Introduction and Motivation:

Snake was a game created for early cell phones in the late 1990s. A snake, which is represented by a few connected squares, moves around a grid to eat food using its head. For each piece of food the snake eats, the snake grows in length by one square, and the score is incremented by one. This process continues until the snake dies by either crashing its head into its own body or hitting the map's boundary. Today, many variations of Snake exist, but the general idea remains the same. An agent controls the snake by selecting an action from a set of possible actions at each timestep, with the goal of maximizing the score. The field of artificial intelligence has developed various algorithms for such problems, and among the most popular is reinforcement learning. This project aims to compare various reinforcement learning agents and explore whether a reinforcement learning agent can outperform a human user.

Reinforcement learning is a family of algorithms for training agents to act intelligently in well-defined environments. Classical reinforcement learning approaches such as Q-learning suffer from the curse of dimensionality. These approaches rely on exploring the state space of the environment, and the larger the state space, the more impractical these approaches become. However, with the expressive power of deep neural networks, deep reinforcement learning methods have been shown to be far more effective for these problems [1]. In this project, we implement reinforcement learning and deep reinforcement learning approaches for Snake and compare their performance. While Snake itself is only a game, the potential for reinforcement learning extends to more complex real-world problems.

Problem Definition:

The game environment is defined as a snake and a food on a square grid. The snake is a list of connected positions as Cartesian coordinates on the grid. The head is the first position in the list. The snake also has a direction; up, down, left, or right. In each game timestep, the head moves one unit in the snake's current direction, and the body is updated such that each element of the list moves to the position of the previous element of the list. At each timestep, the snake can change its direction. If it is moving vertically, it can choose a horizontal direction. If it is moving horizontally, it can choose a vertical direction. In either case, it also has the choice of continuing in the direction it had in the previous timestep. The environment also contains food. The food spawns at the beginning of the game and remains in the same position until the snake reaches it. When the snake head reaches the food, the score increases by 1, and the food moves to a different position which can be any position that the snake body is not currently in.

The computational problem presented is how to create an agent to select an action at each timestep, with the goal of maximizing the score of Snake using our defined environment. The agent will take a representation of the environment as input and will output an action. The best possible series of actions will guide the snake toward the food and avoid game-over states. These game-over states are defined as those in which the snake's head overlaps with its body, or when the head goes beyond the grid boundaries.

Proposed Method:

There are many types of reinforcement algorithms that can be applied to create a Snake AI. Q-learning is an active reinforcement learning method that explores the environment state space by playing through many training episodes and estimating the Q-values for state-action

pairs with the goal of learning the best action at a given state. Q-learning builds a Q-Table and updates it using the Bellman Equation as the agent takes actions and receives rewards. The received reward is determined by a defined reward function.

The disadvantage of Q-learning is that it becomes impractical to store all the Q-values when the environment state space becomes very big. To address this issue, there is feature-based approximate Q-learning. This is a modification of Q-learning that reduces the state space by extracting certain features from the environment and defining the state using those features. In our implementation of approximate Q-learning, we are extracting three features: the distance from the head to the food, the minimum distance from the head to the body, and the presence of a body square in the direct path from the head to the food. The disadvantage of approximate Q-learning is that a state space defined by a linear feature extractor may not be a good representation of the true state space.

With recent advances in deep learning, a deep reinforcement learning algorithm called Deep Q-Network (DQN) was introduced, based on approximate Q-learning. DQN has been shown to perform more efficiently and better maximize reward functions compared to exact and approximate Q-learning implementations [1]. DQN learns a policy by maintaining a policy network and a target network of identical structure. To learn the policy, we implemented a deep neural network with three convolutional layers and two fully-connected layers, which takes the state of the game as a 10x10 matrix with 0s in empty positions, 1s in snake body positions, 2 in the snake head position, and 3 in the food position and outputs an action. Loss is calculated using the smooth L1 loss function between the output of the policy network and the target network. The target network is synchronized with the policy network every 10 epochs.

