TUM School of Computation, Information and Technology
Technische Universität München

Policy Regularization for Model-Based Offline Reinforcement Learning

Phillip A. Swazinna

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr.rer.nat)

genehmigten Dissertation.

Die Dissertation wurde am 22.02.2023 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 21.06.2023 angenommen.

Acknowledgements

I would like to thank everyone who has made my PhD adventure possible through their continuous and unwaivering support.

Firstly, I would like to express my gratitude to my supervisor, Prof. Dr. Thomas Runkler, for his excellent supervision during my PhD journey, including encouraging me to find worthwhile ideas and areas of interest, shaping the focus of the resulting research, as well as providing guidance on how to best publish the findings. Thomas has been an exceptionally involved supervisor, who I could always rely on and was quickly available to help whenever difficulties arose. He has also played a vital role in connecting all the important dots to create a coherent story and present the research findings in this monolithic theses. I am furthermore grateful to my advisor, Dr. Steffen Udluft, for his magnificent mentorship. Steffen has valuably contributed towards shaping my research focus, and we have held countless creative meetings discussing new, interesting (and sometimes crazy) ideas filling whiteboard after whiteboard. I thank him for his dedication to creating purposeful research, for sharpening the storylines and claims in our publications until satisfaction, for many hours of proofreading, as well as for broadening my mind in many philosophical discussions about non-research topics.
I'm also thankful for my other colleagues at Siemens Learning Systems, who have helped to create a supportive and stimulating research environment. I thank the former PhD students at our research group, Dr. Daniel Hein, Dr. Markus Kaiser, and Dr. Stephan Depeweg, for helping me find my way into reinforcement learning research especially in the beginning, for their passion for good research, and for many interesting talks in the coffee kitchen as well as at the PhD Stammtisch. I thank Daniel also for his additional advising and efforts on our joint publications, as well as for his excellent taste in music. Further, I thank the head of our research group, Volkmar Sterzing, for encouraging us PhD students to pursue ideas that interest us, enabling us to focus on our research, and for providing everything necessary from compute resources over contacts until travel budgets.

Finally, I express my deep appreciation to my parents Ute & Andreas, as well as my brother Leon, for always being there for me, believing in me, and encouraging me to pursue my interests and to keep going also during difficult times. I thank my grandparents Inge, Lu, Ulrich, and Günter, for supporting me and my brother throughout our educational and academic endeavours and for their infinite kindness and wisdom. Last but not least I thank Anna for her continuous support throughout all the highs and lows during my dissertation, for her understanding of PhD life, many discussions about academia as a whole, and her love.

Abstract

In this thesis, three novel algorithms for offline reinforcement learning, as well as one method to judge the suitability of a dataset for offline RL are presented. The algorithms are evaluated on challenging, high-dimensional, continuous, partially observable as well as stochastic environments and it is shown that they are able to produce well performing policies from static datasets reliably, using model-based return estimation paired with newly proposed methods for policy regularization. The final algorithm even exposes a control parameter in order to allow users to tune the behavior of the policy to their personal preferences based on the desired proximity to the dataset generating policy. The method thus naturally facilitates trust and allows users to find a better trade-off between familiarity and estimated performance of the solution, ultimately also resulting in better performance.

While deep reinforcement learning has seen a surge in popularity over the past decade, it has not transformed the world in the way that many researchers had expected. This falling short of expectations is largely due to the fact that classical reinforcement learning algorithms need to interact with the environment they are trying to find a policy for. While this is not a problem for entirely virtual problems like playing Atari video games, it is an issue for real world (physical) systems, since learning directly on them is usually infeasible due to cost and safety concerns, and because accurate simulations as a replacement are often not available. Thus there exists a natural need for algorithms that can train policies not by interacting with the environment itself, but instead simply taking as input all the interaction data that has in the past been collected during ordinary operation.

The first two proposed algorithms, MOOSE and WSBC, aim to make policy learning possible from static pre-collected datasets by employing learned transition models as a replacement for the environment, and regularizing the policy so that it remains in a region of the state-action space that is well represented in the dataset, since otherwise the transition model predictions will not be accurate and the policy will upon deployment perform much worse than estimated by the dynamics models. MOOSE (MOdel-based Offline policy Search with Ensembles) performs the regularization of the policy in action space, by training an extra variational autoencoder model of the dataset generating policy and penalizing the target policy for actions that cannot be well reconstructed under it. WSBC (Weight Space Behavior Constraining) on the other hand constrains the policy directly in its parameter space, by training an MLP model of the generating policy from the data, and then restricting the target policy to remain close to it in the neural weight space. Experiments on the industrial benchmark and MuJoCo datasets show that both approaches are valid and produce well performing policies on a variety of tasks, and thus effectively enable policy learning without the need for direct environment interaction. While the action space penalty appears to have advantages in the MuJoCo tasks, WSBC generally performs better in the industrial benchmark domain, which is likely due to the gradient-free policy search paired with the lower complexity policies required for these tasks. Finally, the third algorithm, LION (Learning in Interactive Offline eNvironments), aims to integrate the user to find even better solutions: Since offline RL algorithms cannot tune their most important hyperparameter - how strictly to regularize the policy - due to the lack of permitted environment interaction, such algorithms practically always perform below their theoretically best option. To circumvent this problem, LION trains policies that condition on a desired proximity to the data generating policy, making it possible for a user to test out and find well performing trade-offs at runtime, while at the same time regaining control over the system, increasing trust in the solution.

Comparing the proposed algorithms with a variety of prior and concurrently proposed model-free as well as model-based offline RL algorithms, such as BCQ, BEAR, BRAC, TD3+BC, CQL, MOPO & MOReL, this thesis provides a comprehensive overview on the performance of the methods on a variety of benchmark datasets. The tasks and datasets considered in this thesis cover low and high exploration settings, very bad as well as close to expert generating policies, fully as well as partially observable and stochastic as well as deterministic environments. Other (real world) tasks and datasets may however still exhibit different characteristics and choosing the correct methods for the task at hand thus must still be handled with great care. We find however that generally, model-based offline RL, especially when coupled with proximity conditioned policies, is a very promising path to more RL deployments in real world systems in the future.

Zusammenfassung

In dieser Arbeit werden drei neue Algorithmen für Offline-Reinforcement Learning sowie eine Methode zur Beurteilung der Eignung eines Datensatzes für Offline RL vorgestellt. Die Algorithmen werden in anspruchsvollen, hochdimensionalen, kontinuierlichen, teilweise beobachtbaren sowie stochastischen Umgebungen evaluiert und es wird gezeigt, dass sie in der Lage sind, zuverlässig leistungsstarke Policies aus statischen Datensätzen zu erzeugen, indem sie modellbasierte Return Schätzungen mit neu vorgeschlagenen Methoden zur Regularisierung der Policy kombinieren. Der letzte Algorithmus enthält sogar einen Steuerparameter, mit dem Benutzer das Verhalten der Policy - basierend auf der Nähe zur generierenden Policy - an ihre persönlichen Präferenzen anpassen können. Die Methode schafft so auf natürliche Art und Weise Vertrauen und erlaubt es eine bessere Abwägung zwischen Vertrautheit und geschätzter Leistung der Policy zu erzielen, was sich am Ende auch in verbesserter Performanz auszahlt.

