Training Rabbits with Reinforcement Learning

Alexander Zhang Zhaomeng Chen Krishi Sabarwal Omar Elamri

Abstract

We used reinforcement learning algorithms to train models that play a simple 2D grid game. We achieved a high win rate of 87% using the deep Q-network algorithm, a convolutional neural network, and dense rewards that encourage beneficial behaviors while penalizing risky actions. We found that environments with highly random rewards are challenging for the methods that we tried and a well-designed reward function is essential for good performance.

1. Introduction

The goal of our project is to use reinforcement learning to play a simple game from a CS 31 assignment called Bad Bunny. In this game, the player is in a grid arena along with a number of randomly placed "killer rabbits." In each time step, the player can move one square in each of the four directions or drop a "poisoned carrot" on the current square (dropping a poisoned carrot onto a square that already has a poisoned carrot has no effect). After the player makes a move, each of the rabbits will move one square in a random direction. If a rabbit moves onto the square where the player is, the player loses. If a rabbit moves onto a square with a poisoned carrot, the rabbit eats the poisoned carrot and dies. The player wins if all rabbits die. For our experiments, we set the game's grid size to 16×16 and the initial number of rabbits at the start of each game to 16. A screenshot of the game is shown in Fig. 1.

We chose this game because although its mechanics are simple, playing it optimally is nontrivial. The player has to avoid being killed by a rabbit while also placing carrots near the rabbits so that the rabbits die as soon as possible. A simple strategy would be to stay away from rabbits, place carrots when possible, and wait for the rabbits to wander into the carrots. A more sophisticated strategy might involve distributing the carrots around the arena to increase the chances of rabbits eating them or surrounding rabbits with carrots.

The observation given to the model consists of three matrices with the same size as the arena stacked together, where the entries in the matrices correspond to grid squares. The first matrix encodes the position of the player. It has a 1 where the player is and 0 everywhere else. The second matrix indicates whether there is a carrot at each grid square, with a value of 1 at squares with carrots and 0 at squares without carrots. The third matrix indicates the positions of the rabbits, with a 1 at squares with at least one rabbit and 0 at squares with no rabbits. Note that the observation does not contain enough information to determine the entire game state because there can be multiple rabbits on the same square. However, this doesn't happen often and is unlikely to affect performance so we did not use RNN policies or frame stacking.

We experimented with training models using deep Q-networks (DQN) [2] and proximal policy optimization (PPO) [5]. We used model architectures consisting of convolutional layers with batch normalization and max-pooling followed by fully connected layers.

2. Results

We measured the performance of our models by their win rate, which is the proportion of games played that end in a win. Detailed results are shown in Tab. 1. We were not able to achieve good results using PPO, so we will focus on the results that we obtained with DQN.

Our best model achieved a win rate of 0.87 using DQN

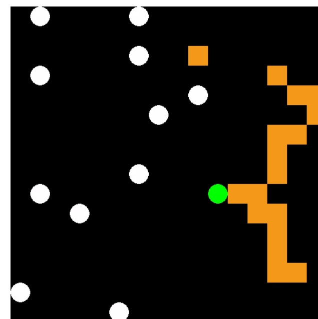

Figure 1. A screenshot showing our implementation of the Bad Bunny game. The player is represented by a green circle and white circles represent rabbits. Orange squares represent squares with poisoned carrots. Note that this graphical rendering is only used to visualize the game for humans. The model uses a different simplified representation.

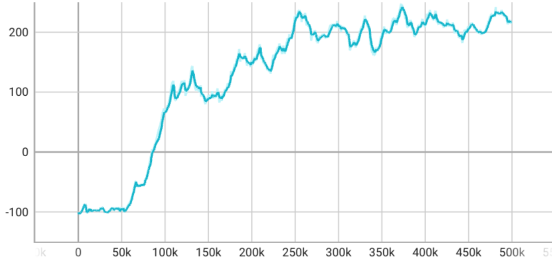

Figure 2. A graph of the training progress for the the best model. The horizontal axis is the number of training time steps and the vertical axis is the mean reward. This model was trained using reward function 1 described in Appendix B.1.1.

