Using Mixture Density Networks to Model Continuous Action Priors

WITS
UNIVERSITY

Sicelukwanda Zwane

Supervisor: Dr. Benjamin Rosman

Co-supervisor: Dr. Pravesh Ranchod

School of Computer Science and Applied Mathematics

University of the Witwatersrand

A dissertation submitted for the degree

Master of Science

March 2019

Acknowledgements

I would like to thank my supervisor Dr. Benjamin Rosman, without whom this research would not have been possible. I am particularly grateful for his patience towards me and for being my compass throughout my MSc despite his busy schedule. I have always found his energy contagious and consider it an honour to have worked under his supervision.

I would like to thank my family for their never-ending support, for always believing in me and allowing me the freedom to pursue my dreams.

I would also like to extend my gratitude towards the Council for Scientific and Industrial Research (CSIR) and the Mobile Autonomous Intelligent Systems department for financially supporting my MSc and funding training opportunities (workshops and conferences), both locally and internationally, which helped add more direction to my research. I am also grateful to the department for allowing me access to their robot platforms as well as the computational resources they provided during my studentship.

Lastly, I would like to thank the RAIL Lab (Robotics, Autonomous Intelligence and Learning Lab) at the University of the Witwatersrand for giving me a platform to discuss and share ideas related to my research. This has greatly improved my ability to communicate my ideas.

Abstract

Given enough data and access to an environment, model-free deep reinforcement learning algorithms allow for direct unsupervised behaviour learning in an environment without relying on a model of the environment's dynamics. As such, these algorithms have typically been favoured when learning tasks defined in continuous environments such as robotics where such models are hard to come by. Despite their success here, these approaches still suffer from high sample complexity and other drawbacks imposed by the use of the function approximators they rely on. This includes poor generalisation across tasks, poor initial performance and generally unsafe learning procedures due to inherent randomness in the learning process where actions sampled from random policies are executed for the purpose of exploration in the environment.

In this work we address these issues by presenting a method for transferring knowledge from multiple, previously solved tasks (source tasks) to new target tasks in the same continuous environment. Equipping successive agents with this body of prior knowledge allows for easier acquisition of subsequent behaviours in the given continuous environment. We also empirically show that our proposed method "mixture density network action priors" demonstrates improved safety as well as improved learning compared to a standard model-free deep reinforcement learning agent in continuous environments.

Contents

List of Figures

Chapter 1

Introduction

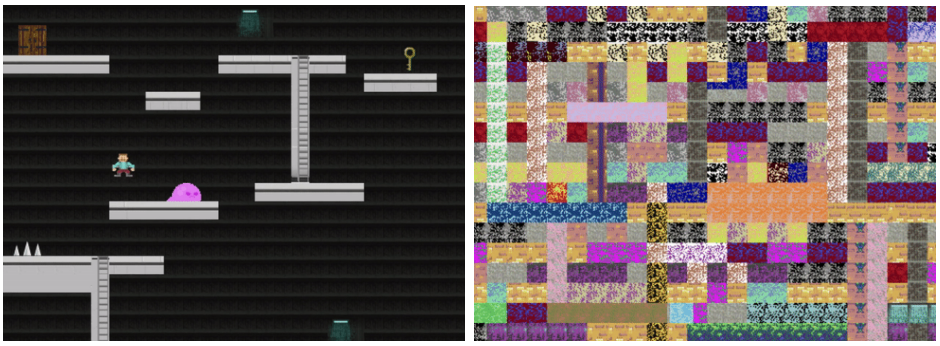
1.1 Motivation

Machine learning methods such deep reinforcement learning have shown great promise and versatility in enabling complex behaviour learning in robotics. However, these methods typically require a large magnitude of data in order to converge to optimal, yet sufficiently general solutions. This is largely due to the fact that it is difficult to equip a robot with prior knowledge when presented with a task. As such, new tasks have to be learned from scratch through extensive data collection. However, collecting this data in robotics is a slow and expensive process as most robots have short battery life and require constant monitoring which places a limit on the amount of data that can be recorded in a single instance. In some cases, this data collection process requires the robot to operate undisturbed for long periods of time [Finn et al., 2016, Levine et al., 2018], which may cause wear and tear on robot hardware.

Compared to robots, humans require far fewer interactions with the environment when learning new tasks. This can be attributed to their ability to exploit prior knowledge gained from previously solved tasks. This body of prior knowledge is used to constrain exploration, heavily reducing the amount of interaction time with the environment required to solve the task. Dubey et al. [2018] confirmed this observation by investigating the effect of such priors on human performance when playing video games. In the study, the authors employed randomly selected humans and distributed them across the original video game and versions of the video game that were modified to mask prior knowledge such as the concept of free space, ladders can be climbed, the

visual similarity of objects to real world counterparts, etc (Figure 1.1a). It was found that human players took longer to learn how to play the video game as more visual prior information was masked, thus highlighting the significance of having access to prior knowledge when learning.

Similar studies also found that humans develop strong prior knowledge about specific objects, game rules and observational learning experience in video games to construct theory-like representations [Tsividis et al., 2017] or "start-up software" [Lake et al., 2017] which can be transferred to similar scenarios, allowing humans to learn how to play video games more rapidly. These findings suggest that humans employ a transfer learning mechanism to facilitate learning new tasks.



(a) Original video game

(b) Game with masked visual similarity.

Figure 1.1: Video game used to investigate the effect of prior knowledge in humans when learning to play video games [Dubey et al., 2018].

Given this evidence, we are interested in implementing a transfer learning mechanism to facilitate robot learning via deep reinforcement learning.

Although there have been successful attempts at achieving transfer learning for deep reinforcement learning agents, implementing transfer in robotics still remains a challenge.

1.2 Research Problem

Deep reinforcement learning algorithms are effective methods for learning optimal action selection strategies for a given environment. However, the continuous, high-dimensional and noisy nature of data in robotics environments makes it difficult to apply these algorithms. Unlike in discrete environments, optimal actions for every possible robot configuration cannot be explicitly stored. Instead, deep reinforcement learning algorithms rely on function approximators such as deep neural networks to represent these state-action mappings.

The function approximators used in deep reinforcement learning are typically randomly initialised. This results in pseudo-random control policies that obtain poor initial performance on the target task, and are unsafe to execute on safety-critical systems.

Due to the infinite size of continuous environments, it is not feasible to try all possible state-action combinations during learning. As such, good exploration strategies are crucial for finding the optimal solution. However, the standard paradigm for exploration in deep reinforcement learning algorithms is to alternate between executing randomly generated actions (exploration) and executing actions sampled from the current partially–trained control policy (exploitation). While executing random actions can be good for exploration, it also poses a safety risk as it can lead agents to harmful states; possibly causing wear and tear or damage to the physical agent.

In this work we use transfer learning (particularly action transfer) to address issues of sample inefficiency (long training times), poor initial performance and safety in deep reinforcement learning algorithms for continuous environments. Thus making deep reinforcement learning algorithms more practical for physical systems such as robotics, where safety is a major concern.

1.3 Research Objectives

As a step towards applying deep reinforcement learning algorithms to physical, continuous environments such as robotics, we aim to address issues of sample inefficiency, poor initial performance and safety through developing a framework for performing sample efficient and safer deep reinforcement learning through knowledge reuse in the continuous setting.

1.4 Contributions

In this work we make novel improvements to an existing multi-task transfer learning framework for discrete environments [Rosman and Ramamoorthy, 2012], allowing it to function in continuous environments. We use mixture density networks to learn a multi-modal prior knowledge representation from a set of source tasks which is used as an action selection strategy for achieving safer exploration in the learning process of new tasks. Furthermore, we empirically show that incorporating prior knowledge into deep reinforcement learning algorithms in this way can lead to benefits such as improved asymptotic performance, safer policy training as well as a jump-start in initial performance on the target task.

1.5 Dissertation Outline

This dissertation is arranged as follows:

- Chapter 2 introduces fundamental concepts used in this work, such as the reinforcement learning problem as well common approaches for learning tasks in discrete environments.
- Chapter 3 explains how function approximators (particularly deep neural networks) can be used to scale discrete reinforcement learning algorithms to continuous environments under the deep reinforcement learning framework.
- Chapter 4 discusses common approaches in multi-task transfer learning where the goal is to reuse previously obtained solutions from solving prior tasks in reinforcement to improve learning and generalisation in deep reinforcement learning.
- Chapter 5 presents our solution: "mixture density network action priors", which extends a previous framework [Rosman and Ramamoorthy, 2012] for doing multi-task transfer learning in discrete settings – into continuous settings using mixture density networks.
- Chapter 6 contains the experiment results and related discussions that show that continuous action priors are a viable technique for performing multi-task transfer in continuous environments.
- Chapter 7 summarises the conclusions of this study and lists possible directions for future work.

Chapter 2

Reinforcement Learning

In this chapter we provide a brief introduction to reinforcement learning in discrete environments and set the mood for later chapters, which discuss: how to scale reinforcement learning to continuous environments (chapter 3), how reinforcement learning can be improved through knowledge reuse (chapter 4), and how we implement multi-task transfer for reinforcement learning in continuous environments (chapter 5).

In section 2.1, we discuss the components that make up the reinforcement learning problem according to Sutton and Barto [1998]. This includes the Markov Decision Process – the environment for which we learn the optimal policy, as well as how value functions can be used to specify the optimal policy. We explore two main approaches for learning the optimal policy, namely value-based reinforcement learning (section 2.3) and policy-based reinforcement learning (section 2.4). We then discuss common approaches to solving the exploration-exploitation trade-off (section 2.5) and lastly summarise the chapter in section 2.6.

2.1 The Reinforcement Learning Problem

Reinforcement learning is a field of machine learning where agents[1] learn how to behave optimally in some target environment according to some objective. This objective is typically encoded using a numerical reward or feedback signal the agent receives after executing an action based on its configuration in the environment.

[1]Decision-making entities

Finding the optimal behaviour then becomes a sequential decision-making problem where agents have to take actions that maximise the sum of these rewards.

Unlike in supervised and unsupervised learning, where training data for the task in question is presented as input to the learning process, in reinforcement learning, training data is sequentially generated through repeated interaction with the environment. This makes reinforcement learning an ideal tool for solving challenging tasks where labelled or unlabelled data is difficult to produce. For example, creating a sufficiently complete labelled dataset $\mathcal{D} = \{\boldsymbol{x}_i, \boldsymbol{y}_i\}_{i=1}^{N}$ for playing chess such that game states $\boldsymbol{x}_i$ have corresponding human-expert moves $\boldsymbol{y}_i$ is difficult since the state space is too large. Furthermore, a supervised learning model trained on this dataset would only learn to imitate the expert instead of learning the best possible chess strategy and only manage single timestep predictions instead of learning long-horizon strategies.

2.1.1 Markov Decision Processes

The standard formulation of a reinforcement learning task is a tuple consisting of the following elements.

- **States** $\mathcal{S}$. The set of possible states or configurations the agent can occupy in the environment at any time t.
- **Actions** $\mathcal{A}(s)$. The set of available actions or controls the agent can take in state $s \in \mathcal{S}$. The agent may have different actions depending on its current state. However, for simple agents, all actions in the action space $\mathcal{A}(s)$ are admissible in all states $s \in \mathcal{S}$. Reinforcement learning agents use actions to transition between different states in the environment, traversing the state space in pursuit of goal states.
- **Transition Function**. A function $\mathcal{T} : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ that dictates what state s_{t+1} the agent transitions to after taking action a_t while in state s_t. In non-deterministic environments, the transition function is defined by a probability distribution over the set of states $\mathcal{S}_{s,a} \subseteq \mathcal{S}$ reachable by taking action a in some state s, i.e. $P_{\mathcal{T}} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$.

- **Reward Function**. A function $r : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ that returns a value the agent receives for entering state s' from some state s using action a. This value provides a measure of how desirable a state is for the agent. As such, goals or behaviours in reinforcement learning are encoded using reward functions.

- **Discount Factor**. A constant $\gamma \in [0, 1)$ that regulates reward priority based on whether rewards occur immediately or in the long-term. If γ is close to 1, then the agent considers future rewards more strongly. But if γ is close to 0 then the agent acts more greedily, only prioritising immediate rewards.

If the above problem formulation is constrained to obey the **Markov property**, which states that the next state of the system is only determined by the current state i.e. $P(s_{t+1} \mid s_t) = P(s_{t+1} \mid s_t, s_{t-1}, s_{t-2}, ..., s_0)$, then it can be referred to as a **Markov Decision Process (MDP)**[2].

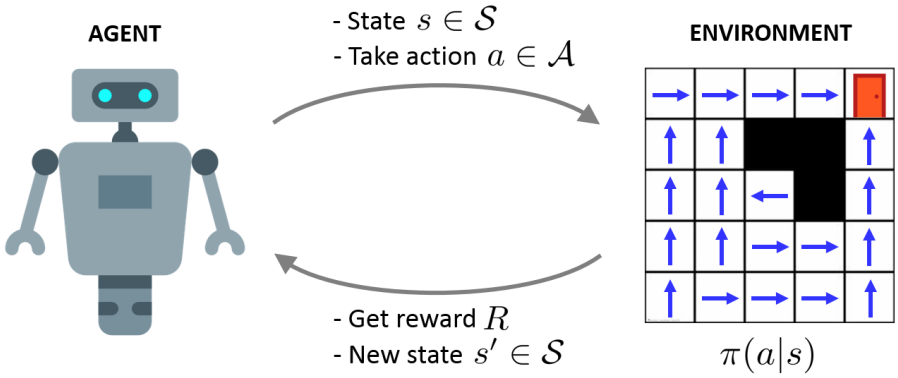


Figure 2.1: In reinforcement learning, an agent interacts with the environment (the MDP) to learn a policy (shown in blue) for how it should behave in the environment to achieve the highest cumulative reward.

2.1.2 The Policy

A solution to an MDP is a function $\pi : \mathcal{S} \rightarrow \mathcal{A}$, also referred to as the **policy**. In other words, for any state $s \in \mathcal{S}$ the policy will return an action $a \in \mathcal{A}$ for the agent to execute (shown as blue arrows in Figure 2.1). For non-deterministic environments, policies are given by a probability distribution $\pi(a|s)$ over all possible actions admissible to the agent in state s. Policies are action selection strategies which reinforcement

[2]In this work, Markov Decision Process and reinforcement learning environment are used interchangeably.

learning agents use to act in the environment. As such, a solution to a reinforment learning task is typically encoded as a **trajectory** $\tau = \{s_0, a_0, ...s_{t-1}, a_{t-1}, s_T, a_\emptyset\}$[3] from some start state s_0 to some terminal state s_T (in the case of episodic tasks). Agents generate these solutions by executing actions sampled from the policy $\pi(s_t|a_t)$ at every timestep until the terminal state s_T is reached.

In general, not all policies are equally good for a given MDP. Therefore, a mechanism for comparing them is required. Such is the role of the **value function**.

2.1.3 Value Functions

Unlike the reward function which dictates immediate rewards, the value function places emphasis on future rewards. Formally, the value of a state s_t denoted $V^\pi(s_t)$ under policy π, is the discounted sum of the future rewards (also termed the **return**) observed when starting in state s_t and following policy π thereafter. i.e.

$$V^\pi(s_t) = r_t + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{k=0}^{T} \gamma^k r_{t+k+1} \tag{2.1}$$

where $T = \infty$ for **continuing** (infinite horizon) tasks[4], in cases where an agent has to continue acting forever in the environment to solve the task, or $T \in \mathbb{N}^+$ for **episodic** (finite horizon) tasks, in cases where the agent has to solve the task in a finite number of steps in the environment. In this work, we only consider episodic tasks.

In non-deterministic environments, this discounted sum of rewards at a state (equation 2.1), is weighted by the corresponding state transition probabilities under policy π. Writing this out recursively leads to a form of the value function known as the **Bellman equation** [Sutton and Barto, 1998]:

$$V^\pi(s_t) = r_t + \gamma \sum_{s_{t+1}} P_\mathcal{T}(s_{t+1} \mid s_t, \pi(s_t)) V^\pi(s_{t+1}) \tag{2.2}$$

As suggested by the recursion in the Bellman equation, the value of a state depends on the values of its neighbouring states. More precisely, the value of a state is given by

[3]The agent cannot execute actions in the terminal state s_T since it is an absorbing state. Therefore, $a_\emptyset$ is the null action, a special action which leads back to the terminal state.

[4]In continuing tasks, the discount factor γ prevents the return from going to infinity.

the weighted average of the values of the neighbouring states. This weighted average can also be written as the following expectation[5]:

$$V^{\pi}(s_t) = \mathbb{E}_{\pi, P_{\mathcal{T}}} \left[r_t + \gamma V^{\pi}(s_{t+1}) \right] \tag{2.3}$$

A policy π is defined to be better than or equal to a policy $\pi^{'}$ if its expected return (i.e. equation 2.3) for all states is greater than or equal to that of $\pi^{'}$. Consequently, the **optimal policy**, denoted π^*, is the policy with the highest expected return across the state space. In other words, the optimal policy is the policy whose value function returns the highest value across all states than other policies i.e:

$$\pi^*(s_t) = \arg\max_{\pi} V^{\pi}(s_t) \tag{2.4}$$

The optimal policy can be obtained using an **optimal value function**:

$$V^*(s_t) = \mathbb{E}_{\pi, P_{\mathcal{T}}} \left[r_t + \gamma \max_{s_{t+1}} V^*(s_{t+1}) \right] \tag{2.5}$$

In this particular value function, the value of a state is no longer given by the weighted average of neighbouring state values, but by the value of the neighbouring state with the highest value. An agent with access to an optimal value function can follow the optimal policy by taking the action that leads to the neighbouring state with the highest value at each state.

The value function can also be defined as the value of taking an action a in state s and thereafter following policy π, this particular value function is more commonly known as the action-value function or the Q-value function:

$$Q^{\pi}(s_t, a_t) = \mathbb{E}_{\pi, P_{\mathcal{T}}} \left[r_t + \gamma \, V^{\pi}(s_{t+1}) \right] \tag{2.6}$$

which results in the following optimal Bellman equation:

$$Q^*(s_t, a_t) = \mathbb{E}_{\pi, P_{\mathcal{T}}} \left[r_t + \gamma \max_{a_t \in \mathcal{A}} Q^*(s_{t+1}, a_{t+1}) \right] \tag{2.7}$$

Similarly, an agent can follow the optimal policy by greedily choosing actions with the highest value at each state:

$$\pi^*(s_t) = \arg\max_{a_t \in \mathcal{A}} Q^*(s_{t+1}, a_{t+1}) \tag{2.8}$$

[5]More commonly known as the **expected return**

2.2 Model-free and Model-based Methods

If all the elements of the MDP are known, then the policy can be computed using planning-based approaches where agents are able to simulate future states and rewards, allowing them to learn a policy without ever having to execute actions in the environment. However, this is not the case for practical reinforcement learning problems as the agent does not know how the environment will change in response to its actions (the transition function) or what immediate rewards (the reward function) it will obtain after applying these actions. In such cases the agent has to act in the actual environment and learn a policy based on its observations.

Model-based reinforcement learning agents solve this problem by learning these functions from data generated through interacting with the environment. These learned functions can then be used to plan a solution for the given MDP.

Unlike model-based approaches, **model-free** reinforcement learning agents do not attempt to learn models for predicting the next state and reward before they execute actions. Instead, they direct effort towards learning quantities that represent the policy directly, through trial and error. Model-free reinforcement algorithms can be further classified as being either **off-policy** or **on-policy**. In reinforcement learning literature, on-policy reinforcement learning refers to cases where agents execute actions sampled from the current policy (i.e. the data-generating policy is the current policy), whereas in off-policy reinforcement learning, the data used to update the current policy is not generated by the current policy.

2.3 Value-based Reinforcement Learning

So far we have discussed different ways of formulating the value function (i.e. as the value of individual states or as the value of state-action pairs) as well as the corresponding optimal value functions (equations 2.5 and 2.7) and further explained how to extract the optimal policy from these (equation 2.8). Algorithms that learn the value function and then extract the policy in this way are referred to as value-based reinforcement learning algorithms. An example of such an approach is TD-learning.

