

CopyConda: Playing Snake With Imitation vs Reinforcement Learning

Rio Blumenthal, Chase Carstensen, Lucas Lamenha, Keira Schoolcraft

CS539: Machine Learning

Final Report

1. Introduction:

Snake is one of the oldest video games in the world having been created in the mid '70s, and it is still widely played today. It is a very straightforward game that requires the user to change the direction of a pixel snake as it tries to collect fruit which are randomly placed on the board, and the pixel snake will increase in length for each fruit collected. The game can be decomposed into snapshots of the gamestate containing the current position of each segment of the snake, and the position of the fruit. The idea of using machine learning to play games at a high level has been around for a long time, and started with simple board games (Furnkranz, n.d.). Similarly, the simplicity of Snake makes it an excellent game for training ML models, because it's easy to break it down into its component parts. For this project, the goal is to use both Imitation Learning and Reinforcement Learning to create agents which can play Snake at an expert level. The motivation behind this project is to demonstrate the pros and cons of using different machine learning techniques to solve similar problems.

2. Background

2.1 Imitation Learning

The initial goal of the project was to use Imitation Learning to train a ML model to play the classic video game Snake. The central idea behind imitation learning is taking a set of observations labelled with actions, and having a machine approximate the function which maps new observations to actions (Hussein et al., 2018). In the context of our project, this means taking Snake game board states (the current position of the food, and the position of the snake

head and body) and labelling them with a move (up, down, left, or right). The limitations of this model are that it requires lots of initial training data, and that it is difficult to train the model to be better than the best humans who created the data due to the presence of human error in the training data (Hussein et al., 2018). In this project two models, Convolutional Neural Networks (CNNs) and Random Forests were used to perform the Imitation Learning. CNNs work by performing a convolution on a multidimensional input, followed by an optional pooling step, and lastly passing the convolution output to a fully connected feed forward neural network where final predictions are made. Random forests are a type of ensemble learning method, which uses bootstrap aggregating (bagging) to aggregate the predictions of a number of weak learners. Random forests achieve this by training a large number of different decision tree models, and then generating a final prediction through a majority vote.

2.2 Reinforcement Learning

To overcome this limitation, the team decided to also implement Reinforcement Learning. This involves having an agent play Snake many times, giving it rewards for successes and penalties for mistakes, with the goal of maximizing rewards (Kaelbling et al., 1996). Hopefully, the model can improve over many iterations without human guidance. The benefits to this approach are that it does not require a prelabeled dataset, and it also has the capacity to become better/more creative than humans, unlike with imitation learning. At the end of the project, the objective is to have at least one functional model for each system, both imitation and reinforcement learning, and be able to run them against each other to compare performance.

3. Progress and Learning Pipelines

3.1 Game Environment Development and Data Collection

Before being able to train the model using either Reinforcement Learning or Imitation Learning, the team needed an implementation of the game of Snake itself. It needed to be modular, meaning we wanted both a computer and a person to play it, and it needed to export the data collected within the game to some external source for compilation and training. There are many existing implementations of Snake out there, but because of the simplicity of the game, and our desire for complete control over our system we decided to develop our own app. The first step was to implement the Snake game logic in Python. We then developed a UI for the game using the Pygame library (Figure 1, 2). Next, a logger was added to be able to quickly export the data. The team developed versions of our app for Linux, Mac, and Windows to allow users to play the game on all platforms. Once the data was available in Google Sheets (around 500 games, with 100,000+ individual game states) it needed to be cleaned and filtered. First, the games in which the player earned less than a certain number of points were dropped, with the specific number varying across testing different models. Secondly, a Python script was created to parse the downloaded data and remove the last five moves of every game, since those moves resulted in the death of the snake. This prevents the model from being trained on any games that failed / moved incorrectly. Finally, a script was run to extract only the relevant columns: the current direction and the board state.

3.2 Imitation Learning Pipeline

First, Imitation Learning required a way to efficiently collect the necessary training data. In order to do this, the team modified the game we had developed to export game data to a Google Sheet. We chose Google Sheets due to its large capacity, convenient API, and easy access across devices. The app now allows the user to play games of Snake, and automatically send the data collected back to the central Google Sheet following the conclusion of the game (Figure 3). Specifically we sought to collect pairs of observations and actions. The logger implemented in our app collects this data for every board state in a game played and stores that in our Google Sheet.

A large quantity of data is needed in order to properly implement imitation learning, so we shared our app with our peers and asked them to play some games of Snake. This data was then sent back to our central Google Sheet. At the moment, we are still collecting data to use for the imitation learning phase of our project, and the goal is to eventually use the data collected by our game to train an imitation learning model.

Given the nominal nature of our target variable (the action to take at the present board state) we plan on using classification models to implement Imitation Learning. This allows us to test a variety of different classification models discussed in this class, such as logistic regression, k-nearest neighbors, and decision trees to and compare the performance of each approach.

The final two models tested were a CNN and a Random Forest decision tree. Several specific CNN designs were tested, with the largest difference between them (beside parameter tuning) was that the first CNN model took input in the form of (16, 16, 1) and the second took it in the form of (16, 16, 3). The first one only looked at the current board state, while the second took a 3-dimensional input, training on the previous two states as well. After tuning, the best

model converged after only 5 epochs, and output a score for each possible direction (UP, DOWN, LEFT, RIGHT) representing its likelihood. The `ai_pick_direction` function in the game took the largest of these scores and used that as the AI move, with the exception of the AI trying to go backward. In this specific case several different solutions were tested, with the best being forcing the AI to turn left or right.

