Real-time System Control with Deep Reinforcement Learning

GUSTAV GYBÄCK
FREDRIK RÖSTLUND
Reglering i realtid med förstärkningsinlärning

GUSTAV GYBÄCK
FREDRIK RÖSTLUND
Abstract—We reproduce the Deep Deterministic Policy Gradient algorithm presented in the paper Continuous Control With Deep Reinforcement Learning to verify its results. We also strive to explain the necessary machine learning framework needed to understand the algorithm. It is a model-free, actor-critic algorithm that implements target networks and mini batch learning from a replay buffer to increase stability. Batch normalisation is introduced to make the algorithm versatile and applicable to multiple environments with varying value ranges and physical units. We use neural networks as function approximators to handle the large state and action spaces.

We can show that the algorithm can learn and solve multiple environments using the same set up. After proper training the algorithm has produced a real-time decision policy which acts optimally in any state given that the environment is not too sensitive to noise.

I. INTRODUCTION

There are many different applications for reinforcement learning in a multitude of industries. In manufacturing it can be used to make smarter robots whom in turn can speed up production, in the financial sector it is of great interest due to its ability to learn optimal trading policies. Our reinforcement learning algorithm has many similarities to the philosophy of tabula rasa. It is a model free algorithm that is applicable to many systems, learning how to behave in it and thrive according to the preset rules. So far the possible applications has been limited to simpler problems with discrete or low dimensional action spaces where it is possible to access the outcome of every action. The number of possible actions multiplies astonishingly fast for even a simple discretisation of a robot arm. If it has seven joints and they can bend left, right, or do nothing it will accumulate to $3^7 = 2187$ possible actions. For fine tuning of modern robots a naive guess is to be able to control it with $360^{9}$ precision which means $360^{7} \approx 7,8 \cdot 10^{17}$ actions. The discrete algorithms are simply not applicable. Recently a new reinforcement learning algorithm has been proposed in Continuous Control With Deep Reinforcement Learning [1] which is capable of learning in these high dimensional action spaces. This is made possible by the utilisation of neural networks as an approximations of functions which are almost impossible to learn completely. It is an actor-critic algorithm with a replay buffer that store transitions during exploration, these transitions are used in a mini batch to improve the learning rate and avoid learning from correlated data. Batch normalisation is introduced to address the issue of learning from different sources or different units of measure. The algorithm acts in a deterministic manner and to guarantee adequate exploration noise is added to the action space. When fully trained, the algorithm has produced a policy which can make decisions in real-time.

We will present a self contained explanation of the methods used in the algorithm.

II. BACKGROUND

A. Preliminaries

We work with reinforcement learning algorithms where an agent interacts with a deterministic environment $E$ in discrete timesteps as seen in Figure 1. We model the problem as a Markov decision process (MDP) [2] which comprises of: a compact state space $S$, an action space $A = \mathbb{R}^n$, an initial state distribution $p(s_1)$, transition dynamics $p(s_{t+1}|s_t, a_t)$, and a reward function $r(s_t, a_t)$ over $S \times A \rightarrow \mathbb{R}$ where $s_t \in S$, $a_t \in A$.

For each timestep $t$ the agent takes an action $a_t$ in $E$ according to a policy $\mu$, receives an observation $x_t \in S$ of $E$, and a reward $r_t = r(s_t, a_t)$ from $E$. To meet the Markov property, that every transition is independent of past events $p(s_{t+1}|s_1, a_1, \ldots, s_t, a_t) = p(s_{t+1}|s_t, a_t)$, we assume that the system is fully observed $x_t = s_t$.