2.3.1 TD-Learning

As opposed to computing the value function directly (i.e. calculating the total discounted reward), Temporal Difference (TD) learning [Sutton, 1988] agents incrementally update the value function through bootstrapping, using the **Temporal Difference error**.

$$\delta = \underbrace{r + \gamma V(s')}_{\text{The TD target}} - V(s) \tag{2.9}$$

This error (equation 2.9) is defined to be the difference between temporally successive estimates of the value function. The TD-Learning algorithm[6] is as follows:

Algorithm 1 TD-Learning

Input: initialise $V(s) \quad \forall \quad s \in \mathcal{S}$
repeat
  for each episode **do**
    Initialise s
    $a \leftarrow$ action from the estimated policy π in s
    Take a, observe r, s'
    Update V:

$$V(s) \leftarrow V(s) + \alpha \left[r + \gamma V(s') - V(s) \right]$$

    $s \leftarrow s'$
  end for
until s is terminal

In this algorithm, the current, possibly incorrect estimate of the value function $V(s)$ is improved over time as the agent continues to interact with the environment. Since the agent behaves according to the implicit policy in the value function $V(s)$, it is an on-policy algorithm.

The on-policy nature of the TD-Learning algorithm makes it sensitive to the initialisation of the value function $V(s)$ since the algorithm commits to following the same implicit policy during learning. This becomes an issue in environments with large state-action spaces where good exploration strategies are imperative. Furthermore, the above TD-learning algorithm does not make use of the action information at each

[6]The algorithm presented here is also known as TD(0). TD(0) is a special case of the more general TD(λ) algorithm, which instead of using a single timestep to update the value function, uses multiple timesteps, yielding a more accurate estimate of value function.

state. As such, a one-step look-ahead model is required to extract the policy. These issues are addressed in the Q-Learning algorithm presented in the next section.

2.3.2 Q-Learning

Q-learning [Watkins and Dayan, 1992] is an off-policy variant of TD-Learning where the optimal action-value function $Q^*(s, a)$ can be learned independent of the policy being followed. This off-policy nature, allows for a straightforward integration of an exploration strategy where the agent can sometimes execute actions from another policy (e.g. a random policy) to explore. Being able to behave off-policy means that Q-learning is less reliant on the initialisation of $Q(s, a)$. As such, it has better chances of discovering better solutions compared to the TD-Learning algorithm. The Q-learning algorithm is summarised below:

Algorithm 2 Q-Learning

Input: initialise $Q(s, a) \quad \forall \quad (s, a)$
repeat
  for each episode **do**
    Initialise s
    for each step in episode **do**
      $a \leftarrow \arg\max_a Q(s, a)$
      Take a, observe r, s'
      Update Q:
      $Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right]$
    end for
    $s \leftarrow s'$
  end for
until s is terminal

The action-value function in this algorithm is stored in a Q-table – a table-like data structure where each cell stores the Q-value for the corresponding state row and action column. The Q-values for each (s, a) pair would then be updated during learning. However, such a representation is only possible for discrete environments with countable states and actions.

2.4 Policy-based Reinforcement Learning

In contrast to value-based reinforcement learning, policy-based reinforcement learning methods approximate the policy directly, and learn what action to take in a state in order to maximise some notion of cumulative reward. As such, policy-based algorithms explicitly represent the policy π and iteratively update this policy using data obtained through direct interaction with the environment.

However, policy-based reinforcement learning algorithms generally suffer from high variance which results in them requiring a large number of interactions with the environment before converging to a good solution. This effect is exacerbated in large state-action spaces. One way to address this issue is employ a good exploration strategy.

2.5 Exploration Strategies

Exploration strategies play an important role in reinforcement learning in that they help balance between exploration and exploitation. In general, agents start out with more exploration than exploitation, and explore less in later episodes where the current policy is closer to optimality. This paradigm is apparent in the common exploration strategies we discuss below.

2.5.1 ϵ-greedy Exploration

In **ϵ-greedy** exploration, the agent selects actions for exploration and exploitation according to some parameter $\epsilon \in [0, 1]$ such that the agent chooses a random action with probability ϵ (exploration) and chooses an action from the current estimate of the policy with probability $1 - \epsilon$. In practice, ϵ is initialised to be large so that the agent performs more exploration than exploitation in the early episodes and decayed over time so that the agent exploits more in later episodes. This ϵ-greedy action selection can be incorporated into the standard Q-learning algorithm (algorithm 2) in the following way.

Algorithm 3 ϵ-greedy Q-Learning

1: Input: initialise $Q(s,a) \quad \forall \quad (s,a)$
2: **repeat**
3: **for** each episode **do**
4: Initialise s
5: **for** each step in episode **do**
6: $a \leftarrow \begin{cases} \arg\max_a Q(s,a) & \text{with probability} \quad 1-\epsilon \\ \text{randomly selected action} & \text{with probability} \quad \epsilon \end{cases}$
7: Take a, observe r, s'
8: Update Q:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$$

9: **end for**
10: $s \leftarrow s'$
11: **end for**
12: **until** s is terminal

In the above algorithm, line 6 has been modified to include the ϵ-greedy action selection strategy.

2.5.2 Boltzmann Exploration

In **Boltzmann exploration**, actions are selected according to a softmax distribution as opposed to selecting the action with the max Q-value at each state. The effect is that the agent selects good actions (actions with higher Q-values) more often (exploitation) and other actions less often (exploration).

2.5.3 Random Exploration

Another approach to exploration is to explore randomly, where agents select actions uniformly at random at each state. In continuous action spaces, actions sampled from the current estimate of the policy are polluted with additive noise (zero mean and unit variance) i.e. $a \leftarrow \pi(a|s) + \eta$, where $\eta \sim \mathcal{N}(0,1)$. In some cases, authors substitute η with temporally correlated noise such as the Ornstein-Uhlenbeck process, where the additive noise on current actions depends on previous actions [Lillicrap et al., 2015].

2.5.4 Other Exploration Strategies

More advanced exploration strategies include count-based exploration where state-visitation or action counts are maintained during learning and are used to bias agents towards taking actions or visiting states with lower counts [Tang et al., 2017, Thrun and Möller, 1991, Rosman and Ramamoorthy, 2012, Thrun and Möller, 1992]. Other approaches include intrinsic motivation where agents are able explore based of some internal notion of state novelty or curiosity model [Pathak et al., 2017, Burda et al., 2018, Barto, 2013].

Exploration in reinforcement learning is typically random in nature, where agents have to execute actions sampled from random exploration policies. This presents an issue for safety critical systems since there is no guarantee that the agent won't perform the most harmful action in the environment. In this work, we present a method which relaxes the randomness in the exploration policy, thus resulting in safer reinforcement learning.

2.6 Summary

This chapter introduced the basic concepts in traditional reinforcement learning. This includes the reinforcement learning problem formulation (summarised in Figure 2.1), common optimal value-function formulations as well as how the policy can be extracted from them. Algorithms for learning the policy can be 1) model-based where the agent prioritises first modelling the environment and planning using the resulting model, or 2) model-free where agents interact with environment and update their policy based on observations in the environment. Reinforcement learning agents rely on good exploration strategies to discover good solutions for the given MDP. Algorithms that learn off-policy are able to interleave between executing actions from the estimated policy and actions from a different policy, used solely for exploration. This exploration step is typically achieved by selecting actions randomly, this poses safety issues and makes it difficult to apply reinforcement learning on safety-critical systems.

The concepts and algorithms discussed in this chapter are only compatible with environments where the state and action spaces are discrete in nature, and rely on tabular representations to store reinforcement learning quantities during learning. However,

in more practical settings, such data structures are impractical. In the following chapter, we explain how function approximation coupled with traditional reinforcement learning be used to enable learning in richer spaces.

Chapter 3

Deep Reinforcement Learning

Having discussed the reinforcement learning problem for discrete environments (chapter 2), we now port these concepts to continuous environments through function approximation and discuss work that has done this successfully under the broad umbrella of deep reinforcement learning. First we describe the deep-Q network in section 3.1, then we discuss the transition from this value-based deep reinforcement learning algorithm to policy-based deep reinforcement learning using policy gradients in section 3.2. We then explain how these approaches can be combined under the Actor-Critic framework in section 3.3, and present the DDPG algorithm in section 3.4 – an algorithm that combines all the ideas discussed in this chapter. We list issues with deep reinforcement learning in section 3.5, and finally summarise the chapter in section 3.6.

In general, deep reinforcement learning is defined as the combination of function approximation (more commonly deep neural networks) with reinforcement learning. Where function approximation is used to approximate the different elements of the reinforcement learning problem. particularly the policy (for policy-based deep reinforcement learning), the value function (deep reinforcement learning), and the dynamics model (for model-based deep reinforcement learning). We discuss these (with the exception of model-based deep reinforcement learning) in more detail in the following sections.

3.1 Deep-Q Networks (DQNs)

The Deep-Q Network (DQN) [Mnih et al., 2015] is one of the pioneering works in value-based deep reinforcement learning. In their work, the authors solve the previously difficult task of incorporating deep neural network function approximators in reinforcement learning, producing an algorithm (Deep-Q Learning with Experience Replay) that was able to play different games in the Atari game set using the same learning architecture (Figure 3.1). This algorithm was able to achieve superhuman-level performance on most Atari games, exceeding the performance of closest matching work in similar domains.

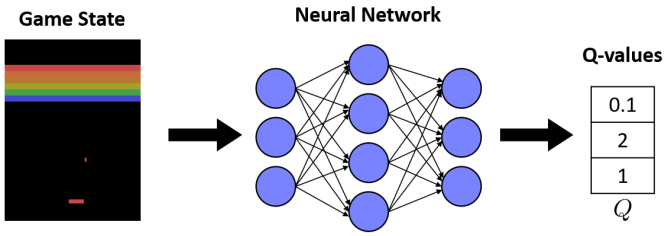


Figure 3.1: An overview of the Deep-Q Network (DQN) used by Mnih et al. [2015] to represent the action value function $Q(s, a)$. In their method, the neural network would predict the average return for each possible action for the given state.

Before Deep-Q Learning with Experience Replay was introduced, reinforcement learning algorithms that used non-linear function approximation (particularly neural networks) to estimate the action-value function Q were believed to be inherently unstable and were thus likely to diverge [Tsitsiklis and Van Roy, 1996].

This instability was partly caused by: 1) the violation of the standard assumptions in the stochastic gradient descent optimizer used to train the network, i.e. the training data must be i.i.d.[1] and drawn from a stationary probability distribution. However, data in reinforcement learning is sequentially generated and is thus temporally correlated, which breaks the i.i.d. assumption. Furthermore, the distribution over these transitions changes as the agent learns over time [Mnih et al., 2015]. 2) non-stationary targets caused by the temporal correlations between the current estimate of the action-value function Q and the target $r_t + \gamma \max_{a'} Q_{\phi_{\text{targ}}}(s_{t+1}, a')$– the temporally successive estimate of Q.

[1]Independent and identically distributed.

To remove the temporal correlations between transition samples, Mnih et al. [2015] used experience replay by Lin [1993]. This method entails storing the agent's transitions (s, a, r, s') in a memory replay buffer, and then training on transitions randomly sampled from this buffer. To handle the non-stationary target issue, the target network weights ϕ_{targ} were held fixed and only periodically updated after every C-steps during training. This stabilised the training, allowing the DQN to learn action-values $Q_\phi(s, a)$ end-to-end[2] using image pixels as the state s, joystick positions as actions a, and the in-game score as the reward r.

The DQN was initialised with random weights and trained using backpropagation and stochastic gradient descent to minimize the mean square error objective function:

$$I = (y_j - Q_\phi(s_j, a_j))^2 \tag{3.1}$$

Where $y_j = r_j + \gamma \max_{a'} Q_{\phi_{\text{targ}}}(s_{j+1}, a')$ is the Q-Bellman equation[3] target estimated using the target network ϕ_{targ} and $Q_\phi(s, a)$ parameterized using deep-Q network ϕ, the current estimate of the action-value function. This loss function (equation 3.1) was evaluated using a minibatch of transitions $B = \{(s_j, a_j, r_j, s_{j+1})\}$ sampled randomly from the experience replay buffer. The complete algorithm is as follows:

[2]Training a deep Q network using raw data (image pixels in this case) without the use of predefined features or any form of pre-processing.

[3]Equation 2.7

Algorithm 4 Deep-Q Learning with Experience Replay

Initialise replay memory $\mathcal{D}$ to capacity N
Initialise the Deep-Q Network with random weights ϕ
Set target network parameters equal to main parameters $\phi_{\text{targ}} \leftarrow \phi$
for episode $= 1, M$ **do**
  Observe initial state s_1
  for $t = 1, T$ **do**
    Select action $a_t \leftarrow \begin{cases} \arg\max_a Q_\phi(s,a) & \text{with probability} \quad 1-\epsilon \\ \text{random action} & \text{with probability} \quad \epsilon \end{cases}$
    Execute action a_t in the environment
    Observe reward r_t and next state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in replay memory $\mathcal{D}$
    Randomly sample mini batch of transitions $B = \{(s_j, a_j, r_j, s_{j+1})\}$ from $\mathcal{D}$
    Compute target $y_j = \begin{cases} r_j & \text{for terminal} \quad s_{j+1} \\ r_j + \gamma \max_{a'} Q_{\phi_{\text{targ}}}(s_{j+1}, a') & \text{for non-terminal} \quad s_{j+1} \end{cases}$
    Update Deep-Q Network by one step of gradient descent using

$$\nabla_\phi \frac{1}{|B|} \sum_{(s_j, a_j, r_j, s_{j+1}) \in B} \left(y_j - Q_\phi(s_j, a_j)\right)^2$$

    Update target network parameters every C steps

$$\phi_{\text{targ}} \leftarrow \phi$$

  end for
end for

Given a state s, the trained Deep-Q Network produces Q-values over a discrete action space i.e.:

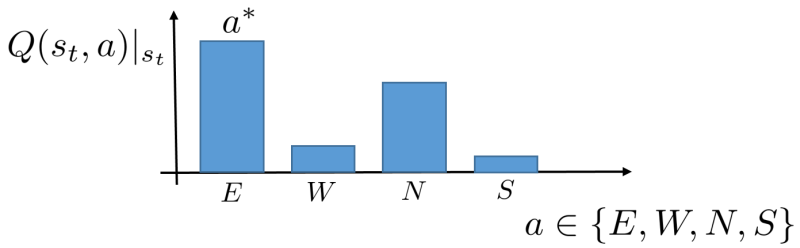


Figure 3.2: A possible Q-value function for a discrete gridworld environment where the agent can execute any of the four actions: $\{E, W, N, S\}$ at some state s_t.

To extract the optimal policy from the approximated Q-value function, the agent greedily selects the action with the highest Q-value at every time step as follows:

$$\pi(s_t) = \arg\max_{a_t} Q_\phi(s_{t+1}, a_{t+1}) \tag{3.2}$$

However, this policy extraction method does not extend to environments with continuous action spaces since the corresponding Q-value function in such environments is also continuous:

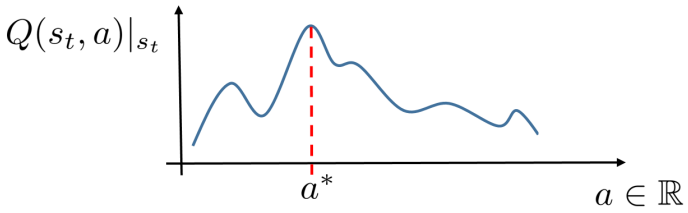


Figure 3.3: A possible Q-value function for a continuous environment where the agent can execute a real-valued action. For example, this can be a joint torque in the case of a robot arm.

As such, extracting the greedy policy here, requires a global optimization operation over possible actions. This optimization problem can be roughly formulated using the 2nd-order sufficient conditions for local maxima i.e.:

$$\text{Set} \quad \pi(s_t) = a^* \quad \text{iff} \quad \nabla_a Q_\phi(s_t, a)|_{a^*} = 0 \quad \text{and} \quad \nabla_a^2 Q_\phi(s_t, a)|_{a^*} < 0 \tag{3.3}$$

This is a computationally expensive operation since in addition to first having to learn the action-value function Q_ϕ, this optimization problem (equation 3.3) has to be solved for every state s_t we encounter at every time step t. Furthermore, this optimization problem assumes that Q_ϕ at any state s_t is smooth and twice differentiable. However, this may not be the case in some reinforcement learning problems. Also, the overall computation time for this cannot be reduced by storing previous solutions (as is the case with most dynamic programming algorithms) since we have uncountable states and actions as well as no guarantee that the agent will revisit the same state s_t, thus requiring the same solution $\pi(s_t) = a^*$.

A more reasonable approach is to instead maintain a function approximator for the policy, and train this approximator using sample-based optimization such that adjusting the parameters of the policy changes the policy at all states simultaneously.

This eliminates the need to solve for the optimal action in every state encountered since recovering the policy under this parameterization is now a simple evaluation of the function approximator at some state s given weight configuration θ.

As with the Deep-Q Network in the Deep-Q Learning with Experience Replay algorithm, having a parameterization requires us to formulate the objective in terms of the policy parameters such that the optimal parameters θ^* are given by:

$$\theta^* = \arg\max_{\theta} J(\theta) \tag{3.4}$$

In most deep reinforcement learning algorithms that model the policy in this way, the objective $J(\theta)$ is chosen to be the expected[4] return $\mathbb{E}_{\tau\sim\pi_\theta(\tau)}[\,R(\tau)]$. In such cases, the policy parameters θ are adjusted in the direction of increasing expected return. Thus yielding the **policy gradient**:

$$\nabla_\theta J(\theta) = \nabla_\theta \mathbb{E}_{\tau\sim\pi_\theta(\tau)}[\,R(\tau)] \tag{3.5}$$

3.2 Policy Gradient Methods

Policy gradient algorithms are a class of continuous action reinforcement learning algorithms that make use of function approximation θ (more commonly deep neural networks) to explicitly represent the policy. In general, the policy parameters θ are adjusted in the direction of increasing performance objective $J(\theta)$ using variants of the general stochastic policy gradient theorem [Sutton et al., 2000]:

$$\nabla_\theta J(\theta) = \int_{\mathcal{S}} \rho^\pi(s) \int_{\mathcal{A}} \nabla_\theta \pi_\theta(a|s) Q^\pi_\phi(s,a) \mathrm{d}a\mathrm{d}s \tag{3.6}$$

where $\rho^\pi(s)$ is the discounted state visitation distribution, π_θ is the policy and $Q_\phi(s,a)$ is the action-value function under some compatible function approximator with parameters ϕ. Intuitively, the policy parameters θ are adjusted to make high-value actions more likely, thus maximising the total discounted reward [Silver et al., 2014].

[4]For stochastic environments

Different policy gradient methods have different ways of estimating the policy gradient $\nabla_\theta J(\theta)$. However, in almost all policy gradient algorithms, the estimated gradient $\nabla_\theta J(\theta)$ updates the policy parameters θ as follows:

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta) \tag{3.7}$$

where α is the learning rate.

We now briefly discuss the more common formulations of the policy gradient. This includes, the Likelihood-ratio policy gradient (section 3.2.1), the Natural policy gradient (section 3.2.2) and finally, the Deterministic policy gradient (section 3.2.3).

3.2.1 Likelihood-ratio Policy Gradient

The likelihood-ratio policy gradient [Williams, 1992] is amongst the earliest policy gradient methods. This particular formulation (see Appendix A.1) uses the "likelihood-ratio" identity to express the standard policy gradient (equation 3.6) in a convenient form where where it does not depend on the state visitation distribution or the transition function.

$$\nabla_\theta J(\theta) = \mathbb{E}_{s\sim\rho^\pi, a\sim\pi_\theta}\left[\nabla_\theta log\pi_\theta(a|s) Q^\pi_\phi(s,a)\right] \tag{3.8}$$

This formulation makes it an ideal candidate for model-free continuous action reinforcement learning, where through Monte Carlo sampling approaches, the above expectation can be approximated from data[5] (see Appendix A.2):

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T}\nabla_\theta log\pi_\theta(s_{i,t}|a_{i,t})\right)\left(\sum_{t=1}^{T} R(s_{i,t},a_{i,t})\right) \tag{3.9}$$

where $R(s,a)$ is a general proxy for the value function. The independence of this policy gradient from the state-visitation distribution implies that the policy gradient can be evaluated off-policy using trajectories generated offline from a separate policy ρ^β. As such, the exploration policy is allowed to be different and can also be performed offline. In this work, we exploit this result, and use exploration as a mechanism to administer knowledge from other tasks in the same environment into the target task.

[5]In the form of transition samples from trajectories in the environment

3.2.2 Natural Policy Gradient

Standard policy gradients rely on the learning rate α to control the magnitude of the policy gradient $\nabla_\theta J(\theta)$ used to update the policy parameters θ during learning. When α is too small, the learning process is slow and takes longer to converge. On the other hand, a large learning rate α causes large changes in the policy parameters space $\boldsymbol{\theta}$, causing aggressive fluctuations in the policy space. This produces bad policies, which in turn, hurts further learning as successive policy gradient estimates depend on transition samples generated using the current policy.

