Zelda, the environment produces a downsized grey-scaled image of size 84x84. These images make a 4-frame stack with the newest being inserted at every timestep and the oldest removed [9]. The agent receives this stack as its observation. In this environment, the opponent is the boss of the first dungeon. It has limited movement and is only capable of back-and-forth motions in a restricted area while attacking Link with fireballs from a distance. These fireballs only travel in a single direction—away from the boss, but the boss' movement can be in either direction. Because of this, stacking these last 4 frames allows the agent to see in which direction the boss is moving, allowing it to approach the boss and attack from a distance without colliding with it and receiving damage.

2) Action: In this environment, the agent only has access to Link's sword, which functions as a projectile when Link has full health and a close-range weapon when not. Mapping the agent's actions to the buttons of the controller, it uses these to interact with the game. Up, Down, Left, Right are directional inputs which control the movement of the agent. The A action corresponds to the same button on the controller. This button uses Link's sword and makes the agent attack when selected. The agent also has the choice of not pressing any button. The No Op value gives this ability to the agent. Table I lists these actions. The agent must choose a single action from each row in the table, giving the agent a multi-discrete output. E.g., the agent may choose Up from the first row, Right from the second, and No Op from the third to move towards the upper right corner of the game in a single output.

3) Reward: The agent receives a reward of +1 for doing damage to the boss and punishment equal to the health it lost. The agent can get a punishment of -0.5 if hit by a fireball, or -1

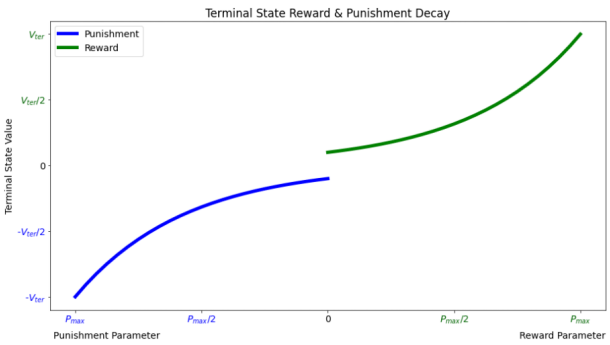

Fig. 1: V_{ter} is the value to be decayed. The reward and punishment parameters are chosen to encourage mastery and mitigate cowardice respectively. Their P_{max} values are the maximum value these parameters can take with P_{obs} being a position on the x-axis depending on win or loss.

Fig. 2: The Legend of Zelda, the first dungeon boss.

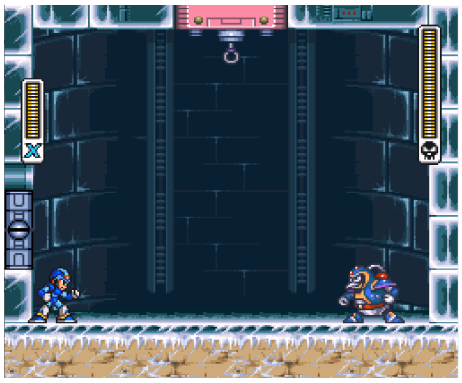

Fig. 3: Megaman X, X vs Chill Penguin.

if colliding with the boss. Closing the distance between itself and the boss rewards the agent with +0.001, while making this distance larger punishes it with -0.001. These distance rewards and punishments encourage Link to move towards the boss, which is required when the agent's sword is no longer a projectile.

There are three terminal states in this environment. The agent leaving the room results in a punishment of -10 as this is not a valid option for this scenario. Defeating the boss results in +10 given as a reward and dying to the boss gives -10 as punishment.

One agent receives these terminal state rewards as static values, while the other receives them decayed for dying to or defeating the boss. Using the boss' remaining health as the punishment parameter allows the agent to overcome its cowardice through learning that doing damage reduces its punishment at the terminal state. Reward decays via its remaining health as the reward parameter to encourage mastery, which leads to killing the boss while taking as minor damage as possible.