![Common Reinforcement learning setup with an agent acting in an environment, gets a reward and a new state.](image)

In the general case, a policy is stochastic and maps states to a probability distribution over the action space $A$, but since we work with a deterministic policy that maps states to actions, one-to-one, instead of a probability distribution, the transition dynamic is $p(s_{t+1}|s_t, a_t) = \delta_{\mu(s_t), a_t}$. The return is the sum of the discounted rewards over all future state-actions pairs with the discount factor $\gamma \in (0, 1)$, $r^\gamma = \sum_{k=t}^{\infty} \gamma^{k-t} r(s_k, a_k)$.
The value function, \( V^\mu(s) = \mathbb{E}[\gamma \rho_s(S_1 = s; \mu)] \) and the action-value function, \( Q^\mu(s,a) = \mathbb{E}[\gamma r(S_1 = s, A_1 = a; \mu)] \) defines the expected return following the policy \( \mu \) starting in the state \( s_t \) and taking an action \( a_t \). We denote the discounted state distribution as \( \rho(s') = \int \sum_{t=1}^{\infty} \gamma^{t-1} p_t(s)p(s \rightarrow s', t, \mu) ds \). With it we can define the performance objective that we want to optimise as

\[
J(\mu) = \int S \rho(s) \int A \mu(s,a)r(s,a)dadS = \mathbb{E}_{s \sim \rho, a \sim \mu}[r(s,a)]
\]

where \( J(\mu) \) is the expected return with \( s \) chosen from the discounted state distribution \( \rho \) and \( a \) from the parameterised policy \( \mu_0 \) with the parameter vector \( \theta \in \mathbb{R}^n \).

To optimise \( J(\mu) \) we need to find the policy which follows the action-value function in each step and since none of them are given we will need to learn them. We will make use of the recursive property of the Bellman equation [3] to approximate the action-value function as \( Q^\mu(s_t, a_t) = \mathbb{E}_{s_{t+1} \sim \rho} \left[ r(s_t, a_t) + \gamma \mathbb{E}_{a_t+1 \sim \mu}[Q^\mu(s_t+1, a_t+1)] \right] \).

Since we will work with a deterministic policy, \( \mu \), the inner expectation is redundant and the remaining equation is simpler to work with.

\[
Q^\mu(s_t, a_t) = \mathbb{E}_{s_{t+1} \sim \rho} \left[ r(s_t, a_t) + \gamma Q^\mu(s_t+1, \mu(s_t)) \right] \tag{2}
\]

We need data to train our model, in MDP data is called observations and when learning from them we make two distinctions, on- and off-policy learning.

On-policy learning is when the samples used for training are generated by the same policy \( \mu^\theta \) along its trajectory in the environment. The distribution of samples is determined by the policy and if correlated samples are not treated properly it can lead to biased learning [4].

Off-policy learning is when we train our policy on samples that are generated by watching a second policy act in the environment. Learning the action-value function \( Q^\mu \) off-policy is possible since the expectation in equation (2) is solely dependent on the environment.

### III. Methods

Here we present the two learning methods, Q-learning [5] and Deterministic Policy Gradient [6] that we will combine in an architecture called actor-critic [7]. Due to the heavy computations and the difficulty of learning the true action-value in the continuous case we will utilise function approximation.

#### A. Function approximation

Function approximation is commonly used in mathematics to simplify expressions or to get an understanding of how it behaves in a small, well defined neighbourhood. In reinforcement learning we use it to address the huge state, and action spaces that we encounter in the continuous case. Optimising over every action which is used in discrete Q-learning isn’t viable when the number of possible actions goes to infinity. Function approximation address this and reduces the size of computations, thus improving the time efficiency dramatically.

Just introducing any approximation in the policy gradient algorithms won’t necessary guarantee that it follows the true gradient w.r.t \( \theta \). D. Silver et al [6] solved this and defined a class of compatible function approximators \( Q^\mu \) such that the gradient \( \nabla_\mu Q^\mu(s,a) \) won’t be affected if we replace it with \( \nabla_\mu Q^w(s,a) \).

1) **Conditions for compatible function approximators:**

A function approximator \( Q^w \) is compatible with a policy \( \pi\theta \), \( \nabla_\mu J(\theta) = \mathbb{E} \left[ \nabla_\mu \rho_{\pi\theta}(s) \nabla_\mu Q^w(s,a) | a = \pi_\theta(s) \right] \), if

1. \( \nabla_\mu Q^w(s,a) |_{a = \pi_\theta(s)} = \nabla_\mu \rho_{\pi\theta}(s) w \)
2. \( \gamma \) minimises the mean-squared error,

\[
MSE(\theta, w) = \mathbb{E} \left[ \kappa(s, \theta, w)^\top \kappa(s, \theta, w) \right] \quad \kappa(s, \theta, w) = \nabla_\mu Q^w(s,a) |_{a = \pi_\theta(s)} - \nabla_\mu Q^\mu(s,a) |_{a = \pi_\theta(s)}
\]

Proof is found in Appendix B.

