2021 Fall STATSM231A/CSM276A: Pattern Recognition and Machine Learning

Deniz Orkun Eren
Department of Electrical and Computer Engineering
University of California, Los Angeles
Engineering IV Building, Los Angeles, California 90095
deren@g.ucla.edu

Abstract

In this course, you will learn multiple algorithms in machine learning. In general, for each homework, you are required to write the corresponding codes. We will provide a structure of code and a detailed tutorial online. Your job is (1) to deploy and understand the code. (2) Run the code and output reasonable results. (3) Make ablation study and parameter comparison. (4) Write the report with algorithms, experiment results and conclusions. For detailed instructions, please check the specification of each homework.

Homework submission forms

You have to submit both code and reports. For the code, zip everything into a single zip file. For the report, use this latex template. When submitting, no need to submit the tex file. Submit one single pdf file only.

1 Coding part: DQN and Policy Gradient

1.1 Problems

In this homework, we are asked to use two different reinforcement learning methods to solve certain toy problems in the open-ai gym environment. The first method that we use is called Deep Q-Network (DQN) and the second method is called the Policy Gradient method. We use these two methods to first solve the Cartpole environment. We then use the same training methods to solve the Lunar_Lander environment.

1.2 Introduction: Reinforcement Learning

Before moving further into the details of both methods, it is important to define what reinforcement learning is and how it is used.

Reinforcement learning is a machine learning method that aims to teach an agent how to best navigate a foreign environment. This is accomplished by giving feedback to the agent in terms of reward signals as the agent interacts with the environment. Unlike supervised learning, the agent in reinforcement learning is not given the most optimal set of actions as reference. Instead, the agent relies on its own past experiences to update how it will navigate the environment. This also implies that there has to be a lot of trial and error before the agent determines the optimal policy.

The agent interacts with the environment as follows: at each time step, the agent is said to be in a certain state. From this state, the agent can choose to perform an action that will take it to a new state and the agent receives a reward from the environment based on the action it took.

There are two important concepts that must be mentioned when discussing reinforcement learning: **Exploration** and **Exploitation**. Exploration refers to the agent's tendency to try different actions than the ones it has previously performed to see the change in reward and exploitation refers to the agent's tendency to stick with an action that it has observed to perform well. Balancing these two tendencies is required to reach an optimal solution.

Now that we have a better understanding of reinforcement learning, let us explore the details of the two methods used in the homework.

1.3 Q-learning

Q-learning is a type of reinforcement learning algorithm that aims to learn the value of an action at a particular state. The vanilla q-learning algorithm does not depend on a model of the environment, which is why it is called a model-free algorithm. It was introduced by Prof. Chris Watkins in his PhD thesis titled "Learning from Delayed Rewards".

The vanilla Q-learning algorithm works by storing a table that maps all the states and action combinations with their corresponding Quality (Q) value. This table is referred to as the Q-table. During training, the agent chooses an action to change its current state and observes a reward. After this, the corresponding Q value stored in the Q-table is updated using the Bellman equation, which is defined as follows:

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\Big(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \Big)}^{\text{temporal difference}}$$
$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\text{new value (temporal difference target)}}$$

where a_t is the action taken at time t, s_t is the state the agent is in at time t, r_t is the reward observed at time t, α is the learning rate and γ is the discount factor. The learning rate determines how much the observed reward affects the new state and the gamma value determines the importance of future reward. A low gamma will make the agent only consider immediate rewards while a high gamma will make it look for long-term rewards.

The action that the model chooses to update the Q values is determined by an ϵ greedy algorithm. With probability ϵ, the agent chooses a random action from all possible actions. On the other hand with probability $1 - \epsilon$, the model chooses the action that has the maximum Q value. This approach is used to balance exploration and exploitation during training.

