Reinforcement Learnning Project

Ruben Wolhandler , Moshe Berchansky

March 2022

1 Introduction

Recently, deep-reinforcement learning has had the capability to solve difficult tasks, many of which require the approahces to perform novel reactions to previously unseen situations. One primary focus of these works has been the ability to train an autonomous agent to navigate across unfamiliar and challenging terrains. Such scenarios are included in OpenAI's environments **LunarLander** and **BipedalWalker**. In this work, we attempt to solve these environments using state-of-the-art techniques, based on the latest works in the field of deep-reinforcement learning. Our full code and approaches are in https://colab.research.google.com/drive/15iIWoM3xRDCC4ud80sdIj3KHbceEd6XQ?usp=sharing.

2 Related Work

2.1 DQN-DDQN

Generally in Reinforcement learning has five principal components: the agent , the agent's action space, the environment and the state result from the agent's action, and the rewards we get after each action. On the same way of classic Q-Learning ,Deep Q network [Mni+13] aims to found the best agent's action for each state using function approximation instead of Q-table.It's advantage over Q-learning, is that we don't need to explore all the pairs state-actions, but DQN do some generalisation thanks to its function approximation (neural networks). In the case of the Q-table is too large, We can't explore all the pairs state-action and we won't be able to predict what is the best action for a lot of state contrary to DQN. Like Q-Learning, DQN aims to maximise expected cumulative rewards, then DQN aims to maximise this equation , a variation of bellman equation:

$$Q(s,a) = r(s,a) + \gamma max_{a'} Q(s',a') \tag{1}$$

when s is the current state and a is the current agent's action, in order to maximise this equation. We minimize the DQN loss defined as:

$$Loss = \frac{1}{2}([r(s,a) + \gamma max_{a'} Q_\theta(s',a')] - Q_\theta(s,a))^2 \tag{2}$$

where θ is the parameters of the Q network, $r(s,a) + \gamma max_{a'} Q_\theta(s',a')$ is the target, and $Q_\theta(s,a)$ is the prediction. To address the problem of overestimation that we can occur in DQN [HGS16] show that adding a target network can be benefit. the target will be calculated by a new target network instead of the Q-network. But selecting the next action a' according to the Q-network. The new loss function is defined by:

$$L(\theta) = \frac{1}{2}([r(s,a) + \gamma Q_{\theta'}(s', a^{max}(s',\theta)] - Q_\theta(s,a))^2 \tag{3}$$

where θ is the parameters of the Q-Network and θ' is the parameters of the target Network.

2.2 Actor critic Models

In [FHM18], they introduce *TD3*, a Double-Critic (Q-learning) [HGS16] system, as an update to the standard Actor-Critic Architecture. The system takes the minimum value between a pair of critics. This method bounds the possibility of our models overestimating. In the work done by [Haa+18], they propose soft actor-critic (SAC), a variation of actor-critic. Here, the actor aims to maximize expected reward while also maximizing entropy. Building on top of the previous approaches, [WY21] uses an Actor-Critic scenario, with the addition of a system and reward model. These models are tasked with predicting the future state given the current state and action, and its corresponding reward. In our solution to the BipedalWalkerHardcore system, we primarily extend this work to realize our goal.

3 Background

In the reinforcement learning setting, an Actor model A_ϕ produces an action a_t, given the state s_t, at the current time step t. For optimization, the Actor-Critic method includes an additional Critic model Q_ψ, which receives the state s_t and the predicted action by the actor $A_\phi(s_t)$.

3.1 Discrete and Continuous Approaches

In our implementation of the solution to the environment, we utilize both a discrete and continuous approach. Namely, the continuous version uses an action space $a \in R^n$, where every dimension i of the action is bound by $[a_i, b_i]$. Thus, the Actor model has an output linear layer of shape of the same dimension. In the discrete case, we divide each dimension i in the action space into k_i ranges:

$$a_{dim=i} = \{r_{i,j}\}_{j=1}^{k_i}; r_{i,j} \in [a_i, b_i] \tag{4}$$

For example, if the first dimension of the action is in range $[-1, 1]$, a possible discretization can be $a_{dim=1} = -1, -0.5, 0, 0.5, 1, k_1 = 5$. A part of the solution to the discrete case is to set the k_i values for each action dimension.