At the terminal states, we calculate reward and punishment using (1). For punishment, its value depends upon the boss' health remaining at the time the agent dies. The potential values are the punishment curve of Fig. 1. This lets the agent learn that doing damage to the boss will cause smaller punishment when it dies. Similarly, the reward value depends upon the agent's remaining health at the time the boss dies. The potential values of this reward is the reward curve in Fig. 1. With this, the agent can now tell one loss apart from another to understand dying while doing damage is better than dying without. The same applies to winning—the agent now can tell one win apart from another to understand winning with full health remaining is better than winning with health missing.

	Actions		
1	Up	Down	No Op
2	Left	Right	No Op
3	A		No Op

TABLE I: Link's Actions.

B. Megaman X

Megaman X, Fig. 3, is a 2D side-scrolling shoot 'em up action game, which uses [8] for its implementation, where the player controls X. The goals are to explore and complete levels which requires combat. The player character, X, attacks his opponents with a projectile he always has available. The agent controls X and fights against a boss during one of the earlier stages of the game. The agent must learn to avoid its attacks and successfully defeat the boss. Cowardice in this environment is hiding in the top corners of the screen to avoid the boss.

1) State: This environment follows the same approach as the Legend of Zelda's environment for its observations. Images are grey-scaled and downsized to 84x84 and used to make a 4-frame stack. Inserting the newest frame into the stack while removing the oldest—this stack becomes the observation. Since both the agent and the boss move frequently and at differing speeds, depending on which actions they take, the agent needs to be aware of the velocity of elements within the environment.

2) Action: In this environment, the agent has access to X's arm cannon to attack. This shoots projectiles and is always available to the agent. We map this attack to the controller's Y button and give the agent the ability to use it. X can jump using the controller's B button, as well as dash with the A button. We give these mappings to the agent and allow it to interact with the game through these. We map the controller's directional input buttons to Up, Down, Left, and Right. These actions allow the agent to move in those corresponding directions.

The agent must choose a single action from each row in Table II, giving the agent a multi-discrete output. E.g., the agent may choose Up from the first row, Right from the second, A from the third, B from the fourth, and Y from the last to jump dash to the right while attacking in a single output.

3) Reward: The agent receives +1 for every point of damage it does to the boss. Similarly, it receives -1 for every point of damage it takes. Both the agent and boss are capable of dealing multiple points of damage in a single hit. At the terminal state, the static reward variant receives +10 for winning or -10 for losing. The decay-based variant uses (1)

Fig. 4: M.U.G.E.N, a fighting-game engine made by Elecbyte.

to decay reward and punishment. Using the curves in Fig. 1 again, the reward parameter is X's remaining health and punishment parameter is the boss' remaining health.

C. M.U.G.E.N

M.U.G.E.N, Fig. 4, is a fighting-game engine made by Elecbyte. Implemented using Open AI Gym [10], the goal is to attack your opponent and reduces its health to zero. Like Fig. 3, the players are on a 2D stage. In this environment, hiding and inaction are not viable options. The agent must learn to avoid its opponent's attacks and deny it the ability to attack by controlling space and applying pressure. Characters can perform combos and special moves by inputting a sequence of actions within a strict time limit, resulting in doing significant damage to their opponent. Cowardice in this environment is constantly walking and running away from the opponent.

1) State: Unlike The Legend of Zelda and Megaman X, M.U.G.E.N doesn't use image data. Instead, we either calculate the observation space features or read them from game memory. The agent's last 15 actions are components of its observations, allowing it to perform combos and special moves. Table III shows features in the observation space related to the agent.

The opponent's features included in the observation space are in Table IV. Two distinct differences between this table and Table III is the lack of Last N Actions and Time-stop Meter. We prevented the agent from knowing its opponent's inputs, as this is akin to cheating and gives the agent an unfair advantage. Time-stop Meter is a feature unique to the agent's character. Table V holds the rest of the remaining features in the observation space. These are features that are calculated from the agent and opponent's features or are from the match itself.

2) Action: In this environment, the agent doesn't have a multi-discrete action space. Instead, it must choose one from a list of actions, as seen in Table VI. Down, Back, Forward, and Up are the directional inputs accepted by the game. Unlike Tables I and II, this agent has their Left and Right directional inputs replaced with Forward and Back.

