Benjamin Evans
bdevans@sun.ac.za

Deep Reinforcement Learning (DRL)

An introduction for engineering application

https://github.com/BDEvan5/BaselineDRL

2023-02-10

1 Introduction

This document introduces deep reinforcement learning (DRL) for use in engineering applications. The desire is to provide a balance between providing enough information to understand how the algorithms work and remaining simple enough to be easily consumed. The document also accompanies a repository that includes all the algorithms discussed.

This document will be useful to you if after reading it you study the coded examples until you are able to replicate them independently.

1.1 Problem Description

Reinforcement learning (Rl) is typically used to solve difficult problems that conventional solution methods cannot model and control. Figure 1 shows the reinforcement learning problem formulation where we have an agent that selects actions that affect the environment. The environment returns a state telling the agent the condition of the environment and a reward indicating how good or bad the state/action combination is.

DRL Aim: *Train agents to select actions that maximise the sum of cumulative rewards.*

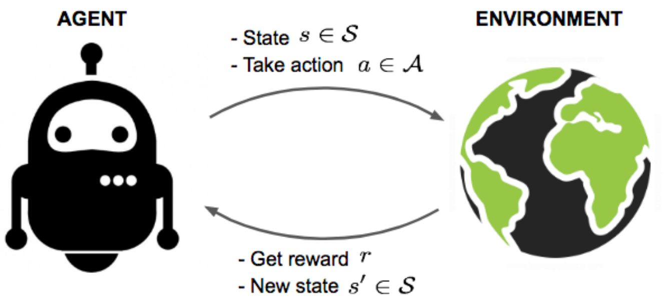
AGENT
ENVIRONMENT
- State $s \in \mathcal{S}$
- Take action $a \in \mathcal{A}$
- Get reward r
- New state $s' \in \mathcal{S}$

Figure 1: The standard reinforcement learning configuration with the agent receiving a state, selecting an action and the environment returning a reward.

1.2 Mathematical Basis

Formally, we consider the RL problem as a Markov Decision Process (MDP). The Markov property is that each state depends only on the previous state, not the history, $\mathbb{P}[s_{t+1}|s_t] = \mathbb{P}[s_{t+1}|s_1, ..., s_t]$. Formally, MDPs have five components:

- $\mathcal{S}$ - State space: The state $s \in \mathcal{S}$ represents the condition of the environment.
- $\mathcal{A}$ - Action space: The action $a \in \mathcal{A}$ is the quantity selected by the agent that is used to control the environment.
- $\mathbb{P}$ - Transition probability function
- R - Reward function: Used to calculate rewards
- γ - Reward discount factor

The state and action spaces are all the possible states the environment can be in and all the actions that could be selected. The transition function is the probability of a next state s' (or s_{t+1}) occurring for a given state-action combination, $\mathbb{P}(s', r|s, a)$. The reward function maps the state and action spaces to a reward as $\mathcal{S} \times \mathcal{A} \mapsto R$ and is used to calculate the reward for the agent. The discounting factor γ, usually 0.9 or 0.99, is how much future rewards are discounted, i.e. how much are present rewards better than future rewards.

Policy: We refer to agents using a policy to select actions. The policy, denoted by π, is the mapping of a state to an action. The "deep" in deep reinforcement learning refers to the use of a deep (multi-layered) neural network to represent the agent's policy. We write that the agent selects action a using its policy for a given state as $a = \pi(s)$.

Return: The return is defined as the sum of discounted rewards. The return can be simplified from an infinite sum to the sum of the reward at the next timestep plus the discounted return. The return G_t starting from time t using discount factor $\gamma \in [0, 1)$ is:

$$\begin{aligned} G_t &= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... \\ &= \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \\ &= r_{t+1} + \gamma G_{t+1} \end{aligned} \tag{1}$$

Value Functions: We define the state-value as the expected return if we start in state s at time t and follow policy π as $V_\pi(s) = \mathbb{E}_\pi[G_t|s]$. Similarly, we define the action-value (Q-value) for a state-action pair as $Q_\pi(s, a) = \mathbb{E}_\pi[G_t|s, a]$. The expectation operator $\mathbb{E}$ is used because the value are sampling based estimates of future returns.

Bellman Equation: The Bellman equation allows us to represent the state-values and Q-values as functions of the immediate reward plus the discounted future rewards.

$$\begin{aligned} V(s) &= \mathbb{E}[r_{t+1} + \gamma V(s')|s] \\ Q(s, a) &= \mathbb{E}[r_{t+1} + \gamma \mathbb{E}_{a\sim\pi} Q(s', a)|s, a]. \end{aligned} \tag{2}$$

The Bellman equation for the Q-value depends on the reward and the discounted Q-value for the next state. The Q-value for the next state is an estimated value depending on an action being selected by the policy $a \sim \pi$.

