DEMOCRITUS UNIVERSITY OF THRACE | DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING
MSc in QUANTUM COMPUTING AND QUANTUM TECHNOLOGIES
NATIONAL CENTRE FOR SCIENTIFIC RESEARCH "DEMOKRITOS"

M.Sc. in Quantum Computing and Quantum Technologies

Department of Electrical and Computer Engineering, Democritus
University of Thrace (DECE)
Institute of Nanoscience and Nanotechnology
National Centre of Scientific Research "Demokritos" (INN-D)

Qubit control with Reinforcement Learning methods

Dimitris Koutromanos 60632

Supervisor: Prof. Emmanuel Paspalakis

April, 2024

Contents

Abstract

Progress in machine learning during the last decade has a considerable impact on many areas of science and technology, including quantum technology. In this article, we explore the application of Reinforcement Learning (RL) methods to the quantum control problem of state transfer in a single qubit. The goal is to create an RL agent that learns an optimal policy and thus discover optimal pulses to control the qubit. The most crucial step is to mathematically formulate the problem of interest as a Markov Decision Process (MDP). This enables the use of RL algorithms to solve the quantum control problem. Deep learning and the use of deep neural networks provide the freedom to employ continuous action and state spaces, resulting to expressivity and generalization of the process. We exploit this flexibility and formulate the quantum state transfer problem as a MDP in several different ways. We also propose an alternative formulation that enables RL algorithms to approximate the optimization process which find the optimal parameters for the control function represented as finite trigonometric series. We test all the developed methodologies by applying them to the fundamental problem of population inversion in a qubit (creation of the NOT gate). In most cases, the derived optimal pulses achieve fidelity equal or higher than 0.9999, as required by quantum computing applications.

Keywords: Quantum Technologies, Quantum Control, Machine Learning, Reinforcement Learning, Qubit systems

Acknowledgments

I would like to acknowledge and express my gratitude to my supervisor Prof. Emmanuel Paspalakis for his guidance and for making this work possible. His support and advice were really valuable and carried me throughout the process of writing this thesis. I would like to thank the members of the committee Prof. Dionisios Stefanatos and Prof. Ioannis Thanopulos for their help, their constructive comments in the progress of this work, and their participation in the evaluation of this thesis.

I would also like to express my special thanks to my partner Nikoletta and my family for their continuous support and their understanding during the writing of the current thesis.

Dedicated to my father.

Chapter 1

Introduction to quantum control and thesis goals and structure

Quantum control is the discipline that addresses the problem of controlling a quantum system and creating desired behaviours. The dynamics of quantum systems are governed by quantum mechanics and depends on one or more control functions [12]. The controlled objects are quantum objects, such as atoms, molecules, quantum dots, quantum spins, superconducting qubits, NV centers in diamond, etc, and the control functions are electromagnetic fields (or sometimes time-dependent magnetic fields) that are shaped in an appropriate way. Quantum control is mainly involved in the dynamics of the quantum system. Some of the most common methods that are used in the control of quantum systems include resonant methods [33], adiabatic methods, like rapid adiabatic passage [33, 36] and stimulated Raman adiabatic passage [41], variances of adiabatic techniques termed shortcuts to adiabaticity [18], and optimal control methods [5, 15].

The above mentioned methods are well established and tested for many years. They are already applied to current quantum devices, such as the existing quantum computers. The rise of Artificial Intelligence and its application in many real-life applications is a common fact in the last decade. More precisely, the specific discipline of machine learning, called Reinforcement Learning (RL) is of great interest in many real life applications which are related to decision making and optimal control. For that reason it is quite natural to wonder whether Reinforcement Learning methods can be applied to Quantum Control [44] and Quantum Technologies in general [9, 22]. It is believed that AI could help Quantum Control techniques to be more effective, it could give alternative methods of solving problems and perhaps it could give some insights with new techniques that otherwise would be difficult to be found by existing methods. There are also other categories of Machine Learning other than Reinforcement Learning, such as Supervised Learning, that can be used in qubit characterization [8], and assist qubit manipulation and readout [4]. We stress that in the present work only RL methods will be applied.

The mathematical formulation of quantum mechanics is well established for several decades. The big question is whether problems from quantum control can be formulated and structured in the right mathematical context that is important and crucial for RL methods to be applied. This context can be described by the Markov Decision Process (MDP). The structure of an MDP will be given in a following chapter of the thesis. It turns out that many problems from Quantum Control can be formulated as MDPs, so a new set of methods from RL are now available in the arsenal of Quantum Control. This is what the current master thesis will try to investigate. More specifically, we will try to address the following basic questions:

- How and in which ways quantum systems can be formalized as MDPs? This is the first and most important step which will allow RL methods to solve quantum control problems.
- Which RL methods can be used in the current context?
- How can the control be done in an optimal way with results equivalent to optimal control methods?

The current thesis uses RL methods to control the basic quantum information system of two level system [36] or qubit. First, tabular methods are applied, in which state and action spaces are finite and low dimensional. Their name comes from the fact that value functions are represented by tables. Their use is although limited, since a specific MDP formulation is required, so that it is finite dimensional. If one wants to increase and generalize the problem, approximate methods are the appropriate ones. They arise in a more natural way, since using the state of the quantum system as the state of the MDP gives rise to an infinite dimensional state space. On top of that, if pulses are not restricted to step-size functions, the actions of the MDP which involve the electromagnetic pulse, belong to infinite dimensional action spaces. Approximate methods are used for very high dimensional or infinite dimensional state and action spaces. More specifically, Deep Neural Networks (NN) are used as approximators for the value functions or the policies.

There is no clear MDP dynamics model that can be followed so that dynamic programming techniques can be applied. Thus model-free methods of RL are used throughout the thesis trying to control the qubit system. There are already several papers that use RL methods to control quantum systems [7, 14, 35, 27, 11, 29, 28, 6, 2, 25]. All this prior knowledge will be used to build a study of many different formulations of an MDP for the basic quantum system of qubit. Each formulation can help to explore many different RL methods to solve the same problem. The main goal is to produce optimal pulses for the effective control of the qubit and try to achieve fidelities and results similar to optimal control. Dynamic programming and optimal control are closely connected. Using this connection we will try to apply approximate dynamic programming or RL to control quantum systems and more specifically solve the quantum state transfer problem optimally.

On the realization of the control problem, we note that all the algorithms have been developed using the open source machine learning platform TensorFlow [1]. TensorFlow includes a specific library for Reinforcement Learning, named TF-Agents which accelerates the algorithm implementation and execution. The quantum system evolution is simulated by QuTiP [20], which is an open-source software for simulating the dynamics of quantum systems. All algorithms will be uploaded in a github repository and will be fully accessible.

The structure of the master thesis is the following: After the theoretical analysis of the MDP (Chapter 2) and the Dynamic Programming (Chapter 3) which solves the MDPs, then there is a small introduction in Deep learning (Chapter 4), followed by an extensive description of the RL methods (Chapter 5) that will be used to control the qubit system. Then, the practical application of qubit system along with the formulation of the MDPs for the control of the qubit, the relevant results and discussion of the RL algorithms (Chapter 6) are presented. The master thesis ends by presenting some conclusions and future directions (Chapter 7).

Chapter 2

Markov Decision Processes

Markov Decision Process (MDP) is a formalization of sequential decision making in which actions affect both immediate and future rewards. MDP is the mathematical basis of dynamic programming and reinforcement learning problems, so that theoretical statements can be made [38].

2.1 Agent-Environment interplay

MDP is the framework for the problem of learning from interaction with the environment to achieve a goal. The entity that is learning from this interplay and is making the decisions is called an agent. The entity that the agent interacts with is called the environment. The agent selects actions based on observations of the environment, and the environment responds to these actions with rewards and new states in which the agent is put after the action. The agent tries to maximize its rewards over time through its actions. The MDP framework is an important formulation of the problem of goal-directed learning from interaction.

The agent interacts at discrete time steps $t \in \mathbb{N}$. At each step, the agent receives a representation of the environment, which is called state, denoted by $S_t \in \mathcal{S}$. Based on the state, the agent selects an action $A_t \in \mathcal{A}$. In the next step, the response of the environment to the agent's action is a numerical reward, $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$. In the new step, the environment will be in a new state S_{t+1}. Following this procedure, a sequence of states, actions and rewards is generated, also called a trajectory:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, ...$$

If sets of states ($\mathcal{S}$), actions ($\mathcal{A}$) and rewards ($\mathcal{R}$) are finite, then the MDP is called finite. In the finite case, the random variables R_t and S_t have well defined discrete probability distributions that depend on the preceding state and action. This means that there is a probability that particular values of these variables, $s' \in \mathcal{S}$ and $r \in \mathcal{R}$, occurring at time step t, given the preceding state s and action a:

$$\begin{aligned} &\forall s', s \in \mathcal{S}, r \in \mathcal{R}, a \in \mathcal{A} : \\ &\mathbf{p(s', r|s, a)} := \mathbf{Pr\{S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a\}} \end{aligned} \tag{2.1}$$

The function p defines the dynamics of the MDP. The dynamics function $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ is a deterministic function which gives the probability of the tuple (s', r, s, a). p defines a probability distribution for each choice of s and a, that is:

$$\sum_{s' \in \mathcal{S}} \sum_{a \in \mathcal{A}} p(s', r|s, a) = 1, \ \forall s \in \mathcal{S}, \ a \in \mathcal{A} \tag{2.2}$$

This probability distribution completely characterizes the dynamics of the environment. The probability of each state S_t and reward R_t depends only on the immediately preceding state S_{t-1} and action A_{t-1}. This is a restriction on the state of the environment. The state must have information about the past agent-environment interaction. If this is true, then the state has the Markov property.

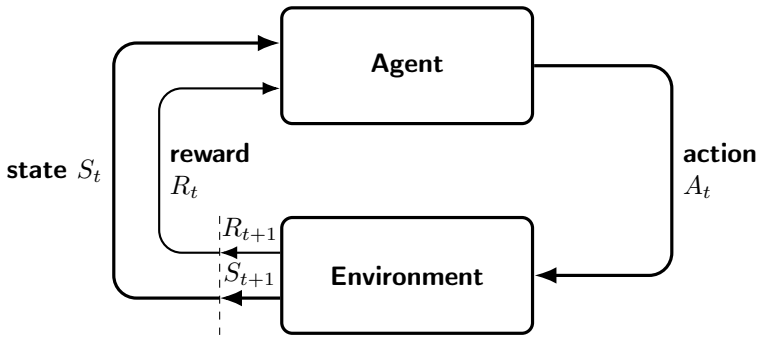


FIGURE 2.1: MDP process and agent-environment interplay

2.2 Goals and Rewards

The purpose of Reinforcement Learning is based on a signal, also called a reward, which is given by the environment to the agent. At each time step, there is a reward $R_t \in \mathbb{R}$. Agent tries to maximize the total reward it receives. This does not include only immediate reward, but the cumulative reward in the long run. This is formally called the reward hypothesis. It is clear that the learning of the agent is based on the rewards it receives from the environment. This means that based on the final goal, the agent can be trained correspondingly if one adjusts the rewards the agent receives to the goal that is aimed. Reward should not impact the agent with prior knowledge about how the goal should be achieved.

The expected return G_t is the sum of rewards of the next time step until the end time step T and is defined by:

Expected return G_t

The expected return at time step t is the total reward of all succeeding time steps

$$G_t = R_{t+1} + R_{t+2} + ... + R_T$$

This notion makes sense only in problems where the agent-environment interaction is composed of subsequencies, which are called episodes. Each episode terminates when it is in a special case, which is called a terminal state.

There is also the logic to not validate the rewards of all time steps with the same weight. At time step t, expected reward is the sum of next time steps until the terminal time step T. Each time we continue to this sequence, the added reward is weighed less and less, which practically means that we value more rewards that are closer to the time step

t and less the rewards that goes closer to the terminal time step. This gives rise to the notion of discounted return which is defined as:

Discounted return

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{k=T-t} \gamma^k R_{t+k+1}$$

2.3 Policies and Value Functions

Most of the reinforcement learning algorithms is about estimating value functions. These are functions that estimate the value of the agent being in a state or being is a state and performing an action [38]. This practically shows how good is a state (or state-action). The value is defined in terms of the expected return. The future rewards of an agent depend on the actions that the agent takes in different states. So it is natural that the value functions are defined with respect to the different ways of acting, named policies.

Policy is a mapping that maps states to probabilities of selecting each possible action. That is, if a policy π is followed, then $\pi(a|s)$ is the probability the action a is selected when environment is in state s.

State-value function under a policy π

The expected return starting in state s and following policy π

$$v_\pi(s) := \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s],\ \forall s \in \mathcal{S}$$

where $\mathbb{E}_\pi$ is the expectation value of a random variable following the policy π. In the same sense, we can define the value of taking an action on a specific state, called action-value function.

Action-value function under a policy π

The expected return starting in state s and taking action a under policy π

$$q_\pi(s,a) := \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a],\ \forall s \in \mathcal{S}$$

Value function in general satisfies recursive relations. Starting from the definition of value function, we can arrive at a recursive relation.

$$
\begin{aligned}
&\forall s \in \mathcal{S} \\
&v_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s] \\
&\quad = \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\
&\quad = \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) \Big[r + \gamma \mathbb{E}_\pi[G_{t+1} \mid S_{t+1} = s'] \Big] \\
&\quad = \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)[r + \gamma v_\pi(s')] \qquad (2.3)
\end{aligned}
$$

Last equation is called the Bellman equation for v_π. If we construct the backup diagram of the value function over the policy π, we will have the following:

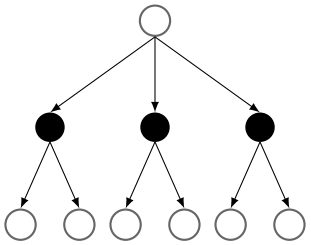


FIGURE 2.2: v_π backup diagram

A backup diagram is a visualisation of an algorithm, in which the big white circles represent states and the small black circles represent actions. These diagrams are very helpful for visualizing how an algorithm works.

Compare Policies

Value functions define a partial ordering over policies

$$\pi \geq \pi' \iff \forall s \in \mathcal{S} : v_\pi(s) \geq v_{\pi'}(s)$$

Optimal Policy

Optimal is a policy that is better or equal to all other policies and it is denoted by π_*

Optimal value functions

$$\forall s \in \mathcal{S} : v_*(s) := \max_\pi v_\pi(s)$$

$$\forall s \in \mathcal{S} : \forall a \in \mathcal{A} : q_*(s, a) := \max_\pi q_\pi(s, a)$$

Value function u will also be written with symbol V in some relations. Both symbols will be used interchangeably. There is a relation that connects the two value functions:

$$
\begin{aligned}
q_*(s) &= \max_\pi q_\pi(s,a) \\
&= \max_\pi \mathbb{E}[G_t | S_t = s, A_t = a] \\
&= \max_\pi \mathbb{E}[R_{t+1} + \gamma V_\pi(s) | S_t = s, A_t = a] \\
&= \mathbb{E}[R_{t+1} + \gamma \max_\pi V_\pi(s) | S_t = s, A_t = a] \\
&= \mathbb{E}[R_{t+1} + \gamma V_*(S_{t+1}) | S_t = s, A_t = a] \qquad (2.4)
\end{aligned}
$$

Since v_* is the value function of the optimal policy, it must satisfy the Bellman equation for $\pi = \pi_*$ (2.3).

$$
\begin{aligned}
v_*(s) &= \max_{a \in \mathcal{A}} q_*(s,a) \\
&= \max_a \mathbb{E}_{\pi_*}[G_t | S_t = s, A_t = a] \\
&= \max_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a] \\
&= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a] \\
&= \max_a \sum_{s',r} p(s', r | s, a)[r + \gamma v_*(s')] \qquad (2.5)
\end{aligned}
$$

So we have the two forms of the Bellman equation for the optimal state-value function.

Bellman optimality equation for v_*

$$v_*(s) = \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a]$$

$$v_*(s) = \max_a \sum_{s',r} p(s', r | s, a)[r + \gamma v_*(s')]$$

Similarly, for the optimal action-value function:

$$
\begin{aligned}
q_*(s,a) &= \mathbb{E}[R_{t+1} + \gamma V_*(S_{t+1}) | S_t = s, A_t = a] \\
&= \mathbb{E}[R_{t+1} + \gamma \max_{a'} q_*(s,a') | S_t = s, A_t = a] \\
&= \sum_{s',r} p(s', r | s, a)[r + \gamma \max_{a'} q_*(s,a')] \qquad (2.6)
\end{aligned}
$$

Bellman optimality equation for q_*

$$q_*(s,a) = \sum_{s',r} p(s', r | s, a)[r + \gamma \max_{a'} q_*(s,a')]$$

Chapter 3

Dynamic Programming

Dynamic programming (DP) is the set of methods that are used to compute optimal policies from Markov decision process (MDP) with the model fully given [38]. Due to its limitations in terms of computational needs and its assumptions that a perfect model for the MDP exists, its applicability is limited. Although DP is very important theoretically. Most of the reinforcement learning methods are attempts to approach and approximate the facts and results of DP, with less computational resources and without the assumption of a perfect model.

The key idea of DP is using the value functions for finding good policies. So the DP algorithms are created by transforming Bellman equations into update rules that try to improve the approximations of value functions and achieving optimal or near optimal policies.

3.1 Policy Evaluation

An important heuristic in DP and RL is to find the state-value function v_π for a given policy π. This is called policy evaluation or the prediction problem. Analyzing the relation (2.3), there is a unique v_π if $\gamma < 1$ or there is a termination from all states under this policy. If the dynamics are known, we have complete knowledge of the transition probabilities p. This means that the relation (2.3) is a system of $|\mathcal{S}|$ linear equations with $|\mathcal{S}|$ unknowns. This solution is known and straightforward, but it can be computationally expensive.

For the DP/RL problems, the iterative solutions seem more suitable. Let there be a sequence $\{v_i\}_{i=1}^{\infty}$, where $v_i : \mathcal{S} \to \mathbb{R}$. v_0 is the initial approximation of the value function. The next steps approximations are given by an update rule using the Bellman equation (2.3) for all $s \in \mathcal{S}$:

$$\begin{aligned} v_{k+1}(s) :&= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)[r + \gamma v_\pi(s')] \end{aligned} \tag{3.1}$$

The sequence $\{u_i\}$ converges to v_π as $i \to \infty$ under the same conditions that v_π exists. This procedure is called iterative policy evaluation. The algorithm replaces the old values of state s with a new value that comes from the old values of the next states of s and the immediate rewards. The algorithm requires to preserve two arrays, one for the old values of of $v_k(s)$ and one for the new ones $v_{k+1}(s)$. There is also an one-array variant which usually converges faster than the two-array version. This variant updates the values

directly, overwriting the old value. The algorithm converges only on the limit, but since reaching the limit may require a lot of resources, it may be sufficient enough to stop to a good enough approximation. Approximation accuracy is measured by the quantity:

$$accuracy = \max_{s \in \mathcal{S}} |v_{k+1}(s) - v_k(s)|$$

The pseudocode of the algorithms is the following:

Algorithm 1 Policy Evaluation

Input: a policy π
Algorithm parameter: threshold θ determining accuracy of estimation
Initialize V(s) $\forall \in \mathcal{S}$, terminal state $V(terminal) = 0$:
Loop:
  $\Delta \leftarrow 0$
  Loop $\forall \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
  Until $\Delta < \theta$

3.2 Policy Improvement

The policy evaluation algorithm is used to find the value function for a given policy π. This can be used to find a better policy. So the question here is whether a change in policy (action a) in a state s will improve the policy. Let $s \in \mathcal{S}$ be a state of the MDP and consider an action $a \in \mathcal{A}$. The action-value is given by:

$$\begin{aligned} q_\pi(s, a) &= \mathbb{E}\Big[R_{t+1} + \gamma v_\pi(s) | S_t = s, A_t = a\Big] \\ &= \sum_{s',r} p(s', r|s, a)\Big[r + \gamma v_\pi(s')\Big] \end{aligned}$$

If this action-value $q_\pi(s, a)$ is greater then the state-value $v_\pi(s)$ under policy π, which means it is better to select action a once in state s and then continue to follow policy π. If this is the case, maybe it will be better to act with an every time system is in state s in general. This is in fact true and it comes from the policy improvement theorem.

Policy improvement theorem

Let π, π' be two deterministic policies such that the following holds:

$$\forall s \in \mathcal{S} : q(s, \pi'(s)) \geq v_\pi(s) \tag{3.2}$$

This means that the policy π' must be better or at least equally good with policy π. This statement is well defined since the policies are partially ordered in the set of all possible policies. So in mathematical terms this means that:

$$\forall s \in \mathcal{S} : v_{\pi'}(s) \geq v_\pi(s) \tag{3.3}$$

For the proof of the theorem see Appendix (A).

The previous result shows that at a change in the policy at a state and an action can improve the policy. This can be generalized for all states and all possible actions. At each state the action that appears to be the most valued based on the value function $q_\pi(s, a)$ can be selected. This is the definition of the greedy policy in the current action value function.

Greedy policy with respect to action value function q

Let π be a policy and q be the action value function. The greedy policy with respect to the action value function is given by the selecting the higher valued state-action pair

$$\begin{aligned}
&\forall s \in \mathcal{S} : \\
&\pi'(s) = \operatorname*{argmax}_a (q_\pi(s, a)) \\
&\quad = \operatorname*{argmax}_a \mathbb{E}\Big[R_{t+1} + \gamma V_\pi(S_{t+1}) | S_t = s, A_t = a\Big] \qquad (3.4) \\
&\quad = \operatorname*{argmax}_a \sum_{s',r} p(s', r|s, a)[r + \gamma V_\pi(s')] \qquad (3.5)
\end{aligned}$$

where argmax gives the action at which the expected value of the state is maximum, with ties to be broken arbitrarily.

Greedy policy satisfies by its definition the policy improvement theorem, so the new greedy policy will be at least as good as the previous one. This process is called Policy improvement. When the greedy policy is exactly as good as the previous policy, that is

$$\begin{aligned}
&\forall s \in \mathcal{S} : V_{\pi'} = V_\pi \implies \\
&\qquad V_{\pi'} = \max_a \mathbb{E}\Big[R_{t+1} + \gamma V_{\pi'}(S_{t+1}) | S_t = s, A_t = a\Big]
\end{aligned}$$

The last relation is the same as the Bellman optimality equation, so this means that the previous policy π and new greedy policy π' are both the same as the optimal policy π^*. The policy improvement process can be extended into stochastic policies too [38].

3.2.1 Policy Iteration algorithm

If a policy π is improved by the policy improvement process, then a new value function can be estimated, and using this estimation an even better policy can be computed. This is a cyclic process which involves two routines, Policy Evaluation and Policy Estimation. This generates a sequence of monotonically improving policies and value functions [38].

From the results of the previous version, each new greedy policy is better that the previous policy until they are exactly the same, which means that the optimal policy is achieved. This cyclic process is called Policy Iteration Algorithm. The pseudocode of the algorithm is given bellow.

Algorithm 2 Policy Improvement Subroutine

Initialize value function V(s) and policy $\pi(s)$ for all states $s \in \mathcal{S}$, small number θ
repeat
  $V_{tmp} \leftarrow V(s)$
  for all states s **do**
    $V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s)) \Big[r + \gamma V_\pi(s') \Big]$
  end for
  $\Delta \leftarrow \max_s(V_{tmp} - V)$
until $\Delta < \theta$

Algorithm 3 Policy Evaluation Subroutine

Initialize value function V(s) and policy $\pi(s)$ for all states $s \in \mathcal{S}$, small number θ
repeat
  $\pi_{prev} \leftarrow \pi$
  for all states s **do**
    $\pi(s) \leftarrow \underset{a}{\text{argmax}} \sum_{s',r} p(s', r|s, a)[r + \gamma V_\pi(s')]$
  end for
until $\Delta < \theta$

Algorithm 4 Policy Iteration

Initialize value function V(s) and policy $\pi(s)$ for all states $s \in \mathcal{S}$
while True **do**
  Policy Evaluation
  Policy Improvement
  if $\pi = \pi_{prev}$ **then**
    Break and return V, π
  end if
end while

3.2.2 Value Iteration algorithm

Value iteration algorithm is inspired by the Bellman optimality equation:

$$v_*(s) = \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a]$$

This can be turned into an update rule which can be used to get the (sub-)optimal value function and then the greedy policy will be the corresponding (sub-)optimal policy. The update rule is given by:

$$\forall s \in \mathcal{S} : v_{k+1}(s) = \max_a \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) | S_t = s, A_t = a] \tag{3.6}$$

$$= \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V_k(s')] \tag{3.7}$$

The pseudocode of the Value Iteration algorithm is given bellow.

Algorithm 5 Value Iteration

Initialize value function V(s) for all states $s \in \mathcal{S}$ and accuracy threshold parameter $\theta > 0$
repeat
  $\Delta \leftarrow 0$
  $V_{tmp} \leftarrow V$
  for all states s **do**
    $V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
  end for
  $\Delta \leftarrow \max_s(|V_{tmp}(s) - V(s)|$
until $\Delta < \theta$

3.2.3 Generalized Policy Iteration (GPI)

Policy iteration is a combination of two processes:

- Policy evaluation
- Policy improvement

Policy evaluation is the process that finds the value function that corresponds to the current policy. Policy iteration is the process that gets a new policy acting greedily based on the value function. In the policy iteration method, the two processes act one after the other, without being necessary to act in strict turns. As long as all states are constantly updated by the two processes, the process converges to the optimal value function and the optimal policy.

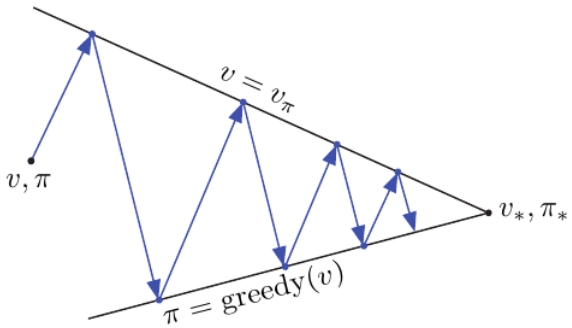


FIGURE 3.1: Generalized Policy iteration [38]

Most of the reinforcement learning methods fit in the GPI model, since they have policies and value functions that constantly improve with respect to each other. The two processes are almost orthogonal, but as they keep interacting they arrive at an equilibrium point, in which any update does not change the policy and the value functions. This means that the policy and the value function are optimal.

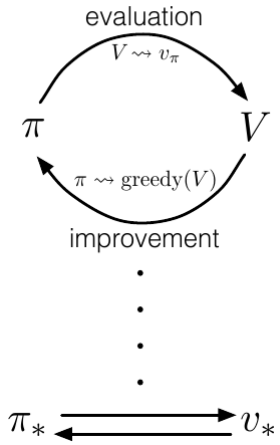


FIGURE 3.2: Generalized Policy iteration convergence [38]

3.2.4 Complexity

Dynamic programming is quite efficient in solving MDPs compared to other methods, such as direct search or linear programming. In the worst case DP methods have polynomial time complexity to find optimal policies, even though the total number of policies is k^n, where n is the number of states and k is the number of actions [38]. This means that DP methods are exponentially faster than direct search approaches. Some linear programming methods can be faster than DP methods, but they become not practical quite earlier as the number of states increases.

Chapter 4

Deep Learning

4.1 Introduction

Deep learning are methods that belong to the field of Representation learning. Representation learning methods allow systems to discover representations by getting raw data as input. Deep learning methods compose simple non-linear layers that transform the representation at one level, going from raw data to more abstract levels [23]. This decomposition allows complex function learning. The key aspect of deep learning is that these composed layers are learned from data and are not designed by humans.

4.2 Learning process

A key aspect of the learning process is the objective function, which measures the error between outputs and desired results. The model modifies its internal parameters in an attempt to minimize the objective function. The parameters are often called weights and are real numbers. The weights update is done by computing the gradient vector, which indicates in each entry the update amount for the corresponding weight. The weight vector is adjusted in the opposite direction to the gradient vector [23].

The objective function defines a loss landscape in a high dimensional space and the negative gradient vectors give the direction of the descent, which leads to lower errors closer to minimums. The most common approach that is used for the parameter optimization is the stochastic gradient descent (SDG) algorithm. The procedure is that for few examples, the objective function and the average gradient that are computed are used to adjust the weights. This process repeats for many small sets of examples until the objective function stops decreasing. The process is called stochastic because the set of examples are small and they give noisy estimates of the gradient.

Algorithm 6 Stochastic gradient descent (SGD) [16]

Initialize learning rate ϵ and optimization parameters θ
while stop criterion not true **do**
  Sample minibatch from training set $\{x_i\}_{i=1}^{m}$
  Compute gradient: $\boldsymbol{\nabla}_\theta J(\theta)$, where $J(\theta)$ is the objective function
  Update parameter vector: $\theta \leftarrow \theta - \epsilon \boldsymbol{\nabla} \theta$
end while

Learning rate is quite important parameter for the optimization process. In many cases a constant learning rate might be sufficient to navigate througth the loss landscape. But

in other cases it is better to gradually decrease the learning rate over time. A sufficient condition to guarrantee convergence of SGD is:

$$\sum_{k=1}^{\infty} \epsilon_k = \infty \text{ and } \sum_{k=1}^{\infty} e_k^2 < \infty$$

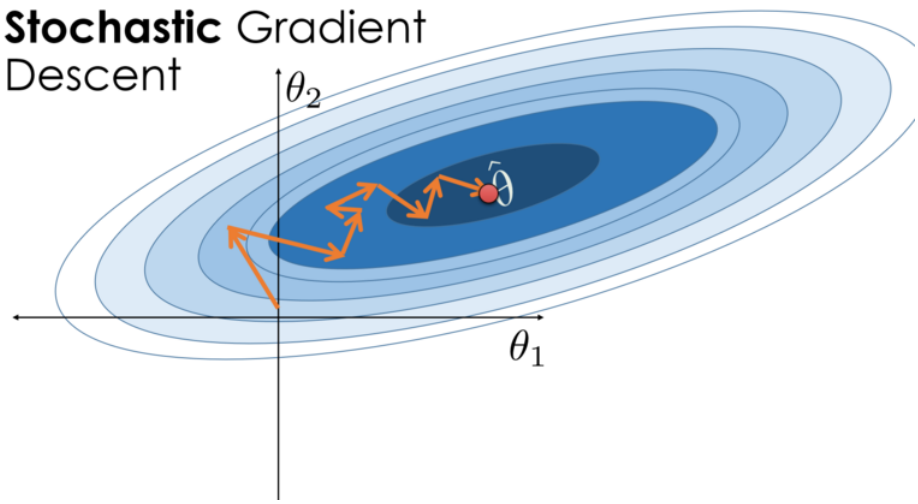


FIGURE 4.1: Stochastic Gradient Descent (SGD) in 2D

4.2.1 Backpropagation

The most common deep learning models are the feedforward neural networks. They get data vectors as input and produce output. The information moves forward through the network, and this process is called forward propagation. In the training process, the result of the forward propagation produces a scalar cost (objective function). The gradient is computed via a reversed process, called backpropagation [30]. Cost information now moves backwards through the network. Backpropagation refers only to the process that computes the gradient. After that, the stochastic gradient descent or another algorithm is employed to perform the learning [16]. The process of backpropagation can be found in more details in one of the original papers [30] or in more recent reviews and books [23, 16].

