VILNIUS UNIVERSITY
FACULTY OF MATHEMATICS AND INFORMATICS
INSTITUTE OF COMPUTER SCIENCE
DEPARTMENT OF COMPUTATIONAL AND DATA MODELING

Master's thesis

Pareto Conditioned Networks in Continuous Action Spaces

Done by:
Vaidas Slavinskas signature

Supervisor:
Prof., Dr. Aistis Raudys

Vilnius
2024

Contents

Abstract

Reinforcement learning is one of the main paradigms of machine learning methodologies, a technique for sequential decision making in a fully or partially observed environment. The main objective of this approach is to optimise accumulated rewards over time. Generally, these rewards are scalar and have a single objective, but many real-world tasks involve multiple conflicting objectives that cannot be adequately described by a single scalar reward. This has led to a growing interest in the field of multi-objective reinforcement learning (MORL).

A recently introduced method that can deliver the state of the art results in various multi-objective reinforcement learning task is the Pareto Conditioned Network (PCN). Using a single neural network to learn all non-dominated policies, PCN has several advantages over other MORL algorithms, including fewer hyperparameters and better scalability with large objective counts for multi-objective problems. However, the original PCN methods are only applicable to discrete action space problems, which is a major limitation since many real-world scenarios inherently require actions that are continuous rather than choosing actions from discrete, limited options. In order to overcome this limitation, this Master's thesis presents a modified PCN that can operate efficiently in continuous action spaces. The main modification was related to the reconfiguration of the PCN model output layer as well as introducing a different exploration strategy. Instead of producing multiple discrete outputs, the network was adapted to produce a single continuous prediction representing a policy action. This change transformed the training task from a classification task to a regression task and required a change in the loss function which is used during training. In addition, the exploration strategy, which is an essential component of reinforcement learning, was significantly modified. In discrete action spaces, PCN facilitated exploration by sampling actions from a categorical distribution, where the probability of each action was proportional to its confidence score calculated using the softmax function in the last layer of the network. However, this approach was not compatible with continuous action spaces, where the output value directly reflects the action itself. Therefore, a new strategy was developed to encourage exploration while ensuring the network's ability to learn and adapt to the continuous nature of actions.

To evaluate the performance of the modified PCN method, different multi-object environments were used. These environments were selected to reflect the different complexity and characteristics that allow a reliable assessment of the performance of the method in different scenarios. The modified PCN method was compared to two other MORL algorithms.

In summary, this Master's thesis not only presents the necessary theoretical and practical adaptations to extend the PCN in continuous action spaces, but also provides a comprehensive evaluation of its effectiveness. The insights gained from this research represent a significant contribution to the field of multi-objective reinforcement learning, opening up new possibilities for applying these advanced techniques to a wider range of real-world problems.

Santrauka

Skatinamasis mokymasis yra viena iš pagrindinių mašininio mokymosi metodikų paradigmų, taikoma žingsninių sprendimų priėmimo strategija visiškai arba iš dalies matomose aplinkose. Pagrindinis šio metodo tikslas - optimizuoti sukauptą atlygį per tam tikrą žingsnių skaičių. Paprastai šie atlygiai yra skaliariniai ir turi vieną reikšmę, tačiau daugelis realaus pasaulio užduočių apima kelis, dažnai prieštaringus, tikslus, kurių negalima tinkamai apibūdinti vienu skaliariniu atlygiu. Dėl to paskutiniu metu vis labiau dėmesio skiriama daugiakriterinio pastiprinimo mokymosi (angl. multi-objective reinforcement learning, MORL) sričiai.

Visai nesenai buvo pristatytas naujas metodas kuris gauna gerus rezultatus sprendžiant daugiakriterinius pastiprinto mokymosi užduotis, pavadintas Pareto sąlyginiu tinklu (angl. Pareto Conditioned Network, PCN). Metodas naudoja vieną neuroninį tinklą, išmokti visas nedominuojančias strategijas. Palyginus su kitais MORL algoritmais PCN pranašumas yra tai, kad metodas turi mažesnį hiperparametrų skaičių ir veikia geriau su užduotimis, kurios turi daug tikslų. Tačiau originalus PCN metodas buvo pritaikytas spręsti užduotis, kuriose veiksmai yra diskretūs, o tai yra didelis apribojimas, nes didelė dalis užduočių reikalauja sprendimų, kurie yra besitęsiantys, o ne diskretūs. Siekiant išspręsti šį apribojimą, šiame magistro darbe pateikiamas modifikuotas PCN metodas, kuris gali efektyviai veikti nenutrūkstamose veiksmų erdvėse. Pagrindinis PCN metodo pakeitimas atliktas darbe buvo neuroninio tinklo pakeitimas. Vietoj to, kad tinklas grąžintų kelis diskrečius rezultatus, tinklas buvo pritaikytas taip, kad būtų gaunama viena tęstinė prognozė, atspindinti veiksmą. Tinklo mokymas iš klasifikavimo užduoties tapo regresijos užduotimi, todėl buvo pakeista mokymo metu naudojama nuostolių funkcija. Taipogi pakeista metodo žvalgymo (ang. exploration) strategija, kuri yra esminė skatinamojo mokymosi dalis. Orginaliame PCN metode žvalgymas buvo atliekamas pasirenkant veiksmus iš kategorinio pasiskirstymo, kai kiekvieno veiksmo tikimybė yra proporcinga jo patikimumo balui, apskaičiuotam naudojant *softmax* funkciją paskutiniame tinklo sluoksnyje. Tačiau šis metodas buvo nesuderinamas su tolydžiomis veiksmų erdvėmis, kuriose išėjimo reikšmė tiesiogiai atspindi patį veiksmą. Todėl buvo sukurta nauja strategija, skatinanti tyrinėjimą ir kartu užtikrinanti tinklo gebėjimą mokytis ir prisitaikyti prie nenutrūkstamo veiksmų pobūdžio.

Tam kad modifikuoto PCN metodo efektyvumas būtų įvertinas, buvo panaudotos skirtingos daugiakriterinės mokymosi aplinkos. Šios aplinkos buvo parinktos taip, kad būtų skirtingo sudėtingumo ir charakteristikų, tokiu būdu leidžiančios įvertinti metodo veikimą įvairiuose scenarijuose. Modifikuotas PCN metodas buvo palygintas su dviem kitais MORL algoritmais.

Magistriniame darbe pristatomos būtinos teorinės ir praktinės adaptacijos, skirtos išplėsti PCN metodo veikimą nepertraukiamo veiksmo erdvėse, taipogi pateikiamas išsamus jo veiksmingumo įvertinimas. Šio tyrimo metu gautos įžvalgos yra reikšmingas indėlis į daugiakriterinio skatinamojo mokymosi sritį, atveriantis naujas galimybes taikyti šiuos pažangius metodus sprendžiant įvairesnius realaus pasaulio uždavinius.

Introduction

Reinforcement learning (RL) is one of the main machine learning branches, tracing back its roots to the early days of computer science and other fields [Kaelbling et al., 1996]. In general the intent of RL techniques is to have an algorithm which could learn through trial and error to achieve a goal, without needing to specify steps of how that goal should be achieved. More precisely RL can be defined as a method for a sequential decision making in a fully or partially observed environment while maximising reward [Liu et al., 2015]. Unlike in many other machine learning techniques, training RL agents do not require large data sets of labeled data. RL algorithms usually only require environment which provides a set of actions an agent can take, a reward, and an observation when the action is taken.

The field of reinforcement learning has been extensively studied in previous research, modern RL algorithms have attracted a lot of interest after their successful application in practical scenarios. In particular, deep RL agents have demonstrated their proficiency in achieving and outperforming human-level performance in controlling Atari games [Mnih et al., 2013, Mnih et al., 2015] using only high-dimensional video signals as input. Furthermore the same agents were trained on different games showing generalization. This has been achieved by applying rapidly advancing deep learning techniques with RL (such algorithms are often called Deep Reinforcement Learning (DRL)). DRL have even been used in the papers achieving super-human performance in well known games GO and Chess [Silver et al., 2017], without having any labeled data used for training, except for the rules of the game.

Despite the above, most of the research has focused on single-objective reinforcement learning. In this system, the agent tries to optimise a reward, expressed as a scalar value, usually representing a single goal that the agent aims to achieve. However, in many real-world tasks, a balance has to be struck between multiple, often conflicting, goals that are not adequately represented by a single scalar reward. Therefore, there is a growing interest in Multi-Objective Reinforcement Learning (MORL) to address this complexity. Despite the focus on single-objective RL, it has been argued that multi-objective RL is still relatively underrepresented in the wider spectrum of RL research [Roijers et al., 2015].

A promising multi-objective reinforcement learning method known as Pareto Conditioned Networks (PCN) [Reymond et al., 2022] has been introduced in recent years and is considered state-of-the-art in performance. Utilising a single neural network to identify all non-dominated policies, the Pareto conditioned networks model is trained to replicate the transitions for a given return and desired number of steps. This approach effectively transforms the optimisation problem into a classification problem. In general, PCN is superior to other MORL algorithms because it has fewer hyperparameters and better scalability for multi-objective problems. However the original PCN method is applicable only to tasks with discrete action spaces, which means that the method is unable to learn in environments where actions are continuous. This is a significant drawback as many real world problems inherently demand continuous choices rather than discrete, limited options. This Master's thesis solves this issue, by adapting Pareto conditioned network to work in continuous action spaces.

The goal of this Master's thesis - to modify Pareto conditioned network method that can be efficiently adapted to operate in continuous action spaces to improve decision-making processes and performance in complex environments.

The main tasks:

1. Provide a comprehensive literature review on multi-objective reinforcement learning and

analyse the Pareto conditioned networks method.

2. Perform modifications to Pareto-conditioned networks method to allow training and action execution in continuous action spaces.
3. Evaluate the changes and validate the performance of the updated Pareto conditioned networks.
4. Compare the updated Pareto conditioned networks method with other multi-objective reinforcement learning algorithms.

1 Background and Related Work

Reinforcement learning has its early roots in studying animal learning psychology in the beginning of 20th century. Since then a solid theoretical background for RL and practical methods has been developed [Sutton and Barto, 2018]. At the current time RL has established itself as one of the main machine learning branches. It is worth mentioning that reinforcement learning is distinctly different from other two major (supervised and unsupervised) machine learning paradigms. Where supervised learning is learning from a training set of labeled examples provided by a knowledgeable external supervisor, RL algorithms learn from experience in the environment, meaning that the only data needed for training RL algorithm is the environment. In unsupervised learning there is no knowledgeable external supervisor, but the goal of it is to find structure in unstructured data which solves a different problem than maximising a reward signal in RL [Sutton and Barto, 2018].

RL solves a problem of mapping some state (or observation) to actions in a way that maximizes a reward [Sutton and Barto, 2018]. RL system can be described as an agent (model) which observes an environment (state) and a set of actions which the agent can take given a state and a reward function which indicates how good a particular action or state is (see Fig. 1). During the learning process the goal of the agent is to learn which actions it should take in order to achieve maximum total long term reward. The strategy for choosing an action given a state is called a policy.

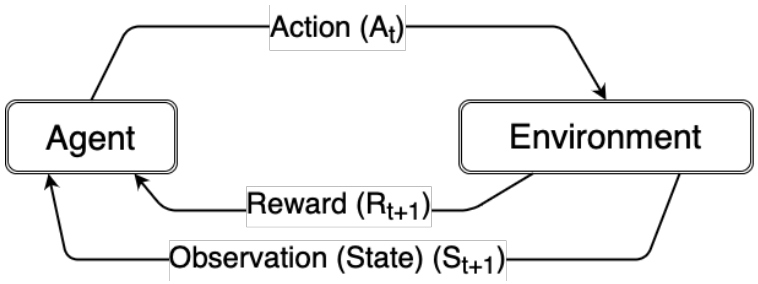


Figure 1. Basic reinforcement learning algorithm example.

1.1 Markov Decision Process

A common way to formalize reinforcement learning is to use Markov Decision Processes (MDPs). MDPs are a mathematically idealized form of the reinforcement learning problem for which precise theoretical statements can be made [Sutton and Barto, 2018]. An MDP is formally defined as a tuple (S, A, P, R), ([Sutton and Barto, 2018]) , where:

- S is the state space, a set of all possible states.
- A is the action space, a set of all possible actions.
- P is the state transition probability function, $P(s'|s, a)$, representing the probability of transitioning from state s to state s' after taking action a.
- R is the reward function, $R(s, a, s')$, indicating the immediate reward received after transitioning from state s to state s' due to action a.

A function that assigns probabilities to each available action for every state is formally known as a policy π. In other words, a policy defines how an agent acts by specifying the action it should take

in each state. It can be either deterministic or stochastic function [Bertsekas, 2012]. The mapping from states to actions of deterministic policy is fixed, meaning that for a given state policy will always choose the same action. Formally, this can be represented as $\pi(s) = a$, where:

- $\pi(s)$ is a deterministic policy given a state s.
- a is an action.

In case of stochastic policy, actions are selected based on probability - policy maps states to a probability distribution over the set of possible actions. It can be represented as $\pi(a|s) = P(A_t = a|S_t = s)$, where:

- $\pi(a|s)$ is a stochastic policy, which maps a given state s to a probability distribution over the actions a.
- $P(A_t = a|S_t = s)$ is a probability of taking an action a at timestep t given state s.

The objective is to find a policy π, that maximizes the expected sum of discounted rewards over time. The expectation is taken over the initial state distribution p_0 and the stochastic transitions.

In MDP the future states depend only on the current state and action, not on the sequence of states and actions that preceded it. This is called Markov property and is important in RL learning, as this means that agents do not need to store all the previous states in order to have an optimal policy, remaining memory efficient. While the MDP framework may not adequately capture all aspects of decision-learning problems, it has demonstrated broad utility and applicability [Sutton and Barto, 2018]. In practice, processes often might not perfectly satisfy the Markov property, but in most cases, such tasks can still be addressed by reinforcement learning algorithms. An example of this - the agent is trained in an environment which has two actions: clean stains and pick an item. However one of the actions is allowed only after the other is completed - picking an item is permitted only if all stains were previously cleaned. In this scenario Markov property is violated, but RL agent can still effectively learn a policy to reach the goal [Gaon and Brafman, 2019].

