Teaching BipedalWalker to Walk with Various Reinforcement Learning Algorithms

Francisco Hu[1], Eddie Xu[2], Syed Affan[3]
Stony Brook University
francisco.hu@stonybrook.edu[1], eddie.xu@stonybrook.edu[2], syed.affan@stonybrook.edu[3]

Abstract

In our implementation, we integrated three reinforcement learning algorithms: Q-learning, Deep Q-Network, and Twin Delayed DDPG. Our purpose was to start with a simple algorithm and with each iteration, build complexity in order to illustrate the difference between each application's performance. Using these methods, each had varying degrees of success training the "BipedalWalker-v3" environment from OpenAI Gym, a toolkit used to develop reinforcement learning algorithms. The agent's ultimate goal was to travel forward as far as it could without falling. We found that the most complex technique, Twin Delayed DDPG, was overwhelmingly successful in performing the task compared to the other two.

1. Introduction

Reinforcement learning is a fascinating area of machine learning in which agents learn to perform a particular task through the concept of maximizing reward. Our goal during this project was to implement several learning algorithms with the agent (BipedalWalker-v3) and compare the performance of each. The agent's task was to formulate a strategy that would yield the largest reward with four actions at any given time-step as its input.

The motivation is to use the reinforcement learning techniques we have learned to apply it to other problems. For instance, teaching a pair of mechanical legs the concept of walking could aid those with disabilities. Another case would be that they could be used to teach cars how to automatically park without driver assistance, a core part of autonomous vehicles. As shown by these examples, reinforcement learning algorithms are important to analyze due to its practical applications in solving real world problems. It is also important to highlight and understand reinforcement learning's limitations.

In this work, we built an application that teaches an agent how to walk using OpenAI's Gym. Reinforcement learning algorithms applied were Q-learning, Deep Q-Network (DQN), and Twin Delayed Deep Deterministic Policy Gradient (TD3) which were implemented by Syed, Francisco, and Eddie, respectively. The result of our work showed that algorithms with more complexity performed better than the simpler ones. We discovered that Q-learning is better suited for smaller learning problems since it requires no model and operates on just the epsilon-greedy policy. We found that DQN performs better than Q-learning, but still does poorly for this particular task because of the continuous action domain. We then implemented TD3 which aims to solve the problems with DQN which demonstrated considerably more success than the other models; however, TD3 was shown to have issues of its own.

2. Description

All of the algorithms use the basic fundamentals of the Q-function that is implemented as an optimal action-value function. This function adheres to the principle of the Bellman equation: in each update, it selects the action that will maximize the estimate of the next time-step's Q-value. This equation is shown as follows,

$$Q(s,a) = Q(s,a) + \alpha\,(r + \gamma \max Q(s',a') - Q(s,a)) \quad (1)$$

in which s is the current state, a is the current action, s' is the next state, a' is the next action, r is the current reward observed, α is the learning rate, and γ is the discount factor for future rewards.

Each algorithm uses some variation of the Q-function as well as several other changes from one another to improve performance. Despite the many optimizations each successive algorithm implements, the algorithms have their own shortcomings that prevent them from perfectly learning the task.

2.1. Q-Learning

Q-Learning is a simple algorithm used in reinforcement learning. Like other algorithms that use the Q-function to

implement the Bellman equation, the algorithm seeks to select actions which would maximize Q-value. The action is chosen with an ε-greedy policy, meaning that the algorithm has ε probability to choose a random action for the next time-step instead of the action that would maximize the Q-value. This behavior helps prevent the algorithm from being stuck in local minima/maxima. In our implementation, we used a dynamically changing value for ε that decays as episodes progress. In early episodes, ε is close to 1, and actions are chosen almost always randomly. After the completion of 500 episodes, the probability of choosing a random action is about 0.5. This algorithm does not make use of a model; possible actions and their associated Q-values are stored on a Q-table.