1.3 Learning Algorithm Components

The general flow of a DRL algorithm is to collect experience by interacting with the environment, store that experience in a buffer and then use the experience to train the agent. When considering a learning algorithm there are a few important components to be aware of:

- **Neural Network:** The neural network that represents the policy. Its input is the size of the state vector and its output is the size of the action space.
- **Replay Buffer:** The replay buffer stores past experiences that are used to train the agent. After each action has been implemented, the state, action, reward, next state, done tuple is stored.
- **Training Loop:** The training loop is the code that repeatedly takes actions, stores the experience and trains the agent.
- **Learning Algorithm:** The learning algorithm is the code that trains the neural network. In addition to the specific replay buffer and networks required, it includes:
  - `act()`: The act method uses the neural network to select an action based on the state vector.
  - `train()`: The train method samples from the replay buffer, then calculates a loss function for the networks and uses the optimiser to adjust the network parameters to minimise the loss.

In the accompanying repository, the buffers, loops and networks are kept as standard as possible to allow for the learning algorithm to be the focus.

2 Value-based Methods (DQN)

We describe the deep-Q-network algorithm, which is the most popular value-based method. The thrust of Q-learning is to use calculate Q-values for each state-action combination. At each step, the action is selected that leads to being in the state with the highest Q-value. The DQN algorithm is limited to discrete action spaces and is ideal for contexts with a small number of actions.

Algorithm 1 DQN Algorithm

```
1: Initialize replay memory $D$ to capacity $N$
2: Initialize action-value function $Q$ with random weights $\theta$
3: Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
4: for Episode = 1:M do
5:     Reset the simulator $s = s_0$
6:     for $t = 1, T$ do
7:         With probability $\epsilon$ select a random actions $a_t$
8:         otherwise select $a_t = \text{argmax}_a Q(s_t, a; \theta)$
9:         Execute action $a_t$ in emulator and observer reward $r_t$ and state $s_{t+1}$
10:        Set $s_{t+1} = s_t$
11:        Store transition $(s_t, a_t, r_t, s_{t+1})$ in $D$
12:        Sample random mini-batch of transitions $(s_j, a_j, r_j, s_{j+1})$ from $D$
13:        Set $y_i = \begin{cases} r_j & \text{if episode terminates at step} j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(s_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
14:        Perform a gradient descent step on $(y_i - Q(s_j, a_j; \theta))^2$ with respect
15:        to the network parameters $\theta$
16:        Every $C$ steps reset $\hat{Q} = Q$
17:    end for
18: end for
```

Algorithm 1 presents the DQN algorithm as presented in the original work (Minh. et al., 2015). The algorithm is modified from using preprocessed images to directly using an available state vector. Lines 1-3 initialise the replay memory (replay buffer) and the Q-networks. Two neural networks are used for the Q-function, a model Q and a target $\hat{Q}$. The model is used to select actions and the target is used to update the target values in line 13. This is done in almost all DRL algorithms to improve stability by preventing the agent from changing too quickly. Every C steps, the target is updated to the model network.

The task is divided into episodes consisting of interactions starting at an initial state and running until a terminal state is reached. The end of an episode is marked by a Boolean done flag. For each step in each episode (lines 7-16), the agent selects an action, executes the action (line 9), and stores the transition in the memory (line 11). Selecting actions brings the *exploitation versus exploration* challenge, which is when the agent should use its current knowledge to select the best action it can (exploit) or experiment with a random action to see if there is something better to do. The DQN algorithm does this by selecting a random action for a certain percentage ϵ and otherwise selecting a greedy action (line 8). The exploration percentage starts off high at the beginning of training and decreases as the agent gains experience.

After this process of collecting experience is complete the agent is trained by sampling a random mini-batch of transitions from the replay buffer (line 11) and using them to update the networks. Target values are calculated using the Bellman equation (Equation 2) using either the terminal reward or the intermediate reward plus a bootstrapped (estimated) Q-value from the nest state that $\hat{\max}_{a'} Q(s_{j+1}, a')$. The bootstrapped Q-value uses the action that results in the highest Q-value. Gradient descent is used to adjust the networks towards the target values using the mean squared error (MSE) loss (line 14). This process continues for the set amount of training steps.

3 Policy-based Methods

Policy-based methods refer to algorithms that directly learn a policy (mapping of states to actions). Policy-based methods can handle both discrete and continuous action spaces and are thus considered in those categories.

3.1 Discrete Methods

3.1.1 Policy Gradient Algorithm

The REINFORCE algorithm is the most basic policy gradient (PG) method and uses only an actor network that learns a policy of how to select actions for a state. The actor outputs the probability of an action being optimal and then an action is selected by selecting a random sample according to the distribution produced by the actor. The actor is trained using the policy gradient algorithm.

Policy Gradient: The policy objective $J(\theta)$ is what we want the policy (parameterised by θ) to achieve. The policy gradient is the "direction" that the policy should move to maximise the objective and is written as $\Delta\theta = \alpha\nabla_\theta J(\theta)$ where α is the step size. Using the policy gradient, the parameters are updated as $\theta_{k+1} = \theta_k + \alpha\nabla_\theta J(\pi_\theta)|_{\theta_k}$.