There are many extensions to classical MDPs. For example some RL problems are fully-observable, meaning that the agent gets to observe the complete state of the environment, while others are only partially observable - the agent only gets to observe part of the state of the environment [Monahan, 1982]. One such extension is called Multi-Objective Markov Decision Process which will be discussed in section 1.3

1.1.1 Reinforcement Learning: Solving Markov Decision Process

An important consideration in RL algorithms is balancing exploration and exploitation. In RL exploration means choosing trajectories that were less explored while exploitation means choosing trajectories which are known to bring the highest reward. Choosing higher exploration usually means encountering lower immediate returns, however more of the state action space is explored which may lead to a higher rewards in the long term. Conversely choosing more exploitation risks converging on local minimum, and exploring less of the action state space allows for choosing more optimal actions if the algorithm already has good understating of the environment. Common strategies for balancing exploration and exploitation in RL include epsilon-greedy where the agent explores with a probability of ε and exploits with a probability of $1 - \varepsilon$, decaying epsilon-greedy, where ε is decreased during the learning process and various different strategies for decaying the ε, softmax exploration where the action selection is based on the softmax function and others.

The goal of solving an MDP is to find an optimal policy π^*, that maximizes the expected cumulative reward over time horizon. It is described in formula 1.1, where:

- G_t is the cumulative reward at timestep t.
- R_t is the reward at timestep t.
- γ is the discount factor. $0 \leq \gamma < 1$. The closer discount factor is to 1 the more future rewards are taken into account. A discount factor of 0 means the objective becomes to maximise only one step into the future. Such policy has no long term planning.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{1.1}$$

Most of RL algorithms in the process of solving MDPs estimate value function which estimates how good is it to be in a certain state. It is denoted as $v_\pi(s)$. Formally it is expressed by formula 1.2, where:

- $v(s)$ is the value of state s. It represents expected discounted return if agent follows policy π when starting in state s.
- π is the policy that will be followed when starting from state s.
- E is expected value of a random variable.
- γ refers to discount factor that determines how far future rewards are taken into account in the return.
- s_t is a state at a timestep t.
- R is the immediate reward.

$$v(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R_t(s_t, s_{t+1}) \middle| S_t = s\right] \tag{1.2}$$

In other words given a particular policy and a state, value function returns an expected reward if the policy is followed starting with the given state. A closely related function is called state-action value function which returns expected reward given an action a is taken, expressed by formula 1.3, where:

- $q(s, a)$ is the state-action value for a given state-action pair (s, a).
- E is the expectation, which in this case is the average of the sum of discounted rewards obtained over many possible futures starting from state s, taking action a, and then following policy π.
- $\sum_{t=0}^{\infty} \gamma^t R_t(s_t, s_{t+1})$ is the sum of the rewards R_t at each time step t, discounted by γ^t.
- $R_t(s_t, s_{t+1})$ is the reward received after transitioning from state s_t to s_{t+1} due to action a at time t.

- $S_t = s, A_t = a$ denotes that the expectation is conditioned on the agent being in state s at time t and taking action a at that time.

$$q(s,a) = E\left[\sum_{t=0}^{\infty} \gamma^t R_t(s_t, s_{t+1}) \middle| S_t = s, A_t = a\right] \tag{1.3}$$

It is the name of the state-action value function that gives the name to a well known RL algorithm called Q-learning.

Basic idea of of Q-learning is to store expected state-action values in a table and choose an action that has the largest value for a given state [Sutton and Barto, 2018]. Then the expected state-action value for the given state and action is updated based on the new reward, learning rate and expected value of the following state. Initially stored q values are random. More formally Q value update is defined by formula 1.4, where

- α represents the learning rate (between 0 and 1).
- S_t, A_t - state and action at a given step t.
- R_{t+1} is the reward received after taking action A_t in state S_t and moving to the new state S_{t+1}.
- $\max Q(S_{t+1}, a)$ is the maximum estimated Q-value for the next state S_{t+1} over all possible actions a.

$$Q(S_t, A_t) \text{¸} Q(S_t, A_t) + \alpha(R_t + 1 + \gamma * max\ Q(S_{t+1}, a) - Q(S_t, A_t)) \tag{1.4}$$

In deep Q-learning (DQN) Q-value table is replaced with a neural network function approximators [Mnih et al., 2013]. Deep Q-Network (DQN) algorithms have been used effectively to match and exceed human performance levels in Atari games and other tasks. Nowadays, there are many variants of the Deep Q-Network algorithms is available for utilization.

1.2 Multi-Objective Reinforcement Learning

A large amount of real world problems involves solving a task which has multiple objectives. When these objectives do not conflict directly with each other, task can be solved using conventional RL methods, as reward function can be a sum of different goals. For example given a task where the goal is to navigate to some location using a car, while at the same time not exceeding the speed limit, RL algorithm reward function can be modeled as a sum of two rewards - one for successfully reaching intended destination, another, negative one, for exceeding the speed limit. However issue arises when the objectives are competing. In this example it could be having a goal of reaching the destination as fast as possible while at the same time minimizing the fuel amount used. If the relative importance of those goals is known before training, one can use the sum of the reward with each reward multiplied by it's weight, indicating importance. However there are multiple issues with this approach [Liu et al., 2015]:

- Assigning preferences is a manual process.

- Without knowing what tradeoffs different preferences might result in, decision maker cannot make a well informed decision.
- Preferences over goals are often unknown.
- Human decision makers and users may have preferences that it cannot fully accommodate (in case of non-linear scalarisation function).
- Preferences might change after training, during run time.

Therefore multi-objective approach is used (see Fig. 2). The aim of multi-objective reinforcement learning is to find optimal policies when balancing multiple conflicting objectives.

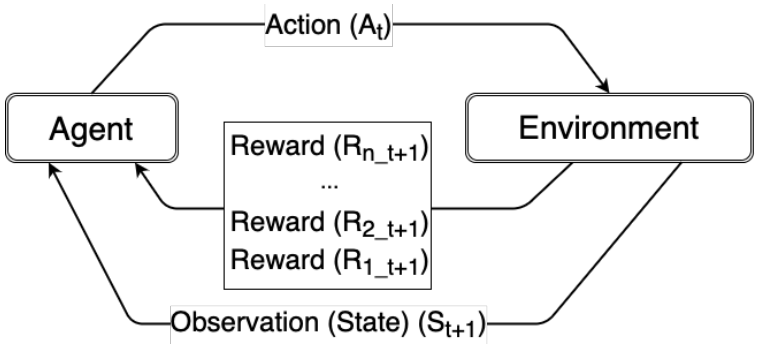


Figure 2. Basic multi-objective reinforcement learning algorithm example.

1.3 Multi-Objective Markov Decision Process

To formalise multi-objective reinforcement learning problems an extended version of MDP is used called Multi-Objective Markov Decision Process (MOMDP). It is typically defined as a tuple $(S, A, P, R, \gamma, p_0)$ [Hayes et al., 2021], where:

- S, A, P, γ, p_0 have the same definition as in standard MDP 1.1.
- R^d is vector valued reward function for each $d \geq 2$ objectives.

The main difference compared to standard MDP is that reward function returns a vector value instead of a scalar value. The reward vector length is equal to the number of objectives being optimised as each value represents a separate goal. Like in standard MDP state and action spaces can be discrete and finite but it can also be infinite. In practice this happens when values describing state or actions are continuous instead of discrete. Also for more complex problems state space can be too large to enumerate for example when state is in format of pixel values (images). Like with standard MDP there are also numerous extensions to MOMDP like partially observed MOMDP [Nian et al., 2020] or multi-agent versions of MOMDP [Rădulescu et al., 2019]. In this paper only problems applicable to standard version of MOMDP are considered.

In contrast to standard MDPs, MOMDPs have multiple objectives that need to be optimised which means that generally there is no single policy that optimises all objectives simultaneously. Therefore the goal of solving MOMDPs is finding the set of optimal policies.

1.4 Utility Functions

When solving single-objective problems, the two policies can be easily compared by comparing the expected value as explained in formula 1.2, and this can lead to the optimal policy, but when there are multiple objectives, then the reward is a vector, and the formula is no longer valid.

However in case of multi-objective problems the value is a vector, making direct comparison non viable without knowing preferences over the objectives. In order to convert reward to a scalar value a utility function is used. It can be represented as a mapping $u : \mathbb{R}^n \rightarrow \mathbb{R}$ where utility function u maps n-dimensional space of rewards corresponding to n objectives to real number $\mathbb{R}$ for the utility value. In essence, the utility function embodies the decision-maker's preferences.

In MORL, utility functions u are generally assumed to be monotonically increasing across all objectives. This assumption means that any increase in an objective's value, without a decrease in others, should result in a higher scalarised value. Monotonicity of utility functions is an improtant assumtion in MORL as it means that higher value of any objective is always desired. This foundational assumption simplifies the optimization landscape by allowing for scalarization techniques that transform multi-objective problems into single-objective ones, without the risk of encountering preference reversals due to non-monotonic behavior.

1.5 Solution Sets

Possible solution sets are often narrowed down even more in order to reduce the solution space and simplify the problem. When reviewing the literature on Multi-Objective Reinforcement Learning (MORL), it is common to find references to solution sets that are commonly used in the field. These sets provide a framework for understanding and applying different strategies and solutions in the context of MORL.

1.5.1 Undominated Set and Coverage Set

The undominated set, denoted as $U(\Pi)$ contains all the policies (Π) and their corresponding value vectors which are optimal for at least one utility function. This means no other policy has a higher utility for that function. The set $U(\Pi)$ may possess redundant policies — those that are optimal for specific utility functions but not exclusively. Formally, the undominated set is defined by formula 1.5 [Hayes et al., 2021], where:

- $U(\Pi)$ is the undominated set of policies.
- π a single policy in the policy space.
- $u(V_\pi)$ the utility of the expected return when following policy π.
- V_π the expected return (value) when following policy π.
- $\forall \pi' \in \Pi$: For every policy π' in the policy space.
- $u(V_\pi) \geq u(V_{\pi'})$: The utility of the expected return of policy π is greater than or equal to that of any other policy π'.

$$U(\Pi) = \{\pi \in \Pi | \exists u, \forall \pi' \in \Pi : u(V_\pi) \geq u(V_{\pi'})\} \tag{1.5}$$

A coverage set, $CS(\Pi)$, is a subset of $U(\Pi)$ that, for every utility function u, contains at least one policy with the maximal scalarised value with the goal to minimize the size of $CS(\Pi)$.

1.5.2 Pareto Front

When the utility function u is any monotonically increasing function, A policy is said to Pareto-dominate another if its expected return ($V - value$) is higher or equal across all objectives. The Pareto Front consists of non-dominated policies, where no other policy in the set has a value vector that is equal to or better in all objectives (see Fig. 3). Pareto front can contain infinite number of policies especially when policies are stochastic. The Pareto Front is formally defined using the Pareto dominance relation [Roijers et al., 2013], formula 1.6, where:

- PF is the Pareto front.
- π represents a specific policy.
- Π is the set of all possible policies.
- V_π is the value function associated with policy π.
- $\succ_P$ represents the Pareto dominance relationship. In multi-objective optimization, a policy π is said to Pareto dominate another policy π' if it is better or equal in all objectives and strictly better in at least one objective. The notation $V_\pi \succ_P V_{\pi'}$ means the value (or utility) obtained by policy π is Pareto superior to that obtained by policy π'.
- $\nexists \pi' \in \Pi$ states that there does not exist any other policy π' in the set of all policies Π such that $V_{\pi'}$ is Pareto superior to V_π.

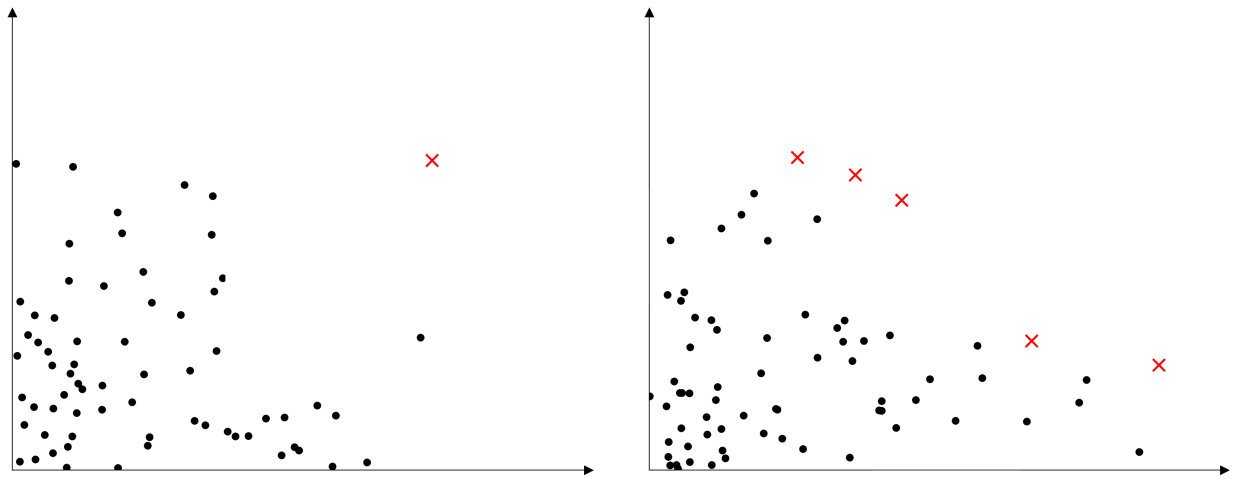
$$PF = \{\pi \in \Pi | \ \nexists \pi' \in \Pi : V_{\pi'} \succ_P V_\pi\} \tag{1.6}$$

Figure 3 demonstrates Pareto front in two-objective environment in two distinct scenarios. Each point represents cumulative return ($V_i - value$) achieved by executing a specific policy π_i. Red 'x' denote Pareto optimal solutions, collectively forming the Pareto front. While it is possible for a single policy to represent the entire Pareto front (as in Graph A), more commonly multiple optimal policies exist, each representing unique trade-offs (as in Graph B).

1.5.3 Convex Hull and Convex Coverage Set

The convex hull (CH) represents a subset of all possible policies, where for a linear utility function u, there is a corresponding weight vector w that maximizes the scalarized value of these policies. Or in other words, the convex hull consists of policies that, for some weight vector w from the real-valued space $\mathbb{R}^d$, yield the maximum combined objective value [Roijers et al., 2013]. It can be expressed as formula 1.7, where:

- x a single solution from the set Π.
- Π the space of all possible solutions.
- w a weight vector that reflects the importance of different objectives.



A B

Figure 3. Pareto front in two-objective environment in two distinct scenarios.

- v^{π} the value vector associated with an element π, where each component of the vector represents the performance or utility with respect to a particular objective.

$$CH = \{x \in \Pi \mid \exists w, \forall \pi' \in \Pi : w^T v^{\pi} \geq w^T v^{t'}\} \tag{1.7}$$

A linear utility function is expressed as the inner product of a weight vector w and a value vector V_{π}, where w consists of positive real numbers that sum to 1.