In most fighting games, if a player presses Right on the controller, this action is a forward motion towards the opponent if the player is on the left side of the screen. If the player is on the right side of the screen, this is a backward motion moving away from the opponent. The Left button functions similarly.

In M.U.G.E.N, the characters have special abilities and combos which require directional inputs of Forward and Back, regardless of what side of the screen they're on. We simplify the action space by allowing the environment to handle whether Right or Left is Forward or Back. To do this, we compare the x-coordinates of the agent with its opponent's. If the agent's x-coordinate is smaller, then it's on the left side of the screen. If it's larger, then it's on the right side. This means that when the agent selects Forward as its directional input, we can know if the agent needs to move right or left towards its opponent, depending on which side of the screen it's on.

The remaining actions map to the various attacks the agent is capable of. X is a punch attack the agent has, while A, B, C are all different kicks. The agent can hold certain buttons listed in Table VI instead of press them. Within M.U.G.E.N, holding specific buttons performs certain actions. E.g., holding Back will block the opponent's attacks and deal reduced damage to the agent. We give the agent the option to hold these buttons instead of pressing them repeatedly because of the controller library used, and the specific requirements of the game. The

	Actions		
1	Up	Down	No Op
2	Left	Right	No Op
3	A		No Op
4	B		No Op
5	Y		No Op

TABLE II: Megaman X's Actions.

Feature	Norm. Value	True Value	Size
Health	0 ∼1	0 ∼1000	1
Meter	0 ∼1	0 ∼5000	1
Time-stop Meter	0 ∼1	0 ∼1000	1
State	0 ∼1	0 ∼5120	1
X Position	0 ∼1	-1350 ∼1350	1
Y Position	0 ∼1	-1000 ∼0	1
Is In Corner	0 or 1	0 or 1	1
Last N Actions	0 ∼1	0 ∼16	15

TABLE III: M.U.G.E.N Agent Features

Feature	Norm. Value	True Value
Health	0 ∼1	0 ∼1000
State	0 ∼1	0 ∼5120
X Position	0 ∼1	-1350 ∼1350
Y Position	0 ∼1	-1000 ∼0
Is In Corner	0 or 1	0 or 1

TABLE IV: M.U.G.E.N Opponent Features

Feature	Norm. Value	True Value
Relative X Distance	0 ∼1	0 ∼1160
Relative Y Distance	0 ∼1	0 ∼1160
Full Screen Apart	0 or 1	0 or 1
Match Time Remaining	0 ∼1	0 ∼5999

TABLE V: M.U.G.E.N Misc Features

X button gives the agent's character the ability to charge a resource meter, which allows it to teleport or perform some special moves. This resource builds while the agent is holding this button.

Buttons in the hold column of Table VI behave differently than those in the press column do. Pressing a button causes the agent to hold the button briefly before releasing it. This behaviour caused issues with the agent being able to block its opponent effectively, which we resolve through giving the agent the ability to hold a button. When the agent holds a button, it doesn't release it until the agent chooses a different action. For example, if the agent chooses hold back, the agent will hold this button. If the next action is again hold back, nothing will happen as the agent is already holding back. However, if the next action is to hold forward, we release the hold back command on the controller and execute hold forward. Similarly, if the agent chooses a hold command that isn't followed by that same action, we release the previous hold. We do this to prevent the controller from releasing the agent's block command early via repeatedly pressing back to block, which lets off the button at the end of the action.

We give combinations of directional input to the agent by combining directions together. In Table VI, these take the form of a direction + direction. We do this to further simplify the action space for the agent to perform special moves. E.g., the agent has a special move which shoots a rocket at its opponent. To do this, the agent must press in sequence and on a strict timer: Down, Down + Forward, Forward, C. M.U.G.E.N enforces this input sequence, as well as several others. These moves are essential in the agent defeating its opponent as they do significantly more damage than any of the standard attacks the agent is capable of.

3) Reward: Several agents trained in this environment with differing reward and punishment sources. Doing and receiving damage is common to all. A scaling factor of 0.0025 multiplies the health difference between the current state and the previous state. If the agent did damage, this as a reward. If it took damage, this is a punishment. An interim punishment source that one agent received mimics [3] and [4], where the agent receives constant ticking punishment to encourage aggressive behaviour and ending matches quickly.