4 FORK: Forwaring-Looking Actor

We primarily use the work of [WY21] as an inspiration for our approach. This work extends the standard Actor-Critic approach, by adding two new models: The *system* model and the *reward* model. Given the current state and corresponding action, the **system model** predicts the next state. In addition, the **reward model** predicts the reward.

4.1 System Network

This model F_θ predicts the next state, given a state s and an action a:

$$\hat{s}_{t+1} = F_\theta(s_t, a_t) \tag{5}$$

Since, we have the next state, we can use the smooth $L1$:

$$L(\theta) = ||s_{t+1} - F_\theta(s_t, a_t)|| \tag{6}$$

4.2 Reward Network

This model R_η predicts the reward, given a state s and an action a:

$$\hat{r}_t = F_\theta(s_t, a_t) \tag{7}$$

This is a supervised scenario, in which we can use r_t:

$$L(\eta) = ||r_t - R_\eta(s_t, a_t)||^2 \tag{8}$$

We note, that we use a variant of the reward model, which also takes the next state as input, to better predict the reward:

$$\hat{r}_t = F_\theta(s_t, a_t, s_{t+1}) \tag{9}$$

In the rest of our experiments, we will use the next state variant of the reward model.

4.2.1 Convolution-Based Reward Model

Since the reward model receives the state $s_t \in R^d$ next state $s_{t+1} \in R^d$ for reward estimation, we propose and experiment with a 1D-Convolution based model, that considers the states as two frames. The model first performs L 1d-convolution operations on the state frames, and produces a resulting state $s_t^{CONV} \in R^d$. Formally, we produce reward r_t in the following method:

$$o_0 = s_{t,t+1} = [s_t, s_{t+1}] \in R^{2\times d} \tag{10}$$

$$\forall l \in [1, L] : o^l = ReLU(Conv1d(o^{l-1}, kernel = 2, stride = 1, pad = 1)) \in R^2 \tag{11}$$

$$o_t = \frac{1}{2}\sum_{i=1}^{2} o_i^L \in R^d \tag{12}$$

$$r_t = MLP([o_t; a_t]) \in R \tag{13}$$

where [,] if the **stack** operation, [;] is the concatenation operation, and MLP is a standard 1 hidden layer Multi-Layer Perceptron. We compare this variant to the standard MLP implementation of the reward model in the Experimental Results section.

4.3 Overall Loss

Using the system and reward models, we can utilize a loss that optimizes their predictive outputs. We use the system model to predict two states, and the reward model predict the rewards to each future state. Overall, the loss is defined as:

$$\begin{aligned} L(\phi) = E[-Q_\psi(s_t, \hat{a}_t) - R_\eta(s_t, \hat{a}_t, s_{t+1}) - \gamma R_\eta(\hat{s}_{t+1}, A_\phi(\hat{s}_{t+1}), \hat{s}_{t+2}) \\ - \gamma Q_\psi(\hat{s}_{t+1}, A_\phi(\hat{s}_{t+1})) - \beta\gamma^2 Q_\psi(\hat{s}_{t+2}, A_\phi(\hat{s}_{t+2})] \end{aligned} \tag{14}$$

where $\hat{a}_t = A_\phi(s_t)$, γ is the discount factor,and $\hat{s}_{t+2} = F_\theta(\hat{s}_{t+1}, A_\phi(\hat{s}_{t+1}))$. In addition, the loss of the critic over the 2-step ahead prediction, $Q_\psi(\hat{s}_{t+2}, A_\phi(\hat{s}_{t+2}))$, is multiplied by a factor β, which is a hyperparameter we explore as well.

4.4 Adaptive Weight Loss

As proposed in [WY21], we switch from using a standard γ weight for the loss factors, to a dynamic weight:

$$w = \begin{cases} r, & \text{if } r \in [0,1] \\ 0, & \text{otherwise} \end{cases}, r = w_0 \frac{\hat{r}}{r_0}$$

where $\hat{r}$ is the moving average of cumulative reward per episode, r_0 is the predefined goal average reward, and w_0 is an initial weight given. This changes our loss to:

$$\begin{aligned} L(\phi) = E[-Q_\psi(s_t, \hat{a}_t) - R_\eta(s_t, \hat{a}_t, s_{t+1}) - w\gamma R_\eta(\hat{s}_{t+1}, A_\phi(\hat{s}_{t+1}), \hat{s}_{t+2}) \\ - w\gamma Q_\psi(\hat{s}_{t+1}, A_\phi(\hat{s}_{t+1})) - w\beta\gamma^2 Q_\psi(\hat{s}_{t+2}, A_\phi(\hat{s}_{t+2})] \end{aligned} \tag{15}$$