The convex coverage set (CCS) is a subset of the convex hull $CH(\Pi)$ that contains, for every weight vector w, a policy whose linearly scalarised value is maximal. This set is crucial not only for linear utility functions but also for constructing coverage sets for all monotonically increasing utility functions, including non-linear ones.

1.5.4 Practical Implications

As described before (section 1.4) the optimal utility function is often unknown and might change during run time therefore using scalarised expected cumulative reward is undesirable when comparing algorithms. More so the utility functions can have different properties and one algorithm might perform well with certain kind of utility functions and poorly with different kind. For example, if one algorithm performs well under a linear utility function but poorly under a non-linear one, it suggests that the algorithm may be good for problems where objectives are independent but not for those where they interact in complex ways. Moreover, in order to find non-convex Pareto fronts, non-linear utility functions must be used as linear utility functions are unable to handle such solution spaces. The choice of utility function significantly impacts the behaviour of MORL algorithms, making it an important consideration in both the development and evaluation phases. In addition to that, the way utility function is applied affects the optimal policies. The utility function can be applied at the end of each episode. Such approach is called Scalarized Expected Returns (SER) criterion and is formalized by formula 1.8, where:

- V_u^{π} represents the expected utility of following policy π.

- u is the utility function.
- $\mathbb{E}$ is the expectation operator, which calculates the expected value or mean of a random variable.
- $\sum_{i=0}^{\infty} \gamma^i r_i$ is the infinite sum of discounted rewards, where r_i is the reward received at time step i, and γ is the discount factor.
- π is the policy π.
- s_0 is the initial state from which the agent starts.

$$V_u^{\pi} = u\left(\mathbb{E}\left[\sum_{i=0}^{\infty} \gamma^i r_i | \pi, s_0\right]\right) \tag{1.8}$$

However it is also possible to apply utility function at each timestep to scalarize immediate rewards and then compute the expected utility. This method is called Expected Scalarized Returns (ESR) and is described by formula 1.9 where values are equivalent to those in formula 1.8.

$$V_u^{\pi} = \mathbb{E}\left[u\left(\sum_{i=0}^{\infty} \gamma^i r_i(s_0, \pi)\right)\right] \tag{1.9}$$

For example in a portfolio management problem where the objectives might be to maximize returns while minimizing risk, it might be preferable to evaluate the overall performance of a trading strategy over an extended period. SER would be more suitable in this situation. While in a robotic navigation task where the robot has to reach a destination while avoiding obstacles and minimizing energy consumption, the importance of each objective might change dynamically (e.g., avoiding an obstacle may temporarily become more important than minimizing energy). ESR would be more suitable in this case.

1.6 Current Approaches to Multi-Objective Reinforcement Learning

Current MORL methods are divided into two categories [Liu et al., 2015]:

- Single policy - aims to learn a single optimal policy, given some objective preferences (utility function).
- Multi policy - aims to learn a set of optimal policies for different objective preferences (utility functions) known as a coverage set.

In single policy algorithms the multiple objectives are converted into a single objective by defining preferences among multiple objectives (section 1.4), usually by user or by problem domain [Liu et al., 2015]. The agent then optimizes this single objective using traditional RL algorithms like Q-learning. The challenge here lies in accurately capturing the user's or domain's preferences, as an incorrect specification can lead to sub-optimal policies. Also changes in preferences requires retraining the model. Meanwhile multiple policy algorithms find a set of policies that approximate Pareto optimal solution.

One of proposed algorithms for MORL is Modular Multi-Objective Deep Reinforcement Learning with Decision Values (MODQN-DV) [Tajmajer, 2018]. For each objective in the task it uses a

separate DQN , which are extended with parameter called decision value. Decision value indicates importance of the objective in a given state. The output of each DQN including the decision values are then combined with user preferences (which can be modified during runtime) to get optimal action for each state. It is demonstrated that when different objective priorities are used compared to the training phase, MODQN-DV shows an improved performance in comparison to DQN which uses weighted sum utility function for scalarisation of rewards. The author notes that although the results are promising, more tests should be performed using different benchmarks as the algorithm was tested only with a single problem in the original paper. Author notes that he is unaware of any MORL benchmarks that would be comparable to Atari games or provided by OpenAi.

1.7 Reinforcement Learning Agents Performance Evaluation

Evaluation and comparison of single objective RL agents is relatively straight forward. Because the agent's goal is to maximise a scalar reward, policies found by two different algorithms can be directly compared by comparing their expected sum of rewards [Zintgraf et al., 2015]. Empirically algorithms can be compared directly by comparing the reward achieved by the algorithms solving the same task. However this only compares the end result of a trained agent. In many cases it might be beneficial to use an algorithm that achieves slightly sub-optimal policy but does so using significantly less training steps (samples). For example it might be important for agent to learn using least amount of training steps possible if the agent has to learn while being deployed. In these cases it is useful to compare different algorithms cumulative reward collected during training.

Because reward in MORL is a vector not a scalar, comparison of MORL agents is more difficult. For example what would be a better policy for an autonomous car - the one which consumed less fuel but travelled faster or the one which did the opposite? In case the user preference (utility function) is known, solutions can be directly compared. However with the utility function not known and in case of multi-policy networks, different comparison methods are needed.

1.8 Multi-Objective Reinforcement Learning Evaluation Metrics

Evaluating the solution sets produced by different algorithms is a complex challenge. For example while the results from one algorithm may be superior in certain objectives, it may be inferior in others. This raises a question of how to compare algorithms producing such results. Metrics for comparison can be categorised as axiomatic and utility based [Hayes et al., 2021], where axiomatic ones compare policies to the true Pareto front (or Convex hull in case of linear utility functions) while utility based ones compare the utility outcomes.

The hypervolume metric [Zitzler and Thiele, 1999] measures how much of the value-space (hypervolume) a set of solutions covers. It is retrieved by calculating how much volume the non dominated solutions take up when compared to a reference point. Hypervolume correlates with spread of the undominated solutions - a larger hypervolume suggests that the solutions are spread out over a larger region of the objective space, covering more possible trade-offs between objectives which means a set of solutions with a high hypervolume is likely to contain solutions that are diverse in terms of their objective values. It can be used to compare the solution set of competing algorithms or to compare the solution set to the pareto front if it's known. There have been MORL algorithms proposed which directly utilise hypervolume for action selection [Van Moffaert et al., 2013]. Figure 4 illustrates hypervolume. Non-dominated solutions are represented by blue circles, while dominated policies are denoted by black crosses. The reference point

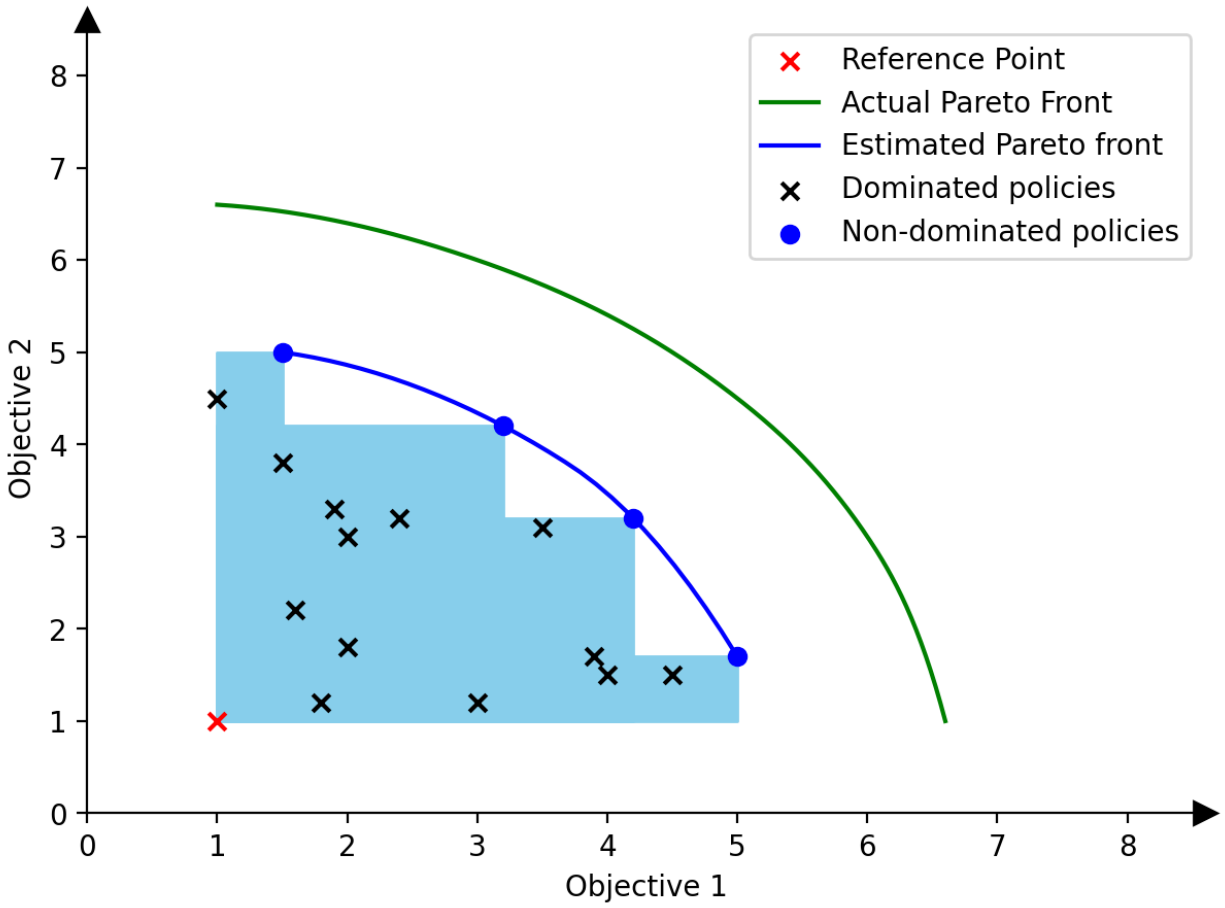


Figure 4. Visualization of the Pareto front with associated hypervolume in a two objective scenario.

is indicated by a red x. The shaded area in sky blue represents the hypervolume, which measures how much of the objective space is covered by the Pareto-optimal solution set with respect to the reference point. The green line depicts the actual Pareto front. The blue line represents an estimated Pareto front, constructed from the non-dominated solutions. Better algorithms should have a larger hypervolume. However, hypervolume metric can be hard to interpret because it does not directly relate to the real world benefits of the solutions. For example, adding one extreme solution can greatly increase the hypervolume without being useful, while adding a valuable solution might not change the hypervolume much. Also, in real-world problems, the best possible set of solutions is often unknown, which makes it hard to use the hypervolume to judge how close to an optimal set a solution set is. Plus, the hypervolume is not useful when not all solutions contribute to the overall goal, for example when the goal is straightforward and linear [Hayes et al., 2021].

Algorithms that performs well across a broader range of practical scenarios are generally preferred as it allows decision makers to have more options, covering the full spectrum of trade-offs between different objectives. And while hypervolume correlates with the spread of the solutions, it is not equal to the spread of the solutions [Hayes et al., 2021]. Using a separate metric for measuring spread of the solutions have been used to preferentially select more diverse trajectories during training [Abels et al., 2018], [Reymond et al., 2022]. In addition to the spread of the solutions, a high resolution is also desired. Intuitively, this means that it is preferred not only to have a broad range of solutions but also a detailed and fine-grained set of options within that range so that

decision-maker would have more options to choose from. For this hypervolume metric have been combined with sparsity metrics [Xu et al., 2020a].

Epsilon metric (ε) [Zitzler et al., 2008] indicates how close are two solution sets. It can be used to compare solution sets produced by two algorithms or to compare a single solution set to a Pareto front to indicate how much worse the solution is compared to Pareto front. It's formalized by formula 1.10, where:

- n is the number of objectives.
- $V^{\pi} \in \mathbb{R}^d$ is a d-dimensional value vector corresponding to a policy π, where d is the number of dimensions, which could refer to different criteria, metrics, or aspects that the policy impacts.
- PF stands for the Pareto front, which is a set of policies that are considered non-dominated in a multi-objective optimization framework. A policy V^{π} is non-dominated if there is no other policy that is better in all objectives.
- The solution set S is an ϵ-approximate Pareto front. This means that for each policy value vector V^{π} on the actual Pareto front PF, there is at least one policy value vector $V^{\pi'}$ in the solution set S, which is "close enough" to V^{π} such that for every objective i, the value $V_i^{\pi'}$ is at most ϵ smaller than V_i^{π}.

$$I_{\epsilon+} = \inf\{\epsilon \in \mathbb{R} \mid \forall V^{\pi} \in PF, \exists V^{\pi'} \in S : V_i^{\pi} \leq V_i^{\pi'} + \epsilon, \forall i \in \{1, \ldots, n\}\} \tag{1.10}$$

The ϵ-approximate Pareto front S is a set of policies that are within an ϵ margin of the Pareto front in every objective. The $I_{\epsilon+}$ is looking for the smallest ϵ that makes this true - quantifying how close the solution set S is to the true Pareto front PF. The smaller the $I_{\epsilon+}$, the closer the approximate Pareto front is to the actual Pareto front, meaning the solution set S provides a good approximation of the optimal set of policies.