#### B. Q-learning

Q-learning as proposed by Watkins [5] is an off-policy algorithm that wants to learn the action-value function \( Q \) by an iterative update function. \( Q \) is initiated as an arbitrary function and then slowly progress towards the experienced value for each update,

\[
Q^\mu(s, a) \leftarrow (1 - \alpha) \cdot Q^\mu(s, a) + \alpha \cdot \begin{cases} \text{old value} & \text{if } s' \neq s \\ \text{learning rate} \cdot (r(s, a) + \gamma \cdot \max_a [Q^\mu(s', a)]) & \text{otherwise} \end{cases}
\]

where \( s' \) is the next state after taking action \( a \) in state \( s \). It utilises the Bellman property, which we can see inside equation (2), to find the true value in a transition. To guarantee convergence and learn \( Q \) each state-action pair needs to be visited infinitely often [8], this grows impractical for a continuous algorithm because of the size of the state-action pair set. Parameterised Q-learning uses an approximation which learns by minimising a loss function for each update. It is similar to the previously mentioned one but is adapted to a parameterised approximation of \( Q \). The loss function \( L(\theta^Q) \) is designed to describe the squared difference between the estimate of \( Q(s_t, a_t) \) and \( r(s_t, a_t) + \gamma Q(s_{t+1}, \mu(s_{t+1})) \).

\[
y_t = r(s_t, a_t) + \gamma Q(s_{t+1}, \mu(s_{t+1})) \tag{4}
\]

\[
L(\theta^Q) = \mathbb{E}_{s_t \sim \rho, a_t \sim \beta, r_t \sim E} \left[ |Q(s_t, a_t | \theta^Q) - y_t |^2 \right] \tag{5}
\]

Minimising the loss function optimises the Q-function approximation for the given transition and not for all transitions and can thus become unstable. Implementing the loss in an update function will increase its stability but slow down the learning rate [1].

#### C. Policy gradient

Policy gradient algorithms as defined in Deterministic Policy Gradient Algorithms [6] try to optimise the parameterised policy \( \mu_\theta \) by evaluating the performance objectives gradient and adjust the parameter vector \( \theta \) to converge towards it. It is guaranteed to converge to at least a local minima. To escape the local minimas we need adequate exploration when
learning which can be achieved by introducing noise in the action space.

\[
\nabla_\theta J(\mu_\theta) = \int_S \rho_\mu(s) \nabla_\theta \mu_\theta(s) \nabla_a Q_\theta(s, a)_{a=\mu_\theta(s)} ds \\
= \mathbb{E}_{s \sim \rho_\theta} \left[ \nabla_\theta \mu_\theta(s) \nabla_a Q_\theta(s, a)_{a=\mu_\theta(s)} \right]
\]

(6)

**Proof** is found in appendix A.

**D. Actor-critic**

Based on the policy gradient and Q-learning, the actor-critic is an algorithm which uses two simultaneous working components. The critic learns a parameterised approximation of the action-value function \(Q^\mu(s, a) \approx Q^\mu(s, a)\) according to equation (3) and is then used by the actor to sample the policy gradient. The actor function adjusts the parameters \(\theta\) in the policy \(\mu\), which maps states to an action, by gradient descent, equation (6), where the sampled Q-function from the critic is used.

Both of these functions are parameterised and are thus approximations of said functions. This architecture with approximations was first introduced by Sutton [7] two decades ago and is still widely used [6].