Obwohl das tiefe Reinforcement Learning im letzten Jahrzehnt einen Boom erlebt hat, hat es die Welt nicht in dem Maße verändert, wie viele Forscher erwartet hatten. Dies liegt hauptsächlich daran, dass klassische Reinforcement-Lernalgorithmen mit der Umgebung interagieren müssen, für die sie eine Policy suchen. Während dies bei komplett virtuellen Problemen wie dem Spielen von Atari-Videospielen kein Problem darstellt, tut es dies bei realen (physischen) Systemen umso mehr, da das Lernen direkt an ihnen aufgrund von Kosten und Sicherheitsbedenken oft nicht möglich ist und genaue Simulationen als Ersatz meist nicht verfügbar sind. Daher besteht ein natürlicher Bedarf an Algorithmen, die Policies nicht durch Interaktion mit der echten Umgebung trainieren, sondern als Eingabe einfach alle Interaktionsdaten verwenden, die in der Vergangenheit während des normalen Betriebs gesammelt wurden.

Die ersten beiden vorgeschlagenen Algorithmen, MOOSE und WSBC, sollen das Erlernen von Policies aus statischen, vorab gesammelten Datensätzen durch den Einsatz gelernter Zustandsübergangsmodelle, als Ersatz für die Umgebung, kombiniert mit einer Regularisierung der Policy möglich machen. Letztere soll dafür sorgen, dass sich die Policies in einem Bereich des Zustands-Aktionenraums aufhalten, der durch den Datensatz abgedeckt ist, da ansonsten die Vorhersagen der Transitionsmodelle nicht genau genug sind und die Policy bei der Auswertung in der echten Umgebung schlechter abschneiden wird als durch die Dynamikmodelle geschätzt. MOOSE (MOdel-based Offline policy Search with Ensembles) führt die Regularisierung der Policy im Aktionsraum durch, indem es ein zusätzliches Variational Autoencoder-Modell der Datensatzerzeugungspolicy trainiert und die Zielpolicy für Aktionen bestraft, die unter der Generierenden nicht gut rekonstruiert werden können. WSBC (Weight Space Behavior Constraining) beschränkt die Policy hingegen direkt in ihrem Parameterraum, indem es ein MLP-Modell der generierenden Policy aus den Daten trainiert und die Zielpolicy dann so einschränkt, dass sie im neuronalen Gewichtsraum nahe daran bleibt. Experimente mit den industriellen Benchmark- und MuJoCo-Datensätzen zeigen, dass beide Ansätze gültig sind und für eine Vielzahl von Aufgaben leistungsstarke Policies erzeugen und somit erfolgreich das Erlernen von Policies ohne die Notwendigkeit direkter Umgebungsinteraktion ermöglichen. Während die Aktionsraumbestrafung in MuJoCo Aufgaben Vorteile zu haben scheint, schneidet WSBC im Allgemeinen in dem industriellen Benchmark-Bereich besser ab, was wahrscheinlich auf die gradientenfreie Policysuche gepaart mit der niedrigeren Komplexität der für diese Aufgaben erforderlichen Policies zurückzuführen ist. Schließlich zielt der dritte Algorithmus LION (Learning in Interactive Offline eNvironments) darauf ab, den Benutzer zu integrieren, um noch bessere Lösungen zu finden: Da Offline RL-Algorithmen aufgrund des Fehlens von erlaubten Umgebungsinteraktionen ihren wichtigsten Hyperparameter - wie strikt die Policy zu regulieren ist - nicht einstellen können, schneiden solche Algorithmen praktisch immer unter ihrer theoretisch besten Möglichkeit ab. Um dieses Problem zu umgehen, trainiert LION bedingte Policies, die bedingt der gewünschten Nähe zur Datensatzerzeugungspolicy den Return optimieren, was es dem Benutzer ermöglicht, zur Laufzeit gut performende Werte für das Regularisierungsgewicht zu finden und gleichzeitig Kontrolle über das System zurückzubekommen, was das Vertrauen in die Lösung verbessert.

Indem die vorgeschlagenen Algorithmen mit einer Vielzahl von vorherigen und gleichzeitig vorgeschlagenen modellfreien sowie modellbasierten Offline-RL-Algorithmen, wie BCQ, BEAR, BRAC, TD3+BC, CQL, MOPO und MOReL, verglichen werden, bietet diese Arbeit einen umfassenden Überblick über die Leistung der Methoden auf einer Vielzahl von Benchmark-Datensätzen. Die in dieser Arbeit betrachteten Aufgaben und Datensätze umfassen niedrige und hohe Explorationsszenarien, sehr schlechte sowie quasi Expertenpolicies als generierende Policies, vollständig sowie teilweise beobachtbare und stochastische sowie deterministische Umgebungen. Trotzdem können andere (reale) Aufgaben und Datensätze abweichende Eigenschaften aufweisen, weshalb die Auswahl der richtigen Methode für die konkrete Aufgabe weiterhin mit großer Sorgfalt behandelt werden muss. Jedoch finden wir, dass modellbasiertes Offline-RL, insbesondere in Kombination mit bedingten Policies, ein sehr vielversprechender Weg für zukünftige RL-Einsätze ist.

Contents

Acronyms

BC	Behavior Cloning.
BCQ	Batch Constrained Q-Learning.
BEAR	Bootstrapping Error Accumulation Reduction.
BRAC	Behavior Regularized Actor Critic.
CMA	Covariance Matrix Adaptation.
COI	Combined Offline Indicator.
CQL	Conservative Q-Learning.
DDPG	Deep Deterministic Policy Gradient.
EAS	Estimated Action Stochasticity.
ERI	expected return improvement.
IB	Industrial Benchmark.
LION	Learning in Interactive Offline Environments.
ML	Machine Learning.
MLP	Multi Layer Perceptron.
MOOSE	Model-based Offline Policy Search with Ensembles.
MOPO	Model-based Offline Policy Optimization.
MOReL	Model-based Offline Reinforcement Learning.
PSO	Particle Swarm Optimization.
RL	Reinforcement Learning.
RNN	Recurrent Neural Network.
TD3	Twin Delayed Deep Deterministic Policy Gradient.
VAE	Variational Autoencoder.
WSBC	Weight Space Behavior Constraining.

