CS230

SnakeAI

Thomas Jiang, Claire Mai
Stanford University
twjiang@stanford.edu, cmai21@stanford.edu

Abstract

Our goal was to create an AI agent to play the Snake game. We utilized a Deep-Q-Network (DQN) and compared the best scores the AI agent can achieve when we have a naive state space and a screenshot state space. The naive state space consisted of a sparse vector encoding import features of the game such as what direction the snake is facing and where food and obstacles are relative to the snake. This sparse vector was fed into a simple neural network consisting of 2 linear layers and estimated q-values for each action the snake can take. The screenshot state space consisted of a image of a frame of the game itself and feeding that image into a CNN which then estimated q-values for each action. We found that the simple state space with the linear layer model outperformed the complex state space with a CNN model.

1 Introduction

Snake was a popular arcade game played on the computer. The objective of the game was to eat as many apples as you can without crashing into a wall or the snake's body itself. The snake begins with a length of one (i.e. only the head) and each time the snake eats an apple, it increases its length by 1. As children we were frustrated at how difficult it was to beat the game, thus we hope to leave that dream to our new AI friend.
Our goal is to create a deep reinforcement learning agent that plays the snake game. However, because there are many projects out there that have built similar deep q networks for snake AI, we want to differentiate ourselves by experimenting with different state spaces, reward systems and network architecture and observe the performance of the model. We aim to take inspiration from Deep-Q-Networks implemented from other games such as Atari [4] and found that

1. With a complex reward system, the snake will take less time to train
2. A state space consisting of an image of the game proved to learn less efficiently than a naive state space

2 Related work

Since the 1990s, reinforcement learning has become the leading approach to create AI agents for games. Watkins and Dayan created a popular model-free reinforcement learning approach called Q-learning [1]. Q-learning provides agents with the capability of learning an optimal policy for any Markov Decision Process. This is done by trying all actions in all states repeatedly so that the agent learns the best overall action for each state that gives the greatest long-term discounted reward. While revolutionary for learning simple games, Q-Learning proved to perform poorly for games with extremely large state spaces since trying every action in every state will grow exponentially as the state and action space grows.

CS230: Deep Learning, Winter 2018, Stanford University, CA. (LateX template borrowed from NIPS 2017.)

The atari paper took the Q-learning algorithm and further combined it with deep neural networks to give rise to Deep Q-learning Networks or DQNs [2]. Prior to this paper, reinforcement learning on games relied on heavily hand-crafted feature sets. The atari paper was one of the first to utilize CNNs to extract relevant features from the game's video input [2]. This paper introduced the novel concept of the DQN, where after training the deep Q-network, the agent was able to outperform humans on nearly 85% Breakout games [2].

Wei et al. further applied DQNs to the Snake game [3]. In earlier decades, Lin proved that experience replay smooths the training distribution over a large amount of experiences [4]. Thus, Wei et al. also applied experience replay and double experience replay to attain better Snake game results. Wei et al.'s proposed reward mechanism solved the sparse and delayed reward issues that basic DQNs experienced.

However, other changes the DQN model can be changing how the state space is represented. One possible idea that we implement in this paper is to represent the state space as a screenshot of the game and input the screenshot of the game into a convolutional neural network (CNN) in order to predict the best action to taken given the current state. By utilizing a CNN on a screenshot of the game, Kunanusont et al. found that the CNN was able to learn more complex features across a variety of games [5].

3 Dataset and Features

Since our problem was framed inside deep reinforcement learning, our main dataset was extracted from the snake game itself using the experience replay buffer. We implemented the snake game using pygame, heavily referencing from an already implemented snake game found on youtube - specifically the one made by Python Engineer [6]. For our approach we defined these key features: state space, rewards, and experience buffer.

3.1 State space

We trained our model using a memory buffer and (state, reward, action tuples). We implemented a model and agent with two different state spaces: sparse vectors of 0's and 1's representing if key elements that are defined from the game are present, and screenshot of the game.

For the first naive representation, we defined our state space with key elements of the game that we thought would most likely impact whether the snake was able to move forward, turn left/right. This came in the form of determining whether there was a wall or snake body part to our left, front, right which indicates danger, our current facing direction, and food location relative to the snake's head. This sparse vector was the input of our model. A sample of how this sparse array is shown in Figure 1. One key limitation of this model is that the snake is unable to see further beyond than just its front, left and right.

[0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0]

Figure 1: Sparse array representing key elements of the game. Below is table describing what each index of the array represents.

