Case Study: Implementation of Differential Training of Rollout Policies

Roham Ghotbi [1] **Hossein Najafi** [1]

Abstract

One of the issues regarding the popular methods used in the calculation of the rollout policies used for the training and evaluation of reinforcement learning methodologies, is their sensitivity to the simulation and approximation errors(Bertsekas, 1997). Dr. Dimitri P. Bertsekas has proposed a method called, *differential training*, to mitigate the issue mentioned about the sensitivity to randomness, both in the environment and in the observation noise. In this work, we have implemented the suggested technique and evaluated its effectiveness against various noise levels, in multiple kinds of environments. Our implementation of the differential training is able to improve the performance of the RL methods we have used in comparison to their results when using absolute Q-values and rewards-to-go.

1. Introduction

The main argument behind the Bertsekas method, *differential training*, is the lack of robustness in standard Reinforcement Learning methods such as Temporal Difference and Q-Learning toward simulation and approximation error(Bertsekas, 1997). This is a major challenge when it comes to practical computation and he offers two remedies for such issue. One is focused on evaluation of Q-value differences by Monte-Carlo Simulation. The other method, which is the main focus of this work, is what he calls *differential training*.

As we later discuss the details of this method, one might get reminded of another technique called *Advantage Updating* by Baird, Harmon, and Klopf(Harmon et al., 1994), which is also an effort to battle the simulation and approximation error. While this method is an effective way to mitigate the aforementioned issues, some training methods, such as Temporal Difference, cannot be be used in a straightforward way to approximate the Advantage function. Unlike advantage training, the differential training method, has no such limitations thanks to its new state and observation space in which each state is a pair of observations/states (s, s') from the original system.

It is also worth mentioning that the main interest of the paper was on methods that would approximately calculate rollout policy $\bar{\pi}$ for complex large-scale problems. Moreover, they assume that they are able to simulate the system under the base policy π, where they can generate sample trajectories and corresponding rewards consistent with the probabilistic data of the model.

In this project, we are utilizing Q-values and Q-learning in the training process of our model(in the form of DQN) and as Hado van Hasselt points out in his *Double Q-Learning* paper, Q-learning tend to perform poorly in some stochastic environments(van Hasselt et al., 2015) (Hasselt, 2010). Therefore, we have performed all our experiments exploiting the advantage Double Q-Learning has, in order to assure overall better performance and prevention of any hiccups that Q-Learning might cause. The environments we have investigated are all from the *openAI Gym*. The gym is an open source toolkit that is used for development and comparison of Reinforcement Learning. In our studies, we have used Lunar Lander(Discrete and Continuous), Pendulum, Half Cheetah, and Cartpole(also known as inverted Pendulum).

Lastly, the paper was written using the state and control terminology, (x, u), and the whole algorithm has been explained using the cost functions, cost-to-go and Q-factors. In order to keep the familiar vocabulary used in the class and consistency with our own code, our explanation in section 3 uses reward functions, Q-values and denotes state action pairs as (s, a).

For the rest of this article, in section 2, we will briefly explain the background information related to the work done in this project and why such method has been introduced by Bertsekas, in section 3 the algorithm proposed by Bertsekas will be introduced and described, and in Section 4 the performance of the Bertsekas' proposed solution will be empirically investigated compared to traditional method without the differential training in use.

2. Background

This section will give a background and overview of related literature in this area. In section 2.1, the Dynamic Programming approach and how the Reinforcement Learning is setup(Both Discrete and Continuous) will be discussed.

Section 2.2 briefly covers what Deep Q-Networks are and how they are implemented.

2.1. The State-Action Setup

Let's start with defining the stationary setup for the discrete time dynamic system. The next state will be defined as:

$$s_{k+1} = f(s_k, a_k, w_k) \tag{1}$$

where s_k is the state taking values in some set, a_k is the action which is chosen from a finite action set $\mathcal{A}(s_k)$, and w_k is some random disturbance. We also have a one-stage reward function which is denoted by $r(s, a, w)$ - although more so than often, the reward does not depend on the randomness factor w. As a side note, just like the original paper, we have omitted the time indexes for the functions f, r and $\mathcal{A}$, but our discussion also applies to the time-varying context, where they depend on the time step k.

We now assume that we are given a policy $\pi = \{\mu_1, \mu_2, ...\}$ which maps the state s to the action $\mu_k(s)$ at the time step k ($\mu_k(s) \in \mathcal{A}(s)$). For the discrete case, we focus on an N-stage problem where we have discrete time steps ranging from 0, to $N-1$. The reward-to-go can then be described as:

$$J_k(s_k) = E\left\{ \sum_{i=k}^{N-1} \big[r(s_i, \mu_i(s_i), w_i)\big] + R(s_N) \right\} \tag{2}$$

Where $R(s_N)$ is the terminal reward which is dependant on the terminal state s_N we reach at last, and $J_k(s_k)$ is the reward-to-go of π where we start from state s_k and end at the N^{th} time step. In general, the reward-to-go function can be rewritten in the recursive form below:

$$J_k(s) = E\big\{r(s, \mu_k(s), w) + J_{k+1}(f(s_k, \mu_k(s), w_k))\big\} \tag{3}$$

which is equivalent to:

$$J_k(s) = E\big\{r(s, \mu_k(s), w) + J_{k+1}(s')\big\} \tag{4}$$

for:

$$k = 0, 1, ... \qquad \text{and} \qquad J_N(s) = R(s)$$

where s' is the next state we go to after taking the proper action at the state s

