Deadzone: DQN Model on Top-Down Shooter

Akhilesh Neeruganti, Leslie Liu, Hrishikesh Prashant Salway
North Carolina State University

Abstract

In this paper, we will apply an advanced reinforcement learning model on a top-down shooter game. Specifically, we will use a Deep Q-learning model to train our model based on rewards and punishments. We will implement different rewards and punishments to try and simulate different behaviors to determine which rewards system was best.

Introduction

The goal of this paper is to determine how effective reinforcement learning is in a top-down shooter. In order to do this, we decided to experiment with different reward systems and different weights in order accomplish this. This includes different weights into shooting, moving, rotating, etc. Our hypothesis is that rewards that focus on moving and rotating will take precedence as aiming is very important in order to aim at the opponent.

For our model, we want to focus on three aspects: adaptability, learning behavior, and reward optimization. First, we want to see how our reinforcement model can work in different map environments and different player strategies. Second, we want to see what impact the learning algorithm made and how it potentially created a more challenging and engaging opponent for human players. Finally, we want to observe how our model reacts with different reward optimization and determine what is effective and what is detrimental to the AI.

For the specific reinforcement learning algorithm, we decided Deep Q-Learning (DQN) algorithm would be best as it would combine the rewards system with a learning algorithm. This specific algorithm has the ability to remember past states with its memory. On top of that, it can handle more complex state spaces where the size is large or continuous. Finally, there are many examples of this working in games such as DQN being used on Atari games [1].

We believe that this will be a very useful tool to create more sophisticated game AI that sufficiently engages the player with strategies not typically found in NPC behaviors. This reinforcement model can be slightly modified to handle most top-down shooters and even some other games that have top-down movement similar to our game. With this, future developers can work with an AI that is more responsive and robust to the environment. On top of this, this can help with academic research in adaptive systems and reinforcement learning while offering insights well beyond the realm of gaming. While this is a reinforcement learning algorithm, not every game needs it as a simpler model can be sufficient. That is why this paper will implement a DQN model that will use rewards to help facilitate specific behaviors.

Our game will be placed in an enclosed map with walls on all sides. There will be walls in the middle that will act as obstacles that players can use to block bullets. We want the AI model to be forced to move and strategize around these obstacles to successfully win against the opponent. These maps are created with tiles and can be customized to suit a variety of different environments. An example environment is shown in Figure 1.

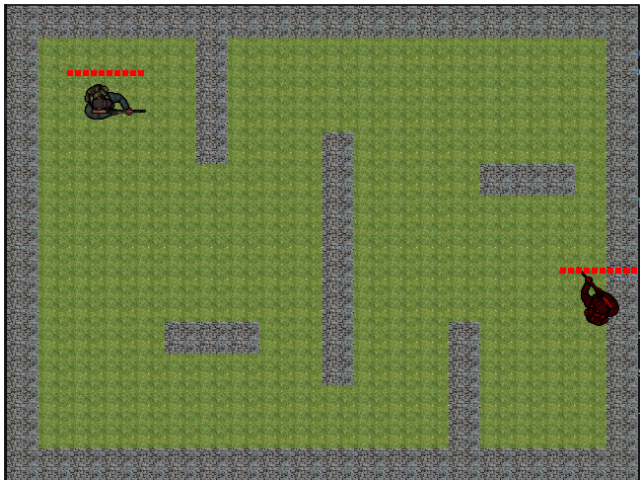

Figure 1: Map Environment

For the players in the game, they will start with ten health. They can freely move around the map in any direction. These players can change their rotation in angular increments of 15 degrees. In terms of their weapons, they start with an assault rifle and a shotgun. The rifle has a faster fire rate and longer range than a shotgun. To compensate for this,

the shotgun does damage three times as much as the rifle. The general goal of the game is to kill the opposing player and be the last man standing. Figure 2 and 3 shows the player using a rifle and a shotgun respectively.