**IV. ALGORITHM**

The Deep Deterministic Policy Gradient (DDPG) is an actor-critic algorithm based on the DPG algorithm [1]. Here the critic \(Q(s, a)\) approximates the action-value function and is learned through the Bellman equation, while the parameterised actor function \(\mu(s|\theta^\mu)\) is updated by applying the deterministic policy gradient theorem:

\[
\nabla_\theta J(s) \approx \mathbb{E}_{s \sim \rho_\theta} \left[ \nabla_\theta Q(s, a|\theta^Q)_{a=s, \mu=s|\theta^\mu} \right] \\
= \mathbb{E}_{s \sim \rho_\theta} \left[ \nabla_a Q(s, a|\theta^Q)_{a=s, \mu=s|\theta^\mu} \nabla_\theta \mu(s|\theta^\mu)_{a=s} \right]
\]

(7)

Three areas of concern for a naive implementation are stability in learning, generality across environments and ability to fully explore an environment. To this end a few additional structures are included into the DDPG algorithm, these are presented in this section.

**A. Batch Learning with Replay Buffers**

Learning in higher dimensional state and action spaces as we do in the continuous case grows impractically slow without the introduction of large non-linear function approximators. Due to the non-linearity this inclusion means that the learning is no longer guaranteed to converge. A contemporary work to the DPG shows that by utilising neural networks as function approximators and learning in batches will ensure increased stability [9]. Batch learning however scales poorly to larger networks so the DDPG therefore instead learns from mini batches.

Drawing inspiration from the Deep Q-network algorithm [10], DDPG introduces a replay buffer \(R\). The replay buffer is used to store transitions, in the form of the current state, the action taken, the reward received and the resulting state \((s_t, a_t, r, s_{t+1})\), during exploration of the environment. This buffer supplies a mini batch, \(M\) of size \(N \in \mathbb{Z}\), of randomly sampled transitions at each update for the actor and the critic to learn from. The randomly selected transitions lets the algorithm train from uncorrelated samples which increases the learning rate [1]. Once the buffer is filled with transitions, the newest sample replaces the oldest.

**B. Target Networks**

The implementation of neural networks as function approximators also increases the risk of divergence for Q-learning due to the fact that the learnt network \(Q(s, a|\theta^Q)\) is also used to calculate the update target \(y_i\) in equation (5). The DDPG algorithm mitigates this problem with the introduction of so called target networks adapted to work with the actor-critic framework. Two new networks are created, \(Q'(s, \theta'^Q)\) and \(\mu'(s|\theta'^\mu)\), these are used to calculate the target values of the critic and the actor respectively. These target networks are updated at a slower rate than the other networks \(\theta' \leftarrow \tau \theta + (1-\tau)\theta'\), \(\tau \ll 1\) where \(\tau\) is the learning rate of the target networks. The slower update rate increases stability to a large enough degree to offset the slight decrease in learning speed.

**C. Batch Normalisation**

A technique adopted into the DDPG to generalise the network input is called *batch normalisation*. A problem which may occur when exploring different environments is the possible variance in observation data, both different physical units and varying ranges. Batch normalisation is a technique which solves this by normalising the samples in each mini batch to have unit mean and variance for each dimension in the batch. This would otherwise have to be manually tuned for each separate environment. Batch normalisation also reduces internal covariate shift [11], an occurrence during training caused by the dependence of a layers input on all the preceding layers. This means that the layers in a network must continuously adapt to new input distributions, thus the problem is of greater concern the deeper the network is.

**D. Noise**

Thorough exploration of an environment is very important as otherwise certain features, such as a large reward at an end goal, might be missed during training causing the learnt policy to be of poor quality. A way to ensure an adequate exploration is a noise function \(N\), this noise is added when the actor selecting which actions to take \(\mu'(s_t) = \mu(s_t|\theta'^\mu) + N\), thus ensuring that more possible actions will be explored. Caution is needed when choosing a noise since a strong noise might push the algorithm past a narrow optima. DDPG uses noise generated from a Ornstein-Uhlenbeck process [12], the process generates temporally correlated values tending closer towards zero with each application. While not part of the original DDPG we also tested a new form of noise called parameter space noise [13], which will be explain further in the discussion.
C5A SYSTEM CONTROL WITH DDPG