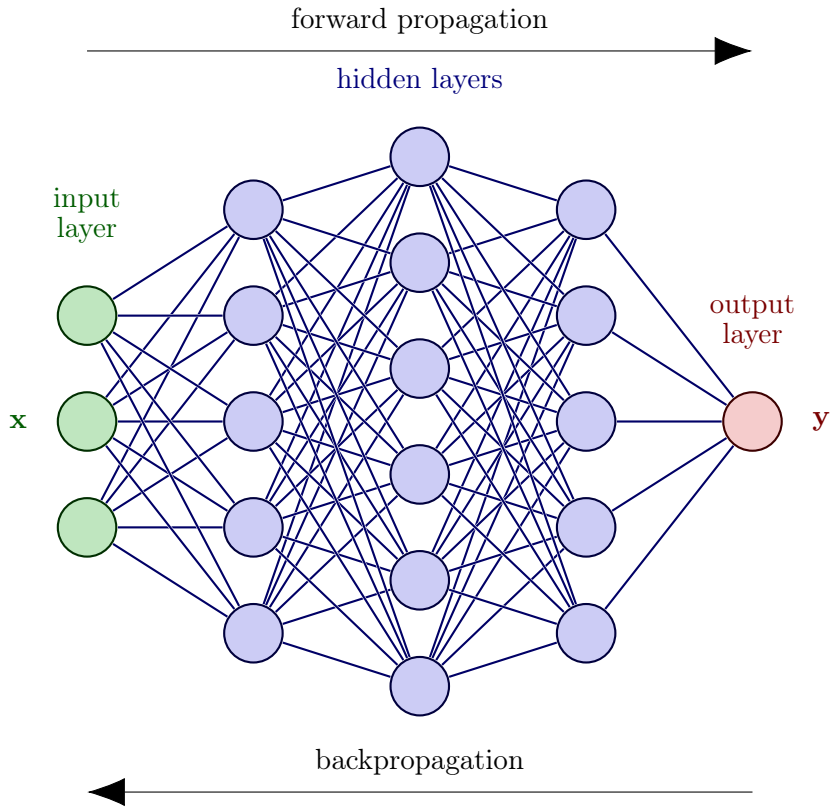


FIGURE 4.2: Deep feedforward neural network

Chapter 5

Reinforcement Learning

5.1 Monte Carlo methods

The term Monte Carlo is usually used for methods that have a random component. Monte Carlo methods require only experience and do not assume complete knowledge of the environment. These methods can work with samples from actual or simulated interaction with the environment. Learning from experience is somewhat surprising, since it does not need any prior knowledge of the environment and its dynamics. A model is required, although it is only used for generating samples from trajectories and not the complete probability distributions of all possible transitions, which is required for dynamic programming (DP).

Monte Carlo methods solve the reinforcement learning problem by averaging the returns from the samples. In case of episodic problems, the samples are divided into episodes that all terminate. Policy and value estimation is done at the end of an episode. So MC can be incremental is an episode level sense, not in a step level sense (online) [38].

5.1.1 Monte Carlo Prediction

An important step is learning the state-value function for a policy. If someone wants to do that from experience, the simplest thing to do is to average the rewards from that state. The more rewards are sampled, the better estimation of the expected value average should give. This is an important notion in MC methods. It is essential to estimate v_π(s) of a state under policy π. Each pass from state s is called a visit to s. State s may be visited several times during an episode. The first time s is visited is called first visit to s. There are two methods that can be used to estimate the value function based on a policy π:

- first visit MC method: Estimates the value function by averaging all returns following the first visit to s
- every-visit MC method: Estimates the value function by averaging all returns following all visits to s

Both methods converge to real value function as the number of (first) visits to s goes to infinity. It is important to note that MC methods estimate each state independently, as the estimate of one state does not depend on other state estimates, as in DP. This means that MC do not bootstrap. This can make MC methods quite preferable when the problem does not require the values of all states but a subset of them.

5.1.2 Monte Carlo Action Values Estimation

If MDP does not have a model, then it is useful to estimate the action values. State values are not sufficient and action values are needed for getting a policy. This is the policy evaluation problem. The previous MC methods can be easily altered to estimate the action values $q_\pi(s, a)$. The convergence conditions are similar as before. The issue is that some many state-action pairs may not be visited if the policy is deterministic. This is a general problem and it is tackled by assuring continual exploration [38], where all action-pair policies have a non-zero visiting probability.

5.1.3 Monte Carlo Control

Control is the process to approximate optimal policies. The idea is to follow the GPI process that has been analysed in the previous chapter. In the MC version of the policy iteration algorithm, one starts with an arbitrary policy and ends with the optimal policy and the optimal action-value function. MC methods can be used to find optimal policies given only sample episodes and no other knowledge of the environments dynamics [38].

5.2 Temporal Difference (TD) methods

Temporal difference learning [38] is a central and novel idea in reinforcement learning. TD learning combines ideas from Monte Carlo learning and Dynamic Programming. TD methods learn directly from experience without any model of the environment. They also update estimates based on other learned estimates, like Dynamic Programming. But without waiting for a final outcome (bootstraping).

5.2.1 TD Prediction

The main difference between TD and Monte Carlo (MC) methods is that MC must wait until the end of the episode to determine the change of the value function V(S_t), when TD methods wait only one time step before the update of the value function. At the next time step t+1, TD makes the update using the observed reward of new step T_{t+1} and the estimate for the next time step value state $V(S_{t+1})$. A simple update that can be used is the update rule:

$$V(S_t) = V(S_t) + l(G_t - V(S_t)) \tag{5.1}$$

$$V(S_t) = V(S_t) + l\Big[R_t + \gamma V(S_{t+1}) - V(S_t)\Big] \tag{5.2}$$

where $l \in (0, 1]$ is the learning rate and γ is the discount factor in the next state value. The following is the backup diagram of the above update rule.

FIGURE 5.1: TD(0) backup diagram

In the above equation, the term in the bracket is the difference between the estimated value of state S_t and the better estimation just received. This term is called TD error and it is an important notion:

$$\delta_t := R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \tag{5.3}$$

If the value V is not updated during the episode, this approach is reduced to a Monte Carlo method and the Monte Carlo error can be written with respect to the TD error using the above relation. We can add and subtract the same term $\gamma V(S_{t+1})$ and see that the relation is recursive.

$$\begin{aligned} MCE_t &= G_t - V(S_t) = R_{t+1} + \gamma G_{t+1} - V(S_t) + \gamma V(S_{t+1}) - \gamma V(S_{t+1}) \\ &= (R_{t+1} + \gamma V(S_{t+1}) - V(S_t)) + \gamma G_{t+1} - \gamma V(S_{t+1}) \\ &= \delta_t + \gamma MCE_{t+1} \end{aligned}$$

If we apply the last result in the MCE_{t+1}, the relation reads:

$$MCE_t = \delta_t + \gamma\Delta_{t+1} + \gamma^2 MCE_{t+2} \quad = \delta_t + \gamma\Delta_{t+1} + \gamma^2\delta t + 2 + \cdots + \gamma^{T-t-1}\delta_{T-1} + \gamma^{T-t}MCE_T$$

Since $MCE_T = G_T - V(S_T) = 0$ then:

$$MCE_t = \sum_{i=1}^{T-1} \gamma^{i-t}\delta_i \tag{5.4}$$

This result is quite important and holds approximately for temporal difference methods, if the step size is small. It can be generalized into important results.

5.2.2 Convergence conditions

Learning rate can be constant or can be reduced as number of episodes increments. In theory, convergence is not guaranteed for all sequences of learning rates $\{l_n\}$ [38]. From stochastic approximation theory, convergence with probability 1 is guaranteed when

$$\lim_{N\to\infty} \sum_{n=1}^{N} l_n \to \infty \text{ and } \lim_{N\to\infty} \sum_{n=1}^{N} l_n^2 < \infty \tag{5.5}$$

5.2.3 Q-Learning

Q-learning [42] is an algorithm under the umbrella of Temporal difference (TD) methods. It can be seen as an asynchronous DP method. The basic concept is the following: the agent tries an action in a state, it receives the immediate reward and goes in a new state. It evaluates and compares the immediate result with the estimate of the value of the previous state. Exploring all actions in all possible states repeatedly, it finally learns the optimal policy based on long-term discounted return. It is considered an off-policy method, since it tries to improve a policy that is different from the one that has been used to generate the data.

Agent is at state S_t at time step t. It chooses an action based on a policy from Q value function (i.e. ϵ-greedy). After the action, it receives an immediate reward. The corresponding update rule of Q-learning is the following:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + l\Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\Big] \tag{5.6}$$

where

- l: learning rate
- γ: is the discount factor
- Q(s,a): is the estimated value function of a state-action pair (s,a)

The learned Q action value function is an approximation of the optimal value function Q^*. The policy still determines the states visited and actions taken during the updates. Convergence is achieved, as long as all state-action pairs are sufficiently visited during the algorithm. This can be achieved if the policy used for the action selection A_t at state S_t is a probabilistic policy, i.e. ϵ-greedy. The pseudocode of the Q-learning algorithm is presented bellow.

Algorithm 7 Q-learning

Parameters: learning rate $l \in (0, 1]$ and $\epsilon > 0$
Initialize arbitrarily state-action value function $Q(S, a)$ $\forall S \in \mathcal{S}, a \in \mathcal{A}$

For each episode:
  Initialize state s in $\mathcal{S}$
  For each time step of episode:
    Choose action $A \in \mathcal{A}$ with policy using Q value function (i.e. ϵ-greedy)
    Take action a, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + l[R + \gamma \max_a(Q(S', a)) - Q(S, a)]$
    $S \leftarrow S'$
  until S terminal

The backup diagram gives a more intuitive picture of the Q-learning algorithm.

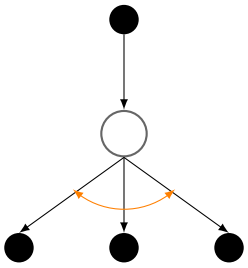


FIGURE 5.2: Q-learning backup diagram

An important condition is that the sequence of episodes must have infinite number of episodes for each starting state and action. This is quite a strong condition. In real applications this is not practical, but in most of the cases the agent is able to learn an optimal or suboptimal policy under weaker finite conditions. The proof of convergence can be found in the paper [42].

5.2.4 Sarsa

Sarsa is an on-policy TD method [38], since it tries to evaluate or improve the policy that is used for decision making. Now instead of taking the maximum value function at time t+1, the policy determines which one to choose, and based on that outcome the update rule is formed as:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + l\Big[R_{t+1} + \gamma(Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t))\Big] \tag{5.7}$$

This rule uses a tuple $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$, from which the algorithm took his name. The value of the policy is constantly evaluated and greedy policy is followed based on evaluated value function. This illustrates that Sarsa follows the general notion of Generalized Policy Iteration (GPI). The pseudocode of the Sarsa algorithm is presented bellow.

Algorithm 8 Sarsa

Parameters: learning rate $l \in (0, 1]$ and $\epsilon > 0$
Initialize arbitrarily state-action value function $Q(S, a) \forall S \in \mathcal{S}, a \in \mathcal{A}$

For each episode:
  Initialize state S in $\mathcal{S}$
  Choose action $a \in \mathcal{A}$ with policy using Q value function (i.e. ϵ-greedy)
  For each time step of episode:
    Take action a, observe R, s'
    Choose action a' using policy on Q
    $Q(S, a) \leftarrow Q(S, a) + l[R + \gamma Q(S', a') - Q(S, a)]$
    $S \leftarrow S', a \leftarrow a'$
  until S terminal

Sarsa converges with probability 1 to the optimal policy and optimal action-value function as soon as all state-action pairs are visited an infinite number of times.

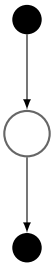


FIGURE 5.3: Sarsa backup diagram

5.2.5 Expected Sarsa

Another algorithm that can be considered as a variation of Q-learning is the Expected Sarsa algorithm [38]. The update rule in this case, instead of the maximum over next state-action value, uses the expected value of the next state-action pair, taking into account how likely each action is to be selected under the current policy. The update rule is given by:

$$\begin{aligned} Q(S_t, A_t) &\leftarrow Q(S_t, A_t) + l\Big[R_{t+1} + \gamma \mathbb{E}_\pi \Big[Q(S_{t+1}, A_{t+1}) | S_{t+1}\Big] - Q(S_t, A_t)\Big] \\ &\leftarrow Q(S_t, A_t) + l\Big[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t)\Big] \end{aligned} \tag{5.8}$$

Expected Sarsa seems to eliminate the variance of Sarsa due to the random selection of action A_{t+1}. It is expected to perform better than Sarsa given the same amount of learning experience.

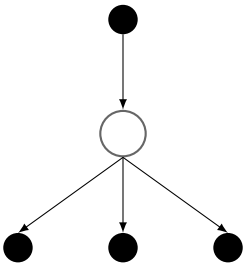


FIGURE 5.4: Expected Sarsa backup diagram

5.2.6 Double Q-learning

Q-learning algorithm can experience poor performance in some stochastic environments. The use of max operator in the update rule (5.6) can cause large overestimations of the action values. To address this issue, authors of [19] propose an alternative double estimator method to find the estimate for the maximum value. This sometimes underestimate rather than overestimates the maximum expected value. This algorithm is called Double Q-learning [19].

The algorithm has two action-value functions Q_1 and Q_2. Each Q function is updated from the other Q function for the next state. The selected action after initial action has been taken, is determined based on one of the Q functions. However the update of that Q function is done by using the other Q function as the estimator. This is demonstrated in the pseudocode bellow. At the end, the extracted policy is the greedy policy with respect to the average of the two Q functions.

Algorithm 9 Double Q-learning

Parameters: learning rate $l \in (0, 1]$ and $\epsilon > 0$
Initialize arbitrarily state-action value functions $Q_1(S, a)$ and $Q_2(S, a)$ $\forall S \in \mathcal{S}, a \in \mathcal{A}$

For each episode:
  Initialize state s in $\mathcal{S}$
  For each time step of episode:
  Choose action $A \in \mathcal{A}$ with policy using the policy ϵ-greedy in $Q_1 + Q_2$
    Take action a, observe R, S'
    With 50% probability:
      $Q_1(S, A) \leftarrow Q_1(S, A) + l[R + \gamma Q_2(S', \underset{a}{\text{argmax}} Q_1(S', a)) - Q_1(S, A)]$
    else:
      $Q_1(S, A) \leftarrow Q_1(S, A) + l[R + \gamma Q_2(S', \underset{a}{\text{argmax}} Q_1(S', a)) - Q_1(S, A)]$
    $S \leftarrow S'$
  until S terminal

5.3 TD - Approximation methods

5.3.1 Value function approximation

Tabular methods use tables to represent the value functions. As number of states and number of actions increase, the complexity of tabular methods increases and in many cases problems become intractable. So in such cases approximation methods may be used to represent the value functions. This representation is a parametrized form with weight vector $\mathbf{w}$ [38]. The value function can be written as

$$V(s, \mathbf{w}) \approx V_\pi(s) \tag{5.9}$$

which means that the function is approximated by a model with parameter vector $\mathbf{w}$. The model can be linear or non-linear. In general the model can be an Artificial Neural Network (ANN), which can be linear or non-linear based on the the activation functions that are used in the neuron activation. It is a known fact that in reinforcement learning environments, which are not static and change constantly, linear models may experience better convergence guarantees [39]. But with the recent advance of deep learning, the complexity of these models with hidden layers are able to approximate functions with arbitrary precision [23]. In general the number of states are much more than the number of parameters, so an update in an arbitrary state can be generalized into other states too. This generalization is what makes ANN powerful and suitable for approximating big state spaces models. Maybe this can be helpful for partially observable environments, in which the full state is not accessible to the interacting agent.

In all encountered methods so far, there is always an update rule that is used to update the value function of an arbitrary state of the MDP. In the tabular case this is done one by one for every state, but as already stated with machine learning models, this update can be generalized to other states as well. This model of learning that is used in reinforcement learning methods is learning by input output examples, and particularly the outputs are the numeric values of the input states. This type of learning is recognised as Supervised learning. So it is possible to use any kind of Supervised learning tool, such as multivariate regression, decision trees and artificial neural networks. Artificial neural networks can be

regarded as generalisations of linear and non-linear models and they will be employed to solve problems of Quantum Control. One of the most popular example of algorithms that is quite successful and uses value function approximation with a deep neural network is the Deep Q-Network algorithm.

5.3.2 Loss function and Stochastic Gradient Descent

A very popular and most widely used learning method for function approximation is stochastic gradient descent (SGD). The goal is to optimize the parameters (weights) of the model, **w** with respect to a loss function. The loss function which is quite appropriate in the context of reinforcement learning is the Mean Squared Value Error ($\overline{VE}$) [38]. The square root of $\overline{VE}$ gives a measure of how far is the approximate value function from the true value function, summing over all available states.

$$\overline{VE}(w) := \sum_{s\in\mathcal{S}} \mu(s)\Big[V^*(s) - V(s,w)\Big]^2 \tag{5.10}$$

where μ is a distribution ($\mu(s) > 0$, $\sum_s \mu(s) = 1$) which represents how important are states.

The best case scenario is to find a global minimum, in which $\overline{VE}$ is the smaller for all possible weight vectors, i.e.

$$\overline{VE}(w_{gm}) \leq \overline{VE}(w),\ \forall w \in R^{dim(w)} \tag{5.11}$$

In most of the cases, for complex problems, convergence to global minimum is not always possible, in most of the cases the optimization procedure converges to a local minimum, in which $\overline{VE}$ is smaller for all weight vectors in a neighborhood of w_{lm}.

Weight parameters are updated in each time step. In an arbitrary time step t, the weight vector is w_t. For simplicity, the μ distribution over the importance of states is the same for all states. So all states are equally important. After each experience generated (example), the weights are updated by a small amount in the direction where the error is reduced.

$$w_{t+1} = w_t - \frac{1}{2} l \boldsymbol{\nabla}\Big[V_\pi(s) - V(S_t, w_t)\Big]^2 \tag{5.12}$$

$$= w_t + l\Big[V_\pi(s) - V(S_t, w_t)\Big]\boldsymbol{\nabla} V(S_t, w_t) \tag{5.13}$$

where l is the learning parameter and $\boldsymbol{\nabla} V(S_t, w_t)$ is the gradient of the value function with respect to vector w.

$$\boldsymbol{\nabla} V(S_t, w_t) = \begin{pmatrix} \frac{\partial V(S_t,w_t)}{\partial w_1} \\ \vdots \\ \frac{\partial V(S_t,w_t)}{\partial w_{dims(w)}} \end{pmatrix} \tag{5.14}$$

In cases where the target value $S_t \mapsto U_t$ is not the true target value V_π of the update rule, but an approximation or a noisy version of it or for any reason it is unknown, the exact update of the weights is not possible, since the real value V_π is not known. In this case,

substituting real value with the approximate version U_t, the approximate weight update rule arises:

$$w_{t+1} = w_t + \alpha\Big[U_t - V(S_t, w_t)\Big]\boldsymbol{\nabla} V(S_t, w_t) \tag{5.15}$$

If U_t is unbiased and the expected value gives the true value [38]

$$\mathbb{E}\Big[U_t | S_t = s\Big] = V_\pi(S_t),\ \forall t \tag{5.16}$$

then optimization converges to a local optimum [38].

5.3.3 Deep Q-Network (DQN)

Deep Q-Network [26] is a reinforcement learning algorithm which combines reinforcement learning, It is a temporal difference method that resembles Q-learning, with deep neural networks. In the current context, a complete fully connected ANN is used for approximating the action-value function Q.

As it is already stated, non-linear approximators such as neural networks with non-linear activation functions can result instability or even divergence of the model. DQN algorithm tries to address this problem with two key ideas:

- Use of experience replay which randomizes the data and removes correlations between sequential states and
- Periodically update of the Q action-value function, which reduces the correlations with the target.

The agent uses a parametrized action-value function $Q(s, a, \theta_i)$, where θ_i is the parameter vector at learning iteration i. Experience replay is manifested by storing agent experiences $e_t = (s_t, a_t, r_t, s_{t+1})$ at each time step t in a dataset $D = \{e_1, ..., e_t\}$. The Q-learning updates are done on random samples from these experiences. The Loss function for the Q-learning update is the following:

$$L_i(\theta_i) = \mathbb{E}_{e_t}\Big[(r + \gamma \max_{a'} Q(s', a', \theta_i^-) - Q(s, a, \theta_i)^2)\Big] \tag{5.17}$$

where γ is the discount factor, θ_i^- are the parameters of the Q-network that are used to get the target at iteration i and θ_i are the parameters of the Q-Network at iteration i. Parameters θ_i^- are updated periodically with the values of the parameters θ_i and are fixed in between. This practically means that the parameters θ_i are updated in each step of the training but they are not directly used in the estimated value inside the max operation. Only after some iterations these parameters are used to update the periodically steady parameters θ_i^- of the target network. This process is achieved by preserving two instances of the same neural network, one with parameters θ_i and another one with parameters θ_i^-. The first network is constantly updated and the second one is updated periodically using the parameter values of the first. In this way the correlations between subsequent experiences are reduced. More details on this procedure are given in Algorithm 10. Concerning the architecture of the network, the input is the state and the action of the current time step, the output neuron values correspond to the action-values of each possible next action. It

is trivial to think that this algorithm works without any further modification with discrete state and action spaces. The advantage of this architecture is that for each state-action pair, the action-values of each possible next action are estimated at once. The pseudocode is given bellow.

Algorithm 10 Deep Q-Network with experience replay

Initialize replay buffer/memory D
Initialize ANN with random weights θ
Initialize target ANN with random weights θ' and update rate C
for each episode: **do**
  Initialize state s in $\mathcal{S}$
  for each time step of episode: **do**
    Choose action $a \in \mathcal{A}$ based on the ϵ-greedy policy in the parametrized action value $Q(s, \cdot, \theta)$
    Take action a, observe r, s'
    Store experience in memory D
    Select a random minibatch experience from memory D
    Get the return based on next maximum valued action:

$$y = \begin{cases} r & \text{if } s' \text{ is terminal state} \\ r + \max_{a'} Q(s', a', \theta^-) & else \end{cases}$$

    Perform gradient descent on the Loss function $L_i(\theta_i)$ (5.17)
    Update parameters so that $\theta^- = \theta$ every C steps
  end for
end for

Figure (5.5) shows the architecture of the Deep Q-Network, with inputs the state-action pairs and output the Q action-values of all the possible next actions. In environments with continuous state space, the density matrix elements are also added as inputs in the ANN as shown in architecture 2 (5.6). This makes the model more physics informed, as the quantum system state is taken into account in the MDP state.

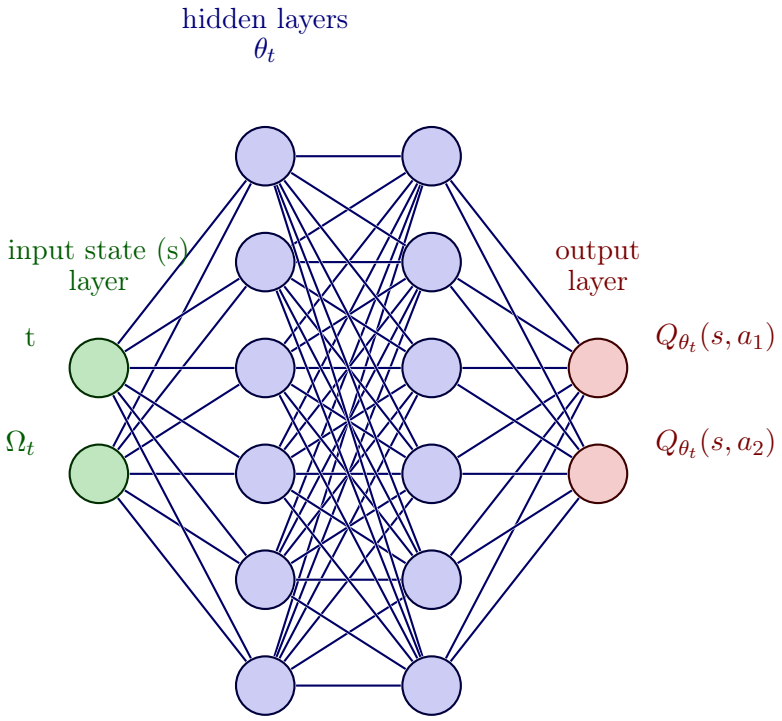


FIGURE 5.5: DQN-architecture 1. Input layer consists of the state $s = (t, \Omega_t)$ and output layer consists of the values of all actions

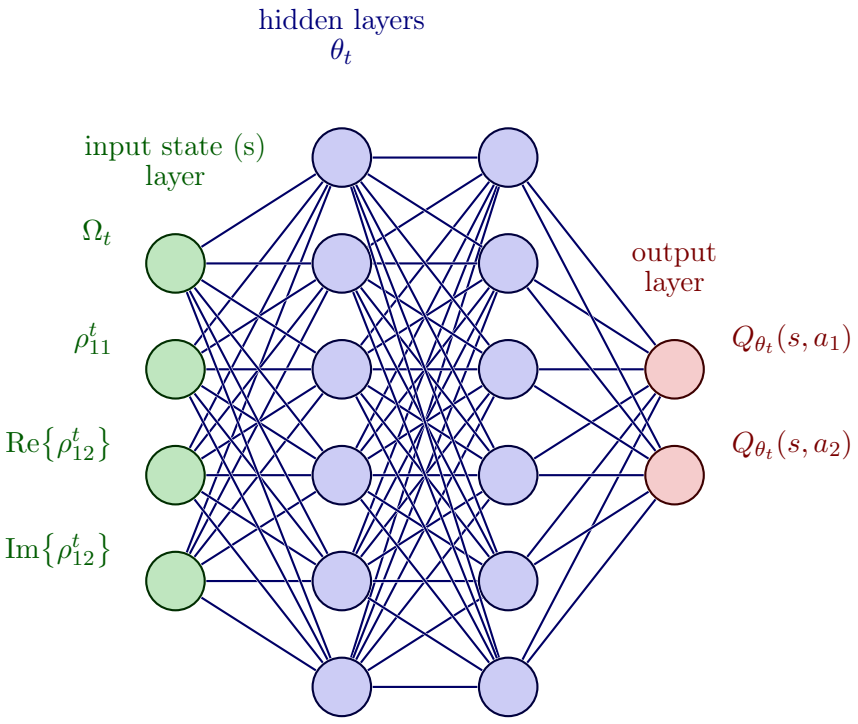


FIGURE 5.6: DQN-architecture 2. Input layer consists of the state $s = (\Omega_t, \rho_{11}, \mathrm{Re}\{\rho_{12}\}, \mathrm{Im}\{\rho_{12}\})$ and output layer consists of the values of all actions

5.3.4 Double Deep Q-Network

Double DQN [40] is a generalization of the Double Q-learning algorithm that involves function approximation, such as deep neural networks. Double Q-learning was proposed to handle with the overestimations that Q-learning algorithm by splitting the max operation in the update rule into action selection and action evaluation. Target network in the DQN architecture can be used as the second value function used in Double Q-learning. The algorithm uses the target network for value estimation while using the online network for greedy policy evaluation. The return in the update rule of Double DQN algorithm is defined by:

$$y_{DoubleDQN} = R_{t+1} + Q(S_{t+1}, \operatorname*{argmax}_a Q(s_{t+1}, a, \theta_t), \theta_t^-) \tag{5.18}$$

The parameters of the second network have been replaced by the parameters of the target network and the periodic update remains the same.

5.4 Policy gradient methods

All previous methods are based in the (action-)value functions and due to this are called action-value methods. They learn the value function of the states and the actions and they extract the policies by being greedy with respect to the value function. They totally depend on the value functions. There is another kind of methods, which do not depend on the action values at all. They are called policy gradient methods, and they use a parameterized policy that improves to select the best actions, based on the state, the system is [38]. A value function can be used to learn the policy parameter, but is is not required for action selection.

5.4.1 Policy approximation

Policy parameterization can be done with several methods and use different models, but with one necessary prerequisite: policy $\pi(a|s,\theta)$ must be differentiable with respect to the parameters θ. Such as in action-value methods, ϵ-greedy collect policy was used to generate samples, in policy gradients methods exploration should also be ensured. This can be achieved if the policy never becomes deterministic, that is $\forall s, a, \theta : \pi(a|s,\theta) \in (0,1)$.

Parameterized policy

Let π be a policy and θ be the parameters of the policy. The probability to select action $a \in \mathcal{A}$ when being in state $s \in \mathcal{S}$ and with current parameters θ is:

$$\pi(a|s,\theta) = Pr\{A_t = a | S_t = s, \theta_t = \theta\} \tag{5.19}$$

The methods learn the optimal policy parameters by using approximate gradient ascent in a performance measure $J(\theta)$.

Stochastic gradient ascent on performance measure J

$$\theta_{t+1} = \theta_t + l\nabla J(\theta_t) \tag{5.20}$$

where l is the learning rate, θ are the previous step parameters and θ_{t+1} are the current time step parameters.

5.4.1.1 Discrete action space parameterization

If the state space is not too large, then policy is commonly parameterized according to an exponential soft-max distribution:

$$\pi(a|s,\theta) := \frac{e^{h(s,a,\theta)}}{\sum_b e^{h(s,b,\theta)}} \tag{5.21}$$

where $h(s,a,\theta)$ is the parameterized numerical preferences for each action-state pair and b is an index that represents actions. This is in fact a probability distribution, since the sum of the action probabilities on each state sum up to one.

