DQN Agent for Snake Game Report

Intelligent Systems II - Reinforcement Learning

Daniel Emídio
dept. of Electronics, Telecommunications and Informatics
University of Aveiro
Aveiro, Portugal
daniel.emidio@ua.pt

David Palricas
dept. of Electronics, Telecommunications and Informatics
University of Aveiro
Aveiro, Portugal
davidpalricas@ua.pt

Márcio Tavares
dept. of Electronics, Telecommunications and Informatics
University of Aveiro
Aveiro, Portugal
marciotavares@ua.pt

***Abstract*—This report presents the development of a Deep Q-Network (DQN) agent to learn and play a modified Snake game using reinforcement learning. The system uses a client-server architecture with WebSocket communication between the game and the learning agent. The training environment provides the agent with game state information. A reward system encourages food collection and survival while penalizing game termination. The DQN agent uses a neural network that can adapt to different map sizes and employs standard techniques like experience replay and epsilon-greedy exploration. Despite these implementations, the agent showed limited learning performance with consistently low scores and short episode durations. The system serves as a foundation for testing other reinforcement learning algorithms in discrete action environments.**

***Index Terms*—reinforcement learning, Deep Q-Network, TensorFlow Agents**

I. Introduction

The goal of this project was to develop an agent capable of learning to play a modified version of the Snake game using reinforcement learning. This customized Snake game was designed by **Professor Diogo Gomes** as part of a project for the **Artificial Intelligence** and **Intelligent Systems I** modules at **University of Aveiro**, where students were tasked with creating agents employing AI search algorithms to play this game effectively.

II. Architecture

In Fig.1, which shows the architecture of our implementation, we can see the following components:

1) Server.py
   - Function: Responsible for controlling the state of the game.
   - Input: w,a,s,d key to control the snake's movement.
   - Output: The new game state represented in a dictionary after each action.
2) Student.py
   - Function: Orchestration of the training loop
3) snake_train_env.py

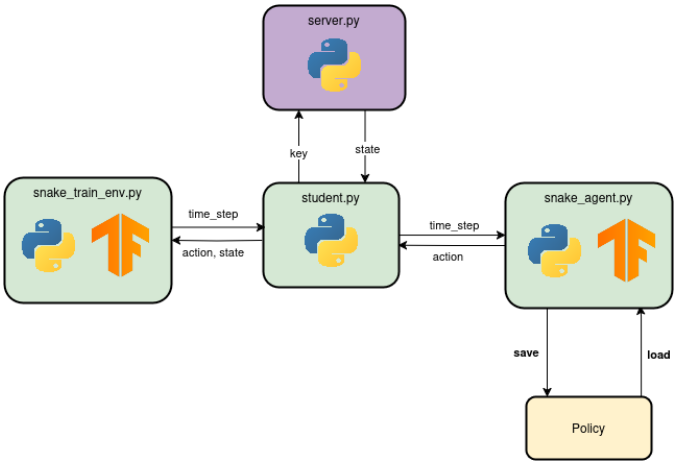

Fig. 1. Project architecture

   - Function: Processes server states and acts as a training environment for the agent
   - Input: The server's state dictionary and the agent's action.
   - Output: A time step consisting of an observation and a reward.
4) snake_agent.py
   - Function: Implements the DQN algorithm and controls the game's actions.
   - Input: The time step sent by the environment.
   - Output: The next key (action).

III. Student

The central component of the architecture that makes the link between server, environment and agent possible.

The program can operate in two different modes: training mode, where the agent learns through successive episodes, and single run mode, where a pre-trained model runs the game

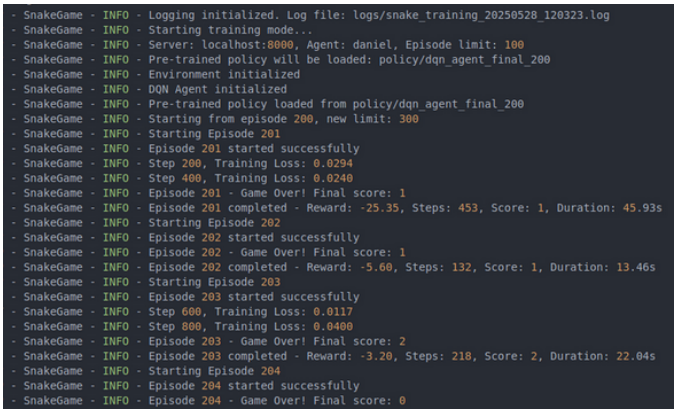

Fig. 2. Log file

without additional learning. Communication with server.py takes place via WebSockets.

