The Ministry of Silly Walks

Learning to Walk Funny But Efficiently

COMP3667 Reinforcement Learning Assignment 2022/2023

Piotr Borowiecki (svmm25)
PIOTR.BOROWIECKI@DURHAM.AC.UK

Abstract

This paper presents a re-implementation of the Randomized Ensembled Double Q-Learning (REDQ) [1] with the Soft Actor Critic (SAC) [2] algorithm for solving the OpenAI Gym [3] BipedalWalker-v3 continuous environment. The motivation behind this work is to demonstrate the effectiveness of such approach in solving one of the most fundamental problems in modenr Reinforcement Learning, namely the Value (or Action-Value) Function overestimation. Results show that the considered algorithm is robust and sample-efficient, although the way the Agent walks in the end might not entirely resemble humans. Several sources were consulted and the accompanying code is based on existing implementations [4, 5, 6].

1 Background

Reinforcement Learning (RL) is a subfield of Artificial Intelligence (AI) concerned with training agents to make decisions in environments, in order to maximize rewards they receive. It is fundamentally based on Markov Decision Processes (MDPs) and the Bellman Equation. The equation expresses the relationship between the current state value function and the next state value function which can be used to compute the optimal policy, which is the policy that maximizes the expected cumulative reward over time. In recent years, tremendous progress has been made in the field, allowing agents to achieve spectacular results in a wide range of applications. In this paper, a simple implementation of the Randomized Ensembled Double Q-Learning (REDQ) algorithm with the Soft Actor Critic (SAC) for solving the Gym BipedalWalker-v3 environment is considered, together with an overview of its strengths and limitations. Traditional algorithms often suffer from issues, such as overestimation bias and instability when operating in large, continuous state and action spaces. REDQ addresses these issues by utilizing an ensemble of Q-networks, which reduces the variance of the Q-value estimation and helps preventing overfitting. SAC, on the other hand, uses a soft update mechanism that helps to stabilize the training process and improve the convergence rate. The author believes that this paper can serve as a good example of ensembling techniques in RL, and provide a solid base for further exploration.

Before continuing with REDQ, it is important to understand the base on which it builds, namely Q-Learning [7] and Double Q-Learning [8], which are both model-free algorithms based on the even more fundamental idea of Temporal-difference (TD) Learning [9]. One approach for solving problems that involve making sequential decisions is to estimate the optimal value of each action. The optimal value of an action represents the expected sum of future rewards that can be obtained by taking that action and then following the optimal policy. For a given policy π, the true value of an action a in a state s can be defined as $Q_\pi(s, a)$, which is the expected sum of future rewards starting from the current state and taking action a while following policy π. The discount factor $\gamma \in [0, 1]$ is used to balance the trade-off between immediate and future rewards. The optimal value of an action in a given state, denoted as $Q^*(s, a)$, is defined as the maximum value of $Q_\pi(s, a)$ over all policies π: $Q^*(s, a) = \max_\pi Q_\pi(s, a)$. An optimal policy can be obtained by selecting the action with the highest value in each state. This approach allows us to learn the optimal

policy in a given environment, which maximizes the expected cumulative reward obtained by the agent over time. In many cases, it is not feasible to learn all action values in all states separately, due to the large size of the problem. Instead, we can learn a parameterized value function $Q(s, a; \boldsymbol{\theta}_t)$, which can be updated after taking action A_t in state S_t and observing the immediate reward $A_t + 1$ and resulting state $S_t + 1$. The Q-learning update for the parameters can be expressed as $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \left(Y_t^{\mathrm{Q}} - Q(S_t, A_t; \boldsymbol{\theta}_t)\right) \nabla_{\boldsymbol{\theta}_t} Q(S_t, A_t; \boldsymbol{\theta}_t)$, where α is a scalar step size and the target Y_t^{Q} is defined as $Y_t^{\mathrm{Q}} \equiv R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \boldsymbol{\theta}_t)$. Q-learning updates can be seen as an instance of stochastic gradient descent, as they involve updating the current value function $Q(S_t, A_t; \boldsymbol{\theta}_t)$ towards a target value Y_t^{Q}. By iteratively applying such update, we can learn an approximation of the optimal Q-value function, with the aim of minimizing the Mean Squared Error between the predicted and true value.