5.4.1.2 Continuous action space parameterization

Policy gradient methods are not only applicable on discrete action spaces. With the appropriate parameterization, they can also deal with very large or even infinite actions spaces [38]. In this case models learn the statistics of the probability distribution and actions are sampled by this distribution. One common choice is to sample actions from a normal Gaussian distribution. The probability density function is given by:

$$d(a) := \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(a-\mu)^2}{2\sigma^2}} \tag{5.22}$$

where μ is the mean and σ is the standard deviation of the normal distribution. The probability to select an action from a subset of the complete action space is given by the following integral:

$$Pr\{a \in \mathcal{X}\} := \int_{\mathcal{X}} d(a) d\mu_a \tag{5.23}$$

where $\mathcal{X}$ is the action subset and $d\mu_a$ is the probability measure in the action distribution space. The integral can be thought as the more general Lebesgue integral, since an action can be arbitrary and not only on $\mathbb{R}$. Using this probability distribution, policy can be thought as the probability density function, in which the mean and the standard deviation can be approximated by function approximators depending on states.

$$\pi(a|s,\theta) := \frac{1}{\sigma(s,\theta)\sqrt{2\pi}} e^{-\frac{(a-\mu(s,\theta))^2}{2\sigma(s,\theta)^2}}$$

5.4.2 The Policy Gradient Theorem

Policy gradient methods offer advantages over ϵ-greedy actions selection. They can approach a deterministic policy in contrast to ϵ-greedy action selection. Other than this, there is also an important theoretical advantage. In continuous policy parameterization action probabilities change smoothly as a function of the learned parameter, in contrast with the ϵ-greedy selection, in which the action probabilities may change dramatically for an arbitrary small change in the action values [38]. Due to this, stronger convergence guarantees can be given for policy gradient methods. The continuity of the policy dependence in the parameters allows policy gradient methods to approximate gradient ascent.

The challenge with approximating the policy is that performance depends not only on action selections, but also on distributions of states. Both depend on the parameters of the policy. When the state of the MDP is known, the effect of the policy parameters on the actions and on rewards is trivial to find. But their effect on the distribution of states is in general unknown. So the question here is how to estimate the performance gradient with respect to the policy parameter, since the one of the two main effects are unknown. The answer can be given by the the Policy Gradient Theorem, which states that the gradient of the performance metric does not depend on the distribution of states.

Policy Gradient theorem

Let $J(\theta)$ be the performance metric which gives the performance of the parameterization. The gradient of the performance metric is proportional to the following relation:

$$\nabla J(\theta) \sim \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\theta) \tag{5.24}$$

where θ are the policy parameters, π the policy and μ the distribution of the importance of states under policy π ($\mu(s) > 0$, $\sum_s \mu(s) = 1$).

The proof of the policy can be found in [38].

5.4.3 REINFORCE algorithm

REINFORCE [43] is a policy gradient algorithm which is also named as Monte Carlo Policy Gradient, for reasons that will become clear later in this section. The same procedure as in [38] is used to derive the stochastic gradient ascent rule for the policy parameter θ updates. The starting point of course is the Policy Gradient Theorem. For the derivation, discount factor γ is set to 1 and it will be added later in the results in the pseudocode of the algorithm.

$$\nabla J(\theta) \sim \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\theta)$$

The right hand side of the relation is a sum over the state appearance probabilities under policy π. So it can be written as the expectation value sampling under the policy π:

$$\nabla J(\theta) \sim \mathbb{E}_\pi \Big[\sum_a q_\pi(S_t,a) \nabla \pi(a|S_t,\theta) \Big] \tag{5.25}$$

A good stochastic gradient ascent algorithm would be to use the above relation for the update rule:

$$\theta_{t+1} \leftarrow \theta_t + l \sum_a \hat{q}(S_t, a, \mathbf{w}) \boldsymbol{\nabla} \pi(a|S_t, \theta) \tag{5.26}$$

where $\hat{q}$ is an approximation of the action-value function under policy π, $\mathbf{w}$ are the parameters of that approximation and l the learning rate. This rule is called in all actions. REINFORCE algorithm uses a similar update rule, but it does not involve all actions, only the action that is taken at time step t. From (5.25) we can multiply and divide with the same term, which will be the probability to select action a:

$$\boldsymbol{\nabla} J(\theta) \sim \mathbb{E}_\pi \Big[\sum_a \pi(a|S_t, \theta) q_\pi(S_t, a) \frac{\boldsymbol{\nabla} \pi(a|S_t, \theta)}{\pi(a|S_t, \theta)} \Big]$$

Now that there is a sum of probabilities, sums and probability term can be replaced by the expectation under policy π and replace a with the action at time t A_t:

$$\boldsymbol{\nabla} J(\theta) \sim \mathbb{E}_\pi \Big[q_\pi(S_t, A_t) \frac{\boldsymbol{\nabla} \pi(A_t|S_t, \theta)}{\pi(A_t|S_t, \theta)} \Big]$$

Using the following relation that connects expected return and action value function under policy π:

$$\mathbb{E}_\pi \Big[G_t | S_t, A_t \Big] = q_\pi(S_t, A_t)$$

the gradient relation reads:

$$\boldsymbol{\nabla} J(\theta) \sim \mathbb{E}_\pi \Big[G_t \frac{\boldsymbol{\nabla} \pi(A_t|S_t, \theta)}{\pi(A_t|S_t, \theta)} \Big]$$

If we use also the relation: $\boldsymbol{\nabla} lnx = \frac{\boldsymbol{\nabla} x}{x}$ then we can rewrite the gradient relation:

$$\boldsymbol{\nabla} J(\theta) \sim \mathbb{E}_\pi \Big[G_t \boldsymbol{\nabla} \ln \pi(A_t|S_t, \theta) \Big] \tag{5.27}$$

From the last result, one can extract the stochastic gradient ascent rule that is used in REINFORCE algorithm:

$$\theta_{t+1} \leftarrow \theta_t + \alpha G_t \boldsymbol{\nabla} \ln \pi(A_t|S_t, \theta) \tag{5.28}$$

The parameter vector is updated to the direction in the parameter space which increases more the probability of repeating action A_t when state S_t is revisited. The parameter vector is increased proportional to the return G_t and inversely proportional to the probability that action A_t is selected. It is intuitive to go to the direction that increases the expected return, which is the goal of all the reinforcement learning algorithms. On the other hand it is also good that this direction is inversely proportional to the probability to select the action A_t. This means that most selected actions are not preferred over other actions. REINFORCE

algorithm has good theoretical convergence properties. However, being a Monte Carlo method, experiences high variance and slow learning [38].

Since REINFORCE algorithm uses the total return G_t of an episode to proceed to the parameter updates, it can be regarded as a Monte Carlo method, which is why it has the second name Monte Carlo Policy Gradient. The pseudocode of the algorithm is presented bellow.

Algorithm 11 REINFORCE algorithm

Initialize parameters θ of the policy $\pi(a|s,\theta)$ and learning parameter α
for each episode: **do**
Generate a full episode $S_0, A_0.R_1, ..., S_{T-1}, A_{T-1}, R_T$
for each time step of episode t = 0, 1, ..., T-1 **do**

$$G_t \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$$

$$\theta_t \leftarrow \theta_t + \alpha \gamma^t G_t \nabla \ln \pi(A_t|S_t,\theta_t)$$

end for
end for

In case the actions space is discrete, the neural network will have an output layer with number of neuron the same as the number of actions. This final layer gives the discrete probability distribution, the probabilities to select all actions based on the input state.

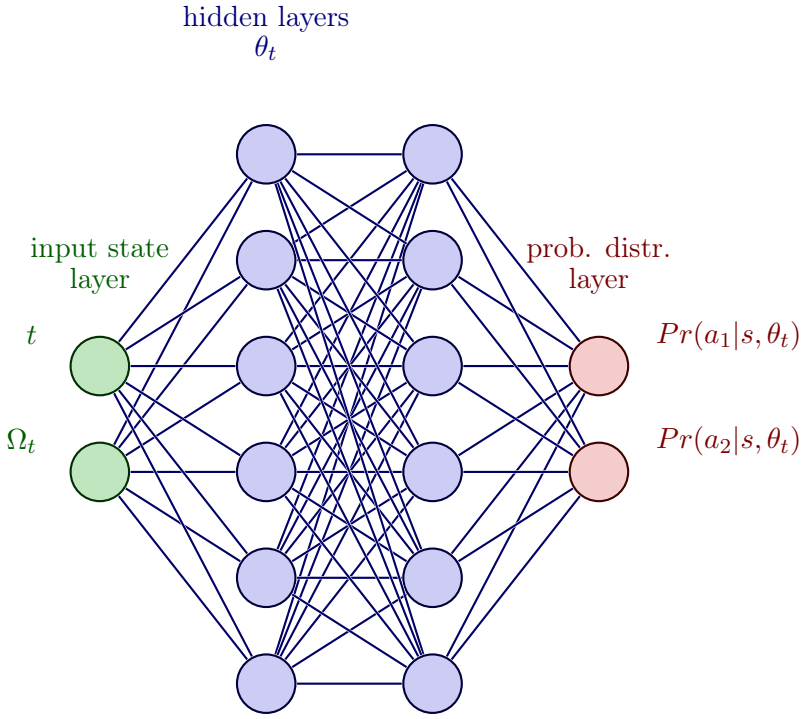


FIGURE 5.7: REINFORCE - Parameterized policy ANN - Discrete action space $\mathcal{A} = \{a_1, ..., a_n\}$ and state space $\mathcal{S} = \{t, \Omega_t\}$. Input layer consists of the states $s \in \mathcal{S}$ and the output layer consists of the discrete probability distribution for action selection $\{Pr(a_1), ..., Pr(a_2)\}$.

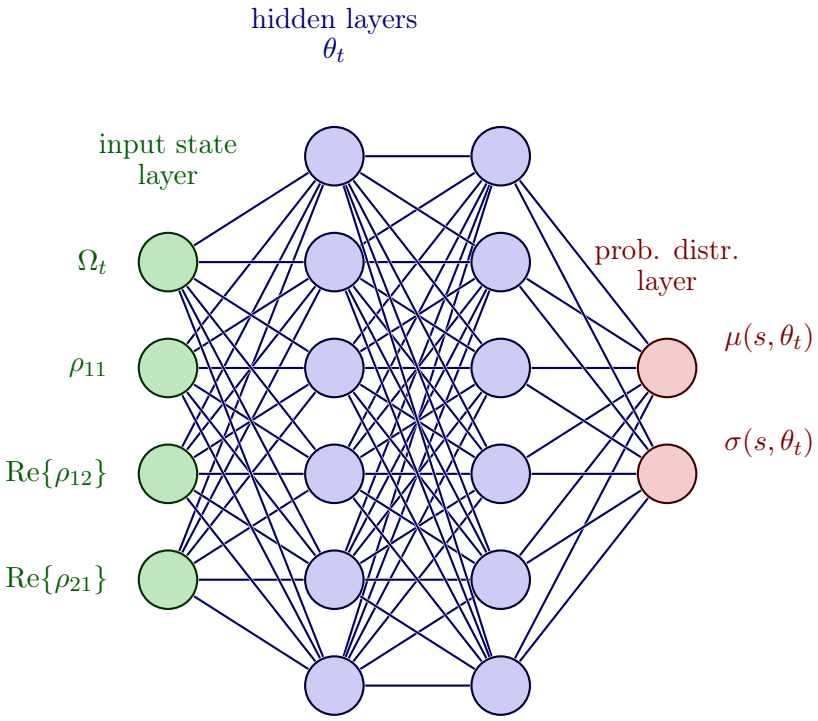


FIGURE 5.8: REINFORCE - Parameterized policy ANN - Continuous action space $\mathcal{A} = \{a_t | t \in N\mathbb{N}\}$ and state space $\mathcal{S} = \{\Omega_t, \rho_{11}, \mathrm{Re}\{\rho 12\}, \mathrm{Im}\{\rho 12\}\}$. Input layer consists of the states $s \in \mathcal{S}$ and the output layer consists of the parameters of the normal probability distribution.

The output of the continuous action policy Neural Network are the parameters of the normal distribution:

$$\pi_\theta(a|s) = \frac{1}{\sigma_\theta(s)\sqrt{2\pi}} e^{-\frac{(a-\mu_\theta(s))^2}{2\sigma_\theta^2(s)}}$$

5.4.4 Introducing Baseline in REINFORCE algorithm

The action value function can be compared to a baseline function, which can be arbitrary, with the restriction that it cannot depend to the action a. Thus the policy gradient theorem can be generalized as follows [38]:

$$\boldsymbol{\nabla} J(\theta) \sim \sum_s \mu(s) \sum_a \Big(q_\pi(s,a) - b(s) \Big) \boldsymbol{\nabla} \pi(a|s,\theta) \tag{5.29}$$

In case the baseline function is zero in all its domain, then the gradient reduces to the policy gradient theorem relation. If one follows the same procedure from the previous section, starting from the relation with the baseline (5.29), will end up to the following result for the new update rule of the REINFORCE algorithm with baseline:

$$\theta_{t+1} \leftarrow \theta_t + \alpha \Big(G_t - b(S_t) \Big) \boldsymbol{\nabla} \ln \pi(A_t|S_t,\theta) \tag{5.30}$$

The reason of the introduction of the baseline is that it can reduce the variance of the algorithm and with that increase the learning speed.

Algorithm 12 REINFORCE with Baseline algorithm

Initialize parameters θ of the policy $\pi(a|s,\theta)$ and learning parameter α^{θ}
Initialize parameters w of the state-value function $V(s,w)$ and learning parameter α^{w}
for each episode: **do**
Generate a full episode $S_0, A_0.R_1, ..., S_{T-1}, A_{T-1}, R_T$
for each time step of episode t = 0, 1, ..., T-1 **do**

$$G_t \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$$
$$\delta \leftarrow G - V(S_t, w)$$
$$\theta_t \leftarrow \theta_t + \alpha^{\theta} \gamma^t \nabla \ln \pi(A_t|S_t, \theta_t)$$
$$w_t \leftarrow \theta_t + \alpha^{w} \delta \nabla V(S_t, w_t)$$

end for
end for

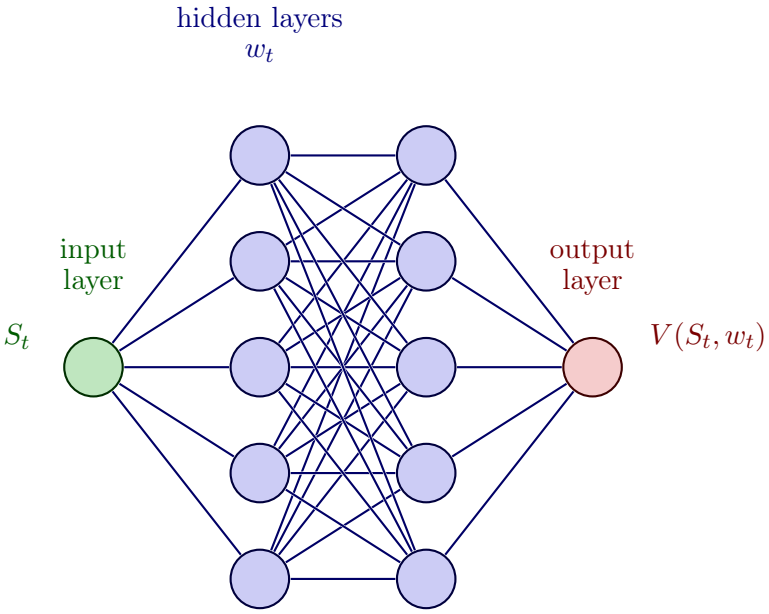


FIGURE 5.9: REINFORCE - Baseline Value NN. Input layer consists of the MDP state $s \in \mathcal{S}$ and the output layer consists value of the input state s.

5.5 Actor-critic methods

The reinforcement learning methods analyzed so far fall into two categories [21, 17]:

1. Valued based (Critic-only) methods. They use tabular or approximated versions of the value functions. They may succeed to give a good approximation of the value function, but they do not guarantee near-optimal policies.

2. Policy based (Actor-only) methods. They use parameterized policies and the basic disadvantages of such methods is the large variance of the gradient estimators and that due to changes in policy there is no accumulation of previous information.

There is also a third category of methods, which is called actor-critic. They try to combine the strong points of both previous categories. Critic approximates the value function and it is used to update the parameters of the parameterized policy. Actor critic methods are based on the important observation that since the number of actor parameters are less than the number of states, the critic does not need to approximate the exact value function. Parameterized actors computes continuous actions without need of optimization on a value function, while critic offers to actor knowledge about the performance with low variance, a combination which speeds up learning.

5.5.1 Deterministic Policy Gradient

This type of methods consider deterministic policies $a = \mu_\theta(s)$. The big question here is whether these methods can follow the same approach as for stochastic policies. It was believed that deterministic policy gradient did not exist for model free MDPs. Although, authors in [34] have showed that deterministic policy gradient does exist and in fact it is proportional to the gradient of the action-value function. Deterministic policy gradient is the limit case of the stochastic policy gradient as the policy variance goes to zero.

Deterministic policy gradient algorithms need to explore in a satisfactory way, so the algorithm proposed in [34] is an off-policy algorithm, which chooses actions based in a stochastic behaviour policy, but it learns about a deterministic target policy which exploits the deterministic policy gradient. The algorithm estimates the action value function with a differentiable function approximator and updates the parameters of the policy in the direction of the action value gradient.

5.5.1.1 Off-Policy Actor-Critic

In the off-policy setting, the performance objective is modified to be the value function of the target policy, averaged over the state distribution of the behaviour policy:

$$\begin{aligned} J_\beta(\pi_\theta) &= \int_{\mathcal{S}} \rho^\beta(s) V^\pi(s) ds \\ &= \int_{\mathcal{S}} \int_{\mathcal{A}} \rho^\beta(s) \pi_\theta(a|s) Q^\pi(s,a) da ds \end{aligned} \tag{5.31}$$

After differentiating the performance metric and applying an approximation, the off-policy policy gradient is given by:

$$\nabla_\theta J_\beta(\pi_\theta) = \int_{\mathcal{S}} \int_{\mathcal{A}} \rho^\beta(s) \nabla_\theta \pi_\theta(a|s) Q^\pi(s,a) da ds \tag{5.32}$$

$$= \mathbb{E}_{s\sim\rho^\beta, a\sim\beta}\Big[\frac{\pi_\theta(a|s)}{\beta_\theta(a|s)} \nabla_\theta \log \pi_\theta(a|s) Q^\pi(s,a)\Big] \tag{5.33}$$

where $\beta(a|s) \neq \pi(a|s), \forall s \in \mathcal{S}$ is a distinct behaviour policy used for sampling trajectories. During the above approximation [10], the term that depends on the action value gradient is emitted. Off-Policy Actor-Critic algorithm proposed in [10] uses a behaviour policy β to generate the samples. The critic agent estimates a state-value function off-policy

from the generated trajectories. The actor updates the policy parameters off-policy from the generated trajectories with stochastic gradient ascent. In the relation above (5.32), the $Q\pi$ is replaced by the temporal difference error δ_t. Both actor and critic use the sampling ratio $\frac{\pi_\theta(a|s)}{\beta_\theta(s|a)}$ to compensate with the fact that actions are selected be π instead of β.

5.5.1.2 Deterministic Policy Gradient Theorem

The main result in [34] is a deterministic policy gradient theorem, analogous to the stochastic policy gradient theorem. Most of model-free reinforcement learning algorithms are based on generalised policy iteration (GPI). Although the subroutine of policy improvement is problematic in continuous action spaces since it requires global maximization at every step. A more simple alternative would be to follow the direction of gradient of the action value Q instead of maximizing it. Because every state may follow a different direction, the update can be averaged by taking the expectation value over the state distribution ρ^μ,

$$\theta_{t+1} = \theta_t + \alpha \mathbb{E}_{s\sim\rho^{\mu_t}}\Big[\boldsymbol{\nabla}_\theta Q^{\mu_t}(s, \mu_\theta(s))\Big] \tag{5.34}$$

Applying the chain rule, the relation now reads:

$$\theta_{t+1} = \theta_t + \alpha \mathbb{E}_{s\sim\rho^{\mu_t}}\Big[\boldsymbol{\nabla}_\theta \mu_\theta(s))\boldsymbol{\nabla}_\theta Q^{\mu_t}(s, a)|_{a=\mu_\theta(s)}\Big] \tag{5.35}$$

This result does not guaranty policy improvement, since the state distribution changes as the policy changes. This is what the deterministic policy gradient theorem comes to resolve.

Deterministic Policy Gradient theorem

Let $\mu_\theta : \mathcal{S} \to \mathcal{A}$ be a deterministic policy with parameters $\theta \in R^n$, $\rho^\mu(s)$ be the discounted state distribution. The performance objective is given by:

$$\begin{aligned} J(\mu(s)) &= \int_{\mathcal{S}} \rho^\mu(s) r(s, \mu_\theta(s)) ds \\ &= \mathbb{E}_{s\sim\rho^\mu}\Big[r(s, \mu_\theta(s))\Big] \end{aligned} \tag{5.36}$$

Assuming that gradients of μ_θ, $Q^\mu(s, a)$ and deterministic policy gradient exist, then:

$$\begin{aligned} \boldsymbol{\nabla} J(\mu_\theta) &= \int_{\mathcal{S}} \boldsymbol{\nabla}_\theta \mu_\theta(s) \boldsymbol{\nabla}_\theta Q^\mu(s, a)|_{a=\mu_\theta(s)} ds \\ &= \mathbb{E}_{s\sim\rho^\mu}\Big[\boldsymbol{\nabla}_\theta \mu_\theta(s) \boldsymbol{\nabla}_\theta Q^\mu(s, a)|_{a=\mu_\theta(s)}\Big] \end{aligned} \tag{5.37}$$

There is an underlying connection between the stochastic and deterministic theorems. In fact, deterministic case is a limit case of the stochastic case. The proof the this statement as well as the deterministic policy gradient theorem proof are given in the supplementary Material of [34].

5.5.1.3 Deep Deterministic Policy Gradient (DDPG)

Q-learning algorithm cannot be applied in continuous action spaces. So the DPG approach should be followed. Actor network and $\mu(s|\theta^{\mu})$ gives the policy which deterministically maps states to a specific action. Critic network Q(s,a) is learned using the Bellman equation as in Q-learning. The actor is updated by the following update rule, following the chain rule to the expected return:

$$\boldsymbol{\nabla}_{\theta^{\mu}} J \approx \mathbb{E}_{s_t \sim \rho^{\beta}} \Big[\boldsymbol{\nabla}_{\theta^{\mu}} \mu(s|\theta^{\mu})|_{s=s_t} \boldsymbol{\nabla}_a Q(s,a|\theta^Q)|_{s=s_t, a=\mu(s_t)} \Big] \tag{5.38}$$

As it's name suggests, DDPG [24] is an extension of DPG algorithms that use the deterministic Policy Gradient theorem to learn about a deterministic policy. This modification is inspired by the success of DQN algorithm analyzed in a previous section. This algorithms takes advantage of the two important practices used in DQN, use of a replay buffer to avoid correlation between the training samples and use of a target network to avoid potential divergence effects.

A similar approach is also followed for this algorithm, but modified for the actor-critic architecture and using soft target updates instead of directly copying the NN weights. The copies of target actor and critic networks are used to calculate the target values. Their weights are updated by slowly tracking the learned networks,

$$\theta' \leftarrow \tau\theta + (1-\tau)\theta', \ \tau \ll 1 \tag{5.39}$$

The parameters are updated slowly, improving the learning speed.

A big challenge with continuous action spaces is exploration. DDPG can treat the exploration problem independently from the learning algorithm, by using a exploration policy μ' by adding noise sampled from a noise process $\mathcal{N}$ to the actor policy:

$$\mu'(s_t) = \mu(s|\theta_t^{\mu}) + \mathcal{N} \tag{5.40}$$

where $\mathcal{N}$ can be chosen to fit the environment. For example the noise model that used in [24] was the Ornstein-Uhlenbeck process. The pseudocode of the algorithms is given bellow.

Algorithm 13 DDPG algorithm

Initialize parameters θ_Q of the critic policy $Q(s, a|\theta_Q)$ and θ_μ of the actor policy $\mu(s|\theta^\mu)$
Initialize parameters of the target networks Q' and μ': $\theta^{\theta'} \leftarrow \theta^Q$ and $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize the replay buffer R
for each episode: **do**
  Initialize random process $\mathcal{N}$ for exploration
  Get initial state/observation s_1
  for each time step of episode t = 1, ..., T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ using current policy and exploration noise
    Apply the action and get the reward r_t and the new state s_t
    Store transition tuple (s_t, a_t, r_t, s_{t+1}) in the replay buffer R
    Sample a random minibatch of N transition tuples from R (s_i, a_i, r_i, s_{t+i})
    Get the result from the value estimation from target networks Q', μ':

$$y_i = r_i + \gamma Q'(s_{t+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$$

    Get the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update critic parameters by minimizing L
    Update actor parameters by the deterministic policy gradient:

$$\boldsymbol{\nabla}_{\theta^\mu} J \approx \mathbb{E}_{s_t \sim \rho^\beta}\Big[\boldsymbol{\nabla}_{\theta^\mu}\mu(s|\theta^\mu)|_{s=s_t}\boldsymbol{\nabla}_a Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t)}\Big]$$

    Soft update of the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

5.5.2 Proximal Policy Optimization algorithms (PPO)

PPO algorithms [32] is a family of policy gradient methods which optimizes a surrogate objective function by stochastic gradient ascent. Most common policy gradient methods perform parameter updates to the direction of the gradient for each data sample, PPO updates multiple epochs of minibatch updates. This algorithm is inspired by the algorithm Trust Region Policy optimization (TRPO) [31]. It has same benefits of TRPO but it is simpler to implement and has better sample complexity.

In TRPO, the surrogate objective function which is maximized is the following expectation value:

$$L^{CPI}(\theta) = \mathbb{E}_t\Big[\frac{\pi_\theta(a_t|s_t}{\pi_{\theta_{old}(a_t|s_t)}} A_t\Big] \tag{5.41}$$

Updates are subject to a constraint on the size of the policy update:

$$\mathbb{E}_t\Big[KL[\pi_{\theta_{old}}(\cdot, s_t), \pi_\theta(\cdot|s_t)]\Big] \tag{5.42}$$

θ are the current policy parameters, θ_{old} are the policy parameters before the update, KL is the Kullback–Leibler divergence which measures the distance between two stochastic

policies (before and after the update) and A_t is called the advantage function and given by:

$$A_t(s,a) = Q_\pi(s,a) - V_\pi(s),\ \forall s \in \mathcal{S}, a \in \mathcal{A}$$

Theory behind TRPO suggests that instead of the constraint, a penalty should be added in the objective function:

$$L(\theta) = \mathbb{E}_t\Big[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}(a_t|s_t)}}A_t - \beta KL[\pi_{\theta_{old}}(\cdot, s_t), \pi_\theta(\cdot|s_t)]\Big] \tag{5.43}$$

where β is a coefficient. TRPO uses the constraint instead of the penalty, because it is hard to find the value of β that performs well across many problems. If one defines the probability ratio

$$r_i(\theta) = \frac{\pi_\theta(a_t|s_t}{\pi_{\theta_{old}(a_t|s_t)}} \tag{5.44}$$

where $r(\theta_{old}) = 1$, then the CPI objective can be written as:

$$L^{CPI}(\theta) = \mathbb{E}_t\Big[r_i(\theta)A_t\Big] \tag{5.45}$$

Without any constraint, the maximization of the objective would give a very large policy update, which would lead to divergence. So authors in [32] propose a penalty to the updates that change the policy away from 1:

$$L^{CLIP}(\theta) = \mathbb{E}_t\Big[\min(r_t(\theta)A_t, \mathbf{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t)\Big] \tag{5.46}$$

where ϵ is a hyperparameter.

The penalty term gives the probability ratio clipped between the interval $[1-\epsilon, 1+\epsilon]$. The final objective is the minimum between the clipped and unclipped objective.

An alternative approach to the clipped surrogate objective is to use a penalty on KL divergence and to adapt the penalty coefficient so that some target value of KL d_{targ} is achieved. KL divergence is a type of statistical distance, a measure of how one probability distribution is different from a second, reference probability distribution. This objective has the form:

$$L^{KLPEN}(\theta) = \mathbb{E}_t\Big[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}(a_t|s_t)}}A_t - \beta KL[\pi_{\theta_{old}}(\cdot, s_t), \pi_\theta(\cdot|s_t)]\Big] \tag{5.47}$$

Then, the expectation of KL term is computed:

$$d = \mathbb{E}_t\Big[KL[\pi_{\theta_{old}}(\cdot, s_t), \pi_\theta(\cdot|s_t)]\Big]$$

- If $d < d_{targ} : \beta \leftarrow \frac{\beta}{2}$
- If $d > d_{targ} : \beta \leftarrow \beta \cdot 2$

The updated β is used for the next policy update. With this update, there are occasions with big changes in policy, but they are rare and does not affect the algorithm.

The surrogate losses already shown can be computed and differentiated. Practical implementations with automatic differentiation can use losses L^{CLIP} or L^{KLPEN} and perform multiple steps of stochastic gradient ascent on this objective. There is an option to share parameters across the two Neural Networks that approximate the policy and the value function. In this case the loss function should combine the policy surrogate loss and the value function error loss, or even add an additional entropy term that ensures sufficient exploration:

$$L_t^{CLIP+CF+S}(\theta) = \mathbb{E}_t\Big[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t)\Big] \tag{5.48}$$

where c_1, c_2 are some coefficients, S is the entropy term and L_t^{VF} is the squared error loss $(V_\theta(s_t) - V_t^{targ})^2$.

5.6 Trigonometric Series Optimisation Algorithm (TSOA)

For experimental implementations smooth optimal control might be more relevant. Inspired by [37], one can approximate a continuous function to an arbitrary precision by using trigonometric series form. Rabi frequency and detuning can both be represented by finite trigonometric series as bellow:

$$\Omega(t) = a_0 + \sum_{k=1}^{p}(a_{2k-1}\cos kt + a_{2k}\sin kt) \tag{5.49}$$

$$\Delta(t) = b_0 + \sum_{k=1}^{p}(b_{2k-1}\cos kt + b_{2k}\sin kt) \tag{5.50}$$

where p is the number of harmonics. The parameters vector is normalised, that is:

$$\|\mathbf{v}\| = 1$$

where $\mathbf{v} = [a_0, a_1, ..., a_{2k}, b_0, b_1, ..., b_{2k}]$.