The program accepts parameters via command line arguments. The operating mode is determined by the first argument, if it is "train" it starts training mode, otherwise it starts single run mode. If it is in training mode, the maximum number of episodes to train the model must be passed in the second argument, and finally a third argument can be passed to train a model that has already been trained in a previous run by passing its name in the argument.

A. SnakeGame

The SnakeGame class implements a asynchronous communication pattern, using a queue to manage incoming messages and an event system for synchronization. The first state received is stored separately since it is a special state that contains unique information like the inicial state of the map. The WebSocket listener system operates continuously in a separate task, processing messages in a non-blocking way and feeding them to the main queue. This architecture prevents message loss and guarantees orderly processing of game states.

B. Logging

The setup_logging function establishes an event logging system, automatically creating a log directory and configuring multiple handlers. The system generates log files with timestamps and simultaneously maintains output on the console, facilitating both real-time monitoring and subsequent analysis.

C. Training Loop

1) Function play_single_episode: Each training episode follows a structured sequence: resetting the game state, establishing the WebSocket connection, sending the command to join the game and waiting for the first valid state. This initialization ensures that each episode starts with consistent conditions. The episode's main loop processes each step through the sequence: obtaining an action from the agent, sending the action to the server, receiving the new state, and updating the training environment. This sequence continues until the game ends or a communication error occurs.

End-of-game detection is based on the presence of the "highscores" field in the received state, signalling that the episode has ended and the final results of the episode are available.

During each step, the system stores experiences in the DQN agent's buffer and runs training steps periodically. The agent receives structured time_steps from the environment, allowing reinforcement learning algorithms to be applied as standard. The agent's on_episode_end method is called at the end of each episode for final updates.

2) Function agent_loop: The main loop initializes all the necessary components, those being the training environment, DQN agent and logging system. As mentioned before, it supports pre-trained models allowing continuation of previous training, with automatic adjustment of the episode counter and total limit. The system runs episodes sequentially, creating a new WebSocket connection for each one. The model is persisted periodically through automatic backups every 100 episodes and a final save when training is complete. The system generates progress reports, including training statistics, elapsed time and completion estimates, it also keeps track of other statistics like rewards per episode and maximum scores achieved.

3) Function run_policy: The single run mode loads a pre-trained model and runs the game without training components. This separation allows pure evaluation of the agent's performance without learning overhead. The agent only uses the learned policy, without further exploration.

IV. Training Environment

This implementation of a TensorFlow environment provides a reinforcement learning interface for training AI agents to play our modified version of the game Snake. The system accepts four directional actions (up, down, left, right) and returns comprehensive observations. The environment receives game states from an the Snake game server (provided by Student.py), calculates rewards and manages episode lifecycle through the mandatory reset and step operations that conform to TensorFlow Agents specifications.

A. action_spec

The action specification defines the valid moves available to the reinforcement learning agent within the Snake game environment. The system implements a bounded array specification that constrains actions to four discrete directional commands: up (0), down (1), left (2), and right (3). This specification uses numpy's int32 data type and establishes clear boundaries through minimum and maximum values of 0 and 3 respectively. The action_spec method returns this specification to inform the training algorithm about the available action space, ensuring that generated actions fall within valid parameters for Snake game movement.

B. observation_spec

The observation specification establishes a comprehensive multi-component state representation that provides the

agent with all necessary information to make informed decisions. The system defines six distinct observation components through a dictionary structure.

