Benjamin Evans
bdevans@sun.ac.za

Deep Reinforcement Learning (DRL)
An introduction for engineering application
https://github.com/BDEvan5/BaselineDRL

2023-02-16

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.

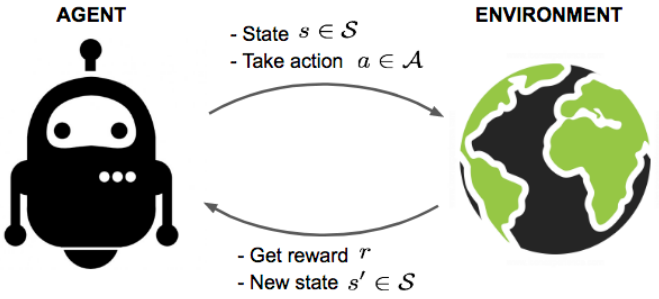

Figure 1: The standard reinforcement learning configuration with the agent receiving a state, selecting an action and the environment returning a reward.

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: The expected return (often denoted using the expectation operator $\mathbb{E}$) represents the anticipated sum of rewards for following a policy. The expectation is used because the values are estimates of future returns as opposed to true values. 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 $Q_\pi(s, a) = \mathbb{E}_\pi[G_t|s, a]$.

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 [5]. The DQN algorithm is limited to discrete action spaces and is ideal for contexts with a small number of actions. The thrust of Q-learning is to use calculate values for each state-action combination that can represent the expected rewards and can be used to select actions that will maximise the reward. The critical components of the Q-networks, action selection and network updates are described, followed by the pseudo-code in Algorithm 1.

Algorithm Flow: The task is divided into episodes (line 4) 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 d. For each step in each episode, the agent selects an action a, that is executed (line 7,8). The transition of state, action, reward, next state and done $(s_t, a_t, r_t, s_{t+1}, d_t)$ is stored in the replay buffer D (line 11). DQN is an off-policy algorithm meaning that after each action has been executed, a random mini-batch is sampled from the replay buffer and used to update the networks (line 12-14).

Algorithm 1 DQN Algorithm

1: Initialize replay buffer R
2: Initialize action-value function Q with random weights ϕ
3: Initialize target action-value function Q' with weights $\phi' = \phi$
4: **for** Episode = 1:M **do**
5: Reset the simulator $s = s_0$
6: **for** $t = 1, T$ **do**
7: With probability ϵ select a random actions a_t
8: otherwise select greedy action using Equation 3
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 R
12: Sample random mini-batch of N transitions (s_j, a_j, r_j, s_{j+1}) from R
13: Calculate targets using Equation 4
14: Adjust the model network parameters ϕ to minimise the loss in Equation 5.
15: Every C steps reset $Q' = Q$
16: **end for**
17: **end for**

Q-Networks: The DQN algorithm uses two neural networks to learn Q-values which estimate the value of an action for each state. Two networks are used, a model Q and target Q' that have the parameters (neural network weights and biases of ϕ and ϕ' respectively. The model network Q is used to select actions and the target network Q' is used to calculate the targets for the updates. After every C steps the target is updated to be the same as the model. Two networks are used to improve the stability of the learning by slowing down the effect of the policy updates on the calculation of the targets.

Action Selection: 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. The exploration percentage starts off high at the beginning of training and decreases as the agent gains experience. Generally, the exploration percentage is decreased by multiplying it with a decay rate after a certain number of steps. The agent selects greedy action by selecting the action with the highest Q-value. Mathematically, the "argmax" operator is used and written as

$$a_t = \text{argmax}_a Q(s_t, a) \tag{3}$$

Network Updates: At each training step, a random mini-batch is sampled from the replay buffer. For each sample, a target is calculated using the Bellman equation (Equation 2). The target is based on the immediate reward and if the next state is not terminal, an estimated value of the next state is added. The estimates next state value is calculated as the max over the action space of the Q-values in the next state. The target is thus

calculated as

$$y_j = \begin{cases} r_j & \text{if episode terminates at step} j+1 \\ r_j + \gamma \max_{a'} Q'(s_{j+1}, a') & \text{otherwise} \end{cases} \tag{4}$$

Gradient descent is used to adjust the networks towards the target values using the mean squared error (MSE) loss (line 14). The mean squared error defines the loss as the squared difference between the target y_j and the current Q-value $Q(s_j, a_j)$ for each sample j and is calculated as

$$L = \sum_j (y_i - Q(s_j, a_j))^2. \tag{5}$$

The networks are updated by adjusting the neural network parameters ϕ to minimise the loss.