This process repeats until the agent reaches a terminal state. This concludes an episode. Training goes on until the episode limit is reached. In summary, the Q-learning algorithm can be summarised as follows:

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
    $S \leftarrow S'$;
  until S is terminal

However, in the homework tutorial, we implement a variation of Q-learning called Deep Q-learning. When using Q-learning for complex environments with many states and actions, it is impractical to store a Q-table for each combination. Instead of that, we utilise a neural network to generate the Q values for every action when given a state as input.,

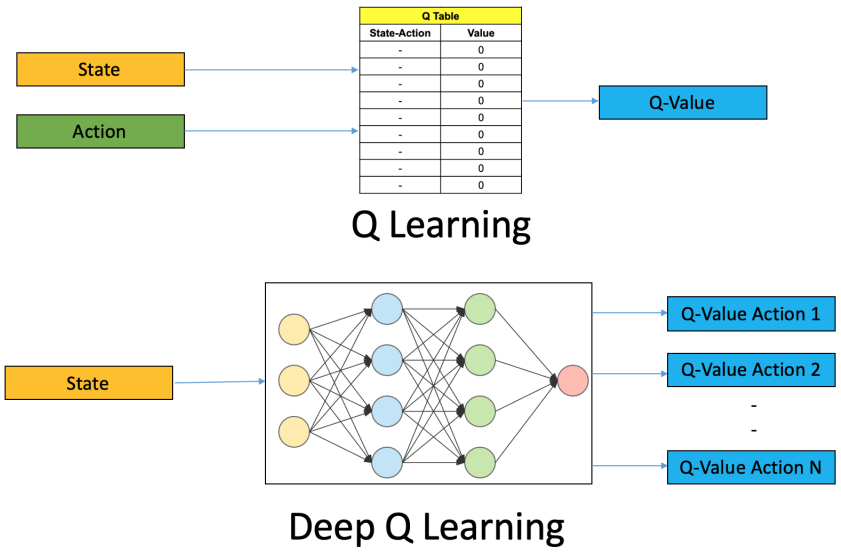

Figure 1: Illustration of the difference between Q-learning and Deep Q-learning

Training of the model is accomplished by minimising the following loss function:

$$L_i(\theta_i) = \mathbb{E}_{s,a,r,s' \sim \rho(.)} \left[(y_i - Q(s,a;\theta_i))^2 \right] \text{ where } y_i = r + \gamma \max_{a'} Q(s',a';\theta_{i-1})$$

here, $Q(s,a,\theta_i)$ represents the q value of action a taken at state s estimated by a neural network with parameters θ and $Q(s',a',\theta_{i-1})$ represents the generated Q value for next state and action combination.

Training of the model is performed using a technique called experience replay. Here, all of the transitions experienced by the network are stored in a buffer. When the weights of the model are to be updated, instead of using the latest transition, a mini-batch of transitions are sampled from the buffer to calculate the loss and gradient. This approach is beneficial in terms of data efficiency and makes the network updates more stable.

1.4 Policy Gradient

Policy gradient is a type of reinforcement learning algorithm that optimizes policies with respect to the expected return. The concept was solidified by Richard S. Sutton et. al. in their paper "Policy Gradient Methods for Reinforcement Learning with Function Approximation".

As mentioned, the goal of the policy gradient method is to determine the policy that maximises the expected reward in an environment. Compared to Q-learning, the learned policy is stochastic since it gives a probability distribution over all the actions and using the policy gradient method allows learning with continuous action spaces. However, a major disadvantage of this method is that they give high variance estimates of the gradient updates, which could destabilize the learning process.

Let us move on to the mathematics behind this method. First, let $\tau = (s_0, a_0, s_1, a_1, ...a_{T-1}, s_T)$ be the sequence of states and actions and agent takes to reach the terminal state with T steps. Also, let $R(s_t, a_t)$ be the reward obtained after performing action a_t at state s_t. We can now define the discounted sum of rewards as: $R(\tau) := \sum_{t=0}^{T-1} \gamma^t R(s_t, a_t)$. The goal of reinforcement learning then is to maximise the expected sum of rewards:

$$\max_{\theta} E_{\pi_\theta} R(\tau)$$

Here, π_θ is the policy dependent on parameters θ. This policy gives us a probability over all the actions at a state s. To update the parameters of the policy, we can use gradient ascent.

$$\theta \leftarrow \theta + \alpha \nabla_\theta E_{\pi_\theta} R(\tau)$$

The detailed calculation of $\nabla_\theta E_{\pi_\theta} R(\tau)$ can be found on the references section of this report. Based on this information, we can see that after some calculation, the gradient can be expressed as follows:

$$\nabla_\theta E_{\pi_\theta} R(\tau) = E_{\pi_\theta}(\sum_{t=0}^{T-1} \nabla_\theta log \pi_\theta(a_t|s_t) \sum_{t'=1}^{T-1} \gamma^{t'-t} R(s_{t'}, a_{t'}))$$

Now that we know how to update the policy parameters, we can define the vanilla policy gradient method algorithm.