- **map** component represents the game grid as a bounded array with values ranging from 0 to 4, where different integers represent empty spaces, walls, food, snake head, and snake body segments respectively.
- **traverse** indicates if the snake is capable of traversing walls or the edges of the map;
- **direction** tracks the current heading of the snake;
- **range** component specifies the agent's vision distance;
- **timeout** component maintains remaining steps as a bounded value defined by the initial state received;
- **score** counts the points obtained by the agent as an unbounded integer.

C. Reset

The call_reset method initializes the environment for new training episodes by accepting a map dictionary containing the initial game state. This method updates internal dimensions if the game grid size has changed from previous episodes, ensuring compatibility with varying map configurations. The method creates a fresh observation dictionary with proper data types and resets all episode-specific variables including game-over flags, scores, and distance tracking. The system returns a restart time step that signals to the training framework that a new episode has begun with the provided initial state.

D. Step

The call_step method processes individual agent actions within the environment by accepting the current game state and the selected action from the learning algorithm. This method orchestrates the complete step cycle through internal state updates, reward calculations, and termination condition checks. The implementation handles error conditions gracefully by returning termination time steps when exceptions occur. The method distinguishes between transition time steps for continuing episodes and termination time steps for completed games, providing appropriate discount factors and reward signals to guide the training process.

E. State Update Operations

The _update_state method represents the core synchronization mechanism between the reinforcement learning environment and the external Snake game server

The state update process starts with immediate termination condition assessment by examining the presence of highscores data within the received state dictionary. When highscores information exists, the system recognizes this as a definitive game-over signal and sets the internal game termination flag accordingly. This early detection prevents unnecessary processing of subsequent state updates when the episode has concluded.

If a game-over didn't occurred the method begins updating individual scalar components of the observation space using data extracted from the server state including the range, traverse and score components. The timeout component calculates the remaining steps by subtracting the current step count from the total allocated timeout period.

Fig. 3. Reward hierarchy

The sight information processing constitutes one of the most complex aspects of state updates, involving coordinate-based parsing of the agent's environmental visibility. The system iterates through sight data structured as nested dictionaries with coordinates while validating boundaries against the grid dimensions. Object type mapping ensures that detected environmental elements fall within the valid range of zero to four, with unknown objects defaulting to empty space representation.

Finally for the snake body the system first clears existing snake body markers from the current map representation, then systematically places new body segment markers based on the received coordinate list.

F. Reward System

The agent's reward system Fig.3 is designed to encouraging beneficial actions and penalizing destructive ones during its training. The system incorporates specific mechanisms to ensure the agent prioritizes critical objectives while avoiding harmful behaviours. Below is the hierarchy of our reward system, described in ascending order.

- **Game Over Penalty:** A significant penalty (-100.0) is imposed when the game ends, signaling the termination of the episode. This helps the agent recognize and avoid actions leading to its demise.
- **Food Reward:** Eating food is prioritized as the primary objective, with a substantial base reward of 50.0. Additional bonuses are calculated based on the snake's length, rewarding growth and emphasizing the importance of consuming food.
- **Distance-Based Rewards:** The agent earns a small reward (or incurs a minor penalty) based on its proximity to food (fruits).A reduction in the distance to food results in a reward, while increasing distance incurs a minimal

penalty. This encourages the agent to navigate towards the fruits and eat them.

- **Immediate Danger Penalties:** A penalty of -8.0 is applied for actions leading to immediate death, such as collisions with walls or the snake's body. This penalty is selective and avoids discouraging the agent from learning to handle risky situations.
- **Efficiency Penalty:** A tolerance threshold of 200 steps without eating food minimizes harsh penalties for inefficiency. Beyond this limit, the penalty is gradual and capped at -2.0, encouraging exploration while avoiding undue punishment.
- **Survival Reward:** A survival reward of 0.5 per step encourages the agent to stay alive longer, emphasizing the value of longevity in gameplay.