Index	What the value indicates at the index
0	If there is danger (i.e. an obstacle) in front of the snake
1	If there is danger to the right of the snake
2	If there is danger to the left of the snake
3	If the snake is facing left
4	If the snake is facing right
5	If the snake is facing up
6	If the snake is facing down
7	If the apple is to the left of the snake
8	If the apple is to the right of the snake
9	If the apple is above the snake
10	If the apple is below the snake

For the second representation, at each step we took a screenshot of the game, resized it to be 60x60, converted the image to a numpy array, grey scaled the image, and then normalized the image values. The result 2D array was then fed into our CNN. A sample screenshot of the game is shown in Figure 2.

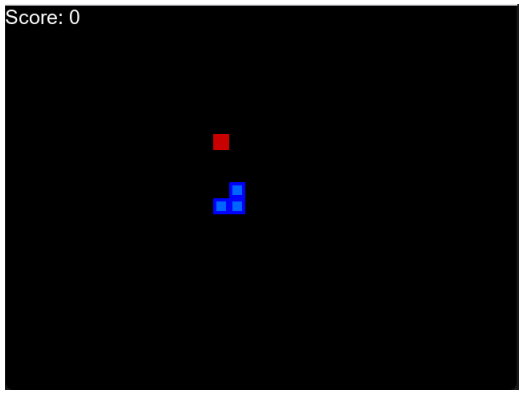

Figure 2: Screenshot of snake game

For both state representations, the output of the model is an array of length 3 (output_dim = 3), to represent a confidence score of the snakes action going (left, forward, right). By taking the argmax of the output array, we can get the best action to take in our current state.

3.2 Rewards

We want the snake to take the shortest path to the apple in the most 'logical way' (without killing itself). We experimented with a few different rewards as described below.

3.2.1 Simple Rewards

For simple rewards, when the snake dies, it receives reward of -10 and when it eats an apple, it is rewarded +10.

3.2.2 Distance Reward

We also wanted a reward system where the snake is rewarded positively if the snake takes a step in the direction of the apple and rewarded negatively if the snake takes a step away from the apple. This was because we noticed that the snake would take a considerably very long time to train in the initial stages without this reward since it has no idea where to step. However, we also took into account that as the snake enlarged - it was probably harder for the snake to take the shortest path to the apple since it often would be obstructed by its own body. Hence we defined our reward by a function of the length of the snake and the euclidean distance from the snake's head to the apple via:

$$R = \log(\frac{L_t + D_{t-1}}{L_t + D_t}) \tag{1}$$

where L_t represents the length of snake at time-step t and D_t represents the normalized (by pixel value) euclidean distance of snakes head at time-step t. We received inspiration for this method from Wei et al. [4].

3.3 Experience buffer

We used a memory queue of length MAX_MEMORY to store (state, action, reward, next_state, done) at every step of the snake. Once this memory queue is completely filled up, we pop the oldest memory out of the buffer. For training purposes, this experience buffer is necessary since we use a short term training method and long term training method. For short term, we train at every step for the most recent memory. For long term, once the snake dies, we take a random sample of length BATCH_SIZE from this buffer and use it to train our Deep Q network.

4 Methods

We implemented a simple Deep Q-network by referencing the pseudocode presented in the Atari paper[4] and in Figure 3. This algorithm is similar to Q-learning but replaces the regular Q-table with a simple neural network. Regular Q-learning utilizes the Bellman equation to update the new state action pair. This Bellman equation is defined as follows:

$$Q(S_t, A_t) = (1 - \alpha)Q(S_t, A_t) + \alpha(R + \gamma Q(S_{t+1}, a)) \tag{2}$$

Our Deep Q-learning models this by using the memory buffer to train using previous states, actions and rewards. Our learning process uses two networks to do this - the main network and target network. The main network is updated for every training step and the target network is updated after the snake dies. Using this model we can then predict the best next action for the snake.

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Figure 3: Atari pseudo-code for DQN.

We conducted experiments by playing with a variety of state and reward mechanisms as explained in section 3. For the simple state space, we used a very simple architecture consisting of two Linear Layers and one ReLU layer in between with an input size of the state dimensions and an output size of possible actions. These linear layers were trained using an experience buffer so that each state element can map to a possible good action.
For the more complex state space (screenshot method), we created a CNN architecture with the following layers: Conv, ReLU, Conv, ReLU, Dense, Dense. We hypothesized that this state space would be able to capture more meaning as to where the snake was relative to the wall and the apple and its body. We resized the screenshot to size 60x60 since this was optimal to training time (larger numbers would result in the snake becoming slower due to our machines' low processing power.

For our final approach, we also modelled dropout by implementing epsilon greedy action selection displayed in figure 4. The idea is that while our model and deep Q network might output the action most suitable given the state space, we also want our model to learn from "bad moves" or if the model consistently is moving badly we want it to move randomly such that some of the moves it can learn are good. There is a trade-off between exploration and exploitation here as the higher the epsilon value, the more our snake is allowed to explore. For our implementation, the epsilon is initialized to 0.4 and slowly decreases to 0.005 as the game continues. In this way, at earlier stages of the game we want the snake to 'explore' more rather than later when it has learned what good and bad moves are given the state space.