4.5 Removal of the Reward Model

In the implementation of the system and reward models, we consider the possibility of removing the reward model altogether. Thus, the overall loss is instead:

$$L(\phi) = E[-Q_\psi(s_t, \hat{a}_t) - w\gamma Q_\psi(\hat{s}_{t+1}, A_\phi(\hat{s}_{t+1}) - w\beta\gamma^2 Q_\psi(\hat{s}_{t+2}, A_\phi(\hat{s}_{t+2})] \tag{16}$$

4.6 Reward Scaling During Training

Inspired by[Cre], we use additional approaches to help solve the Hardcore environment of BipedalWalker.

1. **Falling reward clipping**: The environment setup returns a reward of with the value $r = -100$ once the agent tumbles. However, the this signal can be very blunt for our network to learn. Hence, its overall value is not relevant. We experiment with either replacing it by 0 or -5. The results will be further explained in the Experimental Results section.
2. **Reward Increase**: rewards can be amplified to better aid the model to learn proper behavior. Hence we experiment with the multiplication value of the non-done rewards.

In particular, we modify the following parameters:

1. **Fall Reward Replacement**: Once the agent tumbles, we replace the cummlative reward by some small amount, usually 0.
2. **Fall Reward Addition**: Once the agent tumbles, we replace the added reward by some small amount, usually -1.
3. **Non-Fall Reward Multiplication Factor**: When agent succeeds, we scale up the reward, to increase the signal strength.

5 Experimental Setup

5.1 DQN-DDQN

Generally, our DQN and DDQN implementation and additions are based on a repository available here [1]. We specify the parameters used in the run, and their possible values:

1. γ: The loss scaling factor.
2. ϵ: The ϵ hyperparamter for the epsilon-greedy action selection. It allows for better exploration of the action space.
3. Hidden Dimension: The dimension of the hidden layer in the MLP.
4. Last Dimension: The dimension of the last layer in the MLP.
5. Dimension: The dimension of the first layer in the MLP.
6. Target Update: In DDQN, this is the frequency by which the ttarget model is being updated during training.

[1] https://goodboychan.github.io/python/reinforcement˙learning/pytorch/udacity/2021/05/07/DQN-LunarLander.html

7. Buffer Size: Size of the replay buffer during training.

8. Seed: Random seed used.

9. Discretization Scheme: The discritezation ranges used, as described in 3.1.

5.2 TD3 FORK

Generally, our [WY21] implementation and additions are based on a repository available here [2].

5.3 Discrete or Continuous Action Spaces

When considering the regular environment, the LunarLanderContinuous-v2 environment has an action space of $[-1, 1]^2$. The divisions attempted are as follows:

1. Divide the first dimension into $k_1 = 3$ slices: $[0, 0.5, 1.0]$, and second dimension into $k_1 = 5$ slices: $[-1.0. - 0.75, 0, 0.75, 1.0]$. Thus, our total discrete action space is 15.

2. Divide the first dimension into $k_1 = 5$ slices: $[0, 0.25, 0.5, 0.75, 1.0]$, and second dimension into $k_1 = 5$ slices: $[-1.0. - 0.5, 0, 0.5, 1.0]$. Thus, our total discrete action space is 25.

5.4 Velocity Removal

In order to evaluate the importance of the velocity feature, we drop it in favor of feeding into the model the previous step, alongside the current one. Thus, allowing the model to learn the necessary features from the input. To replace the velocity features, we use the elementary rule of physics, stating that the velocity is the delta in postion, divided by the delta in time:

$$v_t = \frac{p_t - p_{t-1}}{t - (t-1)} = p_t - p_{t-1} \tag{17}$$

where v_t is the velocity at time step t, and p_t is the position in the same step. Thus, we construct the velocity in the x and y axis using the position of the current and previous step:

$$v_x(s_t) = p_x(s_t) - p_x(s_{t-1}) \tag{18}$$

$$v_y(s_t) = p_y(s_t) - p_y(s_{t-1}) \tag{19}$$

where $v_x(s_t), v_y(s_t)$ are the velocities in axis x and y respectively at time step t. In addition, $p_x(s_t), p_y(s_t)$ are the positions in axis x and y respectively at time step t.