Algorithm 1: The Vanilla Policy Gradient Algorithm

Result: A trained policy π_θ
Initialize policy parameters θ;
for *number of episodes* **do**
  Run through the environment using the current policy beginning from the start state to the terminal state;
  for *amount of steps* **do**
    Calculate discounted rewards;
    Calculate gradient using state and actions;
  end
  Dot product the gradient vector and the discounted rewards vector;
  Update policy parameters using gradient ascent;
end

Now that we have defined both algorithms, we can move on to their implementation in the homework.

1.5 Experiments

1.5.1 Q-learning on Cartpole environment

How to run To run the code for this section, use the cartpole.py python file located in the cartpole-master file in the QLearn file. Use section 1 to load the necessary libraries, set the training parameters, define the neural network to generate the Q values, define training function and run the training algorithm. Use section 2 and 3 to plot the loss curve. Note that this part of the code was written using the Spyder IDE, so to run the sections separately, code must be opened with this IDE.

In this part of the homework, we are asked to perform deep Q learning to get an agent that can navigate the Cartpole-v1 environment of the OpenAI gym. In this environment, there is a pole attached by an un-actuated joint to a cart, which moves along a frictionless track. The system is controlled by applying a force of +1 or -1 to the cart. The pendulum starts upright, and the goal is to prevent it from falling over. A reward of 1 is given as feedback for every step where the pole is upright.

To generate the Q values, we utilise a neural network with the following architecture:

```
Linear Layer (input dim = state vector size, output dim = 24)-->
Relu () -->
Linear Layer(input dim = 24, output dim = 24) -->
Relu () -->
Linear Layer(input dim = 24, output dim = action amount) -->
```

For Cartpole-v1, each state is represented by a vector of dimension 4, so the state vector size is 4. The agent can perform two actions depending on the direction of the force applied, so the action amount is 2.

Let us allow our deep Q learning setup to run enough times so that it can get an average reward higher than 195 for 100 consecutive runs. After letting the agent learn for 176 iterations, the winning condition is met and training stops. The visualisation of the training progress can be given as below:

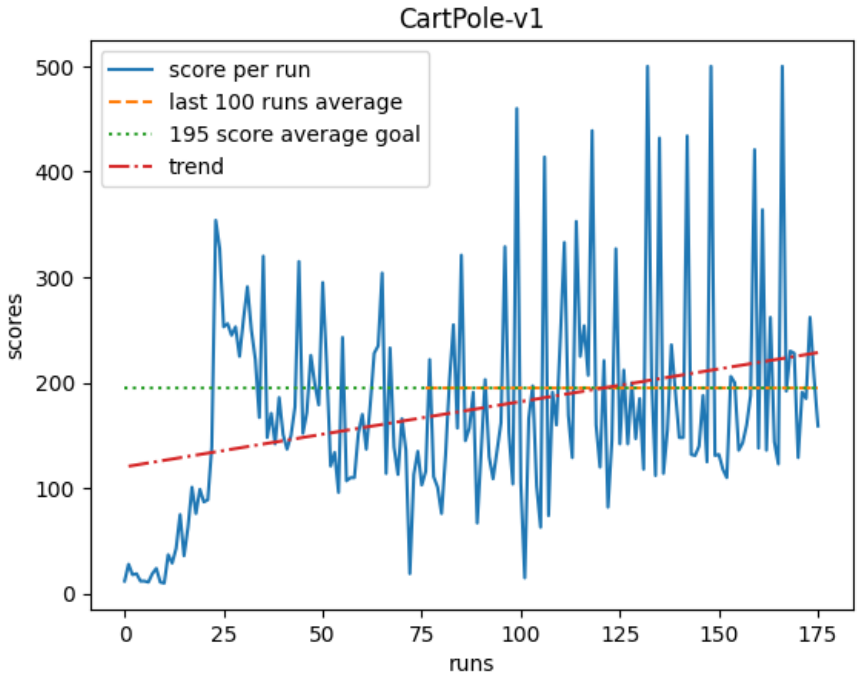

Figure 2: Results of the Q-learning algoritm on the Cartpole environment

As can be seen from the figure, during the initial runs the model is prone to performing poorly. After about 25 iterations, the model discovers a set of actions that get a large reward from the environment. Based on this information, it keeps updating the neural network to get better results. This process stops when the average of the last 100 runs passes the 195 threshold.

