STATS M231A Homework 5

Laxman Dahal

Abstract

This homework consists of two major parts: 1) Install required packages and train an agent for openAI gym environment, specifically CartPole-v0. For the first part, the reference code given were implemented for most part. Modifications were made to the baseline code to plot loss curve and reward curves as required by the problem statement. 2) For the second part, mountainCar-v0 environment from openAI was used to train an agent. A car is on a one-dimensional track, positioned between two "mountains". The goal is to drive up the mountain on the right; however, the car's engine is not strong enough to scale the mountain in a single pass. Therefore, the only way to succeed is to drive back and forth to build up momentum. To train the agents, Deep Q-Learning Network and Policy Gradient within reinforcement learning framework was implemented. It was found that, agent for both the algorithms were able to solve the problem after about 1000 episodes on average.

1 Reinforcement Learning

Reinforcement learning (RL) is a part on unsupervised machine learning method with the idea of reinforcing desired actions by rewarding and/or punishing undesired ones. In general, a RL agent is able to perceive and interpret its environment, take actions and learn through trial and error. In general, RL is designed such that a positive value is assigned if the agent takes desired actions and a negative value is assigned otherwise. The idea it to program the agent to seek long-term and maximum overall reward to achieve an optimal solution. These long-term goals help prevent the agent from achieving local optimal value instead of global goal. With time, the agent learns to avoid the negative/bad actions and seek the positive/good actions.

2 Deep Q-Network (DQN)

Q-learning is a model-free reinforcement learning algorithm that utilizes bootstrapping technique and is designed to learn the value of an action in a particular state. In Markov Decision Process (MDP), Q-learning finds an optimal policy in the sense of maximizing the expected value of the total reward over any and all successive steps, starting from the current state. Let $Q_\theta(s,a)$ be the value network. Let s' be the state observed after playing a at state s. Let R be the return if we play out as best as we can. Then we can update θ by

$$\Delta\theta \propto \frac{\partial}{\partial\theta}[R - Q_\theta(s,a)]^2.$$

We use the bootstrap method to get

$$R = r + \gamma \max_{a'} Q_\theta(s',a').$$

At state s, a is chosen by taking $a = \arg\max_a Q_\theta(s,a)$, where ε represent the probability of taking random action. It is important to note that, in learning phase, the greedy policy is not chosen because the idea is to have Q-learning explore other actions. Here $Q(s,a)$ is defined by: $Q(s,a) = r(s,a) + \gamma max_a Q(s',a)$.

Figure 1 illustrates network structure of deep Q-learning. It is called deep Q-learning because it utilizes deep neural networks to approximate the Q-value function. The state is given as the input and the Q-value of all possible actions is generates as the output.

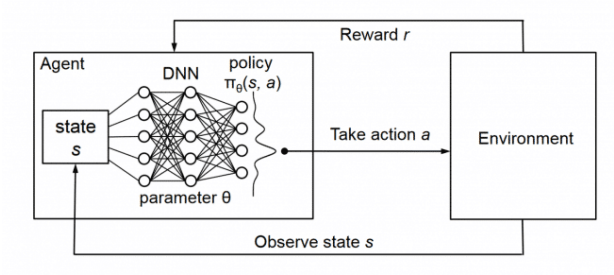

Figure 1: Network structure of deep Q-learning.

2.1 Algorithm

Figure 2 highlights the algorithm of the DQN with experience replay. To perform experience replay agent's experiences are stored as a data. To implement the algorithm outlined above, the following

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Figure 2: Algorithm of the q learning.

steps are taken:

1. Preprocess and feed state s to the network (DQN) which will return the Q-values of all possible actions in the state
2. Select an action using the $\varepsilon-$greedy policy. WIth the probability ε, select a random action a, and with 1 - ε probability, select an action that has a maximum Q-value such that $a = \arg\max_a Q_\theta(s, a)$
3. Perform this action in the state s and move to a new state s' to receive a reward. This state s' is the preprocessed image of the next game screen. We store this transition in our replay buffer as $\langle s, a, r, s' \rangle$.
4. Sample some random batches of transitions from the replay buffer and calculate the loss

$$Loss = (r + \gamma max_{a'} Q(s', a'; \theta') - Q(s, a; \theta')$$

   which is just the squared difference between the target Q and predicted Q
5. Perform gradient descent with respect to actual network parameters to minimize the loss
6. After every C iterations, copy actual network weights to the target network weights
7. repeat these process for M number of episodes