The natural policy gradient addresses this issue by first representing the policy step-size (i.e. the "distance" between $\pi_{\theta+\delta\theta}(a|s)$ and $\pi_\theta(a|s)$) in parameter space $\boldsymbol{\theta}$, and then restricting this change in the policy to be less than some parameter ε i.e.:

$$\mathrm{KL}(\pi_\theta(a|s)||\pi_{\theta+\delta\theta}(a|s)) \approx \delta\theta^T \mathbb{F}_{\pi_\theta} \delta\theta \leq \varepsilon \tag{3.10}$$

where $\mathbb{F}_{\pi_\theta}$ is the Fisher Information Matrix:

$$\mathbb{F}_{\pi_\theta} = \mathbb{E}_{a\sim\pi_\theta}\left[\nabla_\theta log\pi_\theta(a|s) \nabla_\theta log\pi_\theta(a|s)^T\right] \tag{3.11}$$

The natural policy gradient $\nabla_\theta^{\mathrm{NPG}} J(\theta)$ is then given by:

$$\nabla_\theta^{\mathrm{NPG}} J(\theta) = \mathbb{F}_{\pi_\theta}^{-1} g \tag{3.12}$$

where $\mathbb{F}_{\pi_\theta}^{-1}$ is the inverse Fisher Information Matrix which approximates the policy space difference between two points in parameter space $\boldsymbol{\theta}$ and $g = \nabla_\theta J(\theta)$ is the standard policy gradient.

Unlike the vanilla policy gradient (equation 3.8), the natural policy gradient achieves faster and more stable learning [Peters and Schaal, 2008]. Other variants of the natural policy gradient [Schulman et al., 2015, Wu et al., 2017], also offer monotonic policy improvement in exchange for more computational complexity. However, recent work [Schulman et al., 2017] has succesfully brought this computational complexity down while retaining the useful properties of the natural policy gradient.

3.2.3 Deterministic Policy Gradient

Following the stochastic policy gradient by Sutton et al. [2000], Silver et al. [2014] presented the deterministic policy gradient (DPG),

$$\nabla_\theta J(\theta) = \int_{\mathcal{S}} \rho^\mu(s) \nabla_\theta \mu_\theta(s) \nabla_a Q^\mu_\phi(s, a) \mathrm{d}s \tag{3.13}$$

where $\rho^\mu(s)$ is the discounted state visitation distribution, μ_θ is the deterministic policy and $Q_\phi(s, a)$ is the action-value function under some compatible function approximator with parameters ϕ. Compared to its stochastic counterpart, the deterministic policy gradient theorem (equation 3.13) does not integrate over possible actions and therefore, can be estimated more efficiently.

The deterministic policy gradient, also termed the action-value gradient, assumes a deterministic policy μ. This allows the exploration to be handled completely off-policy by a separate behaviour policy $\pi^\beta(s)$. Silver et al. [2014] shows that if π^β is chosen to be stochastic i.e. $\pi^\beta(a|s)$ then we retain the properties of the stochastic policy in the stochastic policy gradient, with the main difference being that the policy gradient can now be computed more efficiently.

3.3 The Actor-Critic Framework

The vanilla policy gradient in equation 3.8, computes N individual policy gradient estimates $\nabla_\theta J(\theta)_i$ using full trajectories $\tau = \{(s_t, a_t)\}_{t=1}^T$ and then takes the average over these to compute the final policy gradient:

$$\nabla_\theta J(\theta) = \frac{1}{N} \sum_{i=1}^{N} \nabla_\theta J(\theta)_i \tag{3.14}$$

Computing the policy gradient this way produces a policy gradient with high variance. This is because each trajectory τ_i is an evaluation of a stochastic policy. Due to this stochasticity, different trajectories might receive different rewards for applying the same action at a single time step t. This variance in rewards between trajectories accumulates as more actions are taken in the episode, thus leading to a noisy policy gradient estimate.

The actor-critic framework reduces this variance in the policy gradient by estimating the returns directly, using a learned value function instead of waiting for a full episode in order to compute the returns through Monte Carlo sampling. In this framework, a dependence between the policy and the value function is established such that when the policy "the actor" executes an action "acts" in the environment based on the state it occupies, the value function "the critic" estimates the value of that action "criticises" in the occupied state. The value function is then used to compute the temporal difference error (the criticism), which in turn is used to update the value function and the policy. This relationship between the actor and the critic is summarised in Figure 3.4:

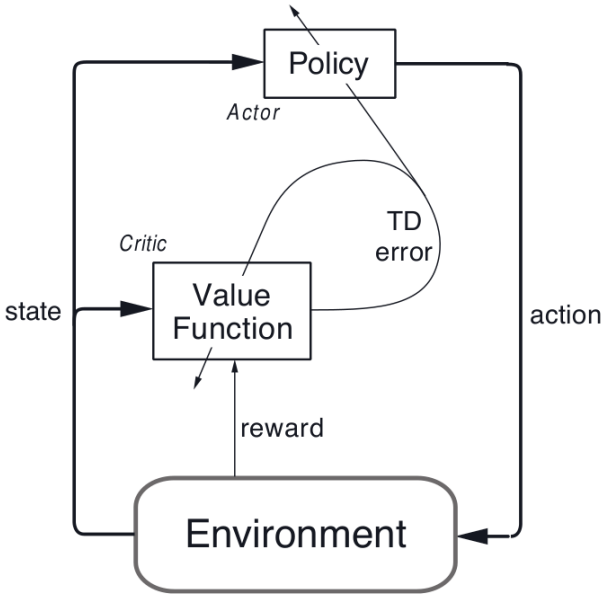


Figure 3.4: The actor-critic framework. Image adapted from [Sutton, 1988].

If the critic, "critises" actions generated by the current actor (i.e. the trained policy), then the resulting algorithm is referred to as the on-policy actor-critic. Conversely, updating the actor and the critic according to transitions generated off-policy results in the off-policy actor-critic algorithm. Using principles from the off-policy actor-critic algorithm as well as their deterministic policy gradient (equation 3.13), Silver et al. [2014] developed the "Off-Policy Deterministic Actor-Critic" algorithm. This algorithm was later extended by Lillicrap et al. [2015] in their Deep Deterministic Policy Gradient algorithm, which we discuss in the following section.

3.4 Deep Deterministic Policy Gradients

Lillicrap et al. [2015] combined the ideas discussed in the previous sections i.e. the Deep-Q Network [Mnih et al., 2015] and the deterministic policy gradient [Silver et al., 2014] to produce the Deep Deterministic Policy Gradient (DDPG) algorithm.

In the DDPG algorithm, Lillicrap et al. [2015] utilises deep neural network function approximators to represent the action-value function and the policy, together with novelties introduced in the Deep Q-Learning algorithm[6] to enable stable learning, this includes 1) target networks, which in DDPG were made to track the main networks slowly using some parameter ρ during learning. 2) An experience replay memory buffer to break correlations in the agent transitions.

The policy and value function networks in DDPG are trained in an off-policy actor-critic manner using a mini-batch of off-policy transitions $B = (s, a, r, s', d)$ sampled from an experience replay memory buffer. First the action-value network Q_ϕ is updated in a way similar to the vanilla DQN [Mnih et al., 2015], where the network weights are updated by means of stochatic gradient descent over the mean square error between the temporal difference target:

$$y(r, s', d) = \begin{cases} r & \text{for terminal} \quad s' \quad (d = 1) \\ r + \gamma Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s')) & \text{for non-terminal} \quad s' \quad (d = 0) \end{cases}$$

and the current estimate of the value function Q_ϕ:

$$\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_\phi(s, a) - y(r, s', d)\right)^2$$

Then the policy network μ_θ parameters were updated in the direction of increasing action-values Q_ϕ using the deterministic policy gradient (equation 3.13):

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_\phi(s, \mu_\theta(s))$$

Finally, the target networks are updated using a linear combination of the current target network weights and the current network weights (see line 15 in algorithm 5).

In this work we use the DDPG algorithm to learn multiple tasks for the purpose of evaluating our proposed solution to multi-task transfer learning for continuous action environments.

[6] See algorithm 4

Algorithm 5 Deep Deterministic Policy Gradient

Input: initial policy parameters θ, Q-function parameters ϕ, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$
repeat
  Observe state s and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
  Execute a in the environment
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  If s' is terminal, reset environment state.
  if it's time to update **then**
    for however many updates **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute the temporal difference target

$$y(r, s', d) = r + \gamma(1 - d)Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$$

      Update Q-function by one step of gradient descent using

$$\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_\phi(s, a) - y(r, s', d)\right)^2$$

      Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_\phi(s, \mu_\theta(s))$$

      Update target networks with

$$\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1 - \rho)\phi$$
$$\theta_{\text{targ}} \leftarrow \rho\theta_{\text{targ}} + (1 - \rho)\theta$$

    end for
  end if
until convergence

3.5 Issues with Deep Reinforcement Learning

Deep reinforcement learning agents tend to be sample inefficient, requiring lots of interactions with the environment during training before converging to good behaviours. While this is acceptable for simulated environments (e.g. video games), it is undesirable in the real world. For example, running millions of timesteps on a robot may take too long, possibly requiring that the robot be left uninterrupted for months on end. There is also no guarantee that the training will finish during this time since the actuators and batteries will surely get worn out.

Function approximators in deep reinforcement learning, particularly for policies, are typically initialised randomly. Executing these randomly initialised policies in the environment results in poor initial performance. Despite this randomness persisting after initialisation, agents continue to query these function approximators for actions to execute during training. This random nature, coupled with random exploration policies, poses a safety risk for physical systems.

While deep reinforcement learning algorithms present excellent solutions for learning individual tasks, their solutions typically do not generalise to new tasks [Garnelo et al., 2016]. As such, deep reinforcement agents solving new tasks in the domain have to be trained from scratch, and would have to reface the issues listed above.

3.6 Summary

In this chapter we outlined deep reinforcement learning, providing a detailed review of the pioneering work in this area – the deep-Q-network [Mnih et al., 2015] which achieved superhuman-level performance on a series of video games with discrete actions (controls) through neural network function approximation. We then discussed policy gradients as an approach for applying reinforcement learning to continuous action spaces through function approximation, and briefly mentioned recent work that makes use of these ideas.

We also presented DDPG, an algorithm which extends the Deep-Q Learning with Experience Replay algorithm to continuous action spaces and extends the "Off-Policy Actor-critic Algorithm" [Silver et al., 2014] to use deep neural network function approximators, resulting in a model-free, off-policy algorithm that is able to learn tasks

defined in environments specified using high-dimensional, continuous state and action spaces. In the previous section, we discuss that only relying on deep reinforcement learning algorithms such as DDPG leads to issues of sample inefficiency, poor initial performance, unsafe learning and poor generalisation across tasks.

We address the above issues using multi-task transfer learning and show how our proposed method can be incorporated into an off-policy deep reinforcement learning algorithm such as DDPG. The use of an off-policy algorithm allows for experience from other trained policies to be incorporated into the target policy in a straightforward manner. We describe common approaches for transferring knowledge between tasks as well as related benefits in the following chapter (chapter 4).

Chapter 4

Multi-task Transfer Learning

In this chapter we discuss related work in multi-task transfer learning. More specifically, we decompose multi-task transfer learning into two sub-problems: 1) knowledge compression (section 4.2) where we list methods that combine knowledge from multiple related source tasks into a single model or proxy for common knowledge between the source tasks, and 2) knowledge transfer (section 4.3) where we list methods for administering the common knowledge into the learning process of a related target task. We discuss the limitations for some of these methods and summarise the chapter in section 4.4.

4.1 Transfer Learning

In the reinforcement learning context, transfer learning involves agents that leverage the experience gained when learning a source task(s) to improve the learning of a related target task [Taylor and Stone, 2009]. According to Taylor and Stone [2009], a transfer learning agent must be able to:

1. Select the appropriate source task or set of tasks from which to transfer.
2. Learn how the source task(s) and target task are related.
3. Efficiently transfer knowledge from the source task to the target task.

In their review, Taylor and Stone [2009] also discuss the benefits of transferring knowledge in reinforcement learning domains. Among these is the Jumpstart, this is a measure of initial performance improvement in the trained policy. In practice, reinforcement learning agents that transfer knowledge between tasks are expected to start training with better initial performance. Another benefit of transfer is sample complexity or Time to Threshold, this metric evaluates the difference in convergence times between agents that use transferred knowledge and agents that learn from scratch. Here, agents with earlier convergence times are regarded as sample efficient as they need fewer samples (interactions with the environment) to converge. Another common evaluation metric is Asymptotic Performance, this refers to the ability of transfer learning algorithms to learn better solutions and retain those solutions over time. These metrics are summarised in Figure 4.1:

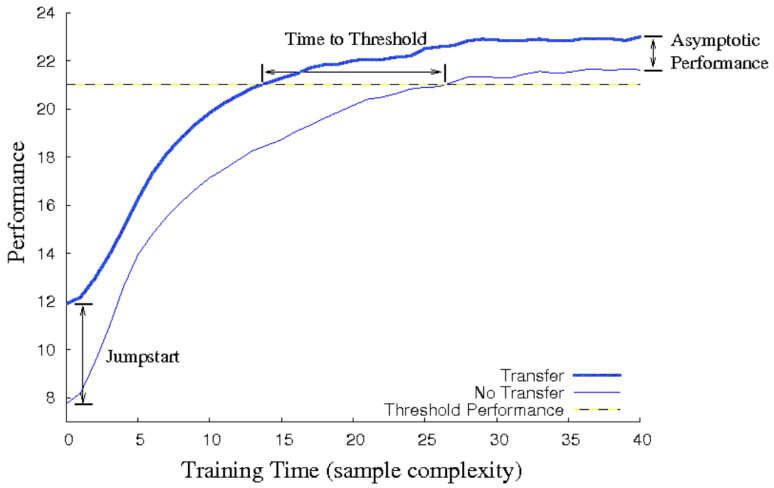


Figure 4.1: Transfer learning performance metrics. Adapted from [Taylor and Stone, 2009].

In some cases however, transferring knowledge can worsen the learning on the target task and possibly result in poorer performance compared to not using transferred knowledge at all. Such cases are referred to as negative transfer. This phenomenon can arise for a number of reasons namely: transferring knowledge between unrelated tasks, transferring knowledge using inaccurate task mappings [Taylor et al., 2007], etc. Intuitively, negative transfer is synonymous to transferring "bad advice".

In this chapter, we discuss common approaches for multi-task transfer learning, more specifically, approaches that align with the following pattern. As a decomposition of two sub-problems; **knowledge compression** and **knowledge transfer** (see Figure 4.2).

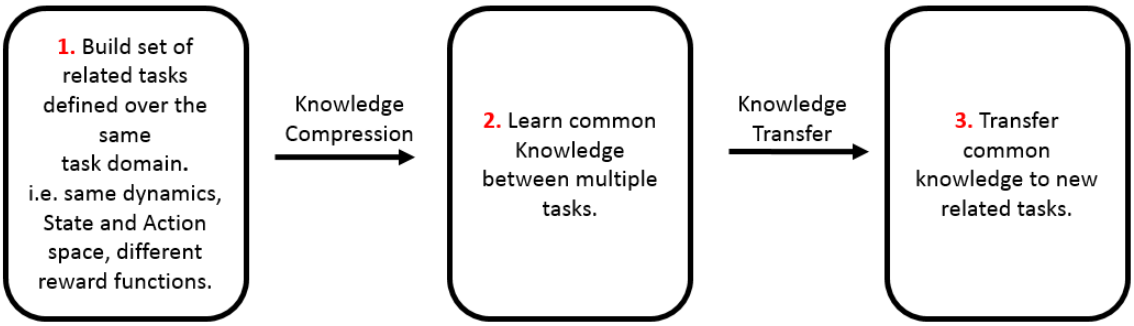


Figure 4.2: Multi-task transfer learning pipeline.

4.2 Knowledge Compression

Deep reinforcement learning agents tend to rely on large knowledge structures (e.g. deep neural networks) for implicitly [Mnih et al., 2015] and explicitly [Lillicrap et al., 2015] representing the policy. Maintaining a library of such policies for purposes of transferring knowledge to new tasks becomes more impractical with a large, but still growing number of tasks. Consequently, the more prior knowledge we wish to transfer to a target task, the higher the associated space complexity cost.

The knowledge compression step in the transfer learning process thus serves to increase the portability of prior knowledge by reducing the amount of total parameters required by combining multiple models or multiple sources of information into one. In the context of deep reinforcement learning, this can refer to policies, value functions or trajectories.

4.2.1 Knowledge Distillation

Related work on knowledge compression in deep reinforcement learning includes policy distillation [Rusu et al., 2015]. In this method, knowledge is transferred or "distilled" from a set of teacher models $\{\mathbb{T}_k\}_{k=1}^{K}$ (see Figure 4.3), to a student model $\mathbb{S}$ (see Figure 4.4). Under this model, multiple teacher models (Deep-Q Networks) are trained on different tasks. The teacher model outputs (i.e. the Q-values) are passed into a softmax function to convert the Q-values into action probabilities i.e.:

$$Softmax_{\tau}(Q) = \frac{\exp(Q/\tau)}{\sum_{i=1}^{N} \exp(Q(i)/\tau)} \tag{4.1}$$

such that each teacher model (DQN) generates a dataset $D_k = \{s_i, Softmax_{\tau}(Q_i)\}_{i=1}^{N}$. Using this dataset, the student model $\mathbb{S}$ is trained (in a supervised learning manner) to predict/track the output of the different teacher models using the distillation loss function[1]:

$$L_{MSE}(D^T, \theta_S) = \sum_{i=1}^{|D|} \parallel Softmax_{\tau}\left(Q_T(s,a)\right) - Q_S(s,a) \parallel^2 \tag{4.2}$$

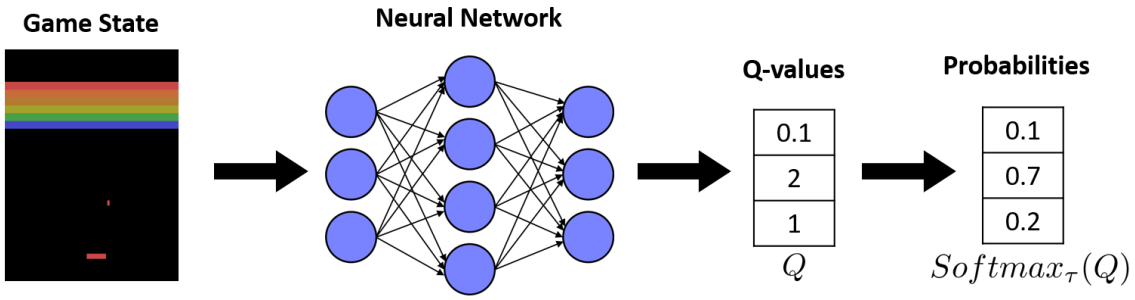


Figure 4.3: An example of a teacher model used in policy distillation to train a student model.

[1]Aside from the Mean Square Error loss shown here, the authors also showed that policy distillation is compatible with other loss functions namely: the negative log-likelihood loss and the KL-divergence loss.

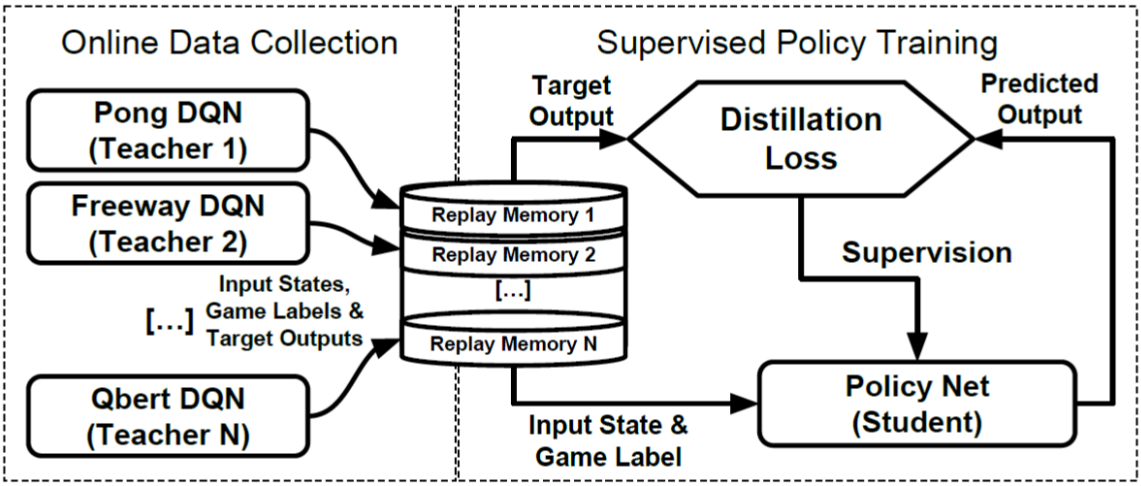


Figure 4.4: In this model, knowledge from multiple teacher models (see Figure 4.3) is compressed into a single student model through policy distillation [Rusu et al., 2015].