[2] https://github.com/Rafael1s/Deep-Reinforcement-Learning-Algorithms

6 Experimental Results

6.1 BipedialWalkerHardcore Experiments

6.2 Hyper Parameter Sweep Results

For our various models, we use MLPs, with various parameters possible for each one. In addition, we wish to optimize the training process, by altering the learning and batch size. We specify the parameter list in 1.

Parameter	Values
Dropout	0, 0.1
Activation	Relu
Layer Count	2, 3
Dimension:	300
Learning Rate	1e-4, 2e-4, 4e-4
Batch Size	100, 200

Table 1: MLP and training hyperparameters for the models.

We present the initial hyperparameter search in Figure 1.

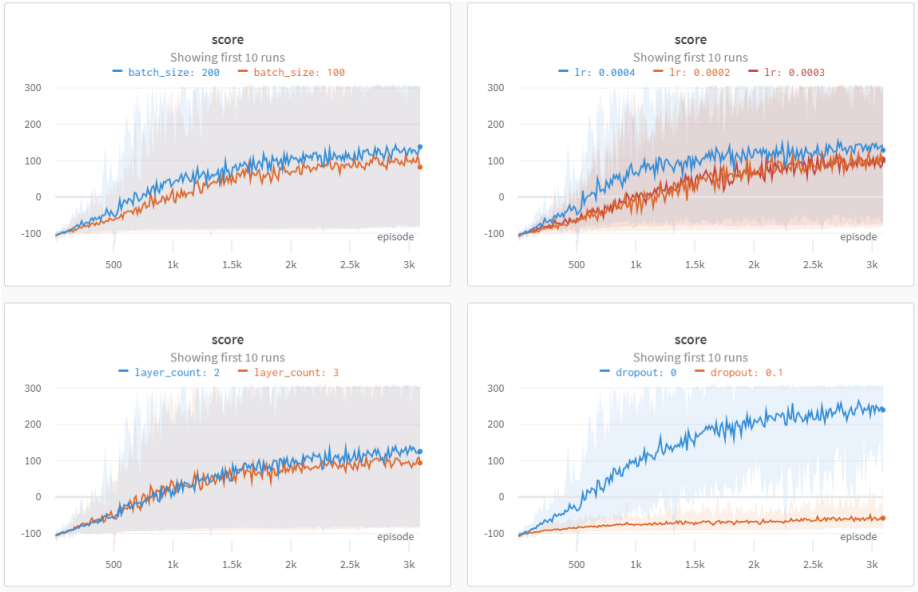

Figure 1: Average score, when varying across multiple training parameters.

From them, we deduce a few key observations:

1. **Dropout hinders performance**: As seen in Figure 1, adding even a

small amount of dropout causes the model training to diverge to a low average score.

2. **Higher batch size and learning rate**: Increasing the learning rate from 3e-4 to 4e-4, and increasing the batch size from 100 to 200, improves performance.

3. **Smaller layer count**: Using 2 layers instead of 3 in the various models, results in slightly better performance overall.

6.3 Critic Next State Loss weight

We also experiment with value the of the β hyperparameter, as described in equation 16.

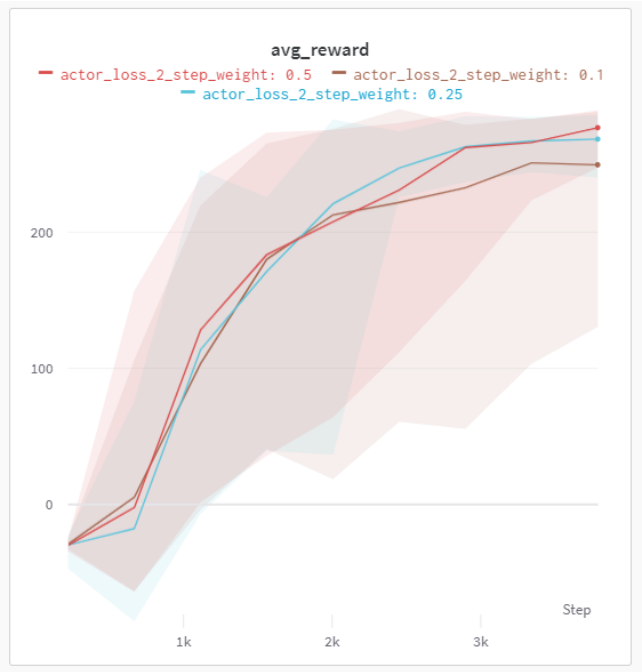

Figure 2: Average score over 100 episodes, when varying the weight of the critic next state loss weight.