For most real world problems it is very difficult or may be impossible to compute the actual Pareto front so such benchmarks are usually pretty limited. Also it has been argued that these metrics can be difficult to interpret, where it is not clear which agent is better when competing agents are outperforming each other in different metrics [Zintgraf et al., 2015]. Therefore it has been suggested that it is better to use utility based metrics [Zintgraf et al., 2015]. Where user utility (preferences) is known it is best to directly compare the scalarised utility values, as this is actually what agents are trying to maximise and what matters to the users. When user utility is not known different metrics based on utility can be used. One example of such metrics is an expected utility metric (EUM) [Zintgraf et al., 2015]. EUM indicates how much utility user will get from the agent on average given solution set and set of scalarisation functions defined by formula 1.11, where:

- $EUM(S, f, P)$ denotes the Expected Utility metric given a set of policies S, a utility function f, and a probability distribution P.
- $V_w^{\pi} = f\left(\mathbf{V}^{\pi}, \mathbf{w}\right)$ is a set of scalarisation functions for the solution set.
- $P(w)$ is probability distribution $P(w)$ over weights w. When expectation cannot be computed exactly for a given solution set, sampling methods, can be used where a set of weight samples are used to approximate this expected value.

$$EUM(\mathcal{S}, f, P) = \mathbb{E}\left[\max_{V^{\pi} \in \mathcal{S}} f\left(\mathbf{V}^{\pi}, \mathbf{w}\right)\right] \tag{1.11}$$

1.9 Empirical Evaluation, Generalization

Empirical evaluation of models is a crucial part of machine learning especially when comparing different algorithms [Vamplew et al., 2011] meaning that to find out how well a RL algorithm actually performs it needs to be observed and evaluated in action rather than only in theory. There are a wide variety of tasks and benchmarks proposed and used for such evaluation for single objective RL. The variety of available benchmarks is important because like in other machine learning methods, an important criteria when evaluating RL algorithm is it's ability to solve different problems without users having to make specific adjustments to the algorithm depending on a problem. While an algorithm may perform well in one task it might fail in another or it may require extensive fine tuning which is time expensive. This is sometimes called method overfitting [Whiteson et al., 2009]. One of the suggested solutions is called generalized domains [Whiteson et al., 2009]. The idea is to use sequential decision problem generators, which would generate tasks of the same class but with different MDPs, ensuring the algorithm performs well in a specific domain but also does not overfit to one specific task. Therefore for RL research it is important to have a wide variety of tasks that could be easily accessible by different RL agents. To make a valid comparison of two RL algorithms it is important that standardised evaluations methodologies are used.

To help with RL algorithm comparison and to speed up development of new algorithms various RL training toolkits have been developed. They offer a common interface for wide variety of tasks, often called environments. One of the most well known of such toolkits is Gymnasium [Towers et al., 2023] (formerly known as OpenAi gym [Brockman et al., 2016]). This toolkit allows agents to be trained independently of the language the agent was written in as it focuses on providing a common interface for environments. Gymnasium offers various tasks such as Atari games, robot control, algorithm imitation and others. A multi-objective adaptation of Gymnasium is called MO-Gymnasium [Alegre et al., 2022] and it contains multi-objective versions of many of the same environments as in Gymnasium and also a number of problems unique to multi-objective reinforcement learning. These toolkits are open source and many of the environments are contributed by third party developers as outside contributions are encouraged.

1.10 Artificial Neural Networks

Essentially an artificial neural network is a function approximator, capable of approximating non-linear functions. The information required for making the approximation is stored in the network's neurons' biases and weights, with activation functions providing non-linearity. The various ways neurons are connected, the number of neurons, and the choice of activation functions are part of the neural network's architecture.

Biological neurons, which are the basic units of the brain, receive signals through dendrites, process these signals, and transmit them via axons. This biological process inspired the development of artificial neural networks, where each node mimics the function of a biological neuron [Russell and Norvig, 2010]. The development of neural networks can be traced back to the 1940s

with the introduction of the McCulloch-Pitts neuron model. This was followed by the perceptron (see Fig. 5) in the 1950s, a simple neural network capable of linear classification.

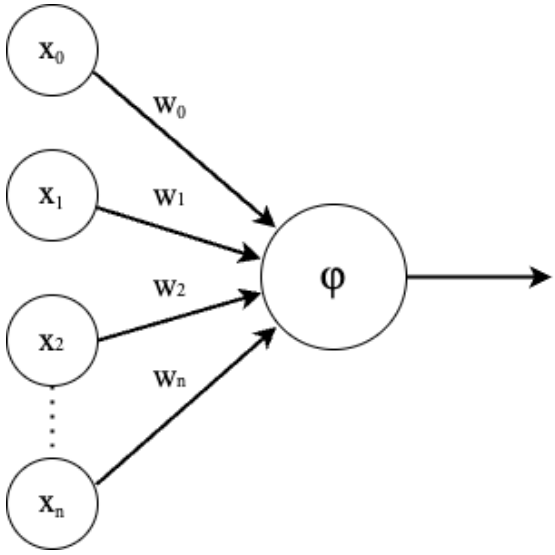


Figure 5. Artificial neuron - perceptron.

It can be formalized as 1.12:

$$y_k = \phi\left(\sum_{j=0}^{m} w_{kj} x_j\right) \tag{1.12}$$

where neuron y_k value is defined as an output of ϕ is activation function. w_{kj} represents the weight of the connection from the j-th input to the k-th neuron and x_j denotes the j-th input to the neuron.

The backpropagation algorithm, introduced in the 1980s, enabled the training of multi-layer networks, marking a significant advancement in the field [Rumelhart et al., 1986].

Artificial neural network is comprised of simple computation elements called neurons and connections between those neurons (see Fig. 6). These neurons are stacked in layers - the input layer, hidden layers, and the output layer. The input layer receives the raw data, while the hidden layers perform computations and feature extractions. The output layer produces the final result or prediction. Neurons are usually only connected to the neurons in a subsequent layer.

Connection between neurons is associated with a numeric number called weight [Wang, 2003]. During a forward pass (inputting data through the network to produce an output/prediction) each neuron multiplies each input (connection) by their weight, sums up results and adds a bias (a constant value) to this sum. The result is then passed through an activation function. This output is either fed into neurons in the next layer or forms part of the network's final output, depending on the neuron's position in the network. During a forward pass, the network processes input data sequentially through its layers. Neurons within each layer take inputs from preceding layers, apply specific weights to them, and then transmit the output through an activation function. This sequence is repeated until it gets to the final layer.

A very important aspect of neural network design is the selection of the activation function for each neuron in a layer, as it has a major impact on the network's ability to process and learn complex data patterns.Functions like Rectified Linear Unit (ReLU)[Hahnloser et al., 2000] are

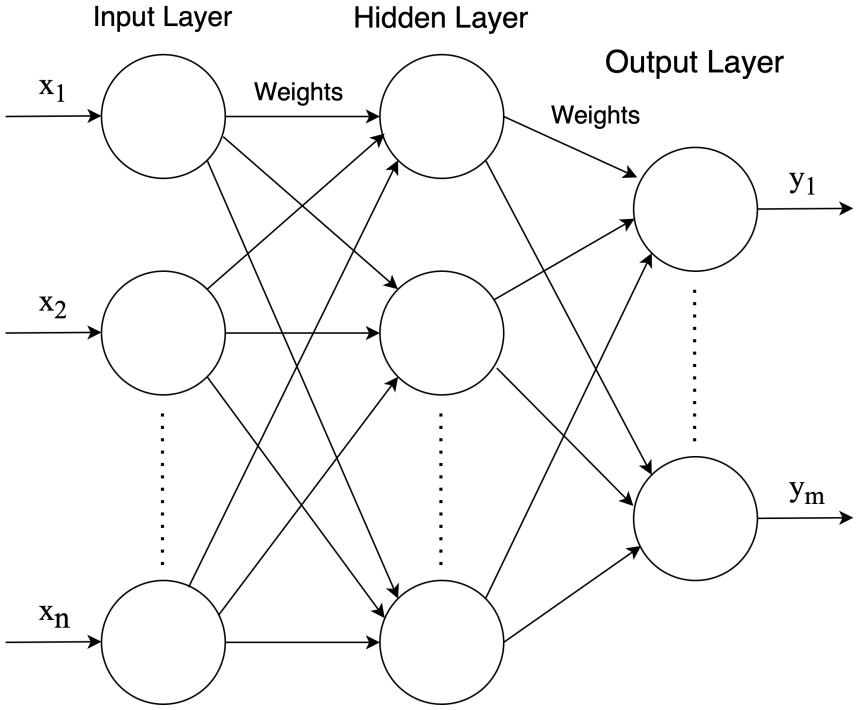


Figure 6. Artificial neural network model where x_1 to x_n represents input data, y_1 to y_m - output. Arrows represent weights, circles represents neurons.

popular in hidden layers due to their computational efficiency and ability to mitigate the vanishing gradient problem. Sigmoid and softmax functions are often used in output layers for binary and multi-class classification tasks, respectively (these functions are explained in more depth in [Russell and Norvig, 2010]).

Weights of the neuron connection determine the strength and direction (positive or negative) of the influence one neuron has on another. The weights and biases are adjusted through a process known as learning or training. During training neuron weights and biases are iteratively adjusted in such way as to minimize a defined loss function. There are various training algorithms although usually it is done by first doing forward passes to compute predictions then backpropagation to calculate gradients, and optimization algorithms like gradient descent to update parameters. Training continues for multiple iterations until a stopping criterion is met.

The effectiveness of the network during training is evaluated by a loss function, which quantifies the discrepancy between the predicted forecasts and the actual predicted values. This metric is an indicator of the network's efficiency and is used in the optimisation process. Mean Squared Error is common for regression, while Cross-Entropy is often used for classification [Russell and Norvig, 2010]. The selection of a loss function influences the way errors are calculated and sent back through the network while it is being trained.

During ANN training the method usually used is backpropagation. During backpropagation network's weights are adjusted in such way as to minimise the error seen in the output. It calculates the gradient of the loss function for each weight, using the chain rule in calculus. The gradient helps update the weights, usually through a gradient descent algorithm. The size of the steps in gradient descent is set by the learning rate, which is a hyperparameter [Russell and Norvig, 2010].

1.10.1 Optimization Algorithms

Beyond basic gradient descent, several advanced optimization algorithms are used in neural network training. These include:

- Stochastic Gradient Descent (SGD) - SGD is a simple optimization algorithm that updates the weights of the neural network based on the gradient of the loss function with respect to the weights. It updates the weights in small batches, which can result in faster convergence and lower memory requirements. However, it can sometimes get stuck in local minima.[Russell and Norvig, 2010].

- Momentum - adds a fraction of the previous update to the current one, helping to accelerate SGD and dampen oscillations.

- Adaptive Learning Rate Methods - algorithms like Adam [Kingma and Ba, 2017] adjust the learning rate during training for each weight individually, often leading to better performance and faster convergence.

The initialization of weights in a neural network significantly influences its training efficiency and convergence. Techniques like Xavier/Glorot [Glorot and Bengio, 2010] or He initialization [He et al., 2015] are commonly used. Such techniques initialize the weights based on the size of the previous layer, ensuring consistent variance of activations across different layers.

Optimisation challenges for neural networks are usually related to finding the most efficient way to adjust the weights and biases of the network to minimise losses. These challenges include:

- Vanishing and Exploding Gradients - when training deep networks, as gradients are passed back through layers, they can either shrink to a very small size (vanish) or grow too large (explode). This can make learning very slow or make it unstable. Using techniques such as ReLU activation functions, batch normalization, and various initialization techniques can help overcome these problems [Pascanu et al., 2013].

- Hyperparameter Tuning - Selecting the right hyperparameters (like learning rate, number of layers, size of layers, type of activation function) is crucial for the performance of neural networks. Techniques like grid search, random search, and Bayesian optimization are used to explore the hyperparameter space and find the optimal configuration [Bergstra and Bengio, 2012].

- Avoiding local minima - local minima is a point where the loss is lower than in the nearby points but not the lowest possible loss. This is challenging because for the learning algorithm there are few ways to know if it has reached a local minima and not a global one and settling for such point means sub-optimal performance.

- Overfitting and Underfitting - Overfitting occurs when the model learns the training data too well limiting it's ability to generalise to new unseen data. Underfitting happens when the model is too simple to capture the underlying patterns. Regularization, dropout, and cross-validation are common strategies to address these challenges. The capacity of a neural network refers to its ability to model complex functions. A network with higher capacity can model more complex functions but also risks overfitting the training data. Balancing this capacity is crucial for effective learning and generalization.

1.10.2 Overfitting, Mitigation and Stabilization

To select appropriate model, various models and hyperparameters are usually empirically tested and those which perform the best on validation set are selected. The number of layers, the number of neurons in each layer, and how these neurons are connected significantly impact the network's ability to process and learn from data with the key aim in training neural networks being generalization, which is the network's skill in handling new, unseen data effectively.

To avoid overfitting, various regularisation strategies are used. These include L1 and L2 regularisation, which involves adding penalties to the loss function depending on the size of the weights. This approach encourages the model to choose smaller values for the weights, thus promoting simpler models that are less prone to overfitting to the training data. Another is dropout [Srivastava et al., 2014] - randomly setting a fraction of the input units to 0 at each update during training, which helps prevent over-reliance on any one neuron and promotes a more robust network. Additionally batch normalization [Ioffe and Szegedy, 2015] is used to stabilize the learning process and cut down on the training time needed for a network by normalizing the input for each layer. It does this by adjusting and scaling the output of each layer to have a mean of zero and a variance of one, which helps reduce internal covariate shift.

1.10.3 Reward Conditioned Policies

Combining the foundational concepts of reinforcement learning and supervised neural networks learning techniques Reward Conditioned Policies (RCP) [Kumar et al., 2019] introduce a new RL method capable of solving RL tasks with less hyperparameter tuning and potentially better sample efficiency than the traditional methods.

Traditional RL methods focus on maximising rewards. In contrast RCPs focus on correctly replicating trajectories, no matter if they are optimal or sub-optimal. The main insight is that leveraging neural networks generalization capabilities it is possible to produce new trajectories that are better than the ones which were used during training of RCPs. RCP algorithm has two variants, where one focuses on achieving expected reward while the other aims to optimize the advantage value of actions within specific states. The algorithm takes these steps: 1. Initialization of replay buffer where random trajectories are used to fill the replay buffer. 2. Sample target values $\hat{Z}$ from a distribution from replay buffer. In case of RCP target values is cumulative reward that should be achieved by a policy. 3. Executing current policy using sampled target value $\hat{Z}$ and adding it to the replay buffer. A trajectory consists of a sequence of states, actions, and observed rewards. 4. Update the current policy using replay buffer using traditional supervised learning methods. With each iteration algorithm should replicate trajectories better and as the policy is exposed to a variety of scenarios and outcomes, it should generalize, learning to perform well across a range of conditions and not just the ones it was initially trained on.

1.11 Pareto Conditioned Networks

Pareto Conditioned Networks (PCN) described in [Reymond et al., 2022] is method heavily inspired by RCP-R algorithm, adapted to work in multi-objective environments. PCN still consists of a single neural network trained to replicate trajectories from a replay buffer. However the rewards are vector and trajectories stored in replay buffer pruned in such way as to store only those trajectories with the most diverse rewards across each goal. This is done by assigning each trajectory a 'dominating score' which combines crowding distance and L2-norm distance. Only a

predefined number of trajectories with largest 'dominating score' are stored in replay buffer. The buffer is updated the same way as in RCP case - by continuously adding new trajectories generated by the network. To generate new trajectories, a target values must be selected (that the network will try to achieve). This is done by randomly choosing a non-dominated return and its corresponding horizon from the dataset, and then increasing the target return for one of the objectives. The increase of the desired value for that objective is based on a standard deviation from all non-dominated returns. In such way the new target return is challenging yet hopefully achievable, pushing the network to explore and extend the Pareto front. In a similar way desired horizon is also reduced for selected trajectory subtracting a small, predefined amount. This gradual reduction in horizon encourages the network to focus on achieving the set objectives more efficiently.