When it comes to random forests, we used the Scikit-Learn implementation of random forests to fit our trees. We engineered a number of features specifically for decision tree classification such as food direction, and encoded obstacles (walls, snake body). We then tested many different forest topologies in an attempt to maximize snake performance. Similarly to the CNN, our implementation of the snake game could then use the random forest model to examine the current board state and predict the next direction.

3.3 Reinforcement Learning Pipeline

A Reinforcement Learning Model was trained to run comparisons against the Imitation Learning Model. For this, a Deep-Q Network (DQN) was implemented. This was done due to the discrete nature of the Snake Game, which aligns perfectly with DQN's ability to discern between what discrete action to do by estimating Q-values for each possible action, along with its built-in experience replay, which allows the agent to learn better from past experiences - a valuable ability to have when playing Snake (Sebastianelli et al., 2021). The model takes in 7 features, each of which represent a game state (relative food position, danger detection in 4 directions, normalized snake length, etc.), with 3 hidden layers of 128, 256, and 128 neurons with ReLU (Rectified Linear Unit) activation. The output layer for the model contained 4 neurons representing action values for each direction the snake can move in, with Xavier weight initialization to prevent vanishing gradients during training. The algorithm used a double DQN

approach with separate policy and target networks, the latter of which updates every 1000 steps for stability. To balance exploration and exploitation for the model, an epsilon-greedy strategy was employed, with an epsilon decay of 0.9999 for each step, with a minimum epsilon value of 0.05 to maintain exploration.

For training, a reward structure was used to allow the model to understand when it did well in a particular run through the game. The following reward function was used:

- Survival reward: +0.01 per step
- Directional reward: $+0.1 \times \text{distance improvement toward food}$
- Food consumption: +1.0 for eating food
- Starvation penalty: -0.5 for not finding food after 200 steps
- Death penalty: -1.0 for game over

5. Model Statistics and Comparison

5.1 Imitation Learning Results

While the imitation learning CNN reported a testing accuracy around 94%, in practice no imitation learning model was able to play Snake to any significant degree. The models were trained on around 66,000 game states, which included only those games which matched the specifications described in section 3.1. The biggest issue noted with the CNN approach was that while it very reliably avoided the first wall, it did a bad job at both avoiding subsequent walls and seeking the fruit. Without intervention, due to its lack of memory, it had a tendency to get stuck in loops. Additionally, most of the time when it died, it was due to the model attempting to make an illegal move (trying to move backward), sometimes with high confidence and

sometimes with low. The team attempted to resolve this issue by including “memory” in the form of including the previous two game states, but this only resulted in minimal improvements. The model struggling at the beginning is likely due in large part to the snake starting at the same position facing the same direction every game. This means that, due to humans not reacting instantly, most of the beginnings of the game look the same and involve avoiding the wall. However, since the model has only been trained on avoiding that one wall, it was much worse at avoiding any other wall. This meant that if it couldn’t find the fruit on the first attempt, it would likely run into the wall. This could be potentially mitigated in the future by adding data augmentation in the form of rotations and mirrors.

Similarly, the random forest model showed very poor performance on real time snake games. This was surprising given that this model had a very high validation accuracy ($>97\%$). The random forest models we generated, for most topologies, was prone to getting stuck in loops (moving back and forth over 2 squares) or running into a wall just like the CNN.

Something that made tuning the CNN and RF models difficult was the fact that both models had decent metrics. They had reasonable accuracy and loss values, but performed poorly on any real time data. A lot of our testing of these models was based on trial and error, which was inefficient.

5.2 Reinforcement Learning Results

The architecture described in section 3.3 was trained over 100,000 episodes, and at the end of it the best model, ranked based on the size of the snake, was selected. Even though the model worked best around episode 30,000, the architecture was trained for the full 100,000 episodes and the model with the greatest score was selected from these 100,000 episodes. The best DQN model trained was able to attain considerably higher scores than the imitation learning

models. On a rolling average of 5 games, the DQN model averaged a snake size of 17.60, surviving for an average of 212.40 steps, with a best score of 27. Often the model's snake would die by running into a wall, or by coiling and hitting its own body. This indicates that the model can be improved further with better tuned hyperparameters.

6. Conclusion

By comparing the results of each model, the average results of the Deep Q-Network implementation were much higher than the results for the imitation learning model. These results show that for this problem domain, Reinforcement Learning models outperform imitation learning models. This general outcome was expected, however, the group did not expect it to be this pronounced. For future projects, the group would consider using different approaches to this problem domain within the realm of classical Machine Learning/Imitation Learning, with more extensive data collection. For the DQN model, network architecture and environment design changes can be made to attempt a higher score in newer models. Also, training this architecture for longer may incur a better outcome in the final model.

6. Bibliography

- Furnkranz, J. (n.d.). *Machine Learning in Games: A Survey*.
- Hussein, A., Gaber, M. M., Elyan, E., & Jayne, C. (2018). Imitation Learning: A Survey of Learning Methods. *ACM Computing Surveys*, 50(2), 1–35. <https://doi.org/10.1145/3054912>
- Kaelbling, L. P., Littman, M. L., & Moore, A. W. (1996). Reinforcement Learning: A Survey. *Journal of Artificial Intelligence Research*, 4, 237–285. <https://doi.org/10.1613/jair.301>