2.2 Implementation

To implement the algorithms, MountainCar-v0 environment is used from gym package. In the MountainCar environment, A car is on a one-dimensional track, positioned between two "mountains" as shown in Figure 3. The goal is to drive up the mountain on the right; however, the car's engine is not strong enough to scale the mountain in a single pass. Therefore, the only way to succeed is to drive back and forth to build up momentum

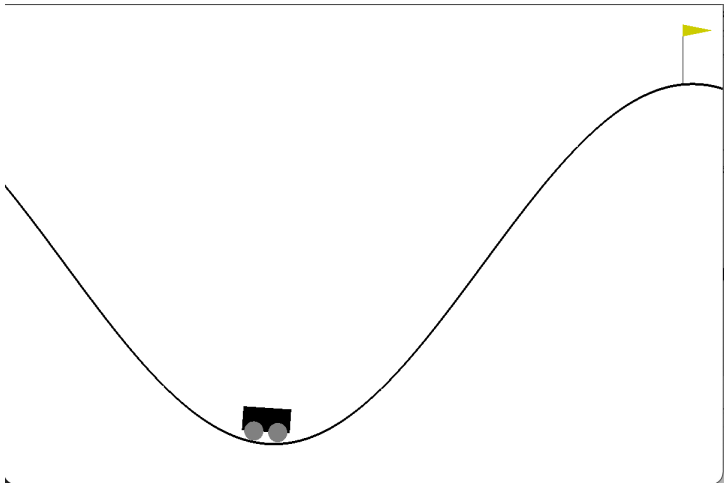

Figure 3: Graphical representation of the MountainCar-v0 environment from the gym. The goal is to drive up the mountain.

2.2.1 Loss Curve

Figure 4 shows the loss curve of the the algorithm of the DQN with experience replay. To perform experience replay agent's experiences are stored as a data.

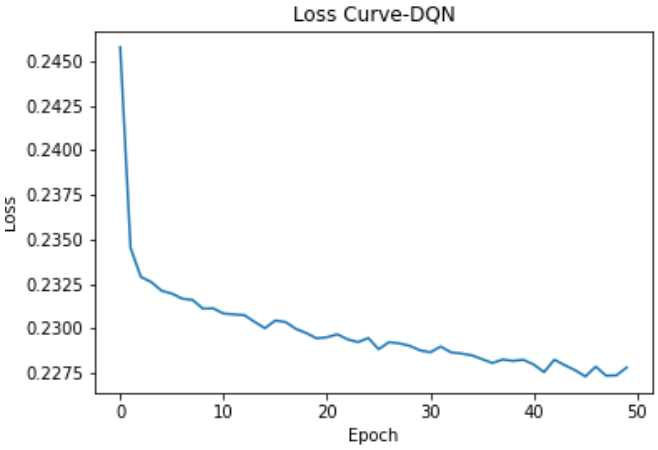

Figure 4: Loss curve of DQN.

2.2.2 Reward Curve

Figure 5 shows the reward curve of the the algorithm of the DQN with experience replay. To perform experience replay agent's experiences are stored as a data.

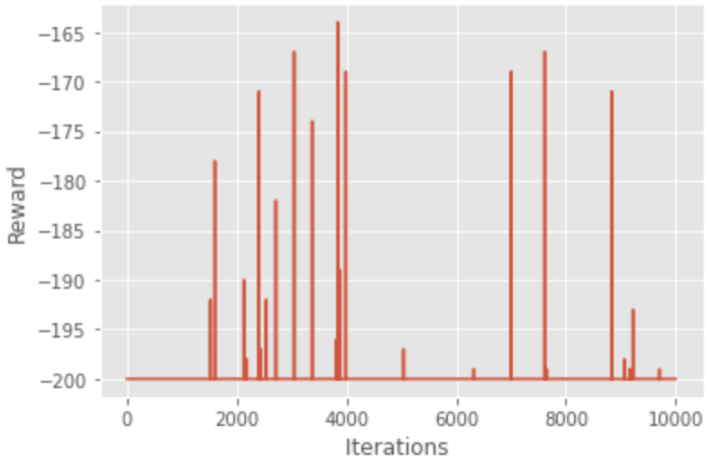

Figure 5: Reward curve of DQN.

2.2.3 Total Reward

The average reward was found to be -130.81. For calculation of the average reward, please refer to the attached notebook.