Algorithm 1: Q-Learning

Initialize $Q(s,a)$
for t = 1 to T **do**
  With probability ε select a random action a_t
    otherwise select $a_t = \max Q(s_t, a_t; \theta)$
  Execute a_t and observe reward r_t and next state s_{t+1}
  $Q(s, a) = Q(s,a) + \alpha (r + \gamma \max Q(s',a') - Q(s,a))$
end for

2.2. Deep Q-Network

DQN keeps features of Q-learning like the ε-greedy policy, but also builds on the algorithm with a deep neural network, experience replay, and stochastic gradient descent for updating weights. This technique has also been used in prior works like playing classic Atari 2600 games (Mnih V. et al. 2015). In the paper, the authors describe the use of the experience replay which works by storing the last N experiences where N is the memory capacity. Each experience is represented as (s,a,r,s',d) where d marks whether or not the experience is in its terminal state. The paper also describes how they improved on the Bellman equation so that the Q-network Q could learn to generalize because the basic approach only calculates the Q-value separately for each state. We used the same technique described in the paper and modified the equation to get the function approximator $Q(s,a;\theta) \approx Q(s,a)$. Using the approximator, Q is trained by adjusting the weights θ at each iteration and performs a stochastic gradient descent on the loss function. This loss function is described as the mean-squared error of the Bellman equation $(y - Q(s, a;\theta))^2$ where y are the targets generated by a second Q-network $\widehat{Q}$ that clones Q every C updates. The purpose of using the second network is to ensure stability since if the network changes frequently, then the targets will also experience many changes, leading to a divergence. Stochastic gradient descent is utilized since it is computationally more efficient than doing gradient descent because it uses a mini-batch of experiences instead of the entire buffer. It has less accuracy due to more noise, but yields a decent approximation.

Algorithm 2 Deep Q-Network

Initialize experience replay E with max capacity N
Initialize Q and $\widehat{Q}$ with random θ weights
for t = 1 to T **do**
  With probability ε select a random action a_t
    otherwise select $a_t = \max Q(s_t, a_t; \theta)$
  Execute a_t and observe reward r_t and next state s_{t+1}
  Store $(s_t, a_t, r_t, s_{t+1}, d_t)$ in E
  Sample mini-batch of $(s_j, a_j, r_j, s_{j+1}, d_j)$ from E
  Generate target $y_j = r_j + \gamma \max \widehat{Q}(s_{j+1}, a';\theta)(1-d)$
  Stochastic gradient descent on $(y_j - Q(s_j, a_j;\theta))^2$
  Every C steps reset $\widehat{Q} = Q$
end for

2.3. Twin Delayed DDPG

TD3 uses the same concepts in DQN by also implementing deep neural networks, experience replay, and stochastic gradient descent. However, the difference is that TD3 aims to fix issues with DQN with the actor-critic architecture. Briefly, actor-critic models are as such: the policy structure is referred to as the *actor* as it is used to select an action to take in a given state. The value estimation function is known as the *critic* as it evaluates the state and action pairs and gives it a value based on how valuable that action was. Value-based reinforcement learning methods like Q-learning are known to dramatically overestimate Q-values, in other words, incorrectly estimating the value of a state (Fujimoto, S., Hoof, H.V, Merger, D.). This ultimately results in suboptimal policies as the overestimated states will be kept and propagated through the network even though they may have been suboptimal. The TD3 algorithm attempts to build on the Deep Double Q-Learning (Double DQN) algorithm to address the overestimation issue by the DDPG. On a high level view, TD3 consists of 6 distinct networks: an actor, two critics, a target actor, and two target critics. TD3 implements three modifications to Double DQN: Target Policy Smoothing, Clipped Double-Q Learning, and Delayed Policy Updates.

In Clipped Double-Q Learning, TD3, learns two Q-functions Q_{θ_1} and Q_{θ_2}, takes the minimum value of the two, and regresses them through mean square Bellman error minimization. This allows the use of two critic networks to estimate the current Q-value by using the minimum of the two critic networks, essentially dealing with overestimation by underestimation. Underestimation will result in the lower-valued states to be left behind through the algorithm as opposed to overestimation.