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 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 Objective: 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. The ∇_θ operator means that the gradient (derivative) is taken with respect to the parameters theta. Using the policy gradient, the parameters are updated as $\theta_{k+1} = \theta_k + \alpha\nabla_\theta J(\pi_\theta)|_{\theta_k}$. The subscript k refers to the iteration of the policy, i.e. the policy is initialised as θ_0, then is updated to θ_1, θ_2 etc.

Algorithm Flow: 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 batch is denoted by the batch number k and consists of a number of trajectories. 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).

Policy Gradient: 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

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(a_t|s_t) G_t] \tag{6}$$

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

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

Algorithm 2 Policy Gradient Algorithm

Initialise policy parameters θ_0
for $k = 0, 1, 2, ...$ **do**
    for $n = 0, 1, 2, ...$ trajectories **do**
        Sample action a according to policy π_k
        Execute action a and observe s', r, d
        Store transition $(s_t, a_t, s'_t, r_t, d_t)$ in experience batch $\mathcal{D}_k$
    end for
    Compute the rewards-to-go (return) G_t for each trajectory
    Estimate the policy gradient using Equation 10
    Compute the policy update using gradient descent, $\theta_{k+1} = \theta_k + \alpha g_k$
end for

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.

For a batch of trajectories $\mathcal{D}_k$ the policy gradient g_k is estimated as the gradient of the mean of the log probability multiplied by the return. In other words, the parameters are adjusted to maximise the log probability multiplied by the return for each of the steps in the batch of trajectories. Mathematically, the update equation is written as

$$g_k^\theta = \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) G_t. \tag{7}$$

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.

Critic Training: The critic is trained to estimate the return by minimising the difference between the actual return G_t and the current value $V(s)$. The equation used to calculate the critic loss is

$$L = \sum_{t=0}^{T} (G_t - V(s_t))^2. \tag{8}$$

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 is calculated as

$$\begin{aligned} A(s_t, a_t) &= Q(s_t, a_t) - V(s_t) \\ &= G_t - V(s_t). \end{aligned} \tag{9}$$

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. The policy gradient using the advantage is written as

$$g_k^\theta = \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) A_t. \tag{10}$$

3.1.3 Proximal Policy Optimisation (PPO)

The proximal policy optimisation (PPO) algorithm improves the A2C algorithm by clipping the policy gradient updates [7].

Algorithm Flow: The pseudo-code for the PPO on-policy actor-critic algorithm is shown in Algorithm 3. The algorithm uses the same actor-critic setup as A2C, with the difference being how the gradient updates are applied.

Algorithm 3 PPO Algorithm

1: Initialise policy parameters θ_0 and value function parameters ϕ_0
2: **for** $k = 0, 1, 2, ...$ **do**
3: Collect a set of trajectories $\mathcal{D}_k = \tau$ by using policy $\pi_k = \pi(\theta_k)$
4: Compute the rewards-to-go R_t
5: Compute the generalised advantage estimates A_t recursively using the current value function V_{θ_k}
6: Calculate surrogate objective using Equation 13
7: Estimate the policy gradient using clipped objective in Equation 14
8: Compute the policy update using gradient descent, $\theta_{k+1} = \theta_k + \alpha g_k$
9: Fit the value function using mean-squared error in Equation 8
10: **end for**

Generalised Advantage Estimation: Generalised advantage estimation (GAE) [6] is an improvement on how to estimate the advantage. Previously, the advantage was calculated by subtracting the current value from the return to go as $A_t = G_t - V)s_t)$, where the return is the discounted sum of rewards until the end of the episode $G_t = r_0 + \gamma r_1 + \gamma^2 r_2 + ...$ This approach has a high variance (can cause the network to overfit to specific experiences). This is because each update is only related to the experience used.

Another way that the advantage can be calculated is $A_t = r_t + \gamma V(s_{t+1}) - V(s_t)$. This equation uses the bootstrapped (estimated) return as $G_t = r_0 + \gamma V(s_1)$. This approach uses only a single reward in each update, resulting in it having low variance but high bias. The bias is the amount that it can be incorrect or underfit to the experience. The high bias is due to the reward and the estimated next state $V(s_1))$ weighing equally.