Now for the infinite time horizon case, we need to assume a stationary policy $\pi = \{\mu, \mu, ...\}$ where the dynamic programming algorithm mentioned before is replaced by its asymptotic version which is defined as:

$$J(s) = E\big\{r(s, \mu(s), w) + \alpha J(f(s, \mu(s), w))\big\} \ \ \forall \ \ s \tag{5}$$

which is equivalent to:

$$J(s) = E\big\{r(s, \mu(s), w) + \alpha J(s')\big\} \ \ \forall \ \ s \tag{6}$$

Where $\alpha \in (0, 1)$ is a discount factor affecting the future reward value. Thus we will have a rollout policy $\bar{\pi} = \{\bar{\mu_1}, \bar{\mu_2}, ...\}$ which has an optimal reward-to-go approximated by the reward-to-go of the base policy π, where the action is:

$$\bar{\mu}_k(s) = arg \max_{a \in \mathcal{A}(s)} E\big\{r(s, a, w) + \alpha J(f(s, a, w))\big\} \tag{7}$$

for: $\quad \forall \ \ s, \ \ k = 0, 1, ...$

Finally, as described in the paper (Bertsekas, 1997), using a standard policy iteration argument and induction, one can clearly show that the rollout $\bar{\pi}$ is indeed an improved policy over the initial base policy π. Now, the importance of the improved policy is even more apparent in practice, where the corresponding reward-to-go J_k may not be known in the close form. In that case, the computation of the rollout control $\bar{\mu_k}$ using the the Eq.(7) will become an important issue since for all the actions in the set $\mathcal{A}(x)$ we need the value of:

$$Q_k(s, a) = E\big\{r(s, a, w) + J_{k+1}(f(s, a, w))\big\} \tag{8}$$

which is know as the *Q-function*. $Q_k(s, a)$ is defined as the total reward from taking the action a at the state s.

The standard methods involved in the approximate computation of such policy $\bar{\pi}$ in highly complex and large problems requires approximation of the aforementioned reward-to-go functions. While these methods are effective ways to create an approximation of the target policies, Bertsekas argues that such standard RL training techniques are not robust to the approximation and simulation errors and the *Differential Training* will help alleviate the issues caused by such noises in the system.

2.2. Deep Q-Network(DQN)

As mentioned in the section 2.1, *Q-functions* play an important role in coming up with an approximate calculation of the improved policies $\bar{\pi}$ in Reinforcement Learning. One of the standard methods used in order to achieve such models, is called *Q-Learning*. *Q-Learning* is an off-policy RL algorithm that seeks to find the best action to take given the current state. In other words, *Q-Learning* tries to learn the best policy that maximizes the total reward. If we assume that $Q^*(s, a)$ is the actual expected value of the total reward if we were to take the action a at the state s and then follow the optimal policy, the *Q-Learning* algorithm uses Temporal Difference(TD) to come up with an estimate value of $Q^*(s, a)$ that would be as close as possible to the original value, and then depending on the reward received, it will try to update its estimate of the Q value. It will then move on to take the next step depending on the exploitation vs. exploration setup it has been given in the beginning of the training.

While *Q-Learning* is an effective method that could be used in TD setup, it has one major issue which has been explained by Hado van Hasselt in his Double Q-Learning paper(van Hasselt et al., 2015)(Hasselt, 2010). It seems that *Q-Learning* tend to perform really poorly in some stochastic environments and he has pointed out this this phenomenon is largely due to the overestimation that takes place because of the way this method tries to estimate the total reward(due to the nature of the max function that is used to formulation of this technique). Therefore, Hado van Hasselt, introduced *Double Q-Learning* in order to mitigate such issue.

The proposed solution is a simple, but rather elegant method. His Suggestion was to maintain 2 *Q-functions* instead of one, where each of them gets its update from the other for the upcoming state. Let's say we have two *Q-functions* Q^A and Q^B. Let a^* be the maximizer of Q^A in the next state. Then using $Q^B(s', a^*)$ we would try to update the Q^A and then perform a similar operation to update Q^B. You can see in the the pseudo-code for the mentioned method at **Algorithm 1**, which is directly from the DQN paper(Hasselt, 2010).

As mentioned before, we have used DQN model instead of the vanilla *Q-Learning* technique in order to make sure that we are not facing this issue while running our experiments, trying to see the effects of the differential training, without having to worry about complications that might arise by *Q-Learning* if we weren't using the DQN.