Algorithm 1: DDPG algorithm [14]
Randomly initialise the critic network \( Q(s,a;\theta^Q) \) and an actor network \( \mu(s;\theta^\mu) \) with weights \( \theta^Q \) and \( \theta^\mu \).
Initialise target networks \( Q' \) and \( \mu' \) with weights \( \theta^Q' \leftarrow \theta^Q \) and \( \theta^\mu' \leftarrow \theta^\mu \).
Initialise replay buffer \( R \).
Initialise return list \( \text{Return} \).

for episode = 1, \( G \) do
  Initialise a random process \( \mathcal{N} \) for action exploration.
  Receive initial observation state \( s_1 \).
  for \( t = 1, T \) do
    Select action \( a_t = \mu(s_t;\theta^\mu) + \mathcal{N} \) according to the current policy and exploration noise.
    Execute action \( a_t \) and observe the reward \( r_t \) and the new state \( s_{t+1} \).
    Store transition \( (s_t, a_t, r_t, s_{t+1}) \) in \( R \).
  end
  Sample a random mini batch \( M \) of \( N \) transitions from \( R \).
  Set \( y_i = r_t + \gamma Q'(s_{t+1}, \mu'(s_{t+1};\theta^\mu')|\theta^Q') \).
  Update critic by minimising the loss:
  \[ L = \frac{1}{N} \sum_{i \in M} (y_i - Q(s_i, a_i|\theta^Q))^2 \]
  Update the actor policy using the sampled policy gradient:
  \[ \nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s_i, a|\theta^Q)|_{a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s_i;\theta^\mu) \]
  Update the target networks:
  \[ \theta^Q' \leftarrow \tau \theta^Q + (1 - \tau) \theta^Q' \]
  \[ \theta^\mu' \leftarrow \tau \theta^\mu + (1 - \tau) \theta^\mu' \]
  for \( t = 1, T \) do
    Select action \( a_t = \mu(s_t;\theta^\mu) \) to evaluate the current policy.
    Execute action \( a_t \) and observe the reward. Store reward in \( \text{Return} \).
  end
end

V. RESULTS

We have tested the algorithm with code and environments provided by OpenAI Gym [15] and Mujoco [16]. MountainCar is a relatively simple environment where the agent tries to drive a car up a slope however the engine is not strong enough so the car needs to first drive up an opposing slope to gather momentum. A positive reward is earned after reaching the goal while a small negative reward is imposed when engine power is used. The HalfCheetah environment has the agent try to run as far and efficiently as possible, a rendition of both these environments can be see in Figure 2. After 20k time-steps the policy is updated and the learnt policy is tested without added noise, Figure 3 and 4 shows the performance of the policy plotted over time trained. In the MountainCar environment the algorithm learns an optimal policy after two to three updates while the HalfCheetah environment could not be solved perfectly by any of the set-ups we tried.

VI. DISCUSSION

A. Exploration and Noise
Ensuring a sufficient exploration is one of the most important factors for learning optimal policies. A common problem in some environments, such as HalfCheetah, is the possibility for the algorithm to optimise towards a local minimum. There exist many proposed ways to mitigate this issue each limited in its own way, such as requiring additional complicated
structures, having a narrow area of application or disregarding temporal structure, thus requiring more samples [13]. DDPG adds temporally correlated noise to the actions chosen to increase exploration, however this is not sufficient to learn optimal policies in certain environments. A similar solution to this is a new method which has shown success by applying noise directly to the parameters of the current policy \( \theta = \theta + N \), where \( N \) is Gaussian noise. Parameter space noise initially behaves similarly to temporally correlated noise but is able to keep exploring more options and is therefore less likely to get stuck in an undesirable policy. As can be seen in 4 this yields greatly improved results over action space noise. Blindly increasing the strength of the noise did not prove fruitful after a certain point, we believe this is due to the very erratic nature of the exploration making it difficult for the algorithm to settle on a particular policy. Parameter space noise has been further shown to yield improved exploration and therefore quality of the learnt policies in a variety of environments especially ones with a sparse reward function [13, Figure. 2].