As seen in Figure 2, the weight of $\beta = 0.5$, which we will use in the rest of our experiments.

6.4 Reward Scaling During Training Results

After running a sweep over multiple seeds, and the space specified in 2.

Parameter	Values
Use Reward Model?	False
Fall Reward Addition	-1, -10
Fall Reward Replacement	-5, 0
Non-Fall Reward Multiplication Factor	5

Table 2: Reward Scaling hyperparameter space.

The results are specified in Figure 3.

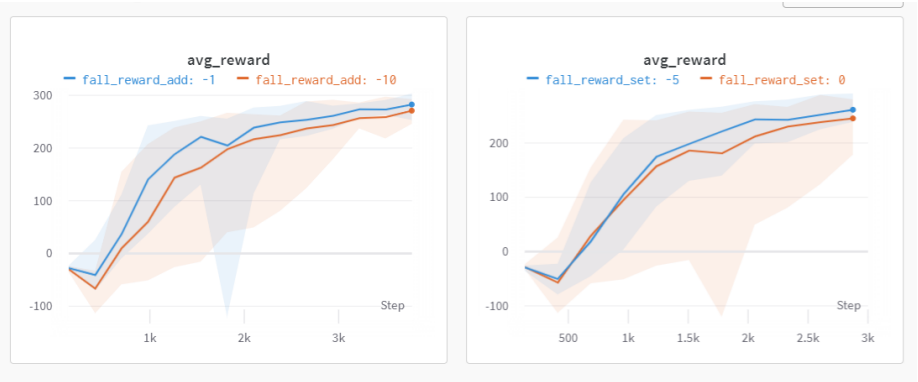

Figure 3: Average score over 100 episodes, when varying the reward scaling parameters.

As can be observed, setting -5 for Fall Reward Replacement and -1 for Fall Reward Addition, you can obtain the best results, though by a relatively insignificant margin.

6.5 Utilizing a System-Model only setting

]In this section, we wish to evaluate the necessity of the reward model in performance. Thus, we take the best configuration the previous search, and run the setting once with the reward model and once without it.

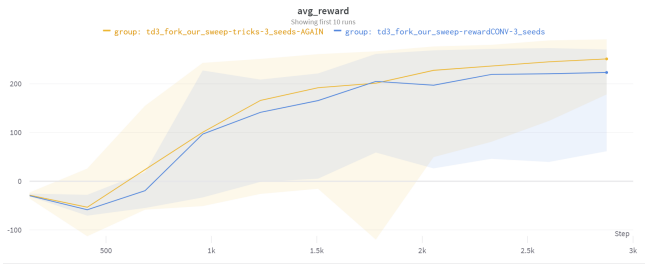

Figure 4: Average score, when varying the usage of the reward model.

As seen in Figure 4, the addition of the reward model leads to lesser performance. As a result, we *remove its usage* from our final solution. We note that these results with the reward model use both a standard MLP and a 1d-Convolution, in different runs. As can be seen, both do not outperform a system model only approach.

6.6 Environment Solving Solution

After conducting the search section 6.4, we used the best parameters to run the training, with the 100 eval episodes in Figure 3. Thus, we received a run that achieved 300.356 after 2793 episodes. The full hyperparameter setting of this run are specified in table 3.

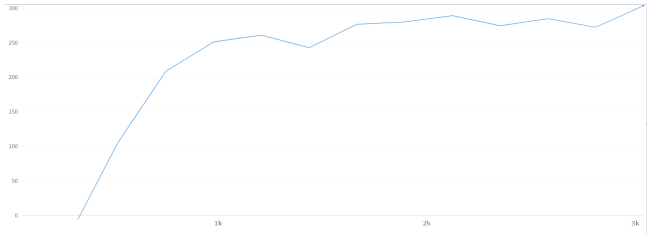

Figure 5: Average score over 100 eval episodes on the hardcore environment, for the best run.

The parameters are then:

Parameter	Values
Dropout	0
Activation	Relu
Layer Count	2
Dimension:	300
Learning Rate	4e-4
Batch Size	200
Use Reward Model?	False
Fall Reward Addition	-1
Fall Reward Replacement	-5
Non-Fall Reward Multiplication Factor	5
Seed	6

Table 3: Best model that solves the BipedalWalkerHardcore environement.