Algorithm 1 Double Q-Learning

Initialize Q^A, Q^B, s
repeat
  Choose a, based on $Q^A(s,.)$, and $Q^B(s,.)$, observe r, s'
  Choose(e.g.random) either UPDATE(A) or UPDATE(B)
  if UPDATE(A) **then**
    Define $a^* = argmax_a\ Q^A(s', a)$
    $Q^A(s,a) \leftarrow Q^A(s,a) + \alpha(s,a)(r + \lambda Q^B(s', a^*) - Q^A(s,a)$
  else if UPDATE(B) **then**
    Define $b^* = argmax_a\ Q^B(s', a)$
    $Q^B(s,a) \leftarrow Q^B(s,a) + \alpha(s,a)(r + \lambda Q^A(s', b^*) - Q^B(s,a)$
  end if
  $s \leftarrow s'$
until end

3. Differential Training of rollout Policies

In the following section, we discuss the details of our algorithm. Section 3.1 introduces the method proposed by Bertsekas and how it is supposed to help mitigate simulation and approximation errors. Subsequently, in the section 3.2 we provide more details regarding how we went about implementing our custom exploration method. Then, in the sections 3.3 and 3.4, we will go deeper into how our Bade DQN model and the Differential versions are implemented.

3.1. General Algorithm

As we have mentioned before, Bertsekas' main argument is the fact that most of the standard Reinforcement Learning techniques are not robust to errors and noise that exists in the system, specifically the approximation and the simulation error, and this is a major issue when it comes to practicality of such models in real life. Many of the real world application have inherent observation noises in them, which may degrade performance if not mitigated or accounted for. Moreover, he points out that although the previous attempts in mitigating the problem such as *Advantage Updating*(Harmon et al., 1994) have been successful, they have one major drawback. The issue is that some of the effective training methods such as Temporal Difference cannot be integrated with such techniques in a straightforward way, making it practically challenging to use these methods in a real world application.

In the technique which he calls *Differential Training*, we construct a function $\tilde{R}_k(s, s', g)$ which aims to approximate the reward-to-go difference $J_k(s) - J_k(s')$, where s and s' are any pair of states , and g is a tunable parameter vector. What gives this method and edge over the *Advantage Updating*, is its ability to utilize any of the standard RL methods including Temporal Difference, thanks to its special *differential definition of the pseudo-environment*, whose state is defined as a pair of the states, (s, s') taken from the original environment. As a result, this new pseudo-environment now has twice as many observation dimensions, and also twice as many action dimensions as well.

Before jumping to the main algorithm, Bertsekas goes over a similar approach to his *Differential Training* method which is worth mentioning as it might also help us better understand how the *Differential Training* helps reducing the error caused by the variance during the training. In this initial method, he suggests computation of a rollout control using Monte-Carlo simulation. In this method, we essentially simulate a large number of trajectories starting from an initial state of s, taking the action a, and then following the policy π from that point forward. Then, once we have all the generated states, considering all the possible actions $a \in \mathcal{A}(s)$, we will be able to calculate the rewards corresponding to all these trajectories by simply just averaging out the values we have computed through our simulations in order to come up with an approximation $\tilde{Q}_k(s,a)$ to the Q-function which is defined as:

$$Q_k(s,a) = E\{r(s,a,w) + J_{k+1}(f(s,a,w))\} \quad (9)$$

Keep this in mind that we will have simulation error here due to the fact that we are only using a limited amount of trajectories in order to come up with our approximation. Thus, the larger the number of simulated trajectories, the better the approximated function. Once, we have done this process for all $a \in \mathcal{A}(s)$, we can then generate an approximate rollout action control $\tilde{\mu_k}(s)$ through maximization of the Q function. Therefore we will have:

$$\tilde{\mu_k}(s) = arg \max_{a \in \mathcal{A}(s)} \tilde{Q}_k(s,a) \tag{10}$$

Now as mentioned before, there is a serious issue with this method due to the simulation error that exists in the calculation of the individual Q-functions. In addition to the randomness of the environment which manifests itself in the form of simulation errors in Q, there is also the other critical matter of observation noise, which adds another layer of uncertainty to this equation and method. In addition, the arg max function above is implying an implicit calculation of the Q-function differences($Q_k(s,a) - Q_k(s,\hat{a})$) for all the appropriate actions at each state s. As he points out in his paper, the concern that arises when computing these implicit differences, is that the effect of the simulation noise in each of the values of $Q_k(s,a)$ is now doubled when evaluating this difference, and this results in poor convergence during training. As mentioned before as one of the flaws of the Q-Learning algorithm, the natural property of the max function in the calculation of the Q-functions leads to an overestimation that is mainly responsible for the simulation error. What makes this a big issue is magnification of such error through the preceding differentiation operation(Bertsekas, 1997).

This is why he suggests the alternative idea of approximation through simulation of the Q-function difference $(Q_k(s,a) - Q_k(s,\hat{a}))$ itself by sampling the difference. He then moves on to mathematically show that by introducing the zero mean sample errors(D_k) and the assumption that the changes in the value of the action taken at the first state(a) have little effect on the value of the error D_k, relative to the effect induced by the randomness w_k in the system, one can guarantee that direct obtaining of the cost difference samples will result in lower error variance compared to the differentiation of cost samples(Bertsekas, 1997).