In fact, if we examine the reward values, we can see that the average of the last 100 rewards is 195.02. As this number is greater than 195, training was stopped.

Now, let us plot the change in average loss. Here, the loss is the average mean square error between the discounted reward and the Q value generated by the network for the current state and action combination.

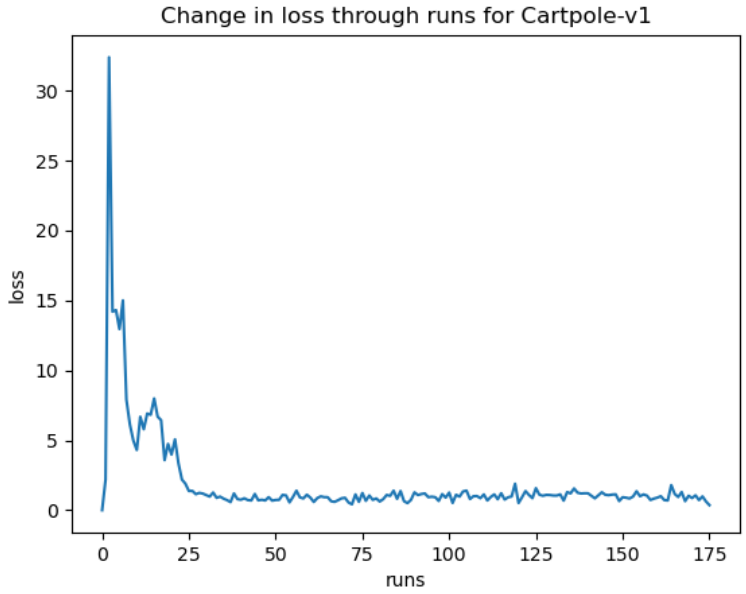

Figure 3: Change in loss

As can be seen from the loss graph, the neural network used to generate the Q values progressively gets more accurate and the loss stabilizes around 0.

1.5.2 Q-learning on LunarLander environment

How to run To run this code, use the LunarNotebook.ipynb notebook in the newEnv-master file inside the QLearn file. Use section 1 to import the libraries, section 2 to define the parameters and the environment, section 3 to define the agent, section 4 to define the training progress and section 5 to run the algorithm.

In this part of the homework, we are asked to to perform deep Q learning to get an agent that can navigate the LunarLander-v2 environment. Here, the goal is to land a spaceship to a landing area without it crashing. If the lander moves away from the landing area, it receives negative rewards. Episode finishes if the lander crashes or comes to rest, receiving additional -100 or +100 points. Each leg ground contact is +10. Firing main engine is -0.3 points each frame. Solved is 200 points. The agent can perform four actions: do nothing, fire left orientation engine, fire main engine, fire right orientation engine. The environment can be claimed to be learned when the average score for the last 100 runs is greater than 200.

Since the amount of actions and the state vector of the environment is different, we need to modify the used neural network structure. The new structure utilised in this part of the homework is as follows:

```
Linear Layer (input dim = state vector size, output dim = 150)-->
Relu () -->
Linear Layer(input dim = 150, output dim = 120) -->
Relu () -->
Linear Layer(input dim = 120, output dim = action amount) -->
```

For LunarLander-v2, each state is represented by a vector of dimension 8, so the state vector size is 8. The agent can perform four different actions, so the action amount is 4.

To run the deep Q learning algorithm in this environment, we needed to make some modifications. We changed how the reward was accumulated. Furthermore, we modified the code so that there was a limit on the total number of steps an agent can perform for each run. Finally, we changed the memory size to 1000000, batch size to 64, gamma to 0.99 and the decay rate to 0.996. The changes here were based on reference 7.