5.6.1 Optimization process

The optimization process can be formulated as an MDP, so that it can be solved by reinforcement learning algorithms. Policy gradient algorithms and actor critic methods provide the right algorithms to formulate MDP with continuous state and action spaces. The MDP will have only one time step, although this time step does not coincide with the time of the quantum system evolution. In this time step, the agent will observe the state of the MDP and will produce an action, which will represent the parameters of the trigonometric series of the two controls, Rabi frequency and detuning.

5.6.2 State and Action space

The state space of the optimization MDP will be a continuous space. The state of the MDP will be the elements of the density matrix of the current quantum state.

$$\mathcal{S} = \{s \mid s = (\rho_{11}, \mathrm{Re}\{\rho_{12}\}, \mathrm{Im}\{\rho_{12}\})\} \tag{5.51}$$

The state of the quantum system will be fed in the parameterized policy and the policy will output an array of parameters that will be used as the parameters of the trigonometric series. So an action is a tuple of parameters and the actions space is defined as follows:

$$\mathcal{A} = \{(a_1, ..., a_{4k}) \mid a_i \in R \text{ where } 1 \leq i \leq 4k \text{ and } k \in \mathbb{N}\} \tag{5.52}$$

where a_i is a parameter of the trigonometric series and k is the number of harmonics.

The model of the agent consists of two deep Neural Networks, the one will represent the policy and the other will represent the value function. With many repetitions, searching through the loss landscape, the agent will hopefully arrive at a point that will be able to output parameters that achieve the desired fidelity. The policy NN can be visualized as follows:

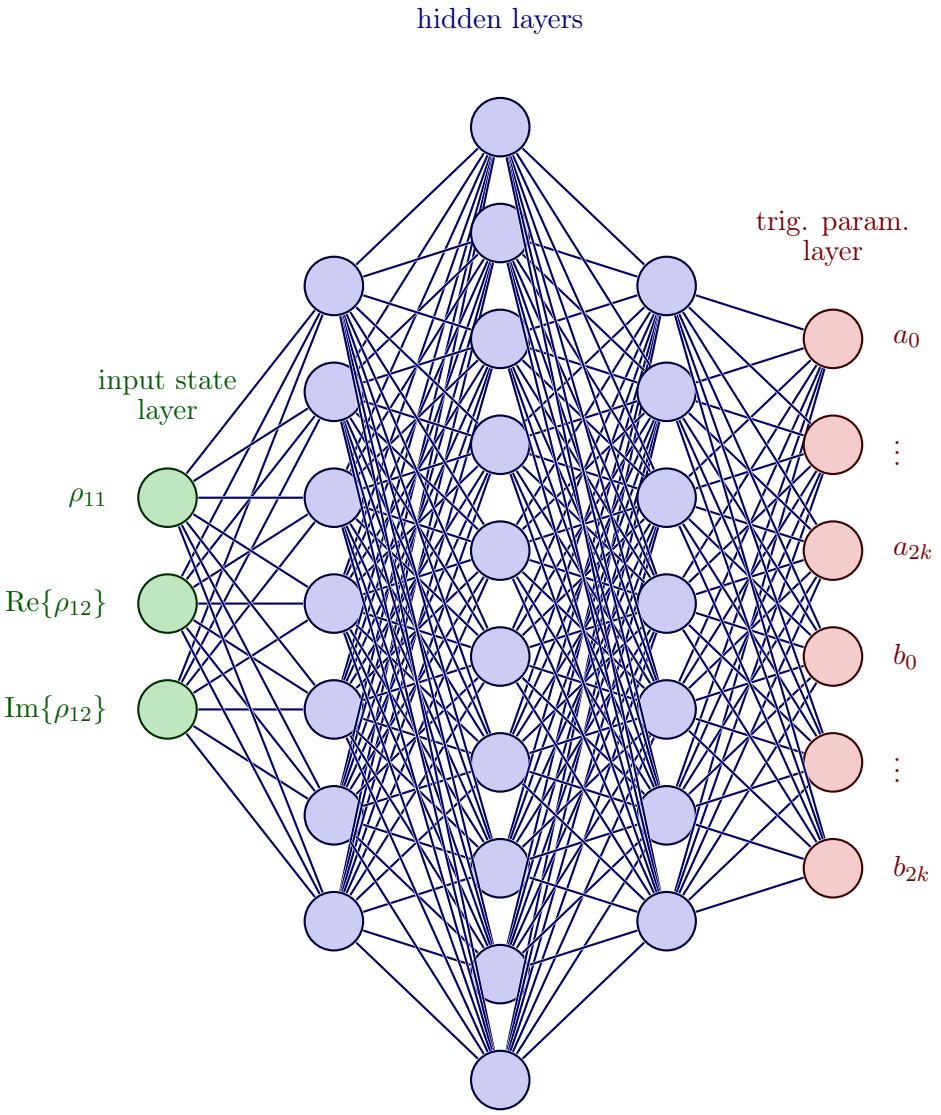


FIGURE 5.10: Parameterized policy ANN. Input layer consists of the states $s \in \mathcal{S}$ and the output layer consists of the parameters of the trigonometric series for Ω and Δ

The training process will follow the same process as for the other algorithms used so far. Backpropagation and stochastic gradient descent algorithms will be used to update and optimize the parameters of the policy and value function neural networks. The agent in this process will try to represent a parameters optimization algorithm. The same problem can be stated in a different way with different assumptions, with a different MDP formulation. The same algorithms can be used to solve the same problem with a differently constructed environment.

Chapter 6

Qubit control: Methodologies, results and discussion

6.1 Quantum Control Hamiltonian

The Hamiltonian in quantum control is usually separated in two terms and can be expressed in the following form [12]:

$$H = H_0 + H_I(t) \tag{6.1}$$

The first operator H_0 is called the free Hamiltonian and represents the sum of kinetic and potential energy operators, without any external interacting field. The second operator $H_I(t)$ is the interaction Hamiltonian and represents the interaction of the quantum system with an external field. The time-independent Schrodinger equation gives the eigenenergies and eigenvectors of the free Hamiltonian:

$$H_0|\psi_n\rangle = E_n|\psi_n\rangle, \forall|\psi_n\rangle \in \mathcal{H} \tag{6.2}$$

Using the eigenvectors as basis for the Hilbert space $\mathcal{H}$ of the quantum system, the arbitrary vector in the Hilbert space can be expanded in this basis as:

$$|\psi(t)\rangle = \sum_{n=1}^{N} a_n(t)|\psi_n\rangle \tag{6.3}$$

The basis of the Hilbert space is still the free Hamiltonian eigenvectors, and the interaction will affect the coefficients of the eigenvectors. From the Time-Dependent Schrodinger Equation (TDSE), for a qubit (a two-level system, $N = 2$) the problem reduces to a system of coupled ordinary differential equations with variables the probability amplitudes. The vector $|\psi(t)\rangle$ can be written in array form

$$|\psi(t)\rangle = \begin{pmatrix} a_1(t) \\ a_2(t) \end{pmatrix}$$

and from the TDSE:

$$i\hbar\frac{d}{dt}|\psi(t)\rangle = \big[H_0 + H_I(t)\big]|\psi(t)\rangle \tag{6.4}$$

the system of ODE arises:

$$i\hbar\begin{pmatrix}\dot{a_1}(t)\\ \dot{a_2}(t)\end{pmatrix} = \big[H_0 + H_I(t))\big]\begin{pmatrix}a_1(t)\\ a_2(t)\end{pmatrix} \tag{6.5}$$

The system can be solved analytically in some cases or numerically. The simulations of the quantum system in this thesis are numerical solutions of this coupled system (6.5).

6.2 Two level system Hamiltonian

Let's assume that a two level system interacts with an external control electromagnetic field. The electric field, under the electric dipole approximation, has the general form:

$$E(\vec{r},t) \approx E(t) = \epsilon(t)\cos[\omega t + \phi(t)] \tag{6.6}$$

where $\epsilon(t)$ is the envelope, ω the angular frequency and $\phi(t)$ the time-dependent phase of the field. The lower energy state $|\psi_1\rangle$ is called the ground state and the higher energy state $|\psi_2\rangle$ is called the excited state. The state of the qubit can be represented by the 2×2 density matrix:

$$\rho = \begin{bmatrix}\rho_{11} & \rho_{12}\\ \rho_{21} & \rho_{22}\end{bmatrix}, \tag{6.7}$$

where the diagonal element ρ_{ii} is the population of state $|\psi_n\rangle$, $n = 1, 2$, and the off-diagonal elements $\rho_{21}^* = \rho_{12}$ express the coherence between the ground state and the excited state of the qubit. For a closed system without losses, as the one considered in the current context, it is $\rho_{22} = 1 - \rho_{11}$, thus its general state can be represented by the triplet $(\rho_{11}, \mathrm{Re}\{\rho_{12}\}, \mathrm{Im}\{\rho_{12}\})$.

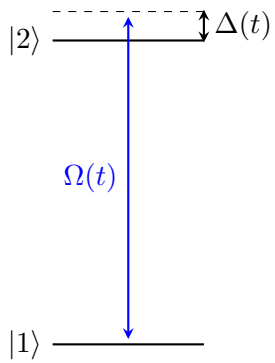


FIGURE 6.1: Two-level system

The free Hamiltonian of the system, without the external field can be written in diagonal form as:

$$H_0 = \begin{bmatrix} E_1 & 0 \\ 0 & E_2 \end{bmatrix} \tag{6.8}$$

where E_n, $n = 1, 2$ are the eigenvalues (eigenenergies) of the free Hamiltonian operator (H_0). The interaction of the system with the external field, can be described by adding the interaction term in the Hamiltonian operator:

$$H_I(t) = -\vec{d} \cdot \hat{\epsilon} E(t) \tag{6.9}$$

where $\vec{d}$ is the dipole moment operator and $\hat{\epsilon}$ is the unit polarization vector of the electric field. The total Hamiltonian which can express a two level system (qubit) interacting with a classical external field under the Rotating Wave Approximation (RWA) can be written in the following form [36]:

$$H(t) = \frac{\hbar}{2} \begin{bmatrix} \Delta(t) & \Omega(t) \\ \Omega(t) & -\Delta(t) \end{bmatrix} = \hbar \frac{\Delta(t)}{2} \sigma_z + \hbar \frac{\Omega(t)}{2} \sigma_x \tag{6.10}$$

where $\Delta(t) = \omega - \omega_0 + \dot{\phi}(t)$ is the detuning from the qubit frequency, $\Omega(t) = -\frac{d_{21}\ \epsilon(t)}{\hbar}$ is the time-dependent Rabi frequency, $\omega_0 = \frac{E_2 - E_1}{\hbar}$ is the resonance frequency and d_{21} is the induced matrix element of the dipole moment operator. In the presence of chirp ($\dot{\phi} \neq 0$), the detuning is also time-dependent and serves as an extra control function besides the Rabi frequency. The basic steps of the derivation of the total Hamiltonian can be found in the appendix A.

The dynamics is governed by the von-Neumann equation:

$$i\hbar \frac{d\rho}{dt} = [H, \rho]. \tag{6.11}$$

The problem of interest is the complete state transfer from state 1 to state 2. Initially the qubit is in the ground state 1, $\rho_{11}(0) = 1$, $\rho_{22}(0) = 0$ and $\rho_{12}(0) = \rho_{21}(0) = 0$,.

6.3 Qubit MDP

Dynamic programming and reinforcement learning algorithms have a very specific requirement from the problem formulation. It should be expressed as a Markov Decision Process (MDP). As already stated in previous section, an MDP is a tuple $(\mathcal{S}, \mathcal{A}, p, \gamma)$, where

- $\mathcal{S}$ is the state space
- $\mathcal{A}$ is the action space
- $p : \mathcal{S} \times \mathbb{R} \times \mathcal{S} \times \mathcal{A}$ is a map that gives the probability that a state s' and a reward r happens based on previous state $s \in \mathcal{S}$ and the action $a \in \mathcal{A}$
- γ is the discount factor

These spaces have to be defined for the problem of population transfer for the two level system.

6.3.1 Continuous State space

The state space of the MDP can be defined in many ways. A natural choice would be to use the state of the quantum system, i.e. the density matrix ρ or the vector in the Hilbert space $\psi \in \mathcal{H}$ in the state of the MDP. The quantum state can also be stated to have the Markov property, since the current state and the next actions do not depend on the history of the process. Although the quantum state space is an infinite dimensional space. Even if it was a discrete space, as the number of states increases the complexity of the MDP increases exponentially. This is the point where deep learning comes into the picture, since deep neural networks can generalize into unseen states, so there is no need to be trained for all states. At this formalism, the state can be defined as a function of quantum state (density matrix elements) and the previous time step Rabi frequency and/or detuning of the system:

$$s_t^{\Delta} = (\Omega_t, \Delta_t, \rho_{11}, \mathrm{Re}\{\rho_{12}\}, \mathrm{Im}\{\rho_{12}\}) \tag{6.12}$$

or

$$s_t^{res} = (\Omega_t, \rho_{11}, \mathrm{Re}\{\rho_{12}\}, \mathrm{Im}\{\rho_{12}\}) \tag{6.13}$$

where the quantum system here is a simple two level system (qubit) and the two possible state spaces can include detuning Δ or not in the case of the resonant case ($\Delta = 0$).

6.3.2 Finite State space

It would be helpful to start from something more simple. Following the formulation of [7], we can define a discrete state space, which is the tuple $(\mathcal{T} \times \mathcal{O})$, where $\mathcal{T}$ is the set of the time steps in an episode or trajectory and $\mathcal{O}$ is the discrete space of the available Rabi frequencies. Step size pulses are only allowed, with $\Omega \in \mathcal{O}$ taking discrete values in the interval $[-1, 1]$. The considered pulses are resonant and without chirp. The state at time step t is given by the tuple:

$$s_t = (t, \Omega_t) \tag{6.14}$$

where $t \in \mathcal{T}$ and $\Omega_t \in \mathcal{O}$.

6.3.3 Action space

The action space in this environment can be discrete or continuous and it corresponds to the changes in the Rabi frequency from the previous time step. If Ω_t is the Rabi frequency at time t, the action that is taken at next time step is $a_t = \delta\Omega_t$. So the Rabi frequency at time t would be:

$$\Omega_{t+1} = \Omega_t + \delta\Omega_t \tag{6.15}$$

In two control setup, detuning is also in the involved in the action space, so the action is defined as a tuple of two corrections $a_t = (\delta\Omega_t, \delta\Delta_t)$, one for the Rabi frequency and another one for the detuning. In this case, the action corrects or perturbs both Rabi

frequency and detuning, and other than the correction (**??**), there is another correction for the detuning at time step t:

$$\Delta_{t+1} = \Delta_t + \delta\Delta_t \tag{6.16}$$

6.3.4 Reward function

The metric that shows the success of state transfer process is the fidelity. Fidelity is given as the trace of the matrix product:

$$\mathcal{F} = \mathrm{Tr}[\rho_{tar}\rho(t)]. \tag{6.17}$$

Here, $\rho(t)$ is obtained from the evolution of Eq. (6.11), when starting from the initial condition $\rho(0)$. For the population inversion problem the fidelity is simply the population of the excited state:

$$\mathcal{F} = \rho_{22}(t). \tag{6.18}$$

The reward function amongst others, should be a function of the fidelity at all time steps or only at the terminating time step, when the episode (pulse sequence) ends. The optimal policy should achieve the maximum fidelity at the least possible time, so time steps of the episode is also something that should be taken into account. In fact each extra time step is penalized with -1 in the reward. By this, the agent that is trying to find the optimal policy will try to reduce the time steps and also achieve maximum fidelity. The MDP is an episodic process, so it is divided into episodes, which are the application of pulse sequences to achieve the state transfer. The episode ends when the maximum allowed time steps is reached, since due to decoherence effects, that process should not take more time that the decoherence timings of each system that is studied. It can also end earlier than final time T, if the target fidelity which is set as a parameter of the system is already achieved earlier in time.

The reward function will have the following form:

$$R_t = \begin{cases} \sqrt{\mathcal{F}(t)} + cu(\mathcal{F}(t)), & \text{if the episode terminates} \\ \sqrt{\mathcal{F}(t)} - 1, & \text{in all intermediate steps} \end{cases} \tag{6.19}$$

where c is a constant reward that is given to the agent if it achieved the target fidelity during the episode and u is the Heaviside function that is defined as:

$$u(\mathcal{F}) = \begin{cases} 1, & if \mathcal{F} \geq \mathcal{F}_{target} \\ 0, & else \end{cases} \tag{6.20}$$

In the function of the reward, we use the square root of the fidelity because due its value range 0 to 1 the squared fidelity gives higher rewards and during the training the convergence to optimal policies was faster when we used the squared fidelity. It is important to note that the squared fidelity is only used for the reward value. When we compare the final fidelity with the threshold we want to achieve we use actual value of the fidelity as input in the unit step function in relations (6.19) and (6.20).

6.4 Model of MDP

There is no model for the MDP, there is no transition probabilities from state to state defined for the quantum system for this formulation of the MDP and the specific quantum system. So the problem will be solved with model-free reinforcement learning methods Temporal Difference (TD), policy gradient methods or actor-critic methods.

6.5 Simulation

The reinforcement learning agents interact with the MDP environment via simulation of quantum state evolution. For the simulation and numerical evolution of the quantum dynamics the open source tool of QuTiP [20] is used. QuTiP is an open-source software for simulating the dynamics of open quantum systems. For the purpose of the current master thesis, QuTiP offers all the tools needed for simulating the two level (qubit) system.

6.6 Units of the qubit system

In all simulations and quantum system evolution the unit system is determined by setting some characteristic values in units:

$$\hbar = \Omega_0 \ (or \ \Omega_{max}) = t_0 = 1$$

This means the units of the Rabi frequency or the detuning are of Ω_0 or Ω_{max} units and the time unit is $t_0 = \Omega_0^{-1}$. This unit system is the same for all figures throughout the document.

6.7 Temporal Difference Methods

6.7.1 Tabular methods

Three classic temporal different methods are applied to the state transfer problem in the two level system defined above. More specifically the transition from the one eigenstate to the other. The environment of the MDP is defined with the discrete state and action spaces and the reward as defined in the previous section. The algorithms used are the following:

- Q-learning [42]
- Expected Sarsa [38]
- Double Q-Learning [19]

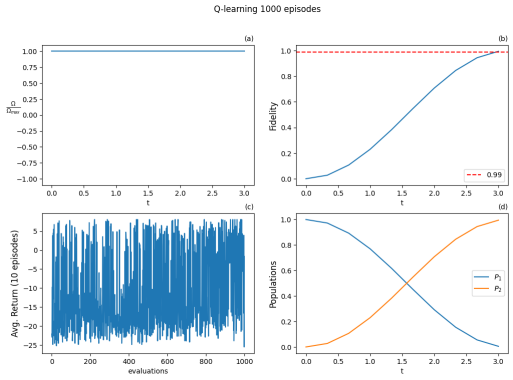
During the algorithms the sampling policy used is the ϵ-greedy policy, which chooses the optimal policy based on Q value function with probability $1 - \epsilon$ and samples a random action from the action space with probability ϵ. This ensures that in sufficient amount of time all states and actions combinations are explored sufficient times so that they are well valued. In the current implementations of the algorithms, ϵ starts with value 1, which means that the policy is a random policy and gradually reduces as the number of episodes grow, until it reaches a minimum value 0.05 (choose randomly 5% of the times). The threshold fidelity for the tabular methods is set at $\mathcal{F}_{th} = 0.99$.

Maximum time steps	30
discount factor γ	0.99
minimum ϵ	0.05
Detuning Δ	0
Rabi frequency Ω	$-1 \leq \Omega \leq 1$
Actions $\delta\Omega$	{-2,2}
Target fidelity	0.99

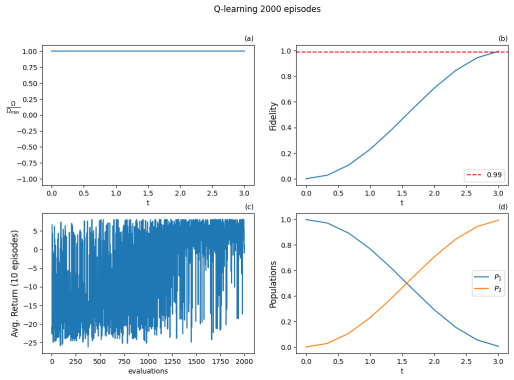
TABLE 6.1: Tabular methods parameters - 2 actions

6.7.1.1 Q-Learning

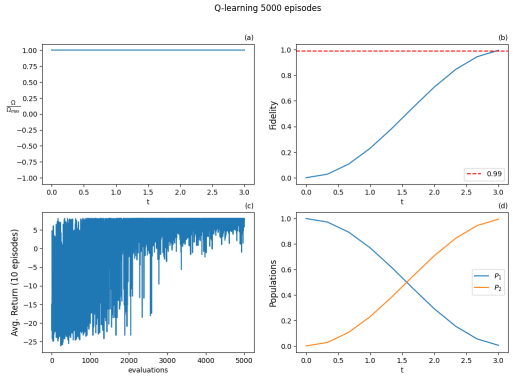
First, the Q-learning algorithm is utilized. Staring with only two allowed actions, pulses are only restricted to bang-bang form. It can be seen from figure (6.2a) that the achieved policy is sub-optimal, since system reaches terminal time with target fidelity 0.99. The parameters of the algorithm are shown in table 6.1, while the results are displayed in Figs. 6.2a, 6.2b, 6.3c, 6.3d. These figures show the evolution of the learning process as more and more episodes are presented to the model. We can see that even though the variance of the average return is very high at the early state of the training, the agent is able to produce the optimal policy and $\Omega(t)$ attains the optimal π-pulse. Due to the small action space, the agent quite quickly finds the optimal policy for the complete state transfer.



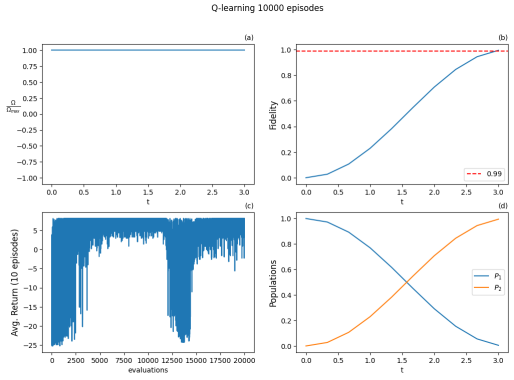
(A) 1000 episodes



(B) 2000 episodes



(C) 5000 episodes



(D) 20000 episodes

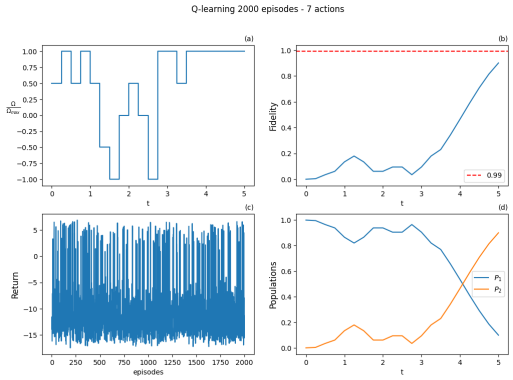
FIGURE 6.2: Q-Learning - 2 actions

It is important to check what if we allow Rabi frequency to take more intermediate

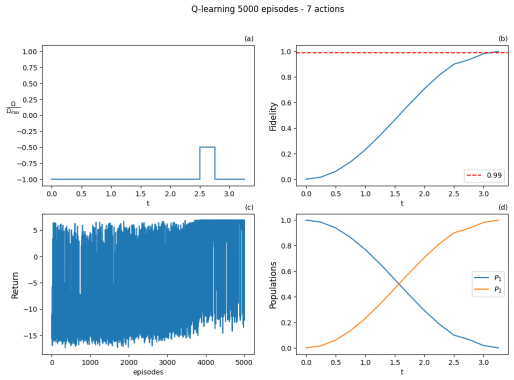
values between -1 and 1 and whether the agent would be able to find the optimal policy. This is a more difficult problem for the agent and since the number of actions increase, it might need more exploration and training episodes to converge to optimal policies. The next used environment will allow 7 actions.

Maximum time t	5
Maximum time steps	15
discount factor γ	0.99
minimum ϵ	0.05
Detuning Δ	0
Actions Ω	$\{-1, -\frac{1}{2}, -\frac{1}{4}, 0, \frac{1}{4}, \frac{1}{2}, 1\}$
Rabi frequency $\delta\Omega$	$-1 \leq \Omega \leq 1$
Target fidelity	0.99

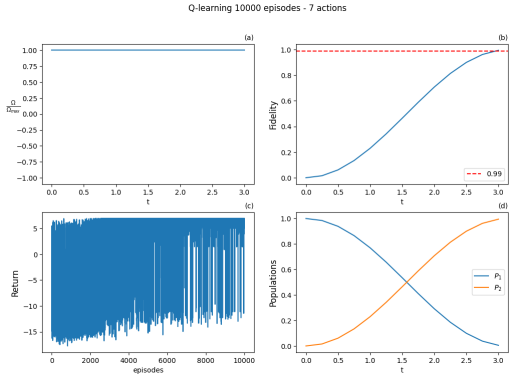
TABLE 6.2: Tabular methods parameters - 7 actions



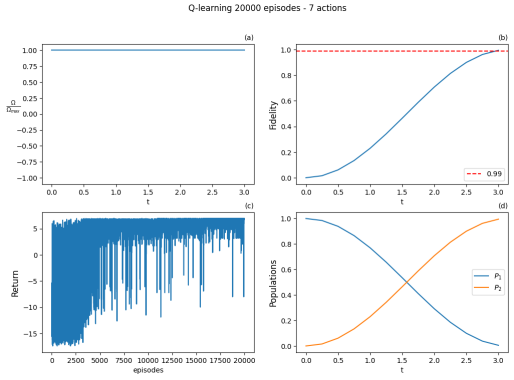
(A) 2000 episodes



(B) 5000 episodes



(C) 10000 episodes



(D) 20000 episodes

FIGURE 6.3: Q-Learning - 7 actions

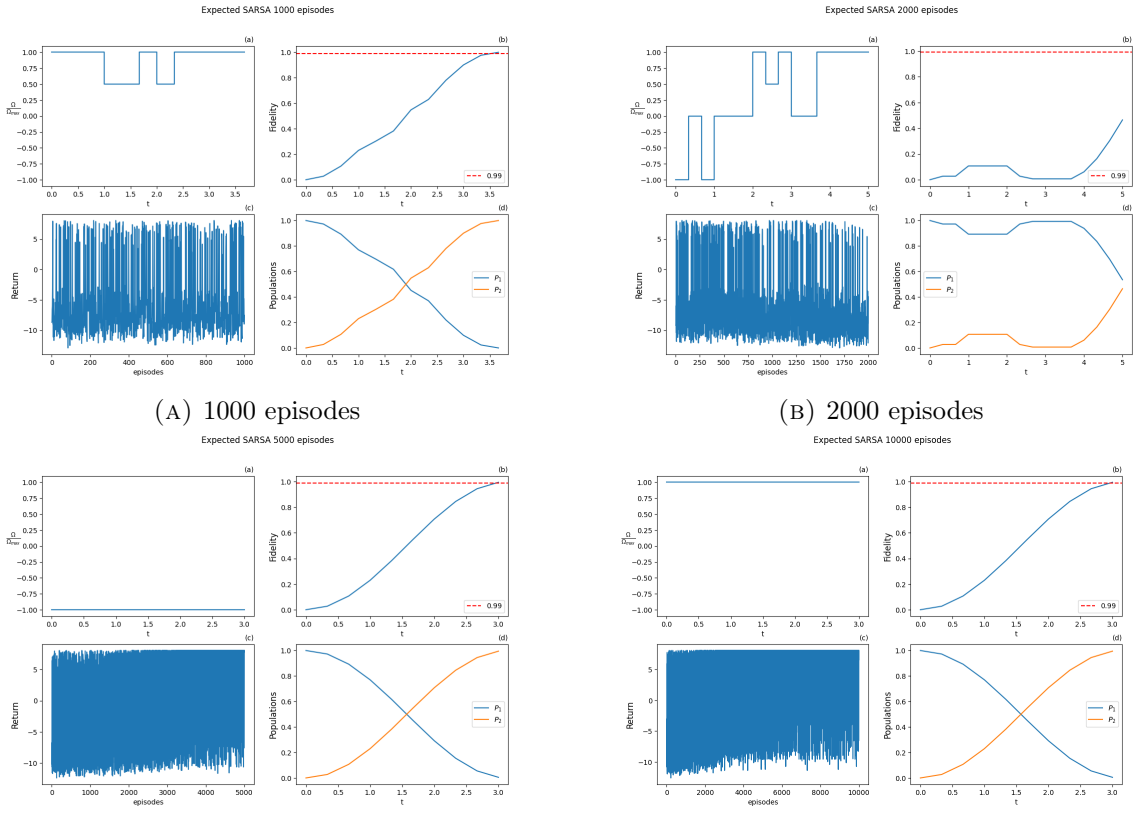
It is clear from the results that the agent is able to generate an optimal policy, since in the case of 7 actions, the optimal $\pi-$pulse is produced by the agent. As expected it took more episodes for the agent to produce the optimal policy, since the numbers of actions and possible states are increased. In the early stage of training, the algorithm has not succeeded to obtain an optimal policy yet, since the fidelity fails to attain the 0.99 threshold, Fig. 6.3a, and $\Omega(t)$ substantially deviates from the optimal π-pulse. On the other hand, at the later stage of 20000 episodes of training, it is clear that the agent is able to produce an

optimal policy which achieves the threshold fidelity, Fig. 6.3d, while $\Omega(t)$ attains the shape of the optimal π-pulse. Note that, since the threshold fidelity is only 0.99, it is obtained earlier than π units of time. The expected return (cumulative rewards) from the training episodes averaged from 10 episode samples, improves with the number of episodes. These figures contain all cumulative returns for all training episodes and show how the episode rewards change as training accumulates.

6.7.1.2 Expected Sarsa

Another algorithm that is investigated, is the Expected Sarsa algorithm [38]. It is a alternation of Sarsa algorithm that resembles Q-learning algorithm. Is uses a similar update rule as Q-learning, but instead of getting the best next action, it get the expected return from the probabilities to select an action based on action value function Q.

This algorithm seems to outperform Q-learning for the early stage of training. since the agent is able to attain the target fidelity as shown in 6.4a. It uses the same parameters as Q-learning algorithm. At early stage of training, the agent achieves the target fidelity but in a time period longer than π units of time, since $\Omega(t)$ deviates from constant π-pulse. In the later stage of training, the results are similar with Q-learning as can be seen in Fig. 6.4d.