1 Introduction

In reinforcement learning (RL), the goal of an agent is to find a control strategy that maximizes a predefined reward function, only by interacting with the environment. Such a control strategy is commonly referred to as a policy, and it maps perceived states of the environment to actions that the agent will take. Agents need to carefully balance between exploring their environment and exploiting the knowledge they have gained about it in order to maximize long term rewards - too little exploration results in not enough knowledge, too much exploration results in behavior that can never fully unleash the potential contained in the collected information [1].
Recently, RL algorithms have had tremendous success in solving tasks better than any human could, such as Go, playing Atari video games, performing robotic locomotion, and other continuous control tasks [2, 3, 4, 5, 6, 7]. These approaches usually rely on a combination of classic reinforcement learning with function approximation by neural networks, enabling to tackle much larger (possibly infinitely large) state and action spaces than before. Many of them belong to the category of model-free, actor-critic methods, meaning they do not contain a model of the environment, but instead train a value function to use as a critic and judge the performance of the actor / policy. It has however been found [8], that these methods are unable to learn from data that has not been collected under the trained, or close to the trained policy, which appears to be a large road blocker on the way to using RL methods in many real world, industrial applications.

While the achievements on the virtual environments mentioned above are tremendous, it is often not the case in industrial applications that an agent may freely interact with its environment, since that would include many downsides for the operation of the system: At least in the beginning, the achieved performance of the agent needs to be assumed very low compared to the one observed during ordinary operation, which implies a cost of under performing. On top of that, in mechanical systems there always exists the risk when performing exploratory actions during the initial phase of an algorithm that they may damage equipment (e.g. abruptly changing mechanical steerings). Furthermore, in some environments there exists even danger for human lives if exploratory actions are taken (consider e.g. self-driving vehicles). It can thus usually be considered that learning in the real system, directly, in an online fashion is at least impractical if the environment is not virtual. Another reason for this is the sheer amount of data that many model-free online algorithms, such as DDPG need in order to solve a task (hundreds of millions of interactions are not uncommon).

There is thus a need for algorithms that are capable to learn policies offline, i.e. without the possibility to collect any on-policy interactions and only with very limited data provided in the form of a static, pre-collected dataset, where algorithms cannot assume control over the collection process. While some batch RL algorithms have been proposed over the past years [9, 10, 11, 12, 13, 14], most of them assume that the dataset sufficiently covers the state-action space of the problem well (i.e. by collecting datasets via random actions), which is in reality an unlikely assumption: In many areas, large datasets have been passively collected during ordinary operation, meaning that often, similar things have happened and that there definitely exist unrepresented areas. Very recently, approaches have thus been proposed to mitigate this problem: Batch Constrained Q-Learning (BCQ)[8], allows to only perform actions that have been reconstructed from a model of the policy that generated the dataset, tweaking them for performance only slightly in order to not deviate too far from the data coverage. While the proposed method constitutes a successful first step on the path to offline RL methods, there are arguably things to improve: BCQ is a model-free method, and as such considered to have inferior data efficiency [15, 16, 17] compared to model-based ones. Since offline RL is a problem where data is naturally scarce due to the inability to interact with the environment, learned transition models are likely to improve the situation further. Furthermore, model-based methods have been found to be more stable during training [18], which is more important in offline RL since it is not possible to evaluate the performance of intermediate policy candidates.

Throughout this thesis, we will consider a wind turbine as a running example of the system / environment to optimize in. To illustrate the above point again: If the wind turbine would generate during early phases of learning only 75% of its normal electricity output, that would cost the operator money immediately, which is undesired and one of the reasons why offline RL is needed in practice.

While learning directly from offline datasets instead of online in the real system is one of the most important aspects to bring reinforcement learning to the real world, it is not the only issue remaining to do so. Since real world systems are usually directly or indirectly controlled by users before an RL based policy can come into play, there is a need to address users' wishes about the newly derived policy as well, since it otherwise will likely not make its way into deployment. Only few prior works address the issue that users might want to influence the policy's behavior if they dislike it or simply find it unfamiliar: E.g. [19, 20] experiment with set valued policies in a medical treatment RL setting, where clinicians can make a choice among the actions that were pre-selected by the policy. For continuous control, a few works, such as [21, 22, 23] have derived policies that remain adaptable in their objectives, i.e. users can later during deployment specify a different reward function than during training. However, works that establish the familiarity of the solution to the user as a dimension the user would like to influence are to the best of our knowledge missing.

In this thesis, we will however start with an issue that arises even before offline RL really starts: While the goal is clear once a task and dataset is given, in practice situations arise where practitioners have the choice of working on a variety of tasks and datasets, but the necessary resources to work on all of them are simply not available. To this end, in Chapter 4, we derive simple and computationally very cheap indicators ERI (expected return improvement), EAS (Estimated Action Stochasticity), and COI (Combined Offline Indicator), in order to assess the quality of the provided datasets in terms of suitability for offline reinforcement learning. Of course, these indicators cannot always accurately predict the success of an offline RL algorithm on the given dataset, but we show empirically that if we need to select a subset of the given datasets that suits the amount of resources we have available, the provided selection by the indicators is very favourable.
Once the dataset to work with has been selected, the goal is to derive a policy that performs as well as possible on the true system dynamics. In Chapter 5, we thus develop two model-based algorithms, MOOSE (MOdel-based Offline policy Search with Ensembles) & WSBC (Weight Space Behavior Constraining), to improve the data efficiency & robustness over prior model-free and hybrid offline approaches. We show that they reliably outperform prior algorithms over a range of relevant tasks and datasets, and pay close attention to the robustness of the algorithms' performance: Since future access to the system may be denied if we underperform the prior controller, we analyze the performance more from a worst case perspective instead of considering the average case. In this context we examine how policy regularization can effectively be performed in a model-based learning setup, by performing the regularization either in the action space (MOOSE), or directly in the policy weight space (WSBC).
Finally, in Chapter 6 we address issues relevant to practitioners which will ultimately use the trained policies: Since in offline RL contexts, we cannot evaluate the policy prior to deployment in order to verify, its performance and whether the policy adheres to the users' wishes in terms of familiarity, we develop LION (Learning in Interactive Offline eNvironments), an algorithm to train policies that condition on the proximity to the generating policy, enabling operators to influence the behavior of the final policy at runtime, so that behavior that is not desired can immediately be corrected.

Comparing the derived algorithms MOOSE, WSBC, and LION to the existing prior works BCQ, BEAR, BRAC, CQL (model-free), MOPO, and MOReL (hybrid) in the offline setting, as well as to DDPG (off-policy, model-free) on nineteen datasets from four different tasks (industrial benchmark as well as MuJoCo Swimmer, Hopper, and Walker2D), we provide a comprehensive overview of the performance as well as robustness and utility that model-based RL together with the right policy regularization can accomplish. We conclude that the proposed algorithms can dramatically improve practical applicability and deployments in the real world.