The policy distillation framework was also used by Tessler et al. [2016] who used distilled DQNs to compress multiple skill[2] networks into a single student model (Distilled Multi-Skill network), for tasks defined in the minecraft domain. This distilled network was then used as a skill module in their Hierarchical Deep reinforcement Learning Network, a framework for lifelong learning in the minecraft domain.

Policy distillation presents an effective solution for knowledge compression and uses fewer parameters to store knowledge from multiple tasks compared to maintaining a library of individual task solutions. However, a separate output layer has to be added for every task (teacher model) presented. As such, policy distillation methods still grow in the number of parameters as more tasks are added.

4.2.2 Fine-tuning and Feature Distillation

Yosinski et al. [2014] discussed that deep neural networks can learn general features over the task space and that these features are generally not specific to any particular instance in the training data but are representative of the entire dataset. For example, in the case of image classification tasks, common practice includes training deep neural networks to perform classification tasks on extremely large datasets [Krizhevsky et al.,

[2]In their work Tessler et al. [2016] defines skills as trained policies. i.e. after training on a task, an agent is said to have obtained a skill for solving that task.

2012], removing the last fully connected and output layer of the resulting deep neural network, then treating the resulting deep neural network as a transferable general representation or feature extractor for other image classification tasks over new image datasets. The more diverse the data, the more effective this approach is.

Parisotto et al. [2015] applied this idea, as well as policy distillation in deep reinforcement learning to perform distillation with features[3]. In this method, a student network was trained to mimic the actions of an expert (teacher) model. This was achieved using the actor mimic objective:

$$\mathcal{L}^i_{ActorMimic}(\theta, \theta_{f_i}) = \mathcal{L}^i_{policy}(\theta) + \beta * \mathcal{L}^i_{FeatureRegression}(\theta, \theta_{f_i}) \tag{4.3}$$

Where the policy regression objective:

$$\mathcal{L}^i_{policy}(\theta) = \sum_{a \in \mathcal{A}_{E_i}} \pi_{E_i}(a|s) log \pi_{AMN}(a|s;\theta) \tag{4.4}$$

is the cross entropy loss between the Actor Mimic Network (AMN, i.e. the student network) and the expert model (E_i, i.e. the teacher model). This objective served to constrain the action distribution of the student model to stay close to that of the teacher model. The feature regression objective:

$$\mathcal{L}^i_{FeatureRegression}(\theta, \theta_{f_i}) = \parallel f_i\left(h_{AMN}(s;\theta);\theta_{f_i}\right) - h_{E_i}(s) \parallel^2 \tag{4.5}$$

was used to compel the Actor Mimic network (student) to learn the same features as the expert using a separate feature regression network θ_{f_i}. In the main loss function (equation 4.3), β controls the relative weighting of the two objectives, treated as a hyperparameter for the model.

4.2.3 Weight-Sharing

In these methods, neural networks training on different tasks are typically stitched together such that knowledge from other tasks transfers (i.e. through network activations) into the current task. As such, knowledge transfer between tasks becomes mutual since tasks are typically trained jointly.

[3]In the paper, features were defined to be the weights before the output layer.

An example of work that makes excellent use of this idea is Progressive Neural Networks [Rusu et al., 2016]. In this work, Rusu et al. [2016] maintains a pool of pre-trained task models in a single model.

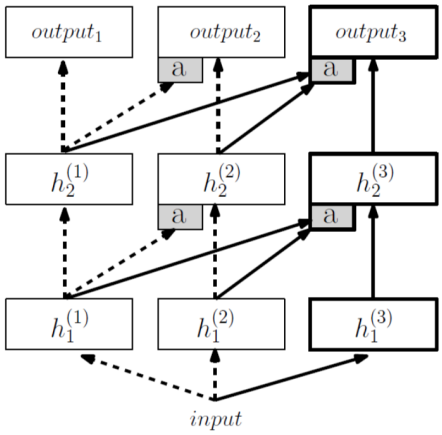


Figure 4.5: Progressive Neural Network [Rusu et al., 2016]

These task models (each column $\mathbf{h}^i$ in Figure 4.5) are connected by means of lateral connections. The lateral connections serve as feature extractors for progressive task models. The extracted features are incorporated directly into the hidden layers of new tasks models in the following way:

$$h_i^{(k)} = ReLU\left(W_i^{(k)} h_i^{(k-1)} + \sum_{j<k} U_i^{(k:j)} h_j^{(k-1)}\right) \tag{4.6}$$

As with policy distillation, weight-sharing methods also grow in size as more tasks are introduced. However this is still better than having to train a separate model for every task.

4.3 Knowledge Transfer

In this subsection we discuss the mechanism of applying or transferring the common knowledge to improve the learning of a target task. In other words, we ask the question, given prior knowledge in some environment, how do we administer this knowledge to an agent learning a task in the same environment such that the transferred knowledge aids the learning process?

4.3.1 Through Exploration

Finding good policies in reinforcement learning is associated with having good exploration strategies. Here, an agent can explore more efficiently if it explores according to the policies (experience) of previous agents in the environment. Intuitively, this translates to the current agent prioritising actions which previous agents in the domain found to be useful. If the agent then learns using such actions then it may discover more rewarding states sooner compared to exploring randomly. Recent work that follows this paradigm includes the action priors framework [Rosman and Ramamoorthy, 2012] which is discussed in detail in chapter 5.

4.3.2 Through Regularization

Desired behaviour in reinforcement learning is encoded using the reward function and consequently, the objective function. As such, it is common practice to include a regularization term that imposes additional constrains on the resulting behaviour. This regularization term is commonly used to prevent overfitting (i.e. forcing the model to prefer simpler configurations). However, it is also possible to constrain for example a task policy to stay "close"[4] to some reference policy or source task policy. Effectively, the task policy is "told" to pick actions that are not too different from those selected by the reference policy in some particular state. The result is a task policy that has similar behaviour to the reference policy but also solves the given target task. Work that transfers knowledge in this way includes that of Teh et al. [2017].

Teh et al. [2017] used a distilled model (the shared policy) as a regularizer for training individual task models. This was achieved by using a shared policy objective which consisted of expected returns (equation 2.1, in chapter 2) $\sum_{t=0}^{T} \gamma^t R_i(s_t, a_t)$, a KL divergence regularization term for constraining task policies to stay close to the shared policy: $c_{KL}\gamma^t log\frac{\pi_i(a_t|s_t)}{\pi_0(a_t|s_t)}$, and an entropy regularization term $c_{Ent}\gamma^t log\pi_i(a_t \mid s_t)$ to encourage exploration. Resulting in the following objective:

[4] According to some distance measure e.g. the KL-divergence

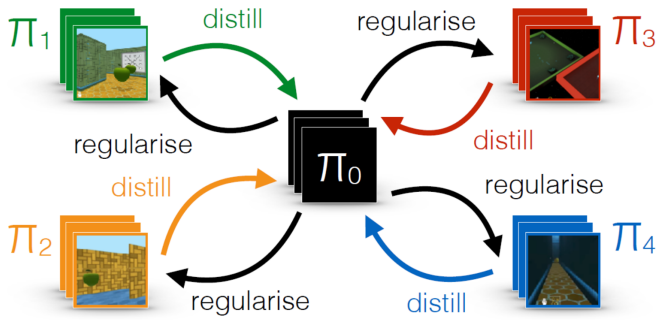


Figure 4.6: Distral: Distill and Transfer Learning [Teh et al., 2017]

$$J_0(\pi_0, \{\pi_i\}_{i=1}^n) = \sum_i \mathbb{E}_{\pi_i} \left[\sum_{t=0}^{T} \gamma^t R_i(s_t, a_t) - c_{KL}\gamma^t log \frac{\pi_i(a_t \mid s_t)}{\pi_0(a_t \mid s_t)} - c_{Ent}\gamma^t log \pi_i(a_t \mid s_t) \right] \tag{4.7}$$

And the corresponding policy gradient for training the task networks:

$$\nabla_{\theta_i} J_i(\pi_0, \theta_i) = \mathbb{E}_{\hat{\pi}_i} \left[\sum_{t=1}^{T} \nabla_{\theta_i} log \hat{\pi}_i(a_t \mid s_t) \left(\sum_{u=t}^{T} \gamma^u R_i^{reg}(s_u, a_u \mid \hat{\pi}_0) \right) \right] \tag{4.8}$$

4.4 Summary

Inline with our initial objective of implementing a multi-task transfer learning framework to address sample inefficiency and to avoid training from scratch for autonomous agents when learning new tasks in continuous environments, we have outlined common approaches in multi-task transfer learning and in some cases, discussed their limitations. This related work is discussed according to: 1) knowledge compression, which involves compressing information from multiple source tasks into a single model, thus improving knowledge portability, and 2) knowledge transfer which involves transferring or administering the compressed knowledge into an agent learning a new task in the environment.

Chapter 5

Continuous Action Priors

In this chapter we describe in detail our proposed solution for implementing continuous action priors. This includes a background on the action priors framework in section 5.1, possible avenues for extending this framework to continuous settings in section 5.2, and a summary of the chapter in section 5.3.

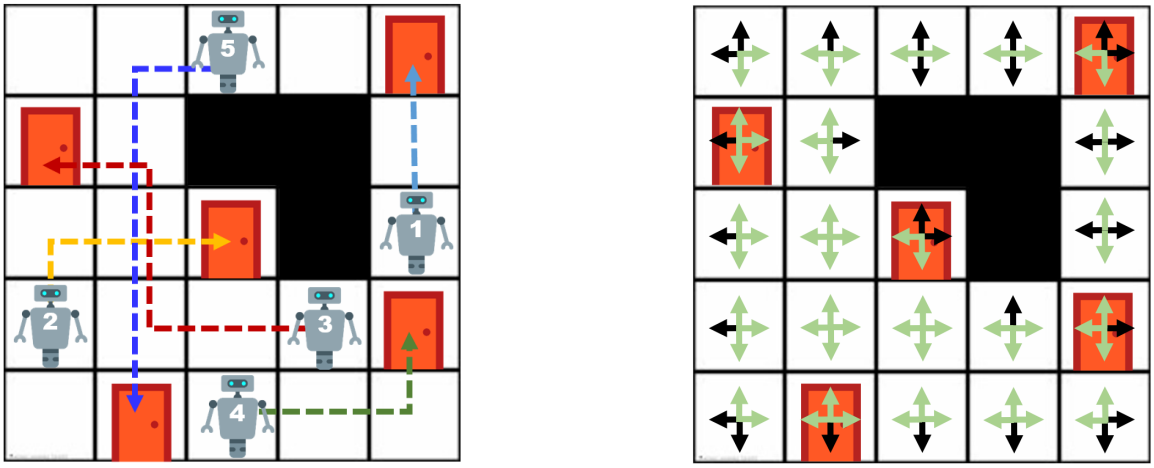
5.1 Discrete Action Priors

The action priors framework [Rosman and Ramamoorthy, 2012] is a technique for multi-task transfer through action selection for tasks $M = \langle D, R \rangle$ defined in the same domain[1] $D = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \gamma \rangle$, where S is the discrete state space, A is the discrete action space, T is the transition function, and γ is the discount factor. Under this framework, trajectories are sampled from multiple policies trained on different[2] tasks in the environment (see Figure 5.1a), to create a dataset of expert trajectories $\mathcal{D} = \{\tau_i\}_{i=1}^{N}$, where $\tau_i = \{(s_0, a_0), (s_1, a_1), ..., (s_T, a_T)\}$. In every state s, Rosman and Ramamoorthy [2012] computed and maintained action counts $\alpha_s(a)$ for each action $a \in \mathcal{A}$, indicating the number of times each action was executed in that state across all trajectories in $\mathcal{D}$. These action counts were normalised[3], resulting in a Dirichlet distribution $\psi_s(a) \sim Dir(\alpha_s)$ over expert actions in each state (see Figure 5.2).

[1] We use domain and environment interchangeably in this work.

[2] In this framework, the state space, action space, and transition function are fixed across all tasks (i.e tasks only differ by reward function).

[3] Such that they sum to 1



(a) (b)

Figure 5.1: a) An illustration of multiple near-optimal trajectories in a discrete environment where agents are tasked with navigating to a specific door. b) An illustration of the resulting action prior distribution, black arrows - actions with low action counts. green arrows indicate actions $a \in East, West, North, South$ with high actions counts.

Under this model, actions most favoured by multiple experts in a state s are assigned higher action counts $\alpha_s(a)$ and consequently higher probabilities in the action prior distribution $\psi_s(a) \sim Dir(\alpha_s)$. This results in these actions being sampled more often and actions that were least favoured by experts to be sampled less. If we assume that expert agents in the environment behave near-optimally (i.e. they do not execute harmful actions or collide into obstacles in the environment), then the action prior distribution can be interpreted as a distribution over good, safe actions across all states in the environment. This means that action priors effectively prune the action space in each state, assigning low probability to actions that were least favourable in a state (see Figure 5.1b). Ultimately, this pruning effect implicitly models the structure of the environment in the sense that actions with low probability would represent an obstacle or a wall.

To update this implicit structure, the state-based action counts $\alpha_s(a)$ first had to be updated according to:

$$\alpha_s^{new}(a) \leftarrow \begin{cases} \alpha_s(a) + 1 & \text{if } (s,a) \in \tau^{new} \\ \alpha_s(a) & \text{otherwise.} \end{cases}$$

When new trajectories τ^{new} were available. In the above, an action count $\alpha_s(a)$ for action a in state s would be updated when action a is executed in state s during τ^{new} (or when the state-action pair (s,a) is encountered in trajectory τ^{new}). Re-fitting a

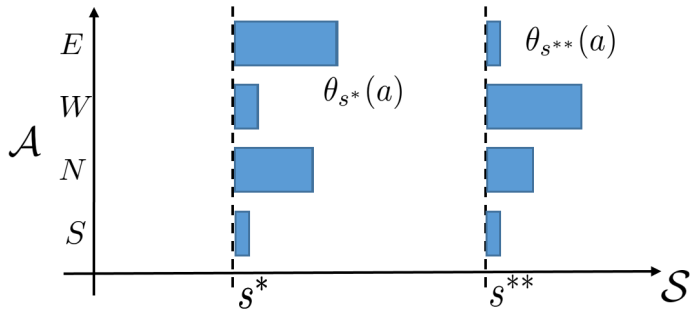


Figure 5.2: An illustration of the discrete action prior distribution $\psi_s(a) \sim Dir(\alpha_s)$ at states s^* and s^{**} respectively.

Dirichlet distribution over the new action counts results in an updated action prior distribution $\psi_s^{new}(a) \sim Dir(\alpha_s^{new})$.

5.1.1 Learning with Discrete Action Priors

In general, the action priors distribution $\psi_s(a) \sim Dir(\alpha_s)$ is a proxy for useful actions across all states in the environment. As such, Rosman and Ramamoorthy [2012] proposed using this distribution as an exploration policy, where an agent training on a new task can sample good, safe exploratory actions from this body of expert advice. Intuitively, such an agent would be exploring by taking actions that were previously successful more frequently, biasing exploration towards prior knowledge or experience when training on the given task in the environment. For example, in the multi-door environment in Figure 5.1a, a new agent can use the action priors (Figure 5.1b) constructed using optimal trajectories from expert agents like 1-5 in Figure 5.1a to bias exploration towards good actions when learning. That is, even though the new agent does not know the location of walls and obstacles, it will seldom collide into them.

To implement this biased exploration paradigm, Rosman and Ramamoorthy [2012] presented ϵ-greedy Q-learning with State-based Action Priors (ϵ-QSAP), a modified version of the standard ϵ-greedy Q-learning algorithm where the randomly selected action in the exploration step (line 5 in Algorithm 6) is replaced with an action sampled from the action prior distribution.

Algorithm 6 ϵ-greedy Q-learning with State-based Action Priors (ϵ-QSAP)

Require: action prior distribution $\psi_s(a)$

Initialise $Q(s, a)$ arbitrarily
for every episode $k = 1...K$ **do**
  Choose initial state s
  repeat
    $a \leftarrow \begin{cases} \arg\max_a Q(s,a) & \text{with probability} \quad 1-\epsilon \\ a \in \mathcal{A} & \text{with probability} \quad \epsilon\psi_s(a) \end{cases}$
    Take action a, observer r, s'
    $Q(s,a) \leftarrow \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right]$
    $s \leftarrow s'$
  until s is terminal
end for
return $Q(s, a)$

Rosman and Ramamoorthy [2012] also showed that administering the action priors as an exploration strategy in this way does not undermine the standard ϵ-greedy exploration strategy since if the action counts are kept constant i.e.:

$$\alpha_s(a) = k \qquad \forall\, s, a \in \mathcal{S} \times \mathcal{A}, k \in [0, 1]$$

Then the action prior distribution reduces to a uniform distribution where all actions are equally likely. Selecting actions ϵ-greedily according to this particular distribution is the same as sampling actions according to the standard ϵ-greedy action selection strategy.

Through navigation experiments in a series of maze navigation tasks, Rosman and Ramamoorthy [2012] showed that ϵ-greedy Q-learning with State-based Action Priors (ϵ-QSAP) performed better than the standard Q-learning algorithm, demonstrating a jumpstart in learning as well as improved long-term performance.

However, the action priors framework is only applicable to discrete environments, where the state and action spaces are countable sets with a finite number of elements. In this work, we extend action priors to continuous environments, where both the state and action spaces can be continuous. We outline the challenges with porting action priors to continuous environments as well as corresponding solutions in the next section.

5.2 Action Priors for Continuous Settings

For environments with large state spaces (as is the case in continuous state spaces), it is unlikely that an agent will revisit the same state or be able to visit all states (or a large portion of the state space) in the environment. As such, even if maintaining action counts was practical, it would not be useful since each state would have, at most, a single action count. In this section we present solutions for solving this representation problem through function approximation.

5.2.1 From Counts to Densities

As stated in Section 5.1, the action priors framework relies on maintaining counts for each possible action in a state and modelling them as a Dirichlet distribution in each state. This method is not practical in the continuous setting since the state and action spaces are no longer countable (i.e. specific actions in a state may never be repeated action across different experts, making the action count paradigm impractical). As such, the action counts can no longer be modelled sufficiently using a histogram (using bins), and consequently, the Dirichlet distribution.

The problem of "counting" occurrences for continuous quantities (i.e. continuous random variables) is analogous to **density estimation**. The objective in density estimation, is to construct a probability density function from data. For modelling action priors in particular, we are concerned with constructing a conditional density function $\psi_s(a) \sim P_\psi(a|s)$ from a dataset of expert trajectories $\mathcal{D} = \{\tau_i\}_{i=1}^N$, where $\tau_i = \{(s_j, a_j)\}_{j=1}^T$ such that $s_j \in \mathbb{R}^L, a_j \in \mathbb{R}^M$. As in the discrete setting, this model $\psi_s(a)$ should return a distribution over actions $P_\psi(a|s)$ given a state s (Figure 5.3).

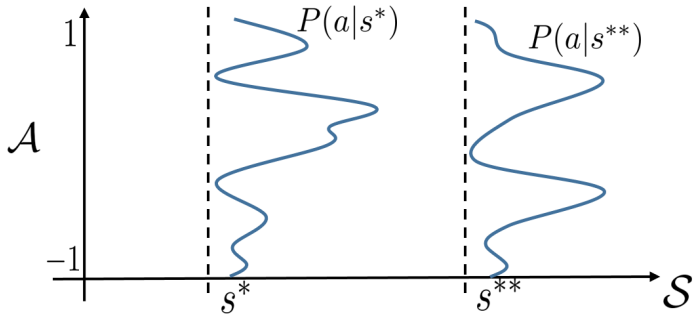


Figure 5.3: An illustration of the continuous action prior distribution $\psi_s(a) \sim P_\psi(a|S)$ at states s^* and s^{**} respectively. Different "peaks" or modes in the distribution represent different action preferences at the given state.

5.2.2 Conditional Density Estimation

Unlike standard regression in supervised learning which seeks to estimate the expectation[4] $\mathbb{E}[y|x]$ for a training set $\{(x_i, y_i)\}_{i=1}^N$, conditional density estimation is concerned with modelling the full distribution $f_\omega(y|x)$. Common approaches for conditional density estimation include Gaussian Process Regression and Deep Neural Networks.

