THE UNIVERSITY of TULSA
Tandy School of Computer Science

THE UNIVERSITY OF TULSA

Project 6, Reinforcement Learning

CS 5333/7333 Machine Learning

In the Machine Learning course, your projects require you to write and run Python code. You will implement or modify one or more learning algorithms, train models with multiple datasets, test the accuracy of your models, and produce graphs and tables showing the performance of your models.

You will be given a code shell to start with. In the code, you will be instructed to import only certain Python libraries. Importing a library that implements an algorithm and training with a model from that library does not satisfy the requirement of implementing an algorithm. Please reach out if you are unsure about whether you may use a library in part of your algorithm. Use the code shell as a guide, but feel free to restructure the code as you prefer. You may discuss algorithms with other students or chatbots, but it is up to you to write the code and the report.

Overview

First, you will write several algorithms for learning by experience from stateless environments. These problems are called Multi-armed Bandit problems because a common metaphor for describing them involves repeatedly choosing to play one of multiple slots machines (one-armed bandits).

Second, you will complete the implementation of algorithms for learning by experience from small tabular environments without deep learning. Specifically, you will implement the on-policy SARSA algorithm and the off-policy Q-learning algorithm in the stochastic Wumpus World environment.

1 Multi-armed Bandit Problems

Implement three algorithms for for Multi-armed Bandit learners. A code shell for each is provided.

The ϵ-greedy algorithm always selects an action uniformly at random with probability ϵ, and otherwise greedily samples the highest measured mean. It is described on Page 10 of these slides.

The Thompson sampling algorithm is provided on Page 4 as Algorithm 2 in this paper.

The Upper Confidence Bound algorithm is provided on Page 17 of these slides.

For each algorithm, record a sample for each action before beginning the main loop.

1.1 Multi-armed Bandit Environment

The provided simulator is an altered version of this simulator, and it produces environments with actions that provide rewards from clipped Gaussian distributions. If the environment has n actions, the means increase linearly, with shared standard deviation $\sigma = 0.1$. All the rewards fall within $r \in [0, 1]$, which is a requirement for the Thompson Sampling algorithm.

1.2 Analysis

Test each algorithm using arms $\in \{5, 10, 20\}$ across horizons of length 500 times the number of arms (500*arms).

For ϵ-greedy, vary ϵ over $\{0.1, 0.2, 0.3\}$. Compare to the UCB algorithm and Thompson sampling algorithm.

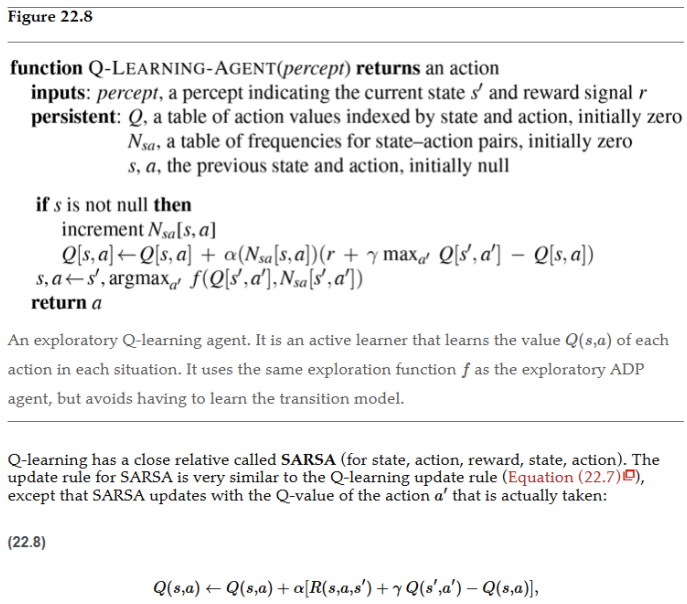

Figure 22.8

function Q-LEARNING-AGENT(*percept*) **returns** an action
  inputs: *percept*, a percept indicating the current state s' and reward signal r
  persistent: Q, a table of action values indexed by state and action, initially zero
    N_{sa}, a table of frequencies for state–action pairs, initially zero
    s, a, the previous state and action, initially null

  if s is not null **then**
    increment $N_{sa}[s,a]$
    $Q[s,a] \leftarrow Q[s,a] + \alpha(N_{sa}[s,a])(r + \gamma \max_{a'} Q[s',a'] - Q[s,a])$
  $s, a \leftarrow s', \text{argmax}_{a'} f(Q[s',a'], N_{sa}[s',a'])$
  return a

An exploratory Q-learning agent. It is an active learner that learns the value $Q(s,a)$ of each action in each situation. It uses the same exploration function f as the exploratory ADP agent, but avoids having to learn the transition model.

Q-learning has a close relative called **SARSA** (for state, action, reward, state, action). The update rule for SARSA is very similar to the Q-learning update rule (Equation (22.7)), except that SARSA updates with the Q-value of the action a' that is actually taken:

(22.8)

$$Q(s,a) \leftarrow Q(s,a) + \alpha[R(s,a,s') + \gamma\, Q(s',a') - Q(s,a)],$$

Figure 1: The Q-learning algorithm, with the alternative SARSA value update rule below.

Compare the results of each algorithm in terms of regret, using the value of arm with the highest mean (1.0). Plot the regret and the estimated best action by the different algorithms over the course of the algorithm.