1.1 Challenges for Reinforcement Learning Deployments in Industrial Contexts

In [24], the authors identify nine major challenges that constitute the main roadblocks on the path to more productive use of reinforcement learning technology in real-world problems. The very first one they establish is "Training off-line from the fixed logs of an external generating policy", i.e. what is now known as offline RL. The authors explain that the necessity for offline learning stems from the fact that simulations are for many real-world applications simply not available since they are either too complex or too large / varied and thus too expensive to be practically built. While superhuman performance is already achievable on cheaply simulatable tasks such as Atari video games, the corresponding training techniques cannot simply be used as is for offline learning, and new techniques need to be found to enable learning from pre-collected datasets by some unknown generating policy. Interestingly, the authors' definition of the offline challenge is more closely related to that of deployment efficient RL [25] than to the pure offline RL setting, since they assume a scenario in which they can iteratively replace the previous policy by learning from the batch of data that was generated by it. However, they also identify the "warm start" phase as the most important challenge, meaning that the first policy π_0 that is learned on the first batch of data needs to achieve a minimum return, since otherwise the access to the environment might in the future be denied due to lack of trust in the method. The authors further state that one of the most pressing issues in this context is the evaluation of policy performances without access to the real environment, since one of the most fundamental differences to regular supervised learning is that we assume the data distribution to be different during training and evaluation: While value functions or transition models are trained under the generating policy, we explicitly want the target policy to do something different (hopefully better) than the generating policy, resulting in a by design different data distribution. The resulting problem has been termed "distribution shift", and is recognized as one of the fundamental problems in offline RL [26].

Additionally, the authors of [24] identify high-dimensional states and actions, partially observable states, as well as delayed reward signals as key road blockers for practical deployments. High-dimensional state and action spaces can be a problem since each dimension exponentially increases the amount of data needed to cover the corresponding state-action space. Especially high-dimensional actions can be problematic in reality since prior existing controllers often do not sufficiently cover the options available. While we do not explicitly include measures to deal with this issue, the experiments we make are conducted on benchmarks that can be considered high-dimensional.
In order to deal with the partial observability of the industrial benchmark, we employ the two most common counter measures: In MOOSE, we provide the policy as well as the transition model with a history of the past 30 time steps, which should be enough to "recover" the hidden state of the system. In WSBC and LION, we train recurrent neural networks that directly learn to maintain a hidden state as a sequence of single-timestep

input states is provided to them. In our experiments, the same measures appear to be sufficient to also deal with the problem of delayed rewards.

Finally, [24] propose "System operators who desire explainable policies and actions" as a key issue that is currently still preventing more real-world RL deployments. While we generally agree, we believe this point is not complete: Since real world systems are "owned and operated by humans", we need to adhere to requirements they might have about the policy, even if these requirements have nothing to do with its performance on the real system. One of these requirements could be explainability: [18] develop ways to find intuitively understandable policies in the form of fuzzy rules or simple algebraic equations, and [27] define their policies in a domain specific programming language. However, explainability is not the only requirement users likely have in practice: Instead of only understanding the policy, they will likely want to influence its behavior if it doesn't act appropriately. This desire for human control and oversight has formally been recognized in the EU's guidelines for trustworthy AI [28]. Researchers also argue that in order to be useful, future AI systems need to adopt a more user-centric approach and behave more like an intelligent tool that users can employ to achieve their goals [29, 30]. That way, the best of both worlds would be combined, yielding both "high levels of human control and high levels of automation", which we aim to do with LION.

[24] also recognize "safety constraints that should never or at least rarely be violated" as a major remaining challenge for RL deployments in real-world systems. While offline learning is certainly related to safety, especially in the training phase (no exploration means no random actions that are potentially unsafe), offline policy learning does not directly solve the issue of hard safety constraints as they are given e.g. in autonomous vehicles. Augmenting (offline) reinforcement learning methods by additional components in order to provide probabilistic safety guarantees or even provably safe controllers, such as explored in [31, 14, 32, 33, 34], is needed to make RL feasible in safety critical domains. Since the focus of this thesis lies on making offline RL possible in general, constraint satisfying efforts are not considered.

1.2 Contributions

In this thesis, three novel algorithms to derive policies from offline datasets, as well as one method to evaluate the suitability of datasets for offline reinforcement learning are proposed:

- The features ERI (Estimated Return Improvement), EAS (Estimated Action Stochasticity), and their combination COI (Combined Offline Indicator) to assess whether a dataset is promising for offline RL [A]
- The algorithms MOOSE (MOdel-based Offline policy Search with Ensembles) & WSBC (Weight Space Behavior Constraining) to derive fixed policies from a pre-collected static dataset, with better performance than prior offline algorithms [B,C,D]

- The LION (Learning in Interactive Offline eNvironments) algorithm to derive adaptive policies that condition on the proximity to the generating policy from a pre-collected static dataset, which enables users to tune the desired proximity-performance trade-off at runtime [E,F]

When multiple datasets are available, the features derived in [A] assess dataset quality in terms of potential return improvement over the generating policy when offline RL is applied. This helps to decide which datasets to work on and in which order, effectively enabling practitioners to save resources by identifying the best datasets to work on.

The offline RL algorithms MOOSE & WSBC can then be used to derive policies from the selected datasets. Compared to a set of prior offline RL algorithms, these two achieve better returns on most of the considered datasets. They also do so more reliably, making them better candidates for offline policy search, and thereby improving operators chances of deploying reinforcement learning based controllers in practical applications.

Since in offline RL, the policies cannot be tested until deployment, it is hard to tune an algorithm's hyperparameters. LION policies remain adaptable in the distance to the generating policy after deployment, so that users may tune this arguably most impactful hyperparameter after training has conceded. This enables them to tune the policy's trade-off between familiarity and return optimization, yielding even better performance than MOOSE & WSBC, and improving the trustworthiness compared to prior works.

[A] Phillip Swazinna, Steffen Udluft, and Thomas Runkler. Measuring data quality for dataset selection in offline reinforcement learning. In *2021 IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE, 2021

[B] Phillip Swazinna, Steffen Udluft, and Thomas Runkler. Overcoming model bias for robust offline deep reinforcement learning. *Engineering Applications of Artificial Intelligence*, 104:104366, 2021

[C] Phillip Swazinna, Steffen Udluft, Daniel Hein, and Thomas Runkler. Behavior constraining in weight space for offline reinforcement learning. *European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning*, 2021

[D] Phillip Swazinna, Steffen Udluft, Daniel Hein, and Thomas Runkler. Comparing model-free and model-based algorithms for offline reinforcement learning. *IFAC Conference on Intelligent Control and Automation Sciences*, 2022

[E] Phillip Swazinna, Steffen Udluft, and Thomas Runkler. Towards user-interactive offline reinforcement learning. *NeurIPS 3rd Offline RL Workshop: Offline RL as a "Launchpad"*, 2022

[F] Phillip Swazinna, Steffen Udluft, and Thomas Runkler. User-interactive offline reinforcement learning. *Accepted at ICLR 2023 - 11th International Conference on Learning Representations*, 2023

Some works presented in this thesis are furthermore protected by the following patent:

- Phillip Swazinna, Steffen Udluft, Thomas Runkler: Verfahren zum Konfigurieren eines Steuerungsagenten für ein Technisches System sowie Steuereinrichtung. European Patent EP3940596A1. 2022-01-19. Siemens AG.