Reinforcement learning methods all require a reward function and a set of hyperparameters, which can be tweaked to optimize the performance of the algorithm in different environments. These hyperparameters include the learning rate, discount rate, and epsilon. An ideal reward function for Q-learning would be to return a very positive reward when the snake consumes a food, and a very negative reward when an action results in a game-over state. To reduce the incentive of the snake infinitely looping, a slightly negative reward is returned for each action in which a food is not consumed.

Experiments and Results:

To examine the differences in performance among different algorithms, we seek to answer the following questions: How do the algorithms compare to each other? How do they compare to human users? And how does their performance change in different environments? To answer these questions, we define our testbed and evaluation metrics as follows. Six different agents were tested: a random agent, a reflex agent, a Q-learning agent, an approximate Q-learning agent, and a DQN agent. Human performance is also recorded as a baseline for comparison. The metrics used to measure performance were mean game score and mean game steps. Our implementation of Snake was adapted from a version of Snake using the Pygame library [2], and our reinforcement learning agents were inspired by existing implementations [3, 4]. The hyperparameter values were set as described in Table 1.

Hyperparameters for Training Reinforcement Learning Agents					
Agent	Alpha	Gamma	Start Epsilon	End Epsilon	Decay Method
Exact-Q	0.3	0.8	0.8	0	Linear
Approx-Q	0.3	0.8	0.8	0	Linear
DQN	0.01	0.8	0.9	0.05	Exponential

Table 1:

Hyperparameter values for our implementations of reinforcement learning agents

All agents were evaluated in a random food spawning environment, where the position of the next food spawned after a food is eaten is chosen randomly (Table 2). Additionally, the Q-learning agent, approximate Q-learning agent, and DQN agent were also evaluated in a fixed food spawning environment, in which the order of the food positions is deterministic and the same as it was in training (Table 3). Each algorithm was tested for 100,000 games, and the human testing occurred for 60 games at 8 frames per second. The score and game length metrics were also recorded over the training period for the approximate Q-learning and DQN agents. The approximate Q-learning agent trained for 20,000 games (Figure 1), while the DQN agent trained for 50,000 games (Figure 2). Both agents were trained on random food spawning.

Random Food Spawning						
Evaluation Metric	Random Agent	Reflex Agent	Exact-Q Agent	Approx-Q Agent	DQN Agent	Human (8 FPS)
Mean Score (*Min, Max*)	0.141 (*0, 4*)	6.383 (*0, 36*)	0.155 (*0, 4*)	11.45 (*1, 34*)	1.485 (*0, 12*)	12.617 (*0, 35*)
Mean Steps (*Min, Max*)	17.026 (*3, 165*)	44.838 (*3, 240*)	17.094 (*3, 169*)	84.872 (3, 262)	150.127 (*3, 1000*)	170.133 (*7, 387*)

Table 2: Testing all agents with randomly spawned food

As seen in Table 2, no algorithm reached the baseline mean score of a human user. Interestingly, the approximate Q-learning agent achieved an average score very similar to that of a human and did so in just over half the average steps per game. This suggests that the approximate Q-learning agent is more efficient than humans at maximizing the score per game step. However, unlike the reflex agent, which is entirely greedy in its attempt to eat food, the approximate Q-learning agent learned to keep itself alive to achieve higher scores in certain situations. The exact Q-learning agent struggles in the random food spawning environment, barely scoring higher than the random agent because it only observed a small fraction of the total state space in training. The DQN agent also does not perform well in this environment and often gets stuck in a loop after eating a single food, evidenced by the average score of just over 1 and a max step count of 1000 (the maximum number of steps before the game times out).

Fixed Food Spawning			
Evaluation Metric	Exact-Q Agent	Approx-Q Agent	DQN Agent
Mean Score (*Min, Max*)	6.247 (*3, 9*)	5.499 (*4, 7*)	10 (*10, 10*)
Mean Steps (*Min, Max*)	4.7861 (*19, 73*)	35.993 (*27, 45*)	73 (*73, 73*)

Table 3: Testing reinforcement learning agents with deterministic food spawning

Table 3 shows how the exact Q-learning, approximate Q-learning, and DQN agents performed after training and testing in a deterministic food-spawning environment. The average scores and game lengths are dramatically different compared to the environment in Table 2 (random food spawning). With deterministic food spawning, the average scores of our exact Q-learning agent and DQN agent are significantly higher. In a fixed food environment, exact Q-learning performs better because it has observed most of the testing states in training. DQN also performs significantly better in a deterministic food environment than in a random one because it is memorizing which actions to take in each training state. The DQN and exact Q-learning agents do not generalize well, as seen from their low mean scores with random food spawning.