Now in order to approximate the Reward-to-go Differences, a common approach would be to compute the approximate rollout policy $\bar{\pi}$ in two steps. In the early stages of the learning the approximation of the reward-to-go functions could be possibly calculated using simulation and least squares fit from a parameterized class of functions such as Temporal Difference methods. In the second part of the process where we have progressed for a good amount, we can use the approximated reward-to-go function calculated previously in the first step, denoted $\tilde{J}_k$, and a state s at time k to approximate the Q-function

$$\tilde{Q_k}(s,a) = E\{r(s,a,w) + \tilde{J}_{k+1}(f(s,a,w))\} \tag{11}$$

for all $a \in \mathcal{A}(s)$. Then they can obtain an approximate rollout control $\tilde{\mu_k}(s)$ similarly to the initial method, where

$$\tilde{\mu_k}(s) = arg \max_{a \in \mathcal{A}(s)} \tilde{Q}_k(s,a) \tag{12}$$

Unfortunately, just as described before, this method also suffers from the simulation error and the error magnification that is inherent in the differentiation of the Q-functions. This leads to the alternative approach, *differential training*, which is the focus of this study. Similar to the initial technique, this method is also based on the reward-to-go difference approximation(Bertsekas, 1997).

In this setup, we can see that one can approximate the rollout action control $\tilde{\mu_k}(s)$ only by utilizing the differences of rewards-to-go. This is due to the fact that the when you want to calculate the max value similarly to the first method introduced before, the subtraction equals the differentiation of the reward-to-go functions as the state's rewards cancel each other out. Therefore, we consider approximating this difference using a function approximator, $\tilde{R_{k+1}}(s,\hat{s})$, which given the states s and $\hat{s}$, gives an approximation of $J_{k+1}(s) - J_{k+1}(\hat{s})$(Bertsekas, 1997). The rollout action control is then

$$\tilde{\mu_k}(s) = arg \max_{a \in \mathcal{A}(s)} \Phi \tag{13}$$

where:

$$\Phi = E\{r(s,a,w) - r(s,\mu_k(s),w) + \alpha \tilde{R}_{k+1}(f(s,a,w), f(s,\mu_k(s),w))\} \tag{14}$$

The Important factor that gives this function an advantage compared to other previously proposed methods, such as *Advantage Updating*, is the fact that training of an approximation architecture to obtain the $\tilde{R}_{k+1}$ can be done through any of the standard RL methods(Bertsekas, 1997). In order to see this more clearly, all you need to do is to rewrite the $\tilde{R}_k(s,\hat{s}$ using the recursive definition of the reward-to-go. Thus, we have:

$$R_k(s,\hat{s}) = J_k(s) - J_k(\hat{s}) \tag{15}$$

where

$$J_k(s) = E\{r(s,\mu_k(s),w) + J_{k+1}(f(s,\mu_k(s),w))\} \tag{16}$$

$$J_k(\hat{s}) = E\{r(\hat{s},\mu_k(\hat{s}),w) + J_{k+1}(f(\hat{s},\mu_k(\hat{s}),w))\} \tag{17}$$

we will then have for all $(s,\hat{s})$ and k

$$\tilde{R}_k(s,\hat{s}) = E\{r(\hat{s},\mu_k(\hat{s}),w) - r(\hat{s},\mu_k(\hat{s}),w) + R_{k+1}(f(\hat{s},\mu_k(\hat{s}),w), f(\hat{s},\mu_k(\hat{s}),w))\} \tag{18}$$

Therefore R_k can be seen as the reward-to-go function for a problem involving a fixed policy, the state$(s, \hat{s})$, and a reward per stage

$$r(\hat{s}, \mu_k(\hat{s}), w) - r(\hat{s}, \mu_k(\hat{s}), w) \tag{19}$$

and a transition equation

$$(s_{k+1}, \hat{s}_{k+1}) = (f(s, \mu_k(s), w_k), f(\hat{s}, \mu_k(\hat{s}), w_k)) \tag{20}$$

This design is what allows this architecture to be trained by any standard method that is used in the Reinforcement Learning literature.

3.2. Exploration

In this section we will discuss the details of our implementation of our custom exploration technique, and briefly elaborate on how the module operates. To remove some of the restrictions and limitations that is associated with the various environments we will use, especially environments such as LunarLander where exploration is key to an acceptable training, we opted to use a custom exploration technique to initialize the memory instead of a purely random agent. The method we have used is a Random-Network-Distillation method, in which a neural network has been used to serve as a reward to train and explorer agent for the exploration period of the training - which in our module was set to be 75,000 steps. During this time, the explorer agent receives the reward from the random neural network and takes gradient steps in the direction that maximizes the exploration reward. Simultaneously, all the state-action pairs generated, are used to train the original critic (classic or differential) as well.

3.3. Base DQN Implementation

As mentioned in the previous sections, our end goal is to demonstrate how differential training in a DQN setting differ from a conventional critic in that same context. As a result, we have implemented a separate critic that uses only the classic training iteration and updates -directly differentiating simulated and estimated Q values-, and we use this case as a reference and comparison point to gauge the performance of the differential DQN algorithm. It is worth noting that for the purpose of fair comparison, we have maintained the same number of parameters for any comparison cases that involved the classic and differential critic, to ensure that the comparison is fair and reliable. The parameters for this base DQN critic were mainly the size of the network's hidden layers, and the number of such layers. The higher the number of layers and the number of parameters, the more expressive and powerful the network will be.
This critic provides the Q-values for a given state-action pair, and the actor subsequently uses this critic and these values to update itself to take steps and get closer to the most rewarding action, at any given state, by taking the action that maximizes this Q-value in that particular state.