The core idea with delayed policy updates is as follows: the policy network should be updated at a lower frequency than the value network. The use of target networks help reduce errors over multiple network updates, and the policy

network has divergent behavior when exposed to high-error states, so the policy network should be updated every few updates by the target network to reduce error in the Q-values (Fujimoto, S., Hoof, H.V, Merger, D.).

Finally, for target policy smoothing, TD3 adds a small amount of random noise sampled from a normal distribution $N(0, 0.2)$ to the target policy. The noise is then clipped to (-0.5, 0.5) to keep the modified action close to the original action.

Algorithm 3 TD3

Initialize critic networks Q_{θ_1}, Q_{θ_2}, and actor π_φ with random parameters $\theta_1, \theta_2, \varphi$
Initialize target networks $\theta'_1 \leftarrow \theta_1$, $\theta'_2 \leftarrow \theta_2$, $\varphi' \leftarrow \varphi$
Initialize replay buffer ß
for $t = 1$ to T **do**
  Select an action with added noise: $a \sim \pi_\varphi(s) + \epsilon$,
  $\epsilon \sim N(0, \sigma)$ and observe reward r and new state s'
  Store (s, a, r, s') in ß
  Sample mini-batch of N transitions (s, a, r, s') from ß
  $\tilde{a} \leftarrow \pi_\varphi(s') + \epsilon$, $\epsilon \sim \text{clip}(N(0, \hat{\sigma}), -c, c)$
  $y \leftarrow r + \gamma \min Q_{\theta_i}(s', \pi_{\phi_1}(s'))$, for i = (1, 2)
  Update critics $\theta_i \leftarrow \text{argmin } N^{-1} \Sigma (y - Q_{\theta_i}(s, a))^2$
  if t mod *policy_delay* **then**
    Update φ by the deterministic policy gradient:
    $\nabla_\phi J(\varphi) = N^{-1} \sum \nabla_a Q_{\theta_1}(s, a) \mid_{a = \pi_\varphi(s)} \nabla_\varphi \pi_\varphi(s)$
    Update the target networks:
    $\theta'_i \leftarrow \tau \theta_i + (1 - \tau)\theta'_i$
    $\varphi' \leftarrow \tau \varphi + (1 - \tau)\varphi'$
  end if
end for

3. Evaluation

To evaluate the performances of each algorithm, we extracted the "score" of the agent in each episode. For this task, the score was defined as the cumulative rewards gathered for a particular episode. Each episode ends given one of three conditions: the agent falls and touches the ground, the agent has hit a terminal state, or the agent reaches the end of the training environment. Each reward is calculated as a combination of costs based on the agent's movements. If the agent were to fall, then it would get a score of -100. As a result, the further the agent gets without falling, the better their score will be. We compared the scores of the Q-learning, DQN, and TD3 algorithms to determine how each performed. This was measured by graphing a particular run for an algorithm and observing the score per episode. We then analyzed the results of each run and hypothesized explanations for each.

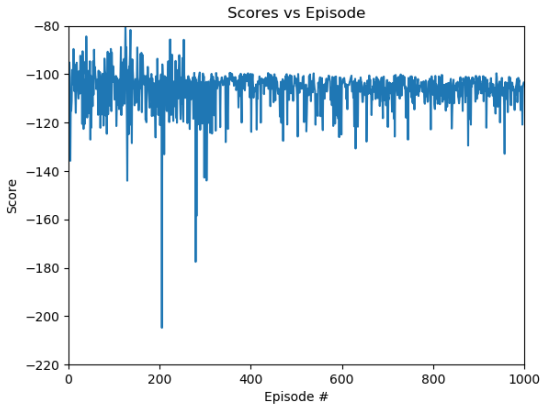

Figure 1: Q-learning performance

As seen in figure 1, Q-learning is not an effective algorithm for training BipedalWalker-v3. Initially, the agent made some progress, achieving a maximum score of -79.6. However, as rounds progressed, the agent consistently garnered poor scores. This is caused by the decaying epsilon. Early in the episodes, the value of epsilon is higher, causing random actions to be chosen more often than actions that would maximize the Qscore. Epsilon decays until it is close to 0 by the end of the episodes. Prioritizing actions that would maximize the Qscore leads to poor performance in this instance because of the discretization of the action space that is required when running the Q-learning algorithm on this environment. The action space for BipedalWalker is continuous, whereas Q-learning works best on action spaces that are discrete and low-dimensional. The algorithm is better-suited for smaller, simpler learning problems.