Let us now let our deep Q-learning setup to solve the LunarLander-v2 environment. After letting the agent play for 713 iterations, learning was stabilised and the agent was able to land the spaceship on the landing zone without crashing. The visualisation of the training progress can be given as below:

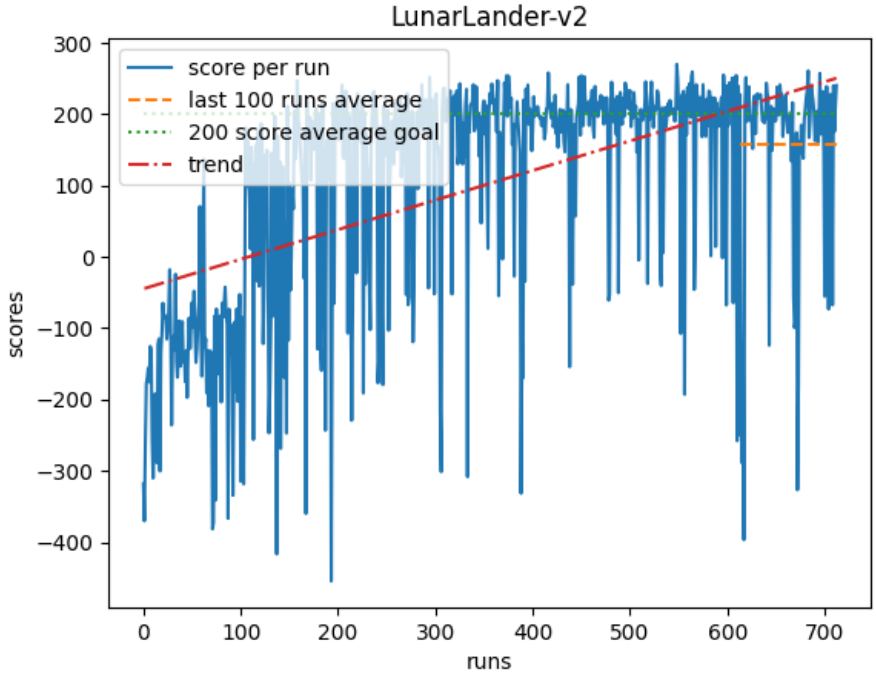

Figure 4: Training progress of the Q-learning algorithm on the LunarLander environment

As can be seen, at the first iterations the agent struggles with landing the ship correctly, which makes the agent get negative rewards. After about 100 iterations, the agent discovers how to land and slowly perfects the process. In the end, it is able to land the ship accurately for almost all of the runs, with an average reward of 156.87 for the last 100 iterations. Although this average score is less than the desired amount of 200, it is still enough to see that the agent learns and is able to consistently get high rewards. If the training was allowed to continue for longer, the average would have been higher. However due to time constraints, training was stopped at average 156.87.

As before, we can also plot the change in loss through the training process as follows using the difference between the discounted reward and generated Q values:

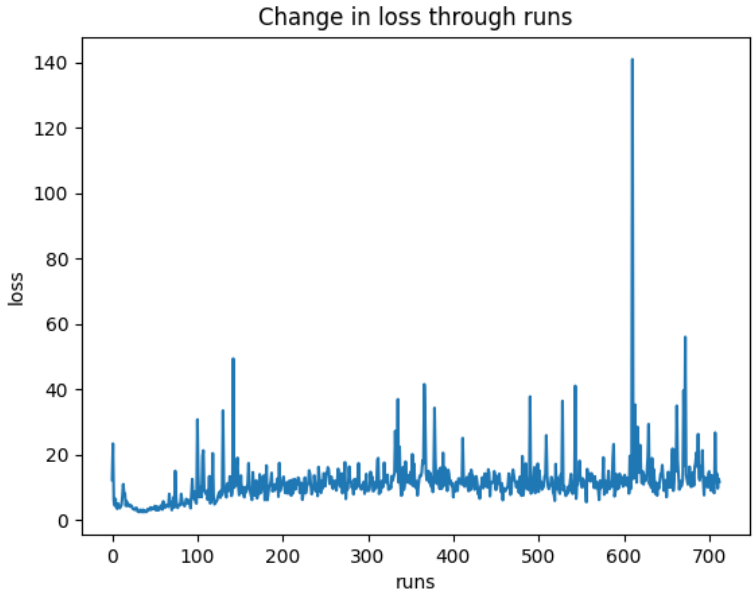

Figure 5: Change in loss of the Q-Learning algorithm for the LunarLander-v2 environment

As an be seen, the loss progressively decreases for the first 70 iterations. Then, it reaches a stable value where it stays during the rest of training. Although there are occasional jumps that increase the loss, indicating that on certain occasions the Q values generated by the network are not accurate, when looking at the entire graph it can be seen that the average loss pretty much stays the same.