3.4. Differential DQN Implementation

The main part of this work focuses on evaluating this module in particular. The particular kind of critic used here, as described qualitatively in the previous sections, is a neural network, which represents the critic, that will learn and get trained on the differences of the Q-values for any two given states s_1 and s_2 and their respectively taken actions, a_1 and a_2. For a continuous environment, this network outputs the direct difference of the Q-values, meaning
$Q(s_1, a_1) - Q(s_2, a_2)$
and for discrete environment, the network will output, for a state pair of (s_1, s_2) a matrix of Q-values, one for each action pair (a_1, a_2) possible, with dimensions $m \times m$, where m is the number of discrete actions possible in the said environment.
Similar to the base critic described in section 3.3, the main tunable parameters of this differential critic are the size and number of hidden layers, and the more and higher they are, the more expressive this network will be.
However, compared to a base critic with equal number of hidden layers and similar size, the differential critic is at a slight disadvantage in terms of expressiveness. This is due to the fact that the task of the differential network, meaning learning the difference of the Q-values of any two given state-action pairs, is itself inherently a higher dimensional task than learning the individual values of Q, which happens to be the task of the base critic. So even though the overall size of the neural networks are selected to be the same, the differential critic suffers from a slight disadvantage, which could manifest itself in not being able to properly learn the Q-value differences. in that case, increasing the number of parameters will mitigate this issue

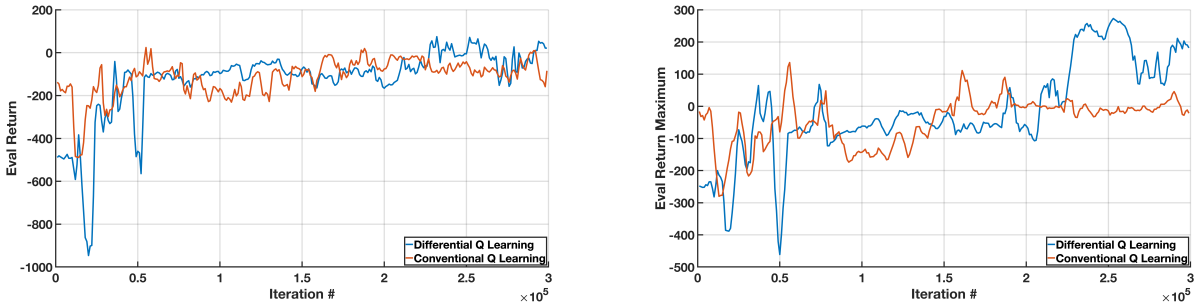
4. Experiments

In this section, different experiments are conducted to evaluate particular aspects of both the classic and base DQN algorithm and the differential training and differential DQN. In this section we answer the following questions:

1. How do the two algorithm differ in terms of performance in stochastic environments?

2. How does observation noise impact their performance and training convergence?

3. Are there cases where differential DQN is inherently unable to perform?



(a) (b)

Figure 1. Test and evaluation performance of classic DQN and differential DQN in the LunarLander (discrete) environment (neural network: n=2, size = 64). a) Average evaluation return of classic DQN and differential DQN. b) Maximum evaluation return of classic DQN and differential DQN.

4.1. Performance In Stochastic Environments

4.1.1. LunarLander (Discrete)

This environment includes a continuous set of observation space, and a discrete set of actions which are used as control for the spaceship in the environment. One of the challenges of this environment is exploration. This environment has a massive reward at the winning stage of the game, and conversely, a very large penalty if the spaceship crashes or doesn't land properly and where it should. As a result, it is very crucial for the agent to properly explore the state-action space to gather all necessary environment snapshots to learn the task at hand.

As mentioned previously, both methods, classic DQN and differential DQN, have an initial 75,000-step random-network-distillation exploration stage, to precisely help them overcome this exploration challenge in environments like the LunarLander-discrete.

Using the same size for both neural networks (critics), we have trained and evaluated this exploration/exploitation agent in this environment with both methods of DQN and the results of their performance is shown in Figure 1.

As observed in Figure 1a both methods are able to deliver improvements in the average return with each training step, but after about 200,000 training steps, the average return becomes much higher for the differential DQN compared to the classic DQN method - their return difference often reaching to more than 100.

In addition to this observation, it could also be seen in Figure 1b that using the differential DQN, the maximum return achieved is drastically higher than that of the case with classic DQN. This score of larger than 100 in 1b for the differential DQN, means that the spaceship has successfully learned how to land on the platform, a task that the other method was not successful at.

Looking at Figure 1 and comparing the two performances, one key factor is worth emphasizing, and that is the initial disadvantage that a differential critic inherently has compared to its classic counterpart. However, even under these circumstances, with the same size of network, it is still able to deliver a better performance than the classic DQN.