5.2.2.1 Gaussian Process Regression

A Gaussian process is a stochastic process that models a prior over functions [Rasmussen, 2004]. Given some data, a Gaussian process will convert this prior over functions into a posterior that describes the underlying distribution of the data. Under a Gaussian process f, the joint distribution $P(f(\mathbf{x}_1), ..., f(\mathbf{x}_n))$ over a finite set of random variables $f(\mathbf{x}_1), ..., f(\mathbf{x}_n)$ for the corresponding set of input points $\mathbf{x}_1, ..., \mathbf{x}_n$, are Gaussian distributed according to:

$$\begin{bmatrix} f(\mathbf{x}_1) \\ \vdots \\ f(\mathbf{x}_n) \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} M(\mathbf{x}_1) \\ \vdots \\ M(\mathbf{x}_n) \end{bmatrix}, \begin{bmatrix} K(\mathbf{x}_1, \mathbf{x}_1) & \cdots & K(\mathbf{x}_1, \mathbf{x}_n) \\ \vdots & \ddots & \vdots \\ K(\mathbf{x}_n, \mathbf{x}_1) & \cdots & K(\mathbf{x}_n, \mathbf{x}_n) \end{bmatrix} \right) \tag{5.1}$$

where $M(\cdot)$ is the mean function and $K(\cdot, \cdot)$ is a positive definite kernel function, and the co-variance functions respectively. Generally, the co-variance or kernel function

[4] Also referred to as the conditional mean [Hansen, 1994].

$K(\cdot,\cdot)$ is chosen by the domain expert based on their domain knowledge regarding the smoothness of the landscape of observations.

The objective in Gaussian processes is to adjust the parameters of the co-variance function such that the resulting Gaussian process denoted $f(\cdot) \sim \mathcal{GP}(M(\cdot), K(\cdot,\cdot))$

Given the dataset of expert trajectories $\mathcal{D}_\tau = \{(s_j, a_j)\}_{j=1}^{T} = \{\mathbf{S}, \mathbf{a}\}$, the Gaussian process makes the assumption that the actions s can be described by some function of the state s and some i.i.d. (independent identically distributed) noise ϵ i.e. $a = f(s)+\epsilon$, where the noise term ϵ is drawn from a Gaussian distribution with zero mean and variance σ_n^2. The conditional density function $P(\mathbf{a}^*|\mathbf{s}^*)$ at test locations $\mathbf{s}^*$ can be modelled by first assuming a Gaussian process prior over the action space:

$$\begin{bmatrix} \mathbf{a} \\ f(\mathbf{s}^*) \end{bmatrix} \sim \mathcal{N}\left(\mathbf{0}, \begin{bmatrix} K(\mathbf{S},\mathbf{S}) + \sigma_n^2 I & K(\mathbf{S},\mathbf{s}^*) \\ K(\mathbf{s}^*,\mathbf{S}) & K(\mathbf{s}^*,\mathbf{s}^*) \end{bmatrix}\right) \tag{5.2}$$

And computing the mean function $\bar{f}(\mathbf{s}^*)$ and the co-variance function $\text{cov}(f(\mathbf{s}^*))$ according to:

$$\bar{f}(\mathbf{s}^*) = \mathbb{E}\left[f(\mathbf{s}^*) \mid \mathbf{S}, \mathbf{a}, \mathbf{s}^*\right] = K(\mathbf{s}^*,\mathbf{S})\left[K(\mathbf{S},\mathbf{S}) + \sigma_n^2 I\right]^{-1}\mathbf{a} \tag{5.3}$$

$$\text{cov}(f(\mathbf{s}^*)) = K(\mathbf{s}^*,\mathbf{s}^*) - K(\mathbf{s}^*,\mathbf{S})\left[K(\mathbf{S},\mathbf{S}) + \sigma_n^2 I\right]^{-1} K(\mathbf{S},\mathbf{s}^*) \tag{5.4}$$

Such that actions $\mathbf{a}^* = f(\mathbf{s}^*)$ at test locations $\mathbf{s}^*$ can be drawn according to the following posterior distribution:

$$f(\mathbf{s}^*) \mid \mathbf{S}, \mathbf{a}, \mathbf{s}^* \sim \mathcal{N}\left(\bar{f}(\mathbf{s}^*), \text{cov}(f(\mathbf{s}^*)\right) \tag{5.5}$$

Quantities that do not depend on the actual query points $\mathbf{s}^*$ in equations 5.3 and 5.4 are typically pre-computed in order to reduce computation time when making predictions. This posterior (equation 5.5) can thus be used to model continuous action priors $\psi_s(a)$ where:

$$\psi_s(a) \sim \mathcal{N}\left(\bar{f}(s), \text{cov}(f(s)\right) \tag{5.6}$$

However, standard Gaussian Processes tend to be slow when given large state spaces due to the matrix inversion operation in equations 5.3 and 5.4. Furthermore, Gaussian Processes take a long time to evaluate, as such they are undesirable for the purposes

of reinforcement learning since these evaluations for actions have to be performed per time step (i.e. every time we sample an action from the action priors), as such the learning process will be slowed substantially.

5.2.2.2 Deep Neural Networks

Another approach would be to use conventional approaches in using deep neural networks for conditional density estimation to represent the condition density function $\psi_s(a)$ with a parametric Gaussian distribution $\mathcal{N}(\boldsymbol{\mu}_\omega, \boldsymbol{\sigma}_w^2\mathbf{I})$, where $\boldsymbol{\mu}_\omega$ and $\boldsymbol{\sigma}_w^2$ are the predicted mean and variance, and ω are the weights of the network[5]. To satisfy the constraint $\boldsymbol{\sigma}_w^2 \geq 0$, the corresponding logits[6] for the variance are passed through the exponetial function. In deep reinforcement learning, this formulation is commonly used for representing **stochastic policies**[7] [Heess et al., 2015, Duan et al., 2016, Williams, 1992].

The action priors $P(a|s)$ can thus be modelled using $\mathcal{N}(\boldsymbol{\mu}_\omega, \boldsymbol{\sigma}_w^2\mathbf{I})$, where actions are sampled in state s with a mean $\boldsymbol{\mu}_\omega$ and the diagonal covariance matrix $\boldsymbol{\sigma}_w^2\mathbf{I}$.

A naive implementation of the above proposals (i.e. Gaussian process regression and the Deep Neural Network) model the action priors distribution $P(a|s)$ using a uni-modal Gaussian distribution. This has the effect of computing the conditional average over expert actions in a state (as shown in Figure 5.4). However, in some states, the true action prior distribution may be significantly non-Gaussian, as different expert agents are in no way constrained to prefer actions described using a single mean and variance in any particular state. For example, at a T-junction, an optimally behaving self-driving car policy may choose to steer left or right, but taking the average action translates to stopping or going straight (due to a steering angle of zero), which is a bad action. In such cases, the average of expert actions (i.e. optimal actions) at a state does not produce an expert (or even desirable) action. Subsequently transferring such an action prior distribution to other agents in the same environment (i.e. another self-driving cars) is akin to negative transfer, i.e. giving bad advice at the particular state.

[5]This model is structurally similar to the encoder network in variational autoencoders [Kingma and Welling, 2013]

[6]pre-activations

[7]Since the action priors are a stochastic mapping from states to actions, they can be regarded as a policy in reinforcement learning language.

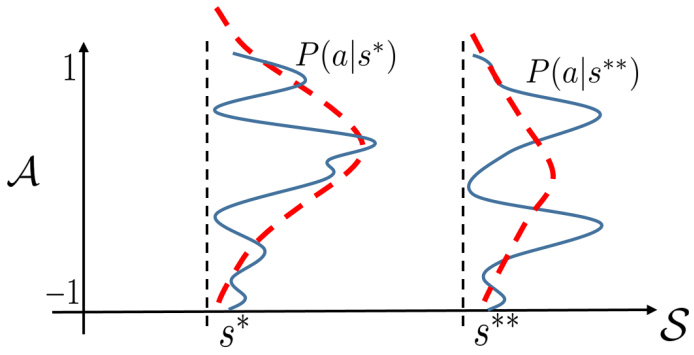


Figure 5.4: Standard Gaussian Processes and Deep Neural Networks return uni-modal distributions (shown by the dotted red line) and tend to "average" (shown red dotted line) over the underlying modes (expert action preferences), resulting in a limited description of the action prior distribution.

Instead, it is more appropriate to treat the problem of modelling action priors in continuous settings as an inverse problem, where a single input state s can be mapped to multiple possible actions a. As such, we propose to model continuous action priors using a multi-modal distribution over actions across the state space, such that we have a mixture model at each state. To achieve this objective, we turn to Mixture Density Networks.

5.2.3 Mixture Density Networks

Deep neural networks are widely-used function approximators that excel at supervised learning tasks. Given data $\{(x_i, y_i)\}_{i=1}^N$, a neural network is able to learn a mapping (i.e. the conditional density function) $P(x|y)$ between inputs and outputs in a straightforward manner. However, when presented with a dataset where the relationship between inputs and outputs is "*one-to-many*"[8], standard neural networks model the conditional average $\mathbb{E}[y|x]$ over the set of possible solutions [Bishop, 1994]. This results in a poor fit of the data (see Figure 5.5a), and consequently poor performance on the given regression task, since the conditional average only provides a limited statistic concerning the true probability density function.

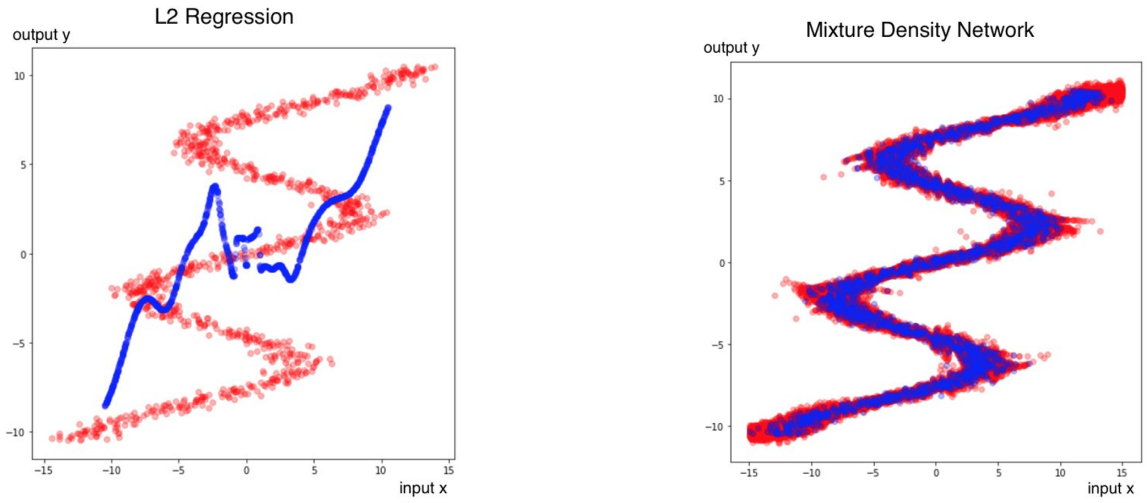
Mixture density networks [Bishop, 1994] address this limitation by modelling the conditional density function in the target data $P(y|x)$ as a parametric mixture model:

[8] A single input has multiple possible outputs.

$$P_z(y|x) = \sum_{k=1}^{K} \pi_k(x)\phi_k(y|x) \tag{5.7}$$

Where $P_z(y|x)$ is a linear combination of K kernel functions $\phi_k(y|x)$ (i.e. the mixture model components) and their corresponding mixing co-efficients $\pi_k(x)$. The parameters of the mixture density $\boldsymbol{z}$ are modelled by a neural network. The neural network takes an input vector x and produces the parameters $\boldsymbol{z}$ of the mixture density model from which the output vector y can be sampled. Combining the representational power of a neural network with a mixture model in this way, results in a neural network that can model arbitrary conditional distributions [Bishop, 1994].

Mixture density networks (see Figure 5.6) are a model for performing conditional density estimation with neural networks, thus extending the conventional neural networks to handle multiple output cases, including noisy multi-target data (see Figure 5.5b).



(a) (b)

Figure 5.5: a) A comparison between a conventional neural network (a) and a mixture density network (b) in a regression task, where in addition to being noisy, the training data[9](red circles) has multiple possible outputs for a single input. The conventional neural network (a) trained using the Mean Square Error (MSE) loss predicts conditional averages (blue circles) which do not reflect the underlying distribution. However, the mixture density network (b) is able to accurately model the underlying distribution.

[9]Generated using $x = 7.0sin(0.75y) + 0.5y + \epsilon$, images extracted from:
`http://blog.otoro.net/2015/11/24/mixture-density-networks-with-tensorflow/`
(*Accessed: 22-03-2019*)

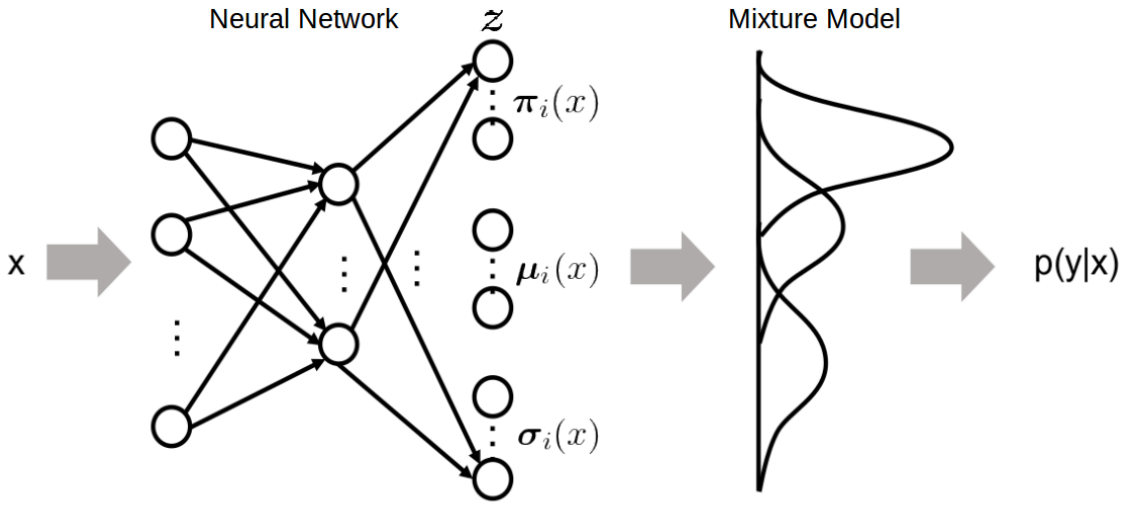


Figure 5.6: The mixture density network. Image adapted from [Vossen et al., 2018].

Although other kernel functions $\phi_k(y|x)$ are possible, Bishop [1994] chose to formulate the Mixture Density Network using Gaussian kernels of the form:

$$\phi_k(y|x) = \frac{1}{(2\pi)^c \sigma_k(x)^c} \exp\left\{ -\frac{\|y - \mu_k(x)\|^2}{2\sigma_k(x)^2} \right\} \tag{5.8}$$

As such, the mixture density model has parameters $\boldsymbol{z} = \{\pi_k(x), \mu_k(x), \sigma_k(x)^2\}$, where $\pi_k(x)$ are the mixing co-efficients, $\mu_k(x)$ are the means, and $\sigma_k(x)^2$ are the variances. To ensure that the predicted parameters $\boldsymbol{z}$ (i.e. the neural network output) produce a valid probability distribution, Bishop [1994] applied the following manipulations to the parameter vector $\boldsymbol{z}$:

1) the mixing coefficients are a categorical distribution over the components of the mixture density network. Therefore they must satisfy:

$$\sum_{k=1}^{K} \pi_k(x) = 1, \qquad 0 \leq \pi_k(x) \leq 1 \tag{5.9}$$

This constraint can be achieved using the softmax activation function:

$$\pi_k(x) = \frac{\exp(z_k^\pi)}{\sum_{i=1}^{N} \exp(z_k^\pi(i))} \tag{5.10}$$

2) The variances must satisfy:

$$\sigma_k(x)^2 \geq 0 \tag{5.11}$$

This constraint can be implemented using the exponential function as follows:

$$\sigma_k(x)^2 = \exp(z_k^\sigma) \tag{5.12}$$

3) The means can be real-valued. Therefore, they can be represented directly by the network outputs:

$$\mu_k(x) = z_k^\mu \tag{5.13}$$

For training the mixture density network, Bishop [1994] used the following negative log likelihood loss function:

$$E^q = -\ln \left\{ \sum_{k=1}^{K} \pi_k(x^q) \phi_k(y^q | x^q) \right\} \tag{5.14}$$

To model K (a hyperparameter) mixture model components, we require the length of $\boldsymbol{z}$ to be $|\boldsymbol{z}| = (1 + 2n)K$, where n is the dimension of the observations y. Under this formulation, higher dimensions, i.e. $n > 1$, require the use of the multivariate Gaussian distribution kernels with diagonal co-variance matrices [Bishop, 1994].

However, diagonal co-variances matrices, by definition, place restrictions on the shape of the resulting Gaussian distribution. As such, we prefer to use a full co-variance matrix in this work. However, obtaining a valid co-variance matrix from a neural network is difficult. Williams [1996] solved this difficulty by first exploiting the symmetry of the co-variance matrix which dictates that a co-variance matrix only has at most $n(n+1)/2$ independent entries. Thus, implying that a neural network only has to output enough parameters to construct an upper or lower triangular matrix. Williams [1996] then defined the inverse of the co-variance matrix using Cholesky factorization of a symmetric positive definite matrix as $A^T A$ where A is an upper triangular matrix i.e.:

$$\Sigma^{-1} = A^T A = \begin{bmatrix} \alpha_{11} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ \alpha_{n1} & \cdots & \alpha_{nn} \end{bmatrix}, \begin{bmatrix} \alpha_{11} & \cdots & \alpha_{1n} \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \alpha_{nn} \end{bmatrix} \tag{5.15}$$

To ensure a valid co-variance matrix, the following restrictions were imposed on the elements of the upper triangular matrix A:

$$\alpha_{ii} = \exp(z_k^{\Sigma\_diag}) \tag{5.16}$$

$$\alpha_{ij} = z_k^{\Sigma} \qquad , \quad i \neq j \tag{5.17}$$

Unlike equation 5.12 which only uses a single vector z_k^{σ} from the neural network to construct the co-variance matrix. The full co-variance formulation requires two vectors $z_k^{\Sigma\_diag}$ for the diagonal elements of A and z_k^{Σ} for the off-diagonal elements of A. This formulation allows for mixture density components of the form:

$$\phi_k(y|x) = (2\pi)^{-n/2}|\Sigma|^{-1/2}\exp\left\{-\frac{1}{2}(y-\mu_k)^T\Sigma^{-1}(y-\mu_k)\right\} \tag{5.18}$$

where:

$$|\Sigma|^{-1/2} = \prod_{i=1}^{n}\alpha_{ii} \tag{5.19}$$

This leads to a full co-variance matrix formulation of the mixture density network which can be trained using the negative log likelihood loss function (equation 5.14).

In this work we use this mixture density network to model continuous action priors, $\psi_s(a) \sim P_{MDN}(a|s)$. To sample actions from the mixture density network given a state s, we pass s into the network, construct a Gaussian mixture model from the resulting parameters $\boldsymbol{z}$, select a component k in the mixture according to the categorical distribution over the mixing co-efficients $\pi_k(x)$, and finally sample an action from the selected mode's Gaussian distribution parameters $\mu_k(x)$ and $\sigma_k(x)$.

In addition to being able to model complex, multi-modal distributions, the mixture density network has the additional benefit of being flexible with regards to the internal neural network architecture. This means that the multi-layer perceptron architecture used in this work can be replaced with other neural network architectures such as recurrent neural networks [Brando Guillaumes, 2017] or convolutional neural networks [Iso et al., 2017].

Other methods [Ostrovski et al., 2017, Bellemare et al., 2016] have been successful in using psuedo-counts in continuous environments for the purposes of constructing better exploration policies. These methods use density models purely for "counting" state

visitations. The resulting model can then be used to bias new agents towards states with the highest visitation "counts". In this work however, we maintain "counts" of the actions executed by other agents in a particular state. As such, our exploration policy is directly conditioned on expert behaviour (i.e. action preferences) instead of expert state preferences.

.

5.2.4 Learning with Continuous Action Priors

Having discussed the knowledge compression step in our proposed solution i.e. fitting a mixture density network over expert trajectories, we now discuss how to transfer the prior knowledge modelled by the mixture density network to agents learning new tasks in the same environment.