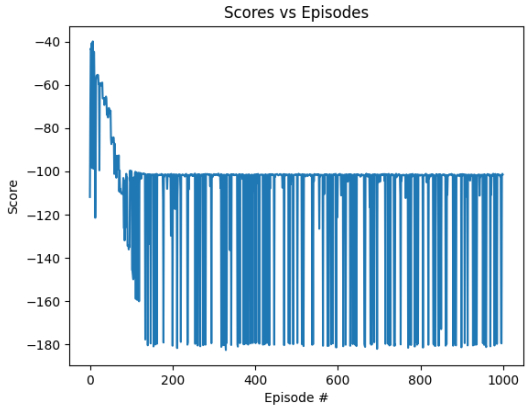

Figure 2: DQN performance

As demonstrated in Figure 2, the DQN agent had a great start and was able to achieve a maximum score of -40. However, as the agent went through more episodes, it no longer made any progress and could only achieve scores in

the range of -100 through -180 after a short while. The agent also took a total of 1 hour 26 minutes and 17 seconds to complete. We ran the model multiple times to observe its variance and found that the model consistently exhibited similar behaviors. We believe the reason why DQN failed is because the action space of BipedalWalker is continuous. Lillicrap, T. P. et al. (2015) writes in their paper that although DQN works well in high-dimensional observation spaces, it only effectively works with action spaces that are discrete and low-dimensional which explains the high performance of the DQN agent in the Atari 2600 environments. BipedalWalker contains 4 action dimensions (Hip1, Knee1, Hip2, Knee2), but each dimension has a range of values from -1 to 1. Due to this, DQN computes the gradient at every step to evaluate the maximum of the Q-function on a continuous space which is impractical. This also explains the observation of how the model did better at the start. In the beginning of the run, the agent took random actions due to the ε-greedy policy which were often times better than the actions that were selected to maximize the Q-values since the Q-values were computed based on the action domain.

One such way DQN can be improved is to reduce the continuous action space by discretizing it. However, that idea has significant flaws since agents that require accurate control will suffer to complete their task. By reducing the action domain, the agent may not be able to select actions that are crucial for its task and will ultimately fail. The approach we selected to improve DQN was implementing TD3 since the actor-critic architecture has been shown to be effective in tackling problems with high or continuous action spaces.