Two agents trained receiving static rewards of +10 for winning and -10 for losing. Of these two, one agent also had the ticking time penalty. Three other agents trained and received this static reward decayed using (1). All three used the opponent's health remaining as the punishment parameter to discourage cowardice. The reward parameter was different

	Press	Hold	Misc
1	Down	Down	No Op
2	Back	Back	
3	Forward	Forward	
4	Up	X	
5	Down + Forward		
6	Down + Back		
7	Up + Forward		
8	Up + Back		
9	X		
10	A		
11	B		
12	C		

TABLE VI: M.U.G.E.N Actions.

across the three agents. One used the match time remaining, another used its remaining health, and the third used a simple average of the two. These parameters caused aggressive, defensive, and balanced play styles, respectively. Aggressive and defensive's reward and punishment curves mirror Fig. 1 while balanced uses Fig. 5 for its reward decay.

The balanced agent's reward decay is an average of the aggressive and defensive variant's rewards, which are calculated with match time remaining and agent health remaining as the reward parameters in Fig. 1, respectively. Fig. 5 shows the potential rewards the agent can receive depending on how the match ends.

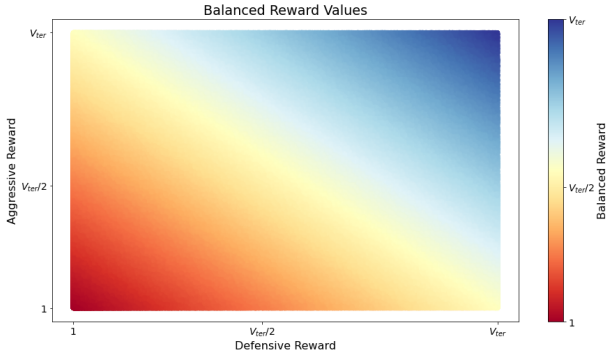

Fig. 5: V_{ter} is the static reward value to be decayed. At the terminal state, we calculate the aggressive (y-axis) and defensive (x-axis) variant's rewards using (1). The balanced decay variant's reward is then a simple average of these two reward values. This leads to multiple ways of obtaining the same value for balanced's terminal state reward.

IV. Reinforcement Learning Algorithm

This paper uses PPO [1] provided by Stable-Baselines3 [11] as the algorithm for training its agents. We chose PPO because of a few factors; all agents across all games have a discrete action space, which narrowed the choice to a Q-Learning algorithm, Advantage Actor Critic (A2C), or one of its derivatives. We required parallelization, as the agents needed to develop

their cowardice and then needed many games to resolve it to varying degrees. As recommended by [11], PPO and A2C are the best choices for discrete actions spaces and parallelized environments. Kim, Park, and Yang [4] used PPO as well for their agents in the FightingICE environment and one of the static valued variants trained in the M.U.G.E.N environment is based on their reward function. Also, the recent use of PPO by Open AI to beat world champions at Dota 2 [12] showed its potential to solve complicated tasks. These factors led to our use of it for these environments.

Table VII shows the hyper parameters used across all 3 environments. During the initial stages of development, we found that the PPO's built-in clipping wasn't enough to create stability in M.U.G.E.N agents. We further restricted the policy updates the agent could perform via a target KL of 0.03.

V. Network Architecture

A. The Legend of Zelda and Megaman X

The network architecture for both environments is the same. Using the defaults provided by [11] for the 'CnnPolicy', the architecture is that found in [9]—three convolutional layers with ReLU activation functions. Flattening this output, it then passes into a fully connected layer of 512 nodes.

B. M.U.G.E.N

The agent in the M.U.G.E.N environment uses the defaults provided by [11] for their 'MlpPolicy', with the architecture from [4]. This is three fully connected layers with ReLU activation functions comprising 330, 330, and 165 nodes. This then splits into the actor and critic branches with 165 and 80 nodes, respectively.

