Cart Pole Agent

Reinforcement Learning with Deep Q-Networks

Michele Ventimiglia

30/10/2023

1 Introduction

The **Cart Pole problem** presents a challenge where an agent must learn to balance a vertically positioned pole on a moving cart. Although it may seem trivial, the Cart-Pole problem serves as an effective introduction to the field of **Reinforcement Learning**.

In this context, we employ a **Deep Q-Network (DQN)** approach. Unlike traditional Q-learning, which relies on tabular methods, a DQN uses Deep Neural Networks to approximate the Q-values. This allows the agent to generalize its learning across a continuous state space, allowing it to handle the Cart Pole's dynamics.

2 Initialization

2.1 System Configuration, Setup, and Libraries

The experiments and implementations for this project were conducted using the following system, software configurations, and libraries:

- **Programming Environment:** Jupyter Notebook on Microsoft Visual Studio Code;
- **Programming Language:** Python (v3.11.6);
- **Hardware Acceleration:** NVidia GPU, leveraging CUDA for parallel computation;
- **Neural Network Framework:** `PyTorch` (v2.1.0), utilized for constructing and training the Deep Q-Network model;
- **Environment Simulator:** `Gymnasium`, a maintained fork of OpenAI's `Gym` library, is used for simulating the Cart-Pole environment.

Note: The utilization of the NVidia GPU, in conjunction with the CUDA backend, significantly expedited the training and evaluation processes, especially given the computationally intensive nature of deep reinforcement learning tasks.

2.2 Game Environment

To feed an input to the model we generate a Cart Pole **game simulator** through the Gymnasium package. During this step, we choose to show the simulator on the display during the training.

2.2.1 Observation Space

The state of the environment, or the observation, is represented by a 4-dimensional vector. Each dimension corresponds to:

- **Cart Position**: the position of the cart along the track. It can take values between -4.8 and 4.8. However, an episode terminates if the cart leaves the (-2.4, 2.4) range;
- **Cart Velocity**: the speed at which the cart is moving. It can theoretically take any value from negative to positive infinity;
- **Pole Angle**: the angle at which the pole is tilted. It can be observed between approximately -0.418 radians (-24°) and 0.418 radians (24°). However, the episode terminates if the pole angle is not within the range of (-.2095, .2095) radians, or $\pm 12°$;
- **Pole Angular Velocity**: the rate at which the pole's angle is changing. Like cart velocity, it can theoretically range from negative to positive infinity.

2.2.2 Action Space

The agent can take two distinct actions:

- **Push left** (0): applies a force to move the cart in the left direction;
- **Push right** (1): applies a force to move the cart in the right direction.

`N.B.` It's important to note that the velocity influenced by the applied force is not constant. It varies based on the angle at which the pole is pointing. This is because the center of gravity of the pole changes the amount of energy required to move the cart beneath it.

2.2.3 Goal and Termination

The primary goal of the agent is to keep the pole upright for as long as possible, receiving a **reward of +1 for every step taken**, including the termination step. An episode can end prematurely if:

- The pole angle exceeds $\pm 12°$;
- The cart's position surpasses ± 2.4 (meaning the center of the cart reaches the edge of the display);
- The episode length exceeds a certain threshold (500).

In essence, the agent must learn to apply the right amount of force in the correct direction to keep the pole from falling, all while ensuring the cart doesn't move too far off-center.

3 Deep Q-Network

3.1 Architecture

The DQN model architecture consists of three fully connected layers with Leaky ReLU (a variant of the standard ReLU that allows a small gradient when the unit is not active) as activation functions.

The input dimension corresponds to the state space of the environment (which as we have seen in paragraph 2.2.1 is 4) while the output represents the action space, which is 2 (left or right).

3.2 Agent

The `Agent` class encapsulates the behavior and learning mechanisms of the Cart Pole agent. It integrates the functionalities of the Deep Q-Network (DQN) with the reinforcement learning environment.

3.2.1 Initialization

When an Agent object is instantiated, several parameters and components are initialized:

- **Q-Network**: An instance of the DQN model is created. This Neural Network approximates the Q-values for given state-action pairs.
- **Optimizer**: An optimizer is used to adjust the weights of the Q-network during training based on the loss;
- **Memory**: A Deque is used as a memory buffer to store transitions. This memory is used for experience replay, a technique that randomly samples previous experiences to break the correlation between sequential experiences and stabilize training;

- **Epsilon**: This is the exploration rate. Initially set to a high value, it determines the likelihood of the agent taking a random action versus choosing the action with the highest Q-value. Over time, this value decays, making the agent exploit its learned knowledge more and explore less.

3.2.2 Action Selection

The agent uses this method to decide which action to take based on the current state:

- **Exploration**: the agent takes a random action with probability `epsilon`;
- **Exploitation**: the agent feeds the current state into the Q-network and chooses the action with the highest predicted Q-value.

3.2.3 Storing Transitions

After taking an action in the environment, the agent observes a reward and the next state. This method allows the agent to store the transition (state, action, reward, next state, done) in its memory.

3.2.4 Training

This method is where the learning happens:

1. If there are enough transitions in memory, **a random batch of transitions is sampled**;
2. For each transition in the batch, the agent **calculates the current Q-value** (using the Q-network) **and the target Q-value** (based on the observed reward and the maximum Q-value of the next state);
3. **The loss is computed** between the current and target Q-values;
4. **The optimizer adjusts the Q-Network's weights** based on this loss;
5. The exploration rate **epsilon is decayed over time**.