As in [Rosman and Ramamoorthy, 2012], we also administer our proposed mixture density network action priors as an exploration strategy. Where, as in ϵ-greedy exploration (see section 2.5 in chapter 2), we maintain the ϵ parameter and use it to control exploration (sampling actions from the mixture density network) and exploitation (sampling actions from the trained policy μ_θ). ϵ is then decayed over time, causing the agent to depend on expert advice less and start relying on its trained policy (i.e. the target policy). Once the agent has successfully solved a task in the environment, the resulting policy can be used to generate optimal trajectories which can be added to the dataset $\mathcal{D}$, which in turn can be used to update the action prior distribution. Thus enabling continued learning. A general overview of our proposed method is provided in Figure 5.7:

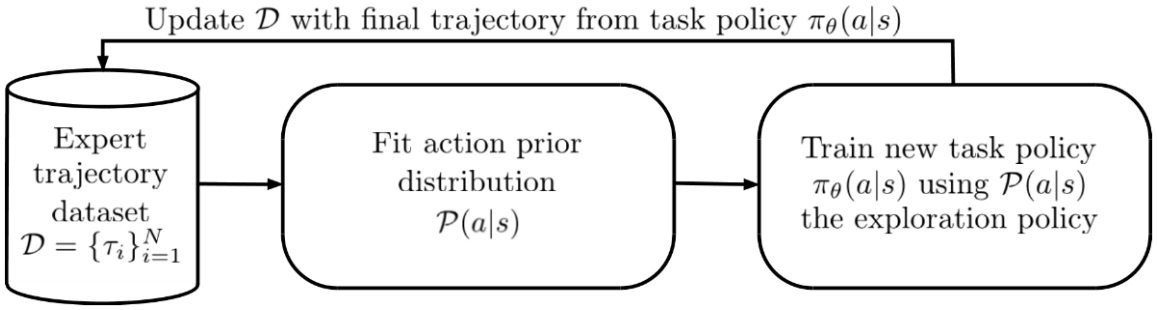


Figure 5.7: The mixture density network action priors framework.

We thus present the Deep Deterministic Policy Gradient with Action Priors (DDPG-CAP) algorithm where we have modified the action selection step (line 5) to incorporate the mixture density network action priors.

Algorithm 7 Deep Deterministic Policy Gradient with Action Priors

1: Input: initial policy parameters θ, Q-function parameters ϕ, empty replay buffer $\mathcal{D}$
2: Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$
3: **repeat**
4: Observe state s
5: Select action $a = \begin{cases} \text{clip}(\mu_\theta(s), a_{Low}, a_{High}) & \text{w.p.} \quad 1-\epsilon \\ \psi_s(a) & \text{w.p.} \quad \epsilon \end{cases}$
6: Execute a in the environment
7: Observe next state s', reward r, and done signal d to indicate whether s' is terminal
8: Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
9: If s' is terminal, reset environment state.
10: **if** it's time to update **then**
11: **for** however many updates **do**
12: Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
13: Compute the temporal difference target

$$y(r, s', d) = r + \gamma(1-d)Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$$

14: Update Q-function by one step of gradient descent using

$$\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_\phi(s, a) - y(r, s', d)\right)^2$$

15: Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_\phi(s, \mu_\theta(s))$$

16: Update target networks with

$$\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1-\rho)\phi$$
$$\theta_{\text{targ}} \leftarrow \rho\theta_{\text{targ}} + (1-\rho)\theta$$

17: **end for**
18: **end if**
19: **until** convergence

5.3 Summary

This chapter has presented a solution for achieving multi-task transfer for reinforcement learning tasks defined in continuous environments. This solution was presented as an extension of the action priors framework Rosman and Ramamoorthy [2012] using mixture density networks. Other possible ways of extending the action priors framework are discussed and dismissed with sufficient motivations.

We select mixture density networks for modelling action priors over Gaussian processes mainly for their flexibility in handling different types of datasets (i.e feature vectors, images, etc) as well as their ease of implementation, i.e. many deep learning frameworks come with intuitive wrappers for implementing neural network architectures while there are only few software packages for implementing Gaussian processes (and even fewer when considering more advanced Gaussian processes models such as sparse Gaussian processes).

In the following chapter (chapter 6), we present experiments which serve as empirical evidence that our proposed solution mixture density network action priors, is able to achieve safer learning, a jump start in performance when learning, as well as increased performance compared to the ϵ-greedy DDPG agent.

Chapter 6

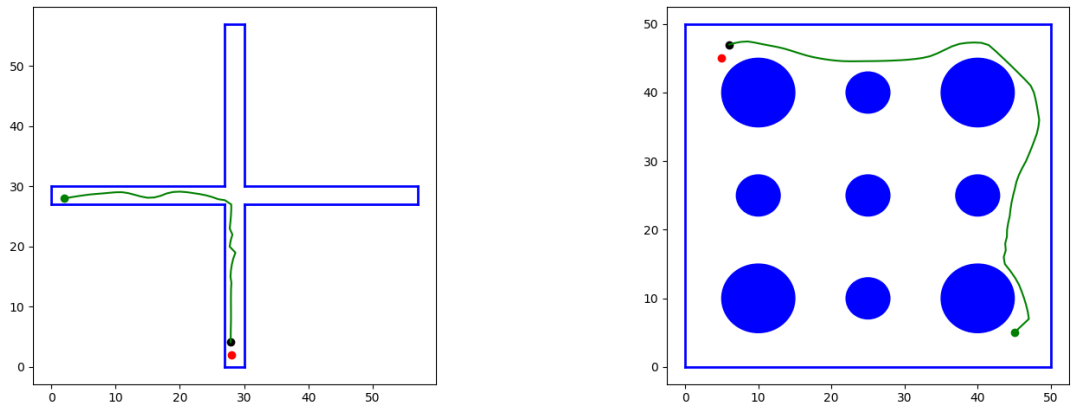
Experiments

In this chapter we evaluate various properties of our proposed method through experiments defined in two custom continuous state-action domains, namely the cross-intersection domain and the pothole domain. We provide formal descriptions of these environments under the Markov Decision Process framework in section 6.1, discuss how mixture density network action priors are learned for each of these environments in section 6.2, and provide experimental results for evaluating knowledge transfer as well as safety in sections 6.3.1 and 6.3.2 respectively. In section 6.4, we discuss our observations from experiments and offer recommendations based on these. Finally, we summarise the chapter in section 6.5.

6.1 Learning Environments

In this section, we provide descriptions for the continuous environments used throughout the chapter for learning and investigating the effect of applying continuous action priors when learning new tasks. Although there are widely available learning environments such as OpenAI gym [Brockman et al., 2016] and mujoco [Todorov et al., 2012], the continuous state-action environments they offer typically do not include mechanisms for evaluating the safety of the learning procedure. As such, we construct custom environments and define the safety measure as the number of collisions with obstacles in the environment during learning. We use these environments to empirically show that implementing continuous action priors using mixture density networks results in action prior distributions which retain the useful properties of their discrete

counterpart [Rosman and Ramamoorthy, 2012], but have the benefit of being compatible with continuous state-action environments. Namely, the cross-intersection environment and the pothole environment in Figure 6.1.



(a) Cross-intersection environment

(b) Pothole environment.

Figure 6.1: In both environments, the agent is tasked with navigating from a start state (●) to a goal state (●). The agent's reward is computed based on its distance from the goal state, such that the closer the it gets to the goal, the higher the reward it receives. However, the agent also receives penalties (large negative rewards) for "colliding" with the boundaries of the environment as well as (in the case of the pothole environment) visiting bad states ("the potholes") represented by the blue circles. We depict examples of successful trajectories by a green line starting at the start state (●) and ending at the black dot(●) since the episode is terminated when the agent is within some radius around the goal state (●).

6.1.1 Cross-intersection Domain

This environment (Figure 6.1a) is a continuous state-action MDP where the agent has to learn a policy for navigating from a given start state and a given goal state. Different tasks in this environment are represented using different start and goal states.

In this work, this environment (Figure 6.1a) is used to demonstrate the ability of mixture density network action priors to capture the expert behavioural preferences in different regions of the state space. More specifically, we expect that expert agents in the horizontal sections of the environment mostly favour continuous actions that generally lead to the left or to the right, agents in vertical sections mostly favour up or

down actions, and agents at the intersection favour both "up-down" and "left-right" actions.

The formal description of this environment is as follows:

- **States:** The continuous 2D position (s_x, s_y) an agent can occupy in the environment[1]. As such, the state space is bounded, i.e. $s_x, s_y \in [0, 56]$.
- **Actions:** The continuous 2D step (a_x, a_y) an agent can take in the environment. For simplicity, the actions are bounded such that $a_x, a_y \in [-1, 1]$.
- **Reward function:** This is given by the sum of the negative of the distance between the agent's current state (s_x, s_y) and the goal (G_x, G_y), as well as the penalty (large negative reward incurred for "colliding" with the wall). More specifically, the agent is given a penalty $r_p = -5$ whenever it collides with the wall i.e. if one or both s_x, s_y are assigned boundary values. Otherwise the penalty is zero i.e. $r_p = 0$.
- **Transition function:** The simple one-step transition given by adding the selected action to the current state. i.e. $(s'_x, s'_y) = (s_x + a_x, s_y + a_y)$. If the transition $(s_x + a_x, s_y + a_y)$ places the agent outside the boundaries of the environment, then that transition is modified such that the affected state component(s) takes on the boundary value.

Later in the chapter, we provide experiments that show that action priors lead to improved safety as well as improved transfer according to some of the various evaluation metrics introduced in [Taylor and Stone, 2009].

6.1.2 Pothole Domain

Similar to the previous environment, the objective in this environment (Figure 6.1b) is to find a policy that navigates from a given starting point (s_{x0}, s_{y0}) to a given goal state (G_x, G_y). The difference here is that the agent also has to avoid bad state regions ("the potholes" depicted using blue circles in Figure 6.1b).

[1] Within the blue boundaries

We introduce this environment mainly to investigate how the proposed mixture density network action priors affect the safety during training. We define safety as the sum of the number of collisions with the boundaries and the number of times the agent visits the bad regions. This way, agents that have fewer collisions with the boundaries and fewer pothole state visitations during training can be regarded as learning safer compared to agents where these quantities are higher.

Similar to the previous environment, we formally describe the environment according to the MDP formalism introduced in chapter 2 as follows:

- **States:** The continuous 2D position (s_x, s_y) an agent can occupy in the environment[2]. As such, the state space is bounded, i.e. $s_x, s_y \in [0, 50]$.

- **Actions:** The continuous 2D step (a_x, a_y) an agent can take in the environment. For simplicity, the actions are bounded such that $a_x, a_y \in [-1, 1]$.

- **Reward function:** This is given by the sum of the negative of the distance between the agent's current state (s_x, s_y) and the goal (G_x, G_y), as well as the penalty (large negative reward incurred for "colliding" with the wall). More specifically, the agent is given a penalty $r_p = -5$ whenever it collides with the wall i.e. if one or both s_x, s_y are 0 or 50. Otherwise the penalty is zero i.e. $r_p = 0$.

  To encourage agents to learn policies that avoid the potholes, an additional term $r_{rep} = -\exp\left(-\frac{1}{2}\left(\frac{d_{sp}^2}{2R_p}\right)^2\right)$ is added to the reward for each pothole in the environment. This depends on: 1) the distance between the agent's current state (s_x, s_y) and the centre of the p^{th} pothole (p_x, p_y). 2) the radius of the p^{th} pothole R_p. This term specifies a repulsive field that is strongest (more negative reward) at the centre of the pothole, weakest at the pothole boundary, and negligible outside the pothole. This repulsive field concept is commonly used in optimal path planning literature to encode obstacles into the planning problem [Park et al., 2001].

- **Transition function:** The simple one-step transition given by adding the selected action to the current state. i.e. $(s'_x, s'_y) = (s_x + a_x, s_y + a_y)$. If the transition $(s_x + a_x, s_y + a_y)$ places the agent outside the boundaries of the environment,

[2]Within the blue boundaries

then that transition is modified such that the affected state component(s) takes on the boundary value.

6.2 Learning Continuous Action Priors

Having introduced our learning environments, we now discuss the process for learning mixture density network action priors for each environment respectively. We also provide evidence that our proposed solution "mixture density network action priors" succeeds in capturing the behavioural preferences of expert agents in the environment. Similar to Rosman and Ramamoorthy [2012], the notion "expert agent" in an environment can refer to optimal trajectories collected from either: 1) a trained deep reinforcement learning agent, 2) a human expert, or 3) a planning algorithm. For each of the following we explicitly state the source of expert trajectories.

6.2.1 Data Collection

Cross-intersection Environment

As mentioned in chapter 5, section 5.2, one challenge with modelling action priors in the continuous setting is the multi-modal nature of expert action preferences (see Figure 5.3). In other words, different experts may prefer different actions at the same state or state region according to their reward functions. This makes it hard to fit a model for optimal actions across multiple experts at a state. We further remarked that due to this reason, we cannot rely on naive implementations of Gaussian processes or conventional Gaussian neural networks and instead presented the mixture density network as a model which is capable of modelling this multi-modal nature of the expert preferences.

To test our claims, we (the human experts) manually construct a dataset of trajectories in the cross-intersection environment where state-action pairs are such that, for each state we choose, we specify multiple optimal actions with different directions. For example, for a single state in the horizontal section of the environment, two optimal actions are specified, a pure "left" action ($a_x = -1, a_y = 0$) and a pure "right" action ($a_x = 1, a_y = 0$). Similarly, a state in the vertical section in the environment has optimal actions: "up" ($a_x = 0, a_y = 1$) and "down" ($a_x = 0, a_y = -1$). This

procedure was repeated at multiple states, to create a dataset with 20000 entries (state-action pairs). We visualise this dataset in Figure 6.2:

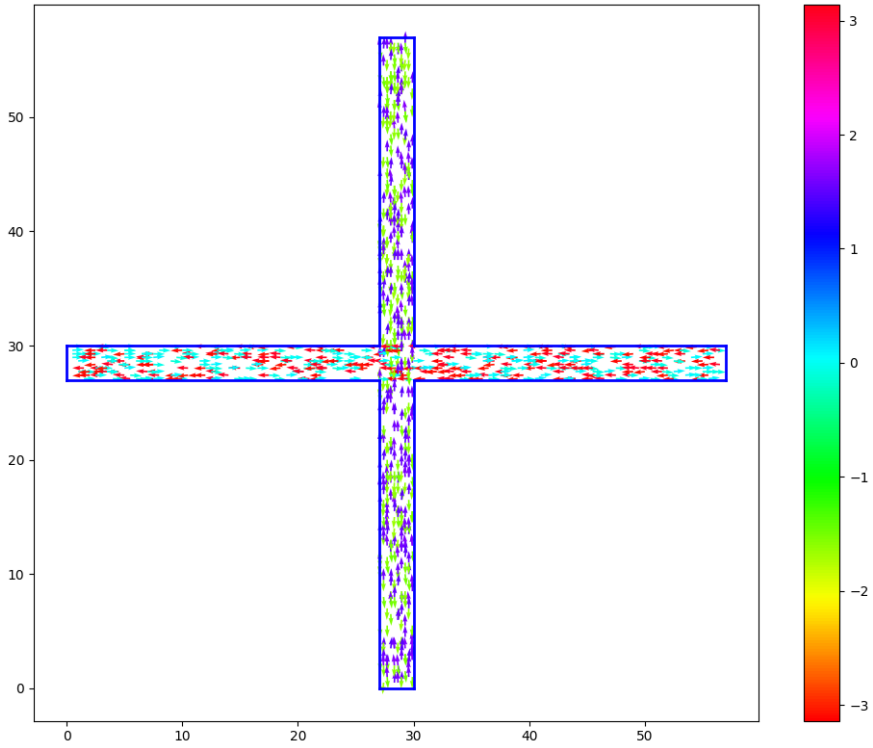


Figure 6.2: A visualisation of human expert data in the cross-intersection environment. For visual clarity, only a subset of the data is shown.

In Figure 6.2, Figure 6.3, Figure 6.4, and Figure 6.6 the arrow tails represent the state and the arrow itself represents the corresponding expert action (i.e. the direction that the agent would move towards if it were to take the action represented by the current arrow). The arrow colour then represents the corresponding orientation or heading of the arrow β where $\beta \in [-\pi, \pi]$ (see colour bar).

Pothole Environment

Since manually specifying expert actions for this environment is difficult, we collect expert trajectories from DDPG (see algorithm 5) agents trained on different tasks with randomly selected start and goal states. We classify the resulting policies as experts if they produce trajectories that do not "collide" with the boundaries or visit the bad state regions (pothole states). Out of 500 trained DDPG agents (50 different

tasks × 10 trials on each task), only 138 DDPG agents meet our criteria. We then sample trajectories from these agents and concatenate them to create a dataset of state-action pairs of size 26328. We visualise this dataset in Figure 6.3.

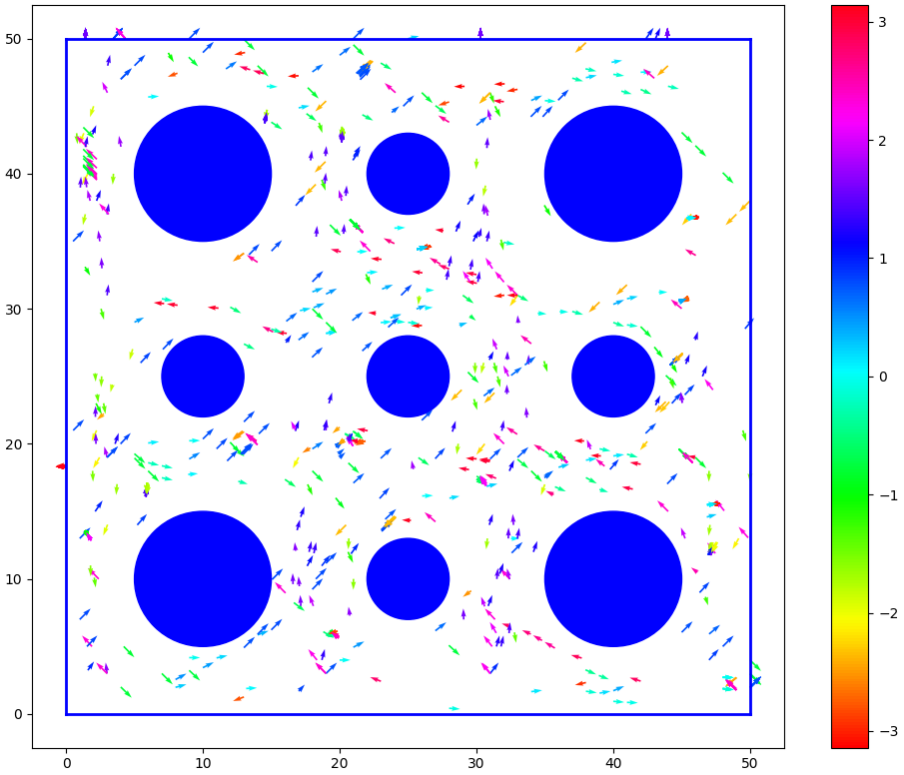


Figure 6.3: A visualisation of trained DDPG agent data in the cross-intersection environment. For visual clarity, only a subset of the data is shown.

In some cases, the tails (i.e. the states of the agent) are located close the boundaries of the environment, this causes the arrow heads to peer outside of the environment. This phenomenon is a consequence of visualising actions as arrows and should not be mistaken for agents located beyond the boundaries of the environment.

6.2.2 The Quality of Learned Continuous Action Priors

To learn the mixture density network action priors, we split each of the collected datasets into training and testing datasets. We then train the mixture density network until convergence on each environment's training dataset respectively. In this section

we provide visualisations of the learned action prior distributions at states selected from the testing datasets to demonstrate that the mixture density network action priors model can generalise to unseen data and as well as capture expert behavioural preferences at various regions in the state space.

Cross-intersection Environment

In this section, we provide visualisations of the learned mixture density network action priors for the cross-intersection environment. First, we visualise actions sampled from the learned continuous action priors model (Figure 6.4) in the environment for visual comparison with our training data (Figure 6.2), and then visualise the actual mixture density model at select states in the environment (Figure 6.5). In this environment, we set the number of components in the mixture density network K to 2.