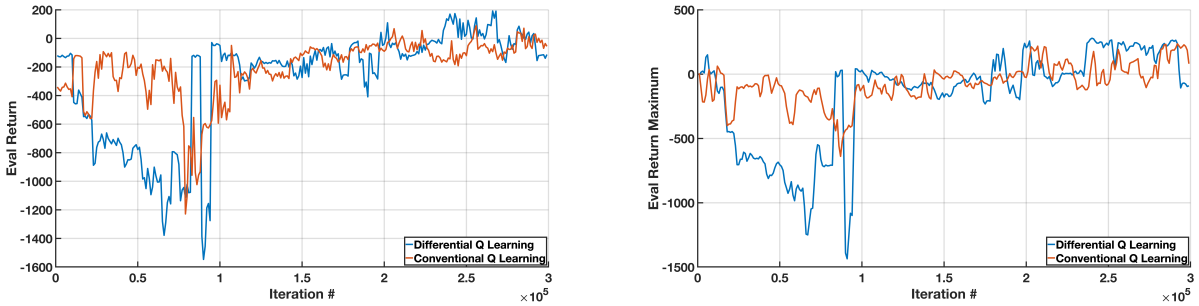
The differential DQN method outperforming classic DQN is a manifestation of the robustness of a differential critic to environment and simulation noise, which the classic DQN falls victim to and fails to achieve the required performance.

4.1.2. LunarLander (Continous)

This environment includes a continuous set of observation space, and a continuous set of actions which are used as control for the spaceship in the environment. Similar to the discrete case in section 4.1.1, the main challenges of this environment is exploration. The final reward in this environment is distibuted in such a way that makes it very unlikely to occur unless if using an extensive and powerful exploration technique. To this end, similar to its discrete counterpart, this task also includes a 75,000-step random-network-distillation exploration technique.

To compare the classic DQN and differential DQN, we have used similar-sized neural networks (as the critics) and trained the agent using those critics for 300,000 steps, with a batch size of 16.

The resulting performance during evaluation for both of those methods is brought and shown in Figure 2. Much like the discrete case, here the differential DQN is also able to outperform the classic DQN, both in terms of average return but also in the case of the maximum return achieved during training. As seen in Figure 2b, the differential DQN learns to land the spaceship a lot earlier during the training, by about 50,000 steps.



(a) (b)

Figure 2. Test and evaluation performance of classic DQN and differential DQN in the LunarLander (continuous) environment (neural network: n=2, size = 64). a) Average evaluation return of classic DQN and differential DQN. b) Maximum evaluation return of classic DQN and differential DQN.

4.1.3. HALFCHEETAH (CONTINOUS)

This environment, unlike the LunarLander cases, has a very larger observation space, as well as a higher dimensional action space, which puts a heavy limit and restriction on how expressive the neural network (critic) can be with a given size.

Unlike the LunarLander cases that we had in the previous section, this environment does not have a complicatedly distributed reward system, and is actually quite well behaved and smooth. As a matter of fact, the reward for this task is based on how fast and how far the half-cheetah is able to run, which is a very well-defined measure of performance. As a result, the initial exploration is not a very critical challenge for this task. However, for consistency of comparison, similar to the LunarLander cases, the module/agent for this environment also includes a 75,000-step random-network-distillation exploration stage.

Figure 3 shows the evaluation performances of both methods in this environment. Seen on the graph of Figure 3, one can observe that the classic DQN here has a slightly higher average return than the differential method. While this may look in conflict with the hypothesis presented in this work and in Bertsekas' paper, it is worth mentioning that this environment has a much broader and higher dimensional state and action space, and as a result, the critics become very parameter-limited since the dimensionality of the input is so large.

This issue becomes especially more pronounced and critical in the case of the differential DQN, since the critic needs to actually learn an even higher dimensional input space due to the doubling of the input size. As a consequence, its expressiveness and representation power is greatly reduced and becomes limited in terms of performance compared to a classic DQN with the same network size. To solve this issue, one can simply increase the size of the differential critic to restore enough functional representation power to the network.

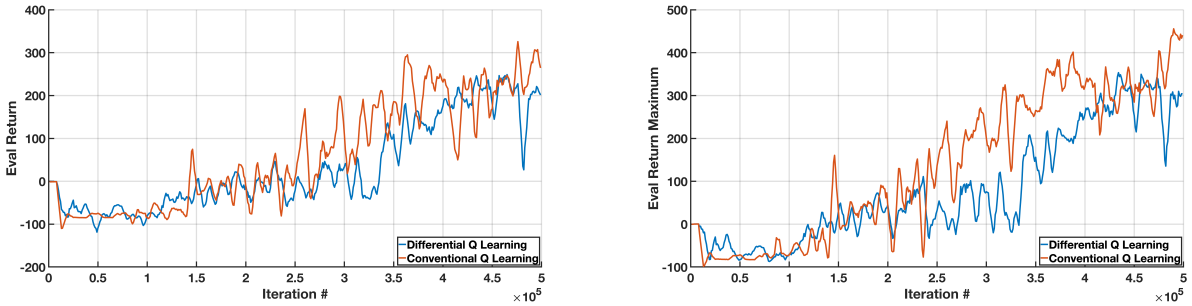
Though not quantitative, another more practical way of comparing the performances of the two methods is to look at the rendered performance in the environment and watch the agent at work within the grid. Upon extracting the rendered performances of the two cases, we observed that the classic DQN fails to actually learn how to walk or run properly, and is only able to achieve the rewards in Figure 3 by tripping and repeatedly falling over and over, whereas the differential DQN in fact learns how to walk smoothly without tripping or jumping like the classic DQN did.

4.2. Impact Of Observation Noise

In this section, we will compare the robustness of each method of training to observation noise, and how it affects the training convergence and trajectory. To this end, we will use a simpler and less complex environment with a well behaved reward distribution to demonstrate the impact of observation noise on the training process.