VI. Results and Discussion

We measure all agents across all environments based on the averaged results of a game-set where the set comprises games across all parallelized environments that share the same index (i.e., it averages the Nth game across all environments.) The Legend of Zelda's data is an average of 12 environments, Megaman X's is 32, and M.U.G.E.N's is 4. We ran each agent in The Legend of Zelda environment for 3 days, Megaman X for 4 days, and M.U.G.E.N. for 14 days.

The metrics we judge the agents on are the number of timesteps in a set and win rates in a pre-defined set of games. Both Megaman and M.U.G.E.N use 1024 games for this, while The Legend of Zelda uses a set of 300 games. We show these results in Fig. 6.

Hyper Parameter	Value
Learning Rate	1e-4
N Steps	2048
Batch Size	64
Gamma	0.99
Target KL	0.03

TABLE VII: Agent Hyper Parameters

A. The Legend of Zelda

In Fig. 6, we see both agents figure out the environment quickly. Decay learns how to deal with the boss much faster than static, but both ultimately defeat their opponent and show mastery. The notable findings in this environment are that the static variant has a large spike in the number of timesteps at the start of its training, while decay's spike is much smaller. Decay's win rate also grows much faster than statics's as it requires far fewer actions per game, showing that it isn't afraid to engage with the boss. Static shows cowardice via the much higher number of timesteps required, and this causes its slow increase in win rate compared to the decay-based variant. At the end of their training, decay achieved a best win rate of 99.7% while static achieved a best win rate of 98.0%.

B. Megaman X

The agents take a little longer than in the last environment to understand this encounter, as the boss' attack patterns have some randomness to them. Fig. 6 shows that cowardice is present once again in the static variant as it requires significantly more actions per game to end its matches. Cowardice leads the static agent to develop a strategy in which it hides in the top corners of the screen and attacks the boss only when given the chance. This works early on, but it's far from the optimal way to fight this boss. Decay engages with the boss much more frequently and doesn't develop this behaviour of hiding in the top corners of the screen. Instead, it learns to dodge attacks and jump over the boss to give itself distance. Mastering this behaviour leads to decay overtaking static's win rate and showing an upward trend even at the end of its training. Decay's win rate doesn't grow as fast as static's at first because it puts itself in more danger than static, but it overtakes static around the 15th set and outperforms static by a good margin. At their bests, decay achieved a 95.5% win rate while static achieved an 84.8% win rate.

C. M.U.G.E.N

Three decay-based agents trained in this environment. All used the opponent's remaining health as their punishment parameter to control its decay. Aggressive used match time remaining and defensive used the agent's health remaining as their reward parameters. Balanced used Fig. 5 to determine its terminal state rewards. Of the two static variants, one received a constant ticking time penalty to encourage the agent to become aggressive. This agent's reward function is based on [4]'s work, while the other is the same as the static variants for The Legend of Zelda and Megaman X.

In Fig. 6, cowardice is visible in both static variants as the number of timesteps per game set fluctuates wildly, while all decay-based variants steadily decrease their action count per game set. An interesting observation is that the aggressive and defensive decay variants both achieve similar results even though the defensive variant doesn't use time to influence its reward and developed a different play style.

Cowardice negatively affects the static variants as their win rates fluctuate throughout their training. The opponent in this