3 Policy Gradient

Let $\tau = (s_0, a_0, \ldots, s_{T-1}, a_{T-1}, s_T)$ be a state-action sequence in a complete trajectory consisting of T steps (note, the final state s_T is a terminal state which results from taking the final action a_{T-1}, after which the environment is reset). Define $R(s_t, a_t)$ to be the reward received after observing the state s_t and performing an action a_t. Also define the (discounted) sum of these rewards to be $R(\tau) := \sum_{t=0}^{T-1} \gamma^t R(s_t, a_t)$. Then our goal is to maximize the expected reward

$$\max_{\theta} \mathbb{E}_{\pi_\theta} R(\tau),$$

where π_θ is a parametrized policy (typically a neural network). The expected value is taken with respect to drawing trajectories τ under the policy π_θ and so solving this problem is equivalent to finding the "best" parameters θ that give the best policy for maximizing the expected reward.

We can do this via the usual gradient ascent, i.e. suppose we know how to calculate the gradient with respect to the parameters, $\nabla_\theta \mathbb{E}_{\pi_\theta} R(\tau)$, then we can update the parameters θ in the direction of the gradient:

$$\theta \leftarrow \theta + \alpha \nabla_\theta \mathbb{E}_{\pi_\theta} R(\tau),$$

where α is the usual learning rate hyperparameter.

Let $P(\tau|\theta)$ be the probability of a trajectory τ under the policy π_θ. Then we can write the gradient as follows:

$$
\begin{aligned}
\nabla_\theta \mathbb{E}_{\pi_\theta} R(\tau) = & \\
&= \nabla_\theta \sum_\tau P(\tau|\theta) R(\tau) && \text{definition of expectation} \\
&= \sum_\tau \nabla_\theta P(\tau|\theta) R(\tau) && \text{swap sum/integral and gradient} \\
&= \sum_\tau \frac{P(\tau|\theta)}{P(\tau|\theta)} \nabla_\theta P(\tau|\theta) R(\tau) && \text{multiply and divide by } P(\tau|\theta) \\
&= \sum_\tau P(\tau|\theta) \nabla_\theta \log P(\tau|\theta) R(\tau) && \text{recognize that } \nabla_x \log(f(x)) = \frac{\nabla_x f(x)}{f(x)} \\
&= \mathbb{E}_{\pi_\theta} \left(\nabla_\theta \log P(\tau|\theta) R(\tau) \right) && \text{definition of expectation}
\end{aligned}
$$

Now we can expand the probability of a trajectory τ as follows:

$$
P(\tau|\theta) = p(s_0) \prod_{t=0}^{T-1} p(s_{t+1}|s_t, a_t) \pi_\theta(a_t|s_t),
$$

where $p(s_{t+1}|s_t, a_t)$ is the probability of transitioning to state s_{t+1} by taking an action a_t in state s_t (as specified by the Markov Decision Process underlying the RL problem) and $p(s_0)$ is the starting state distribution. Taking the gradient of the log-probability (abbreviated as grad-log-prob from here on) of a trajectory thus gives

$$
\begin{aligned}
\nabla_\theta \log P(\tau|\theta) = & \\
&= \nabla_\theta \left(\log p(s_0) + \sum_{t=0}^{T-1} \left(\log p(s_{t+1}|s_t, a_t) + \log \pi_\theta(a_t|s_t) \right) \right) \\
&= \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t).
\end{aligned}
$$

Note that after taking the gradient, the dynamics model $p(s_{t+1}|s_t, a_t)$ disappears, i.e. policy gradients are a *model-free* method.

Putting this all together we arrive at the policy gradient expression:

$$
\nabla_\theta \mathbb{E}_{\pi_\theta} R(\tau) = \mathbb{E}_{\pi_\theta} \left(\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau) \right),
$$

and because it is an expectation it can be estimated by Monte Carlo sampling of trajectories:

$$
\nabla_\theta \mathbb{E}_{\pi_\theta} R(\tau) \approx \frac{1}{L} \sum_\tau \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau),
$$

where L is the number of trajectories used for one gradient update.

With some algebra and rearrangement

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

Now that we know that the dimensions of the observation and action space we can design a policy that takes in observations and produces probabilities of actions. Typically the parametrized policy π_θ is a neural network where θ represents the learnable weights of the network. In our case the problem is so simple that there is no need for a sophisticated neural network policy. We will instead use plain logistic regression to parametrize probabilities of moving left and right. Additionally, we will use this simple form of a policy to manually derive the gradients for the policy gradient update rule.