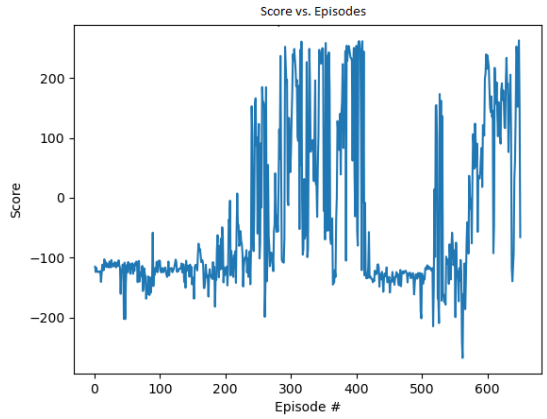

Figure 3: TD3 performance

To our implementation, we have a simple two-layer neural network of 400 and 300 hidden nodes and call ReLU in between each layer followed by a tanh. Our hyperparameters for the algorithm are exactly as described in Fujimoto, S. et al (2018): batch size = 100, γ=0.99, learning rate =10^{-3}, exploration policy=$N(0, 0.2)$, τ=5^{-3}, with Adam optimization. Our findings are reported with estimated times, as the estimated time also relies on hardware. To begin, the agent very quickly learns to stand and crawl forward by dragging its legs at 200 episodes as illustrated in Figure 3. At 300 episodes (~2 hours), the agent is steadily "skipping" rather than walking and rotating between it's right and left legs, however, the rewards peak to values close to 250. The trend of skipping gradually improves until episode 400, at which to multiple tests, we find that the agent drastically begins to worsen and lose it's capability in walking entirely. After further research, we believe this phenomenon to be catastrophic forgetting: an agent prioritizes newly learned information over past experiences and forgets what it had already learnt in the past (Imre, B. 2015). In other words, the experience replay begins to lose old experiences as new information is learned. As the episodes continue to 600 (~ 6 hours) the agent exhibits the ability to walk again with a score trend similar to episodes 200-300.

Catastrophic forgetting, also known as catastrophic interference, is a known issue when it comes to reinforcement learning and generally can be patched with using more memory and a smaller learning rate, using up to 3GB of memory to train another bipedal walker from OpenAI's Gym (Imre, B. 2015). For improvement to our TD3 implementation, the easiest route would be to allocate much more memory and tune the hyperparameters, for example minimizing the amount of noise to apply to our actions. This boils down to a tradeoff issue as well as real-world application issues: the current state of reinforcement learning algorithms like that of DQN and TD3 are limited in terms of storage for agents to recall their previous experiences. To our experiments, we find that even when allocating more space for our experience replay buffer (using a deque of maxlen=$2x10^6$), the graph of the scores remain nearly identical to our first findings with catastrophic forgetting. We notice that our implementation strays from what Fujimoto, S. et al (2018) writes: in their implementation they make use of a replay buffer containing the *entire* history of the agent, whereas our baseline model only maintains the past 1,000,000 transitions. Interestingly, this modification most likely leads to our issue with interference, which ultimately sheds light on steps for improvement not only in our implementation, but the field of reinforcement learning as a whole.

4. Conclusion

In this project, we have learned how to implement the Q-learning, DQN, and TD3 algorithms to facilitate reinforcement learning. Each algorithm takes inspiration from the previous algorithm, but also attempts to tackle the fundamental problems that it faced. Q-learning served as the foundation for our algorithms by utilizing the Bellman

equation as a function that selected the action with the most optimal action-value. DQN polishes the Q-learning implementation by integrating deep neural networks, experience replay to learn from its previous actions, and stochastic gradient descent on the loss function which is evaluated as the mean-squared error of the Bellman equation. However, as shown in the evaluation portion, the DQN agent was still not able to produce satisfactory results due to the environment's continuous action domain. We then refined our previous techniques by employing the use of the TD3 algorithm. The TD3 algorithm corrects the issues of overestimation by its predecessor algorithms through three modifications: target policy smoothing, clipped Double-Q learning, and delayed policy updates. The combined modifications result in performance that outshine the previous algorithms by great magnitudes, however we find that TD3 encounters memory issues with the experience replay prioritizing new experiences thus is prone to catastrophic interference.

As shown in our results, TD3 has much potential as a reinforcement learning method and would be a good future work area to develop further. It is held back by catastrophic interference, but if the issue of memory can be improved in reinforcement learning algorithms, then TD3 would do extremely well for this particular task and it may even be effective enough to be applied to other problems.

References

Fujimoto, S.; Hoof, H.V.; Meger, D. 2018. Addressing Function Approximation Error in Actor-Critic Methods. *ArXiv, abs/1802.09477*.

Lillicrap, T. P.; Hunt, J. J.; Pritzel, A.; Heess, N.; et al. 2015. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; et al. 2015. Human-level control through deep reinforcement learning. *Nature*. 518, 529–533. https://doi.org/10.1038/nature14236

Imre, B. (2021). *An investigation of generative replay in deep reinforcement learning* (Bachelor's thesis, University of Twente).

Sutton, Richard S. and Barto, Andrew G., Reinforcement Learning: An Introduction, MIT Press, 199