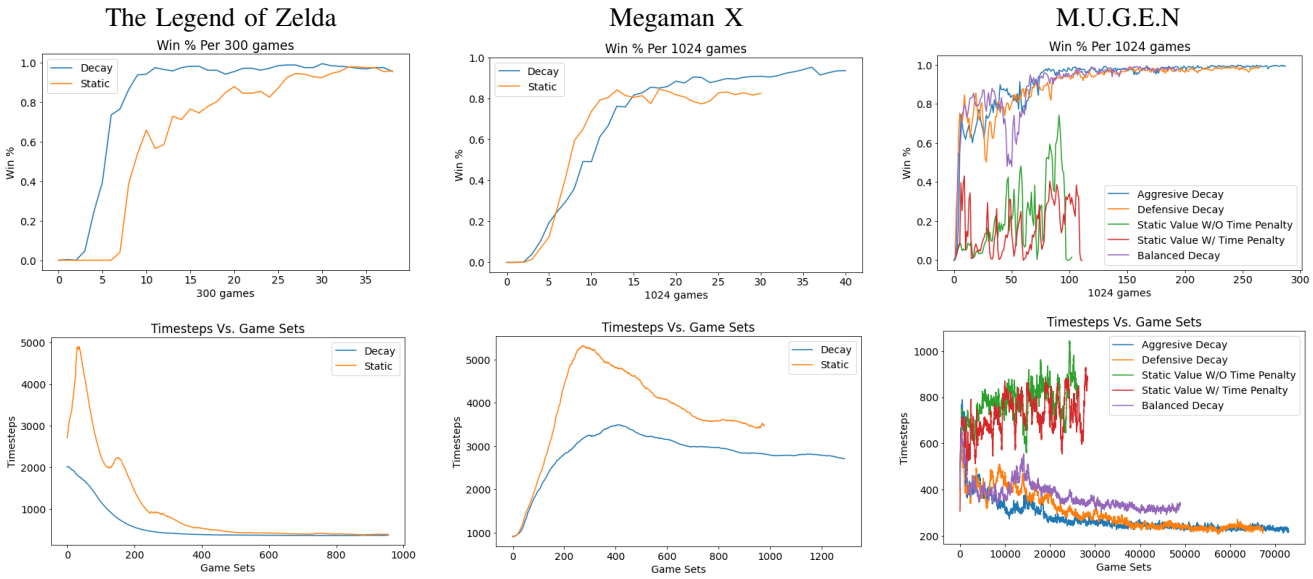

Fig. 6: Results for win rates and timestep counts across The Legend of Zelda, Megaman X, and M.U.G.E.N environments.

environment has multiple strategies for attacking the agent and simply running away won't work. Both static variants developed this strategy and attacked the opponent only when it presented an opportunity instead of making their own. Decay-based variants applied pressure and forced openings in the opponent's strategy, which put it on the defensive. Fig. 6 shows this with the decreasing actions per game set count. The best win rates achieved by these agents are 99.9% for Aggressive Decay, 99.2% for Balanced Decay, 99.0% for Defensive Decay, 74.2% for Static Value W/O Time Penalty, and 43.2% for Static Value W/ Time Penalty.

VII. Conclusion

Cowardice develops when an agent fears dying—it takes every action, many of them unnecessary, it can to avoid this. This fear causes it to develop strategies that can work, but do not scale well with the complexity of the environment. Moreso, in environments where cowardice isn't fatal to the agent's performance, removing its hinderance via decaying the terminal state rewards leads to better performance in terms of win rates, average number of timesteps, and stability. This approach also provides a method of defining what mastery means for each agent—allowing creating different play styles for agents through decaying reward via different parameters or combinations.

References

[1] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017.

[2] T. Marwala and E. Hurwitz, *A Multi-Agent Approach to Bluffing*, 01 2009.

[3] I. Oh, S. Rho, S. Moon, S. Son, H. Lee, and J. Chung, "Creating pro-level ai for a real-time fighting game using deep reinforcement learning," 2020.

[4] D.-W. Kim, S. Park, and S.-i. Yang, "Mastering fighting game using deep reinforcement learning with self-play," in *2020 IEEE Conference on Games (CoG)*, 2020, pp. 576–583.

[5] M. Mendonça, H. Bernardino, and R. Fonseca Neto, "Simulating human behavior in fighting games using reinforcement learning and artificial neural networks," 11 2015.

[6] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," 2013.

[7] K. Shao, Y. Zhu, and D. Zhao, "Starcraft micromanagement with reinforcement learning and curriculum transfer learning," 2018. [Online]. Available: https://arxiv.org/abs/1804.00810

[8] A. Nichol, V. Pfau, C. Hesse, O. Klimov, and J. Schulman, "Gotta learn fast: A new benchmark for generalization in rl," *arXiv preprint arXiv:1804.03720*, 2018.

[9] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015. [Online]. Available: http://dx.doi.org/10.1038/nature14236

[10] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[11] A. Raffin, A. Hill, M. Ernestus, A. Gleave, A. Kanervisto, and N. Dormann, "Stable baselines3," https://github.com/DLR-RM/stable-baselines3, 2019.

[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.