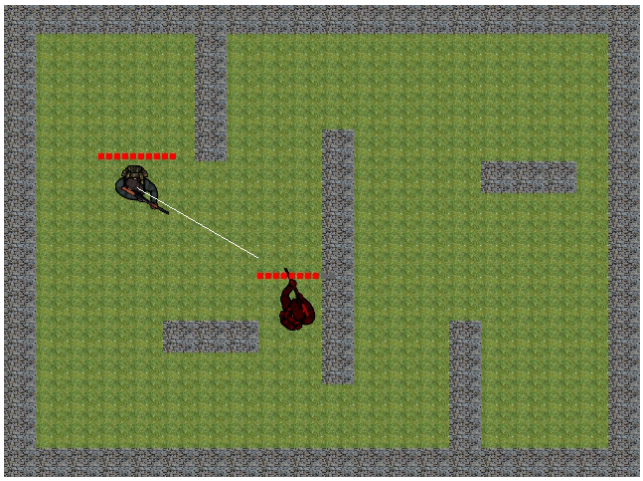

Figure 2: Character with Rifle

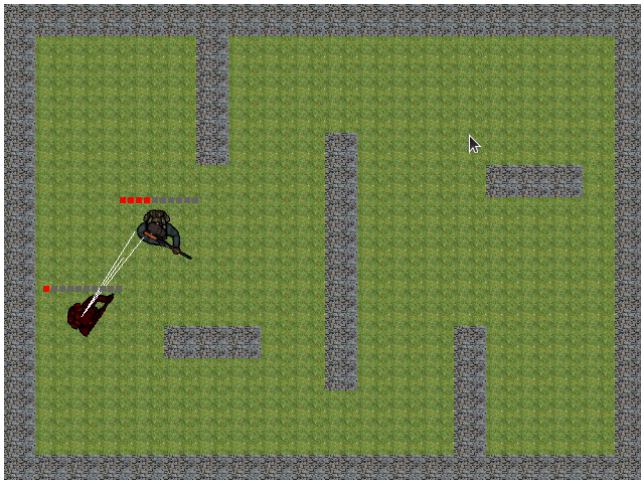

Figure 3: Character with Shotgun

In terms of implementation, our game will run in a C++ environment with the Simple and Fast Multimedia Library (SFML). This can quickly provide a simple game window that updates every frame. For our model, we plan to use Python with the PyTorch library. To connect these two, we plan to simulate the game in Python and train the model on the simulated game. Then we can use sockets to connect our SFML game to the model to predict actions.

Literature Review

Recent developments in deep reinforcement learning have shown the promise of Deep Q-Networks (DQN) in complex environments.

Souchleris et al. shows how reinforcement learning is used in gaming.[7] It explains the basics of reinforcement learning and the fact that Deep Q-Network is a popular algorithm for reinforcement learning in games. This algorithm can make use of rewards and punishments in order to create intelligent agents. This paper also proposes that Double Q-Learning is a viable option, though it may not work as well in stochastic environments with a degree of randomness.

Mnih et al.'s work[2] is widely recognized as a breakthrough in applying deep reinforcement learning to complex environments. In their research, they introduced the Deep Q-Network (DQN) which enabled an agent to learn directly from high-dimensional sensory inputs (e.g., raw pixel data) to achieve human-level performance on several Atari games. This work laid the foundation for integrating deep learning with Q-learning, allowing the algorithm to approximate the optimal action-value function using a convolutional neural network.

Hasselt et al. extended the original DQN framework by introducing Double Q-learning to tackle the overestimation bias often encountered in Q-value estimates.[3] This modification allows the learning process to yield more accurate value approximations, which is particularly crucial in environments where decisions must be made with high precision. The Double DQN approach decouples the action selection from the target Q-value estimation, resulting in improved stability and performance during training.

Wang et al. also proposed the Dueling Network Architecture as an enhancement over the traditional DQN.[4] This architecture separates the estimation of the state value and the advantage for each action, which allows the network to learn which states are valuable without having to learn the effect of each action for every state. This separation can lead to improved performance, particularly in scenarios where the choice of action is not critical for every state.

Resceanu et al. shows a study on using Deep Q-Learning to create intelligent virtual characters that can fulfill a task similar to humans.[5] This paper shows that the character can remember which tasks was rewarded and create a "habit" similar to human psychology. This idea can be used to create agents that will prefer specific actions that leads to better agents.