1.5.3 Policy Gradient on Cartpole environment

How to run To run this part of the homework, run policyGradientCartpole.ipynb notebook in the Cartpole file of the Policy file. Here, run section 3 to import the necessary libraries, section 4 to create the environment, section 6 to define the logistic policy, section 7 to define how to run through the environment from start to finish, section 8 to define the training algorithm and section 9 to start training.

In this part of the homework, we are asked to solve the CartPole-v0 environment using the policy gradient method. There is no need to describe the environment again as it has the same rules as the Cartpole environment solved by utilising Q-learning.

In the policy gradient method, we need a way of generating the policy, which will give us the probability of all the possible actions from the current state. In our implementation, we use the logistic function as our policy. This function gives us the probability for one action. Since there are only two actions in this environment, the probability of the second action is just 1 minus the probability of the first action. In summation, the probability of actions in the environment is given by:

$$P(a_1) = \frac{1}{1 + e^{-x\theta}}$$

$$P(a_2) = 1 - \frac{1}{1 + e^{-x\theta}} = \frac{e^{-x\theta}}{1 + e^{-x\theta}}$$

where a_1 is the first action and a_2 is the second action. Let us now run the policy gradient method on the Cartpole-v0 environment. After running for 524 number of iterations, the agent manages to solve the environment and we get an average reward greater than 195 for the last 100 runs. The training progress can be visualised as below:

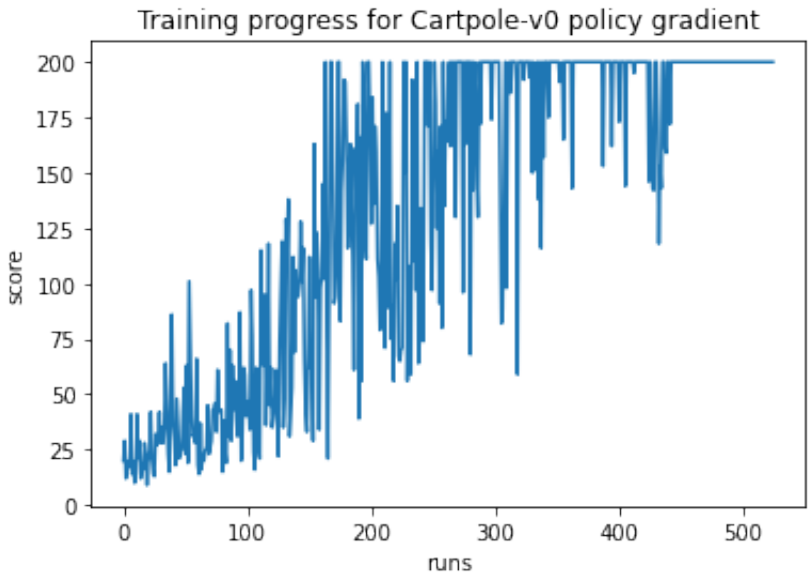

Figure 6: Training progress for the policy gradient algorithm on Cartpole-v0

As can be seen , our logistic policy gets progressively better as the runs progress. There is a spike in score around run 200. After run 430, the policy is able to consistently get the highest reward of 200.

The average reward for the last 100 runs for this method was found to be 195.29. As this value is higher than 195, training of the agent was automatically stopped.

Now, let us examine the progress of the loss function. For a policy π_θ in the policy gradient method, the loss can be defined as the following:

$$Loss = \frac{1}{T}\sum_{t=0}^{T-1} -log\pi_\theta(a_t|s_t)NormalizedDiscountedRewards_t$$

Here, $NormalizedDiscountedRewards$ is calculated by first finding the discounted sum of rewards: $R(\tau) := \sum_{t=0}^{T-1} \gamma^t R(s_t, a_t)$ for each time step. Then, these reward values are normalized by subtracting the mean of all the discounted values and dividing by the standard deviation of the discounted values. $\pi_\theta(a_t|s_t)$ in our case refers to the probabilities generated by the logistic policy at a state s_t.