The dominating score is used to keep more diverse solutions in replay buffer while retaining useful experiences from past solutions. In this case solution is a trajectory and for comparison V-value (expected cumulative return) is used. The diversity is needed to allow agent to explore the environment and not converge on a sub-optimal solution, which would happen only if the pareto optimal solutions were kept in the replay buffer.

The dominating score is calculated by combining crowding distance [Deb et al., 2000] metric and negative L2-Norm (I_{l2}) distance to the nearest non-dominated solution. L2-Norm is described by formula 1.13, where:

- $I_{l2,i}$ is the L2-norm distance score for the $i-th$ solution in the dataset.
- p_i is is the position (in objective space) of the $i-th$ solution in the dataset.
- p_k is the position of a non-dominated solution in the current coverage set.
- $\hat{\Pi}$ denotes the set of current non-dominated solutions.

$$I_{l2,i} = -\min \|p_i - p_j\|_2,\ p_j \in \hat{\Pi} \tag{1.13}$$

The crowding distance is calculated by first, sorting the solutions for each objective. Sorting is done by arranging the solutions for each objective, based on their values for that objective. This sorting is independent for each objective, resulting in a different ordered sets of solutions for each objective. Then for each non-boundary (those that are not the best or worst for any given objective) solution, the normalized differences are summed in each objective's value between the solution and its immediate neighbors (the next and previous solutions in the sorted list for each objective), with each difference normalized by the objective's total range in the population. Initially boundary solutions are set to infinity. The crowding distance for a solution is the sum of these normalized differences across all objectives. The result means that V-values with low scores have close neighbours and V-values with high scores are more unique solutions.

After L2-Norm I_{l2} and crowding distance I_{cd} are calculated, they are combined and the result is then used to prune the experience replay buffer. Dominating score is described by formula 1.14, where:

- $I_{ds,i}$ is dominating score for the $i-th$ solution in the dataset.
- I_{l2} is L2-Norm.
- I_{cd} is crowding distance.

- c is a small penalty to crowded points to identify and eliminate redundant points in the coverage set.

$$I_{ds,i} = \begin{cases} I_{l2,i} & \text{if } I_{cd,i} > 0.2 \\ \frac{I_{l2,i}}{2} - c & \text{if } I_{cd,i} \leq 0.2 \end{cases} \tag{1.14}$$

A few drawbacks of PCN can also be noted. Original paper states that method offers good results with less hyperparameter tuning compared to other RL methods. In original paper a different neural network was used for different environment with different hyperparameters, applying strategies like one hot encoding of the state, increasing step limits to the environments to improve performance. Also there were no examples with sparse reward environments like mountain car indicating that the algorithm might have difficulties training in such environments. In the following sections it will be shown that without these strategies, agent performance is sub-optimal which means that the algorithm still requires careful tuning for each task. It will also be demonstrated that performance varies noticeably between executions.

Overall, the PCN has achieved good results in deterministic multi-objective problems with discrete action spaces. Compared to other algorithms, it offers better performance in scenarios involving a large number of objectives, also offers good sample efficiency and makes little assumptions about utility functions. It is also capable of discovering non-convex Pareto fronts. However, its performance in stochastic environments and with continuous action spaces remains not explored, as noted in the original paper. This paper aims to explore these areas further and prove that a modified version of PCN is capable of effectively handling stochastic scenarios and continuous action spaces, potentially broadening its applicability in more multi-objective problems.

2 Methods

To adapt PCN to work in continuous action spaces, the underlying neural network structure, training and exploration strategy had to be updated. Because PCN was originally designed and tested for discrete action spaces, the changes also had to be verified to prove that PCN can be used in continuous action spaces. To assess the performance of the modified PCN algorithm, a series of multi-objective environments were used. Finally, to provide a benchmark for PCN's performance in these scenarios, a comparative analysis was conducted with the GPI-PD algorithm, also sourced from the MORL baselines.

These adaptations to the PCN framework were implemented based on the PCN model in the MORL baselines repository [Felten et al., 2023] which provides stable implementations of various MORL algorithms, providing a solid foundation for experimentation.

The primary modification needed to make PCN work in continuous action spaces was reconfiguring the output of the PCN. Instead of multiple discrete outputs, the network was changed to output a single continuous prediction, representing the action of the policy. This change transformed the training problem of the network from a classification to a regression. Consequently, the loss function employed during training was transitioned from cross-entropy loss, typically used in classification tasks, to Mean Squared Error (MSE), which is suitable for regression problems.

Next, the exploration strategy, a critical component in reinforcement learning, was modified. In discrete action spaces, PCN increased exploration by sampling actions from a categorical distribution. The probability of sampling each action was directly proportional to its confidence score, calculated using the softmax function in the network's final layer [Reymond et al., 2022]. This approach, however, is incompatible with continuous action spaces where the output value is a direct representation of the action itself.

To address this, several alternative exploration strategies during training were explored:

- Implementing an epsilon-greedy algorithm, where actions are chosen randomly with a certain probability, promoting exploration.
- Adding noise to the action predicted by the PCN with a random value sampled from a distribution.
- Adding noise to the action predicted by the PCN with a random value sampled from a distribution, where the value is scaled, with the scale decreasing during each timestep until it reaches 0.

2.1 Environments for Performance Evaluation

Different multi-object environments were used to evaluate the performance of the modified Pareto conditioned network algorithm. These environments were specifically chosen to vary in complexity and have unique characteristics and challenges. The approach was designed to provide a comprehensive assessment of the performance of the algorithm in a variety of scenarios and to identify its strengths and potential areas of improvement for solving different types of multi-objective problems. All the environments which were used in testing the algorithms were implemented using MO-Gymnasium ([Felten et al., 2023]) library which is an extension of the single-objective gym framework [Towers et al., 2023]. This approach ensured a consistent and robust testing methodology across different environments.

2.1.1 Continuous Convex Deep Sea Treasure

Initially, a continuous variant of the Deep Sea Treasure (DST) environment, inspired by the well-known DST environment [Vamplew et al., 2011], was implemented and tested. While a version of continuous DST was described in [Takayama and Arai, 2022], an implementation was not provided by the authors of the original paper and it was not available in MORL Gymnasium, the implementation had to be implemented from scratch. In the classical DST environment the agent is navigating a grid world with the goal of reaching treasures. Treasures have different values with those further away from the beginning having larger ones. Each step consumes fuel. This creates two conflicting multi-objective goals - consume least amount of fuel possible and find the treasure with the highest value. Episode terminates when the treasure is reached, or a predefined number of timesteps has elapsed. Continuous Convex Deep Sea Treasure (CCDST) environment is a modification of DST where agent takes continuous actions, with the action representing the angle of the direction the agent will move in, while the agent moves by 1 during each timestep. Agent can take maximum of 6 steps. Visualisation of the environment is provided in Fig. 7.

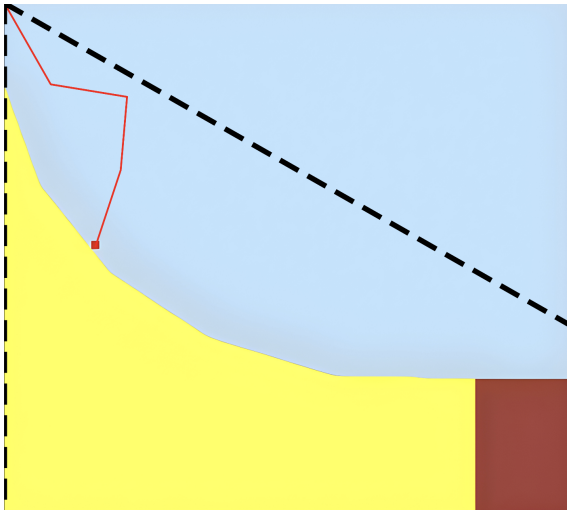


Figure 7. Continuous convex deep sea treasure environment visualisation.

The reward agent receives is described by 2.2, where:

- r_1 is the first reward.
- r_2 is the second reward.
- s_x and s_y is the agent position.

$$r_1 = -1 \tag{2.1}$$

$$r_2 = \begin{cases} \sqrt{25 - \left(\sqrt{s_x^2 + s_y^2} - 6\right)^2} + 1 & \text{(obtain reward)} \\ 0 & \text{(otherwise)} \end{cases} \tag{2.2}$$

Table 1. Actions, Rewards, and Observations in Continuous Convex Deep Sea Treasure

	Description	Min	Max
Reward 1	Time penalty, -1 each step	-1	1
Reward 2	Treasure reward	0	6
Observation 1	Position of the agent along the x-axis	$-\infty$	∞
Observation 2	Position of the agent along the y-axis	$-\infty$	∞
Action 1	The angle of the direction the agent will move in.	0	1

2.1.2 Continuous Lunar Lander

Another environment used for evaluation of PCN in continuous spaces was Continuous Multi-Objective Lunar Lander (see Fig. 8), provided by MO-Gymnasium [Alegre et al., 2022]. It is a version of a Lunar Lander environment from RL gymnasium [Towers et al., 2023], where the goal of an agent is to land a spacecraft on a designated landing spot on the moon's surface. It is set in a 2D space, influenced by gravity, where the lander has to control its fall using thrusters. The agent, can use the main or side engines to adjust the lander's speed and position. The environment's state consists of the lander's position, speed, angle, and any contact with the ground. The goal for the agent is to achieve a gentle landing on the pad by reducing descent speed and accurately positioning the lander.

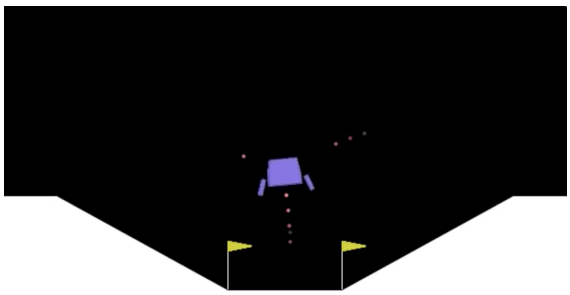


Figure 8. Continuous lunar lander environment visualisation.

Successful landings earn rewards, while crashes or veering off course result in penalties. The challenge is to find the best strategy for efficient fuel use, without crashing while landing precisely on the designated landing spot. Originally the environment is stochastic with starting state of the spacecraft and moon relief randomly changing between episodes. Because original discrete PCN was tested on deterministic environments, testing of the modified PCN algorithm was also done in lunar lander environment modified to be deterministic.

Table 2. Actions, Rewards, and Observations in Continuous Multi-Objective Lunar Lander environment

	Description	Min	Max
Reward 1	-100 if crashed, +100 if landed successfully	-100	100
Reward 2	Shaping reward $-\|\|\mathbf{action}\|\|^2$	$-\infty$	∞
Reward 3	Fuel cost for the main engine $-\|\|\mathbf{action}\|\|^2$	-1	0
Reward 4	Fuel cost for the side engine $-\|\|\mathbf{action}\|\|^2$	-1	0
Observation 1	Position of the car along the x-axis	$-\infty$	∞
Observation 2	The coordinates of the lander in x	-1.5	1.5
Observation 3	The coordinates of the lander in y	-1.5	1.5
Observation 4	Linear velocities in x	-5	5
Observation 5	Linear velocities in y	-5	5
Observation 6	Angle	$-\pi$	π
Observation 7	Angular velocity	-5	5
Observation 8	Left leg is in contact with the ground	-0	1
Observation 9	Right leg is in contact with the ground	-0	1
Action 1	Throttle of the main engine	-1	1
Action 2	Throttle of the side engines	-1	1

2.1.3 Continuous Mountain Car

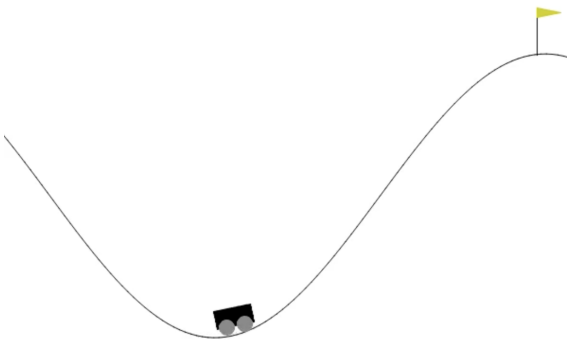


Figure 9. Continuous mountain car environment visualisation.

The continuous mountain car environment (Fig. 9), another platform used for testing, is originally derived from an analogous single-object environment [Moore, 1990] and also provided by MO-Gymnasium [Alegre et al., 2022]. In this environment the agent controls a car, which starts in the bottom of a valley. The goal is reaching the top of the valley, however the car has limited power therefore the agent has to learn how to build momentum by driving back and forward to drive out of the valley. The main challenge of this environment is sparse reward, because the agent receives the reward only when reaching the top of the mountain. This environment is also stochastic with starting position of the car being random. To make the environment deterministic, starting position was modified to always be the very bottom of the valley. The environment state, actions and observations are described in table 3.

Table 3. Actions, Rewards, and Observations in Continuous Multi-Objective Mountain Car environment

	Description	Min	Max
Reward 1	Time penalty, -1 each step	-1	1
Reward 2	Fuel reward, $-\|\|\mathbf{action}\|\|^2$	$-\infty$	∞
Observation 1	Position of the car along the x-axis	$-\infty$	∞
Observation 2	Car velocity	$-\infty$	∞
Action 1	The force applied to the car during the timestep	-1	1

2.2 Comparison

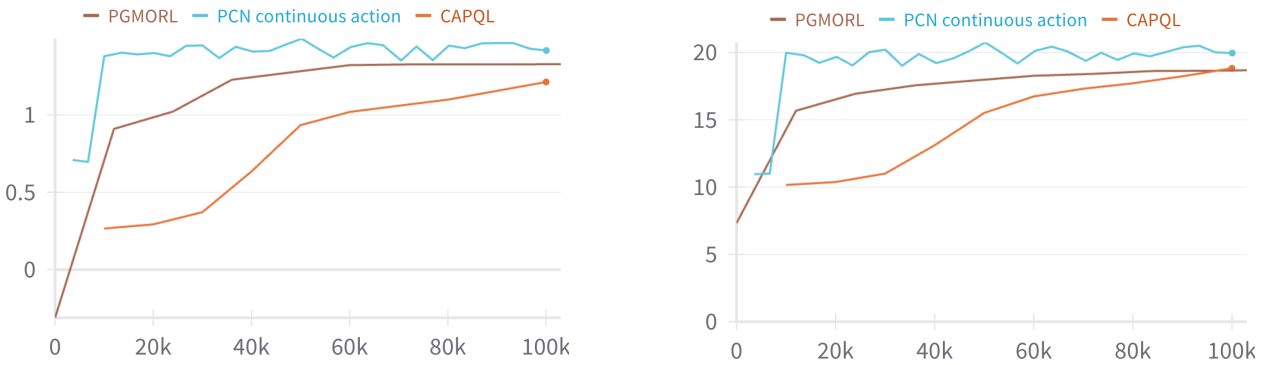
In conclusion, to establish a baseline against which to assess the effectiveness of PCNs in these scenarios, a comparative analysis was conducted with a Q-learning based MORL algorithm called CAPQL [Lu et al., 2023] and Policy Gradient Methods for Multi-Objective Reinforcement Learning (PGMORL) [Xu et al., 2020b] algorithm, also sourced from the MORL baselines. The comparison provided a reference point for the performance of PCN. Hyper-parameter search was performed across different environments using grid search method. Parameters used for the grid search are provided in the appendix.