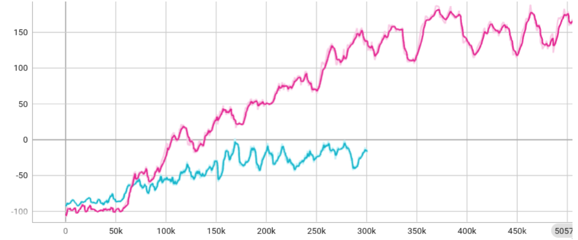

Figure 3. A graph comparing mean reward during training with and without a negative reward penalty for being next to a rabbit. The horizontal axis is the number of training time steps and the vertical axis is the mean reward. The blue line shows a model being trained without the penalty, and the pink line shows a model being trained with the penalty. Note that with the penalty, the mean reward is initially lower since the penalty decreases the reward, but the mean reward quickly increases and surpasses the mean reward without the penalty.

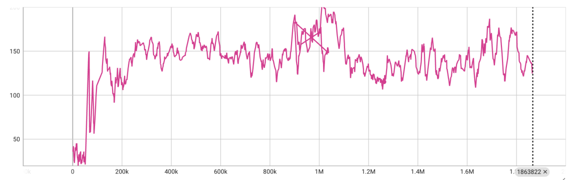

Figure 4. A graph of the mean episode length for the model trained with a reward that incentivizes faster wins. The horizontal axis is the number of training time steps and the vertical axis is the mean episode length. The glitch in the middle is due to a mistake that caused some extra data to be written to the logs.

and dense rewards that explicitly encouraged the agent to drop carrots, kill rabbits, and avoid being next to rabbits. The network uses three convolutional layers with 3×3 kernels and 2×2 max-pooling after each layer, followed by four fully-connected layers. Details are explained in Appendix B (the model was trained with reward function 1 described in Appendix B.1.1 and uses feature extractor 3 described in Appendix B.2.3). A graph of the model's rewards during training is shown in Fig. 2. Videos of the model playing the game show that the model efficiently places a large number of carrots throughout the arena while not getting too close to rabbits. Another model achieved a win rate of 0.86 and used a similar network architecture but without the third convolutional layer.

We found that the biggest factor in the performance of the models was the reward. Adding a negative reward for being next to a rabbit greatly increased the model's performance, as shown in Fig. 3. When we experimented with removing the penalty in reward functions 2 and 3 described in Appendix B.1.2 and Appendix B.1.3 respectively, the resulting models performed poorly. We found that a carefully-designed reward function is essential for helping the model learn.

When we tried incentivizing faster wins with reward function 4 described in Appendix B.1.4, the win rate of the model slightly decreased over time, but the mean episode length also decreased indicating that the model is more aggressive as we expected. A graph of the mean episode length is in Fig. 4 and a video of this model playing the game is available at https://drive.google.com/file/d/1e4cjUZfSVIa9iybIe581DrZYg319GSD5/view?usp=sharing. Note that because the player moves first in each time step, it is safe for the player to be next to a rabbit as long as they move to a square not adjacent to any rabbits in the next time step.

3. Discussion

3.1. Reward Engineering

Achieving the results that we got required dense rewards that incorporated human knowledge of how to play the game well. The model performs poorly if it is only given a positive reward for winning and a negative reward for losing. This reflects one of the major limitations of current reinforcement learning algorithms, and well-designed rewards are often necessary for good performance in other environments [1].

There are likely several reasons why learning with sparse rewards is extremely difficult in our environment: First, an untrained model has a very low chance of winning the game, since this requires effectively avoiding rabbits to stay alive while dropping enough carrots to kill the rabbits. If the only rewards were for losing and winning, the model will never get the winning reward. Learning is difficult even with a reward for killing each rabbit due to the random nature of the game. In order to kill a rabbit, the player must drop a carrot and then wait for a rabbit to randomly wander to the square with the carrot. This can take many time steps, and the player has to stay alive in the meantime. Dropping a

carrot only changes the expected reward by a small amount, but the reward has very high variance due to the random movement of the rabbits. These factors make it challenging for the model to learn the association between dropping carrots and killing rabbits. The small reward for dropping carrots was effective because it was deterministic and given immediately after the action that we are trying to encourage.

