Assignment 3 RL

Ettore Branca

1965733

1 Theory

Step 1: Compute Value Function

$$Q_w(s_0, a_0 = 1) = w_1^T x(s_0) = \begin{pmatrix} 1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = 0$$

$$Q_w(s_1, a_1 = 0) = w_0^T x(s_1) = \begin{pmatrix} 0.5 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = 0.5$$

Step 2: Compute Temporal Difference Error

$$\delta = r_1 + \gamma Q_w(s_1, a_1) - Q_w(s_0, a_0) = 1 + 0.8 \times 0.5 - 0 = 1.4$$

Step 3: Update Value Function Parameters

$$w_1 \leftarrow w_1 + \alpha_w \delta x(s_0)$$

$$w_1 \leftarrow \begin{pmatrix} 1 \\ 0 \end{pmatrix} + 0.5 \times 1.4 \times \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0.7 \end{pmatrix}$$

Step 4: Update Policy Parameters

$$\theta \leftarrow \theta + \alpha_\theta \nabla_\theta \log \pi_\theta(a = 1|s_0) Q_w(s_0, a_0 = 1)$$

Since $Q_w(s_0, a_0 = 1) = 0$, the update does not change θ.

New Values

$$w_0 = \begin{pmatrix} 0.5 \\ 0 \end{pmatrix}$$

$$w_1 = \begin{pmatrix} 1 \\ 0.7 \end{pmatrix}$$

$$\theta = \begin{pmatrix} 0.6 \\ 1 \end{pmatrix}$$

2 Practice

Double DQN with Prioritized Experience Replay (PER)

In this project, we implemented a reinforcement learning agent to solve the CarRacing-v2 environment (with discrete action space) using concepts from the paper "Prioritized Experience Replay" by Tom Schaul et al. (2015). The key focus is on incorporating Prioritized Experience Replay (PER) to enhance the learning process and employing Double Deep Q-Network (DDQN) to mitigate overestimation bias in Q-values.
The choice of a discrete action space is due to the fact that DDQN is an algorithm designed to handle only discrete action spaces: it is based on the calculation of the $Q(s, a)$ function, where for each state s, a discrete and finite set of possible actions ($a \in A$) is evaluated.

Q-Network Architecture

The input to the network consists of 96x96 RGB images, representing the state of the environment. Three convolutional layers (with batch normalization to stabilize training and improve convergence) are used to process the visual input; they use varying kernel sizes and strides to extract hierarchical features from the raw image.
Then, the output from the convolutional layers is flattened and passed through three fully connected layers, which ultimately output the Q-values for each possible action. ReLU is used as the activation function to introduce non-linearity into the network.

Prioritized Replay Buffer

Experiences are assigned priorities based on their Temporal Difference error, with higher error experiences (more surprising or important ones) more likely to be sampled. To correct for the bias introduced by prioritized sampling, importance sampling weights are calculated and used during the update step.

- **Experience Storage:** experiences are stored as tuples (state, action, reward, next_state, done). The buffer uses a `deque` for efficient storage and retrieval, with a maximum size of 300000 samples (the large buffer size helps store a diverse set of experiences, improving learning stability and reducing the effect of early random experiences).
- **Prioritization:** a parallel `deque` stores priorities for each experience. New experiences are assigned the maximum priority to ensure immediate sampling.
- **Sampling Mechanism:** sampling probabilities are computed using priorities raised to the power of α, controlling the level of prioritization.

- **Importance Sampling (IS) Weights:** they correct the bias introduced by prioritization, ensuring unbiased learning updates. The weights are dynamically adjusted based on a parameter β, which increases linearly from 0.1 to 1.0 over training.

$$w_j = \frac{(N \cdot P(j))^{-\beta}}{\max_i w_i}$$

- **Priority Updates:** after learning, priorities are updated using the TD error.

Policy

The core of this implementation is Double DQN, which mitigates the overestimation bias commonly found in Q-learning by using two networks: the local Q-network, used to select the best next actions, and the target Q-network, to compute the Q-value targets.

$$Q_{\text{target}} = r + \gamma Q_{\text{target}}(s', \arg\max_a Q_{\text{local}}(s', a)).$$

During training, for action selection, an epsilon-greedy strategy is used, where with probability epsilon, a random action is chosen and, with probability $1 - \epsilon$, the one with the highest Q-value. Epsilon decays over time to shift from exploration to exploitation (and is set equal to 0 before evaluation).
The agent learns by interacting with the environment and storing experiences in the PER buffer. Once enough experiences are collected, the agent samples a mini-batch from the buffer and updates the Q-network via TD-learning.
The TD-error is used to compute the weighted Mean Squared Error (MSE) loss and the gradients are backpropagated to update the network weights.
Finally, the target network is "softly" updated towards the local network: instead of directly copying the weights from the local network (like in the original paper), a weighted average of the local and target network weights is used, promoting stability in learning.

$$\theta_{\text{target}} \leftarrow \tau\theta_{\text{local}} + (1 - \tau)\theta_{\text{target}}.$$

Hyperparameters of the final model

- Number of episodes for training: 250
- Maximum steps per episode: 1000
- Batch size: 64, commonly used and provides a good balance between computational efficiency and learning quality.

- Priority coefficient (alpha): 0.6, it controls how much priorities influence the experience sampling (common choice for DDQN with PER).
- Initial exploration coefficient (epsilon_start): 1.0
  It encourages maximum exploration during the early training phase.
- Final exploration coefficient (epsilon_end): 0.01
  It represents the minimal exploration rate after decaying epsilon.
- Epsilon decay: 0.99
  The rate at which epsilon decays. A slower decay allows for more exploration before shifting to exploitation.
- Bias correction coefficient start (beta_start): 0.1
  Initially, a low beta value encourages more exploration by prioritizing random experiences more.
- Bias correction coefficient end (beta_end): 1.0
  As the agent learns, beta increases to 1.0, minimizing bias from the priority sampling mechanism.
- Beta change duration (beta_frames): 250000
  Total number of frames over which beta will increase from 0.1 to 1.0, helping balance exploration and exploitation during training (calculated multiplying the number of episodes by the maximum number of steps each).
- Soft update parameter (tau): 0.005, a common value for tau, it ensures that the target network updates slowly towards the local network.
- Discount factor (gamma): 0.99, used to calculate the value of future rewards.

Results of the final model

With these values as hyperparameters, we initiated a training session consisting of 250 episodes, which yielded pretty good results. Although the trend of the training scores displayes some instability, it ultimately converges to notably high values of reward. So, considering the relatively low amount of computational resources, we can consider the performance of our model highly satisfactory. Next, the scores achieved over the last 10 test runs are reported:
561, 142, 528, 361, 590, -3, 325, 414, -38, 765.