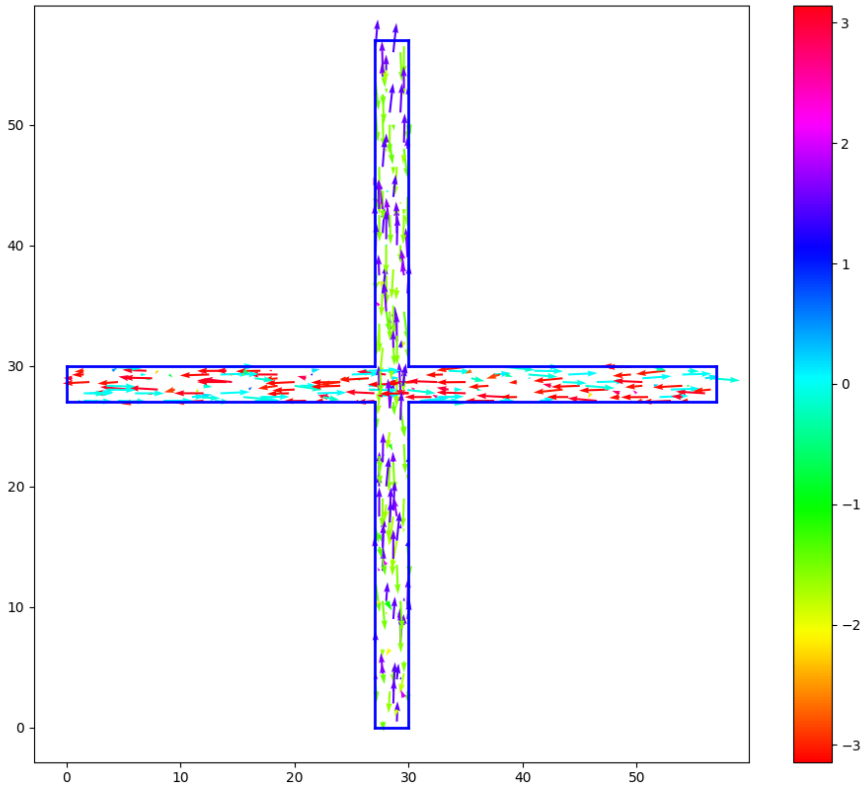
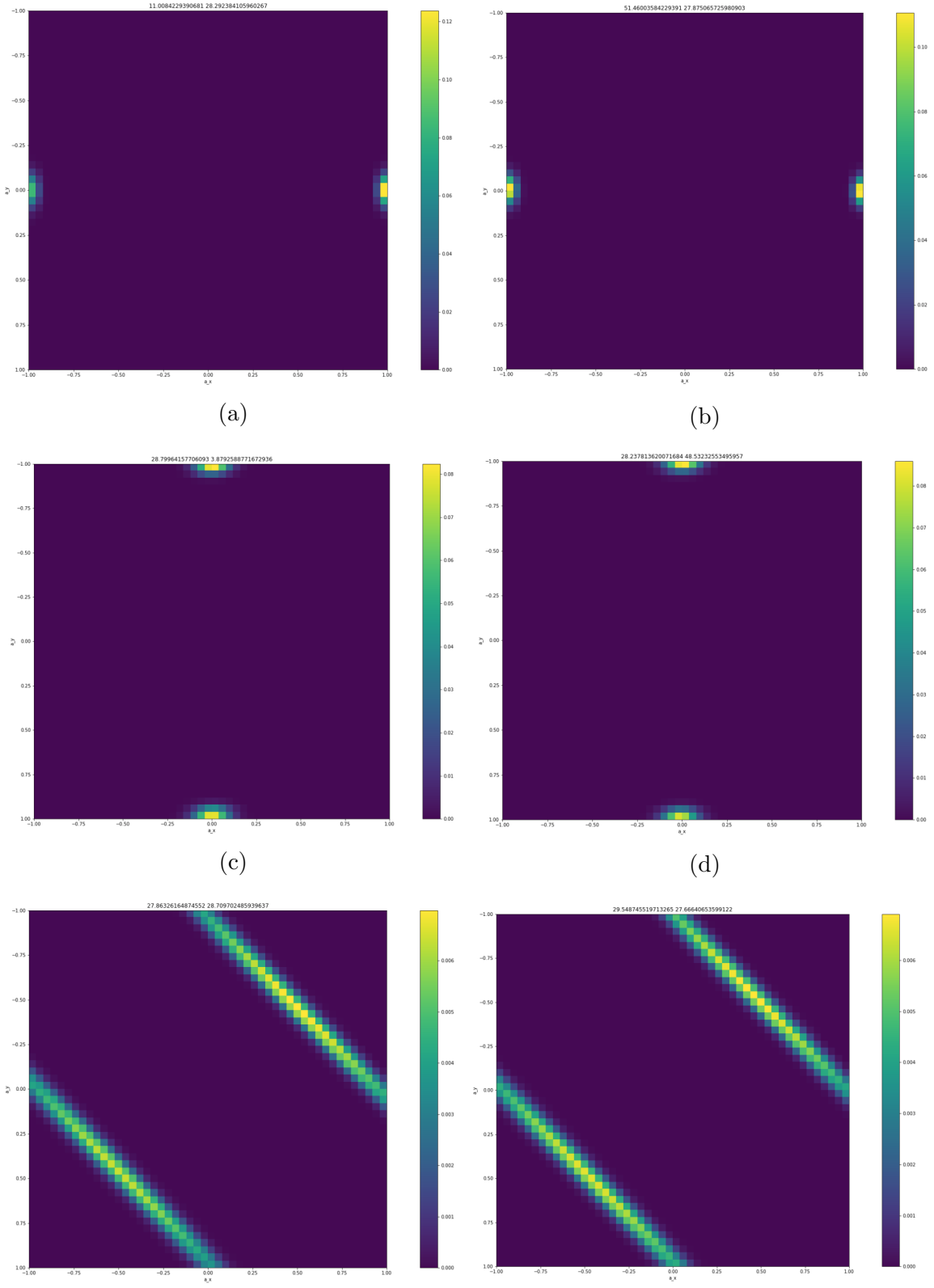


Figure 6.4: A visualisation of actions sampled from the learned mixture density network action priors model in the cross-intersection environment.



(a) (b)

(c) (d)

(e) (f)

Figure 6.5: A visualisation of the learned action prior distribution at select states in the cross-intersection environment.

In Figure 6.5, we visualise the learned action priors (Gaussian mixture densities) for the cross-intersection domain at various states. The y-axis represents "up-down" actions a_y and the x-axis represents "left-right" actions a_x, the colorbar represents the probability of an action pair (a_x, a_y) being sampled. In this case a) and b) are action priors at states in the horizontal section of the environment, and thus restrict other actions besides $\boldsymbol{a} = (-1, 0)$ and $\boldsymbol{a} = (1, 0)$ (left and right movement). Likewise, c) and d) encode similar restrictions along the y-axis. e) and f) are states at the centre of the environment and allow for actions in both directions.

Pothole Environment

In this section, we visualise the learned mixture density network action priors for the pothole environment. As in the previous environment, we first visualise actions sampled from the learned action prior distribution (Figure 6.6), and then visualise the action prior distribution at various states in the environment (Figure 6.7). In this environment, we set the number of components in the mixture density network K to 4.

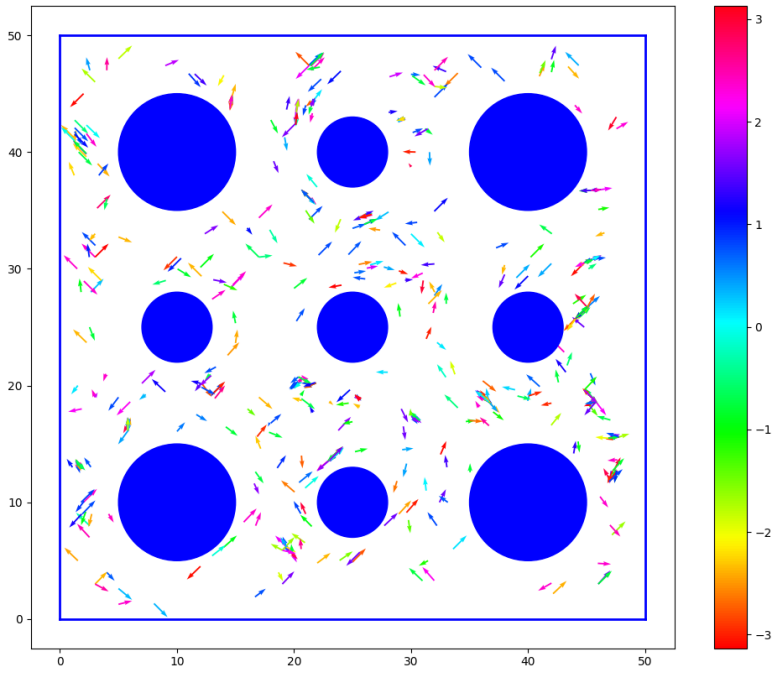


Figure 6.6: A visualisation of actions sampled from the learned mixture density network action priors model in the pothole environment.

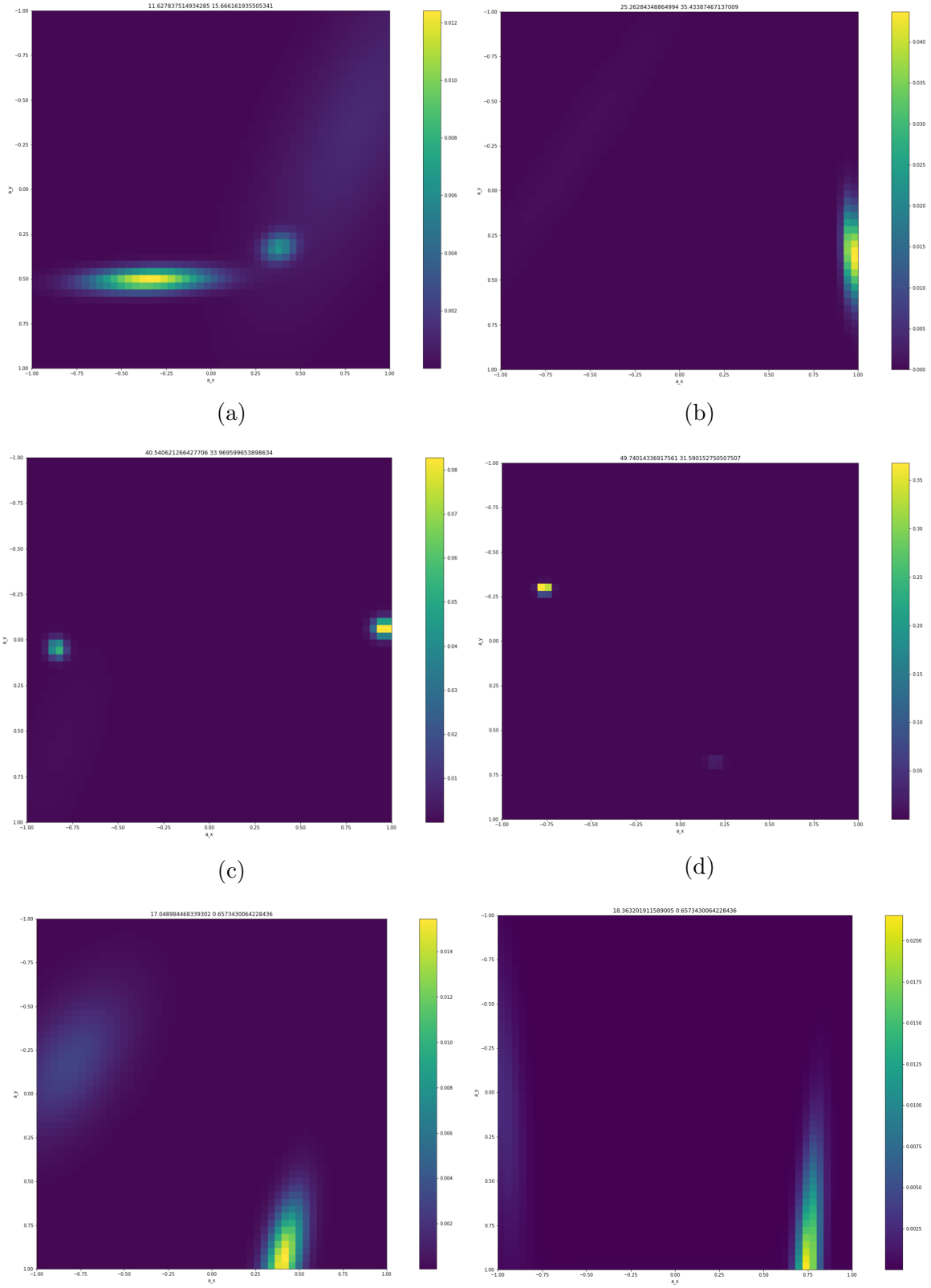


Figure 6.7: A visualisation of the learned action prior distribution at select states in the pothole environment.

In figure 6.7, the learned action priors (Gaussian mixture densities) for the pothole environment at various states. The vertical axis represents actions along the y-axis a_y and the horizontal axis has actions along the x-axis a_x, the colorbar represents the probability of an action pair (a_x, a_y) being sampled. For example, c) can be interpreted as: "left" and "right" actions are most probable at state $s \approx (40, 33)$. As such, this action prior avoids the top-left pothole in the environment.

6.3 Applying Learned Continuous Action Priors

So far, we have shown that mixture density networks are able to successfully compress knowledge (i.e. trajectories from multiple experts) into a single model and represent continuous action priors adequately. In this section we discuss the administration of this prior knowledge model into the learning process of unseen[3] target tasks. In particular, we administer continuous action priors according to Deep Deterministic Policy Gradient with Action Priors algorithm discussed in chapter 5, section 5.2.

This algorithm applies the learned continuous action prior distribution as a exploration policy in an ϵ-greedy manner. Using this algorithm, we train DDPG agents on new tasks in the environment and compare them against standard DDPG agents which learn vanilla ϵ-greedy exploration.

To show that our proposed solution: mixture density network action priors, retains the useful properties of discrete action priors (i.e. improved learning and safer learning). We evaluate transfer as well as safety in the following subsections.

[3]In this context, unseen refers to new tasks that have not been previously solved by an expert in the environment.

6.3.1 Evaluating Transfer

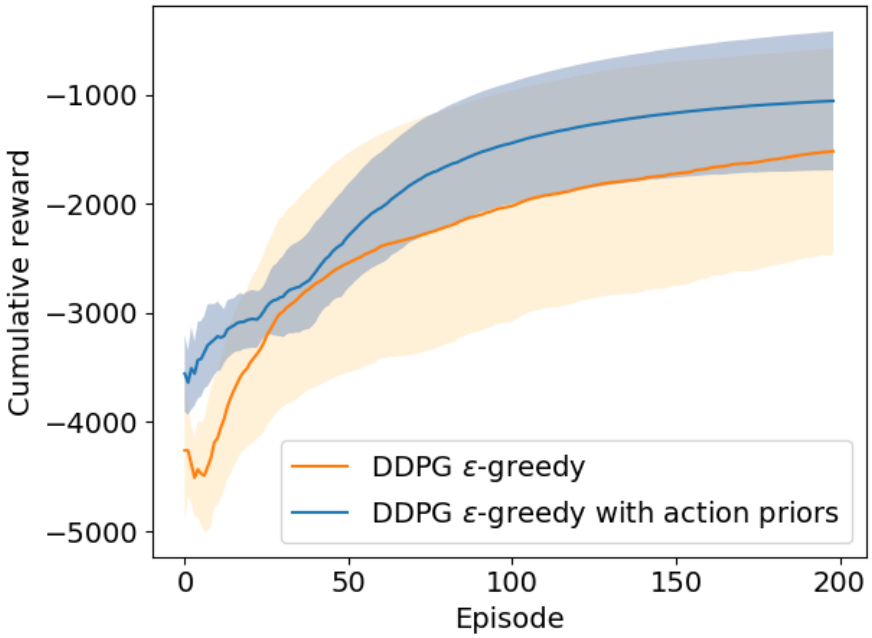


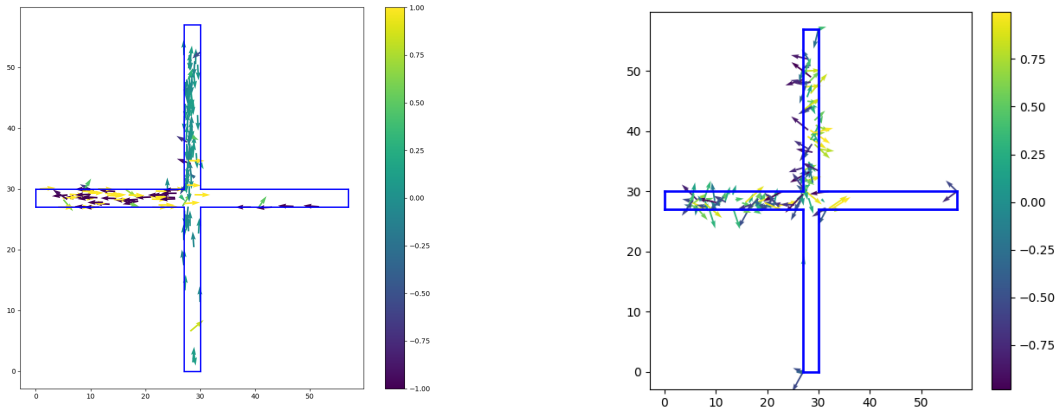
Figure 6.8: Results showing the cumulative reward per episode averaged over 5 DDPG agents with different random seeds in the cross-intersection environment.

In Figure 6.8, we present a performance comparison between the standard ϵ-greedy Deep Deterministic Policy Gradient algorithm and the Deep Deterministic Policy Gradient with Action Priors algorithm on an unseen test task in the cross-intersection environment. In this experiment, we observe an increase in asymptotic performance as well as a jump start [Taylor and Stone, 2009] in performance, implying that our modified agent learns a better overall task policy. This shows that our action-prior policy is a viable technique for multi-task transfer in continuous control environments.

As further evidence to support our observations in Figure 6.8, we present a visualisation in Figure 6.9 of actions sampled from both deep reinforcement learning agents during the learning of the target task in Figure 6.8. To successfully complete the task, the agent had to navigate from the left-end of the environment to the top-end.

In the visualisation (Figure 6.9), the modified DDPG agent explores according to expert action preferences that are relevant in that particular region of the state space (Figure 6.9a), unlike the standard DDPG algorithm which uses a random exploration policy (Figure 6.9b) and prescribes random actions at all the queried regions

of the state space. The mixture density network adequately models the expert action preferences and prescribes correct actions for different regions of the state space i.e. "left-right" actions in the horizontal section of the environment, "up-down" actions in the vertical sections, and both at the intersection.



(a) (b)

Figure 6.9: a) Cross-intersection environment mixture density network exploration policy. b) Cross-intersection environment random exploration policy. The colour of the arrows here, represents the direction of the specified action.

6.3.2 Evaluating Safety

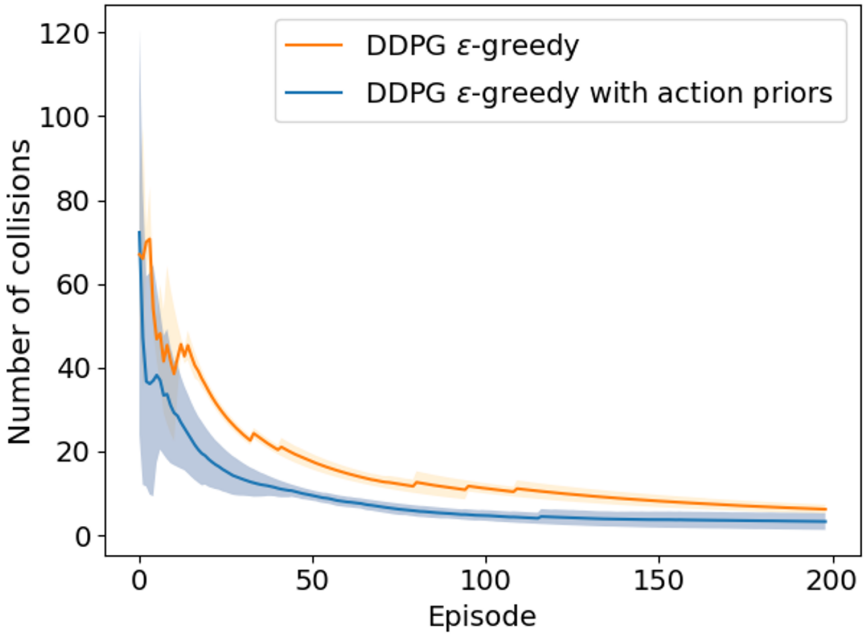


Figure 6.10: Results showing the number of collisions per episode averaged over 2 DDPG agents with different random seeds in the pothole environment.

This experiment (Figure 6.10) presents a safety comparison between both DDPG algorithms. We expect this number to start high and decrease as the agent learns to avoid collisions since they result in heavy negative rewards. Between the two agents, our method (shown in blue) results in a fewer number of collisions with the environment, this shows that exploring using our continuous action prior policy results in a safer deep reinforcement learning agent.