The change of this loss through the runs is as follows:

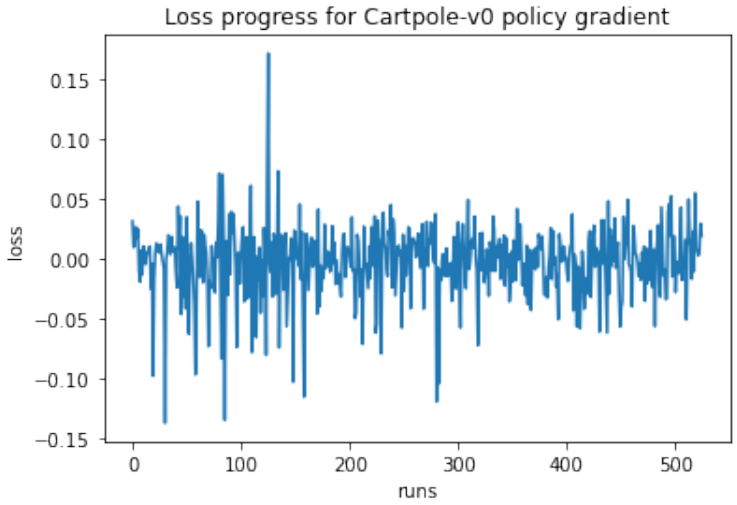

Figure 7: Training progress for the policy gradient algorithm on Cartpole-v0

As can be seen from the figure, there is a decrease in the loss value during the initial runs. After that, the loss stabilises around 0 and remains there during the rest of training. One thing to note here is that there is a lot of variance in the oscillation around 0. This is expected since as explained during the introduction to the policy gradient, the updates made to the agent using this method have a lot of variance.

1.5.4 Policy Gradient on LunarLander environment

How to run To run this part of the code, use the policyLunarLander.ipynb notebook in the LunarLander file of the Policy file. Use section 1 to load the libraries, section 2 to create the environment, section 4 to define the policy, section 5 to initialise the policy, section 6 to set the parameters and section 7 to begin training.

For the final part of the homework, we are asked to implement the policy gradient algorithm on the LunarLander-v2 environment. As this environment was explained on section 1.5.2, there is no need to go through it again.

For Cartpole-v0, we used a logistic policy. However, we can not use this policy in this environment because there are now 4 actions instead of 2. To overcome this issue, we changed the policy to a neural network with the following architecture:

```
Linear Layer (input dim = state vector size, output dim = 10)-->
Relu () -->
Linear Layer(input dim = 10, output dim = 10) -->
Relu () -->
Linear Layer(input dim = 10, output dim = action amount) -->
```

As explained before, the state vector is of size 8 and the action amount is 4. From this network, the action probabilities are generated by passing the output of the network through the softmax function.

To make the existing code suitable for this new environment, we changed some parts of the code. First of all, we started using Tensorflow to calculate the gradients and update the policy parameters instead of doing the operation manually. Furthermore, we moved the runEpisode function inside of the training loop. We also added a time limit of 120 seconds to each run, so that the agent does not take too long to move on to the next run if it has not reached a terminal state for a long time. Moreover, we changed the learning rate to 0.02 and the gamma variable to 0.99. The changes here were based on reference 8.

After letting the agent play for a total of 1000 iterations, the rewards acquired became stable around 200, indicating that the agent was capable of landing the spaceship successfully in the landing area. We can visualise the training progress as follows:

