Deep Reinforcement Learning Practical Session

Large Scale Machine Learning - Mines ParisTech

March 2022

1 Introduction

All of the instructions are on the notebook. We recommend staying in Colab. You should follow the notebook instructions for the deliverables. Here you will find some theoretical background for each question.

Some definitions:

- A *trajectory* τ is a sequence $(s_1, a_1, s_2, a_2 ... s_T, a_T)$ of the visited states and actions. The probability of a trajectory under the policy π is $p_\pi(\tau) = \pi(\tau) = p(s_1) \prod_{t=1}^{T} \pi(a_t|s_t) p(s_{t+1}|s_t, a_t)$.
- The discounted cumulative reward is $R_t = \sum_{k=0}^{T} \gamma^k r(s_{t+k}, a_{t+k})$. Our goal is to maximize $J(\pi) = \mathbb{E}_{\tau \sim p_\pi(\tau)} [R_1]$.
- The *Q-function* $Q^\pi(s_t, a_t) = \mathbb{E}_{p_\pi} [R_t | s_t, a_t] = \sum_{t'=t}^{T} \mathbb{E}_{p_\pi} \left[\gamma^{t'-t} r(s_{t'}, a_{t'}) | s_t, a_t \right]$ is the reward-to-go from state s_t if we pick the action a_t and then follow π.
- The *value function* $V^\pi(s_t) = \mathbb{E}_{p_\pi} [R_t | s_t] = \sum_{t'=t}^{T} \mathbb{E}_{p_\pi} \left[\gamma^{t'-t} r(s_{t'}, a_{t'}) | s_t \right] = \mathbb{E}_{a_t \sim \pi(a_t|s_t)} [Q^\pi(s_t, a_t)]$ is the reward-to-go from state s_t if we follow the policy π.
- The *advantage function* $A^\pi(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t)$ represents the value of a particular action a_t with respect to the average action taken by π at state s_t.

2 Policy gradient

The idea is to do stochastic gradient ascent on the objective $J(\pi) = \mathbb{E}_{\tau \sim p_\pi(\tau)} [R_1]$ to find the best parameterized stochastic policy π_θ. A simple example is the Reinforce algorithm [4].

1. sample $\{\tau^i\}$ from $\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)$ (run the policy)
2. $\nabla_\theta J(\theta) \approx \sum_i \left(\sum_t \nabla_\theta \log \pi_\theta(\mathbf{a}_t^i|\mathbf{s}_t^i) \right) \left(\sum_t r(\mathbf{s}_t^i, \mathbf{a}_t^i) \right)$
3. $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

Figure 1: Reinforce algorithm (source: [1])

Reinforce uses the *likelihood ratio trick* to get rid of the terms depending on the dynamics:

$$\nabla_\theta J(\theta) = \nabla_\theta \int_\tau p_\theta(\tau) r(\tau) d\tau = \int_\tau p_\theta(\tau) \nabla_\theta \log p_\theta(\tau) r(\tau) d\tau = \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\nabla_\theta \sum_t \log \pi_\theta(a_t|s_t) r(\tau) \right] \quad (1)$$

Step 2 of the algorithm uses samples to approximate the expectation. Step 3 is gradient ascent, making good sampled trajectories more likely, and bad trajectories less likely.

2.1 CartPole - Compare different hyper-parameters

Run some experiments in the discrete cartpole environment. Two interesting parameters to tune are:

reward_to_go. We usually replace $\sum_{t=1}^{T} \gamma^{t-1} r(s_t^i, a_t^i)$ with $\sum_{t'=t}^{T} \gamma^{t'-t} r(s_{t'}^i, a_{t'}^i)$, i.e. we hold policies accountable for future rewards only, which reduces the variance but might bring a bias when $\gamma < 1$.

batch_size. Number of state-action pairs sampled while acting according to the current policy at each iteration. A bigger batch size should also reduce the variance.

2.2 LunarLander - Use a neural network baseline

Try using a neural network baseline in the more demanding continuous lunar lander environment.

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t^i | s_t^i) (\sum_{t'=t}^{T} \gamma^{t'-t} r(s_{t'}^i, a_{t'}^i) - b) \tag{2}$$

Reinforce uses the MC (Monte Carlo) estimator of the Q-function with a single sample $\hat{Q}^\pi(s_t, a_t) = \sum_{t'=t}^{T} \gamma^{t'-t} r(s_{t'}^i, a_{t'}^i)$, hence the high variance. To reduce the variance, we can add a state-dependant baseline (fancy term meaning some quantity to compare against) that approximates the value function $b = \hat{V}^\pi(s_t)$, so that we compare the obtained returns with the expected returns. This choice of baseline results in an approximation of the advantage function $\hat{A}^\pi(s_t, a_t)$.

The goal is to approximate the value function with a neural network V_ϕ. To do so, we use the return-to-go MC targets $y_i = \sum_{t'} \gamma^{t'-t} r(s_{t'}^i, a_{t'}^i)$ and minimize the supervised regression loss $\frac{1}{2} \sum_i ||V_\phi^\pi(s_t^i) - y_i||^2$.

- Remark: In practice, we compute normalized targets because it is easier for the network to predict returns with zero mean and standard deviation of one.
- Remark: The PyTorch function detach() stops the gradients from propagating back to the value function network when updating the policy.