6.4 Discussion

6.4.1 Selecting the Number of Mixture Model Modes K

For the given environments, we select K according to a visual assessment of how close the resulting mixture density network was able fit the expert data. As such, we found $K = 4$ to be sufficient for the pothole domain and $K = 2$ for the cross-intersection environment.

The expressiveness of a mixture density network largely depends on K. If K is small, then the mixture density network is forced to average over modes and cram support into the remaining modes. For example, when $K = 1$ we recover the Gaussian neural network discussed in chapter 5, section 5.2. Conversely, when K is large, the mixture density network attempts to allocate a mode (i.e. a kernel) for each observation which could result in poor generalisation.

6.4.2 Selecting ϵ

ϵ is a hyperparameter that controls the proportion of exploratory actions to actions from the trained policy (refer to chapter 2, section 2.5). Typically ϵ is selected such that $\epsilon \in [0, 1]$ and then decayed during learning. In our proposed method, ϵ controls the proportion of applying expert actions and applying actions from the trained policy. In other words, agents that learn with a high value for ϵ prioritise expert actions and produce policies that select actions in a way similar to experts. Such agents are strongly biased to follow expert behaviours and may be hindered from learning their own tasks as a result.

On the other hand, a low ϵ value results in an agent that prioritises actions from the trained policy. Exploiting the trained policy too often does not leave room to discover other possible solutions. This is especially the case for deterministic policy algorithms such as the Deep Deterministic Policy Gradient algorithm used in this work. After trying out multiple values for ϵ in the two learning environments, $\epsilon = 0.6$ was found to work best. This value was then kept fixed for all evaluation experiments and decayed such that the final episode (i.e. episode 200) had $\epsilon = 0$.

6.4.3 Conflicting Reward Functions

The mixture density network action priors in this work are presented such that, different tasks are characterised using the same state space, action space and transition function i.e. tasks only differ by reward function. However, it is possible for any two agents to have conflicting reward functions. For example, the first agent receives positive reward for avoiding obstacles while the successive agent receives positive reward for colliding with obstacles.

Such a scenario presents a challenge since action priors learned for the first agent produce undesirable actions for the successive agent, and would most likely lead to suboptimal behaviour.

Dubey et al. [2018] showed that humans also suffer from the same problem. They showed that when playing video games, humans are biased towards interacting with objects that visually stand out from the background first, compared to objects that are concealed in the background. For example, humans did poorly in games that had objects that were the same texture as the background since they assumed that such objects were non-existent. However, a deep reinforcement learning algorithm that was trained from scratch did not have these biases and was able to achieve a better score than humans for this particular version of the game.

One possible way to address this problem is to explicitly model task similarity such that action priors are only constructed from sufficiently related tasks. However, this extension is outside the scope of this work.

6.5 Summary

In this chapter, we empirically show that mixture density networks are able to successfully port action priors to continuous domains. We conduct experiments where we learn an exploration policy from expert data, that explores according to expert preferences and thus shows improved asymptotic performance, a jumpstart, improved sample complexity and demonstrates safer deep reinforcement learning. We also discussed hyperparameter choices, namely the exploration parameter ϵ and the number of modes in the mixture density network K. In the following chapter (chapter 7), we summarise our findings as well as recommend directions for future work.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In model-free deep reinforcement learning algorithms that learn off-policy, the quality of the target task policy depends on the quality of the transition samples discovered by the exploration policy. In this work, we have presented a method for achieving multi-task transfer in continuous environments. This method successfully extracts useful common knowledge[1] from multiple, previous expert policies in the same environment, producing a summary of expert preferences in different regions of the state space. The presented method "mixture density network action priors" is an extension of the action priors framework [Rosman and Ramamoorthy, 2012] – a framework for achieving multi-task transfer in discrete environments, into environments with continuous states and continuous actions.

This model of prior knowledge is administered to agents learning new tasks in the environment in an ϵ-greedy exploration fashion, where instead of exploring according to a random exploration policy, agents explore by sampling and executing "trusted" expert actions from the learned mixture density network action priors distribution.

We provided an example of how the exploration step of off-policy deep reinforcement learning algorithms (i.e. the Deep Deterministic Policy Gradient) can be modified to incorporate our continuous action priors model in chapter 5, and presented experiments in chapter 6, which demonstrated that our continuous action priors maintain

[1] i.e. in the form of optimal trajectories

the standard benefits of the discrete action priors framework [Rosman and Ramamoorthy, 2012] such as safe exploration, a jump start in performance when learning, and increased performance.

This work has shown that combining deep reinforcement learning and multi-task transfer learning can result in safer and accelerated learning of tasks in continuous environments. Thus bringing us a step closer towards being able to apply model-free deep reinforcement learning to physical robot systems without relying on simulated environments.

7.2 Future Work

7.2.1 Portability of Learned Continuous Action Priors

As done by Rosman and Ramamoorthy [2012], we condition the learned action priors on the state space. This implies that the action priors are only useful as long as the state space remains fixed between tasks. This limits the portability of action priors across different environments. For example, if the cross-intersection environment (chapter 6, section 6.1) is rotated by 45^0, then action priors would no longer be useful and would most likely lead to negative transfer.

In future work, we propose to overcome this restriction by using perception-based action priors [Rosman and Ramamoorthy, 2012], where the action prior distribution is conditioned on observations instead of states. More specifically, these perception-based continuous action priors can be modelled through the use of a convolutional mixture neural network [Iso et al., 2017] – a standard convolutional neural network [LeCun et al., 2015] coupled with a mixture density network layer [Bishop, 1994].

7.2.2 Gaussian Process Action Priors

In this work, we dismiss Gaussian processes as a possible candidate for modelling continuous action priors due to their tendency to produce the conditional "uni-modal" mean over observations given input (see chapter 5, section 5.2). However, there are approaches that extend standard Gaussian processes to cater for multi-modal scenarios [Dutordoir et al., 2018, Lázaro-Gredilla et al., 2012]. These advances together

with sparse approximations, could result in a Gaussian Process distribution that sufficiently models continuous action priors for large datasets.

7.2.3 Uncertainty Measure in the Mixture Density Network

Since the mixture density network is a probabilistic model, it returns a measure of uncertainty[2] [Brando Guillaumes, 2017, Choi et al., 2018] regarding the action prediction for the given state. As such, this uncertainty measure can be used to further guide decision-making for successive agents learning in the environment, possibly resulting in a continuous action priors framework, that weights the probabilities of expert actions according to how much data (trajectories) was available around that state.

[2]In case of Gaussian distribution kernels, this uncertainty measure is the co-variance matrix.

Appendix A

Derivations

A.1 Likelihood-ratio Policy Gradient Derivation

Given the objective function:

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta(\tau)} [\, R(\tau)] \tag{A.1}$$

Where $R(\tau)$ is the value function evaluated at some trajectory $\tau = (s_1, a_1), ..., (s_T, a_T)$. The gradient of this objective function is then given by:

$$\nabla_\theta J(\theta) = \nabla_\theta \mathbb{E}_{\tau \sim \pi_\theta(\tau)} [\, R(\tau)] \tag{A.2}$$

Unrolling the expectation and taking derivatives of the objective:

$$\nabla_\theta J(\theta) = \int \nabla_\theta \pi_\theta(\tau) R(\tau) d\tau \tag{A.3}$$

Likelihood-ratio identity (true for any probability distribution):

$$\pi_\theta(\tau) \nabla_\theta log \pi_\theta(\tau) = \pi_\theta(\tau) \frac{\nabla_\theta \pi_\theta(\tau)}{\pi_\theta(\tau)} = \nabla_\theta \pi_\theta(\tau) \tag{A.4}$$

Substituting equation A.4 into equation A.3 we obtain:

$$\nabla_\theta J(\theta) = \int \pi_\theta(\tau) \nabla_\theta log \pi_\theta(\tau) R(\tau) d\tau \tag{A.5}$$

This leads to the expectation:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta(\tau)} [\, \nabla_\theta log \pi_\theta(\tau) R(\tau)] \tag{A.6}$$

A.2 Estimating the Policy Gradient from Data

Given the policy gradient[1]:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta(\tau)} \left[\nabla_\theta log \pi_\theta(\tau) R(\tau) \right] \tag{A.7}$$

In the above expectation, $\pi_\theta(\tau)$ is a trajectory given by:

$$\pi_\theta(s_1, a_1, ..., s_T, a_T) = \rho(s_1) \prod_{t=1}^{T} \pi_\theta(s_t|a_t) P(s_{t+1}|s_t, a_t) \tag{A.8}$$

If we take logs on both sides we obtain:

$$log \pi_\theta(s_1, a_1, ..., s_T, a_T) = log \rho(s_1) + \sum_{t=1}^{T} log \pi_\theta(s_t|a_t) + log P(s_{t+1}|s_t, a_t) \tag{A.9}$$

And taking derivatives w.r.t θ, the terms $log \rho(s_1)$ and $log P(s_{t+1}|s_t, a_t)$ vanish since they do not depend on θ. Therefore:

$$\nabla_\theta log \pi_\theta(\tau) = \sum_{t=1}^{T} \nabla_\theta log \pi_\theta(s_t|a_t) \tag{A.10}$$

Substituting this expression into Equation A.7 and letting $R(\tau) = \sum_{t=1}^{T} r(s_t, a_t)$, $\gamma = 1$ for finite episodic tasks, results in:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta(\tau)} \left[\left(\sum_{t=1}^{T} \nabla_\theta log \pi_\theta(s_t|a_t) \right) \left(\sum_{t=1}^{T} r(s_t, a_t) \right) \right] \tag{A.11}$$

This expression can be approximated using Monte Carlo i.e.:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=1}^{T} \nabla_\theta log \pi_\theta(s_{i,t}|a_{i,t}) \right) \left(\sum_{t=1}^{T} r(s_{i,t}, a_{i,t}) \right) \tag{A.12}$$

[1] Also known as the likelihood ratio policy gradient.

Appendix B

Implementation Details

The proposed framework (figure 5.7) consists of two main components: 1) the mixture density network for learning the exploration policy (i.e. the continuous action priors), 2) A deep reinforcement learning agent for learning the target task policy. In this work, we implement both these components in tensorflow[1] (cpu version 1.10) and python3. Further details regarding the specifics behind each component's implementation are discussed below.

B.1 Mixture Density Network Architecture

Individual layers in the mixture density network were implemented using the dense layers API (i.e. tensorflow.layers.dense), with Relu activation functions. The last layer $\boldsymbol{z}$ was split into three individual layers that were all connected to the previous layer. These three layers (the mixing co-efficients $\pi_k(x)$, the variances[2] $\sigma_k(x)^2$, and the mean vector $\mu_k(x)$) were allocated activation functions according to equations 5.10, 5.12 and 5.13 respectively. The Gaussian mixture model was implemented using tensorflow_probability (version 0.3) layers, particularly using the Mixture object, which takes as input, mixture components and a categorical distribution. These were obtained from tensorflow_probability distribution objects (the Normal distribution and a categorical distribution) from the separate parameter $\boldsymbol{z}$ layers, which had to be split further, depending on the number of modes K. Using tensorflow_probability

[1] `https://www.tensorflow.org/`, *Accessed: 25-05-2019*

[2] These were implemented as a diagonal co-variance matrix since the actions were multi-dimensional.

simplified the specification of the negative log likelihood loss function, which was given by: $-reduce\_sum(mixture\_model.log\_probability(current\_action))$.

The mixture density network architecture was set up to have 6 hidden layers (excluding the $\boldsymbol{z}$-layer), where each layer had 64 hidden units. The mixture density network was trained using the tensoflow's AdamOptimizer with a learning rate of 10^{-3}. This architecture was kept constant throughout the experiments.

B.2 DDPG Learning Agent

The Deep Deterministic Policy Gradient with Action Priors was implemented as a modification (line 5 in algorithm 7) of an existing DDPG agent implementation. More particularly we used the DDPG agent from the OpenAI baselines [Dhariwal et al., 2017] repository, which implements the DDPG algorithm according to the architectural setup reported in the original paper: [Lillicrap et al., 2015].

We implemented our framework in a virtualenv[3] which we deployed in a cluster with 60 interconnected i7 linux machines running the Ubuntu 16.04 operating system. These machines were setup under the TORQUE[4] resource manager and MPI[5]. To run the experiments (discussed in chapter 6), multiple instances of the same program were sent to multiple computers on the cluster, such that each computer runs the same experiment (task) using different random seeds. The results (csv files) would then be collected after training using secure copy (scp).

[3]A python tool for creating isolated environments, allowing users to install various python software packages without affecting the operating system.

[4]A resource management tool, commonly used for managing parallel computation in high performance computing systems.

[5]An interface for passing messages between multiple computers

Bibliography

A. G. Barto. Intrinsic motivation and reinforcement learning. In *Intrinsically motivated learning in natural and artificial systems*, pages 17–47. Springer, 2013.

M. Bellemare, S. Srinivasan, G. Ostrovski, T. Schaul, D. Saxton, and R. Munos. Unifying count-based exploration and intrinsic motivation. In *Advances in Neural Information Processing Systems*, pages 1471–1479, 2016.

C. M. Bishop. Mixture density networks. Technical report, Citeseer, 1994.

A. Brando Guillaumes. Mixture density networks for distribution and uncertainty estimation. Master's thesis, Universitat Politècnica de Catalunya, 2017.

G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

Y. Burda, H. Edwards, D. Pathak, A. Storkey, T. Darrell, and A. A. Efros. Large-scale study of curiosity-driven learning. *arXiv preprint arXiv:1808.04355*, 2018.

S. Choi, K. Lee, S. Lim, and S. Oh. Uncertainty-aware learning from demonstration using mixture density networks with sampling-free variance modeling. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6915–6922. IEEE, 2018.

P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, Y. Wu, and P. Zhokhov. Openai baselines. `https://github.com/openai/baselines`, 2017.

Y. Duan, X. Chen, R. Houthooft, J. Schulman, and P. Abbeel. Benchmarking deep reinforcement learning for continuous control. In *International Conference on Machine Learning*, pages 1329–1338, 2016.

R. Dubey, P. Agrawal, D. Pathak, T. L. Griffiths, and A. A. Efros. Investigating human priors for playing video games, 2018.

V. Dutordoir, H. Salimbeni, J. Hensman, and M. Deisenroth. Gaussian process conditional density estimation. In *Advances in Neural Information Processing Systems*, pages 2391–2401, 2018.

C. Finn, I. Goodfellow, and S. Levine. Unsupervised learning for physical interaction through video prediction. In *Advances in neural information processing systems*, pages 64–72, 2016.

M. Garnelo, K. Arulkumaran, and M. Shanahan. Towards deep symbolic reinforcement learning. *arXiv preprint arXiv:1609.05518*, 2016.

B. E. Hansen. Autoregressive conditional density estimation. *International Economic Review*, pages 705–730, 1994.

N. Heess, G. Wayne, D. Silver, T. Lillicrap, T. Erez, and Y. Tassa. Learning continuous control policies by stochastic value gradients. In *Advances in Neural Information Processing Systems*, pages 2944–2952, 2015.

H. Iso, S. Wakamiya, and E. Aramaki. Density estimation for geolocation via convolutional mixture density network. *arXiv preprint arXiv:1705.02750*, 2017.

D. P. Kingma and M. Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012. URL `http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf`.

B. M. Lake, T. D. Ullman, J. B. Tenenbaum, and S. J. Gershman. Building machines that learn and think like people. *Behavioral and Brain Sciences*, 40, 2017.

M. Lázaro-Gredilla, S. Van Vaerenbergh, and N. D. Lawrence. Overlapping mixtures of gaussian processes for the data association problem. *Pattern Recognition*, 45(4): 1386–1395, 2012.

Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *nature*, 521(7553):436, 2015.

S. Levine, P. Pastor, A. Krizhevsky, J. Ibarz, and D. Quillen. Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection. *The International Journal of Robotics Research*, 37(4-5):421–436, 2018.

T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

L.-J. Lin. Reinforcement learning for robots using neural networks. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF COMPUTER SCIENCE, 1993.

V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

G. Ostrovski, M. G. Bellemare, A. van den Oord, and R. Munos. Count-based exploration with neural density models. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 2721–2730. JMLR. org, 2017.

E. Parisotto, J. L. Ba, and R. Salakhutdinov. Actor-mimic: Deep multitask and transfer reinforcement learning. *arXiv preprint arXiv:1511.06342*, 2015.

M. G. Park, J. H. Jeon, and M. C. Lee. Obstacle avoidance for mobile robots using artificial potential field approach with simulated annealing. In *ISIE 2001. 2001 IEEE International Symposium on Industrial Electronics Proceedings (Cat. No. 01TH8570)*, volume 3, pages 1530–1535. IEEE, 2001.

D. Pathak, P. Agrawal, A. A. Efros, and T. Darrell. Curiosity-driven exploration by self-supervised prediction. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 16–17, 2017.

J. Peters and S. Schaal. Natural actor-critic. *Neurocomputing*, 71(7-9):1180–1190, 2008.

C. E. Rasmussen. Gaussian processes in machine learning. In *Advanced lectures on machine learning*, pages 63–71. Springer, 2004.

B. Rosman and S. Ramamoorthy. What good are actions? accelerating learning using learned action priors. In *Development and Learning and Epigenetic Robotics (ICDL), 2012 IEEE International Conference on*, pages 1–6. IEEE, 2012.

A. A. Rusu, S. G. Colmenarejo, C. Gulcehre, G. Desjardins, J. Kirkpatrick, R. Pascanu, V. Mnih, K. Kavukcuoglu, and R. Hadsell. Policy distillation. *arXiv preprint arXiv:1511.06295*, 2015.

A. A. Rusu, N. C. Rabinowitz, G. Desjardins, H. Soyer, J. Kirkpatrick, K. Kavukcuoglu, R. Pascanu, and R. Hadsell. Progressive neural networks. *arXiv preprint arXiv:1606.04671*, 2016.

J. Schulman, S. Levine, P. Abbeel, M. I. Jordan, and P. Moritz. Trust region policy optimization. In *Icml*, volume 37, pages 1889–1897, 2015.

J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller. Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 387–395, 2014.

R. S. Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44, 1988.

R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pages 1057–1063, 2000.

H. Tang, R. Houthooft, D. Foote, A. Stooke, O. X. Chen, Y. Duan, J. Schulman, F. DeTurck, and P. Abbeel. # exploration: A study of count-based exploration for deep reinforcement learning. In *Advances in neural information processing systems*, pages 2753–2762, 2017.

M. E. Taylor and P. Stone. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, 10(Jul):1633–1685, 2009.

M. E. Taylor, P. Stone, and Y. Liu. Transfer learning via inter-task mappings for temporal difference learning. *Journal of Machine Learning Research*, 8(Sep):2125–2167, 2007.

Y. Teh, V. Bapst, W. M. Czarnecki, J. Quan, J. Kirkpatrick, R. Hadsell, N. Heess, and R. Pascanu. Distral: Robust multitask reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 4496–4506, 2017.

C. Tessler, S. Givony, T. Zahavy, D. J. Mankowitz, and S. Mannor. A deep hierarchical approach to lifelong learning in minecraft. *arXiv preprint arXiv:1604.07255*, 2016.

S. B. Thrun and K. Möller. *On planning and exploration in non-discrete environments*. GMD Sankt Augustin, Germany, 1991.

S. B. Thrun and K. Möller. Active exploration in dynamic environments. In *Advances in neural information processing systems*, pages 531–538, 1992.

E. Todorov, T. Erez, and Y. Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012.

J. Tsitsiklis and B. Van Roy. An analysis of temporal-difference learning with function approximationtechnical. *Report LIDS-P-2322). Laboratory for Information and Decision Systems, Massachusetts Institute of Technology*, 1996.

P. A. Tsividis, T. Pouncy, J. L. Xu, J. B. Tenenbaum, and S. J. Gershman. Human learning in atari. 2017.

J. Vossen, B. Feron, and A. Monti. Probabilistic forecasting of household electrical load using artificial neural networks. In *2018 IEEE International Conference on Probabilistic Methods Applied to Power Systems (PMAPS)*, pages 1–6. IEEE, 2018.

C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

P. M. Williams. Using neural networks to model conditional multivariate densities. *Neural Computation*, 8(4):843–854, 1996.

R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

Y. Wu, E. Mansimov, R. B. Grosse, S. Liao, and J. Ba. Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation. In *Advances in neural information processing systems*, pages 5279–5288, 2017.

J. Yosinski, J. Clune, Y. Bengio, and H. Lipson. How transferable are features in deep neural networks? *CoRR*, abs/1411.1792, 2014. URL `http://arxiv.org/abs/1411.1792`.