Hester et al. presents a way to perform Deep Q-Learning in an effective manner without relying on large amounts of data.[6] This paper presents a way to use demonstrations to get better performance. The agent will train on a small set of demonstration data to learn from before training on the actual data. This allows the learning agent to quickly train the basics of what it needs to do before learning on larger sets of data.

Roderick et al. shows how to implement Deep Q-Network and how to optimize the model for better results.[8] This paper presents algorithms for DQN and how gradient decent can be used to optimize the parameters. On top of that, this paper delves into the fact that DQN can fluctuate in terms of performance as Q-Learning is an unstable algorithm over a large state space. Because of this, maintaining state space and trying to optimize parameters are very important in order to create a good DQN model.

As our AI agents operate in a 2D shooter with fast-paced decision-making, the ability to accurately estimate the Q-values can greatly enhance the agent's performance. A more stable learning process is very important for RL that can more reliably adapt to the game dynamics.

Implementation

In this project, we build a 2D environment by a tile-based system, allowing for straightforward collision checks and flexible level design. Each tile is represented by a texture (e.g., grass.jpg or stone.jpg) and can be marked as either passable or not. The map is composed of multiple layers with a background layer (for terrain) and an obstacle layer (for walls, barriers, etc.). This multi-layer approach ensures a clear separation of concerns between visual rendering and collision logic.

Environment Implementation

- The background layer fills the entire map with a passable texture (grass).
- The obstacle layer (stone) places around the map boundaries and in strategic positions, forcing players to navigate around them.
- Additional layers can be added for decorative elements or special features (e.g., boxes, triggers).
- A simple point-based collision detection is used: when a character moves, the system checks if the target position intersects with any collidable tile. If so, the character cannot proceed in that direction.

Character Implementation

Each character is represented by a sprite with a specific texture (survivor holding a rifle or shotgun). The sprite's origin is set to the center of the texture, ensuring consistent rotation and collision checks. Additionally, each character has a health bar displayed above the sprite, indicating its remaining health points (HP).

Movement and Rotation

- **Movement:**
  - Characters can move up, down, left, and right using keyboard inputs.
  - A simple collision check is performed before finalizing any movement to ensure that the new position is passable.
- **Rotation:**
  - Characters can rotate in place, either via incremental keyboard commands (e.g., pressing a key to rotate by 15 degrees) or by orienting toward a target when shooting.
  - This rotation is applied to the sprite, which updates the character's facing direction.

Weapon and Health System

- Characters can switch between different weapons, such as a rifle or shotgun.
- Each weapon texture is scaled to maintain a consistent on-screen size, and the character's origin is recalculated whenever a new weapon texture is set.
- A 10-HP system is employed, with each HP represented by a small red rectangle displayed above the character sprite.
- When the character is get hit by rifle, the health will reduced by one, and get hit by shotgun will reduce the health by three.
- Rifles have a range of 300 pixels and shotgun have a range of 100 pixels.
- Rifles have a fire rate of 1 shot every 400 frames. Shotguns is every 800 frames.
- To stop a bullet at another player or obstacle, we iterate across the line until a player or obstacle is found and stop the line at that point.
- If HP reaches zero, the character will be removed from the game.

Methodology

Environment Setup

Before working on building a Machine Learning Model, the input needs to be properly formatted. In this use case, as the game is a top-down shooter game, the whole environment needs to be taken into account for the player to make an educated decision. The training environment is of a fixed size and the obstacles are also present at fixed locations. To make the environment less complex for the model, it is represented as a grid (matrix). Each unit in the grid will be classified either as an obstacle (player or bullets cannot pass through) or open. Multiple obstacle units are used to simulate walls or bigger obstacles in the environment. The players can only move from tile to another (unit) i.e., the movement of the players are not continuous. This presents us with a problem of rough movement. To address this, the tile (unit) size in the grid is going to be small so as to simulate a continuous motion. The player rotates in fixed increments of 5 degrees, and bullets can hit the opponent when the player is aligned in the same direction as the opponent, with a margin of error of ±5 degrees. An example grid representation of the environment is given in Figure 4.