One of the major problems with Q-Learning is that it tends to overestimate the Q-value of actions, which can lead to sub-optimal policies. Double Q-Learning was introduced by Hasselt in 2010 to address this problem, proposing the following modification. Unlike in Q-Learning, where the maximum Q-value is estimated by taking the maximum over all actions, including the one that maximizes the Q-value itself, Double Q-Learning uses two Q-value functions - one to select the action and another one to estimate its value. The idea is to randomly switch between the two during training to mitigate the problem. With the advances in Deep Learning, the ideas were then developed further, porting Q-Learning and Double Q-Learning to the realms of artificial neural networks and introducing Deep Q-Networks (DQNs), or Double Deep Q-Networks (DDQNs), [10, 11, 12, 13].

The main idea behind REDQ is to use an ensemble of N Q-Networks for estimating the Q-function. These networks are trained independently on different random subsets of the data, introducing diversity into the ensemble, which helps to reduce the bias. Letting $Q_i(s, a)$ denote an estimate of the action-value function produced by the ith Q-network in the ensemble, the resulting estimate can be expressed as $Q_{\mathrm{REDQ}}(s, a) = \frac{1}{N} \sum_{i=1}^{N} Q_i(s, a)$, where N is the number of Q-networks used. Similarly to the underlying algorithms, the objective can then be formulated as minimizing the mean-squared error (MSE) between the ensemble estimate and the target y: $\mathcal{L} = \frac{1}{2} \mathbb{E}_{(s,a,r,s') \sim \mathcal{D}} \left[\left(Q_{\mathrm{REDQ}}(s, a) - y \right)^2 \right]$, where y is calculated as $y = r + \gamma \min_{a \in \mathcal{A}} Q_{\mathrm{REDQ}}(s', a) + \lambda \cdot \mathrm{var}_{i=1}^{N} Q_i(s, a)$, and λ is the strength of the ensemble regularization - a hyperparameter encouraging diversity among individual Q-networks by penalizing high variance in their predictions. During training, the networks in the ensemble are updated using stochastic gradient descent to minimize the loss function $\mathcal{L}$. The gradient of the loss with respect to the parameters of these networks is computed as $\nabla_{\theta_k} \mathcal{L}(\theta_k) = \frac{1}{N} \sum_{i=1}^{N} \left(y_i - Q_{\mathrm{REDQ}}(s_i, a_i) \right) \nabla_{\theta_k} Q_k(s_i, a_i)$, where y_i is the target value for the ith data point, calculated as $y_i = r_i + \gamma \min_a Q_{\mathrm{REDQ}}(s_i', a)$, and γ is the discount factor.

In 2018, Soft Actor-Critic (SAC) was introduced by Haarnoja et al. as a way to improve the performance of RL algorithms. SAC uses an entropy regularization term to encourage exploration and prevent premature convergence to sub-optimal policies and has been shown to be able to learn complex robotic control tasks. The algorithm learns the policy and two Q-functions, Q_{ϕ_1} and Q_{ϕ_2}. The loss functions for the Q-networks can be expressed mathematically as $L(\phi_i, \mathcal{D}) = \mathop{\mathrm{E}}_{(s,a,r,s',d) \sim \mathcal{D}} \left[\left(Q_{\phi_i}(s, a) - y(r, s', d) \right)^2 \right]$, where the target is given by $y(r, s', d) = r + \gamma(1 - d) \left(\min_{j=1,2} Q_{\phi_{\mathrm{targ}\ j}}(s', \tilde{a}') - \alpha \log \pi_\theta(\tilde{a}' \mid s') \right), \quad \tilde{a}' \sim \pi_\theta(\cdot \mid s')$. The policy is to maximize not only the expected future return, but also the expected entropy. This is achieved using the reparametrization trick and defined as $\max_\theta \mathop{\mathrm{E}}_{\substack{s \sim \mathcal{D} \\ \xi \sim \mathcal{N}}} \left[\min_{j=1,2} Q_{\phi_j}(s, \tilde{a}_\theta(s, \xi)) - \alpha \log \pi_\theta(\tilde{a}_\theta(s, \xi) \mid s) \right]$. The entropy regularization coefficient (α) is responsible for controlling the explore-exploit trade off, with higher values corresponding to more exploration, and lower values corresponding meaning the reverse.