V. DQN Agent

A Deep Q-Network (DQN) agent is used as our reinforcement learning agent. This agent uses **TensorFlow's** capabilities, employing **Keras** to construct a neural network that approximates the Q-value function, enabling it to make decisions based on the current state of the game and expected future rewards. By optimizing training with gradient clipping, **Huber loss** Fig.4, and efficient tensor computations, the agent ensures robust real-time decision-making, aligning with reinforcement learning principles such as Double DQN and experience replay to learn to play this Snake Game.

The agent architecture builds upon the traditional DQN framework, while incorporating some enhancements for improved stability, adaptability, and performance. The agent uses two neural networks Fig.5: the primary Q-network and a target network. The primary Q-network predicts Q-values for all possible actions given the current state, while the target network provides stable target Q-values during training, helping to minimize oscillations and divergence.

The neural network design includes a convolutional neural network (CNN) for processing the visual map representation of the game. The CNN extracts meaningful spatial features, such as obstacles, the snake's body, and the location of the target (food). Additionally, scalar inputs representing game-specific attributes—such as the snake's traversal status, range to food, direction of movement, and timeout threshold—are processed through embedding layers and concatenated with CNN-derived features. The resulting feature vector is passed through several dense layers to compute the Q-values for each action.

A significant feature of this agent is its ability to dynamically adapt to different map dimensions. Upon encountering a new map size, the agent reinitializes its neural networks to accommodate the updated state representation. This dynamic capability ensures compatibility with various game configurations without manual reconfiguration.

The training process employs experience replay, where the agent stores transitions (state, action, reward, next state) in a memory buffer. By sampling batches of experiences during training, the agent avoids correlated updates and improves

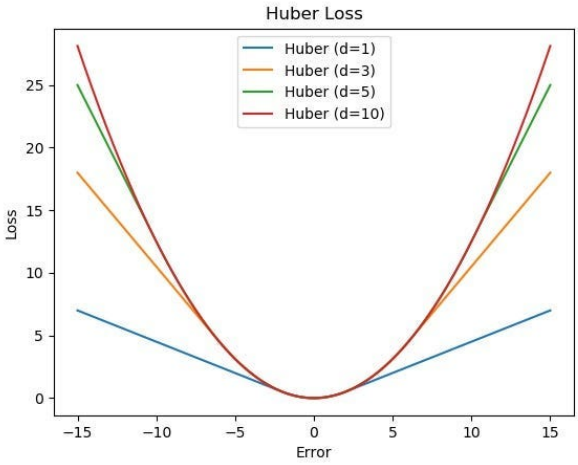

Fig. 4. Huber Loss Function Example

sample efficiency. A Double DQN approach is used to compute target Q-values, reducing overestimation bias. The agent also incorporates gradient clipping and the Huber loss function to enhance training stability, particularly in scenarios with noisy or sparse rewards.

Exploration is managed using an epsilon-greedy strategy, where the agent balances discovering new actions and using known knowledge by probabilistically selecting random actions. Epsilon decays gradually over time to favour learned strategies as the agent becomes proficient. This mechanism ensures robust learning in the early stages while converging toward optimal behaviour in the long run.

To evaluate its performance, the agent conducts periodic evaluation episodes without exploration, assessing its ability to maximize game score and adapt to changing environments. Metrics such as average reward, score, and training loss provide insights into learning progress and overall effectiveness.

In addition to its learning capabilities, the agent supports saving and loading its trained model and configurations. This feature facilitates the resumption of training, transfer learning, and deployment in production environments.

VI. Result Analysis

During development, the **DQN agent** showed limited learning capabilities. After some training iterations, the agent frequently moved in a straight line without collecting food or demonstrating improved behaviour. To address this, we expanded the observation space (providing more information) and redesigned the reward function as described earlier.

Despite these changes, the agent's performance remained poor, as shown in the collected data:

- Fig.6: Reward per episode remained consistently low (typically between -90 and -100), suggesting the agent either failed to collect food or managed to collect just one before dying.