0	0	0	1	1	1	0	0	0	0
0	1	1	1	0	1	0	1	0	0
0	0	0	0	0	0	0	1	1	0
0	1	1	0	1	1	1	1	0	0
0	1	0	0	1	1	1	1	0	0
0	0	0	0	0	0	0	0	0	0
0	1	1	0	1	0	0	0	1	1
0	1	1	0	1	1	0	0	0	1
0	1	1	0	1	1	1	0	1	1
0	0	0	0	1	1	1	0	0	0

Figure 4: Map representation in form of grid

Training Model

The AI has a discrete action space:

- **Movement**: Up, down, left, right, top right, top left, bottom right and bottom left
- **Turn**: left and right (5 degrees)
- **Switch guns**: Rifle or shotgun
- **Shoot**: Shoot the opponent
- **Nothing**: Do nothing

The AI will be guided by following rewards system:

- **Hit opponent:** AI should successfully hit opponent
- **Moving closer to the opponent:** This will encourage the AI to move closer to its opponent to engage in battle.
- **Hiding behind obstacles when health is low:** The AI should keep hiding until it regains its health.
- **Killing an opponent:** AI should win
- **Dying:** AI should not lose.
- **Getting hit:** The AI should try to to be in the firing range (both linear and rotational).
- **Using the wrong gun:** When the distance is high between the two AIs, rifle should be used (low damage, high range), otherwise shotgun (high damage, low range).
- **Being stationary:**
- **Hiding with high health:** This too will encourage the AI to engage in battles when it has enough health.
- **Being too far from the opponent with high health:** The AI should try to engage the opponent instead of running away.

Model Architecture

The core of the model is the employment of a Deep Q-Network (DQN) for learning Q-values for all the possible things the agent can do. The DQN helps the agent navigate the world and make the correct decision based on the total reward of the action. In this context, the state encompasses the spatial information and agent-specific information. Spatial information can be the relative positions of the obstacles and the opponent, and the agent-specific information can be the health of the player, the weapon, the position, and the direction. Representing such information as a structured state gives the model the ability to present the agent with all the context the agent would require for the decision. The DQN structure here is designed to accept the agent's state, the concatenation of the two-dimensional and numeric features, and spit out the predicted Q-values for all the available action. The structure can facilitate learning and adaptation of the agent effectively in the game world.

The input layer accepts a comprehensive state vector that encapsulates the agent's environment and current attributes. Overall, we store 14 features in each state. The features processed in this layer are as follows:

- Relative Obstacle Positions: The three closest obstacles are stored as vectors from the players position
- The Opponent Position w.r.t to the Player ((x, y)): The position of the opponent w.r.t the position of the player helps the agent to plan the combat and the move.
- Player Position ((x, y)): The position of the player around the environment allows determining of the next step, especially in path calculation and positioning.
- Weapon Type: A one-hot encoding of the current gun the player is using so the model can differentiate weapons from weapons as well as action/strategy planning.
- Player Health: this is scaled from 0 to 10. It tells the agent whether the player can survive, and whether to take defensive or assaultive action.
- Opponent Health: this can help the Agent to make more accurate decisions in terms of how to approach the opponent, specifically when it runs away and when it chases the opponent.
- Current Orientation: This can help the agent make decision on how much it needs to turn to aim at the opponent.

All of these aspects are compounded into a state vector, which is the input for the neural network.

The hidden layers are responsible for processing the input features and pulling out meaningful patterns to contribute to decision-making. The subsequent dense layers are employed:

1. First Dense Layer:
   - Contains 256 neurons with ReLU activation.
   - It processes high-dimensional input data, learning the interaction of agent-specific and spatial features.
2. Second Dense Layer:
   - Contains 256 neurons with ReLU activation.
   - It also generalizes the patterns and refines the feature representations of the last layer. It prepares the data for the output layer.

ReLU introduces non-linearity to allow the net to be capable of learning complex functions for the task of predicting good Q-values.