The environment being used for this experiment is the Pendulum environment which is a less complex environment in the gym package. This environment has one observation element, and one action element, both of which are continuous variables.

In this experiment, a zero-mean gaussian noise is added to the observation (1 dimension element), and the variance of this noise is varied as a percentage of the maximum magnitude of the observation space (in this case +1). This noise is added prior to the data being stored in the replay memory, and as a result, the critic will only observe the noisy states and use the noisy data during the training process, which will impact the convergence of the temporal difference updates of the critic. Figure 4 shows the average test return of the classic and differential critic at different noise levels. As seen in Figure 4(a), both methods have very similar performances with no noise present in the system. The similarity in performance persists with 3% noise level as well, but when the noise level reaches 5% and above, the difference



(a) (b)

Figure 3. Test and evaluation performance of classic DQN and differential DQN in the HalfCheetah (continuous) environment (neural network: n=2, size = 64). a) Average evaluation return of classic DQN and differential DQN. b) Maximum evaluation return of classic DQN and differential DQN.

between the two kinds of critics start to show itself. at 5% noise level injected into the system, the classic DQN fails and is unable to perform, however, the differential critic is still able to achieve a decent amount of return during the training, as seen in Figure 4 (c). At 7% noise level, both methods are failing, however the differential critic is still able to achieve a higher average return during the training process, which demonstrates that it is still able to recover some of the environment and reward information from the noisy data, and is thus more robust against observation noise than the classic DQN method.

4.3. Limitation Of Differential DQN

As we have observed, DQN is particularly a great practical architecture using which we often are able to achieve an acceptable performance after enough number of steps of training. The said statement holds for not only all classic DQN that use classic critics (meaning not differential), but also differential DQN as well . However, there are special cases where using a differential critic will not yield an acceptable performance return, no matter how many steps it is trained for. The differential version of such architecture fails at learning a stable model or even achieving a high enough reward (goal: 200) completely (See Figure.5 for more details on the performance). The main issue in this environment that causes such behavior for the differential DQN, is due to the fact that the environment has a reward distribution that is not compatible with differential critic training, namely all steps have a constant and fixed reward (in this case +1), including the last step (there are total of 200 steps). Since the differential models learn through approximation of difference of rewards, namely $r_a - r_b$, the critic does not get any non-zero reward information at every iteration, and therefore is unable to recognize the trajectory the agent is taking, and fails to converge to the target Q-values. In short, if the certain environment being utilized fails provide sufficient non-zero rewards during training, the model will not converge and training will fail (See Figure.5) resulting in poor evaluation performance.

As you can see in Figure.5, in the differential DQN graph, even though a few relatively higher rewards can be observed, these rewards occur very rarely and in addition to that, the agent never truly completes the task, namely never reaches the 200 value of reward achieved by the classic DQN. This environment, and other cases alike, are special cases where differential DQN, no matter how powerful, will not be able to be trained on.

5. Issues and Future Direction

The followings are future directions that we can pursue in order to further study this technique and improve:

- Apply the notion of differential critic and differential training to other standard Reinforcement learning techniques and study any performance improvement (or lack thereof) of such an adaptation.
- Investigate custom and application-specific exploration techniques to mitigate the differential DQN's inability to function in constant-reward environments, such as CartPole.
- Study the robustness of differential and classic DQN against action noise as well, which could be relevant in some application where the agent itself is noisy.

6. Conclusion

In this work, we have implemented the model proposed by Bertsekas called *Differential Training*(Bertsekas, 1997) and have investigated how much improvement one could achieve using this method for training using a DQN as our base model. Even though DQN models are already designed to reduce the error variance caused by the overestimation