GAE combines both of these terms to calculate the lambda exponentially weighted average of the above terms as

$$A_t^{\text{GAE}} = \sum_{t=0}^{\infty} (\gamma\lambda)^t \delta_{t+1} \tag{11}$$

In the equation δ_t is the TD advantage calculated as $V(s_{t+1}) + r_t - V(s_t)$. The equation can be simplified to a recursive equation

$$A_0^{\text{GAE}} = \delta_0(\gamma\lambda)A_1^{\text{GAE}} \tag{12}$$

PPO Objective: 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

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}, \tag{13}$$

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 clipping the objective to keep the old and new policies close to each other. The new objective is written as

$$\nabla_\theta J^{\text{CLIP}}\big(\pi_\theta\big) = min\bigg(r_t(\theta)A_t, clip\Big(r_t(\theta), 1-\epsilon, 1+\epsilon\Big)A_t\bigg). \tag{14}$$

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

3.2.1 Deep Deterministic Policy Gradients (DDPG)

The deep deterministic policy gradient (DDPG) algorithm is the base which introduces the idea of continuous control [4]. Continuous policy-based methods use an actor-critic structure where the actor outputs a number of continuous actions equal to the number of dimensions in the action space. The critic is a Q-network trained to learn Q-values for state-action pairs. The Q-values are used to train the actor-network to select actions with high Q-values. The critical parts of the DDPG algorithm are explained followed by the pseudo-code that ties the steps together.

Algorithm Setup: The continuous actor-network is referred to as $\mu(s|\theta^\mu)$ where θ^μ are the weights. The algorithm starts by initialising model networks Q and μ and the target networks Q' and μ'. DDPG is an off-policy algorithm meaning that experience is collected and stored ina replay buffer R. For each training step, a batch of N transitions is randomly sampled from the buffer.

Algorithm 4 DDPG Algorithm

1: Randomly initialize the actor μ critic $Q(s, a)$ networks with parameters θ and ϕ.
2: Initialise target networks μ' and Q' and with parameters $\theta' \leftarrow \theta,\ \phi' \leftarrow \phi$
3: Initialize replay buffer R
4: **for** $t = 1, T$ **do**
5: Select action according to current policy and exploration noise (Equation 15
6: Execute action a_t and observer reward r_t and new state s_{t+1}
7: Store transition (s_t, a_t, r_t, s_{t+1}) in R
8: Sample a random minibatch of N transitions (s_j, a_j, r_j, s_{j+1}) from R
9: Calculate targets using Equation 16
10: Update critic by minimizing loss in Equation 17
11: Update the actor policy using the gradient in Equation 18
12: Apply soft update to target networks in Equation 19
13: **end for**

Action Selection: Since the DDPG algorithm is deterministic in nature, noise is added to each action for exploration. Actions are calculated using

$$\pi(s_t) = a_t = \mu(s_t) + \mathcal{N}_t, \tag{15}$$

where $\mathcal{N}_t$ is the sampled noise value. Noise is sampled from either an Ornstein–Uhlenbeck process or a normal distribution.

Critic Updates: The training happens after every step by sampling a random minibatch of N transitions from the replay buffer. The Q-value targets y_i for each transition i are calculated using the Bellman equation by adding the reward r_i and a discounted bootstrapped estimate of the next state value as

$$y_j = r_j + \gamma Q'(s_{j+1}, \mu'(s_{j+1})). \tag{16}$$

The bootstrapped next state value is the Q-value for the next states and the actions that the target policy would select for the next states $\mu'(s_{j+1})$. The critic is updated by minimising the mean squared error (MSE) loss,

$$L = \frac{1}{N} \sum_j Q(y_j - Q(s_j, a_j))^2. \tag{17}$$

Policy Gradient: The policy objective is to maximise the expectation of Q-values by following the policy $J(\theta) = \mathbb{E}[Q(s, \mu(s))]$. The policy gradient goes through the Q-function to select actions that lead to increased Q-values. The policy gradient with respect to θ is written as $\nabla_\theta J(\theta) = \nabla_\theta Q(s, \mu(s))$. For the minibatch, the gradient g is estimated as

$$g^\theta = \frac{1}{N} \sum_j \nabla_\theta Q(s_j, \mu(s_j)) \tag{18}$$

Soft Updates: Soft updates are used to gradually update the actor and critic target networks towards the model networks. for each parameter, the soft updates are applied as

$$\phi \leftarrow \tau\phi + (1-\tau)\phi' \qquad \theta \leftarrow \tau\theta + (1-\tau)\theta' \tag{19}$$

3.2.2 Twin-delayed-DDPG (TD3)

The twin-delayed deep deterministic policy gradient (TD3) algorithm features the three improvements over DDPG of, (1) using a pair of Q-networks, (2) delaying policy updates, and (3) target policy smoothing [1]. The thrust of the improvements is to reduce overestimation error and reduce the variance of the updates.