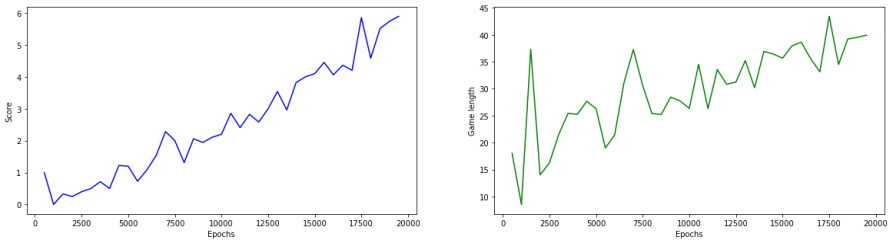

Figure 1:

Training of approximate Q-learning on randomly spawned food

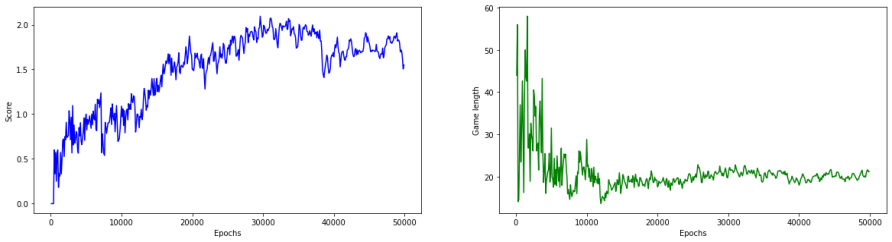

Figure 2:

Training of DQN on randomly spawned food

Figures 1 and 2 show the progression of score and game length over the training period. Figure 1 describes the approximate Q-learning algorithm training, while Figure 2 describes the DQN training. In Figure 1, there is a clear positive relationship between the number of games played in training and the score. The game length displays a similar trend. In Figure 2, however, DQN demonstrated an increase in score as it played more games, but the positive relationship between these variables diminished as the training continued. As training progresses, epsilon decays, resulting in less exploration by the model. This is why the game length converges over the training. In Table 3, the mean score and game length for the approximate Q-learning agent are lower than random spawning, but this is only because the seed is fixed. With a different random seed, the performance of the approximate-Q agent could be better.

Conclusion and Discussion:

This report discusses the performance of different reinforcement learning algorithms on the game of Snake. We presented the methodologies of each approach, examined their learning during training, and evaluated their performances. Although none of the algorithms could exceed human performance, approximate Q-learning proved to be very effective. While our implementation of DQN did not perform as well as expected, one area of future work could be modifying the neural network and the training algorithm. Combined with longer training sessions, these modifications could significantly improve the performance of DQN. Another area of future work could be an implementation of a different deep reinforcement learning algorithm, such as a policy gradient method.

Individual Contribution:

Each project member contributed equally to code development, training and testing models, and writing the report. Individual contributions can be observed on the commit history in our GitHub repository (https://github.com/mhermon/snake-pygame-ai).

Citations

[1] - Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing Atari with Deep Reinforcement Learning. arXiv:1312.5602. Retrieved from https://arxiv.org/abs/1312.5602

[2] - Rajat Biswas. 2021. Manual Snake Game Template. SCM: https://github.com/rajatdiptabiswas

[3] - John DeNero, Dan Klein, Pieter Abbeel. Spring 2022. Artificial Intelligence CS:188 UC Berkeley. Retrieved from: https://inst.eecs.berkeley.edu/~cs188/sp22/

[4] - Adam Paszke. 2022. Pytorch DQN Training Tutorial. Retrieved from: https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html