The solution proposed in this paper combines SAC with ideas introduced by REDQ. One reason to choose REDQ with SAC for a continuous space environment, such as BipedalWalker-v3, is that it can be more sample efficient than other algorithms. This is because REDQ with SAC utilizes a distributional Q-function approximation that can better represent the

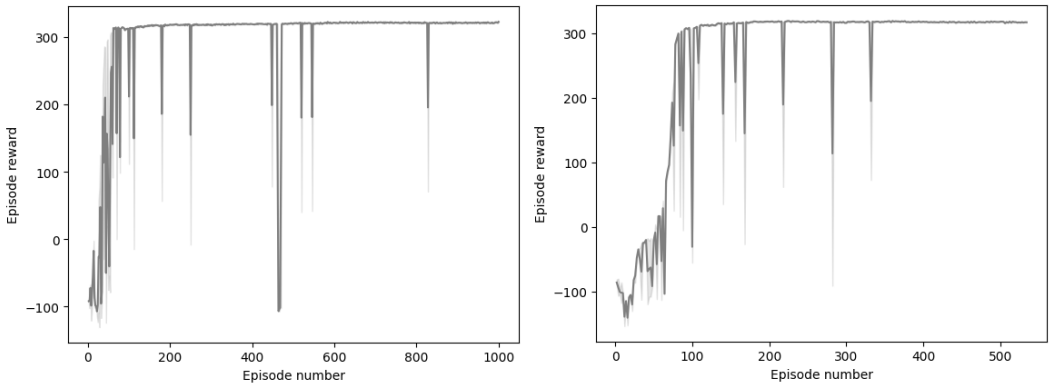
uncertainty in the action-value estimates, leading to more stable learning and better performance.

2 Methodology

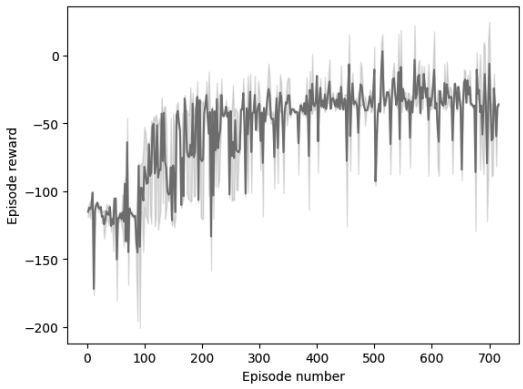
The REDQ-SAC algorithm was implemented and tested with several architectural changes, including adjusting Q-Newtork depth, number of neurons used, and several types of activation functions, but also implementing dropout, batch normalization, skip connections, and a duelling architecture. In this implementation, the Q-values of multiple Q-networks were used to compute the Q-loss, and the minimum of these estimates is used to update the target network. It should be noted that several other approaches exist, such as taking the average, but it was empirically found that using the smallest of the estimated values worked best in this particular environment. The Agent consists of two parts mainly - a Policy Network and a Q-Network. The Q-Network was replicated N times to obtain multiple estimates of the Q-value. The Agent uses the MSE loss for the Q-Network updates and a negative reinforcement signal for the Policy Network update. The updates are performed alternatively, with the Policy Network update being delayed every '`policy_update_delay`' number of iterations. Experience Replay buffer is used for storing and sampling experiences during training, with the Polyak update for improved stability. The Agent also has an adaptive entropy regularization term to promote exploration. Adam optimizer is used for both the policy and all the Q-networks. The three main parameters are N (number of Q-Networks in the ensemble), M (number of Q-values to sample), and G (number of gradient updates to perform before each policy update). Other important parameters include '`random_action_steps`' (number of random experiences actions before the agent starts using the policy), and '`policy_update_delay`' (number of gradient updates to Q-Networks before the Policy Network is updated). The Policy Network consists of two, fully connected layers made of 256 neuron each, with the input layer size equal to the dimensionality of the observation space, and the output layer size equal to the dimensionality of the action space. ReLU activation function was used, but this is configurable via a parameter, since the code currently supports common activation functions such as ReLU, Tanh, and ELU. The stochasticity of the policy is achieved by sampling from a normal distribution with mean equal to the output of the mean layer and standard deviation equal to the exponential of the output of the log standard deviation layer. The Q-Network implementation is a simple feed-forward neural network that outputs a single Q-value. It is composed of a series of fully connected layers with two hidden layers of 256 neurons each. Several other architectures were tried, including deepening the netwrok, trying various numbers of neurons and different activation functions, adding batch normalization, skip connections, dropout, and implementing a dueling architecture. Counterintuitively, it was empirically found that the original, simple architecture performs best, resulting in the Agent achieving the score of 300 faster. Some of the experiments were left commented out in the accompanying Jupyter notebook.