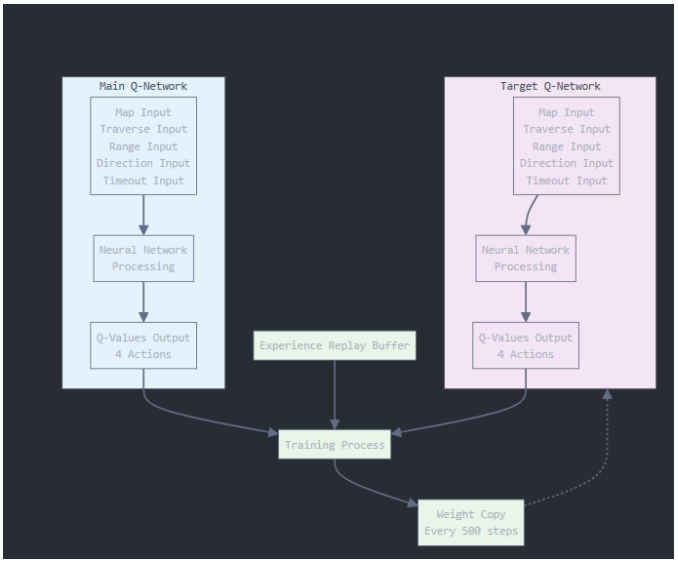

Fig. 5. DQN Agent QNetworks

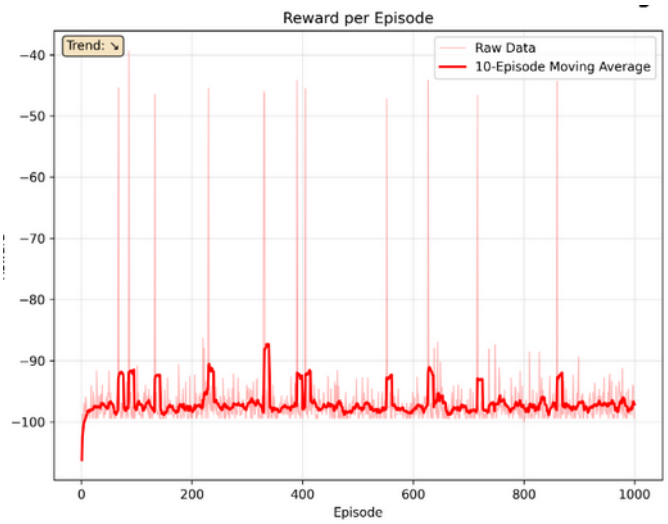

Fig. 6. Reward per Episode

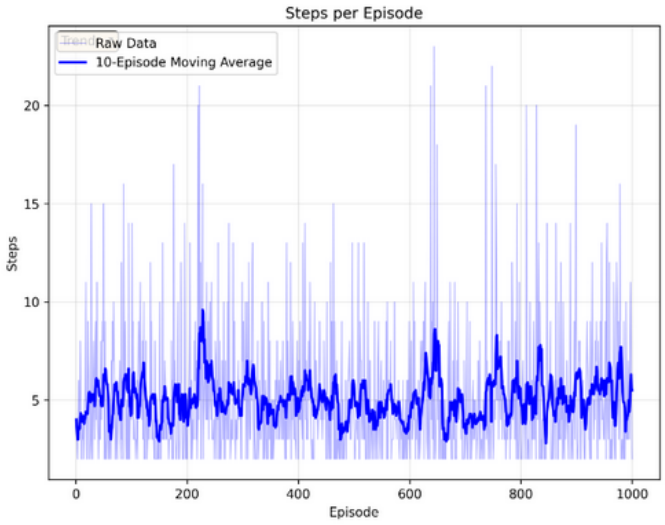

Fig. 7. Steps per Episode

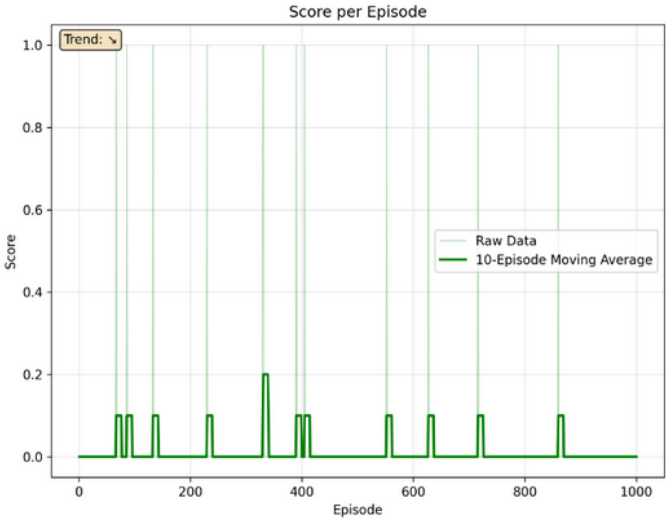

Fig. 8. Score per Episode

- Fig.7: Steps per episode were generally under 10, reinforcing that the agent often died quickly, typically by colliding with itself.
- Fig.8: Scores per episode stayed near zero throughout the training process, with only occasional spikes when the agent collected a single poi

Although we initially suspected the reward function was the main issue, the results suggest the primary bottleneck was likely algorithmic (DQN limitations) rather than environmental.

VII. Challenges and Attempted Improvements

Given DQN's underwhelming performance in our custom environment, we attempted to implement alternative agents supported by TF-Agents, designed for discrete action spaces:

- PPO (Proximal Policy Optimization).
- REINFORCE (Monte Carlo Policy Gradient).
- SAC (Soft Actor-Critic).

To enable these experiments, we created an abstract base class that allow interchangeable agent types within the same training loop. Unfortunately, creating the PPO agent in particular proved more complex than anticipated, and we were unable to get it running successfully. Additionally, the official TF-Agents code examples (notebooks on Google Colab) often failed to work, which made the development more time-consuming and frustrating.