Let x denote the length 4 observation vector. Because the cartpole problem is fully observable, the observation and the state are interchangeable concepts. Let $\pi_\theta(0|x) = \frac{1}{1+e^{-\theta\cdot x}} = \frac{e^{\theta\cdot x}}{1+e^{\theta\cdot x}}$ be the probability of action 0 (move cart to the left), then $\pi_\theta(1|x) = 1-\pi_\theta(0|x) = \frac{1}{1+e^{\theta\cdot x}}$. Thus our policy is parametrized by a vector θ of length four.

To apply the policy gradient update we need to derive $\nabla_\theta \log \pi_\theta(a|x)$:

$$\begin{aligned}\nabla_\theta \log \pi_\theta(0|x) = \\ &= \nabla_\theta \left(\theta\cdot x - \log(1+e^{\theta\cdot x})\right) \\ &= x - \frac{xe^{\theta\cdot x}}{1+e^{\theta\cdot x}} \\ &= x - x\pi_\theta(0|x)\end{aligned}$$

and

$$\begin{aligned}\nabla_\theta \log \pi_\theta(1|x) = \\ &= \nabla_\theta \left(-\log(1+e^{\theta\cdot x})\right) \\ &= -\frac{xe^{\theta\cdot x}}{1+e^{\theta\cdot x}} \\ &= -x\pi_\theta(0|x).\end{aligned}$$

Of course when π_θ is a neural network you would rely on the auto-differentiation capabilities of your deep learning framework to do these calculations for you.

3.1 Algorithm

Figure 6 highlights the algorithm of the Policy Gradient with actor:critic setup.

Algorithm 1 Soft Actor-Critic

Input: θ_1, θ_2, ϕ ▷ Initial parameters
  $\bar{\theta}_1 \leftarrow \theta_1, \bar{\theta}_2 \leftarrow \theta_2$ ▷ Initialize target network weights
  $\mathcal{D} \leftarrow \emptyset$ ▷ Initialize an empty replay pool
  for each iteration **do**
    for each environment step **do**
      $\mathbf{a}_t \sim \pi_\phi(\mathbf{a}_t|\mathbf{s}_t)$ ▷ Sample action from the policy
      $\mathbf{s}_{t+1} \sim p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$ ▷ Sample transition from the environment
      $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\mathbf{s}_t, \mathbf{a}_t, r(\mathbf{s}_t, \mathbf{a}_t), \mathbf{s}_{t+1})\}$ ▷ Store the transition in the replay pool
    end for
    for each gradient step **do**
      $\theta_i \leftarrow \theta_i - \lambda_Q \hat{\nabla}_{\theta_i} J_Q(\theta_i)$ for $i \in \{1,2\}$ ▷ Update the Q-function parameters
      $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi)$ ▷ Update policy weights
      $\alpha \leftarrow \alpha - \lambda \hat{\nabla}_\alpha J(\alpha)$ ▷ Adjust temperature
      $\bar{\theta}_i \leftarrow \tau\theta_i + (1-\tau)\bar{\theta}_i$ for $i \in \{1,2\}$ ▷ Update target network weights
    end for
  end for
Output: θ_1, θ_2, ϕ ▷ Optimized parameters

Figure 6: Algorithm of the policy gradient.

3.2 Implementation

3.2.1 Position

Figure 7 shows the graph of position of the mountain car. It can be seen that around episode 330, the agent begins to successfully complete episodes and around 370 episodes, the agent completes almost every episode successfully. This indicates that the agent has learned to complete the environment.

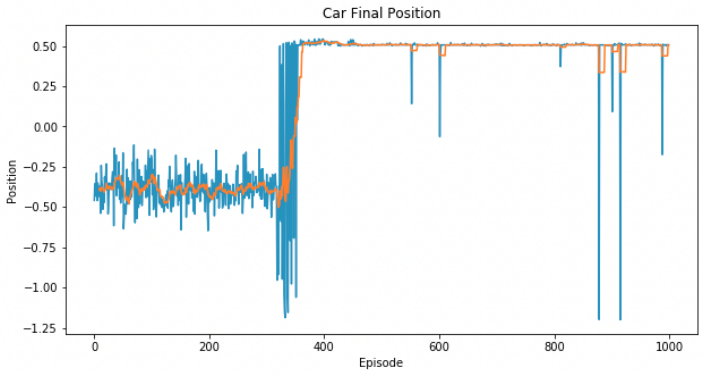

Figure 7: Graphical illustration of the position as a function of number of episodes. If the position is 0.5, it indicates that the agent is able to complete the episode successfully.