1.3 Thesis Outline

The contents of this thesis are structured as follows: At first, Chapter 2 introduces the core concepts of reinforcement learning, presents the differences between model-free and model-based approaches, and introduces the general problem setting of offline reinforcement learning. Furthermore, Chapter 2 shows examples where prior algorithms not specifically designed for the offline setting fail to perform and thus motivates the need for new algorithms. Finally, it presents core techniques, such as particle swarm optimization and variational autoencoders, which will be needed later in Chapter 5, and introduces the offline RL algorithms that have been developed concurrently to this thesis and are used as baselines in the experiments. Chapter 3 introduces the benchmarks that the approaches are tested on, presents the derived datasets and the transition model architectures used for the respective environments.
The main contributions of the thesis are then presented in Chapters 4-6: At first, we develop indicators ERI, EAS, and COI to assess data quality for offline RL in Chapter 4. We evaluate them empirically and show that they are capable to select a favourable subset of datasets in which offline RL algorithms can actually improve. Chapter 5 then presents two novel model-based offline algorithms, MOOSE and WSBC, that aim to derive well performing policies more robustly than prior works. We examine their performance on industrial benchmark and MuJoCo tasks more from a worst case perspective in order to realistically assess their suitability for true offline settings. We compare their results with a representative set of concurrently proposed offline algorithms. Chapter 6 then introduces the LION algorithm, which extends the offline setting in the sense that users need to be able to influence the policy's behavior after training if it does not behave as desired. To this end, LION trains proximity conditioned policies, which enable the user to tune the familiarity of the solution to a level where they are content with both its performance and familiarity. Table 1.1 provides a short overview over the algorithms' characteristics and first appearances that are presented and proposed in this thesis. We summarize and conclude our findings in Chapter 7.

	Model-free	Model-based	Hybrid	Offline	Adaptive	Chapter
DDPG	✓					5.1
BCQ	✓			✓		2.5
BEAR	✓			✓		2.5
BRAC	✓			✓		2.5
CQL	✓			✓		2.5
MOPO		✓	✓	✓		2.5
MOReL		✓	✓	✓		2.5
MOOSE		✓		✓		5.1
WSBC		✓		✓		5.2
LION		✓		✓	✓	6

Table 1.1: Brief characterization of the proposed algorithms (bold) as well as the concurrently developed prior works.

2 Preliminaries

This Chapter presents the basic methodologies and approaches that lay the foundation for the novel algorithms developed in this thesis, as well as prior offline RL algorithms which are throughout this thesis used as baselines in experimental comparisons. Sections 2.1 as well as 2.2 provide a brief introduction to reinforcement learning and its basic components and techniques. Since this thesis focuses on offline RL, Section 2.3 motivates the necessity for learning from static datasets and introduces its additional challenges. To show that the challenges need to be addressed with new methods, 2.4 shows how prior algorithms (even "batch" or "off-policy" methods) fail when the static dataset doesn't cover the state-action space properly. Section 2.5 then introduces concurrently proposed algorithm designs to mitigate the problems caused by offline learning. Finally, sections 2.6 (particle swarm optimization) and 2.7 (variational autoencoders) introduce methods used in the novel algorithms that are proposed throughout this work.

2.1 Reinforcement Learning

Reinforcement Learning is a sub-field of machine learning inspired by the idea that biological systems learn through interaction with their surroundings. As such, it is concerned with computational approaches that can learn to achieve goals by interacting with an environment. The interaction is usually taking place at discrete time steps, during which the learning entity, commonly referred to as the agent, performs an action and then perceives the new state of its environment. The agent is in this case never explicitly told what to do - instead, it additionally receives a reward signal from the environment which it seeks to maximize. This differentiates RL from the two other main areas of machine learning: In *supervised learning*, the goal is to associate data points with the correct labels as precisely as possible (a common example would be object recognition in images). In *unsupervised learning*, the goal is to find structural similarities in collections of unlabeled data (a common example would be document clustering). In reinforcement learning however, the agent neither seeks to replicate a set of proposed actions, nor does it try to uncover some hidden structure in the data (although methods from these areas may also play a role). Instead, it tries to find a way of acting in its environment, such that the expected cumulative reward signal over time is maximized.
Usually, the reward signal at a certain time step is in meaningfully difficult environments not just influenced by the action proposed in that very step, but by the actions taken in the past as well. Similarly, actions taken now influence rewards in the future, giving rise to the credit assignment problem: The agent cannot easily tell which actions lead to which rewards, since they can be arbitrarily delayed. Furthermore, the reward signal

may in realistic scenarios be subject to noise patterns, or states may not be fully observable, complicating the situation further. RL systems thus usually differentiate between rewards (the reward of a single step) and returns (the cumulative rewards of an entire interaction episode, consisting of arbitrarily many time steps), and try to maximize the latter.

More formally, the environment in an RL problem is commonly formulated as a *Markov Decision Process* (MDP) $\mathcal{M} =< \mathcal{S}, \mathcal{A}, T, r, \gamma, \mathbf{s_0} >$, where $\mathcal{S}$ is the set of states describing the environment , $\mathcal{A}$ is the set of actions the agent can choose from to perform in the environment, $T(s'|s,a) : \mathcal{S}\times\mathcal{A}\times\mathcal{S} \rightarrow \mathbb{R}$ is the transition function, specifying the dynamics of how the underlying system moves from one state to the next, $r(s,a,s') : \mathcal{S}\times\mathcal{A}\times\mathcal{S} \rightarrow \mathbb{R}$ is the reward function, $\gamma \in [0,1]$ is the discount factor specifying how much future rewards need to be discounted with respect to current rewards, and $\mathbf{s_0}(\cdot) : \mathcal{S} \rightarrow \mathbb{R}$ is the distribution of starting states.

In the wind turbine example, the state space could be composed of the sensor readings corresponding to electrical power being generated, current rotational speed, torque, orientation as well as angle of attack of the turbine blades, measurements of the wind & its turbulence, and vibrations & velocities of the blades. As actions, we can consider changes in the electrical power generated as well as orientation and angle of attack of the blades. The transition function will translate the actions to new states and the reward function is likely the generated electrical power, possibly combined with a penalty for wear and tear of components.