The output layer is estimating Q-values for every possible action from the agent's action space. All the Q-values are cumulative rewards expected for the respective action taken in the current state. The action space contains:

1. Movement Actions:
   - The agent can travel in any of the eight possible directions, namely, up, down, left, right, and diagonally.
   - They enable the agent to move through the grid world effectively.
2. Turning Actions:
   - The agent can rotate 5 degrees toward the right side or the left side.
   - It provides accurate alignment for firing and navigation.
3. Weapon Switching:
   - The agent is also capable of switching from available weapons. Switching is performed on the basis of strategic circumstances, for instance, switching to a powerful weapon for hand-to-hand combat.
4. Shooting:
   - The agent shoot in the direction they are facing.
   - This is how the agent effectively defeat the opponent.
5. Do Nothing:
   - The model does not perform any action.
   - This is helpful if the character wants to stay hidden behind obstacles.

Given these actions, the output layer consists of 13 neurons, where each neuron corresponds to one action in the action space. The Q-values produced by these neurons guide the agent in selecting the optimal action using an epsilon-greedy policy during training.

Training Strategy

To train the DQN effectively, there is a robust training mechanism that includes the employment of experience replay, inclusion of a target network, and the application of epsilon-greedy exploration. All of these mechanisms cooperate with each other for the effective learning of the dynamic world.

Replay buffer is critical for enhancing the efficiency of the model's samples. As the agent is playing the game, it saves its experience, such as the state at the time, the action it performed, the reward it got and the following state, in the buffer. Instead of learning from consecutive experience, which can be heavily correlated, the DQN samples randomly from the buffer mini-batches of experience. Randomization shatters the sequence of the samples such that the model generalizes and does not overfit the sequence of events. The buffer also provides the agent with the ability to learn from experience, which enhances the efficiency of the samples.

To stabilize training, the DQN employs the idea of the target network. It is a copy of the DQN and is updated less frequently than after each training step. By keeping the weights of the target network frozen for several iterations, the Q-values update is regularized such that the training model does not diverge and oscillate during training. It is programmed for the optimization of the difference of the Q-value predictions from it and the target Q-values from the target network.

The agent's action is guided by epsilon-greedy exploration strategy for the trade-off of exploration and exploitation. During the initial training, the exploration rate (ϵ) is given a high rate (i.e., ($\epsilon = 1$)), and this will cause the agent to choose randomly. The process of exploration will allow the agent to experience things differently and learn possible policies. As time goes on, (ϵ) is reduced and the agent is focused on the act of exploitation, whereby it chooses the action with the highest estimate of Q values. Gradually decreasing is such that the agent will explore the world entirely before it settles on a particular policy. The DQN will run 6000 episodes to try and tune the model, saving the model generated at each episode as a file.

Evaluation

In order to build an evaluation framework for our game. We will focus on several key performance criterias:

- **NumOfHits:** The number of successful shots.
- **NumOfMisses:** The number of unsuccessful shots.
- **EffectiveMoves:** The movement was not stuck on a wall.
- **StuckMoves:** The movement was stuck on a wall.
- **CloseToEnemy:** The player is close to the enemy position.
- **MoveTurnCombo:** Successful moves and turn in succession.
- **GoodTurns:** A rotation that turns towards the opponent position.
- **BadTurns:** A rotation that turns away from the opponent position.
- **CorrectSwitch:** Switch to a shotgun at close range or a rifle at long range.
- **WrongSwitch:** Switch to a shotgun at long range or a rifle at close range.
- **ActionDiversity:** Promote diverse action choices in the model.
- **RepeatCount:** Number of repeated actions.

Where each w variable are the weights for each

Two Phases of Evaluations

We will employ two distinct AI approaches for benchmarking: a reinforcement learning (RL) based agent vs player and the two agents against each other. The same metrics will be used in both scenarios.

Phase 1: Player vs. AI

- **Player vs. RL**: The human player will face an RL-based agent. This phase is critical for training the RL model in the early stage.

Phase 2: AI vs. AI

- **RL vs. RL**: The RL agent will compete against another AI agent, allowing us to measure improvements in the RL agent's strategy compared to the traditional algorithm. This will be the primary training phase.