1.3 Report

Submit code and a write-up of your analysis, including results, plots, and the performance differences between the algorithms.

- **30pts** - Analysis and plots for each algorithm is worth 10 points.

2 Tabular Reinforcement Learning

Complete the implementation for Q-learning and SARSA (Chapter 22, page 802-803, of textbook), exploring reinforcement learning agents that use direct utility estimation. Evaluate and compare their performance in the following three Wumpus World navigation Environments.

2.1 Wumpus World Environment

The provided Wumpus World simulator, a fixed version of this simulator, is a grid environment with grid states and stochastic actions. Test the agents in each of the following tabular environments.

- This 3x4 world described in Chapter 17 of :

Empty	Empty	Gold
Pit	Wumpus	Empty
Empty	Empty	Empty
Empty	Empty	Empty

- A 10x10 world variant with no obstacles and a +1 reward at (10,10).
- A 10x10 world variant with no obstacles and a +1 reward at (5,5).

2.2 Analysis

Train each algorithm in each environment for 1000 episodes, using a discount rate of $\gamma = 0.9$ and a learning rate of $\alpha = \frac{1}{n_{s,a}}$, the inverse of the number of times the action has previously been taken from this state.

2.3 Report

Analyze the rate of convergence of the algorithms, and include color-coded policies in your report.

- **30pts** - Each algorithm on each environment is worth 5 points.

3 Deep Reinforcement Learning

In this section you will implement and experiment with Delayed Deep Q-learning and REINFORCE-MENT / Vanilla Policy Gradient with Baseline to solve the "cartpole" environment with a continuous state space and discrete actions. You will also use a continuous action modification of the Vanilla Policy Gradient algorithm to solve the inverted pendulum continuous action environment. The notebook is provided in the form of a shared link and as a file in the project assignment.

Environment

The main environment we will be solving is the "CartPole-v1" problem. A brief description of the environment could be found at:

https://gymnasium.farama.org/environments/classic_control/cart_pole/

the state space is 4-dimensional and continuous, while the action space is 1-dimensional and discrete, with two actions: 0 and 1 to push the cart left and right. You will use artificial neural networks to learn the Q-values for state-action pairs. The maximum number of steps in this game is 500.

In Part 2, you will also solve the continuous action domain Inverted Pendulum seen here:

https://gymnasium.farama.org/environments/classic_control/pendulum/

Part 1 Deep Q Learning

Provided for you is a notebook which can be run either locally or on Google Colaboratory which implements Deep Q learning Via Temporal difference error. This means that we are trying to generate a network which can predict the remaining expected reward in an episode given a state action pair $Q(s, a) = E|R|$. We will use this Q function to choose the action which maximizes $Q(s, a)$ as our policy. This function will be learned by trying to minimize temporal difference error $Error = [r + (1 - d) * \gamma * max_{a'} Q(s', a')] - Q(s, a)$. Here γ is the discount factor for future rewards,

d is whether this state s' was terminal (done), s' is the next state that resulted from taking action a, and a' is the best next action. Updating Q means that our target $Q(s', a')$ will also change, so the learning is rather unstable. **Implement the Double Deep Q learning modification to the given Q learner by following the comments marked "TODO". Run the experiment 3 times with Deep Q and Double Deep Q learners and record the results.**

Part 2 REINFORCE / Vanilla Policy Gradient

As an alternative to Deep Q Learning, we can learn a policy, $\pi(s)$, directly instead of learning a value function and then taking the action with the best value. This is done using the Policy Gradient Theorem, but the algorithm learns on-policy, which means that it can't look at previous episodes after updating like DQN can. It also relies on an entire episode to be played before training so that the discounted returns for an episode can be sampled. This can make learning noisy and the algorithm can converge too quickly or collapse. Part two of this assignment is to modify the implementation of Vanilla Policy Gradients in two ways.

a.

Add a Value Baseline, V, for the value of each state $V(s)$. Use this function to calculate advantage $Adv = returns(s) - V(s)$ where the value function is trained with mean squared error with the *returns* variable. $Error = \text{returns}[s] - V(s)$. The baseline will make the reward mean zero but with lower magnitude than raw G. This should make learning faster and more consistent, but normalizing returns (mean 0 stdv 1) will amplify this effect. **Report performance with and without Value baseline**

b.

Using the equations provided in comments, transition VPG to a continuous output where *pi* outputs a Gaussian distribution instead of a multinomial distribution. The Loss is equivalent but with a different method for calculating log likelihood and taking actions. **Try this method on the inverted pendulum environment, or use** $a >= 0$ **on the CartPole environment to discretize the action for testing purposes. Report the performance on CartPole and Inverted Pendulum with continuous actions.**

3.1 Report

3.1.1 DQN - 20pts

1. Standard DQN with target_every = 1: 3 runs **- 6pts**
2. Delayed Double DQN by target_every = 32: 3 runs **- 9pts**
3. Soft Delayed Double DQN by Polyak Average tau = 0.01 and target_every = 1: 1 run **- 5pts**

3.1.2 VPG - 20pts

1. VPG trained on G: 3 runs **- 5pts**
2. VPG with Actor trained on advantage $A = G - V$ instead of G: 3 runs **- 5pts**
3. VPG with Advantage and 3 epsilons $0.0, 0.01, 0.1$: 1 run each **- 5pts**
4. Continuous VPG with Advantage on Pendulum: 3 runs **- 5pts**