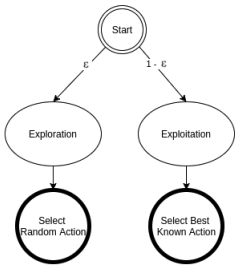

Figure 4: Epsilon greedy action selection.

5 Results and Evaluation

For both state spaces, we trained our Deep Q-learning model with and without the distance reward as explained in section 3.2 with the model architecture defined in section 4.1. We recorded the best score the snake achieved after 100 deaths and after 200 deaths for both state spaces, as seen in Figure 5. We also recorded the average and best score that we achieved while playing the game over 20 times.

	Best Score After 100 Deaths	Best Score After 200 Deaths
Naive state W/o distance reward	32	53
Naive state W distance reward	54	61
Screenshot state W/o distance reward	3	7
Screenshot state W distance reward	4	8

Over 40 games between us,
Human Average score: 6
Human Best score : 15

Figure 5: Table for snake scores

From the table above, we can see that the naive state approach outperforms both the human score and the screenshot approach score. We believe the screenshot approach does worse than the naive due to there being lots of black space for each screenshot. This could cause the CNN to not be able to train well because there is so much background values that don't help the agent train. Another limitation in our screenshot method is that our screenshot method takes raw pixel values which might not be the most accurate representation of the snake game ai - instead we could have used the actual pixel values as the state space. We decided to use this screenshot method as we wanted to see if we could use our CNN on other snake games such as the one played on Google [7]. We also think our relatively small experience memory buffer bandwidth attributed to a decrease in performance. This was because with more complex architectures - neural networks need to handle extremely long training times which means that our snake should be able to grasp experiences from a very big memory buffer. A smaller memory buffer resulted in our snake continuously learning from bad iterations as our epsilon approached minimal value (exploration decreased as explained in section 4).

6 Conclusion

Conclusively, we discovered that complex models do not necessarily mean better performance on some tasks. Infact, in very easy problem tasks, the naive and simple method almost outperforms many different model architectures if we tune the different parameters. We also see a positive correlation that a more intuitive reward system also helped the models perform better. In our case, having a distance reward system enabled the snake to learn to move in the right direction. For future work, we hope to implement Dueling Deep Q Networks (DDQN). Wang et al. proposed dueling netowrk architectures for deep reinforcement learning [8]. With the use of dueling networks, state-value and the advantages for each action are estimated separately and can learn which states are valuable without having to learn the effect of each action for every state. This proves to be more efficient than the original DQN proposed by the atari paper. A DDQN would also complement our screenshot state space and ensure the snake learns a more complex function to adjust to hidden dangers.

7 Contributions

Thomas Jiang - Snake game + AI agent + dense linear Q-training model + distance reward. Coded up project using resources and methods found in other papers. Focused on distance reward, AI agent, and naive training model. Also helped with implementing the CNN screenshot approach. Edited final report and added insights at the end.

Claire Mai - Snake game + AI agent + CNN screenshot model + report. Helped train the model using different reward systems. Coded up project focusing on the screenshot method with CNNs and AI agent. Wrote majority of the final report with insights on related work.

References

[1] Christopher JCH Watkins and Peter Dayan. Q-learning. Machine learning, 8(3-4):279–292, 1992.

[2] Volodymyr Mnih & Koray Kavukcuoglu & David Silver & Alex Graves & Ioannis Antonoglou & Daan Wierstra & Martin Riedmiller (2013) Playing Atari with Deep Reinforcement Learning

[3] Long-Ji Lin. Reinforcement learning for robots using neural networks. Technical report, DTIC Document, 1993.

[4] Wei, Zhepei & Wang, Di & Ming, Zhang & Tan, Ah-Hwee & Miao, Chunyan & Zhou, You. (2018). Autonomous Agents in Snake Game via Deep Reinforcement Learning. 20-25. 10.1109/AGENTS.2018.8460004.

[5] Kunanusont, Kamolwan, et al. General Video Game AI: Learning from Screen Capture - Arxiv. 23 Apr. 2017, https://arxiv.org/pdf/1704.06945.pdf.

[6] Engineer, Python. "Python Snake Game with Pygame - Create Your ... - Youtube." YouTube, 8 Dec. 2020, https://www.youtube.com/watch?v=–nsd2ZeYvs.

[7] Google Snake. https://www.googlesnake.com/

[8] Wang, Ziyu & Schaul, Tom & Hessel, Matteo & van Hasselt, Hado & Lanctot, Marc & de Freitas, Nando (2015) Dueling Network Architectures for Deep Reinforcement Learning https://arxiv.org/abs/1511.06581