Evaluation Metrics

We try different reward system and try to find the best policy to let the AI agent can learning the complex behavior

To capture the overall performance more holistically, we will combine these three indicators using a weighted sum. In this method, each metric is assigned a weight that reflects its relative importance:

$$\begin{aligned}\text{Score} = {} & w_{\text{hit}} \cdot \text{NumOfHits} + w_{\text{kill}} \cdot \text{KillCount} \\ & + w_{\text{combo}} \cdot \text{TurnHitCombo} - w_{\text{miss}} \cdot \text{NumOfMisses} \\ & + w_{\text{move}} \cdot \text{EffectiveMoves} - w_{\text{stuck}} \cdot \text{StuckMoves} \\ & + w_{\text{approach}} \cdot \text{CloseToEnemy} + w_{\text{m2t}} \cdot \text{MoveTurnCombo} \\ & + w_{\text{angle+}} \cdot \text{GoodTurns} - w_{\text{angle-}} \cdot \text{BadTurns} \\ & + w_{\text{switch+}} \cdot \text{CorrectSwitch} - w_{\text{switch-}} \cdot \text{WrongSwitch} \\ & + w_{\text{diverse}} \cdot \text{ActionDiversity} - w_{\text{repeat}} \cdot \max(0, \text{RepeatCount} - 2)\end{aligned}$$

Where w_h, w_k and w_m are the weights for number of hits, kill count, and survival time, respectively.
We plan to conduct experiments by varying these weights and observing how the overall performance scores correlate with qualitative game outcomes. Through systematic testing, we aim to identify the optimal weight configuration that best reflects effective play and that can be used as a benchmark for comparing the AI agents.

Along with this, we will manually observe the model at specific episodes and determine if training seems to yield results. We get values for number of hits, misses, and kills. We also maintain which actions were chosen and their counts. Finally, we store the total reward at each episode.

Result

In practice, we designed and tested two different reward systems to investigate how reward shaping influences agent behavior in a 2D shooter environment.

Attack-Focused Reward System

In our first version, the reward system strongly emphasized aggressive behavior. A large positive reward was given when the agent successfully hit the opponent, with an even larger bonus for achieving a kill. However, this often led to unintended behavior—agents would stay in place and repeatedly shoot, even when the opponent was not visible or within effective range. It is because other strategic actions like repositioning or aiming adjustments were not rewarded, the agent had no incentive to explore or adapt, resulting in static and suboptimal gameplay.

The reward system is as below:

- Shoot
  $R+ = 0.5$
- Successful hit (enemy HP drops)
  $R+ = 7$
- Kill (enemy HP reaches 0)
  $R+ = 10$
- Hit after turn (combo bonus)
  $R+ = 0$
- Missed shot while visible
  $R- = 5$
- Shot when enemy not visible
  $R- = 2$
- Effective movement (position changes)
  $R+ = 0$
- Ineffective movement (stuck)
  $R- = 1.5$
- Moved closer to visible enemy
  $R+ = 0$
- Move $\rightarrow$ Turn (if angle improves)
  $R+ = 0$
- Turn to explore when enemy not visible
  $R+ = 0$
- Turn with better angle to visible enemy
  $R+ = 0$
- Turn with worse angle to visible enemy
  $R- = 0.2$
- Correct weapon switch
  $R+ = 0$
- Move $\rightarrow$ Switch combo
  $R+ = 0$
- Incorrect weapon switch
  $R- = 0.5$
- Different action than previous
  $R+ = 0$
- Repeated same action more than 3 times
  $R+ = 0$

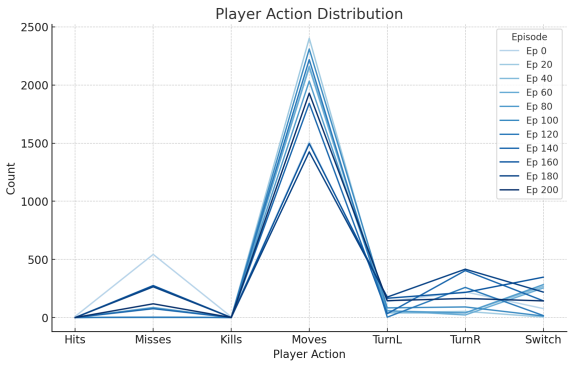

Figure 5: Action distribution of Attack-focused policy

From Fig 5, we can see the action distribution chart under the attack-focused reward system reveals that although shooting was highly rewarded, the agent shows very few successful hits and almost no kills, likely due to poor aiming and the lack of turning behavior, which was not sufficiently learned. Over time, "Misses" decrease, suggesting that the agent learns not to shoot at all to escape punishment. Movement like turning and weapon switching continues to be low, suggesting that in the absence of explicit rewards for such actions, the agent does not learn more advanced movements like aiming, positioning, or reacting to distance. Overall, the agent falls back on constant movement without any actual combat interaction, exposing the vulnerability of a reward system with only an attack outcome.