The agent and environment engage in an action - perception loop (seen in Fig. 2.1), and produce trajectories of the form $\tau = (s_0, a_0, r_0, s_1, a_1, r_1, ..., s_{H-1}, a_{H-1}, r_{H-1})$, where H is referred to as the horizon of the MDP. It may be a finite number, but horizons might as well be infinitely long - or the environment has a set of terminal states after which a trajectory automatically ends. When the future of a trajectory depends only on the current state and future actions, the environment states are said to be *Markov*, which is a common assumption in RL algorithms, however might not always be satisfied (many partially observable environments exist). The discounted cumulative return of a trajectory τ is defined as:

$$R^{\tau} = \gamma^0 r_0 + \gamma^1 r_1 + ... + \gamma^{H-1} r_{H-1} = \sum_{t=0}^{H} \gamma^t r_t \tag{2.1}$$

Note that discounting future rewards makes not only sense from an economical perspective (payments obtained in the future are commonly discounted by the so called risk-free interest rate in economics, since the missed opportunity to earn interest needs to be accounted for), but is also necessary when the horizon is potentially infinitely long: When we assume individual rewards to be bounded, e.g. $0 \leq r_t \leq \alpha \quad \forall t \in \mathbb{N}$, then the infinite

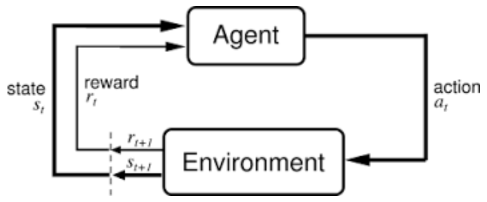


Figure 2.1: Agent-environment interaction in a Markov decision process. Figure from [18].

sum describing the return also has a finite upper bound:

$$R_{\max} = \sum_{t=0}^{\infty} \gamma^t \alpha = \frac{\alpha}{1-\gamma} \tag{2.2}$$

Returns can not only be calculated for full trajectories, but also for partial ones, i.e. $R_k = \sum_{t=k}^{H} \gamma^{t-k} r_t$. These returns then have the property that they can be expressed in terms their future returns, which is essential for theoretical foundations regarding value functions, which we will introduce in the next section:

$$\begin{aligned} R_t &= r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + ... \\ R_t &= r_t + \gamma(r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ...) \\ R_t &= r_t + \gamma R_{t+1} \end{aligned} \tag{2.3}$$

The goal of an RL agent is to maximize its expected cumulative return by optimally choosing the actions in the trajectories. To this end, the agent usually searches for a mapping $\pi(a|s) : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, commonly referred to as policy. The expected return when following policy π can then be written as:

$$\mathbb{E}_\pi[R] = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r(s_t, \pi(s_t)) \quad \middle| \quad s_0 \sim \mathbf{s_0}, a_t \sim \pi(s_t), s_{t+1} \sim T(s_t, a_t) \right] \tag{2.4}$$

Policies can be represented in different ways, including lookup tables or parameterized functions. Throughout this work, policies will always be considered to be a simple feed-forward neural network with parameters θ. Furthermore, the algorithms developed in this thesis all feature deterministic policies, i.e. $a_t = \pi_\theta(s_t)$.

2.2 Model-free vs. Model-based Reinforcement Learning

One of the key distinctions between various reinforcement learning algorithms is whether or not they feature a trained transition model $\hat{T}$ that is supposed to replicate the true transition function T. Methods that do train such a model are commonly referred to as model-based, while those who do not are called model-free. Since many baselines considered in this thesis are model-free methods, and since RL has historically emerged from these methods, this section starts with an introduction of the model-free RL paradigm. Afterwards, we examine the key differences to model-based approaches to solving MDPs, and introduce a key distinction we make from popular literature by differentiating between methods that use **only** transition models and no value functions, and those that use both (which we will refer to as hybrid methods).

2.2.1 Model-free

Model-free RL algorithms usually involve estimating a so called *value function.* In its most basic form, a state-value function $v_\pi(s)$ is supposed to provide the return that an agent would obtain if the environment were currently in state s_t, and the agent would from here on follow the policy π, i.e. $\forall k \in \mathbb{N}: \ a_{t+k} \sim \pi(s_{t+k})$ and $s_{t+k+1} \sim T(s_{t+k}, a_{t+k})$:

$$v_\pi(s) = \mathbb{E}_\pi\left[R_t | s_t = s\right] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \;\middle|\; s_t = s\right] \tag{2.5}$$

The state-value function thus basically tells the agent how good it is to be in a specific state s, when the agent is following policy π. The agent can in turn use this to estimate the expected return of a policy π:

$$\mathbb{E}_\pi[R_0] = \mathbb{E}_{\pi, s\sim\mathbf{s_0}}[v_\pi(s)] \tag{2.6}$$

which lays the basis to evaluate policies and finding better ones.
Similarly, the action-value function $q_\pi(s, a)$ is defined as the expected return R_t when the environment is in state $s_t = s$, the agent performs action $a_t = a$, and from then on follows the policy π:

$$q_\pi(s, a) = \mathbb{E}_\pi\left[R_t | s_t = s, a_t = a\right] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \;\middle|\; s_t = s, a_t = a\right] \tag{2.7}$$

Again, just as the state-value function, the action-value function can thus be used to estimate the expected return of a policy π:

$$\mathbb{E}_\pi[R_0] = \mathbb{E}_{\pi, s\sim\mathbf{s_0}, a\sim\pi(s)}[q_\pi(s, a)] \tag{2.8}$$

Similarly to Equation 2.3, the state- and action-value functions can be expressed in terms of a recursive relationship ensuring their consistency:

$$\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi[R_t|s_t = s] \\
&= \mathbb{E}_\pi[r_t + \gamma R_{t+1}|s_t = s] \\
&= \sum_a \pi(a|s) \sum_{s'} T(s'|s,a) \left[r(s,a,s') + \gamma \mathbb{E}_\pi[R_{t+1}|s_{t+1} = s']\right] \\
&= \sum_a \pi(a|s) \sum_{s'} T(s'|s,a) \left[r(s,a,s') + \gamma v_\pi(s')\right] \qquad (2.9) \\
q_\pi(s,a) &= \mathbb{E}_\pi\left[R_t|s_t = s, a_t = a\right] \\
&= \mathbb{E}_\pi\left[r_t + \gamma R_{t+1}|s_t = s, a_t = a\right] \\
&= \sum_{s'} T(s'|s,a) \left[r(s,a,s') + \sum_{a'} \pi(a'|s')\ \gamma\ \mathbb{E}_\pi[R_{t+1}|s_{t+1} = s', a_{t+1} = a']\right] \\
&= \sum_{s'} T(s'|s,a) \left[r(s,a,s') + \sum_{a'} \pi(a'|s')\ \gamma\ q_\pi(s',a')\right] \qquad (2.10)
\end{aligned}$$

Equations 2.9 and 2.10 are known as the *Bellman Equations* for MDPs.
Value functions impose a partial ordering on the set of policies: A policy π can be said to be better than another policy π', if it is expected to achieve a higher reward in any possible state, i.e. $\forall s \in \mathcal{S} : \pi(s) \geq \pi'(s)$. There always exists at least one policy that is better than or equal to all other policies, which is commonly called the optimal policy and denoted π_*, even though there can be many optimal ones. All policies π_* share the same state- and action-value functions v_* and q_*, which can be defined as:

$$v_*(s) = \max_\pi v_\pi(s) \quad \forall s \in \mathcal{S} \qquad (2.11)$$

$$q_*(s,a) = \max_\pi q_\pi(s) \quad \forall s \in \mathcal{S}, a \in \mathcal{A} \qquad (2.12)$$