Q-Networks: DDPG employs a critic to learn a Q-value that is used to train the actor. The authors of TD3 show that this approach results in overestimating the Q-value. They propose using two critic networks and using the minimum between them to calculate the bootstrapped target. The same target is used to train both of the networks.

Delayed Updates: Using separate target and model networks is known to improve training performance due to slowing down the network's ability to change. The authors show that updating the policy networks at a lower frequency results in more stable learning. This is due to the targets being more stable. Typically, the policy is updated after every second update of the critic networks.

Target Policy Smoothing: They address the concern of the network overfitting to a narrow peak in the value function by adding random noise to the actions that are used to estimate the value targets. The random noise means that they are calculating an estimate of the Q-value for a similar policy to the current one. This stochasticity is shown to improve performance. The noise is sampled from a normal distribution $\mathcal{N}$ with deviation σ and clipped using a constant c.

Mathematically, adding the target policy smoothing to the pair of Q-networks results in calculating the target y value as

$$\begin{aligned} y =& r + \gamma \min_{i=1,2} Q_{\theta'_i}(s', \mu(s') + \epsilon) \\ \epsilon \sim& \text{clip}(\mathcal{N}(0,\sigma), -c, c) \end{aligned} \tag{20}$$

In the equation, γ is the discount factor, i is the number of the Q-network (i.e. $Q_{\theta'_1}, Q_{\theta'_2}$), μ is the actor network, ϵ is the clipped noise, and c is the noise clipping constant

3.2.3 Soft Actor-Critic (SAC)

The soft actor-critic was developed due to the DDPG algorithm's brittleness to hyperparameters [3]. The SAC algorithm trains a stochastic policy to select actions that maximise entropy. Entropy across a distribution of probabilities is a measure of the amount of randomness or unpredictability of events. The entropy $\mathcal{H}$ across a distribution of probabilities $\mathbf{p}$ is the negative sum of the probability of an event p_i multiplied by the log of the probability and is calculated as $\mathcal{H}(\mathbf{p}) = \Sigma_i \; p_i \log_2(p_i)$.

Algorithm Setup: While several versions of the algorithm exist we describe the one in [2] using stochastic actions and auto-tuning of the alpha value. The SAC algorithm uses a single stochastic policy π parameterised by θ and two Q-networks with separate model ϕ_1, ϕ_2 and target ϕ'_1, ϕ'_2 networks. It retains the off-policy flow of keeping a replay buffer R and randomly sampling a minibatch B for training. Training consists of updating the policy network, Q-networks and the entropy temperature. Some implementations use multiple training steps for each sample collected. After the networks have been trained, soft updates are applied to update the targets towards the model network parameters (as in Equation 19).

Policy Reparamertisation: The SAC algorithm uses a stochastic policy π_θ for the actor that learns a distribution over the action space. The stochastic policy uses the actor to learn a mean μ_θ and standard deviation σ_θ for each action dimension. The SAC requires the environment to use normalised actions, meaning that the actions are in the range $[-1, 1]$. Actions are calculated by taking the tanh of the mean $\mu_\theta(s)$ plus the standard deviation $\sigma_\theta(s)$multiplied by a value ξ sampled from a normal distribution $\mathcal{N}(0,1)$ as,

$$a = \pi_\theta(s) = \tanh\big(\mu_\theta(s) + \sigma_\theta(s)\xi\big), \qquad \xi \sim \mathcal{N}(0,1). \tag{21}$$

Algorithm 5 SAC Algorithm

Initialise policy π with parameters θ and pair of Q-functions with parameters ϕ_1, ϕ_2
Initialise Q-function target parameters equal to model parameters $\phi_1' \leftarrow \phi_1, \phi_2' \leftarrow \phi_2$
Initialize replay buffer R
for Step = 1:M **do**
    Select action using stochastic policy, Equation 21
    Execute a and observe next state s', reward r and done d
    Store (s, a, s', r, d) in replay buffer R
    for each training iterations **do**
        Randomly sample minibatch from R
        Update policy weights using Equation 23
        Compute Q targets using Equation 24:
        Update Q-functions using gradient descent:
        Update alpha using Equation 26
        Apply soft update to the Q-target networks
    end for
end for