4 Model Training

4.1 Early Stopping

A custom early-stopping mechanism is implemented to halt training if the model's performance does not improve for a specified number of episodes and store the best values.

4.2 Learning Rate Scheduler

A good practice is to implement a learning rate scheduler that fine-tunes the learning rate during training. Instead of a constant rate, the scheduler adjusts it based on model performance or epoch count. We employed the `ReduceLROnPlateau` scheduler, which reduces the rate when a metric has stopped improving, ensuring efficient convergence.

4.3 Hyperparameters

We used the following hyperparameters for the Cart Pole Agent:

- **GAMMA (Discount Factor)**: Represents the agent's consideration for future rewards. A value close to 1 makes the agent prioritize long-term reward over short-term reward. We set the value to **0.99**;
- **EPSILON_START (Initial Exploration Rate)**: The initial probability of the agent taking a random action. It allows the agent to explore the environment at the beginning of training. We set the value to **1.0**;
- **EPSILON_END (Minimum Exploration Rate)**: The minimum value to which the exploration rate can decay. It ensures that the agent always has some probability of exploring the environment. We set the value to **0.01**;
- **EPSILON_DECAY**: The factor by which the exploration rate is reduced after each episode. This allows the agent to gradually shift from exploration to exploitation. We set the value to **0.995**;
- **LEARNING_RATE**: Determines the step size at each iteration while moving towards a minimum of the loss function. A smaller value makes the optimization more robust, but it requires more iterations. We set the value to 0.001;
- **BATCH_SIZE**: The number of training examples utilized in one iteration. It affects the speed and stability of the training process. We set the value to **64**;
- **MEMORY_SIZE**: The maximum number of transitions (state, action, reward, next state) the agent can store for experience replay. It helps in breaking the correlation between sequential experiences and stabilizing training. We set the value to **10000**;
- **LOSS FUNCTION**: The criterion used to evaluate the discrepancy between the predicted Q-values and the target Q-values. In our project, we employed **MSE** (Mean Squared Error), commonly used for DQN;
- **OPTIMIZER**: We used the **Adam** optimizer, as proposed by (Kingma et al., 2014), to adjust the model's weights based on the computed loss.

4.4 Training Loop

The training loop is the core of the reinforcement learning process where the agent learns to interact with the environment over a series of episodes. Each episode represents a complete interaction from the initial state to a terminal state. Here's a breakdown of the steps within each episode:

1. **Environment Reset**: At the start of each episode, the environment is reset to its initial state. This provides a fresh start for the agent to interact with the environment.

2. **Action Selection**: For each state, the agent decides on an action to take. This decision is based on the current policy, a combination of exploration and exploitation;

3. **Execute Action**: The chosen action is executed in the environment, leading the agent to a new state and receiving a reward for the action;

4. **Store Transition**: The agent stores the transition, which includes the current state, action taken, reward received, the resulting next state, and whether the episode has ended (done flag). This stored data is crucial for the experience replay mechanism, where the agent later samples from these stored transitions to update its Q-network;

5. **Q-Network Update**: The agent samples a batch of transitions from its memory and uses them to update the Q-network. The update aims to minimize the difference between the predicted Q-values and the target Q-values. The target Q-values are computed based on the observed rewards and the maximum Q-values of the next states;

6. **Episode Termination**: If the agent reaches a terminal state or if the maximum number of steps for an episode is exceeded, the episode ends. The agent then proceeds to the next episode, starting with the environment reset;

7. **Epsilon Decay**: After each episode, the exploration rate (epsilon) is decayed, reducing the probability of taking random actions and increasing the likelihood of exploiting the learned Q-values. This ensures a balance between exploration and exploitation as training progresses.

The training loop continues until a specified number of episodes is reached or if an early stopping condition, based on the agent's performance, is met.

5 Evaluation

As depicted in Figure 1, the model improves with successive epochs. With the decay of epsilon, there's a notable shift from exploration to exploitation strategies. This optimization culminates at **epoch 129**, achieving a **peak reward score of 8197**.

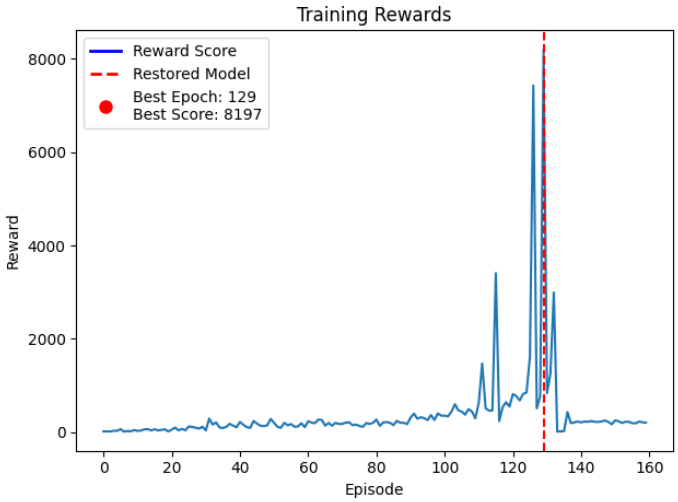

Figure 1: Training Rewards

6 Testing

After restoring the optimal weights, the trained agent was tested in a new game environment. **The agent exhibited superior performance over an extended period**, prompting a manual termination of the test.

7 Conclusion

The Cart Pole Agent, harnessing the power of a Deep Q-Network, has effectively demonstrated its capability in mastering the intricate challenge of pole balancing. This achievement not only highlights the potential and adaptability of Reinforcement Learning techniques but also sets a foundational precedent for more complex applications in the future.