6.7 LunarLanderContinuous, Using Velocity Features

6.7.1 Using DQN

Using the default parameters specified in Table 4, we run the DQN algorithm.

Parameter	Values
γ	0.99
ϵ	0.9
Hidden Dimension	64
Last Dimension	64
Dimension:	300
Learning Rate	5e-4
Batch Size	64
Seed	1
Discretization Scheme	[0, 0.5, 1.0], [-1.0. -0.75, 0, 0.75, 1.0]

Table 4: DQN that solves the LunarLanderContinuous with velocity environment.

With the above, we receive the following results with them:

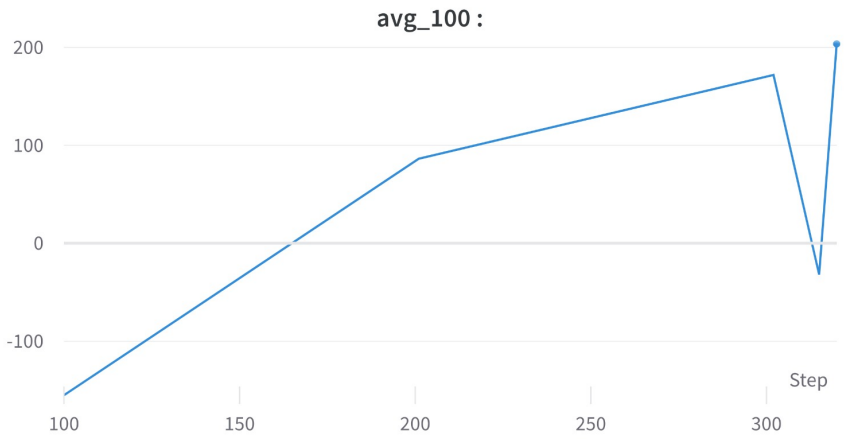

Figure 6: Average score over 100 episodes for the DQN algorithm.

As can be seen in Figure 6, DQN reaches above 200 after 316 episodes.

6.7.2 Using DDQN

Using the default parameters specified in Table 5, we run the DDQN algorithm.

Parameter	Values
γ	0.99
ϵ	0.9
Hidden Dimension	128
Last Dimension	128
Dimension:	300
Learning Rate	5e-4
Batch Size	64
Target Update	50
Buffer Size: 100000	
Seed	1
Discretization Scheme	[0, 0.5, 1.0], [-1.0. -0.75, 0, 0.75, 1.0]

Table 5: DQN that solves the LunarLanderContinuous with velocity environment.

We receive the following results with them:

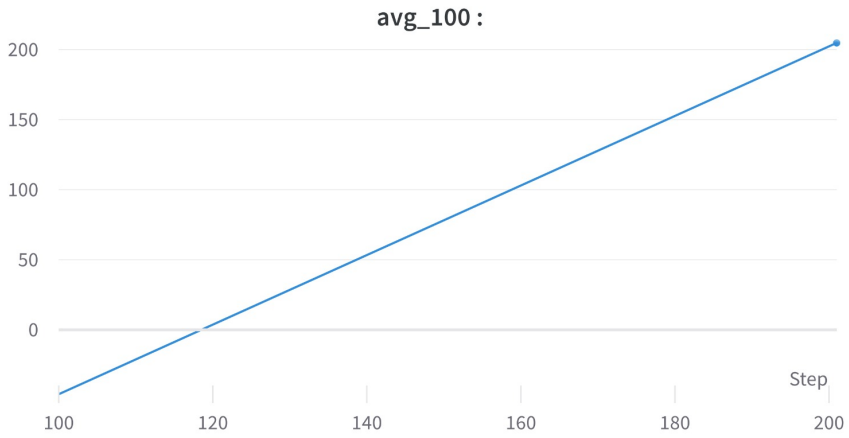

Figure 7: Average score over 100 episodes for the DDQN algorithm.

As can be seen in Figure 7, DDQN reaches above 200 after 200 episodes.

6.7.3 Using TD3-FORK

Using the default parameters of [WY21] found in section 6.4, we also utilize them on the LunarLander case, after discretization, as specified in section 5.3. After trying the discretizations schemes specified, we have achieved a run that solves the environment.