Balanced and Exploratory Reward System

To address the above problem, we introduced a more balanced reward function that encourages a variety of useful behaviors. The new system rewards exploration, action diversity, and tactical combinations:

1. **R**ewards are given for changing actions, such as movement, turning, and weapon switching.
2. **A**ction sequences like "move $\rightarrow$ turn $\rightarrow$ shoot" receive extra bonuses if they bring the agent closer to the opponent or improve shooting accuracy.
3. **C**hoosing the appropriate weapon based on distance and visibility is also rewarded.
4. **R**epetitive behavior, such as taking the same action multiple times in a row (e.g., walking into a wall or spamming shots), now receives increasing penalties.

The reward system is as below:

- Shoot
  $R+ = 0$
- Successful hit (enemy HP drops)
  $R+ = 5$
- Kill (enemy HP reaches 0)
  $R+ = 7$
- Hit after turn (combo bonus)
  $R+ = 2$
- Missed shot while visible
  $R- = 5$
- Shot when enemy not visible
  $R- = 2$
- Effective movement (position changes)
  $R+ = 1$
- Ineffective movement (stuck)
  $R- = 1.5$
- Moved closer to visible enemy
  $R+ = 0.5$
- Move $\rightarrow$ Turn (if angle improves)
  $R+ = 0.5$
- Turn to explore when enemy not visible
  $R+ = 0.3$
- Turn with better angle to visible enemy
  $R+ = 0.5$
- Turn with worse angle to visible enemy
  $R- = 0.2$
- Correct weapon switch
  $R+ = 2$
- Move $\rightarrow$ Switch combo
  $R+ = 1$
- Incorrect weapon switch
  $R- = 0.5$
- Different action than previous
  $R+ = 3$
- Repeated same action more than 3 times
  $R- = 0.5 \times (\texttt{same\_action\_run} - 2)$

This updated strategy has proven more effective. The agent exhibits more realistic and tactical behavior, such as approaching enemies, adjusting its aim, and timing attacks. We've also observed successful engagements resulting in confirmed kills, demonstrating that the agent is learning to coordinate its actions effectively.

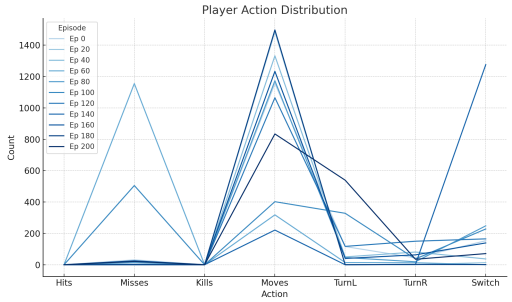

Figure 6: Action distribution of Balanced and Exploratory policy

From Fig 6, the action distribution under the balanced and exploratory reward system exhibits a much more diverse behavioral pattern compared to the attack-focused policy. While movement ("Moves") still dominates, its count is lower than before, indicating that movement is no longer the sole strategy. E.g., left or right turns ("TurnL" and "TurnR") actions increase gradually across the episodes, illustrating the agent is learning to regulate orientation as strategic behavior. "Switch" actions also increase in subsequent episodes, illustrating strategic weapon adaptation by distance or situation. Although "Hits" and "Kills" are low in magnitude in absolute terms, they follow a steep decline in "Misses" over time, illustrating improved decision-making before firing. As a whole, agent behaves more advanced and adaptive behavior combinations such as move-turn-shoot or move-switch combo, since the reward system motivates explanation and forestalls redundant or inefficient behavior.

The model starts with very high negative rewards as it performs poorly at the start of the training. The rewards collected by both the models increases gradually with each episode and sees a significant improvement after 4000 episodes. Figure 7 shows the total rewards collected by the models over 6000 episodes. By the end of training, the models achieved positive rewards consistently, indicating that they have learned an optimal policy to navigate through the environment and win the game.