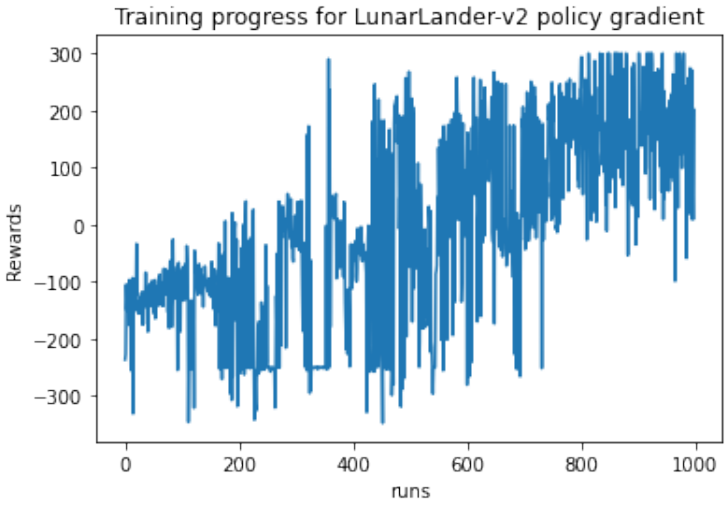

Figure 8: Training progress for the policy gradient algorithm on LunarLander-v2

As can be seen from the graph, in the initial runs the model receives a negative reward due to either crashing the ship or using too many actions to land. At run 357, the agent manages to land the ship and receives a high reward, which prompts it to change its course of action. After run 500, the agent starts getting positive rewards and after run 780, the agent is capable of consistently getting rewards around 200. One thing to notice in this graph is that the rewards have high variance. This is expected since the policy gradient method suffers from calculated gradients with high variance, which makes the learning process unstable, leading to highly varying rewards.

The average reward obtained during the last 100 runs of this method was found to be 174.1. Since this value is close to 200, we can say that the model has successfully learned its environment.

We can also plot the change in loss through training as we did for the Cartpole environment. Here, our loss function remains the same as the one detailed in section 1.5.3.

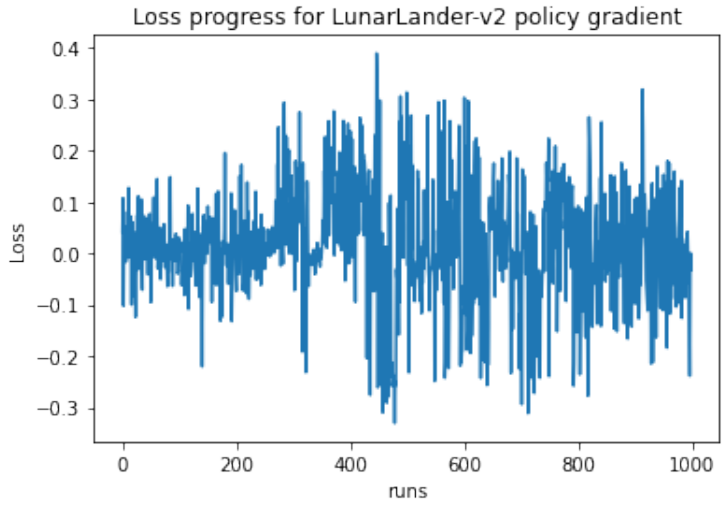

Figure 9: Loss progress for the policy gradient algorithm on LunarLander-v2

As can be seen, the loss calculated by using the normalized discounted rewards oscillates around 0 during training. As with the rewards graph, the loss also suffers from high variance, which is expected from the policy gradient method.

1.6 Conclusion

In conclusion, we learned about deep Q-learning and policy gradient methods and how to use them to teach agents how to navigate various environments.

Running the tutorials involved installing several libraries that clashed with previously downloaded packages. Managing them proved to be challenging. Furthermore, the most difficult part of the homework was changing the code to accommodate the LunarLander-v2 environment for both the policy gradient method and the Q-learning method.

Reference

Watkins, Christopher John Cornish Hellaby. "Learning from delayed rewards." (1989).

Sutton, Richard S., et al. "Policy gradient methods for reinforcement learning with function approximation." Advances in neural information processing systems. 2000.

"Introduction to RL and Deep Q Networks: Tensorflow Agents." TensorFlow, https://www.tensorflow.org/agents/tutorials/0_intro_rl.

"Vanilla Policy Gradient." Vanilla Policy Gradient - Spinning Up Documentation, https://spinningup.openai.com/en/latest/algorithms/vpg.html.

Klaise, Janis. "Reinforcement Learning with Policy Gradients in Pure Python." Wishful Tinkering, https://www.janisklaise.com/post/rl-policy-gradients/.

Surma, Greg. "Cartpole - Introduction to Reinforcement Learning (DQN - Deep Q-Learning)." Medium, Medium, 10 Nov. 2019, https://gsurma.medium.com/cartpole-introduction-to-reinforcement-learning-ed0eb5b58288.

Verma, Shiva. "Train Your Lunar-Lander: Reinforcement Learning." Medium, Medium, 5 Oct. 2021, https://shiva-verma.medium.com/solving-lunar-lander-openaigym-reinforcement-learning-785675066197.

sudharsan13296. "Hands-on-Reinforcement-Learning-with-Python/11.2 Lunar Lander Using Policy Gradients.ipynb at Master · Sudharsan13296/Hands-on-Reinforcement-Learning-with-Python." GitHub