VIII. Conclusion

Although we were not able to achieve high scores with our DQN agent, we implemented a functional custom environment with a client-server architecture, a training script capable of continuing the previous training and a policy runner script

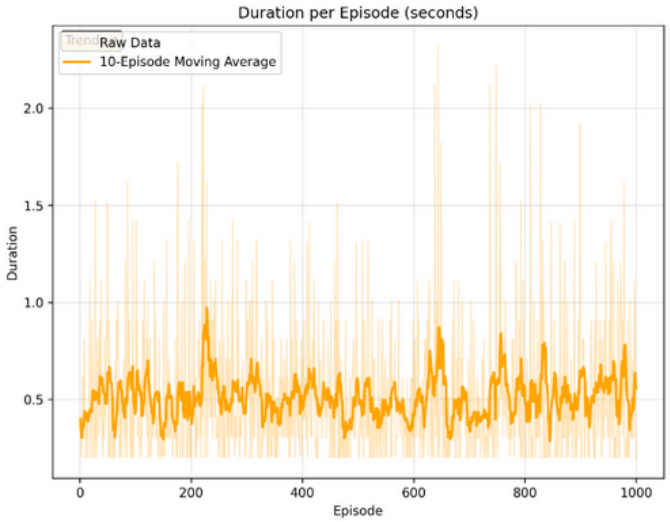

Fig. 9. Duration per Episode

	DQN	DDPG	C51	TD3	PPO	REINFORCE
Discrete	✓		✓		✓	✓
Continuous		✓		✓	✓	

Fig. 10. TensorFlow agent types

that plays a game with the trained agent. We also created the base for experimentation with other agent types, though these implementations proved more challenging than expected.

We identified some possible improvements such as: - Parallelizing training using multiple client-server connections to increase the volume of experiences collected without extending training time. - Exploring alternative agent algorithms (for discrete action spaces).

Finally, our reward structure seemed reasonable, although it could be further refined with aggressive scaling of penalties or rewards.

REFERENCES

[1] TensorFlow, "TensorFlow," TensorFlow, 2019. https://www.tensorflow.org/.
[2] TensorFlow, 2019. https://www.tensorflow.org/agents/
[3] V. Mnih et al., "Human-level Control through Deep Reinforcement Learning," Nature, vol. 518, no. 7540, pp. 529–533, Feb. 2015, doi: https://doi.org/10.1038/nature14236.