(A) 1000 episodes

(B) 2000 episodes

(C) 5000 episodes

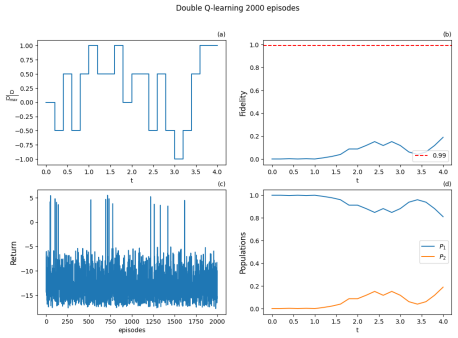
(D) 10000 episodes

FIGURE 6.4: Expected Sarsa - 7 actions

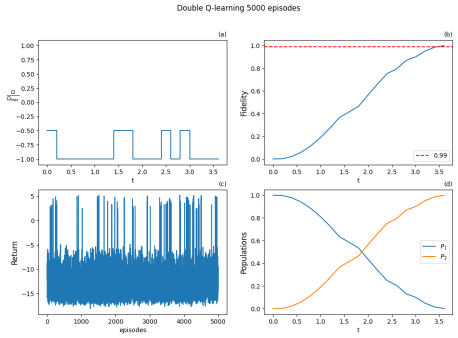
6.7.1.3 Double Q-Learning

Both Q-learning and Sarsa can experience a phenomenon which is called maximization bias [38]. This means that the maximum over estimated values is used implicitly as an

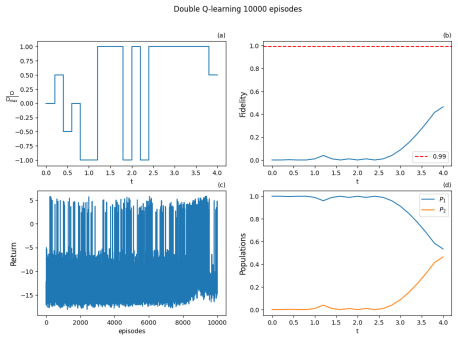
estimate of the maximum value. This can lead to a significant positive bias. The problem is that it uses the same samples for both maximization and estimation of the value.



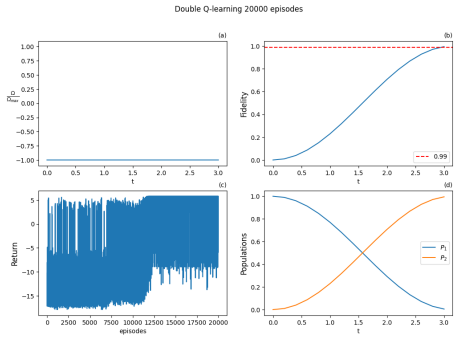
(A) 2000 episodes



(B) 5000 episodes



(C) 10000 episodes



(D) 20000 episodes

FIGURE 6.5: Double Q-Learning - 7 actions

Double Q-learning algorithm is able to produce the optimal policy with the same parameters as the other tabular methods with similar training episodes. During the training, the agent is not always able to attain the target fidelity and only at the latest state of the training it converges to the optimal policy, with less variance and $\Omega(t)$ being the π-pulse.

6.7.2 Approximation methods

Use of state function approximation gives more freedom in the formulation of the MDP and the possibility to use both discrete and continuous state space. The first formulation uses the same discrete state space used so far in the tabular methods. Although, the function approximators are able to approximate the real value function even in very large or even infinite state spaces. This opens up new opportunities in solving problems, such as state transfer or other problems in quantum control field that can be mathematically formalized as MDPs, using continuous spaces. Even with this generalization, TD learning methods cannot handle continuous action spaces, an issue that policy gradient and actor critic methods will come to solve in later sections. For now the action space is still discrete and will have the same form as in tabular cases. Now state space can also be continuous, which offer more flexibility and the possibility to use quantum state to describe the MDP environment state.

6.7.2.1 Deep Q-Network (DQN)

Q-learning algorithm has been generalized to DQN algorithm. Its theoretical analysis has already been given in previous chapter. It is now time to use it in practice and try to control the qubit system. At first the goal is to achieve fidelity in state transfer more than 0.99. The parameters of the MDP and the parameters of the deep neural network used to approximate the action-value function are given in table 6.3.

The training process and its results are shown in Fig. 6.6 and the parameters of the execution in Table 6.3 the agent is able to produce a policy from the approximated action-value function which coincides with the optimal solution of the π-pulse. The fidelity achieved at the end of the pulse is higher than 0.99, as it is shown in figure 6.6b. Average return (10 episodes average), shown in Fig. 6.6c is smaller at the start of the training. As the training continues, more and more states are presented into the agent, and by maximizing the rewards, the agent is able to find a path in the loss landscape that converges to the optimal policy, which indeed coincides with the π-pulse. Although during the whole training process the average Return experiences great variance. The average return in all figures from now on is the cumulative reward from all time steps of the episode averaged in a 10 episode sample. This way we are able to see how the current policy behaves and how its results approach the optimal policy as training process continues. At most of the cases we can observe that the reward is maximized based on our rewarding system, and this is a point at which the optimal policy has been succeeded.

Parameters	$\mathcal{F} = 0.99$	$\mathcal{F} = 0.9999$
Max time steps (N)	15	35
Ω_{max}	1	
End time (T)	$\frac{5}{\Omega_{max}}$	
Time step	$\frac{T}{N}$	
discount factor γ	0.99	
ϵ	0.1	
Detuning Δ	0	
Rabi frequency Ω	$\{-1, -\frac{3}{4}, -\frac{1}{2}, -\frac{1}{4}, 0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1\}$	
Actions $\delta\Omega$	$\{-2, -1, -\frac{1}{2}, -\frac{1}{4}, 0, \frac{1}{4}, \frac{1}{2}, 1, 2\}$	
Target fidelity	0.99	0.9999
Training Iterations	3000	4000
Hidden layers (2)	(100, 75)	(100, 75)
Learning rate	0.001	
Optimizer	Adam	

TABLE 6.3: DQN parameters - Discrete state space

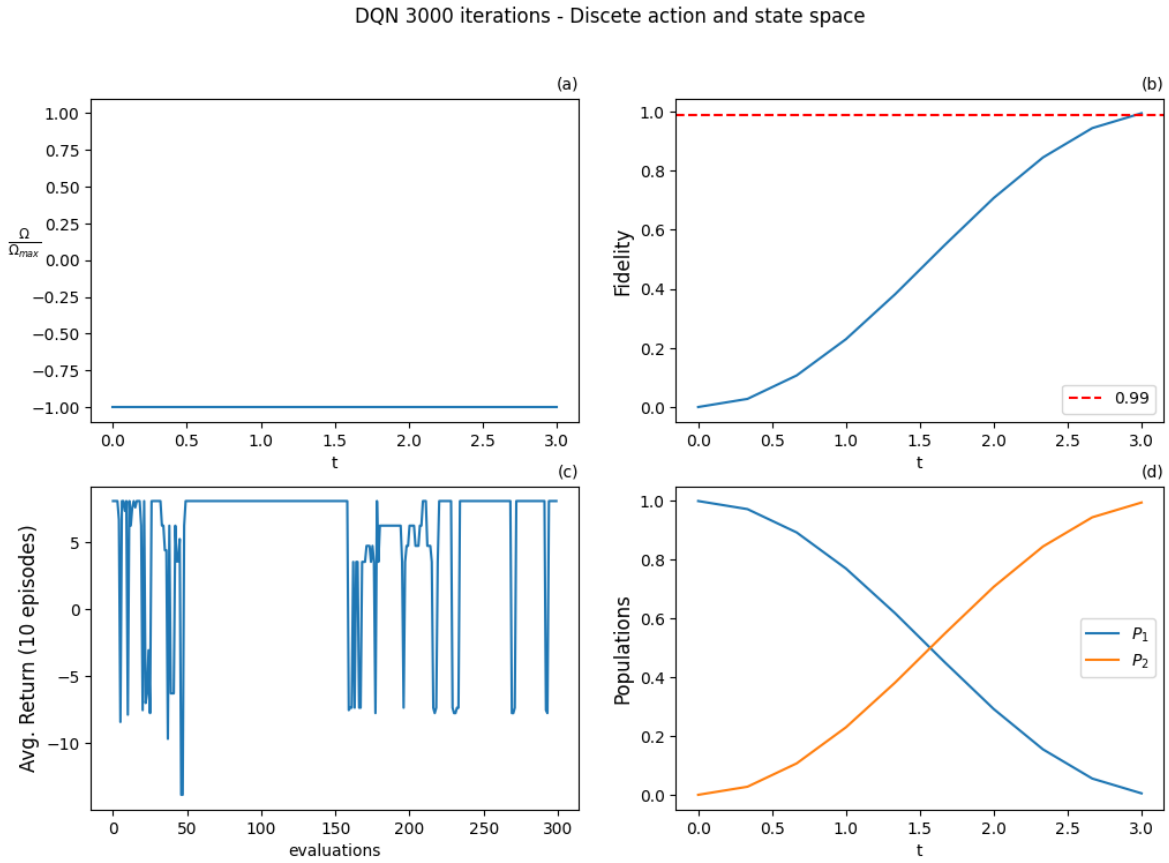


FIGURE 6.6: Results for DQN algorithm with 9 actions and discrete state space: (a) Optimal normalized Rabi frequency $\Omega(t)$, (b) Fidelity (population of excited state $|2\rangle$), (c) Expected return (cumulative rewards) from the training episodes averaged from 10 episode samples, (d) Populations of states $|1\rangle$ and $|2\rangle$.

Even though the fidelities achieved are a bit above 0.99 at the end of the pulse, this is not enough precision for the state of the art systems, such as quantum computers. The new target fidelity is now set to $\mathcal{F} = 0.9999$. Tabular methods are not able to cope efficiently with this increase in the fidelity goal, since the state space increases a lot and more training is required, especially if someone wants to allow more number of actions and pulses other than the bang-bang type.

To achieve fidelities greater than 0.9999, time should be discretized in smaller time steps. Max time the same and it is divided in more time steps. The precision of time steps is crucial for the maximum fidelity that can be achieved. With greater number of time steps, the number of possible MDP states increases. This requires more training time (more iterations) and possibly a larger neural network in terms of number of neurons in the hidden layers. The number of iterations has been increased to 4000.

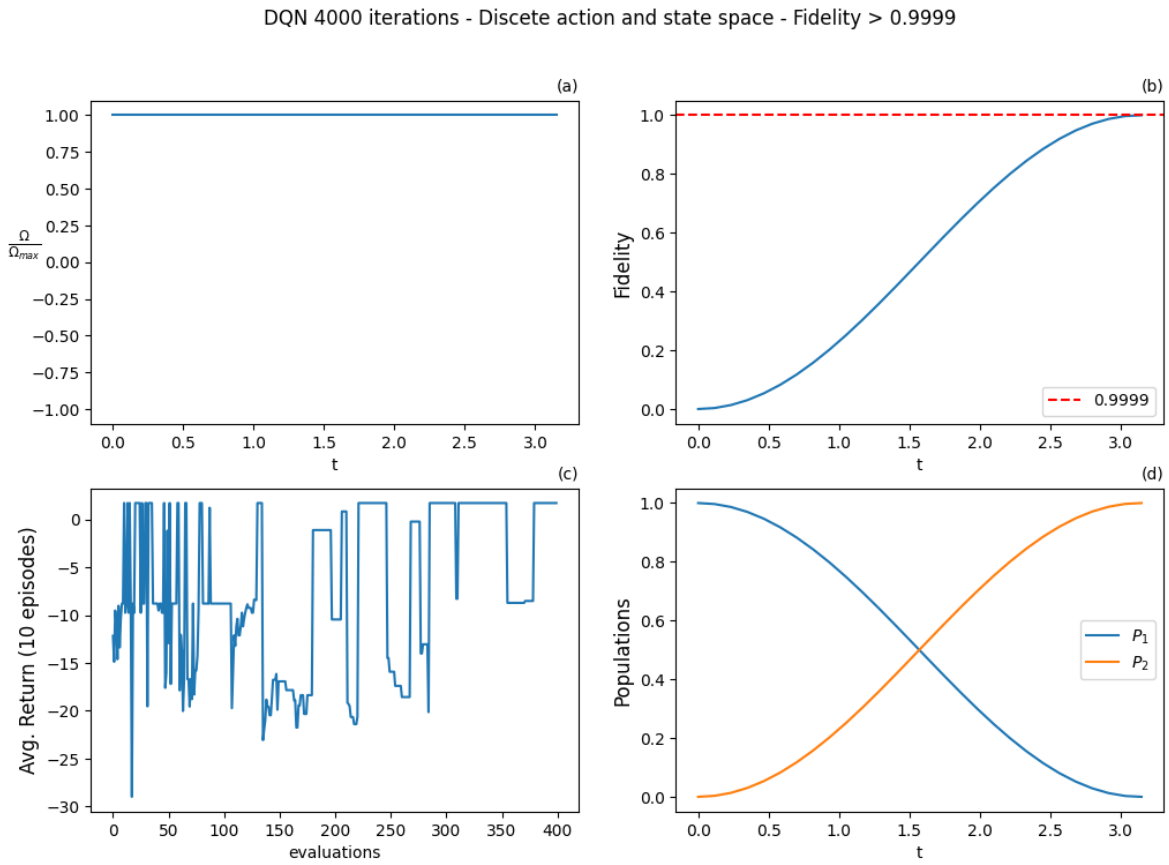


FIGURE 6.7: Results for DQN algorithm with 9 actions and discrete state space - Fidelity > 0.9999: (a) Optimal normalized Rabi frequency $\Omega(t)$, (b) Fidelity (population of excited state $|2\rangle$), (c) Expected return (cumulative rewards) from the training episodes averaged from 10 episode samples, (d) Populations of states $|1\rangle$ and $|2\rangle$.

The parameters set for the MDP are sufficient for the agent to achieve fidelities higher than 0.9999. It seems that the agent is able to learn and approximate the action-value function well enough and is able to generate a greedy policy that is sub-optimal ($\approx \pi$ pulse).

6.7.2.2 Deep Q-Network - Continuous state action

As already stated in a previous section, value function approximation offers more freedom for the state space. It can be even continuous, which allows to a more intuitive and quantum informed state space. This space can take information from the simulation by using the density matrix elements. The changes in parameters are shown in table 6.4. Training and results of the new more quantum-informed agent are shown in figure 6.8.

Parameters	$\mathcal{F} = 0.99$	$\mathcal{F} = 0.9999$
Max time steps (N)	15	30
Ω_{max}	1	
End time (T)	$\frac{5}{\Omega_{max}}$	$\frac{3.5}{\Omega_{max}}$
Time step	$\frac{T}{N}$	
discount factor γ	0.99	
ϵ	0.1	
Detuning Δ	0	
Rabi frequency Ω	$\{-1, -\frac{3}{4}, -\frac{1}{2}, -\frac{1}{4}, 0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1\}$	
Actions $\delta\Omega$	$\{-2, -1, -\frac{1}{2}, -\frac{1}{4}, 0, \frac{1}{4}, \frac{1}{2}, 1, 2\}$	
Target fidelity	0.99	0.9999
Training Iterations	2000	2000
Hidden layers (2)	(100, 50)	(100, 75)
Learning rate	0.001	
Optimizer	Adam	

TABLE 6.4: DQN parameters - Continuous state space

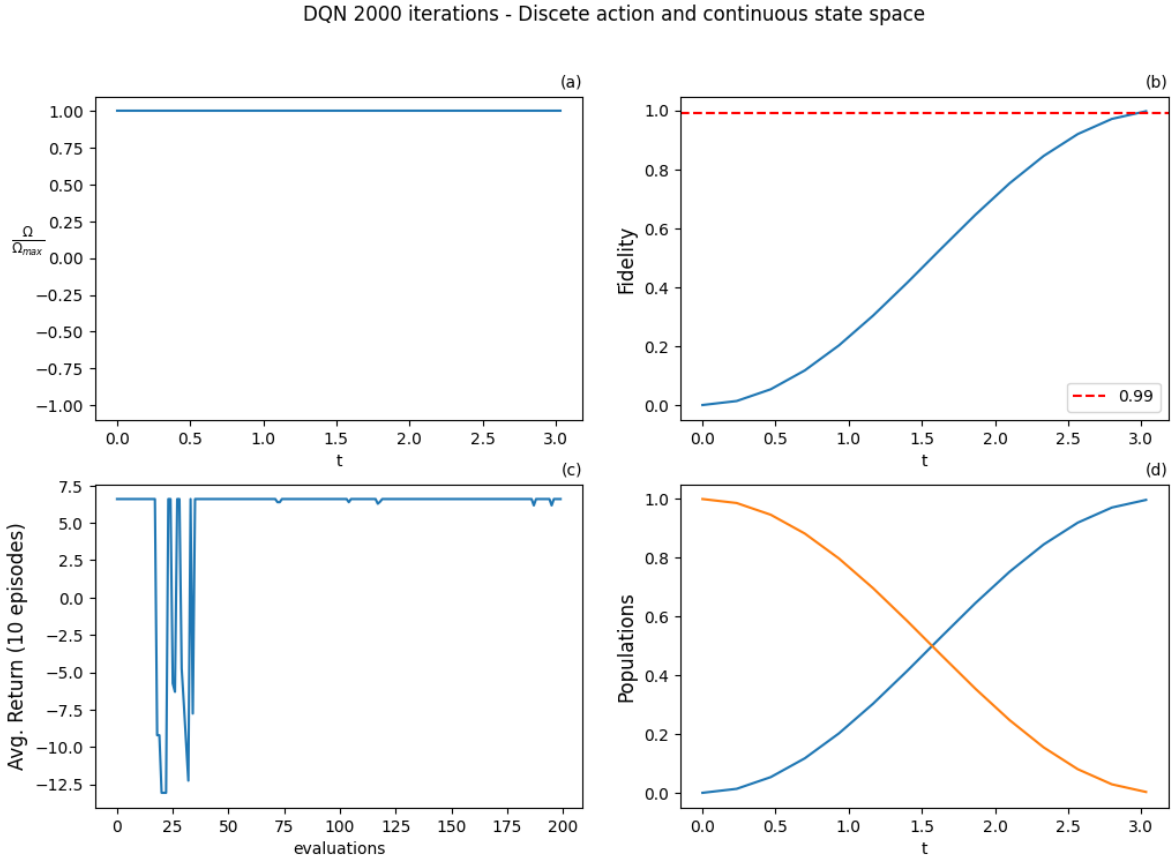


FIGURE 6.8: Results for DQN algorithm with 9 actions and continuous state space: (a) Optimal normalized Rabi frequency $\Omega(t)$, (b) Fidelity (population of excited state $|2\rangle$), (c) Expected return (cumulative rewards) from the training episodes averaged from 10 episode samples, (d) Populations of states $|1\rangle$ and $|2\rangle$.

This formulation was able to find the optimal policy with similar MDP and algorithm parameters with the discrete state space approach. The figure above shows that the agent is able to find the optimal trajectory that maximizes the total return and the state transfer is achieved with fidelity higher than 0.99 and 0.9999.

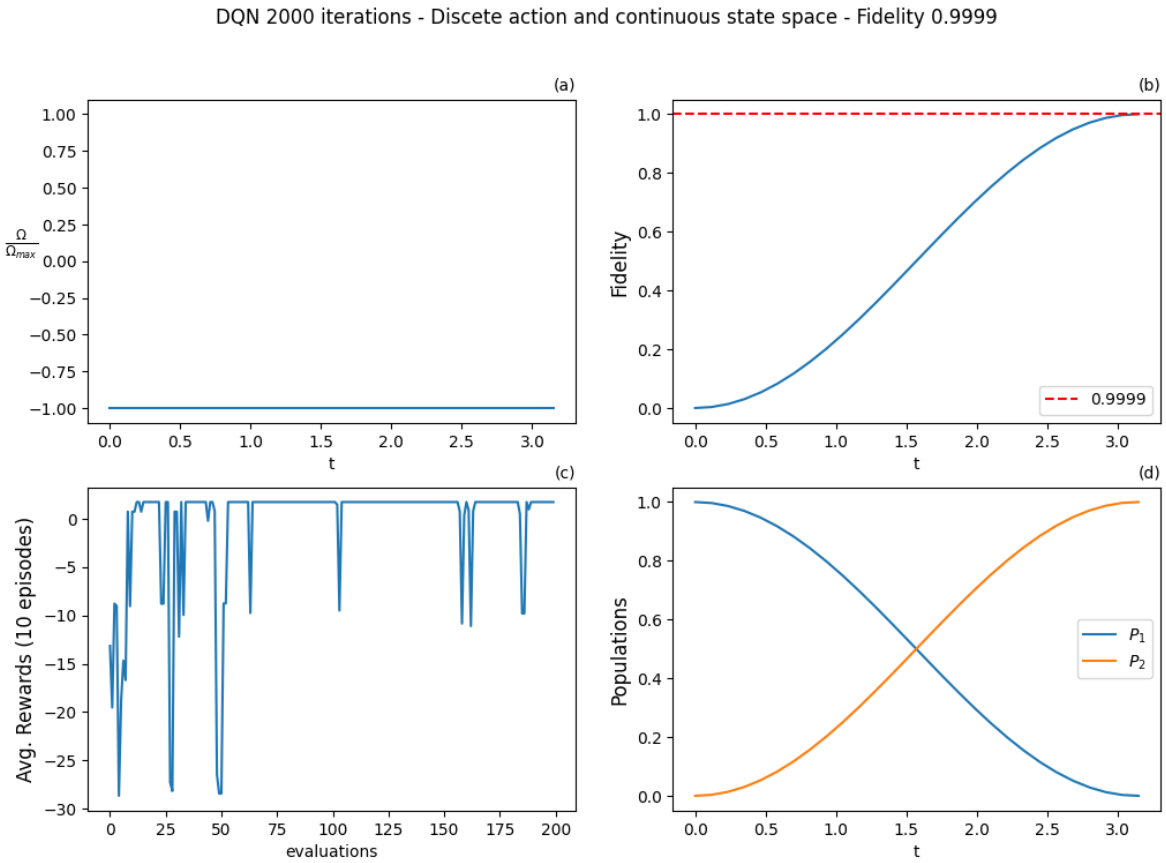


FIGURE 6.9: Results for DQN algorithm with 9 actions and continuous state space - Fidelity $>$ 0.9999: (a) Optimal normalized Rabi frequency $\Omega(t)$, (b) Fidelity (population of excited state $|2\rangle$), (c) Expected return (cumulative rewards) from the training episodes averaged from 10 episode samples, (d) Populations of states $|1\rangle$ and $|2\rangle$.

Continuous state space case produces the same results as the finite state space case, but as it is can be seen by comparing Figs. 6.6c and 6.7c with Figs. 6.8c and 6.9c, it experiences less variance during the training.

6.7.2.3 Double Deep Q-Network

Double DQN algorithm is also applied in the two different environments, with discrete action spaces with 7 discrete actions in both cases, and with finite and continuous state spaces. The parameters of the executions are available in the tables 6.5 and 6.6. In general, the results are the same as in the case of DQN. In both discrete and finite state space scenarios, the agent is able to discover the optimal policy from the training process. It is able to achieve fidelity 0.99 and 0.9999 with great success. $\Omega(t)$ in all executions attains the constant π-pulse form as it can be seen in the Figs. 6.10a, 6.11a, 6.12a, 6.13a. It seems that Double DQN algorithm experiences smaller variance during its training process

compared to DQN algorithm, as demonstrated in Figs. 6.10c, 6.11c, 6.12c, 6.13c.

Parameters	$\mathcal{F} = 0.99$	$\mathcal{F} = 0.9999$
Max time steps (N)	15	35
Ω_{max}	1	
End time (T)	$\frac{5}{\Omega_{max}}$	
Time step	$\frac{T}{N}$	
discount factor γ	0.99	
ϵ	0.1	
Detuning Δ	0	
Rabi frequency Ω	$\{-1, -\frac{3}{4}, -\frac{1}{2}, -\frac{1}{4}, 0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1\}$	
Actions $\delta\Omega$	$\{-2, -1, -\frac{1}{2}, -\frac{1}{4}, 0, \frac{1}{4}, \frac{1}{2}, 1, 2\}$	
Target fidelity	0.99	0.9999
Training Iterations	2000	2000
Hidden layers (2)	(100, 50)	(100, 100)
Learning rate	0.001	
Optimizer	Adam	

TABLE 6.5: Double DQN parameters - Discrete state space

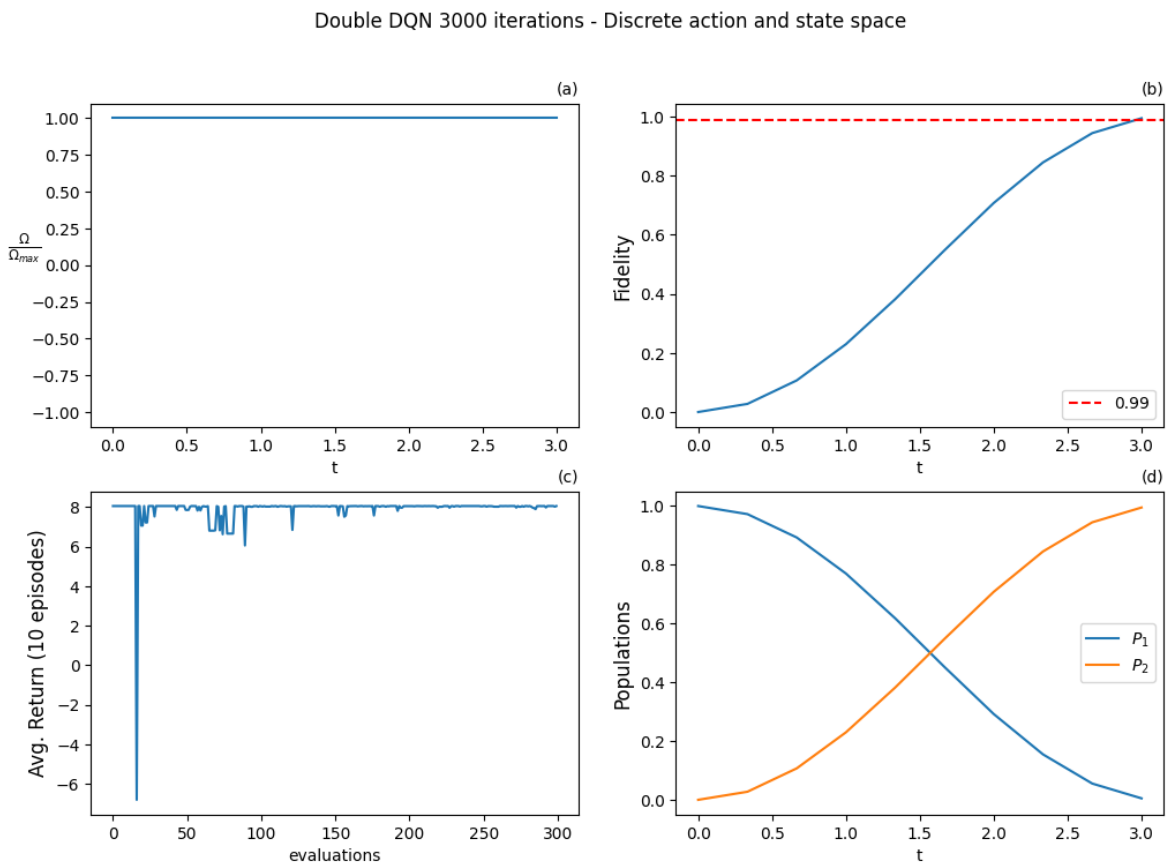


FIGURE 6.10: Results for Double-DQN algorithm with 9 actions and discrete state space: (a) Optimal normalized Rabi frequency $\Omega(t)$, (b) Fidelity (population of excited state $|2\rangle$), (c) Expected return (cumulative rewards) from the training episodes averaged from 10 episode samples, (d) Populations of states $|1\rangle$ and $|2\rangle$.

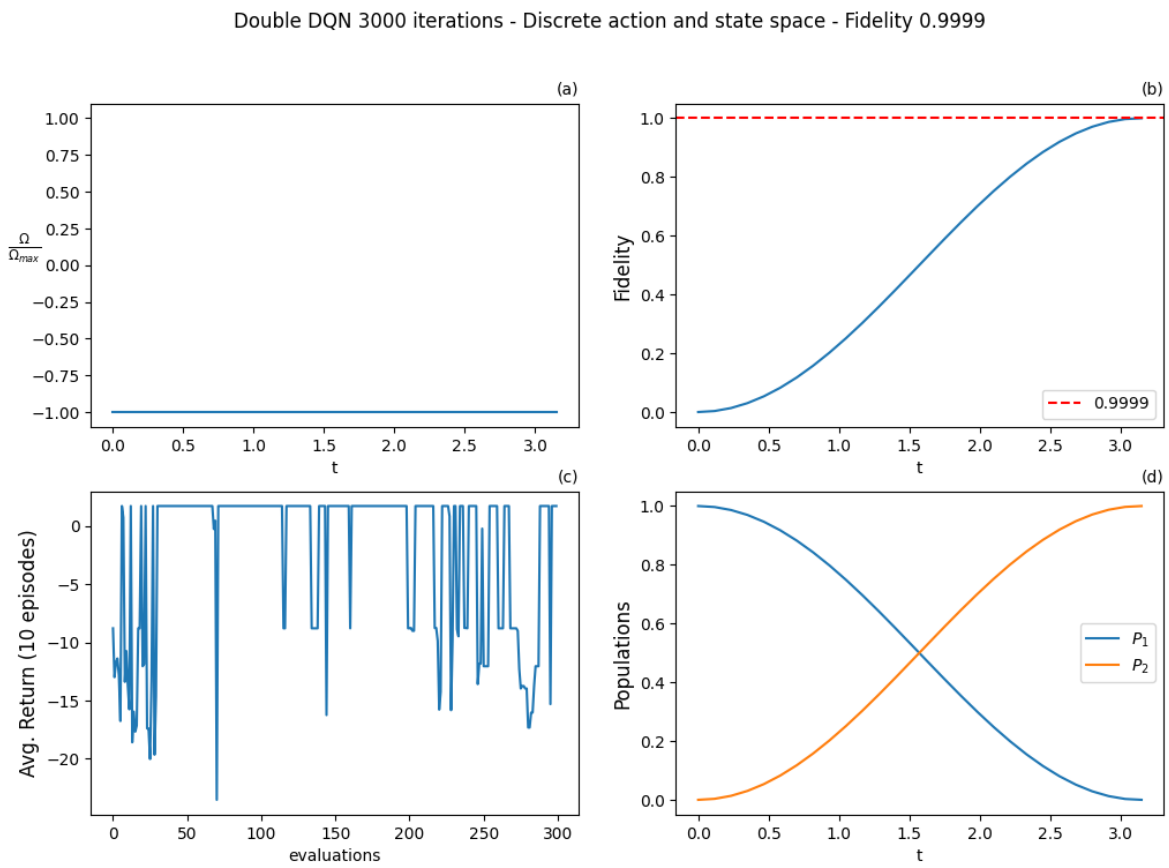


FIGURE 6.11: Results for Double-DQN algorithm with 9 actions and discrete state space - Fidelity > 0.9999: (a) Optimal normalized Rabi frequency $\Omega(t)$, (b) Fidelity (population of excited state $|2\rangle$), (c) Expected return (cumulative rewards) from the training episodes averaged from 10 episode samples, (d) Populations of states $|1\rangle$ and $|2\rangle$.