3 Results

Overall more than 1600 experiments were conducted, taking more than 500 hours of compute time. Most of the experiments were conducted on 2021 Macbook air with M1 processor and some were also conducted using Google Colab [goo, 1 06] runtimes. Changes were implemented using Python 3.11. Test run results were saved in and visualised using WandDb [Biewald, 2020]. All test run results and code changes are available in the appendix. The following section contains the essential part of these experiments.

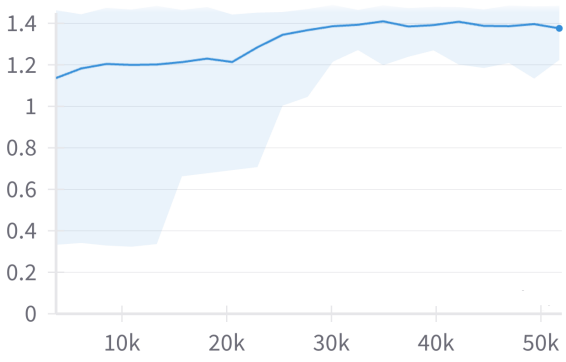
3.1 Continuous Convex Deep Sea Treasure

The modified PCN demonstrated capability to solve simple MORL problems in CCDST environment. The PCN agent was able to outperform both PGMORL and CAPQL in both Expected Utility Maximization (EUM) and hypervolume metrics as demonstrated in Figure 10 during an evaluation run. The best performing PCN agent also outperformed other algorithms in EUM and hypervolume metrics during hyperparameter search as shown in Figures 11, 12 and 13. At the same time PGMORL was more stable and required less steps on average to converge to a near optimal solution. As the environment is quite simple and PCN performed well without more advanced techniques, an exploration was facilitated by a random value sampled from a value distribution multiplied by 0.1 and added to an action during each step. Using no noise for exploration resulted in suboptimal performance. Although loss function result is not a criteria when evaluating MORL algorithm performance it is still useful when checking if models of the algorithms are learning in a stable way. Therefore the loss function results are provided in figure 14. It must be noted that the data which is used for training the models is changing as the agent performs new actions and gather new data during the training process. This can impact the stability of the loss during training. However as visible in the provided graphs, the loss was overall decreasing. Another thing worth noting is that the CCDST environment is rather simple, and was used as a general test to check if the algorithm is able to solve even the most basic MORL tasks.

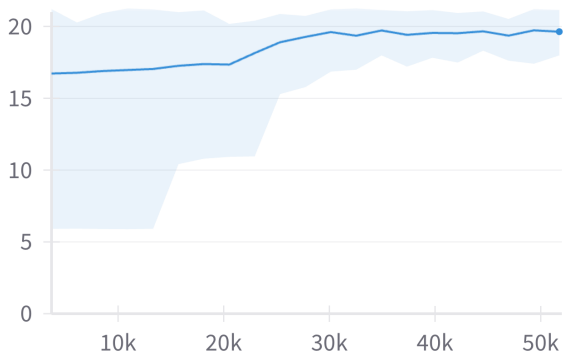


EUM Hypervolume

Figure 10. Metrics for evaluation run with the hyperparameters selected based on the best performing agents during the grid search runs.

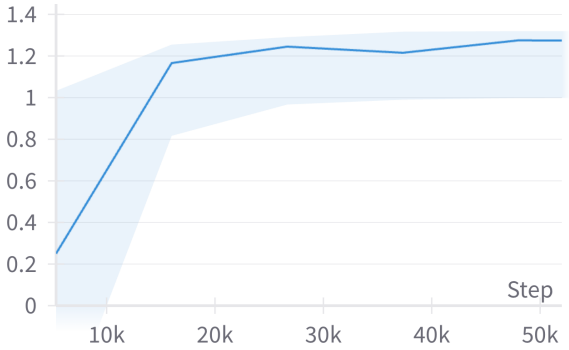


EUM

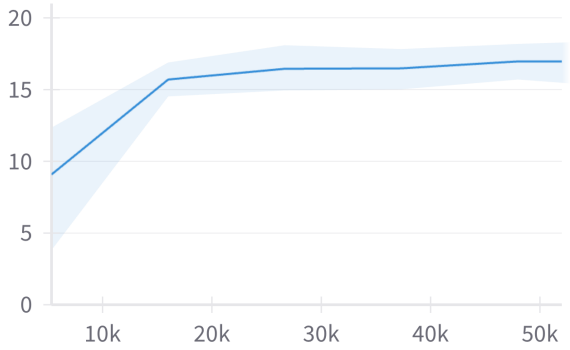


Hypervolume

Figure 11. Metrics for PCN trained in CCDST environment. Blue area denotes min and max values of PCN method trained with different hyperparameters using gridsearch. The highlighted line shows an average value.

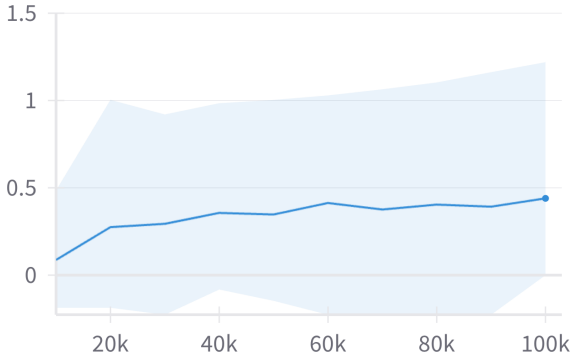


EUM

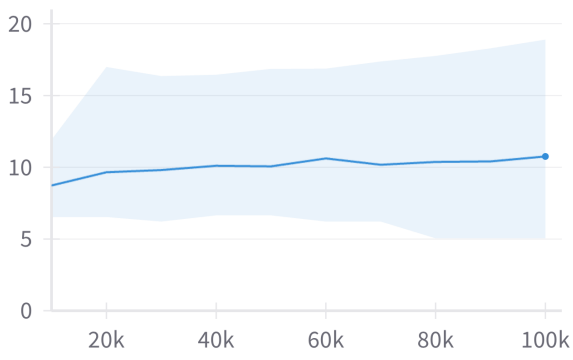


Hypervolume

Figure 12. Metrics for PGMORL trained in CCDST environment. Blue area denotes min and max values of PGMORL algorithms trained with different hyperparameters using gridsearch. The highlighted line shows an average value.

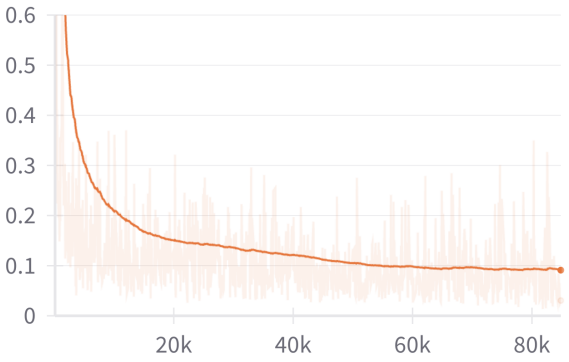


EUM

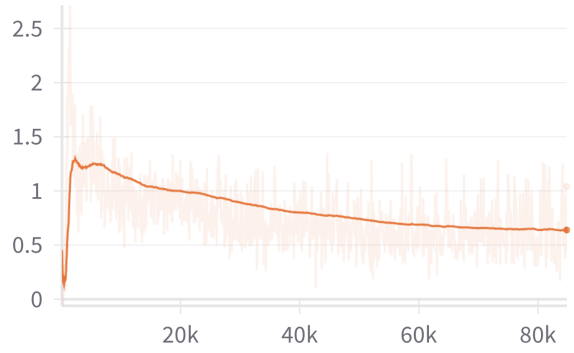


Hypervolume

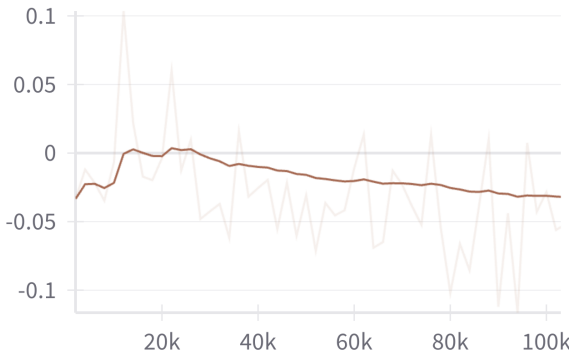
Figure 13. Metrics for CAPQL trained in CCDST environment. Blue area denotes min and max values of CAPQL algorithms trained with different hyperparameters using gridsearch. The highlighted line shows an average value.



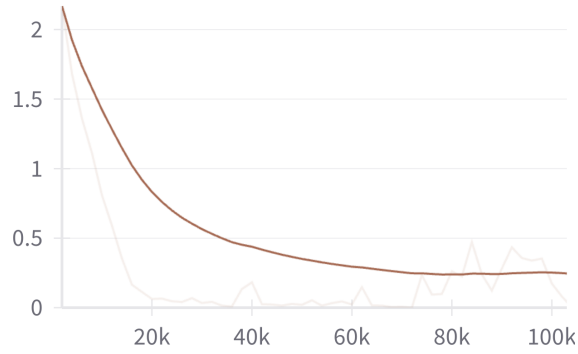
CAPQL critic model loss



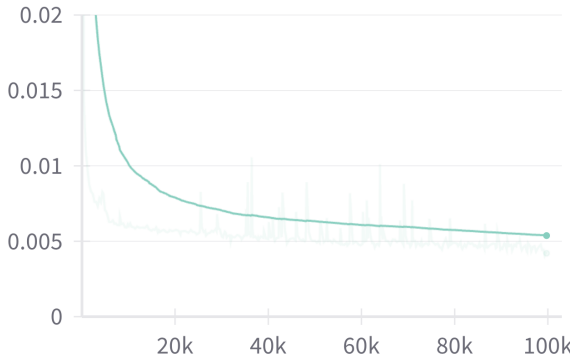
CAPQL policy model loss



PGMORL policy model loss



PGMORL value model loss



PCN model loss

Figure 14. Losses of various networks in tested MORL agents during training in CCDST with the best performing hyperparameters.

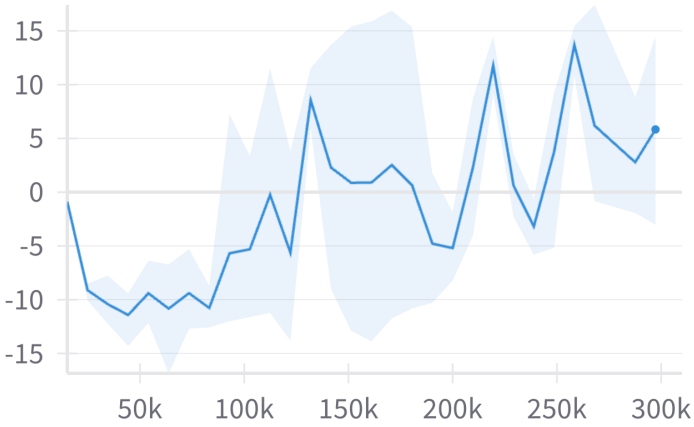
3.2 Continuous Lunar Lander

A more significant challenge was offered by Continuous Lunar Lander environment. Because the environment has more than two objectives and PGMORL algorithm implementation in MORL baselines is limited to two objectives, only CAPQL was used for comparison to PCN.

In this environment the PCN struggled to compete with CAPQL when using static exploration rate. At first initial exploration of hyperparameters was done using 64 PCN agents, using grid search and a limit of 100000 timesteps. As visible in figure 17 this exploration yielded poor results, with agent unable to successfully land agent (1 objective), learn to preserve fuel (objectives 3 and 4) or even to maximise shaping reward (objective 2). The average result seemed to improve at the very end of the 100000 timesteps, therefore hyperparameter search was continued with 1 million timesteps with 32 best hyperparameter sets. Results of these runs are presented in figure 18.

In this case agent was finally able to learn policies that do not crash, preserve fuel and maximise shaping reward. However algorithm was still not able to find policies which allows the lander to successfully touch down on the landing spot, as this would result in objective 1 with positive values. Considering possibility that even more timesteps were needed, a run was performed with 10 million timesteps, however as visible from figure 19 this did not result in improved performance, with algorithm reaching maximum EUM before 1 million timesteps. At the same time CAPQL algorithm reached significantly better and more stable results with almost all agents (16 different hyperparameter sets were tested) that were used for hyperparameter search, as represented in figure 16.

The performance of PCN is highly dependent on the quality and diversity of the trajectories available in the replay buffer, because it is used for training the model which generates new trajectories for training. To help PCN explore more state spaces and collect better initial trajectories to the replay buffer, decaying exploration rate was applied. This resulted in significant improvements as visible in figure 15. Another improvement specific to this task, was adding a Tanh final layer to the neural network model as the environment actions are limited to minimum -1 and maximum 1. With these improvements PCN was able to learn policies that successfully land the lunar lander (figure 15) and the resulting EUM was much closer to that of CAPQL, although the PCN algorithm was more unstable during learning, outputing more variable results during training and resulting performance was still better with CAPQL. Overall in this task PCN was less stable than CAPQL, requiring more steps and fine tuning to reach performance that is somewhat comparable to CAPQL. The fact that PCN learns policies that maximise shaping reward and minimise fuel use while struggling with maximising landing objective might indicate that PCN struggles with delayed rewards. To further test this, Continuous Mountain Car environment was used.