We were disappointed in the fact that we could not make the model learn to avoid rabbits without an explicit penalty for being next to a rabbit. This might be because the model initially loses almost every game, so the Q-network learns to always predict a large negative reward. If the player is next to a rabbit, there is a 25% chance of a large negative reward in the next time step, but the network predicts that there will always be a large negative reward eventually so there is little reason for the player avoid being next to the rabbit.

3.2. Model Architecture

Our best-performing models used two or three convolutional layers with max-pooling after each layer. One possible reason why these architectures performed well is the shift invariance property of the max-pooling layer. Playing the game well requires the model to pay attention to things near the player, and the absolute position of the player in the arena is less important. For example, regardless of where the player is, if there is a rabbit on the left side of the player, the player should move up, down, or to the right. Using several max-pooling layers makes it easy for the network to detect certain patterns regardless of where they occur in the arena.

3.3. Stalling Behavior

One issue that we noticed with some of the models that performed well was that when there are few rabbits left, the player would sometimes stay in one square for a large number of moves. This doesn't affect the win rate much since the player is a safe distance away from the rabbits and the rabbits would eventually wander into carrots, but ideally we would like to see our model win as fast as possible. This might be happening even though there is a reward for dropping carrots because the model is not sensitive to things far away from the player. If the player happens to not move in one time step, the changes in the observation will be small and far away from the player's position. The model's output will therefore only change by a small amount, so the player is likely to continue staying still. Another potential factor is that during training, the ε-greedy policy chooses a random action with a probability of 0.05, but during evaluation, the action with the best predicted value is always chosen. The random actions might prevent the stalling behavior from occurring during training.

We tried to fix this issue by using a decaying win reward that incentivized faster wins, and it appeared to have some effect but the model still occasionally stalls. Another potential way to disincentivize stalling and decrease the win time is adding rewards for exploration. This could be implemented by storing a least-recently visited cache of positions visited and increasing the reward on eviction.

References

[1] Alex Irpan. Deep reinforcement learning doesn't work yet. https://www.alexirpan.com/2018/02/14/rl-hard.html, 2018. 2

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, Feb. 2015. 1

[3] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. 5

[4] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021. 5

[5] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017. 1

[6] Mark Towers, Jordan K. Terry, Ariel Kwiatkowski, John U. Balis, Gianluca de Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Arjun KG, Markus Krimmel, Rodrigo Perez-Vicente, Andrea Pierré, Sander Schulhoff, Jun Jet Tai, Andrew Tan Jin Shen, and Omar G. Younis. Gymnasium, Mar. 2023. 5

A. Detailed Results

Tab. 1 lists the win rates for each experiment.

Algorithm	Feature extractor	Reward function	Other variations	Number of time steps	Win rate
DQN	3	1	50000 buffer size	4×10^5	0.87
DQN	2	1		5×10^5	0.86
DQN	2	1		5×10^5	0.85
DQN	3	1		4×10^5	0.84
DQN	3	4		5×10^5	0.83
DQN	1	1		5×10^5	0.65
DQN	3	1		2×10^5	0.25
DQN	1	3		2×10^5	0.07
DQN	1	2		3×10^5	0.05
PPO	1	1		1×10^5	0
PPO	1	1		2×10^5	0

Table 1. Table showing the results of each experiment. We stopped each trial when the model was no longer improving or when the performance was not better than other trials. Win rates were calculated from 100 games.

Hyperparameter	Value
Learning rate	10^{-4}
Replay buffer size	10^6
Number of steps before learning starts	100
Batch size	32
Soft update coefficient	1
Discount factor	0.99
Number of time steps per update	4
Number of gradient steps per iteration	1
Target network update interval	10^4
Exploration rate	0.05
Maximum gradient norm	10

Table 2. DQN hyperparameters.