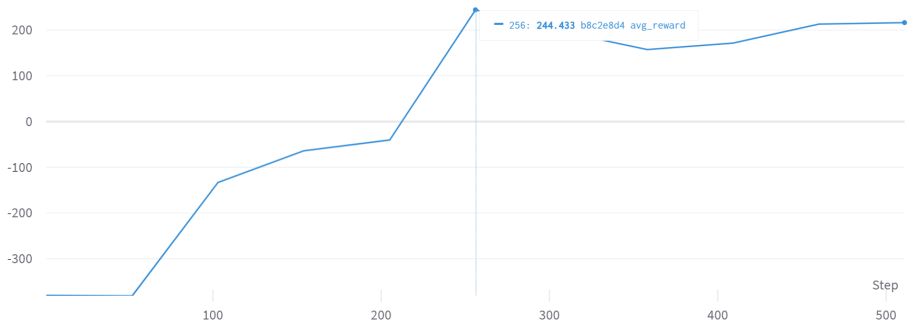

Figure 8: Average score over 100 episodes, when varying across multiple seeds of the FORK algorithm.

As seen in Figure 8, our approach reaches above 200 significantly, 244.345 after 250 episodes. The full parameter list can be found in Table 6.

Parameter	Values
Dropout	0
Activation	Relu
Layer Count	2
Dimension:	300
Learning Rate	4e-4
Batch Size	200
Use Reward Model?	False
Fall Reward Addition	-1
Fall Reward Replacement	-5
Non-Fall Reward Multiplication Factor	5
Seed	42
Discretization Scheme	[0, 0.5, 1.0], [-1.0. -0.75, 0, 0.75, 1.0]

Table 6: Best model that solves the LunarLanderContinous with velocity environement.

6.7.4 Environment Solving Solution

After trying the DQN, DDQN, and TD3-FORK solutions, our best performing model was the DDQN model, with 200 episodes to solve the environment.

For the competitive part, here are the results for 100 consecutive evaluation episodes:

1. For 100 episodes: -45.89
2. For 200 episodes: 204.76

6.8 LunarLanderContinuous, *Without* Velocity Features

6.8.1 Using DQN

Using the default parameters specified in Table 7, we run the DQN algorithm.

Parameter	Values
γ	0.99
ϵ	0.9
Hidden Dimension	64
Last Dimension	64
Dimension:	300
Learning Rate	5e-4
Batch Size	64
Seed	1
Discretization Scheme	[0, 0.5, 1.0], [-1.0. -0.75, 0, 0.75, 1.0]

Table 7: DQN that solves the LunarLanderContinuous with no velocity environment.

With the above, we receive the following results with them:

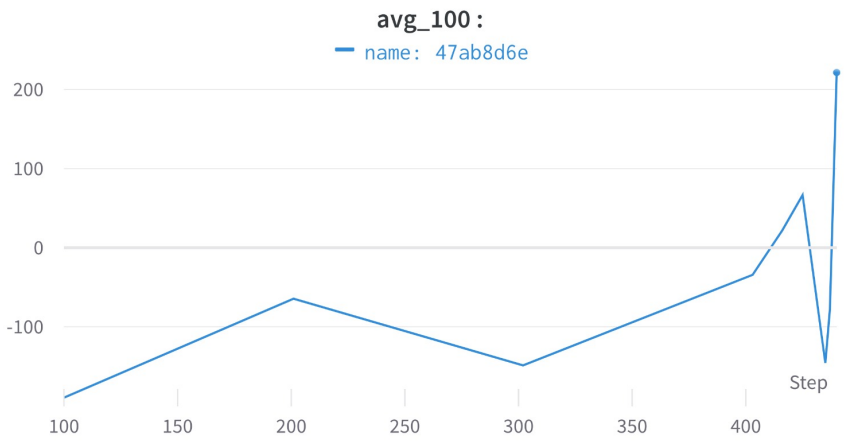

Figure 9: Average score over 100 episodes, for the DQN algorithm.

As can be seen in Figure 9, DQN reaches above 200 after 432 episodes.

6.8.2 Using DDQN

Using the default parameters specified in Table 8, we run the DDQN algorithm.

Parameter	Values
γ	0.99
ϵ	0.9
Hidden Dimension	256
Last Dimension	256
Dimension:	300
Learning Rate	5e-4
Batch Size	64
Target Update	50
Buffer Size: 100000	
Seed	1
Discretization Scheme	[0, 0.5, 1.0], [-1.0. -0.75, 0, 0.75, 1.0]

Table 8: DDQN that solves the LunarLanderContinuous with no velocity environment.