B. Hyperparameters

In the implementation of the algorithm multiple hyperparameters were chosen by field standards. Many of these have a big influence on the performance of the algorithm such as the type and strength of the noise, the learning rate \( \tau \), or the size of the buffers. This can be seen in how different the algorithm learns the HalfCheetah environment when the strength of the parameter noise varies. Despite this no conclusive strategy on how such parameters can be optimally chosen have been found, this is an open field of research which currently garners a lot of attention.

C. Future work

During this project we have found some improvements that can be made to the algorithm and the underlying framework.

A study on environment construction to examine how the reward affects the learning rate of algorithms like DDPG who are sensitive to environments with sparse rewards. Adding potential fields based on the rewards size and location might improve learning and exploration.

There are still many unexplored fields in Machine Learning that can improve or ratify field standards. Network depth and width along with its structure affect the performance in undocumented ways.

Test if a combination of noise can improve early learning rate and still achieve an optimal policy in sparse and complex environments.

VII. CONCLUSION

The Deep Deterministic Policy Gradient algorithm can properly learn competitive policies across multiple continuous environments, with very little input and no prior knowledge of said environment. Despite the implementation of neural networks as function approximators learning proved to be stable in each of the environments tested though the learnt policy was not necessarily optimal for a more complex environment like the HalvCheetah. The algorithms main limitation is a propensity to get stuck in non-optimal policies, mostly in complex environments which require a lot of exploration. The algorithm is however compatible with newer ways to increase exploration and we believe it can be further optimised in the future making it applicable in even more areas.

APPENDIX A

DETERMINISTIC POLICY GRADIENT THEOREM [6]

A. Regularity conditions

\[
p(s'|s,a), \nabla_a p(s'|s,a), \mu_\theta(s), \nabla_\theta \mu_\theta(s), r(s,a), \\
\nabla_a r(s,a), p_1(a) \text{ are continuous in all parameters and variables } s, a, s'.
\]

B. Proof of the deterministic policy gradient theorem

The proof follows the same lines as the stochastic policy gradient in Sutton [7] The regularity conditions imply that both \( V^{\mu_\theta}(s) \) and \( \nabla_\theta V^{\mu_\theta}(s) \) are continuous functions of \( \theta \) and \( s \) and the compactness of \( S \) implies that for any \( \theta, ||\nabla_\theta V^{\mu_\theta}(s)||, ||\nabla_a Q^{\mu_\theta}(s,a)||_{a=\mu_\theta(s)} ||\nabla_\theta \mu_\theta(s)|| \) are bounded functions of \( s \). We will use this to exchange derivatives and integrals, and the order of integration whenever necessary in the following proof.

\[
\nabla_\theta V^{\mu_\theta}(s) = \nabla_\theta Q^{\mu_\theta}(s,\mu_\theta(s)) = \\
\nabla\theta \left( r(s,\mu_\theta(s)) + \int S \gamma p(s'|s,\mu_\theta(s))V^{\mu_\theta}(s')ds' \right) = \\
\nabla_\theta \theta \mu_\theta(s) \nabla_a r(s,a)|_{a=\mu_\theta(s)} + \\
\nabla_\theta \int S \gamma p(s'|s,\mu(s))V^{\mu_\theta}(s')ds' = \\
\nabla_\theta \mu_\theta(s) \nabla_a r(s,a)|_{a=\mu_\theta(s)} + \\
\nabla_\theta \int S \gamma p(s'|s,\mu(s))\nabla_\theta V^{\mu_\theta}(s')ds' + \\
\int S \nabla_\theta \mu_\theta(s) \nabla_a p(s'|s,a)|_{a=\mu_\theta(s)} V^{\mu_\theta}(s')ds' = \\
\nabla_\theta \mu_\theta(s) \nabla_a \left( r(s,a) + \int S \gamma p(s'|s,a)V^{\mu_\theta}(s')ds' \right)|_{a=\mu_\theta(s)} + \\
\int S \gamma p(s'|s,\mu_\theta(s))V^{\mu_\theta}(s')ds' = \\
\nabla_\theta \mu_\theta(s) \nabla_a Q^{\mu_\theta}(s,a)|_{a=\mu_\theta(s)} + \\
\int S \gamma p(s \rightarrow s',1,\mu_\theta)\nabla_\theta V^{\mu_\theta}(s')ds'
\]
Further expanding this gives us an iterative formula that will will take the sum of.