Policy Objective: The SAC algorithm modifies the policy objective to the expectation of the reward plus the entropy across the action distribution from state-action pairs generated by following policy π. The weight of the entropy term is controlled by a temperature parameter α. The new objective is written as

$$J(\pi) = \mathbb{E}_{(s_t,a_t)\sim\rho_\pi}[r(s_t, a_t) + \alpha\mathcal{H}(\pi(\cdot|s_t))]. \tag{22}$$

Policy Gradient: The policy is updated by maximising the Q-values for the actions selected by the policy. Unlike TD3 which always used the same Q-network to train the policy, SAC uses the minimum Q-value $\min_{i=1,2} Q_{\phi_i}$ to train the policy. Additionally, the policy includes an entropy loss $-\alpha \log \pi_\theta \left(\pi_\theta(s)|s\right)$ that encourages the policy to have a large standard deviation across actions. The policy gradient is calculated as

$$g = \sum_j \nabla_\theta \min_{i=1,2} Q_{\phi_i}(s_j, \tilde{a}_j) - \alpha\nabla_\theta \log \pi_\theta \left(\tilde{a}_j|s_j\right) \qquad \tilde{a}_j \sim \pi_\theta(\cdot|s_j). \tag{23}$$

Q-value Updates: The SAC uses two Q-networks in a similar way to the TD3 algorithm by calculating targets and then using the MSE loss. The targets for updating the Q-networks are calculated using the reward and the discounted bootstrapped Q-value from the next state. The bootstrapped Q-values are calculated using the actions that the policy network would take in the next state (next actions) $\tilde{a}' \sim \pi_\theta(\cdot|s')$. The target includes an entropy term for the next actions. The Q-value target is,

$$y = r + \gamma \left(\min_{i=1,2} Q_{\phi_i'}(s', \tilde{a}') - \alpha \log \pi_\theta(\tilde{a}'|s') \right), \qquad \tilde{a}' \sim \pi_\theta(\cdot|s') \tag{24}$$

The MSE loss is used to update both the model Q-networks towards the target.

Temperature Updates: In the entropy loss terms, the parameter α is the temperature parameter that controls how much the entropy loss counts compared to the original loss. While in earlier versions of SAC this was a hyperparameter, the latest algorithms feature auto-tuning of α toward a desired entropy $\tilde{\mathcal{H}}$. The entropy temperature objective is written as

$$J(\alpha) = \mathbb{E}_{a\sim\pi}[-\alpha \log \pi_\theta(a|s) + \alpha\tilde{\mathcal{H}}] \tag{25}$$

The temperature gradient is calculated as

$$\nabla_\alpha J(\alpha) = \sum_j \nabla_\alpha \Big(-\alpha(\log \pi_\theta(\tilde{a}_j|s_j) + \tilde{\mathcal{H}}) \Big) \qquad \tilde{a}_j \sim \pi_\theta(\cdot|s_j). \tag{26}$$

4 Conclusion: Applying Algorithms to Engineering Problems

If you made it to the end - congratulations. Reinforcement learning has a steep learning curve and the algorithms grow quickly in complexity. My advice on using DRL algorithms consists of several steps; firstly,

you should seek to understand the theory and the maths behind the algorithms (this document). Secondly, you should gain a practical understanding of how the algorithm work in code. You can use the associated repository for examples and see many other implementations online. The implementations in the repo are as simple as posssible and all follow the same structure to aid learning. This step includes becoming comfortable with Python and a machine learning library like PyTorch. The third step is to understand your problem and configure it for DRL. This generally means identifying the state space, action space and reward function for a given application.

If you find any mistakes in the document or think that things could be better explained, please reach out to me at `bdevans@sun.ac.za`

References

[1] Scott Fujimoto, Herke Hoof, and David Meger. "Addressing function approximation error in actor-critic methods". In: *International conference on machine learning*. PMLR. 2018, pp. 1587–1596.

[2] Tuomas Haarnoja et al. "Soft actor-critic algorithms and applications". In: *arXiv preprint arXiv:1812.05905* (2018).

[3] Tuomas Haarnoja et al. "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor". In: *International conference on machine learning*. PMLR. 2018, pp. 1861–1870.

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

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

[6] John Schulman et al. "High-dimensional continuous control using generalized advantage estimation". In: *arXiv preprint arXiv:1506.02438* (2015).

[7] John Schulman et al. "Proximal policy optimization algorithms". In: *arXiv preprint arXiv:1707.06347* (2017).