Parameters	$\mathcal{F} = 0.99$	$\mathcal{F} = 0.9999$
Max time steps (N)	15	35
Ω_{max}	1	
End time (T)	$\frac{5}{\Omega_{max}}$	$\frac{3.5}{\Omega_{max}}$
Time step	$\frac{T}{N}$	
discount factor γ	0.99	
ϵ	0.1	
Detuning Δ	0	
Rabi frequency Ω	$\{-1, -\frac{3}{4}, -\frac{1}{2}, -\frac{1}{4}, 0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1\}$	
Actions $\delta\Omega$	$\{-2, -1, -\frac{1}{2}, -\frac{1}{4}, 0, \frac{1}{4}, \frac{1}{2}, 1, 2\}$	
Target fidelity	0.99	0.9999
Training Iterations	3000	4000
Hidden layers (2)	(100, 50)	(100, 100)
Learning rate	0.001	
Optimizer	Adam	

TABLE 6.6: Double DQN parameters - Continuous state space

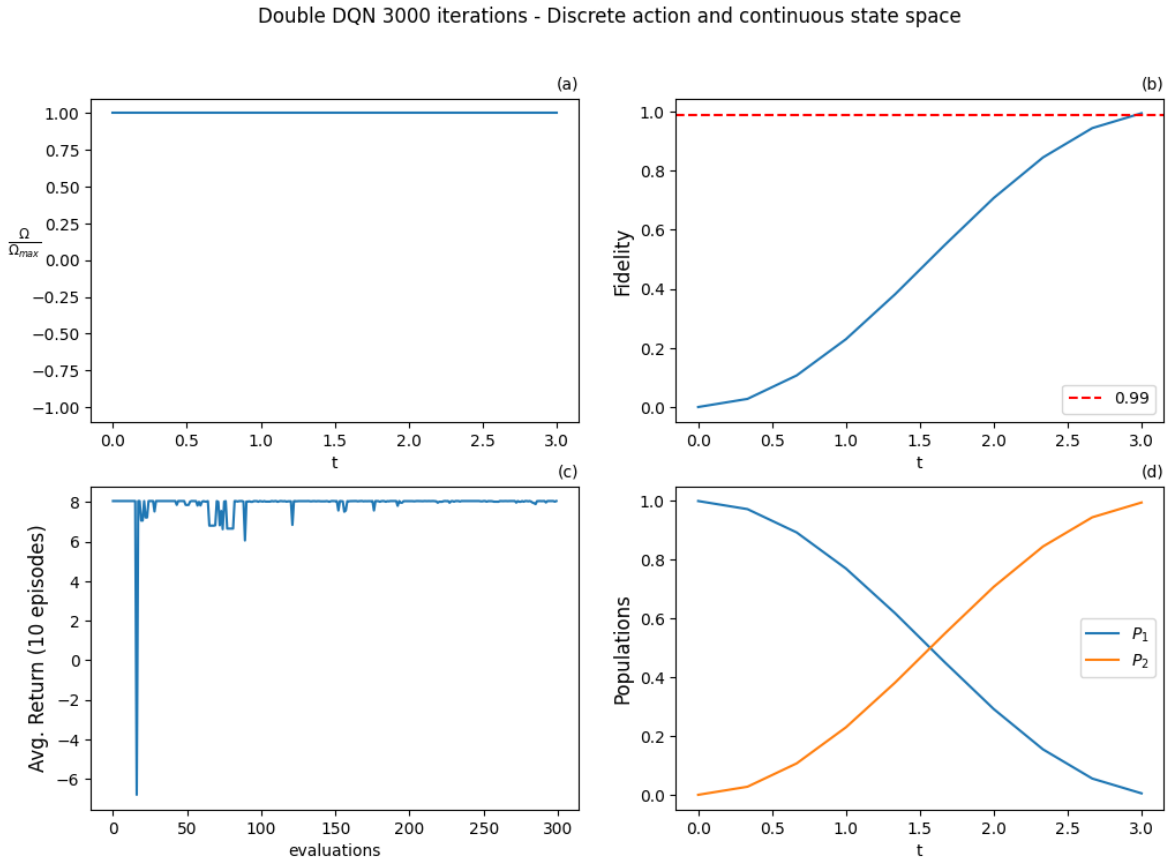


FIGURE 6.12: Results for Double-DQN algorithm with 9 actions and continuous state space: (a) Optimal normalized Rabi frequency $\Omega(t)$, (b) Fidelity (population of excited state $|2\rangle$), (c) Expected return (cumulative rewards) from the training episodes averaged from 10 episode samples, (d) Populations of states $|1\rangle$ and $|2\rangle$.

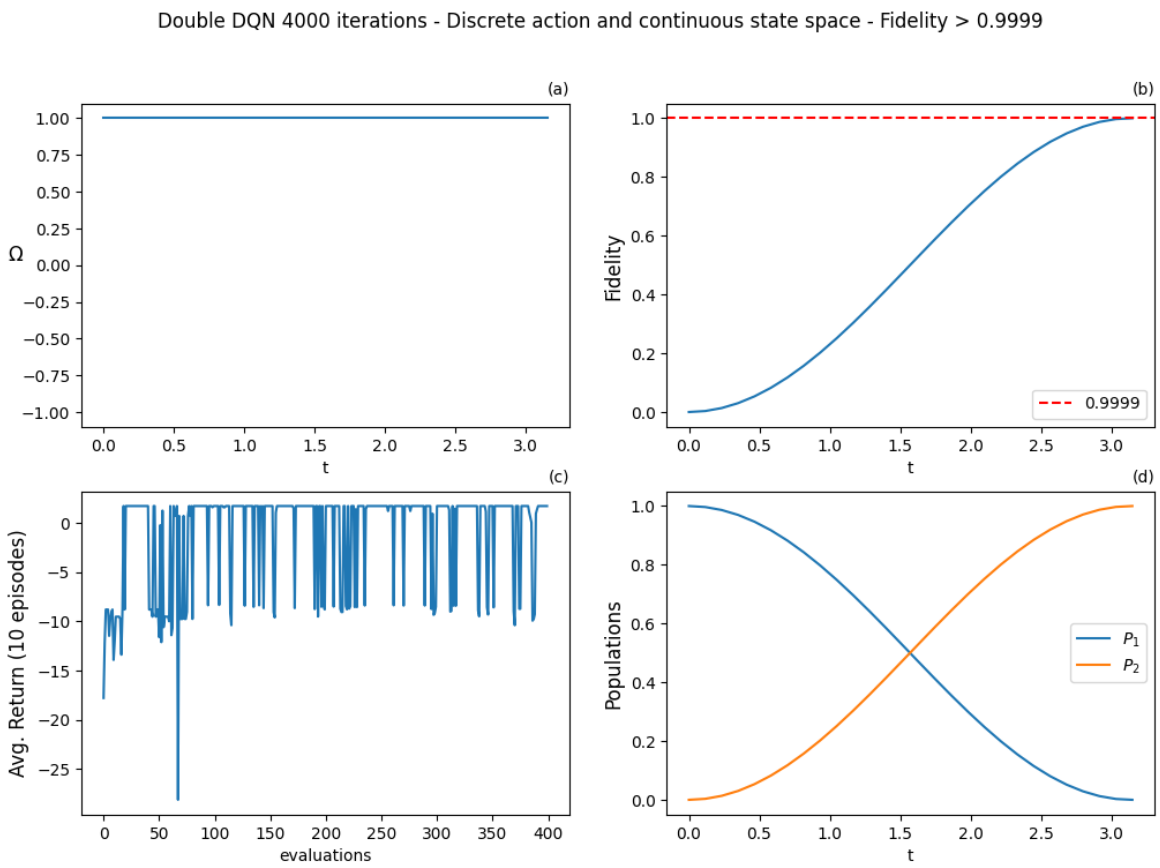


FIGURE 6.13: Results for Double-DQN algorithm with 9 actions and continuous state space - Fidelity > 0.9999: (a) Optimal normalized Rabi frequency $\Omega(t)$, (b) Fidelity (population of excited state $|2\rangle$), (c) Expected return (cumulative rewards) from the training episodes averaged from 10 episode samples, (d) Populations of states $|1\rangle$ and $|2\rangle$.

6.8 Policy Gradient methods

Policy gradient methods offer several advantages over value-based methods. The first algorithm that will be utilized to create an agent that generates (near-)optimal policies for the problem of state transfer is the REINFORCE algorithm. In fact the version with the baseline has be used. REINFORCE algorithm experiences great variance as a Monte Carlo method, although the baseline with the form of a neural network which approximates the value function, reduces the variance, improves the training process and makes the training faster.

6.8.1 REINFORCE with baseline

REINFORCE algorithm is a policy gradient method that uses theoretical results such as policy gradient theorem to establish improvement of policy and convergence to the optimal policy. It uses a critic neural network, representing the parameterized policy, which follows the performance gradient and is step by step improved approaching (near-)optimal policies. The baseline introduced is a value neural network that estimates the value of the input state. There are two architectures of policy Neural Networks used for the executions, one

for the discrete state space setup (5.5) and a second one for the continuous state space setup (5.6).

Parameters	$\mathcal{F} = 0.99$	$\mathcal{F} = 0.9999$
Max time steps (N)	15	35
Ω_{max}	1	
End time (T)	$\frac{5}{\Omega_{max}}$	
Time step	$\frac{T}{N}$	
discount factor γ	0.95	
Detuning Δ	0	
Rabi frequency Ω	$\{-1, -\frac{3}{4}, -\frac{1}{2}, -\frac{1}{4}, 0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1\}$	
Actions $\delta\Omega$	$\{-2, -1, -\frac{1}{2}, -\frac{1}{4}, 0, \frac{1}{4}, \frac{1}{2}, 1, 2\}$	
Target fidelity	0.99	0.9999
Training Iterations	2000	2000
Actor Hidden layers (2)	(100, 75)	(100, 75)
Value Hidden layers (2)	(100, 75)	(100, 75)
Learning rate	0.001	
Optimizer	Adam	

TABLE 6.7: REINFORCE with baseline parameters - Discrete action and state space

First using the MDP form with discrete action and state spaces (6.7), it was quite easy for the REINFORCE agent to find the optimal policy for the two level complete state transfer. From the average returns Fig.6.14 it is clear that even in early stages of training, the policy gradient direction finds the right path and parameterized policy becomes optimal giving the optimal π pulse. The agent is able to find the optimal policy even with fidelities higher than 0.9999. As it is shown in figure 6.15, the training process is less stable than the one one with less fidelity goals, but it is also able to converge to the optimal policy quite fast.

REINFORCE algorithm also succeeds to solve the MDP with discrete action space and continuous state space. The parameters of the environment and the agent are given in Table **??**. The agent is able to converge to the optimal policy as the training accumulates and to achieve fidelities up to 0.99 6.16b and 0.9999 6.17b.

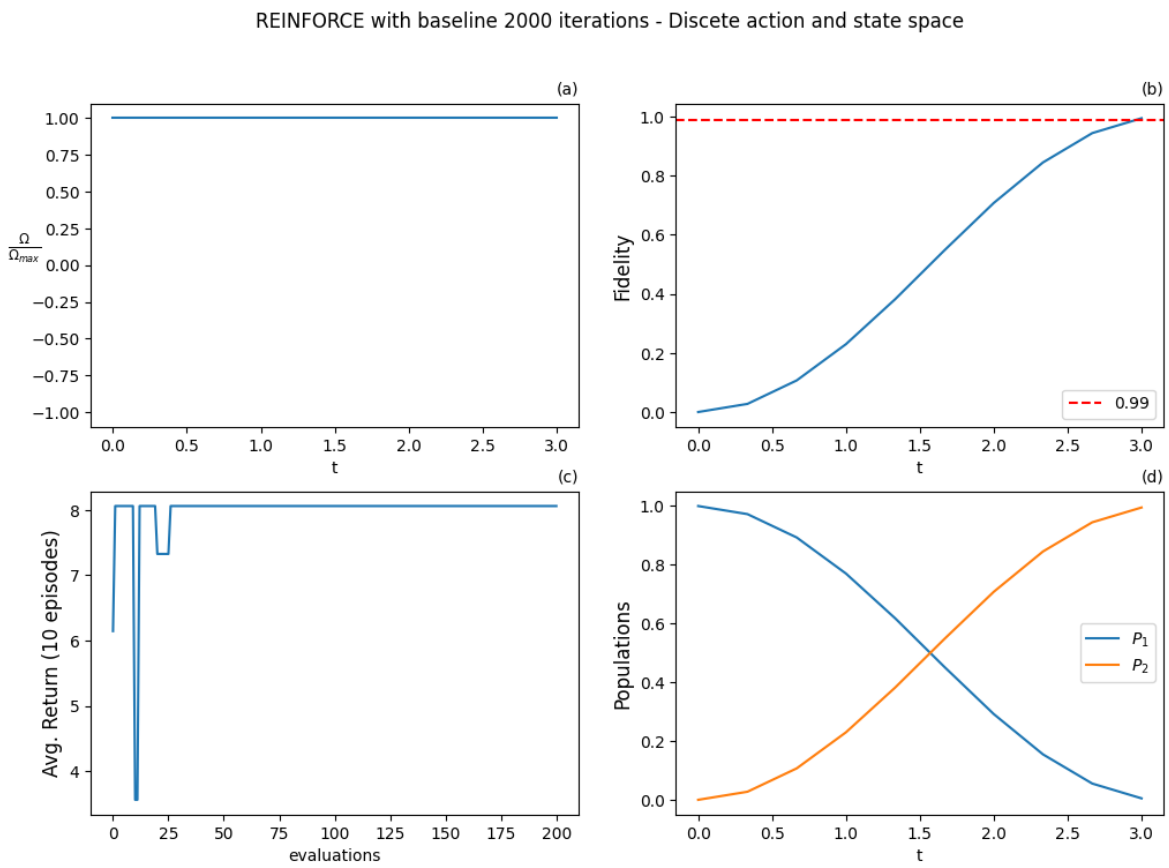


FIGURE 6.14: Results for REINFORCE algorithm with 9 actions and discrete state space: (a) Optimal normalized Rabi frequency $\Omega(t)$, (b) Fidelity (population of excited state $|2\rangle$), (c) Expected return (cumulative rewards) from the training episodes averaged from 10 episode samples, (d) Populations of states $|1\rangle$ and $|2\rangle$.

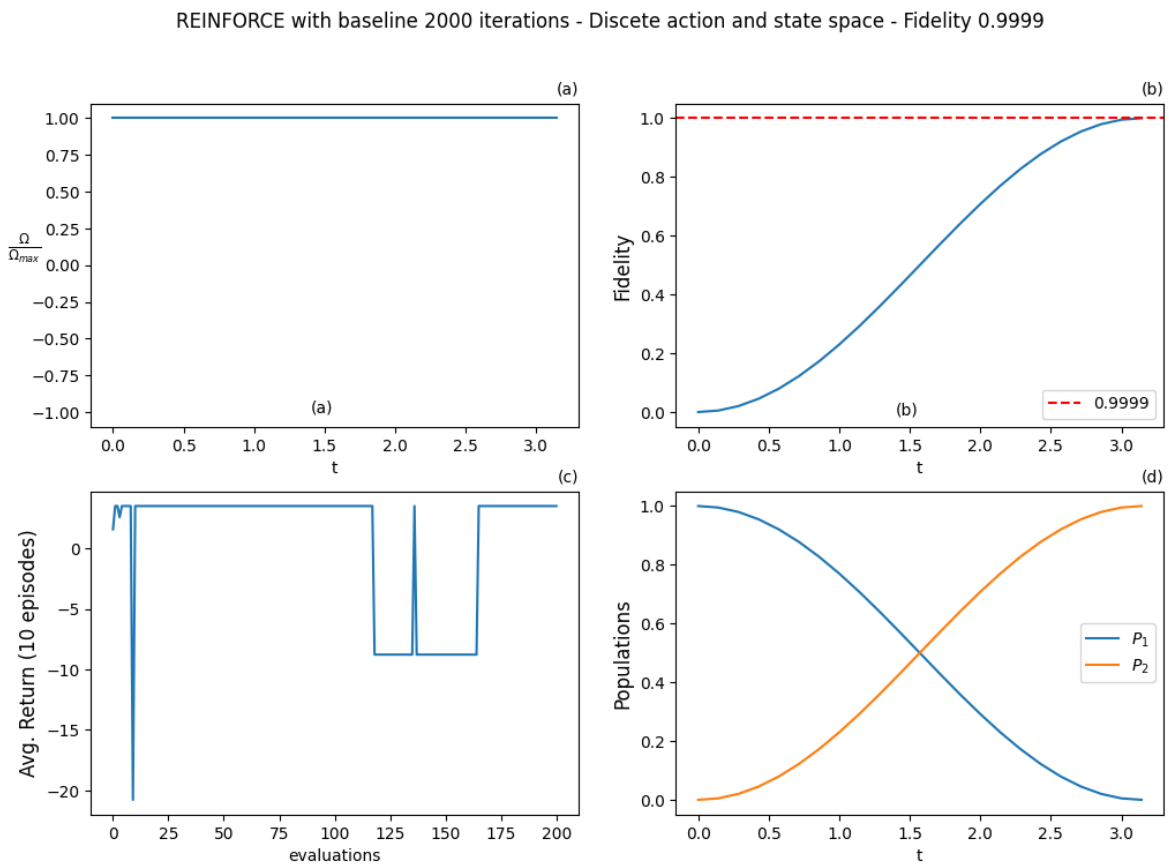


FIGURE 6.15: Results for REINFORCE algorithm with 9 actions and discrete state space - Fidelity > 0.9999: (a) Optimal normalized Rabi frequency $\Omega(t)$, (b) Fidelity (population of excited state $|2\rangle$), (c) Expected return (cumulative rewards) from the training episodes averaged from 10 episode samples, (d) Populations of states $|1\rangle$ and $|2\rangle$.

Parameters	$\mathcal{F} = 0.99$	$\mathcal{F} = 0.9999$
Max time steps (N)	15	35
Ω_{max}	1	
End time (T)	$\frac{5}{\Omega_{max}}$	
Time step	$\frac{T}{N}$	
discount factor γ	0.95	
Detuning Δ	0	
Rabi frequency Ω	$\{-1, -\frac{3}{4}, -\frac{1}{2}, -\frac{1}{4}, 0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1\}$	
Actions $\delta\Omega$	$\{-2, -1, -\frac{1}{2}, -\frac{1}{4}, 0, \frac{1}{4}, \frac{1}{2}, 1, 2\}$	
Target fidelity	0.99	0.9999
Training Iterations	2000	2000
Actor Hidden layers (2)	(100, 50)	(100, 75)
Value Hidden layers (2)	(50, 25)	(50, 25)
Learning rate	0.001	
Optimizer	Adam	

TABLE 6.8: REINFORCE with baseline parameters - Discrete action and continuous state space

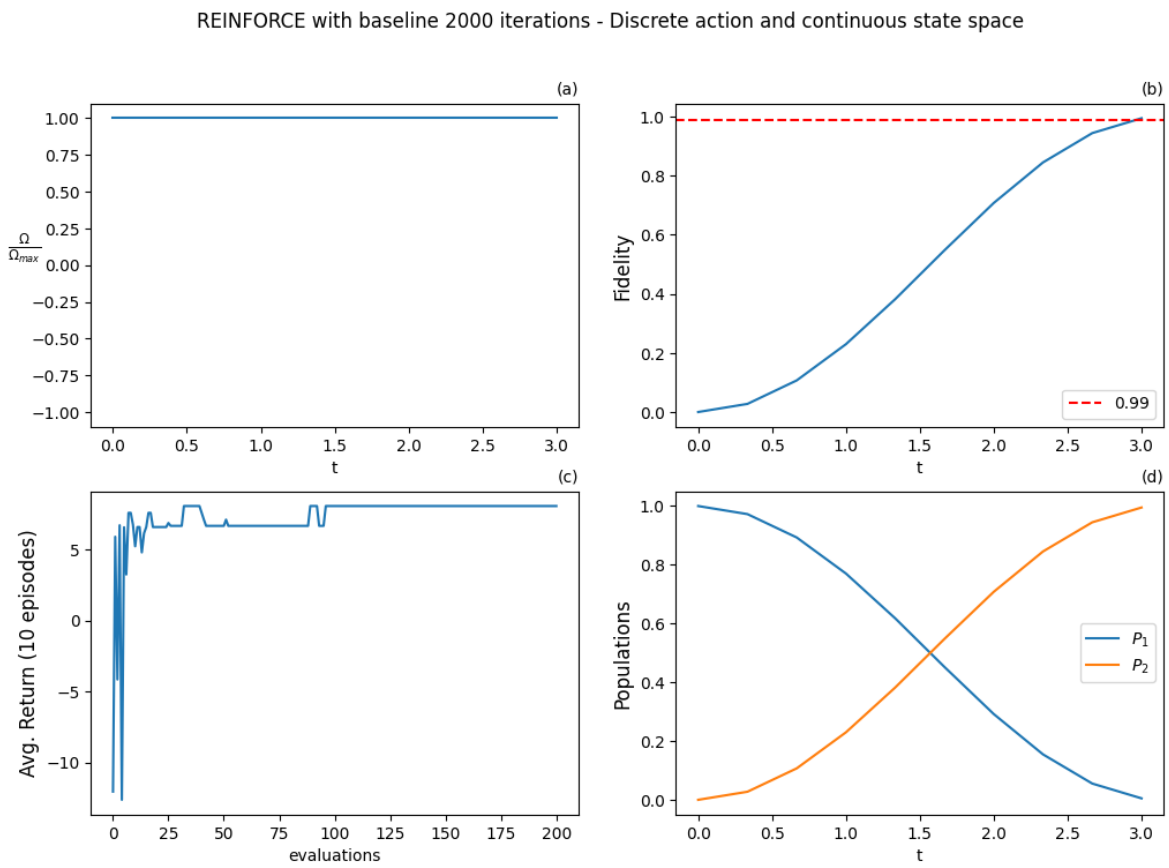


FIGURE 6.16: Results for REINFORCE algorithm with 9 actions and continuous state space: (a) Optimal normalized Rabi frequency $\Omega(t)$, (b) Fidelity (population of excited state $|2\rangle$), (c) Expected return (cumulative rewards) from the training episodes averaged from 10 episode samples, (d) Populations of states $|1\rangle$ and $|2\rangle$.

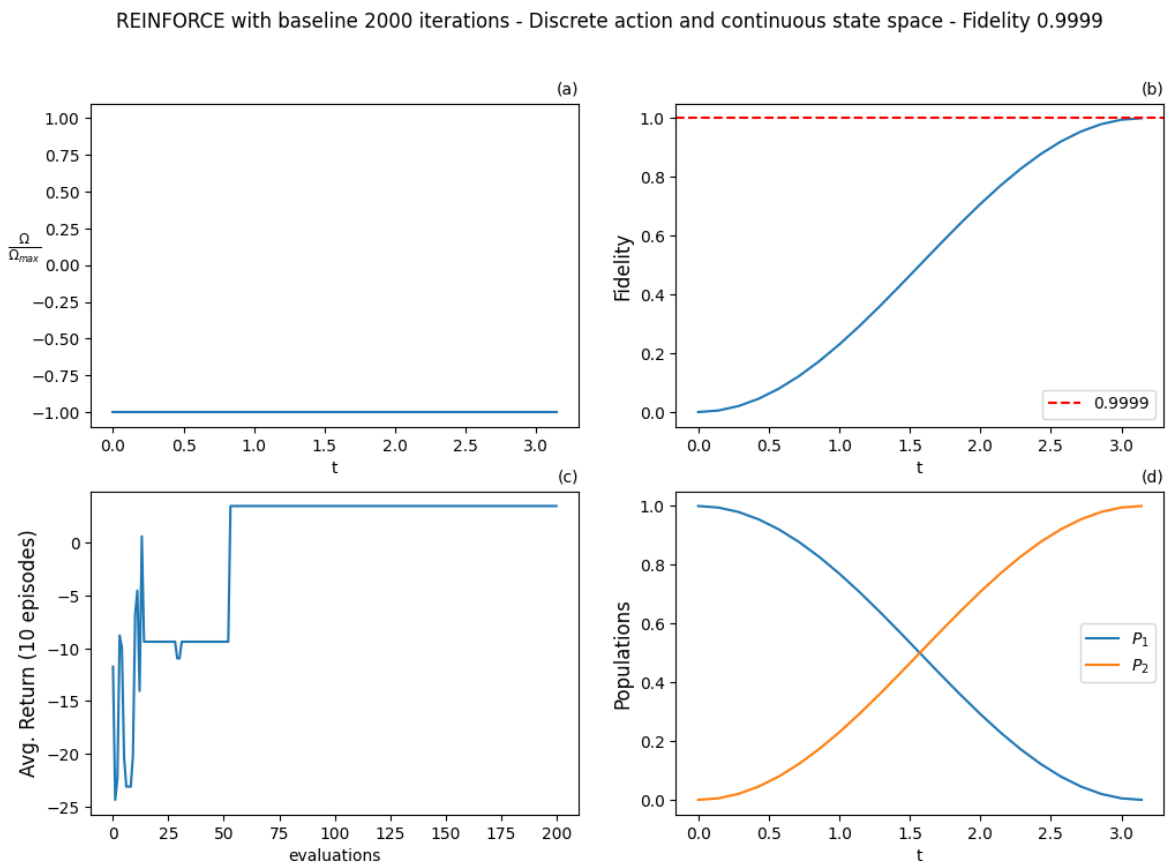


FIGURE 6.17: Results for REINFORCE algorithm with 9 actions and continuous state space - Fidelity > 0.9999: (a) Optimal normalized Rabi frequency $\Omega(t)$, (b) Fidelity (population of excited state $|2\rangle$), (c) Expected return (cumulative rewards) from the training episodes averaged from 10 episode samples, (d) Populations of states $|1\rangle$ and $|2\rangle$.

One of the advantages of policy gradients methods is their ability to handle continuous action spaces. So the MDP can be extended with continuous Rabi frequency, with the only restriction is that it must be bounded, with $\Omega \in [-\Omega_{max}, \Omega_{max}]$, where $\Omega_{max} = 1$. The parameters of MDP and the algorithm change as shown in table 6.10.

The same set of parameters as in 6.10 are used for the continuous action space with detuning Δ included in actions. Training process 6.19 seems less stable at the beginning, but with more training examples the agent is able to approach optimal behaviour and produce optimal pulse sequence for the complete state transfer.

Parameters	$\mathcal{F} = 0.99$	$\mathcal{F} = 0.9999$
Max time steps (N)	20	35
Ω_{max}	1	
End time (T)	$\frac{5}{\Omega_{max}}$	
Time step	$\frac{T}{N}$	
discount factor γ	0.99	
Detuning Δ	0	
Rabi frequency Ω	$\in [-\Omega_{max}, \Omega_{max}]$	
Actions $\delta\Omega$	$\in \mathbb{R}$	
Target fidelity	0.99	0.9999
Training Iterations	2000	2000
Actor Hidden layers (2)	(100, 50)	(100, 75)
Value Hidden layers (2)	(100, 75)	(100, 75)
Learning rate	0.001	
Optimizer	Adam	

TABLE 6.9: REINFORCE with baseline parameters - Continuous action and state space - Resonant case ($\Delta = 0$)

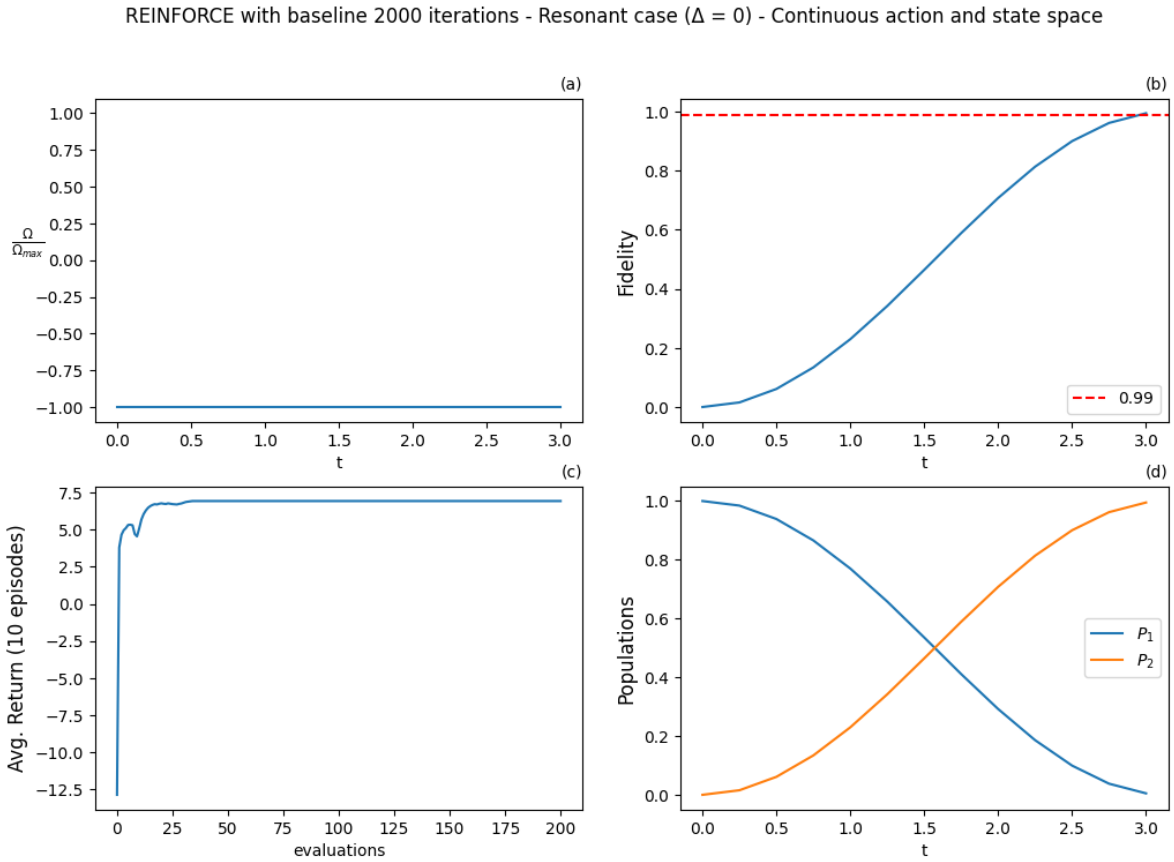


FIGURE 6.18: Results for REINFORCE algorithm with continuous action and state space - Resonant case ($\Delta = 0$): (a) Optimal normalized Rabi frequency $\Omega(t)$, (b) Fidelity (population of excited state $|2\rangle$), (c) Expected return (cumulative rewards) from the training episodes averaged from 10 episode samples, (d) Populations of states $|1\rangle$ and $|2\rangle$.