We receive the following results with them:

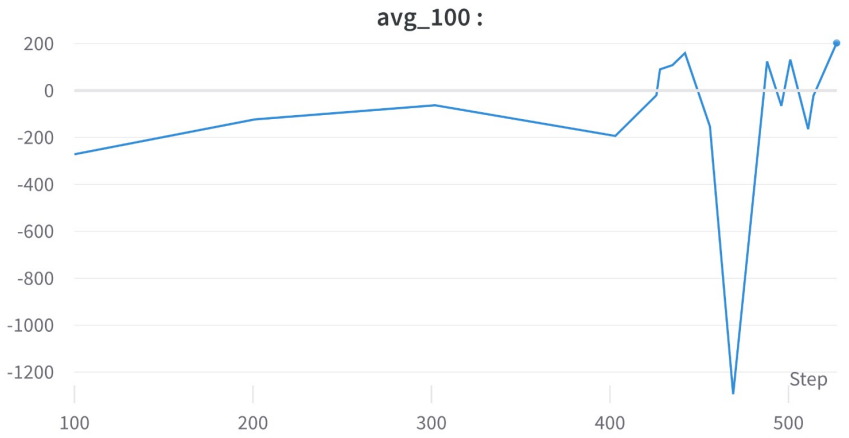

Figure 10: Average score over 100 episodes for the DDQN algorithm.

As can be seen in Figure 10, DDQN reaches above 200 after 512 episodes.

6.8.3 Using TD3-FORK

As stated in section 5.4, we remove the velocity features. As in the with-velocity case, we are using the default parameters of [WY21] found in section 6.4, we also utilize them on the LunarLander case, after discretization, as specified in section 5.3.

Figure 11: Average score over 100 episodes, when varying across multiple seeds of the FORK algorithm.

As seen in Figure 11, our approach reaches above 200 significantly, 231.203 after 800 episodes. We note that solving the environment in this case takes longer than the regular with-velocity case. The full parameter list can be found in Table 9.

Parameter	Values
Dropout	0
Activation	Relu
Layer Count	2
Dimension:	300
Learning Rate	4e-4
Batch Size	200
Use Reward Model?	False
Fall Reward Addition	-1
Fall Reward Replacement	-5
Non-Fall Reward Multiplication Factor	5
Seed	42
Discretization Scheme	$[0, 0.5, 1.0], [-1.0. -0.75, 0, 0.75, 1.0]$

Table 9: Best model that solves the LunarLanderContinous with *no* velocity environement.

6.8.4 Environment Solving Solution

After trying the DQN, DDQN, and TD3-FORK solutions, our best performing model was the DQN model, with 432 episodes to solve the environment.

For the competitive part, here are the results for 100 consecutive evaluation episodes:

1. For 100 episodes: -189.52
2. For 200 episodes: -64.42

3. For 300 episodes: -148.69
4. For 400 episodes: -34.23
5. For 432 episodes: 221.61

7 Conclusion

After running and attempting multiple approaches, we are able to solve the discrete and the hardcore environments successfully. For the discrete case, the DQN algorithm was sufficient to solve it successfully, encouraging the notion that additional algorithmic complexity may be redundant. The algorithm designed by [WY21] performed best overall, allowing our modifications to solve the hardcore environment in an efficient matter. All the results are documented in the Experimental Results section above.

References

[Mni+13] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: *ArXiv* abs/1312.5602 (2013).

[HGS16] H. V. Hasselt, Arthur Guez, and David Silver. "Deep Reinforcement Learning with Double Q-Learning". In: *ArXiv* abs/1509.06461 (2016).

[FHM18] Scott Fujimoto, Herke van Hoof, and David Meger. "Addressing Function Approximation Error in Actor-Critic Methods". In: *ArXiv* abs/1802.09477 (2018).

[Haa+18] Tuomas Haarnoja et al. "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor". In: *ICML*. 2018.

[WY21] Honghao Wei and Lei Ying. "FORK: A FORward-looKing Actor for Model-Free Reinforcement Learning". In: *2021 60th IEEE Conference on Decision and Control (CDC)* (2021), pp. 1554–1559.

[Cre] CreateAMind. "https://mp.weixin.qq.com/s?¨biz=MzA5MDMwMTIyNQ==mid=2649294554idx=
In: ().