Hyperparameter	Value
Number of parallel environments	8
Learning rate	3×10^{-4}
Number of time steps per update	128
Batch size	64
Number of epochs	10
Discount factor	0.99
GAE parameter	0.95
Clipping parameter	0.2
Entropy coefficient	0
VF coefficient	0.5
Maximum gradient norm	0.5

Table 3. PPO hyperparameters.

B. Methods

We implemented the Bad Bunny game in Python with the Gymnasium API [6], and we used implementations of DQN and PPO from Stable Baselines3 [4]. We used PyTorch [3] to implement our models.

B.1. Rewards

Besides a large negative reward for losing and a large positive reward for winning, we used several supplemental rewards to help the model learn. We gave the model a positive reward whenever a rabbit dies, and we also gave a small positive reward to the model for dropping a carrot on a square that doesn't already have a carrot. In some experiments, we gave the model a negative reward for making a move that results in it being next to a rabbit. The following sections describe the reward functions that we used in detail.

B.1.1 Reward Function 1

This function gives a reward of -100 for losing, 100 for winning, 1 for dropping a carrot, 10 for killing a rabbit and -5 for being next to a rabbit. The negative reward for being next to rabbits is calculated after the player moves and before the rabbits move so that it penalizes risky actions that result in the possibility of losing in the next time step.

B.1.2 Reward Function 2

Reward function 2 is the same as reward function 1, except that the negative reward for being next to a rabbit is removed.

B.1.3 Reward Function 3

Reward function 3 replaces the negative reward for being next to a rabbit in reward function 1 with a linearly decreasing positive reward in the first 100 time steps to encourage survival. The additional reward at each time step is given by the following formula, where t is the number of time steps since the start of the current episode:

$$\texttt{turn_reward}(t) = \begin{cases} 1 - \frac{t}{100} & t \leq 100 \\ 0 & t > 100 \end{cases}$$

B.1.4 Reward Function 4

Reward function 4 is the same as reward function 1 except that the win reward decays during each episode. The reward for winning is given by the following formula, where t is the number of time steps in the episode:

$$\texttt{win_reward}(t) = 75 + 200 \times 0.99^{t}$$

This reward was designed to prevent stalling by incentivizing faster wins.

B.2. Model Architectures

Our models consisted of CNN feature extractors followed by fully-connected networks. The feature extractors are described in the following sections. For DQN, the Q-network consist of the feature extractor, two hidden layers each with 64 units, and an output layer with 5 units, one for each possible action. The hidden layers use ReLU activation. For PPO, the policy and value networks share the feature extractor and each have two hidden layers with 64 units and ReLU activation. The policy network has an output layer with 5 units and the value network has an output layer with a single unit.

B.2.1 Feature Extractor 1

Feature extractor 1 has two convolutional layers and one fully-connected layer. The convolutional layers each have 32 filters with a kernel size of 3×3, stride 1, and padding 1. The fully-connected layer has 256 units. Batch normalization and ReLU activation is applied after each layer.

B.2.2 Feature Extractor 2

Feature extractor 2 also has two convolutional layers and one fully-connected layer. The first layer has 32 filters and the second layer has 64 filters. Both layers use 3×3 kernels and are followed by batch normalization, ReLU activation, and 2×2 max-pooling. The fully-connected layer has 256 units, batch normalization, and ReLU activation like the baseline feature extractor.

B.2.3 Feature Extractor 3

Feature extractor 3 is similar to feature extractor 2, but it has a third convolutional layer with 128 filters and 3×3 kernels before the fully-connected layer. The additional layer also has batch normalization, ReLU activation, and 2×2 max-pooling.

B.3. Other Hyperparameters

Due to time limitations, we were only able to tune some of the many hyperparameters that RL algorithms have. We used default values for the other hyperparameters shown in Tab. 2 and Tab. 3. We used the Adam optimizer for all experiments.