Godot DQN Agent Documentation

Agent Overview

Introduction:

A Deep Q-Network (DQN) is a reinforcement learning (RL) algorithm that combines Q-learning with deep neural networks. It learns a policy by approximating the optimal action-value function (Q-function), which predicts the expected cumulative reward for taking a specific action in a given state and following the optimal policy thereafter. Through repeated interactions with the environment, the DQN improves its predictions and learns the best actions to maximize long-term rewards.

Agent Objective:

The objective of the DQN agent is to navigate a 3D environment to reach the power node (goal) while minimizing distance traveled and avoiding boundary penalties. It learns an optimal path through the use of a neural network and q values.

How the Agent Operates:

- Actions: The agent can move in various directions, including forward, backward, left, right, and diagonal combinations, depending on the Q-values predicted by the neural network.
- State Representation: The agent perceives its environment through a normalized state representation, which includes its position relative to the goal, speed, and direction.
- Rewards: The agent is guided by a reward system, receiving positive rewards for reducing the distance to the goal and penalties for moving away or exceeding boundaries. A large reward is given for successfully reaching the goal.

Learning Process:

The agent learns by storing its experiences in a replay buffer and training its neural network on mini-batches of these experiences. This enables the agent to generalize from past interactions, improving its policy over time. Key learning mechanisms include:

- Epsilon-greedy policy: Balances exploration (random actions) with exploitation (selecting the best-known action).
- Target network: Stabilizes learning by providing consistent Q-value targets during training.
- Bellman equation: Updates Q-values based on observed rewards and estimated future rewards.

Agent Implementation:

I initially watched one video on the DQN algorithm to understand what libraries and tools I will need to implement the agent. I identified a neural network library was needed. I searched and experimented with different Godot neural networks until I found one that seemed like it had all the features and functions necessary for the DQN algorithm. After that I tried giving all the code of the game and neural network to ChatGPT, and asked it to implement a DQN agent on the game. The code was very bad but, and I tried endlessly to fix it. After some time I realized there was an extreme lack of understanding I had, and I really did not understand how the DQN algorithm truly worked. So I thought it would be very important to first master the fundamentals of RML and how a DQN works. I repeatedly watched these two videos until I fully grasped the concepts and underlying algorithms.

- https://www.youtube.com/watch?v=EUrWGTCGzlA&t=1019s
- https://www.youtube.com/watch?v=x83WmvbRa2I

After watching and understanding these videos completely, I decided I would restart my code completely and build my agent in steps. Throughout my coding I would constantly check in with ChatGPT, and would ask it for advice or input on how my code looks and if it is following a proper DQN implementation. Often, it would suggest improvements or provide example snippets of code, which I would incorporate into my project. Sometimes I'd copy the code directly and tweak it, adjusting it to fit my project or fixing issues that I encountered during testing. Other times, its answers served as a starting point for brainstorming or refining my understanding of how to proceed.

ChatGPT essentially became my personal tutor, able to provide detailed guidance or clarify doubts. It felt like working alongside an expert in the field. The interaction wasn't just about telling me how DQN works, it really dived into my code, explaining what looks great but also what could be improved.

This iterative learning and development process was extremely helpful. Here is a breakdown of what I slowly implemented overtime:

1. State Representation and Actions:
    - The first step was to define the state space of the agent. ChatGPT suggested I normalized values like distances and velocities to keep them bounded and scaled for the neural network. Actions were mapped to directional movements, such as left, right, up, diagonal, etc.
2. Neural Network Setup:
    - I chose a simple feedforward architecture with ReLU activation and MSE loss. This was because ChatGPT explained how it was best for small state-action spaces.
    - I implemented functionality to synchronize weights between the primary and target networks, this provides stability during training.

3. Reward Function Design:
    - Designing the reward function required multiple iterations. Initially, rewards were very sparse, since the agent only got reward information from picking up the coin or falling off the map. I refined this to include distance-based rewards, penalties for moving away, and bonuses for moving towards the daisy coin.
4. Replay Buffer and Batch Training:
    - To stabilize training, I implemented a replay buffer that stored experiences and sampled mini-batches. Balancing buffer size and batch frequency required experimentation, and I found that a train frequency of 6 steps with a batch size of 64 worked well for this game environment.
5. Epsilon-Greedy Exploration:
    - Starting with a high epsilon (1.0) allowed the agent to explore randomly. Gradual decay ensured a smooth transition to exploitation, which meant the agent was less random and did moves that it learned were the best.
6. Incremental Training and Debugging:
    - I started with frequent updates to the Q-network and observed the agent's behavior in short training episodes by using the replay buffer.
    - Debugging was iterative. I logged intermediate values (e.g., Q-values, rewards, and epsilon) and adjusted hyperparameters based on the agent's performance.
7. Saving and Loading:
    - As the agent became smart and refined, I added functionality to save its parameters and load them for future sessions. This allowed me to resume training without starting from scratch.

So beyond just debugging or explaining concepts, ChatGPT helped me think critically about my code. For example, if I wasn't sure whether my hyperparameters were properly set or the reward function made sense, I could run the idea past it and use its feedback as a foundation for experimentation.

Overtime, I noticed myself asking for help much less often, and I truly started understanding how the DQN agent was working and what needed to be fixed. I believe ChatGPT was extremely useful in developing and completing my DQN agent because it didn't just output code that I just copied and pasted, it was a collaborative partner that allowed me to explore, experiment, and learn without feeling stuck or unsure of where to go next. This approach of coding with an "on-demand tutor" made the entire process smoother, more enjoyable, and very insightful.