For optimal value functions, Equations 2.9 and 2.10 can be written without specific reference to a single policy π, by taking the maximum value over the set of possible actions $\mathcal{A}$, yielding the *Bellman Optimality Equations*:

$$\begin{aligned}
v_*(s) &= \max_{a\in\mathcal{A}} q_*(s) \\
&= \max_{a\in\mathcal{A}} \mathbb{E}_{\pi_*}\left[R_t | s_t = s, a_t = a\right] \\
&= \max_{a\in\mathcal{A}} \mathbb{E}_{\pi_*}\left[r_t + \gamma R_{t+1} | s_t = s, a_t = a\right] \\
&= \max_{a\in\mathcal{A}} \mathbb{E}\left[r_t + \gamma v_*(s_{t+1}) | s_t = s, a_t = a\right] \\
&= \max_{a\in\mathcal{A}} \sum_{s'} T(s'|s,a)\left[r(s,a,s') + \gamma v_*(s')\right] \qquad (2.13) \\
q_*(s,a) &= \mathbb{E}_{\pi_*}\left[R_t | s_t = s, a_t = a\right] \\
&= \mathbb{E}_{\pi_*}\left[r_t + \gamma R_{t+1} | s_t = s, a_t = a\right] \\
&= \mathbb{E}\left[r_t + \gamma \max_{a'\in\mathcal{A}} q_*(s',a') | s_t = s, a_t = a\right] \\
&= \sum_{s'} T(s'|s,a)\left[r(s,a,s') + \gamma \max_{a'\in\mathcal{A}} q_*(s',a')\right] \qquad (2.14)
\end{aligned}$$

In order to use value functions to find well performing policies, generally two things are needed: (i) A method to compute the value function v_π given a policy π (in Dynamic Programming literature this is referred to as policy evaluation), and (ii) a way to improve a policy once its value function has been computed (referred to as policy improvement). It can be shown that policy evaluation can be achieved by iteratively applying the Bellman equation to an arbitrarily initialized (except terminal states, which need to have of value of zero) estimate of a value function v_0 or q_0, i.e.:

$$v_\pi^{k+1}(s) = \sum_a \pi(a|s) \sum_{s'} T(s'|s,a)\left[r(s,a,s') + \gamma v_\pi^k(s')\right] \qquad (2.15)$$

$$q_\pi^{k+1}(s,a) = \sum_{s'} T(s'|s,a)\left[r(s,a,s') + \sum_{a'} \pi(a'|s')\ \gamma\ q_\pi^k(s',a')\right] \qquad (2.16)$$

Policy improvement can then be achieved by examining deviations from the current policy π: If the agent is in state s, then it knows the value of the state if it follows π, however maybe it would be better to perform an action $a^{\text{new}} \neq \pi(s)$. The value of performing the action a^{new} and then following π again is given by $q(s, a^{\text{new}})$, which can also be expressed in terms of $v(s)$:

$$q_\pi(s, a^{\text{new}}) = \sum_{s'} T(s'|s, a^{\text{new}})\left[r(s, a^{\text{new}}, s') + \gamma v_\pi(s')\right] \qquad (2.17)$$

If the value of $q_\pi(s, a^{new})$ is greater than $v_\pi(s)$, then a new policy $\pi'(s) = a^{new}$ has been found, which is better than the prior policy, i.e. $\pi' \geq \pi$. It can be shown that if T and r are known, iteratively applying policy evaluation and policy improvement, will converge to the optimal policy π_* as well as the optimal value function v_*. This procedure is known as *policy iteration*. Using the Bellman optimality equation, the policy evaluation part may even be truncated to a single step (for each state) instead of performing it iteratively until convergence. The approach is known as *value iteration* and can still be shown to converge to the optimal value function v_*:

$$v^{k+1}(s) = \max_{a \in \mathcal{A}} \sum_{s'} T(s'|s,a) \left[r(s,a,s') + \gamma v^k(s') \right] \tag{2.18}$$

Since we can generally not assume complete knowledge over T or r, and it is not explicitly modeled in model-free reinforcement learning methods, algorithms need to learn primarily from experience, i.e. they interact with the environment and learn only from the data that has thereby been collected. Model-free algorithms mostly learn from the data by storing it in a FIFO buffer $\mathcal{B}$ of fixed size, from which they sample to compute the expectations over the transition dynamics. Furthermore, in large action spaces (possibly infinitely large in the continuous case), the expectations are not calculated over all possible future actions, but instead a few actions are sampled from the policy (if the policy is deterministic, a single sample obviously suffices). That way, we may rewrite the Bellman Equations 2.9 and 2.10 as:

$$v_\pi(s) \approx \mathbb{E}_{s,a,s' \sim \mathcal{B}}[r(s,a,s') + \gamma v_\pi(s')] \tag{2.19}$$

$$q_\pi(s,a) \approx \mathbb{E}_{s,a,s' \sim \mathcal{B}, a' \sim \pi(s')}[r(s,a,s') + \gamma q_\pi(s',a')] \tag{2.20}$$

While Monte-Carlo based methods would also be able to compute value functions without knowledge over T or r, they would need to simulate trajectories until termination in order to estimate values, which slows down learning, and has been empirically shown to be much less data efficient than *temporal difference* (TD) methods, which use the Bellman operator directly to bootstrap on the past estimate of the value function to obtain a new one. TD methods aim to minimize the TD error in order to enforce a consistent value function, and it has been shown that this leads to convergence to the correct value function for the policy that is being evaluated:

$$v_\pi^{k+1}(s) = \arg\min_v \mathbb{E}_{s,a,s' \sim \mathcal{B}} \left[[(r(s,a,s') + \gamma v_\pi^k(s')) - v(s)]^2 \right] \tag{2.21}$$

$$q_\pi^{k+1}(s,a) = \arg\min_q \mathbb{E}_{s,a,s' \sim \mathcal{B}, a' \sim \pi(s)} \left[[(r(s,a,s') + \gamma q_\pi^k(s',a')) - q(s,a)]^2 \right] \tag{2.22}$$

Just as in the dynamic programming paradigm, where T and r are assumed to be known, the estimated value functions can then be used in a policy- or value iteration based algorithm style to obtain improving policies. The model-free algorithms considered in this thesis all approximate both the value function as well as the policy by means of a neural network, since the state and action spaces are continuous and thus a lookup table would be infinitely large. The parameters of the neural networks representing the value functions are optimized to minimize the TD-error, as shown in Equations 2.21 and 2.22, using gradient descent (usually only the q-function is used). The policies are then also optimized via gradient descent to maximize their value:

$$\pi^{k+1}(s) = \arg\max_{\pi} \mathbb{E}_{s\sim\mathcal{B}}[v_{\pi^k}^{k+1}(s)] \tag{2.23}$$

$$\pi^{k+1}(s) = \arg\max_{\pi} \mathbb{E}_{s\sim\mathcal{B},a\sim\pi}[q_{\pi^k}^{k+1}(s,a)] \tag{2.24}$$

Due to the function approximation, the convergeance guarantees of the methods are unfortunately lost. They have however been shown to still work well empirically, and thus enable learning in continuous environments, which would otherwise not be possible.