\[
\begin{align*}
= & \nabla_{\theta} J(\theta) = \nabla_{\theta} \int_{S} p(s) V^{\mu}(s) ds \\
= & \int_{S} p(s) \nabla_{\theta} V^{\mu}(s) ds \\
= & \int_{S} \sum_{t=0}^{\infty} \gamma^t p(s \rightarrow s', t, \theta) \nabla_{\theta} \mu_{\theta}(s', \theta) \nabla_{\theta} Q^{\mu}(s', a)|_{a=\mu_{\theta}(s')} ds' ds \\
= & \int_{S} p^{\mu}(s) \nabla_{\theta} \mu_{\theta}(s) \nabla_{\theta} Q^{\mu}(s, a)|_{a=\mu_{\theta}(s')} ds \\
\end{align*}
\]

Which we use to expand the following.

\[
\begin{align*}
\nabla_{\theta} J(\theta) &= \nabla_{\theta} \int_{S} p(s) V^{\mu}(s) ds \\
&= \int_{S} p(s) \nabla_{\theta} V^{\mu}(s) ds \\
&= \int_{S} \sum_{t=0}^{\infty} \gamma^t p(s \rightarrow s', t, \theta) \nabla_{\theta} \mu_{\theta}(s', \theta) \nabla_{\theta} Q^{\mu}(s', a)|_{a=\mu_{\theta}(s')} ds' ds \\
&= \int_{S} p^{\mu}(s) \nabla_{\theta} \mu_{\theta}(s) \nabla_{\theta} Q^{\mu}(s, a)|_{a=\mu_{\theta}(s')} ds \\
\end{align*}
\]

**APPENDIX B**

**COMPATIBLE FUNCTION APPROXIMATION** [6]

A function approximator \( Q^{w} \) is compatible with a policy \( \pi_{\theta} \), \( \nabla_{\theta} J(\theta) = \mathbb{E} \left[ \nabla_{\theta} \mu_{\theta}(s) \nabla_{\theta} Q^{w}(s, a)|_{a=\mu_{\theta}(s)} \right] \), if

1. \( \nabla_{\theta} Q^{w}(s, a)|_{a=\mu_{\theta}(s)} = \nabla_{\theta} \mu_{\theta}(s) \nabla_{\theta} Q^{w}(s, a)|_{a=\mu_{\theta}(s)} \)

2. \( y \) minimises the mean-squared error, \( MSE(\theta, w) = \mathbb{E} \left[ \kappa(s; \theta, w) \nabla_{\theta} \mu_{\theta}(s, \theta, w) \right] \) where

\[
\kappa(s; \theta, w) = \mathbb{E} \left[ \kappa(s; \theta, w) \nabla_{\theta} Q^{w}(s, a)|_{a=\mu_{\theta}(s)} \right]
\]

A. Proof of the compatible function approximation

If \( w \) minimises the \( MSE(\theta, w) \) w.r.t \( w \) must be zero. We then use the fact that, by condition 1,

\[
\nabla_{w} MSE(\theta, w) = 0
\]

\[
\begin{align*}
\mathbb{E}[\nabla_{\theta} \mu_{\theta}(s) \kappa(s; \theta, w)] &= 0 \\
\mathbb{E}[\nabla_{\theta} \mu_{\theta}(s) \nabla_{\theta} Q^{w}(s, a)|_{a=\mu_{\theta}(s)}] &= \mathbb{E}[\nabla_{\theta} \mu_{\theta}(s) \nabla_{\theta} Q^{w}(s, a)|_{a=\mu_{\theta}(s)}] \\
&= \nabla_{\theta} J(\theta) + \nabla_{\theta} J_{\beta}(\theta) \quad (11)
\end{align*}
\]

ACKNOWLEDGMENT

We would like to thank our twin project group and its members, Gustaf Jakobzon and Martin Larsson, for insightful discussions and help with the set up of the environment as well as our professor and supervisor, Alexandre Proutiere, for guiding us through the wonders of reinforcement learning and the rigid mathematics that lies beneath it.

REFERENCES