Policy Objective: DRL algorithms aim to select actions that maximise the expected sum of rewards. Therefore, the gradient of the policy objective is $\nabla_\theta J(\pi_\theta) = \nabla_\theta \mathbb{E}_{\pi_\theta}[G_t]$, where $\mathbb{E}_{\pi_\theta}[G_t]$ is the expected return following policy π. Note that many formulations represent this equation using the notation of $R(\tau) = \sum_{t=0}^{T} r_t$, where τ is a trajectory. The policy gradient can be derived[1] as

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(a_t|s_t) G_t] \tag{3}$$

In the equation $\log \pi_\theta(a_t|s_t)$ is the log probability of selecting an action for a given state using policy π. The equation describes the policy gradient as the gradient of the log probability of selecting an action multiplied by the return produced by that action. Simply put, the chances of selecting an action are multiplied by the return earned by the action. This intuitively makes sense, you want to select actions that produce high rewards more often and decrease the chance of selecting an action that earns a low reward. The ∇_θ refers to the derivative (gradient) with respect to the parameters θ.

Algorithm 2 Policy Gradient Algorithm

```
1: Initialise policy parameters $\theta_0$
2: for $k = 0, 1, 2, ...$ do
3:     Collect a set of trajectories $\mathcal{D}_k = \tau_i$ by using policy $\pi_k = \pi(\theta_k)$
4:     Compute the rewards-to-go (return) $G_t$
5:     Estimate the policy gradient as
           $$g_k = \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)|_{\theta_k} G_t \tag{4}$$
6:     Compute the policy update using gradient descent, $\theta_{k+1} = \theta_k + \alpha g_k$
7: end for
```

Algorithm 2 shows the pseudo-code for the policy gradient algorithm (PG). This is an on-policy algorithm meaning that repeatedly collects a batch of experience $\mathcal{D}_k$ and then uses the batch to update the policy before discarding it. The subscript k denotes the batch number. For each set of collect trajectories, the return G_t is calculated at the end of each episode using the reward-to-go. The reward-to-go starts at the end of the trajectory and recursively calculates the discounted sum of rewards for each step in the trajectory. The policy gradient is calculated by multiplying the log probability of each action in the batch with the computed return (line 5). The policy is then updated using gradient descent (line 6).

[1]Full derivation: https://spinningup.openai.com/en/latest/spinningup/rl_intro3.html

3.1.2 Advantage Actor-Critic (A2C)

Actor-critic methods combine an actor network that represents the policy and a critic network that estimates the value of states. The critic network is a value network that learns a value for each state $V(s)$. The value is used to calculate the policy gradient used to update the actor.

Advantages: While the PG algorithm multiplies the log probabilities by the return G_t, the A2C algorithm multiplies them by the advantage A_t. The advantage for following policy π is defined as

$$\begin{aligned} A^\pi(s_t, a_t) &= Q^\pi(s_t, a_t) - V^\pi(s_t) \\ &= G_t - V(s_t). \end{aligned} \tag{5}$$

The advantage is calculated by taking the difference between the return (reward-to-go) and the value that is estimated by the value function. Intuitively, the advantage is how much better or worse a specific action is compared to the value of the state. Using the advantage instead of the return is shown to reduce variance thus improving training stability.

3.1.3 Proximal Policy Optimisation (PPO)

The proximal policy optimisation (PPO) algorithm improves the A2C algorithm by clipping the policy gradient updates.

PPO is based on the "surrogate" trust-region-policy-objective that replaces the log probability with the ratio between the old and the new probabilities. The ratio is denoted as

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}, \tag{6}$$

where θ_{old} are the parameters before the update and θ are the parameters that are updated. This surrogate objective means that the gradient updates are larger for actions that were more probable under the old policy π_{old}. The surrogate objective is only effective when the old and the new policies are similar. Therefore the TRPO algorithm clipped the update using a constraint on the KL divergence and requires a second-order optimisation. PPO improves over TRPO by removing the need for constrained optimisation by rather clipping the objective to keep the old and new policies close to each other. The new objective is written as,

$$\nabla_\theta J^{\text{CLIP}}(\pi_\theta) = min\Big(r_t(\theta)A_t, clip\Big(r_t(\theta), 1-\epsilon, 1+\epsilon\Big)A_t\Big). \tag{7}$$

The clipped objective takes the minimum of the surrogate objective and the clipped surrogate objective ensuring that the policy remains close to the old policy.

3.2 Continuous Methods

Coming soon....

3.2.1 Deep Deterministic Policy Gradients (DDPG)

3.2.2 Twin-delayed-DDPG (TD3)

3.2.3 Soft Actor-Critic (SAC)

4 Application Notes

5 Useful Resources