Homework 5

ELEN E6885: Reinforcement Learning

December 4, 2019

Problem 1 (**DQN Algorithm, 20 credits**): Explain why we use "experience replay" and "fixed Q-targets" in DQN. In particular, explain why using "experience replay" and "fixed Q-targets" can help stabilize DQN algorithm when the correlations present in the sequence of observations (e.g., Atari games)?

The answer can be found in the paper "Human-level control through deep reinforcement learning" page 1, paragraph 3:

"Reinforcement learning is known to be unstable or even to diverge when a nonlinear function approximator such as a neural network is used to represent the action-value (also known as Q) function. This instability has several causes: the correlations present in the sequence of observations, the fact that small updates to Q may significantly change the policy and therefore change the data distribution, and the correlations between the action-values (Q) and the target values $r + \gamma \max_a' Q(s', a')$. [10 pts]

We address these instabilities with a novel variant of Q-learning, which uses two key ideas. First, we used a biologically inspired mechanism termed experience replay that randomizes over the data, thereby removing correlations in the observation sequence and smoothing over changes in the data distribution. Second, we used an iterative update that adjusts the action-values (Q) towards target values that are only periodically updated, thereby reducing correlations with the target." [10 pts]

Problem 2 (**DDPG,10 Credits**) Recall that DQN is able to stabilize the process of using deep learning for value function approximation, which was believed to be unstable. When it comes to the continuous action space, it is not straightforward to apply DQN directly. One naive approach is to discretize the continuous action space. However, this may not help in practice, why? How does DDPG handle this problem?

It would cause the curse of dimensionality. In DQN, we need to choose the action that maximizes the Q value at every time step. But maximizing the Q value over each action is computationally expensive when the action space is too large. Mostly this maximization is too complicated to achieve. [5 pts]

To handle the problem, DDPG combines DPG with DQN. DDPG is a model-free, off-policy actor-critic algorithm using deep learning for function approximations and thus is capable of learning continuous actions. [5 pts]

Problem 3 **(A3C, 15 Credits)** What is the benefit of using multiple agents in an asynchronous manner in A3C?

The use of multiple agents has several advantages. [5 pts] First, it helps to stabilize the training. Since each agent has its own copy of the environment, agents are allowed to explore different parts of the environment as well as to use different policies at the same time. In other words, different agents will likely experience different states and transitions. Therefore, when agents update the global parameters with their local parameters in an asynchronous manner, the global parameters update will be less correlated than using a single agent. [5 pts] Second, the nature of multi-threads in A3C indicates that A3C needs much less memory to store experience, i.e., no need to store the samples for experience replay as that used in DQN. [5 pts] Furthermore, the practical advantages of A3C is that it allows training on a multi-core CPU rather than GPU. When applied to a variety of Atari games, for instance, agents achieve a better result with asynchronous methods, while using far less resource than these needed on GPU.

Problem 4 **(Gaussian Policy, 25 Credits)**
Assume Gaussian policy is used in the policy gradient reinforcement learning. The Gaussian mean is a linear combination of state features

$$\mu(s) = \phi(s)^T w = \sum_i \phi_i(s) w_i,$$

where $\phi(\cdot)$ represents the vector of feature functions and w represents the weight vector. Also, assume a fixed variance σ^2. Show that that the score function $\nabla_w \log \pi_w(a|s,w)$ of Gaussian policy is given by

$$\nabla_w \log \pi_w(a|s,w) = \frac{(a-\mu(s))\phi(s)}{\sigma^2}\,.$$

Proof. From

$$\pi_w(a|s,w) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(a-\phi(s)^T w)^2}{2\sigma^2}},$$

we have

$$\begin{aligned}
\nabla_w \log \pi_w(a|s,w) &= \nabla_w (\log \frac{1}{\sqrt{2\pi\sigma^2}} - \frac{(a-\phi(s)^T w)^2}{2\sigma^2}) && [5 \text{ pts}] \\
&= -\frac{2(a-\phi(s)^T w)(-\phi(s))}{2\sigma^2} && [10 \text{ pts}] \\
&= \frac{(a-\phi(s)^T w)\phi(s)}{\sigma^2} \\
&= \frac{(a-\mu(s))\phi(s)}{\sigma^2}\,. && [10 \text{ pts}]
\end{aligned}$$

Problem 5 **(MCTS, 30 Credits)**
In this problem, you are asked to review AlphaGo paper "Mastering the game of Go with deep neural networks and tree search" and AlphaGo Zero paper "Mastering the game of Go without human knowledge". After the paper review, you need to

(1) Summarize the RL algorithms used in these papers. Especially, point out the differences between AlphaGo and AlphaGo Zero search algorithm.

(2) AlphaGo Zero is self-trained without the human domain knowledge and no pre-training with human games. Explain in detail how AlphaGo Zero achieves self-training.

Solution.

(1) Alpha Go takes advantage of **two** deep neural networks: a policy network that outputs move probabilities and a value network that outputs a position evaluation. The policy network is trained initially by **supervised learning** to accurately predict human expert moves, and is subsequently refined by **policy-gradient reinforcement learning**. Once trained, the two networks are combined, followed a **Monte Carlo tree search** to provide a lookahead search using the policy network to narrow down the search to high probability moves, and using the value network to evaluate positions in the tree. [10 pts]

AlphaGo Zero on the other hand, is trained **solely by self-play reinforcement learning** without any supervision or use of human data, especially in **input features**. It uses **one** neural network instead of two that predicts the policy and the value at the same time. It also uses a simpler tree search to evaluate positions and sample moves **without performing any Monte Carlo rollouts**. [10 pts]

(2) In AlphaGo Zero, a neural network is trained by a self-play reinforcement learning algorithm that uses Monte Carlo tree search to play each move. First, the neural network is initialized to random weights. At each subsequent iteration, games of **self-play** are generated. At each time-step, a tree search is executed using **the previous iteration of neural network** and a move is played by **sampling** the search probabilities. After a game terminates, the game is **scored** to give a final reward. The data for each time-step is stored as a tuple of board position, move probabilities, and the game winner from the perspective of the player at that time. In parallel, new network parameters are trained from data **sampled uniformly among all time-steps of the last iteration(s) of self-play**. The neural network is adjusted to **minimize** the error between the predicted value and the self-play winner, and to **maximize** the similarity of the neural network move probabilities to the search probabilities. [10 pts]