3 Q-learning

The goal is to recover the optimal Q-function Q^*, which we know verifies the optimal Bellman equation (derived from dynamic programming). A similar equation holds for the value function, but the *max* operator lies outside of the expectation, which prevents us from using samples to approximate the expectation. One advantage of approximating Q^* rather than Q^π is that the algorithms become *off-policy*: theoretically, the training data can come from any policy, as opposed to *on-policy* algorithms such as policy gradient which require data coming from π to approximate Q^π.

$$Q^*(s_t, a_t) = \mathbb{E}_{s_{t+1} \sim p(\cdot | s_t, a_t)} \left[r(s_t, a_t) + \gamma \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1}) \right] \tag{3}$$

To do so, we minimize the TD (temporal difference) error $\sum_i ||Q_\phi(s_i, a_i) - y_i||^2$ where $y_i = r(s_i, a_i) + \gamma \max_{a_i'} Q_\phi(s_i', a_i')$. A very popular Q-learning algorithm is DQN [3]. Let's cover the two main ideas:

- *Replay buffer*. As we interact with the environment sequentially, the samples are strongly correlated which can lead to local over-fitting. The idea is to have a buffer $\mathcal{B}$ where we add the collected data, and then update the Q-function from data sampled uniformly from $\mathcal{B}$.
- *Target network*. We do not regress the Q-function to the TD targets y until convergence because the optimal estimate from the available data will be very poor in early stages. This means that the targets

y are going to change very frequently, which doesn't help with stability. The idea is to do regular backups of the parameters ϕ into a new target network $Q_{\phi'}$ used to compute these targets.

Q-learning with replay buffer and target network: DQN: $M = 1$, $K = 1$

1. save target network parameters: $\phi' \leftarrow \phi$
2. collect M datapoints $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$ using some policy, add them to $\mathcal{B}$

$N\times$ $K\times$

3. sample a batch $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$ from $\mathcal{B}$
4. $\phi \leftarrow \phi - \alpha \sum_i \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i)(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - [r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_{\phi'}(\mathbf{s}'_i, \mathbf{a}'_i)])$

Figure 2: DQN algorithm (source: [1])

3.1 MountainCar - Compare different rewards

Solving the mountain car task is very difficult with the original reward, which gives -1 at every timestep until the car reaches the flag. In order to make any progress, the car needs to reach the flag first, and relies only on exploration to do so. A better strategy would be to reward an increase in mechanical energy (potential plus kinetic) at every time step. Try out this alternative reward in the notebook.

3.2 LunarLander - Implement double DQN

One problem with DQN is overestimation when computing the targets. Since $Q_{\phi'}$ is not perfect, $\max_{a'} Q_{\phi'}(s', a')$ will be an overestimation if there is any upwards noise. If we could select the action according to a different network with decorrelated noise the problem would go away. Since we already have two networks, the idea is to use $y = r + \gamma Q_{\phi'}(s', \arg\max_{a'} Q_\phi(s', a'))$ rather than $y = r + \gamma Q_{\phi'}(s', \arg\max_{a'} Q_{\phi'}(s', a'))$.

You can compare the stability of the learning process with or without DDQN in the lunar lander environment.

4 Actor Critic

Actor-critic algorithms implement a network for both the policy (actor) and the value or Q-function (critic). DDPG [2] is a very popular example. It is an interesting algorithm because it was introduced as an alternative to policy gradient for deterministic policies, and can also be seen as an alternative to Q-learning for continuous actions. A key feature is that it works off-policy.

1. take some action $\mathbf{a}_i$ and observe $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$, add it to $\mathcal{B}$
2. sample mini-batch $\{\mathbf{s}_j, \mathbf{a}_j, \mathbf{s}'_j, r_j\}$ from $\mathcal{B}$ uniformly
3. compute $y_j = r_j + \gamma \max_{\mathbf{a}'_j} Q_{\phi'}(\mathbf{s}'_j, \mu_{\theta'}(\mathbf{s}'_j))$ using *target* nets $Q_{\phi'}$ and $\mu_{\theta'}$
4. $\phi \leftarrow \phi - \alpha \sum_j \frac{dQ_\phi}{d\phi}(\mathbf{s}_j, \mathbf{a}_j)(Q_\phi(\mathbf{s}_j, \mathbf{a}_j) - y_j)$
5. $\theta \leftarrow \theta + \beta \sum_j \frac{d\mu}{d\theta}(\mathbf{s}_j) \frac{dQ_\phi}{d\mathbf{a}}(\mathbf{s}_j, \mathbf{a})$
6. update ϕ' and θ'

Figure 3: DDPG: deep deterministic policy gradient (source: [1])

4.1 Pendulum - Implement the critic and actor updates

You can try out your code in the pendulum environment.

- Hint: For the actor, use the chain rule: $\nabla_\theta Q_\phi\left(s, \mu_\theta(s)\right) = \nabla_a Q_\phi\left(s, a = \mu_\theta(s)\right) \nabla_\theta \mu_\theta(s)$.

References

[1] Sergey Levine. *CS285: Deep Reinforcement Learning*. Fa2019. URL: http://rail.eecs.berkeley.edu/deeprlcourse/.

[2] Timothy P Lillicrap et al. "Continuous control with deep reinforcement learning". In: *arXiv preprint arXiv:1509.02971* (2015).

[3] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (2015), pp. 529–533.

[4] Ronald J Williams. "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Machine learning* 8.3-4 (1992), pp. 229–256.