Parameters	$\mathcal{F} = 0.99$	$\mathcal{F} = 0.9999$
Max time steps (N)	20	30
Ω_{max}	1	
Δ_{max}	0.5	
End time (T)	$\frac{3.5}{\Omega_{max}}$	
Time step	$\frac{T}{N}$	
discount factor γ	0.99	
Detuning Δ	$\in [-\Delta_{max}, \Delta_{max}]$	
Rabi frequency Ω	$\in [-\Omega_{max}, \Omega_{max}]$	
Actions $\delta\Omega$	$\in \mathbb{R}$	
Target fidelity	0.99	0.9999
Training Iterations	3000	3000
Actor Hidden layers (2)	(100, 75)	(100, 75)
Value Hidden layers (2)	(75, 50)	(75, 50)
Learning rate	0.001	
Optimizer	Adam	

TABLE 6.10: REINFORCE with baseline parameters - Continuous action and state space

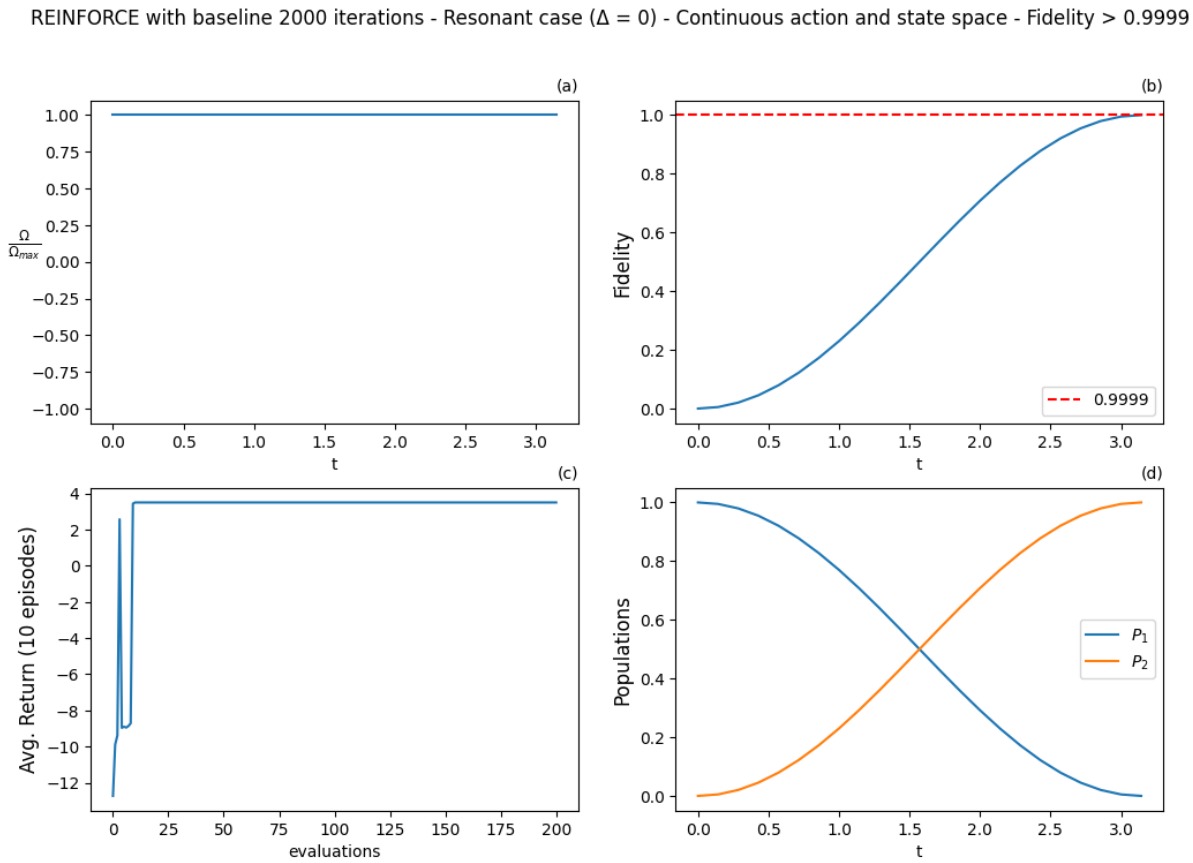


FIGURE 6.19: Results for REINFORCE algorithm with continuous action and state space - Resonant case ($\Delta = 0$) - Fidelity > 0.9999: (a) Optimal normalized Rabi frequency $\Omega(t)$, (b) Fidelity (population of excited state $|2\rangle$), (c) Expected return (cumulative rewards) from the training episodes averaged from 10 episode samples, (d) Populations of states $|1\rangle$ and $|2\rangle$.

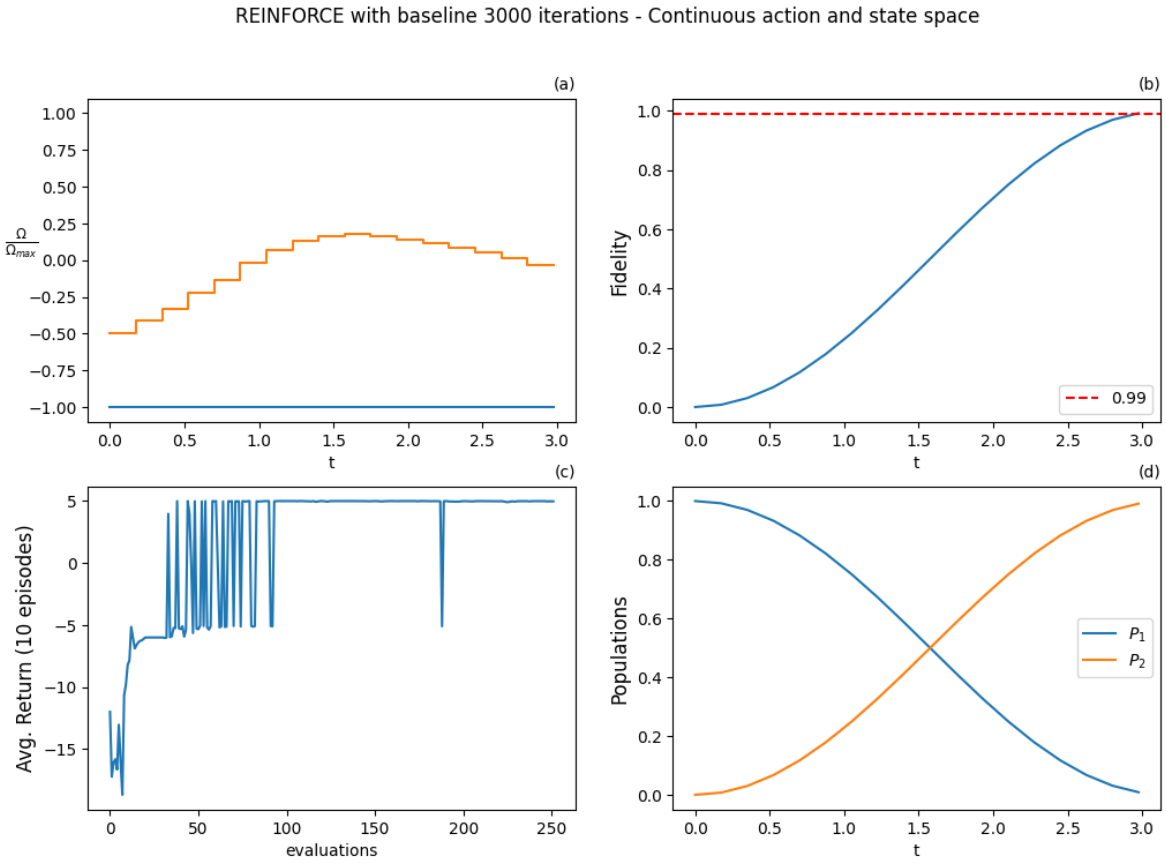


FIGURE 6.20: Results for REINFORCE algorithm with continuous action and state space: (a) Optimal normalized Rabi frequency $\Omega(t)$ and detuning $\Delta(t)$, (b) Fidelity (population of excited state $|2\rangle$), (c) Expected return (cumulative rewards) from the training episodes averaged from 10 episode samples, (d) Populations of states $|1\rangle$ and $|2\rangle$.

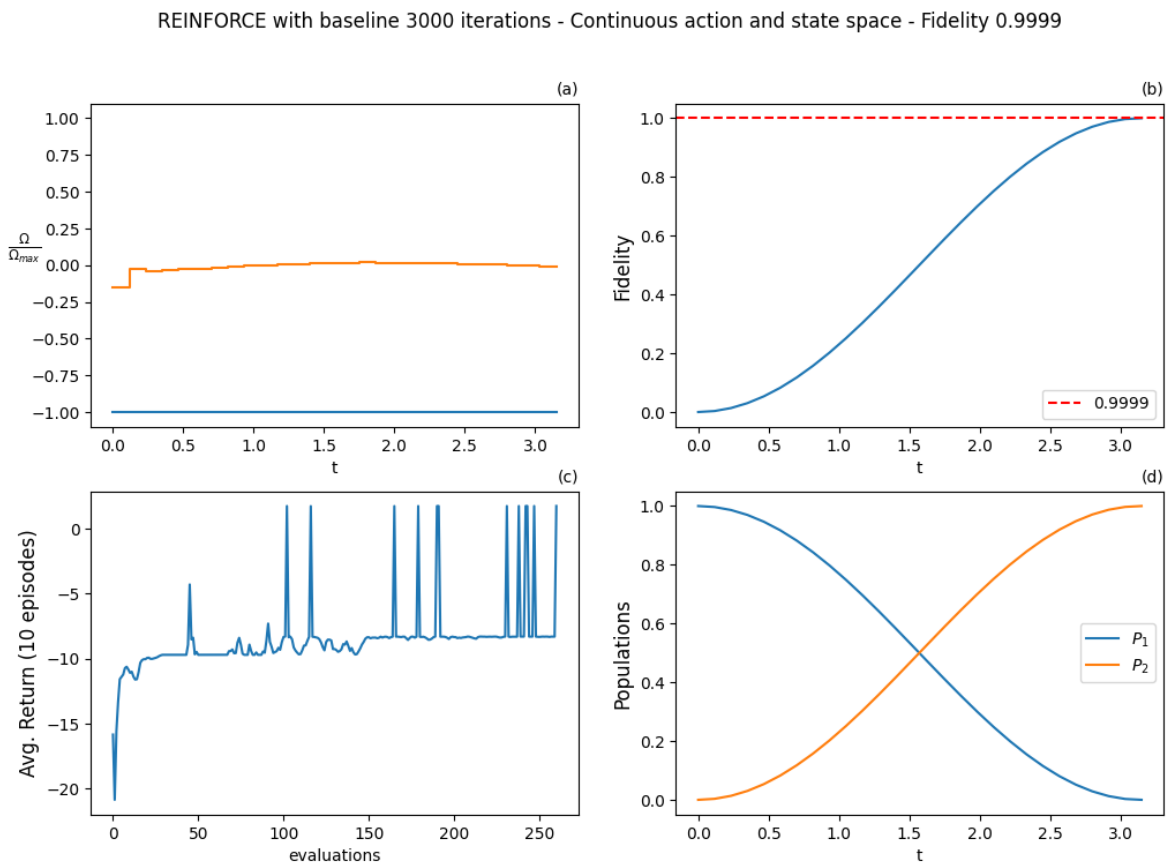


FIGURE 6.21: Results for REINFORCE algorithm with continuous action and state space - Fidelity > 0.9999: (a) Optimal normalized Rabi frequency $\Omega(t)$ and detuning $\Delta(t)$, (b) Fidelity (population of excited state $|2\rangle$), (c) Expected return (cumulative rewards) from the training episodes averaged from 10 episode samples, (d) Populations of states $|1\rangle$ and $|2\rangle$.

It is important to note that with policy gradient methods such as REINFORCE algorithm, the problem can be solved by using two controls, the Rabi frequency and the Detuning. There is more freedom for pulse shapes and no need to limit detuning to a constant value. The results show that the agent is able to produce (near-)optimal policies while using two controls. In fact, in the two control setup 6.20a, 6.21a, the two functions try to approach the resonant case, since the Rabi frequency takes the constant value Ω_{max} everywhere and the detuning approaches 0 value everywhere with small deviations, which cause some delay in the time that it will achieve the threshold fidelity.

6.9 Actor critic methods

Actor critic methods try to use the advantages of both the value based and policy gradient methods. Two actor-critic methods will be used in the state transfer problem.

- Proximal Policy Optimization Algorithms (PPO)
- Deep Deterministic Policy Gradient (DDPG)

Both methods are mainly used with continuous action and state spaces, so they will be both tested within the continuous state and action space MDP setup. Both algorithms

produce successful (near-)optimal policies in the resonant case ($\Delta = 0$). DDPG was not able to produce an (sub-)optimal policy in the two control case. PPO on the other hand was able to produce (near-)optimal pulses even in the two control case, attaining both 0.99 and 0.9999 fidelities.

6.9.1 Proximal Policy Optimization Algorithms

The PPO algorithm finds successful (near-) optimal policies for both the resonant configuration (Table 6.11, Figs. 6.24, 6.25) and the case with additional detuning control (Table 6.12, Figs. 6.24, 6.25), obtaining fidelities higher than 0.9999. In the resonant case the π-pulse is obtained, while in the two control case, there is a bang-bang modulation of the detuning between its minimum and maximum allowed values (orange line in Fig. 6.24a), which does not noticeably speed up the population inversion.

Parameters	$\mathcal{F} = 0.99$	$\mathcal{F} = 0.9999$
Max time steps (N)	15	30
Ω_{max}	1	
End time (T)	$\frac{5}{\Omega_{max}}$	
Time step	$\frac{T}{N}$	
Detuning Δ	0	
Rabi frequency Ω	$\in [-\Omega_{max}, \Omega_{max}]$	
Actions $\delta\Omega$	$\in \mathbb{R}$	
Target fidelity	0.99	0.9999
Training Iterations	1000	1500
Actor Hidden layers (2)	(100, 75)	(100, 75)
Value Hidden layers (2)	(100, 50)	(100, 50)
Learning rate	0.001	
Optimizer	Adam	

TABLE 6.11: PPO parameters - Continuous action and state space - Resonant case ($\Delta = 0$)

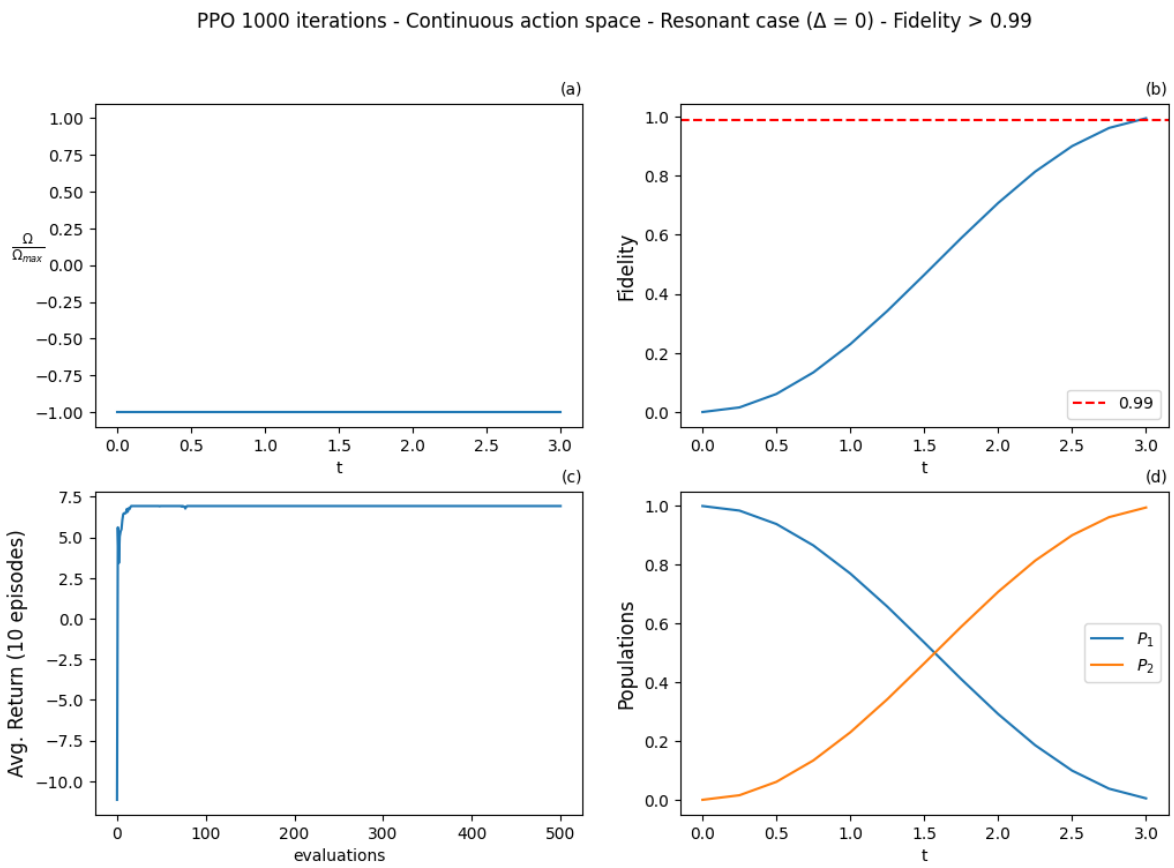


FIGURE 6.22: Results for PPO algorithm with continuous action and state space - Resonant Case ($\Delta = 0$): (a) Optimal normalized Rabi frequency $\Omega(t)$, (b) Fidelity (population of excited state $|2\rangle$), (c) Expected return (cumulative rewards) from the training episodes averaged from 10 episode samples, (d) Populations of states $|1\rangle$ and $|2\rangle$.

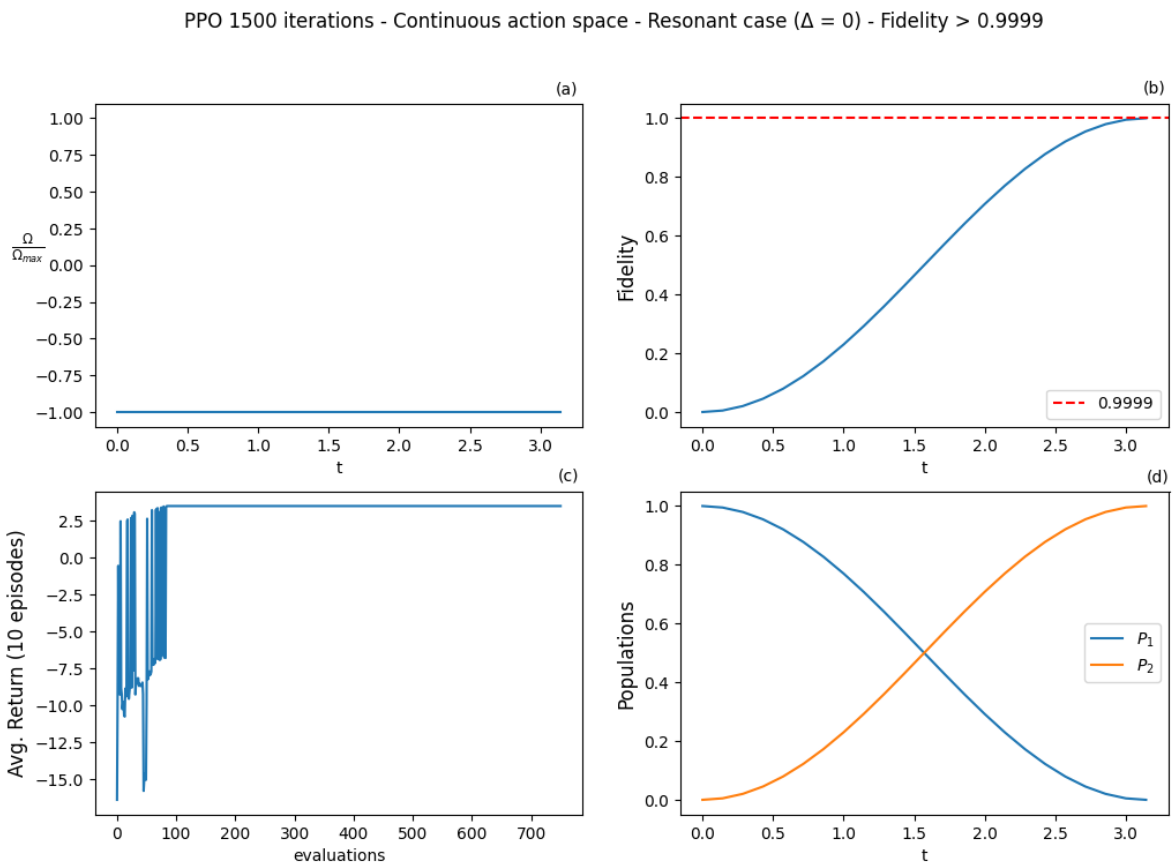


FIGURE 6.23: Results for PPO algorithm with continuous action and state space - Resonant Case ($\Delta = 0$) - Fidelity > 0.9999: (a) Optimal normalized Rabi frequency $\Omega(t)$, (b) Fidelity (population of excited state $|2\rangle$), (c) Expected return (cumulative rewards) from the training episodes averaged from 10 episode samples, (d) Populations of states $|1\rangle$ and $|2\rangle$.

Parameters	$\mathcal{F} = 0.99$	$\mathcal{F} = 0.9999$
Max time steps (N)	20	30
Ω_{max}	1	
Δ_{max}	0.5	
End time (T)	$\frac{4}{\Omega_{max}}$	$\frac{3.5}{\Omega_{max}}$
Time step	$\frac{T}{N}$	
Detuning Δ	$\in [-\Delta_{max}, \Delta_{max}]$	
Rabi frequency Ω	$\in [-\Omega_{max}, \Omega_{max}]$	
Actions $\delta\Omega$	$\in \mathbb{R}$	
Target fidelity	0.99	0.9999
Training Iterations	2000	3000
Actor Hidden layers (2)	(100, 75)	(100, 75)
Value Hidden layers (2)	(100, 50)	(100, 50)
Learning rate	0.001	
Optimizer	Adam	

TABLE 6.12: PPO parameters - Continuous action and state space

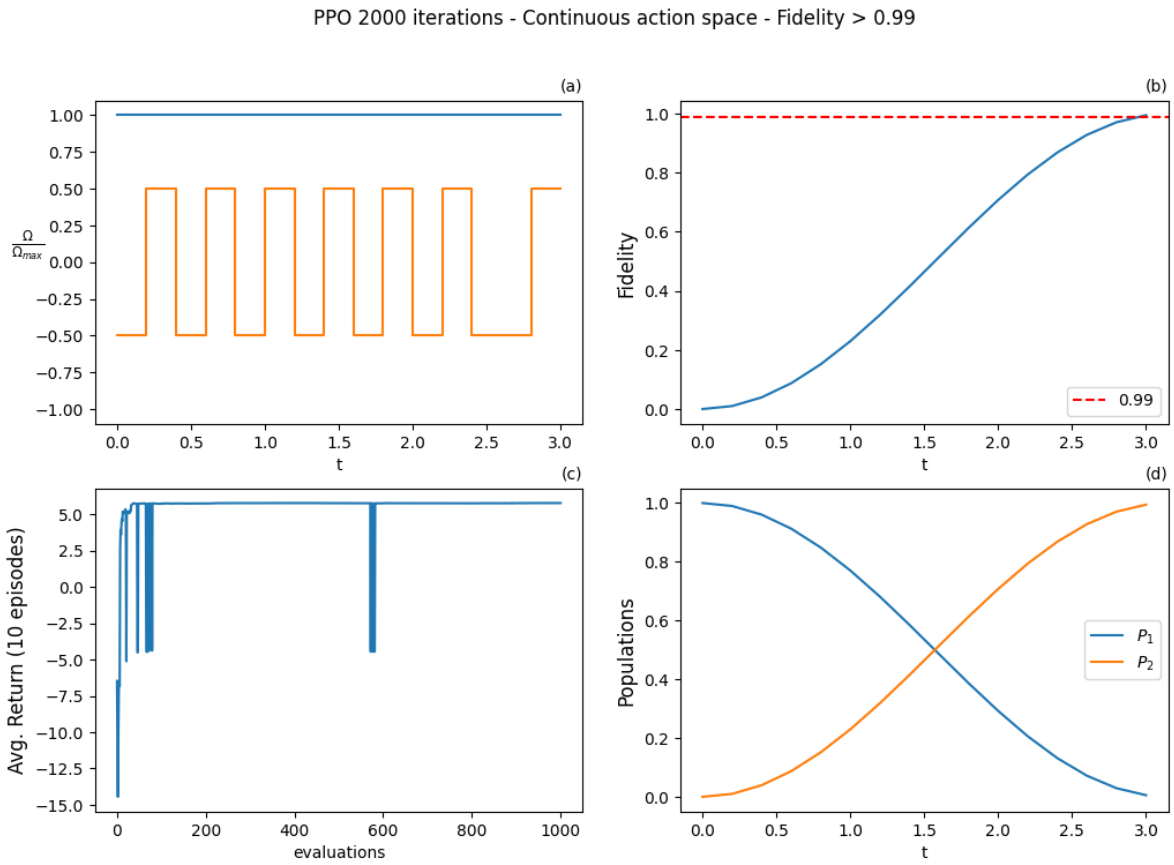


FIGURE 6.24: Results for PPO algorithm with continuous action and state space: (a) Optimal normalized Rabi frequency $\Omega(t)$ and detuning $\Delta(t)$, (b) Fidelity (population of excited state $|2\rangle$), (c) Expected return (cumulative rewards) from the training episodes averaged from 10 episode samples, (d) Populations of states $|1\rangle$ and $|2\rangle$.

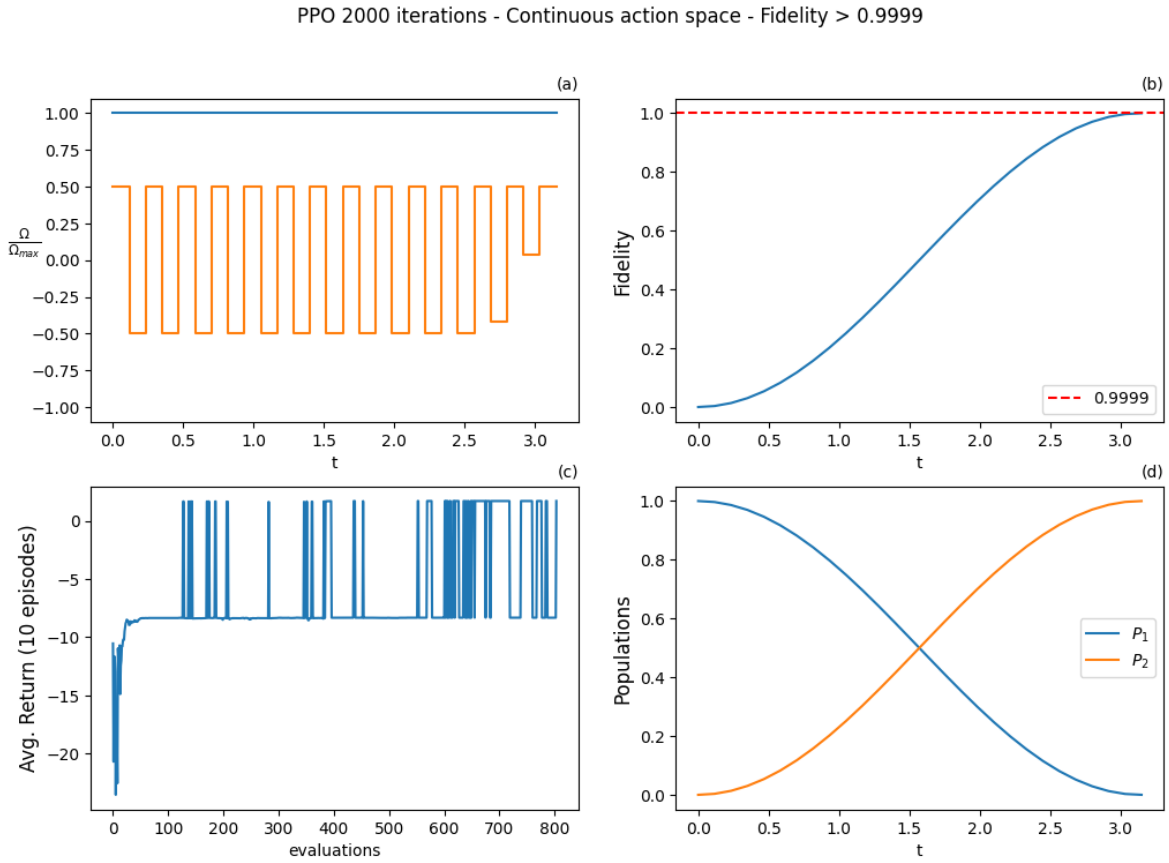


FIGURE 6.25: Results for PPO algorithm with continuous action and state space - Fidelity $>$ 0.9999: (a) Optimal normalized Rabi frequency $\Omega(t)$, (b) Fidelity (population of excited state $|2\rangle$), (c) Expected return (cumulative rewards) from the training episodes averaged from 10 episode samples, (d) Populations of states $|1\rangle$ and $|2\rangle$.