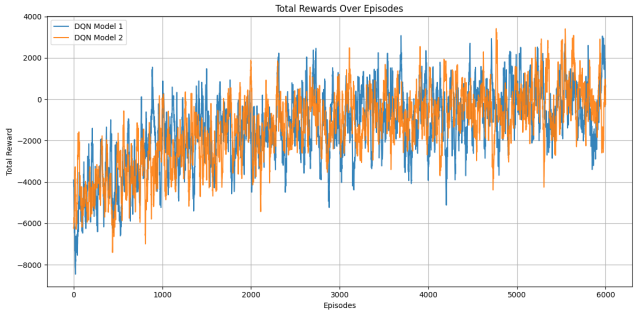

Figure 7: Total Reward vs Episode

Future Work

In this experiment, we can see that our model can be improved. First, we could run more episodes to try, and train longer. Training requires a lot of time along with fine-tuning parameters to determine try and build a good model. If given more time, we would love to run more episodes in order to create a better model. Along with this, we could try and experiment more with other parts of the neural network. For example, our team were not certain which activation functions and loss functions were optimal. With more research we could determine this and create a better model. On top of this, we could create more maps to randomly pick from and train the models on a variety of maps. Along with this, you could train in a battle royale scenario where multiple AI models are facing each other and learning off of each other instead of only a one versus one scenario. This could promote different strategies that are not found in a one versus one scenario.

While this is just a model for a top-down shooter that we made, we believe that this model can be extended to other applications both in and out of gaming. Any top-down game can use the movement logic from this model and fine-tune the model to add actions specific to those games. Outside of gaming, this could be used to plan real-life strategic plans such as military movement. This can also be used in robotics as DQN can be used for path finding in an environment. Finally, this can be used in academic research to enhance the knowledge of artificial intelligence.

Conclusion

Overall, the goal of this project is to learn more about reinforcement learning and determine its value in creating an AI for top-down shooters. We hope to create a fully-functional model that can be used in our current game environment and other similar environments. This model can be implemented in various other applications in gaming as the decision-making can translate to most shooting games. In fact, this could even be extrapolated to applications outside of gaming such as academic research in AI or real-life strategic plans. When tuning our reward system, we found that a good mix of rewards between promoting good movement, rotation, and shooting was important. Too much in one part will lead to the AI only performing that action, leading to a poor performance from the AI.

References

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin A. Riedmiller,
Playing Atari with Deep Reinforcement Learning, CoRR abs/1312.5602, 2013.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, et al.
Human-Level Control Through Deep Reinforcement Learning, Nature 518(7540):529–533, 2015.

[3] Hado Van Hasselt, Arthur Guez, David Silver.
Deep Reinforcement Learning with Double Q-Learning, Proceedings of the AAAI Conference on Artificial Intelligence, 2016.

[4] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Van Hasselt, Marc Lanctot, Nando de Freitas.
Dueling Network Architectures for Deep Reinforcement Learning, In Proceedings of the International Conference on Machine Learning (ICML), 2016.

[5] Ionut Cristian Resceanu, Robert Vlad Iacov, Virginia Radulescu, Stefan-Irinel Cismaru, Florina-Luminita Besnea Petcu, Cristina Pana, Andrei Costin Trasculescu.
A Study Regarding Deep Q-Learning Algorithm for Creating Intelligent Characters in a Graphic Engine, 2022 23rd International Carpathian Control Conference (ICCC), 2017.

[6] Todd Hester, Matej Vecerik, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Dan Horgan, John Quan, Andrew Sendonaris, Ian Osband, Gabriel Dulac-Arnold, John Agapiou, Joel Leibo, Audrunas Gruslys.
Deep Q-Learning from Demonstrations, Thirty-Second AAAI Conference on Artificial Intelligence 2018, 2018.

[7] Konstantinos Souchleris, George K. Sidiropoulos, George A. Papakostas.
Reinforcement Learning in Game Industry—Review, Prospects and Challenges, Appl. Sci. 2023, 2023.

[8] Melrose Roderick, James MacGlashan, Stefanie Tellex.
Implementing the Deep Q-Network, 30th Conference on Neural Information Processing Systems (NIPS 2016), 2016.