3.2.2 Loss Curve

Figure 8 shows the loss curve of the the algorithm of the Policy Gradient.

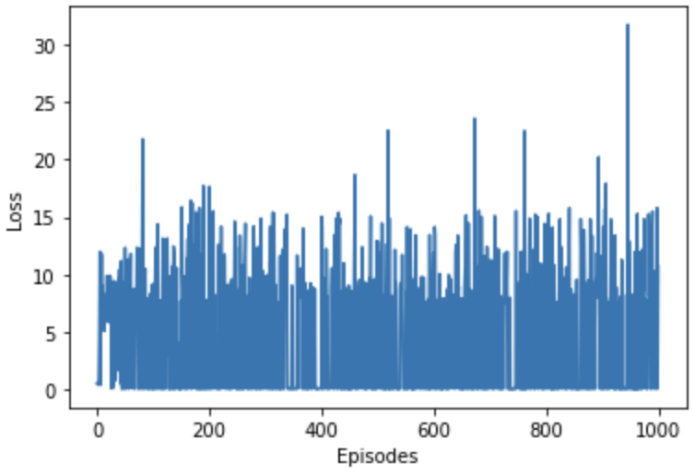

Figure 8: Loss curve of RL Policy Gradient.

3.2.3 Reward Curve

Figure 9 shows the reward curve of when Policy Gradient is implemented.

3.2.4 Total Reward and Policy

Figure 10 shows the visualization of the learned policy. The agent learns to move left when the car's velocity is negative and then switch directions when the car's velocity becomes positive with a few position and velocity combinations on the far left of the environment where the agent will do nothing. In this implementation, the agent was found to solve the episodes 63%. The problem is considered solved, if it is able to solve more than 30% of the time.

How to run There are three Jupyter Notebook attached along with this document. All three notebooks were executed in Google Colab utilizing their GPU capability. Please make sure to install all the dependencies of "gym" package to be able to successfully run the code. The details about the notebooks are listed below:

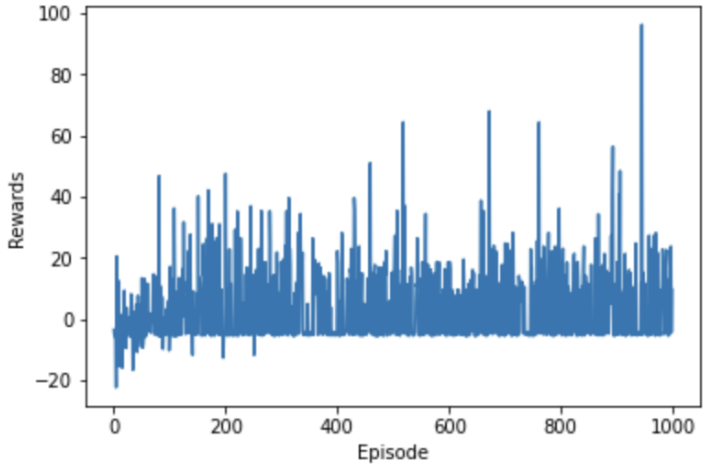

Figure 9: Reward curve of RL Policy Gradient.

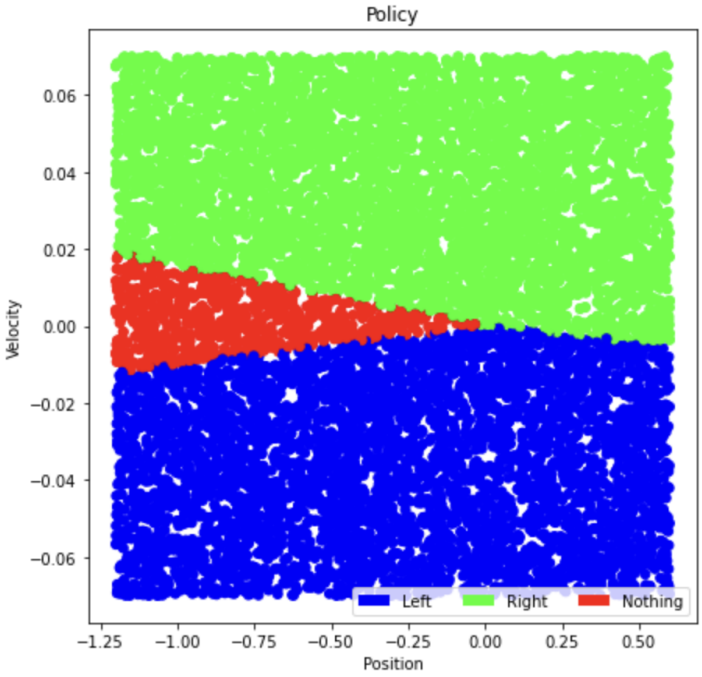

Figure 10: Graphical representation of the policy implemented.