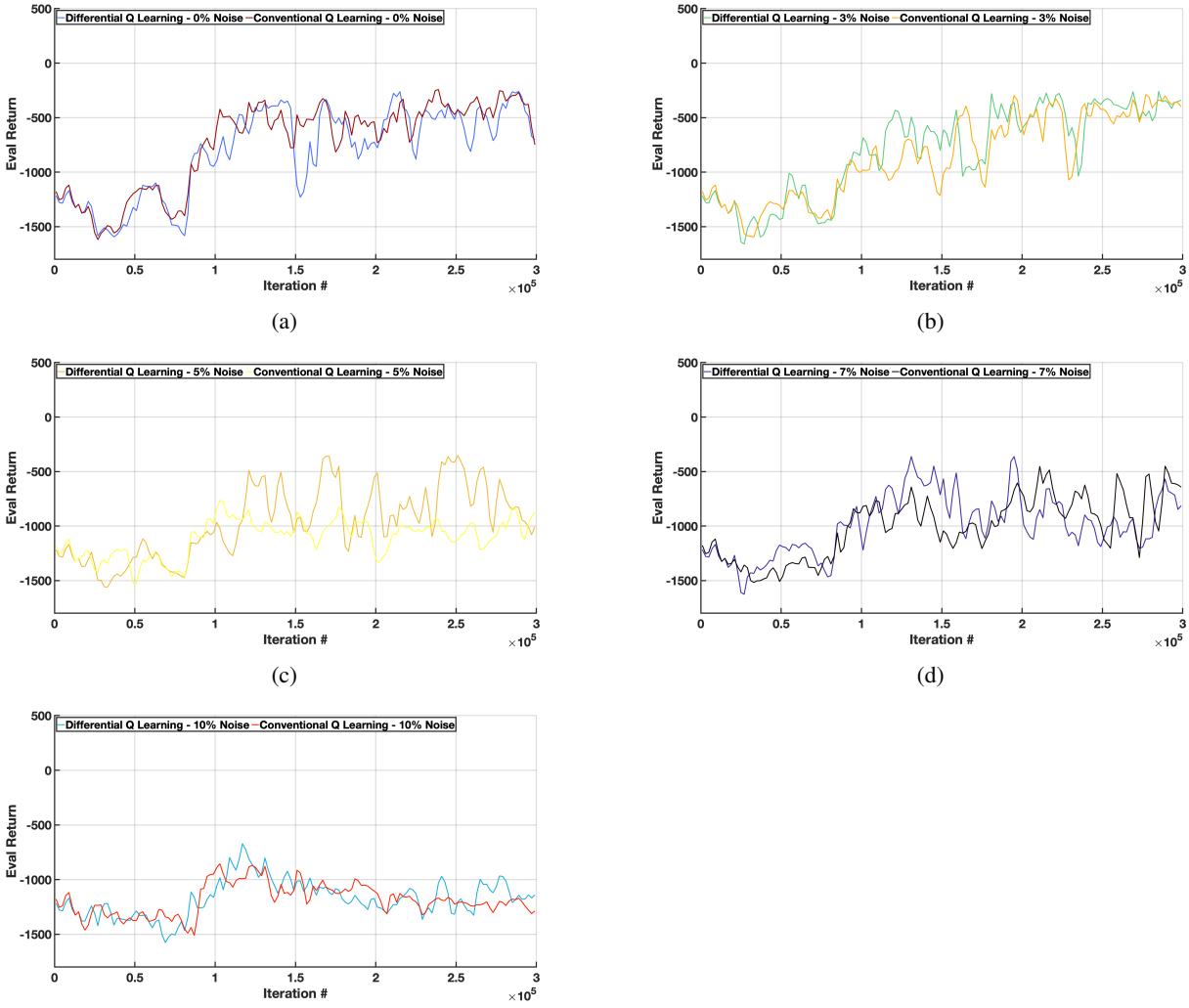


Figure 4. Performance of classic and differential DQN with different observation noise levels. (a) No noise (0%) (b) 3% noise. (c) 5% noise. (d) 7% noise. (e) 10% noise.

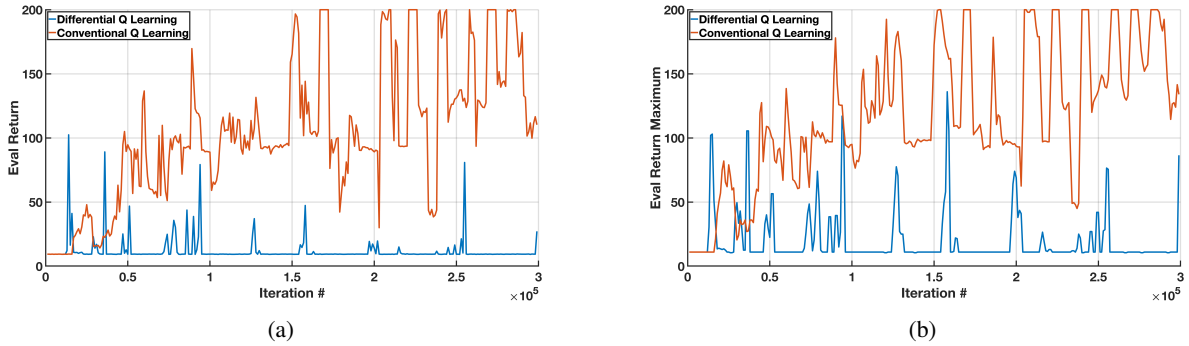


Figure 5. Test and evaluation performance of classic DQN and differential DQN in the CartPole (discrete) environment (neural network: n=2, size = 64). a) Average evaluation return of classic DQN and differential DQN. b) Maximum evaluation return of classic DQN and differential DQN.

in Q-Learning models, we were still able to achieve improved performance while integrating the *differential training* method in it in most of our experiments. The verdict from this investigative report is that differential training does provide substantial robustness against randomness in environment and noise in observation and state space, and thus is a preferred method to a classic DQN. However, as with any other method, differential DQN has its own limitation in terms of performance under certain circumstances, but one can argue that a better exploration technique designed specifically for this task could overcome those challenges and give the differential DQN its edge back compared to other classic DQN methods.

References

Bertsekas, D. P. Differential training of rollout policies. 1997. URL http://web.mit.edu/dimitrib/www/Diftrain.pdf.

Harmon, M. E., Baird, L. C., and Klopf, A. H. Advantage updating applied to a differential game. pp. 353–360, 1994.

Hasselt, H. Double q-learning. 23:2613–2621, 2010. URL https://proceedings.neurips.cc/paper/2010/file/091d584fced301b442654dd8c23b3fc9-Paper.pdf.

van Hasselt, H., Guez, A., and Silver, D. Deep reinforcement learning with double q-learning. 2015. URL https://arxiv.org/abs/1509.06461. arXiv:1509.06461.