6.9.2 Deep Deterministic Policy Gradient

DDPG algorithm is able to solve the state transfer problem successfully in the resonant case, but it did not succeed to converge to a policy in the two control setup. As shown in Figs. 6.26, 6.26, the agent is able to produce the *pi*-shape Rabi frequency with low variance, although it did not succeed in any case to solve the MDP when the additional control of detuning Δ is introduced.

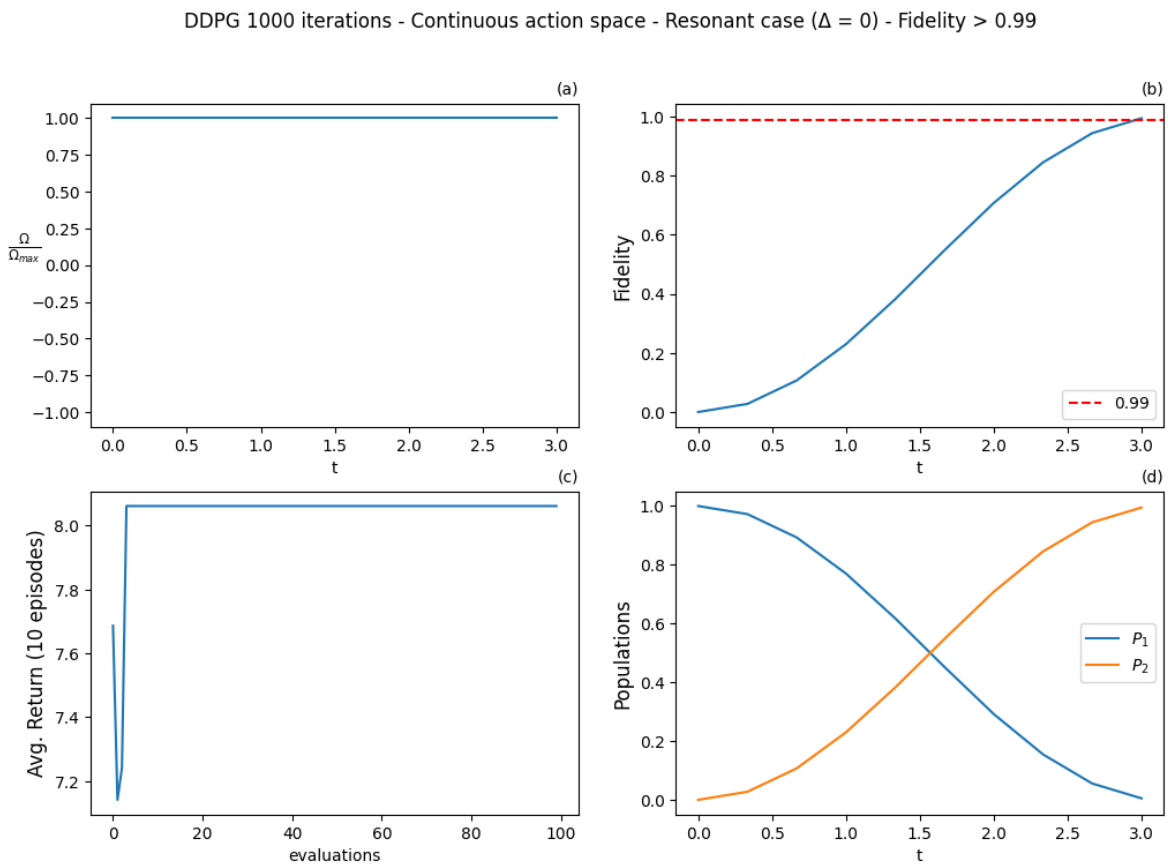


FIGURE 6.26: Results for DDPG algorithm with continuous action and state space - Resonant Case ($\Delta = 0$): (a) Optimal normalized Rabi frequency $\Omega(t)$, (b) Fidelity (population of excited state $|2\rangle$), (c) Expected return (cumulative rewards) from the training episodes averaged from 10 episode samples, (d) Populations of states $|1\rangle$ and $|2\rangle$.

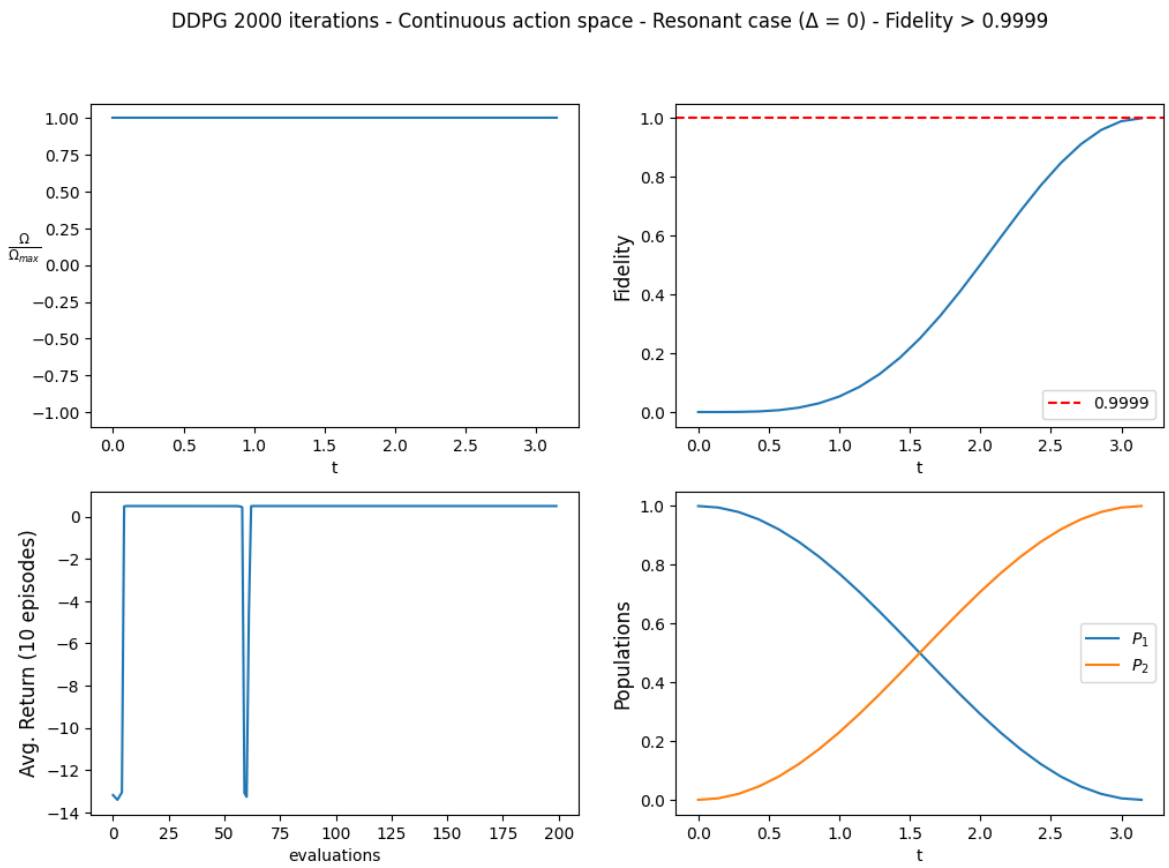


FIGURE 6.27: Results for DDPG algorithm with continuous action and state space - Resonant Case ($\Delta = 0$) - Fidelity > 0.9999: (a) Optimal normalized Rabi frequency $\Omega(t)$, (b) Fidelity (population of excited state $|2\rangle$), (c) Expected return (cumulative rewards) from the training episodes averaged from 10 episode samples, (d) Populations of states $|1\rangle$ and $|2\rangle$.

6.10 Trigonometric series optimization algorithm (TSOA)

It is always important to approach problems from different perspectives. In the state transfer problem, a different MDP formulation allows to define an optimization algorithm, named Trigonometric Series Optimization algorithm (TSOA), which can be solved with Reinforcement Learning. Formulation of the MDP requires continuous state and actions spaces. The suited methods are policy gradient or actor critic methods. The choice here is PPO algorithm, which is an actor critic algorithm. The parameters of the model are given by Table 6.13.

Parameters	$\mathcal{F} = 0.999$	$\mathcal{F} = 0.9999$
Simulation Time steps	300	
Ω_0	1	
End time (Ω_0^{-1})	3.15	
MDP Time step	1	
Actions	$\in \mathbb{R}^{2k}$	
Target fidelity	0.999	0.9999
Actor Hidden layers (2)	(100, 100, 50)	(100, 100, 50)
Value Hidden layers (2)	(75, 75, 50)	(75, 75, 50)
Learning rate	0.002	
Optimizer	Adam	

TABLE 6.13: Parameters for TSOA - PPO algorithm

6.10.1 Results with fidelity 0.999

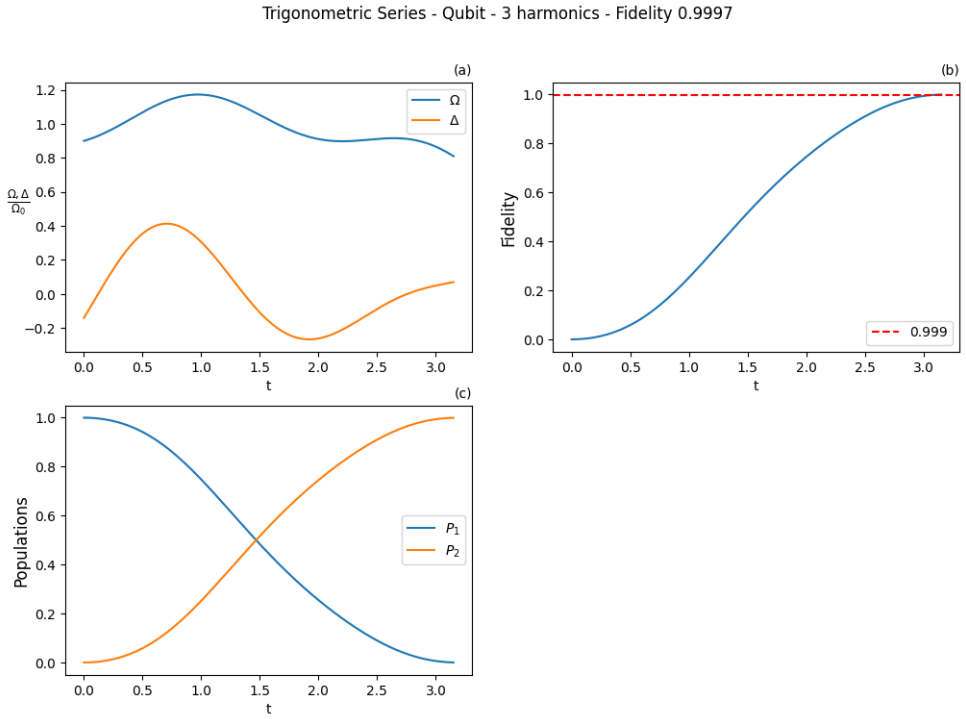


FIGURE 6.28: Results for TSOA - PPO algorithm - Fidelity > 0.999: (a) Optimal Rabi frequency $\Omega(t)$ and detuning $\Delta(t)$, (b) Fidelity (population of excited state $|2\rangle$), (c) Populations of states $|1\rangle$ and $|2\rangle$.

i	$\Omega : a_i$	$\Delta : b_i$
0	0.86193289	0.06261037
1	0.152197	-0.08266446
2	0.19752197	-0.11288086
3	-0.00489036	-0.09862257
4	-0.05913138	0.31972828
5	-0.10916921	-0.02207027
6	0.03801898	0.20511899

TABLE 6.14: Optimal trigonometric series parameters for fidelity > 0.999

6.10.2 Results with fidelity 0.9999

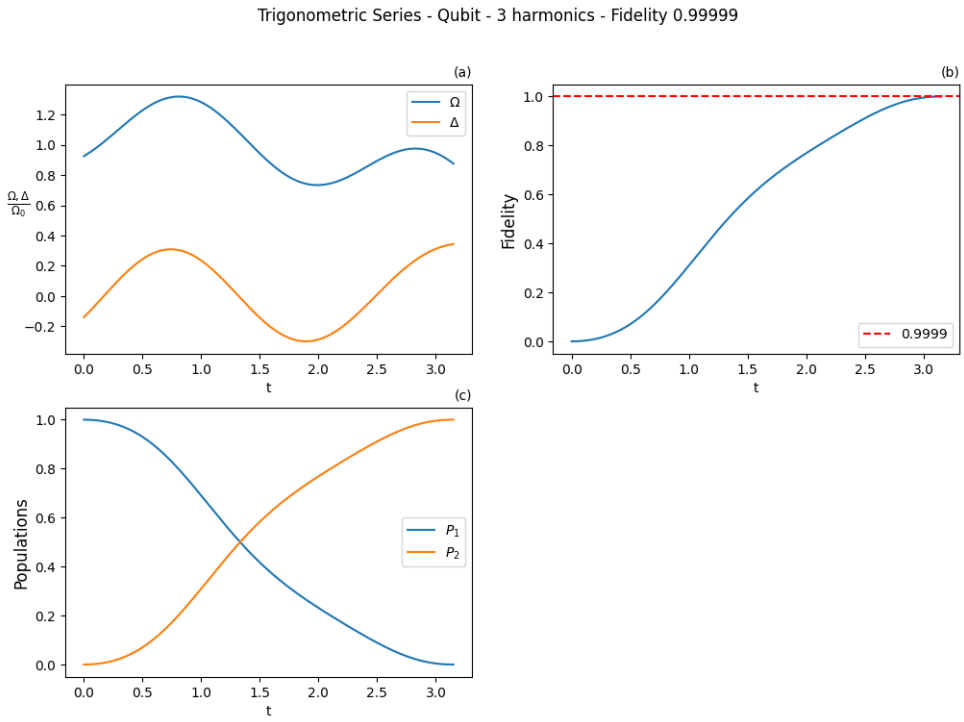


FIGURE 6.29: Results for TSOA - PPO algorithm - Fidelity > 0.9999: (a) Optimal Rabi frequency $\Omega(t)$ and detuning $\Delta(t)$, (b) Fidelity (population of excited state $|2\rangle$), (c) Populations of states $|1\rangle$ and $|2\rangle$.

i	$\Omega : a_i$	$\Delta : b_i$
0	0.87517912	0.07352462
1	0.20610334	-0.13624175
2	0.16243254	-0.09679438
3	0.02755164	0.02831635
4	-0.03201709	0.22532735
5	-0.18376116	-0.10505782
6	0.11923808	0.13957184

TABLE 6.15: Optimal trigonometric series parameters for fidelity > 0.9999

TSOA algorithm was able to produce a policy that represents that parameters of the Trigonometric Series. The controls produced by the process are smooth functions that can efficiently solve the problem and produce arbitrary fidelities. It is a matter of parameters tuning to produce better and better results, stressing the algorithm at its limit point. The main result is that with a few harmonics and quite small neural networks, the optimization process was able to produce smooth pulses that are (near-)optimal and solve the problem with the desired accuracy, as shown in Figs. 6.28, 6.29. Note that the duration of pulses in the TSOA algorithm is fixed apriori, which means that the system is not able to optimize the procedure with respect to timing, but given a time interval is able to obtain optimal smooth pulses attaining the target fidelity, within the algorithm and time discretization limitations.

Chapter 7

Conclusions and future directions

Both from current literature and from the work in this thesis, we can claim that RL methods are able to provide optimal or sub-optimal pulses which can be used to control qubit systems. There are different ways to construct the MDP process, with both finite and infinite dimensional action and state spaces. Each formulation provides different type of freedom and enables the use of even more RL methods as one goes from the finite to the infinite cases. The first and most important step is to clearly define the problem of quantum state transfer mathematically as an MDP. This is the key point that unlocks the use of RL methods to solve the problem.

Tabular methods can be used with limited applicability because they require a lot of training episodes to find the optimal pulses and achieve the high fidelities required by current quantum technologies. They are only restricted to finite action and state spaces. So, Deep RL methods are utilized for solving the defined MDPs. These methods are able to produce optimal pulses much easier and with higher accuracy than tabular methods. Their expressivity and generalization properties offered freedom in the state and action spaces definition, allowing even infinite continuous spaces. Infinite state spaces are more intuitive and closer to the state of the quantum systems and infinite action spaces provide more freedom in the choice of controls. Deep RL methods are able to produce policies and pulses within the current requirements and to achieve fidelities up to 0.9999 or higher. As it was expected, fidelities up to 0.9999 were difficult to achieve since more time steps and larger state spaces were required. Thus, more training and larger models were needed to be able to approximate the optimal policies. In most of the cases, RL methods successfully generated pulses which achieve the desired fidelity.

The problem was also formulated as another MDP, using truncated trigonometric series to express the controls. This different setup was solved by approximating the optimization process via the deep RL methods to obtain the optimal coefficients in the series. We called the created algorithm Trigonometric Series Optimization Algorithm (TSOA). The algorithm was able to produce smooth controls achieving fidelity higher than 0.9999, which might be easier to implement experimentally.

As future projects, a straight forward one it to test the applicability of the present work in creating other single qubit gates or general quantum states, like, for example, the creation of the Hadamard gate or the creation of a general superposition state. However, there are also more exciting prospects. RL methods offer more flexibility in comparison with other control methods, like optimal control. They can be easily adapted and they can be extended to quantum systems with three or more energy levels, encountered in modern quantum technologies. They also can be exploited for the simultaneous control of a collection of qubits with different frequencies, as well as the efficient control of a

qubit in the presence of noise. So, these problems are those that one may readily study as extension of the presented research. Of course, as quantum systems becomes more complex, the hyperparameters of the RL methods should be adapted and the deep neural networks need to be bigger, so they can approximate more complex models. Any problem that can be formulated as an MDP can be solved with the use of RL algorithms. More systematic analysis is needed when RL methods are applied in other problems such as quantum error correction [13], quantum gates creation [3], and quantum circuit optimization and these are additional future directions of the current research.

Appendix A

Appendix A

A.1 Proof of Policy improvement theorem

Policy improvement theorem Proof

Let π and π' be two deterministic policies, such that:

$$q_\pi(s, \pi'(s)) \geq v_\pi(s) \tag{A.1}$$

$$\begin{aligned}
v_\pi(s) &\leq q_\pi(s, \pi'(s)) \\
&= \mathbb{E}\Big[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = \pi'(s)\Big] \\
&= \mathbb{E}_{\pi'}\Big[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s\Big] \\
&\leq \mathbb{E}_{\pi'}\Big[R_{t+1} + \gamma q_\pi(s, \pi'(S_{t+1})) \mid S_t = s\Big] (A.1) \\
&= \mathbb{E}_{\pi'}\Big[R_{t+1} + \gamma(\mathbb{E}_{\pi'}\Big[R_{t+2} + \gamma v_\pi(S_{t+2}) \mid S_{t+1}, A_{t+1} = \pi'(S_{t+1})\Big]) \mid S_t = s\Big] \\
&\leq \mathbb{E}_{\pi'}\Big[R_{t+1} + \gamma R_{t+2} + \gamma^2 v_\pi(S_{t+2}) \mid S_t = s\Big] \\
&\leq \mathbb{E}_{\pi'}\Big[R_{t+1} + \gamma R_{t+2} + \gamma^2 v_\pi(S_{t+2}) + \gamma^3 v_\pi(S_{t+3}) \mid S_t = s\Big] \\
&\leq \mathbb{E}_{\pi'}\Big[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} \mid S_t = s\Big] \\
&\leq \mathbb{E}_{\pi'}\Big[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \ldots \mid S_t = s\Big] \\
&\leq v'_\pi(s)
\end{aligned}$$

So we have showed that $\forall s \in \mathcal{S} : v_\pi(s) \leq v'_\pi(s)$

A.2 Derivation of Total Hamiltonian

Total Hamiltonian

We define:

$$P_{nm} = |\psi_n\rangle\langle\psi_m|$$

Using the completeness identity:

$$P_{11} + P_{22} = \mathbb{I}$$

The total Hamiltonian reads

$$H(t) = H_0 + H_I(t)$$

where

$$H_0 = E_1 P_{11} + E_2 P_{22}$$

and E_1, E_2 are the eigenenergies of the free Hamiltonian operator H_0. We also find that the interaction term of the Hamiltonian after some simple operator algebra, can be written as:

$$H_I(t) = -d_{12}(P_{12} + P_{21})\epsilon(t)\cos[\omega t + \phi(t)]$$

where we have considered a centrosymmetric quantum system with zero permanent dipole matrix elements. Therefore, the electric dipole matrix elements are $d_{nm} = \langle\psi_n|\vec{d}\cdot\hat{\varepsilon}|\psi_m\rangle$ where $d_{11} = d_{22} = 0$ and $d_{12} = d_{21}$.
Defining the Rabi frequency as follows:

$$\Omega(t) = -\frac{d_{21}\epsilon(t)}{\hbar}$$

with $(\Omega \in \mathbb{R})$, then the interaction Hamiltonian reads:

$$H_I(t) = \hbar\Omega(t)\cos(\omega t)(P_{12} + P_{21})$$

In the interaction picture, the interaction Hamiltonian is transformed as follows:

$$H_I^{(i.p.)}(t) = e^{\frac{iH_0 t}{\hbar}} H_I(t) e^{\frac{-iH_0 t}{\hbar}} \tag{A.2}$$

If we substitute H_0 and $H_I(t)$ into (A.2), then:

$$\begin{aligned} H_I^{(i.p.)}(t) = \hbar\frac{\Omega(t)}{2}[e^{-i(\omega_0+\omega)t-i\phi(t)} + e^{-i(\omega_0-\omega)t+i\phi(t)}]P_{21} \\ +[e^{i(\omega_0+\omega)t+i\phi(t)} + e^{i(\omega_0-\omega)t-i\phi(t)}]P_{12} \end{aligned}$$

where $\omega_0 = (E_2 - E_1)/\hbar$. Under the RWA the counter-rotating terms $(e^{\pm i(\omega_0+\omega)t})$ can be omitted :

$$H_{I,RWA}^{(i.p.)}(t) = \hbar\frac{\Omega(t)}{2}[P_{21}e^{i\Delta_1 t+i\phi(t)} + P_{12}e^{-i\Delta_1 t-i\phi(t)}] \tag{A.3}$$

where $\Delta_1 = \omega - \omega_0$ is called the bare detuning of the system.

Total Hamiltonian (Continued)

The arbitrary vector can be written as

$$|\psi(t)\rangle = a_1(t)|\psi_1\rangle + a_2(t)|\psi_2\rangle$$

From the TDSE we obtain

$$\begin{cases} i\dot{a_1}(t) = \frac{\Omega(t)}{2} e^{i\Delta_1 t + i\phi(t)} a_2(t) \\ i\dot{a_2}(t) = \frac{\Omega(t)}{2} e^{-i\Delta_1 t - i\phi(t)} a_1(t) \end{cases}$$

which with a unitary transformation or a simple change of variables to new probability amplitudes $b_1(t), b_2(t)$, reduces to the new probability amplitude equations

$$\begin{cases} i\dot{b_1}(t) = \frac{\Delta(t)}{2} b_1(t) + \frac{\Omega(t)}{2} b_2(t) \\ i\dot{b_2}(t) = \frac{\Omega(t)}{2} b_1(t) - \frac{\Delta(t)}{2} b_2(t) \end{cases}$$

where $\Delta(t) = \omega - \omega_0 + \dot{\phi}(t)$ is the time-dependent detuning.
Introducing $\hbar$ in all terms, the last system gives the following Hamiltonian, that we used in the present study, and which is common in many quantum technology applications:

$$H_{RWA} = \frac{\hbar}{2} \begin{pmatrix} \Delta(t) & \Omega(t) \\ \Omega(t) & -\Delta(t) \end{pmatrix} \tag{A.4}$$

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org. URL: https://www.tensorflow.org/.

[2] Zheng An, Hai-Jing Song, Qi-Kai He, and DL Zhou. Quantum optimal control of multilevel dissipative quantum systems with reinforcement learning. *Physical Review A*, 103(1):012404, 2021.

[3] Zheng An and DL Zhou. Deep reinforcement learning for quantum gate control. *Europhysics Letters*, 126(6):60002, 2019.

[4] Claudio Bonizzoni, Mirco Tincani, Fabio Santanni, and Marco Affronte. Machine-learning-assisted manipulation and readout of molecular spin qubits. *Physical Review Applied*, 18(6):064074, 2022.

[5] Ugo Boscain, Mario Sigalotti, and Dominique Sugny. Introduction to the pontryagin maximum principle for quantum optimal control. *PRX Quantum*, 2(3):030203, 2021.

[6] Jonathon Brown, Pierpaolo Sgroi, Luigi Giannelli, Gheorghe Sorin Paraoanu, Elisabetta Paladino, Giuseppe Falci, Mauro Paternostro, and Alessandro Ferraro. Reinforcement learning-enhanced protocols for coherent population-transfer in three-level quantum systems. *New Journal of Physics*, 23(9):093035, 2021.

[7] Marin Bukov, Alexandre GR Day, Dries Sels, Phillip Weinberg, Anatoli Polkovnikov, and Pankaj Mehta. Reinforcement learning in different phases of quantum control. *Physical Review X*, 8(3):031086, 2018.

[8] Raphaël Couturier, Etienne Dionis, Stéphane Guérin, Christophe Guyeux, and Dominique Sugny. Characterization of a driven two-level quantum system by supervised learning. *Entropy*, 25(3):446, 2023.

[9] Anna Dawid, Julian Arnold, Borja Requena, Alexander Gresch, Marcin Płodzień, Kaelan Donatella, Kim A Nicoli, Paolo Stornati, Rouven Koch, Miriam Büttner, et al. Modern applications of machine learning in quantum sciences. *arXiv preprint arXiv:2204.04198*, 2022.

[10] Thomas Degris, Martha White, and Richard S Sutton. Off-policy actor-critic. *arXiv preprint arXiv:1205.4839*, 2012.

[11] Yongcheng Ding, Yue Ban, José D Martín-Guerrero, Enrique Solano, Jorge Casanova, and Xi Chen. Breaking adiabatic quantum control with deep learning. *Physical Review A*, 103(4):L040401, 2021.

[12] Domenico d'Alessandro. *Introduction to quantum control and dynamics*. CRC press, 2021.

[13] Thomas Fösel, Petru Tighineanu, Talitha Weiss, and Florian Marquardt. Reinforcement learning with neural networks for quantum feedback. *Physical Review X*, 8(3):031084, 2018.

[14] Luigi Giannelli, Pierpaolo Sgroi, Jonathon Brown, Gheorghe Sorin Paraoanu, Mauro Paternostro, Elisabetta Paladino, and Giuseppe Falci. A tutorial on optimal control and reinforcement learning methods for quantum technologies. *Physics Letters A*, 434:128054, 2022.

[15] Michael Goerz, Daniel Basilewitsch, Fernando Gago-Encinas, Matthias G Krauss, Karl P Horn, Daniel M Reich, and Christiane Koch. Krotov: A python implementation of krotov's method for quantum optimal control. *SciPost physics*, 7(6):080, 2019.

[16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[17] Ivo Grondman, Lucian Busoniu, Gabriel AD Lopes, and Robert Babuska. A survey of actor-critic reinforcement learning: Standard and natural policy gradients. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(6):1291–1307, 2012.

[18] D. Guéry-Odelin, A. Ruschhaupt, A. Kiely, E. Torrontegui, S. Martínez-Garaot, and J. G. Muga. Shortcuts to adiabaticity: Concepts, methods, and applications. *Reviews of Modern Physics*, 91(4):045001, 2019.

[19] Hado Hasselt. Double q-learning. *Advances in neural information processing systems*, 23, 2010.

[20] J Robert Johansson, Paul D Nation, and Franco Nori. Qutip: An open-source python framework for the dynamics of open quantum systems. *Computer Physics Communications*, 183(8):1760–1772, 2012.

[21] Vijay Konda and John Tsitsiklis. Actor-critic algorithms. *Advances in neural information processing systems*, 12, 1999.

[22] Mario Krenn, Jonas Landgraf, Thomas Foesel, and Florian Marquardt. Artificial intelligence and machine learning for quantum technologies. *Physical Review A*, 107(1):010101, 2023.

[23] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[24] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[25] Wenjie Liu, Bosi Wang, Jihao Fan, Yebo Ge, and Mohammed Zidan. A quantum system control method based on enhanced reinforcement learning. *Soft Computing*, 26(14):6567–6575, 2022.

[26] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[27] Murphy Yuezhen Niu, Sergio Boixo, Vadim N Smelyanskiy, and Hartmut Neven. Universal quantum control through deep reinforcement learning. *npj Quantum Information*, 5(1):33, 2019.

[28] Iris Paparelle, Lorenzo Moro, and Enrico Prati. Digitally stimulated raman passage by deep reinforcement learning. *Physics Letters A*, 384(14):126266, 2020.

[29] Riccardo Porotti, Dario Tamascelli, Marcello Restelli, and Enrico Prati. Coherent transport of quantum states by deep reinforcement learning. *Communications Physics*, 2(1):61, 2019.

[30] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[31] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International Conference on Machine Learning*, pages 1889–1897. PMLR, 2015.

[32] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[33] Bruce W Shore. *Manipulating quantum structures using laser pulses*. Cambridge University Press, 2011.

[34] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *International Conference on Machine Learning*, pages 387–395. Pmlr, 2014.

[35] VV Sivak, A Eickbusch, H Liu, B Royer, I Tsioutsios, and MH Devoret. Model-free quantum control with reinforcement learning. *Physical Review X*, 12(1):011059, 2022.

[36] D Stefanatos and E Paspalakis. A shortcut tour of quantum control methods for modern quantum technologies. *Europhysics Letters*, 132(6):60001, 2021.

[37] Dionisis Stefanatos and Emmanuel Paspalakis. Efficient generation of the triplet bell state between coupled spins using transitionless quantum driving and optimal control. *Physical Review A*, 99(2):022327, 2019.

[38] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[39] John Tsitsiklis and Benjamin Van Roy. Analysis of temporal-diffference learning with function approximation. *Advances in neural information processing systems*, 9, 1996.

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

[41] Nikolay V Vitanov, Andon A Rangelov, Bruce W Shore, and Klaas Bergmann. Stimulated raman adiabatic passage in physics, chemistry, and beyond. *Reviews of Modern Physics*, 89(1):015006, 2017.

[42] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8:279–292, 1992.

[43] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8:229–256, 1992.

[44] Xiao-Ming Zhang, Zezhu Wei, Raza Asad, Xu-Chen Yang, and Xin Wang. When does reinforcement learning stand out in quantum control? a comparative study on state preparation. *npj Quantum Information*, 5(1):85, 2019.