3 Convergence Results

The algorithm is sample efficient and converges fast in simple environments. The results are consistent with every code run, with negligible differences. First positive rewards can be observed after 18 epochs of training already, and the environment gets solved in around 73 episodes. Lower scores appear every now and then as a result of temporarily abandoning the policy for exploration, but the Agent returns to the policy right after and overall maintains the rewards above 300. Around 120 minutes of training are required to achieve this, with a single episode run time being empirically observed to range from 1 to 4 minutes initially. The episode run time increases the longer we train, resulting in 1,000 episodes being achieved in roughly 74 hours (4488 minutes). The training ends with the reward of 323, and - discounting occasional drop - episode rewards oscillate between 318 and 323 for most of it. A chart below shows exact score history across 1,000 episodes of training, with the `random_action_steps` parameter being set to 1,000 (on the left), and the score history across 5340 episodes, with the same parameter being set to 4,000 (on the right).



In terms of the hardcore environment, a clear uptrend in progress was initially observed, but the algorithm appears to have plateaued around episode 300, with a reward oscillating between -129 and 24, and the mean being equal to -36. Judging from the videos recorded, it could be inferred that the agent gets stuck in local minima, where it learns how to avoid falling, but approaches obstacles in a way, that prohibits him from leaving them and continuing the run. It is possible that with longer training time, a breakthrough and further progress could be achieved, leading to environment being solved, but the training had to be stopped due to limited resources. A chart below shows exact score history across 717 episodes of training.



Limitations

Despite excellent convergence properties and consistent results in the simple environment, the Agent does not appear to run, but rather jumps, resembling "The Ministry of Silly Walks" sketch [14] from Monty Python's Flying Circus [15], as can be seen in the accompanying video recording. Nonetheless, it is quick and clearly learned to walk efficiently, achieving high rewards. The algorithm was less successful in the hardcore environment, which remained unsolved after 500 episodes of training. As already mentioned, it is possible that environment would have been solved with longer training. One approach could be to start the hardcore environment training with the agent already trained on basic environment. Allowing the agent to explore more could also help.

Future Work

Methodical hyperparameter tuning should be performed, as the current values were chosen arbitrarily or as a result of emirical experimentation. Different Q-Network and Policy Network architectures were tried, but with no systematical approach allowing for strong

conclusions to be made. Prioritized Experience Replay (PER) [16] could be implemented in place of the current Replay Buffer. Other popular algorithms, such as TD3 [17], A3C [13], or PPO [18] could also be tried with REDQ. A thorough comparison could then be performed, leading to selecting the best candidate before tuning it further.

References

[1] Xinyue Chen et al. "Randomized ensembled double q-learning: Learning fast without a model". In: *arXiv preprint arXiv:2101.05982* (2021).

[2] 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.

[3] Greg Brockman et al. *OpenAI Gym*. 2016. arXiv: `1606.01540 [cs.LG]`.

[4] Lucas Alegre. *sac-plus*. Accessed: December 2022. 2022. URL: `https://github.com/LucasAlegre/sac-plus`.

[5] Yuhao Zhou. *REDQ: PyTorch implementation of "Reinforcement Learning with Ensembles of Diverse Critics"*. Accessed: January 2023. 2021. URL: `https://github.com/watchernyu/REDQ`.

[6] LabML AI. *Neural Networks Lab*. Accessed: January 2023. 2022. URL: `https://nn.labml.ai/`.

[7] Christopher John Cornish Hellaby Watkins. "Learning from delayed rewards". In: (1989).

[8] Hado Hasselt. "Double Q-learning". In: *Advances in neural information processing systems* 23 (2010).

[9] Richard S Sutton. "Learning to predict by the methods of temporal differences". In: *Machine learning* 3 (1988), pp. 9–44.

[10] Volodymyr Mnih et al. "Playing atari with deep reinforcement learning". In: *arXiv preprint arXiv:1312.5602* (2013).

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

[12] Hado Van Hasselt, Arthur Guez, and David Silver. "Deep reinforcement learning with double q-learning". In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 30. 1. 2016.

[13] Volodymyr Mnih et al. "Asynchronous methods for deep reinforcement learning". In: *International conference on machine learning*. PMLR. 2016, pp. 1928–1937.

[14] Monty Python. *The Ministry of Silly Walks*. Sketch. 1970.

[15] Monty Python. *Monty Python's Flying Circus*. TV Series. 1969.

[16] Tom Schaul et al. "Prioritized experience replay". In: *arXiv preprint arXiv:1511.05952* (2015).

[17] 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.

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