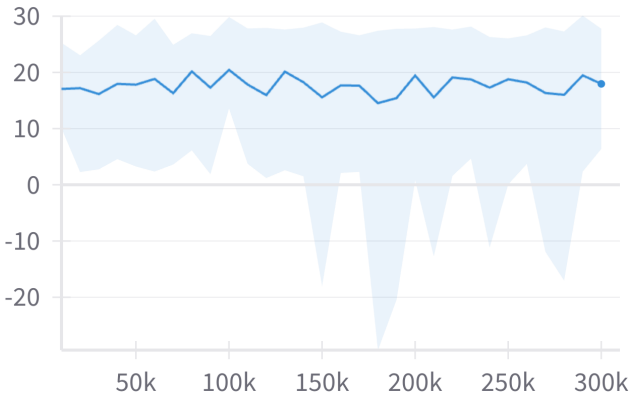


EUM

Obj. 1	Obj. 2	Obj. 3	Obj. 4
73.65	3.31	−31.09	0
73.65	3.31	−31.10	0
1.26	21.71	−110.6	0
−9.34	3.68	−101.8	0
−33.87	49.58	−68.19	0
−65.64	−23.6	−9.22	0
−70.76	−24.06	−0.50	0
−71.12	−28.79	0	0

Pareto front values

Figure 15. PCN training results in Continuous Lunar Lander environment, using decaying exploration noise. . Pareto front value are sampled from one of the best performing runs.

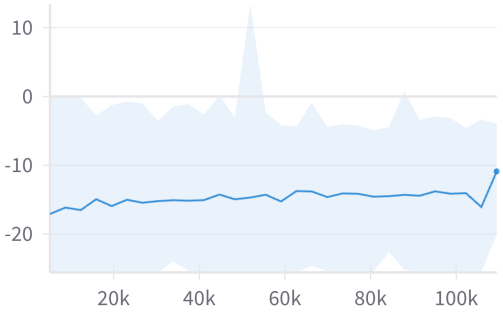


EUM

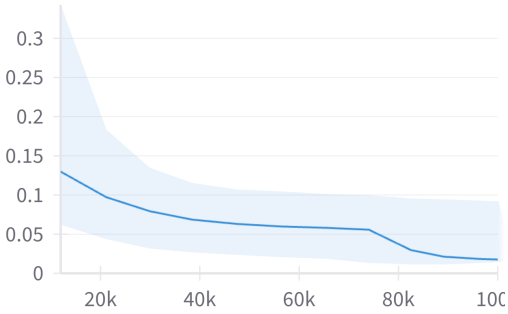
Obj. 1	Obj. 2	Obj. 3	Obj. 4
67.98	3.439	−37.56	0
54.52	3.665	−52.85	0
29.58	83.55	−57.54	0
29.43	83.55	−57.53	0
−43.95	80.83	−55.36	0
−45.52	81.01	−53.17	0

Pareto front values

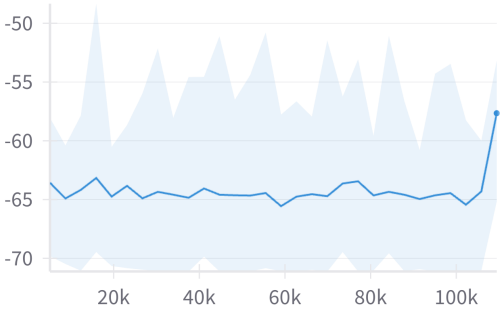
Figure 16. CAPQL training results in Continuous Lunar Lander environment, using 8 different hyperparameter sets. Pareto front value are sampled from one of the best performing runs.



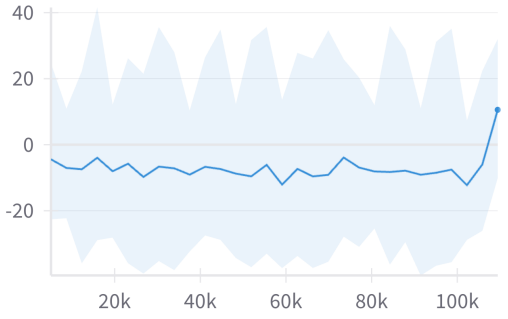
EUM



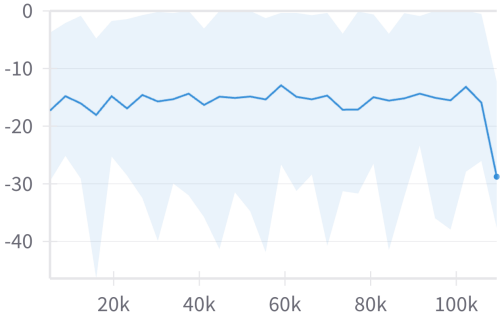
Training loss



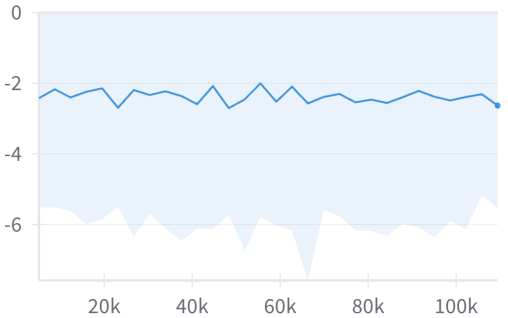
Cumulative return of 1 objective



Cumulative return of 2 objective

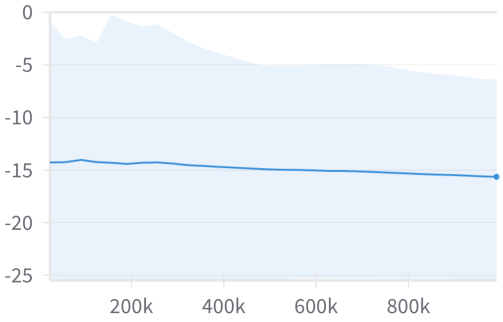


Cumulative return of 3 objective

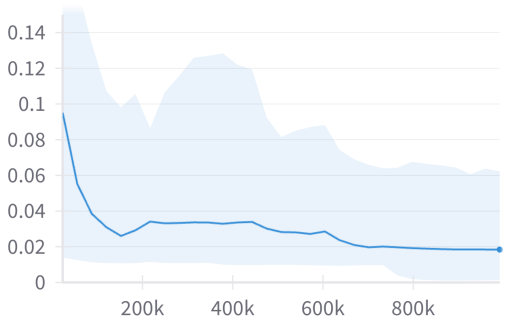


Cumulative return of 4 objective

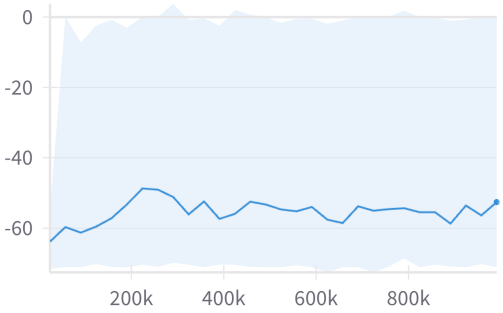
Figure 17. PCN training results in Continuous Lunar Lander environment, using 100 thousand timesteps, 64 hyperparameter sets and static exploration rate



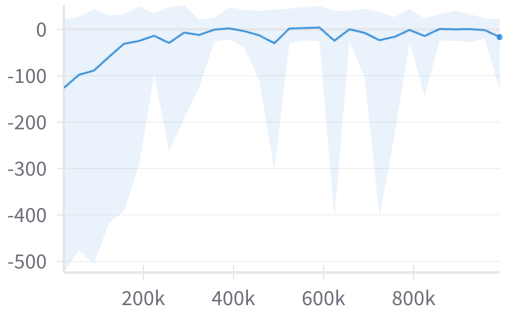
EUM



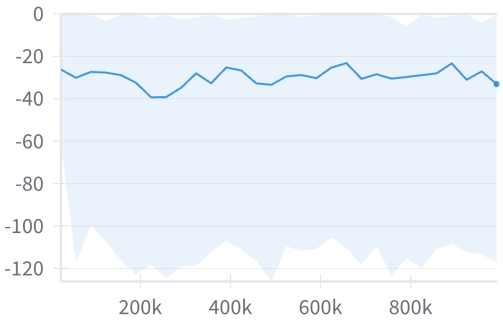
Training loss



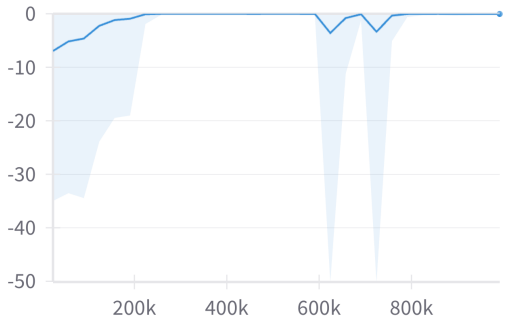
Cumulative return of 1 reward



Cumulative return of 2 reward

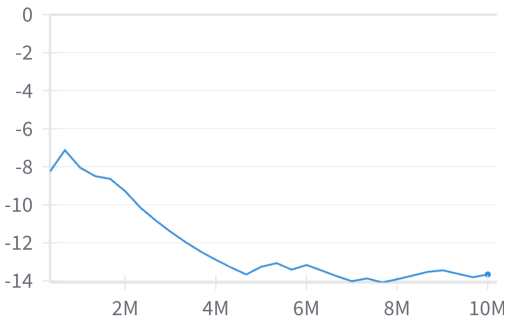


Cumulative return of 3 reward



Cumulative return of 4 reward

Figure 18. PCN training results in Continuous Lunar Lander environment, using 1 million timesteps, 32 hyperparameter sets and static exploration rate



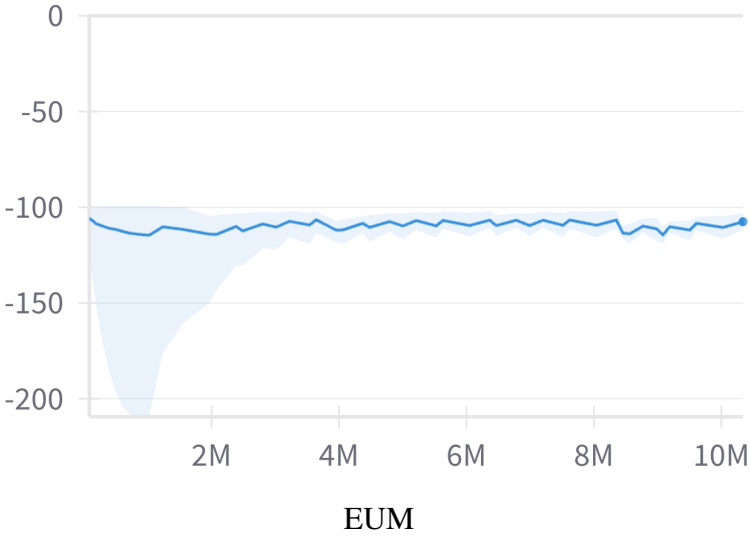
EUM

Figure 19. PCN training results in Continuous Lunar Lander environment, using 10 million timesteps and static exploration rate

3.3 Continuous Mountain Car

In the Continuous Mountain Car environment, the reward is very sparse with positive signal awarded only once, when the task is completed. Therefore this environment is well suited when testing algorithms capability of solving tasks with sparse rewards.

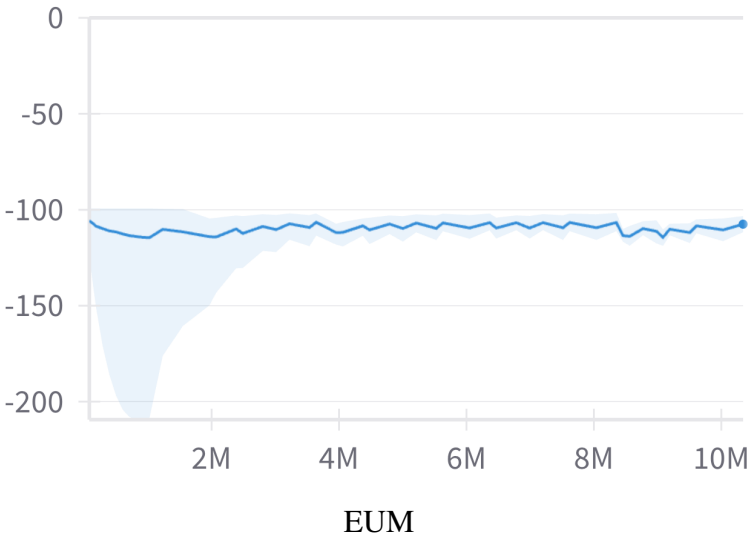
The PCN was unable to solve this task even with 10 million training timesteps, various exploration ratios and various count of exploration decay steps. As indicated by average Pareto front values (figure 20), the agent was able to find policies, where no movement is done, maximising fuel reward (objective 2). However no policies were discovered where mountain car successfully climbs out of the valley (objective 1). In comparison PGMORL was able to find such policies (figure 22), albeit taking much more timesteps compared to what it took to solve Lunar Lander task. CAPQL was also unable to discover policies where car reaches mountain top (figure 21).



Obj. 1	Obj. 2
−198.7	0

Pareto front values

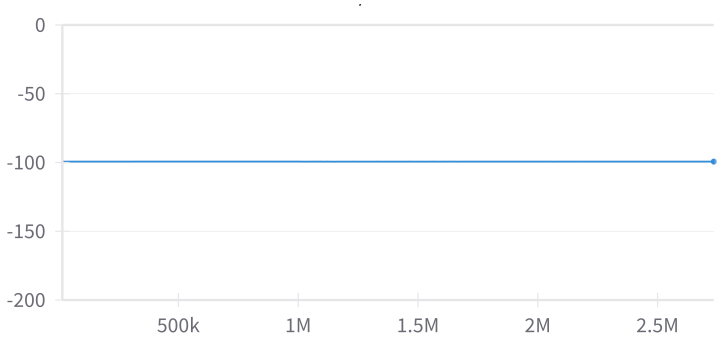
Figure 20. PCN training results in Continuous Mountain Car environment. 64 different hyperparameter sets were used, including decaying exploration rate. Pareto front value are sampled from one of the best performing runs.



Obj. 1	Obj. 2
−198.66	0

Pareto front values

Figure 21. CAPQL training results in Continuous Mountain Car environment. 54 different hyperparameter sets were used. Pareto front value are sampled from one of the best performing runs.



EUM

Obj. 1	Obj. 2
-145.44	-122.46
-194.89	-63.51
-198.16	-60.92
-198.66	-0.02

Pareto front values

Figure 22. PGMORL training results in Continuous Mountain Car environment. 62 different hyperparameter sets were used. Pareto front value are sampled from one of the best performing runs.