1. “hw5_CartPole_DQN.ipynb" implements CartPole with DQN algorithm
2. “hw5_CartPole_PolicyGradient.ipynb" implements CartPole with Policy Gradient algorithm
3. “hw5_MountainCar_DQN.ipynb" implements MountainCar-v0 with DQN algorithm
4. “hw5_MountainCar_PolicyGradient.ipynb" implements Mountaincar-v0 with Policy Gradient algorithm

3.3 Conclusion

In total, two environments (CartPole-v0 and Mountaincar-v0) from the gym package was trained using two algorithms (Deep Q Network and Policy Gradient) within reinforcement learning framework. It was found that for all the cases, the agent was able to learn how to solve the problem.

The following figures show the results obtained with the code given by TA was used. This demonstrates that I was able to successfully run the codes.

Appendix A CartPole-v0 DQN

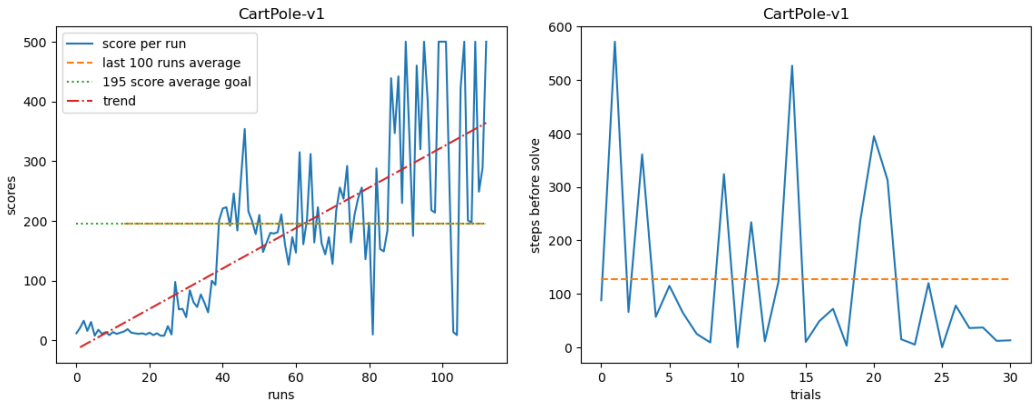

Figure 11: The figure on the left shows the score and the figure on the right shows the solved case for the Cartpole-v0 environment with default code with DQN.

Appendix B CartPole-v0 Policy Gradient

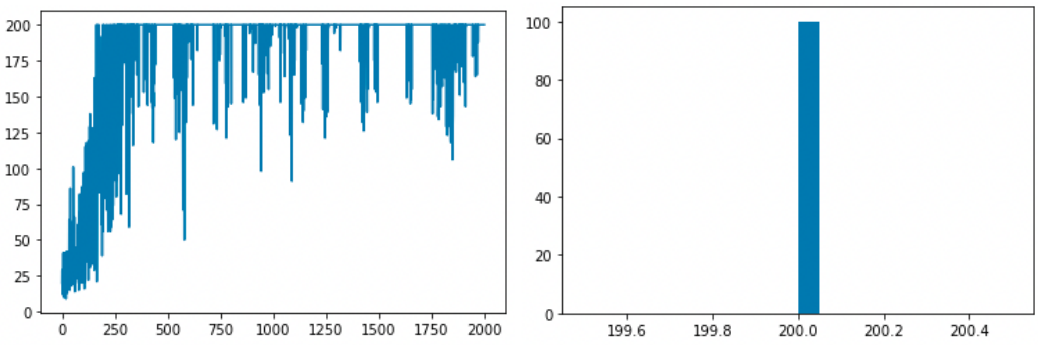

Figure 12: Figure on the left shows the reward curve and the figure on the right shows the solved cases when Policy Gradient was used.