Overall an empirical evaluation of the modified Pareto conditioned network method in the environments of a continuous convex deep-sea treasure, a continuous lunar lander and a continuous mountain car justified the effectiveness of the proposed modifications to the PCN neural network architecture.

Conclusions and Recommendations

In this master's thesis, the purpose was to improve the adaptability and efficiency of Pareto conditioned networks in continuous action spaces. This research concludes with the following conclusions:

1. A comprehensive literature review allowed to identify an area of multi objective reinforcement learning investigation where contributions could be made to modify the Pareto conditioned networks method to work in continuous action spaces. The review showed that the original Pareto conditioned network method could not work in continuous action spaces.

2. Adapting the Pareto conditioned networks method to operate in continuous action spaces, new exploration methods were implemented in the underlying neural network.

3. The modified Pareto conditioned networks method was tested in three different deterministic multi objective reinforcement learning environments, demonstrating the effectiveness of modified method in a continuous action spaces.

4. Experiments have shown that the modified PCN can learn and operate in environments with continuous action spaces. The modified PCN method performs best in environments with continuous action spaces, where rewards are dense (rather than sparse). Alternatively, algorithms such as CAPQL and PGMORL may be more appropriate where rewards are sparse.

Future Work

In the future, adapting PCN to work in stochastic environments could be considered. Another area for improving PCN are performance in environments with sparse rewards. Furthermore new strategies for improving trajectories stored in the replay buffer could be investigated. Also while this work tested PCN in 3 environments with continuous actions, more testing could be done with environments where objective count is large.

References

[goo, 1 06] (Verified: 2024-01-06). Colaboratory. https://research.google.com/colaboratory/faq.html.

[Abels et al., 2018] Abels, A., Roijers, D. M., Lenaerts, T., Nowé, A., and Steckelmacher, D. (2018). Dynamic weights in multi-objective deep reinforcement learning. *CoRR*, abs/1809.07803.

[Alegre et al., 2022] Alegre, L. N., Felten, F., Talbi, E.-G., Danoy, G., Nowé, A., Bazzan, A. L. C., and da Silva, B. C. (2022). MO-Gym: A library of multi-objective reinforcement learning environments. In *Proceedings of the 34th Benelux Conference on Artificial Intelligence BNAIC/Benelearn 2022*.

[Bergstra and Bengio, 2012] Bergstra, J. and Bengio, Y. (2012). Random search for hyper-parameter optimization. *J. Mach. Learn. Res.*, 13:281--305.

[Bertsekas, 2012] Bertsekas, D. (2012). *Dynamic programming and optimal control: Volume I*, volume 4. Athena scientific.

[Biewald, 2020] Biewald, L. (2020). Experiment tracking with weights and biases. Software available from wandb.com.

[Brockman et al., 2016] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). Openai gym. *CoRR*, abs/1606.01540.

[Deb et al., 2000] Deb, K., Agrawal, S., Pratap, A., and Meyarivan, T. (2000). A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: Nsga-ii. In Schoenauer, M., Deb, K., Rudolph, G., Yao, X., Lutton, E., Merelo, J. J., and Schwefel, H.-P., editors, *Parallel Problem Solving from Nature PPSN VI*, pages 849--858, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Felten et al., 2023] Felten, F., Alegre, L. N., Nowé, A., Bazzan, A. L. C., Talbi, E. G., Danoy, G., and Silva, B. C. d. (2023). A toolkit for reliable benchmarking and research in multi-objective reinforcement learning. In *Proceedings of the 37th Conference on Neural Information Processing Systems (NeurIPS 2023)*.

[Gaon and Brafman, 2019] Gaon, M. and Brafman, R. I. (2019). Reinforcement learning with non-markovian rewards.

[Glorot and Bengio, 2010] Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In Teh, Y. W. and Titterington, M., editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249--256, Chia Laguna Resort, Sardinia, Italy. PMLR.

[Hahnloser et al., 2000] Hahnloser, R. H., Sarpeshkar, R., Mahowald, M. A., Douglas, R. J., and Seung, H. S. (2000). Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. *Nature*, 405(6789):947--951.

[Hayes et al., 2021] Hayes, C. F., Radulescu, R., Bargiacchi, E., Källström, J., Macfarlane, M., Reymond, M., Verstraeten, T., Zintgraf, L. M., Dazeley, R., Heintz, F., Howley, E., Irissappane, A. A., Mannion, P., Nowé, A., de Oliveira Ramos, G., Restelli, M., Vamplew, P., and Roijers, D. M. (2021). A practical guide to multi-objective reinforcement learning and planning. *CoRR*, abs/2103.09568.

[He et al., 2015] He, K., Zhang, X., Ren, S., and Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification.

[Ioffe and Szegedy, 2015] Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift.

[Kaelbling et al., 1996] Kaelbling, L. P., Littman, M. L., and Moore, A. W. (1996). Reinforcement learning: A survey. *J. Artif. Int. Res.*, 4(1):237--285.

[Kingma and Ba, 2017] Kingma, D. P. and Ba, J. (2017). Adam: A method for stochastic optimization.

[Kumar et al., 2019] Kumar, A., Peng, X. B., and Levine, S. (2019). Reward-conditioned policies. *CoRR*, abs/1912.13465.

[Liu et al., 2015] Liu, C., Xu, X., and Hu, D. (2015). Multiobjective reinforcement learning: A comprehensive overview. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 45(3):385--398.

[Lu et al., 2023] Lu, H., Herman, D., and Yu, Y. (2023). Multi-objective reinforcement learning: Convexity, stationarity and pareto optimality. In *The Eleventh International Conference on Learning Representations*.

[Mnih et al., 2013] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. A. (2013). Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602.

[Mnih et al., 2015] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529--533.

[Monahan, 1982] Monahan, G. E. (1982). State of the art---a survey of partially observable markov decision processes: Theory, models, and algorithms. *Management Science*, 28:1--16.

[Moore, 1990] Moore, A. W. (1990). Efficient memory-based learning for robot control. Technical report, University of Cambridge.

[Nian et al., 2020] Nian, X., Irissappane, A. A., and Roijers, D. (2020). Dcrac: Deep conditioned recurrent actor-critic for multi-objective partially observable environments. In *Proceedings of the 19th international conference on autonomous agents and multiagent systems*, pages 931--938.

[Pascanu et al., 2013] Pascanu, R., Mikolov, T., and Bengio, Y. (2013). On the difficulty of training recurrent neural networks.

[Rădulescu et al., 2019] Rădulescu, R., Mannion, P., Roijers, D. M., and Nowé, A. (2019). Multi-objective multi-agent decision making: a utility-based analysis and survey. *Autonomous Agents and Multi-Agent Systems*, 34(1).

[Reymond et al., 2022] Reymond, M., Bargiacchi, E., and Nowé, A. (2022). Pareto conditioned networks.

[Roijers et al., 2015] Roijers, D., Whiteson, S., Vamplew, P., and Dazeley, R. (2015). Why multi-objective reinforcement learning?

[Roijers et al., 2013] Roijers, D. M., Vamplew, P., Whiteson, S., and Dazeley, R. (2013). A survey of multi-objective sequential decision-making. *Journal of Artificial Intelligence Research*, 48:67--113.

[Rumelhart et al., 1986] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323(6088):533--536.

[Russell and Norvig, 2010] Russell, S. J. and Norvig, P. (2010). *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3 edition.

[Silver et al., 2017] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T. P., Simonyan, K., and Hassabis, D. (2017). Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815.

[Srivastava et al., 2014] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929--1958.

[Sutton and Barto, 2018] Sutton, R. S. and Barto, A. G. (2018). *Reinforcement Learning: An Introduction*. The MIT Press, second edition.

[Tajmajer, 2018] Tajmajer, T. (2018). Modular multi-objective deep reinforcement learning with decision values. In *Proceedings of the 2018 Federated Conference on Computer Science and Information Systems*. IEEE.

[Takayama and Arai, 2022] Takayama, N. and Arai, S. (2022). Multi-objective deep inverse reinforcement learning for weight estimation of objectives. *Artificial Life and Robotics*, 27(3):594--602.

[Towers et al., 2023] Towers, M., Terry, J. K., Kwiatkowski, A., Balis, J. U., Cola, G. d., Deleu, T., Goulão, M., Kallinteris, A., KG, A., Krimmel, M., Perez-Vicente, R., Pierré, A., Schulhoff, S., Tai, J. J., Shen, A. T. J., and Younis, O. G. (2023). Gymnasium.

[Vamplew et al., 2011] Vamplew, P., Dazeley, R., Berry, A., Issabekov, R., and Dekker, E. (2011). Empirical evaluation methods for multiobjective reinforcement learning algorithms. *Machine Learning*, 84(1):51--80.

[Van Moffaert et al., 2013] Van Moffaert, K., Drugan, M. M., and Nowé, A. (2013). Hypervolume-based multi-objective reinforcement learning. In *Evolutionary Multi-Criterion Optimization: 7th International Conference, EMO 2013, Sheffield, UK, March 19-22, 2013. Proceedings 7*, pages 352--366. Springer.

[Wang, 2003] Wang, S.-C. (2003). *Artificial Neural Network*, pages 81--100. Springer US, Boston, MA.

[Whiteson et al., 2009] Whiteson, S., Tanner, B., Taylor, M. E., and Stone, P. (2009). Generalized domains for empirical evaluations in reinforcement learning. In *ICML Workshop on Evaluation Methods for Machine Learning*.

[Xu et al., 2020a] Xu, J., Tian, Y., Ma, P., Rus, D., Sueda, S., and Matusik, W. (2020a). Prediction-guided multi-objective reinforcement learning for continuous robot control. In *Proceedings of the 37th International Conference on Machine Learning*.

[Xu et al., 2020b] Xu, J., Tian, Y., Ma, P., Rus, D., Sueda, S., and Matusik, W. (2020b). Prediction-guided multi-objective reinforcement learning for continuous robot control. In *Proceedings of the 37th International Conference on Machine Learning*.

[Zintgraf et al., 2015] Zintgraf, L., Kanters, T., Roijers, D., Oliehoek, F., and Beau, P. (2015). Quality assessment of morl algorithms: A utility-based approach.

[Zitzler et al., 2008] Zitzler, E., Knowles, J., and Thiele, L. (2008). Quality assessment of pareto set approximations. In *Multiobjective Optimization*, pages 373--404. Springer.

[Zitzler and Thiele, 1999] Zitzler, E. and Thiele, L. (1999). Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach. *IEEE Transactions on Evolutionary Computation*, 3(4):257--271.

Appendices

Appendix contains two parts: A part contains crucial parts of the PCN method that were modified to allow it to work in continuous action spaces. B part contains parameters used during hyperparameter search.

Implementation of the CCDST environment, code for generating figures 3 and 4 are available at https://github.com/vaidas-sl/ms-code-dump. Commit with the changes for PCN with continuous actions is available at
https://github.com/vaidas-sl/morl-baselines/commit/1716de2821717b107703bbb9c8a81d23e0c2d4a5. Results of all the experiments can be downloaded from https://wandb.ai/sl-vaidas/MORL-Baselines.

A Main parts of PCN code which allow continuous actions

A.1 Noise ratio decay

```
<...>
    noise_ratio = max(1.0 - (self.global_step /
     ↪  self.EXPLORATION_CUTOFF), self.MIN_EXPLORATION_RATIO)
<...>
```

A.2 Tested exploration methods

```
<...>
noise = th.rand_like(prediction) * 2 - 1
prediction = prediction * (1 - noise_ratio) + noise * noise_ratio
action = prediction.detach().cpu().numpy()[0]
<...>
```

```
<...>
noise = th.rand_like(prediction) * 2 - 1
prediction = prediction + noise * noise_ratio
action = prediction.detach().cpu().numpy()[0]
<...>
```

```
<...>
noise = np.random.randn() * noise_ratio
prediction = prediction + noise
action = prediction.detach().cpu().numpy()[0]
<...>
```

```
if random.random() < noise_ratio:
 return self.env.action_space.sample()
else:
  prediction = prediction.detach().cpu().numpy()[0]
```

A.3 Forward method of PCN

```
<...>
def forward(self, state, desired_return, desired_horizon):
    c = th.cat((desired_return, desired_horizon), dim=-1)
    c = c * self.scaling_factor
    s = self.s_emb(state.float())
    c = self.c_emb(c)
    prediction = self.fc(s * c)
    return prediction
<...>
```

B Hyperparameter sets

Parameter	Values
Learning Rate	0.001, 0.01
Num_er_episodes	6, 20
Num_step_episodes	10, 100
Num_model_updates	50, 100, 200
Hidden_dim	64, 256

Table 4. PCN CCDST Parameters

Parameter	Values
Learning Rate	0.001, 0.01
Batch Size	2000, 4000
Eval_iterations	6, 8
Warmup_iterations	6, 8

Table 5. PGMORL CCDST Parameters

Parameter	Values
Learning Rate	0.001, 0.003
Batch Size	128, 64
Buffer Size	100000, 1000000
Net_arch	$[64, 64]$, $[256, 256]$

Table 6. CAPOQL CCDST Parameters

Parameter	Values
Learning Rate	0.001, 0.004
Batch Size	256, 64
Hidden_dim	64, 256
Max_buffer_size	600, 6000
Num_step_episodes	10, 100
Num_model_updates	50, 200

Table 7. PCN Lunar Lander Parameters

Parameter	Values
Learning Rate	0.001, 0.0003
Batch Size	128, 64
Learning_starts	100, 1000
Net_arch	$[64, 64]$, $[256, 256]$

Table 8. CAPOQL Lunar Lander Parameters

Parameter	Values
Learning Rate	0.01, 0.001, 0.004
Exploration Cutoff	100000, 300000, 1000000
Noise	0.1, 0.3, 0
Num_er_episodes	20, 100
Batch Size	256, 64
Hidden_dim	64, 256
Max_buffer_size	600, 6000
Num_step_episodes	10, 100
Num_model_updates	50, 200

Table 9. PCN Mountain Car Parameters

Parameter	Values
Batch Size	128, 64
Learning_starts	1000, 10000
Net_arch	$[64, 64]$, $[128, 128]$, $[256, 256]$
Num_q_nets	2, 4

Table 10. CAPOQL Mountain Car Parameters

Parameter	Values
Learning Rate	0.0003, 0.001
Batch Size	2000, 4000
Eval_iterations	6, 20
Num_weight_candidates	7, 10
Net_arch	$[64, 64]$, $[128, 128]$
Warmup_iterations	20, 80

Table 11. PGMORL Mountain Car Parameters