2.2.2 Model-based

Model-based algorithms train a surrogate model $\hat{T}$, which is supposed to replicate the true transition dynamics function T. Since the true dynamics function is unknown, the surrogate needs be learned entirely from experienced interactions with the agent's environment. Once trained, a transition model predicts the future state s' based on the current state s and the action a chosen by the agent:

$$s' \sim \hat{T}(s,a) \tag{2.25}$$

or

$$s' = \hat{T}(s,a) \tag{2.26}$$

if $\hat{T}$ is deterministic. Throughout this thesis, transition models are considered to be neural networks. They are generally trained on a batch of data $\mathcal{B}$ that contains N tuples of states, actions, future states, and rewards that have been collected by the agent or a different entity, by interacting with the environment. In order to optimize the parameters ϕ of the transition model $\hat{T}_\phi$'s prediction output, a maximum likelihood approach is usually followed:

$$L(\phi) = \prod_{i=0}^{N-1} [p(s_i'|s_i, a_i, \phi)] \tag{2.27}$$

$$\phi^* = \arg\max_{\phi} L(\phi) \tag{2.28}$$

For a probabilistic transition model, many prior works assume a network that predicts mean μ_ϕ and variance Σ_ϕ of a Gaussian distribution. Since the product over many terms leads to numerical instabilities, the logarithm of the likelihood is usually optimized, since it transforms the product into a sum and doesn't change the location of the optimum. As gradient descent usually minimizes the target function, the negative log likelihood

(NLL) is usually the standard loss function used in this setting:

$$\begin{aligned}
L^{prob}(\phi) &= -\log \prod_{i=0}^{N-1} [p(s_i'|s_i, a_i, \phi)] \\
&= -\sum_{i=0}^{N-1} [\log p(s_i'|s_i, a_i, \phi)] \\
&= -\sum_{i=0}^{N-1} \left[\log \mathcal{N}(s_i'|\mu_\phi(s_i, a_i), \Sigma_\phi(s_i, a_i))\right] \\
&= -\sum_{i=0}^{N-1} \left[\log \left[\frac{1}{\sqrt{2\pi\Sigma_\phi(s_i, a_i)^2}} e^{-\frac{(s'-\mu_\phi(s_i,a_i))^2}{\Sigma_\phi(s_i,a_i)^2}}\right]\right] \\
&= \sum_{i=0}^{N-1} \left[-\log \left[\frac{1}{\sqrt{2\pi\Sigma_\phi(s_i, a_i)^2}}\right] + \frac{(s_i' - \mu_\phi(s_i, a_i))^2}{\Sigma_\phi(s_i, a_i)^2}\right] \qquad (2.29)
\end{aligned}$$

For a deterministic model, the result is a little simpler since we assume Gaussian distributed errors with a fixed variance:

$$\begin{aligned}
L^{det}(\phi) &= \sum_{i=0}^{N-1} \left[-\log \left[\frac{1}{\sqrt{2\pi\Sigma^2}}\right] + \frac{(s_i' - \mu_\phi(s_i, a_i))^2}{\Sigma^2}\right] \\
&= \frac{N}{\Sigma^2} - N \log \left[\frac{1}{\sqrt{2\pi\Sigma^2}}\right] + \sum_{i=0}^{N-1} (s_i' - \mu_\phi(s_i, a_i))^2 \qquad (2.30)
\end{aligned}$$

Since the location of the optimum stays the same when a constant is added, the first term may be omitted to yield the well known mean squared error:

$$\phi^*_{det} = \arg\min_\phi \sum_{i=0}^{N-1} (s_i' - \mu_\phi(s_i, a_i))^2 \qquad (2.31)$$

To account for different batch sizes, in both cases the expected value is usually optimized instead of the sum over all elements in the batch:

$$\phi^* = \arg\min_\phi \mathbb{E}_{\mathcal{B}}[L(\phi)] \qquad (2.32)$$

Instead of directly predicting the future state based on current state and action, transition models may as well try to predict the difference $\Delta s = s' - s$. Prior work has found that in some problems this quantity appears to be easier to model than the full state. In any case, it is beneficial to normalize the target as well as the inputs by their observed mean and variance in the batch, i.e. $s_{\text{norm}} = \frac{s-\mu_s}{\sigma_s}$. Prior work has found that this can have large effects on the performance of the model. In the notation above, we implicitly consider the reward as a dimension of the state, however one may also learn an extra model $\hat{r}(s, a, \hat{s})$ on top of $\hat{T}$ in order to predict the reward separately.

When the state of an environment is not fully observable, i.e. there exist dimensions that influence the future state and reward that are not part of the state that the agent can perceive, it is often useful to include past states, in order to maintain an estimate of the hidden state of the environment and still come up with accurate predictions. In this case, recurrent neural networks (RNNs) are usually the model of choice: In addition to the current state and action, they take as an input the learned hidden state h_t at time step t, and are trained so that h_t reflects all the information necessary from the past states in order to predict the future:

$$s_{t+1} \sim \hat{T}_\phi(s_t, a_t, h_t) \tag{2.33}$$

or

$$s_{t+1} = \hat{T}_\phi(s_t, a_t, h_t) \tag{2.34}$$

Like a trained value function, a transition model $\hat{T}$ can be used to evaluate the performance of an RL policy π. Instead of simply evaluating the value function over the set of starting states, the transition model is used to perform a "virtual", or "imagined" trajectory, by iteratively querying the policy for an action and the transition model for a future state, starting with $s_0 \in \mathbf{s_0}$:

$$R(\pi, \hat{T}) = \mathbb{E}_{s_0 \in \mathbf{s_0}} \left[\sum_{t=0}^{H-1} \gamma^t r(\hat{T}(\hat{s}_t, \pi(\hat{s}_t), h_{t-1})) \right] \tag{2.35}$$

Here, the reward is assumed to be either implicitly or explicitly part of the state, so that $r(\cdot)$ simply denotes extraction of the reward output from the transition model. One may however as well use an extra model for the reward, i.e.:

$$R(\pi, \hat{T}) = \mathbb{E}_{s_0 \in \mathbf{s_0}} \left[\sum_{t=0}^{H-1} \gamma^t \hat{r}(\hat{s}_t, \pi(\hat{s}_t), s(\hat{T}(\hat{s}_t, \pi(\hat{s}_t), h_{t-1}))) \right] \tag{2.36}$$

Both times, $\hat{s}$ denote the past state predictions of the model that are fed back into it together with the maintained hidden state and new action.

Since both the transition model $\hat{T}$ as well as the policy π are assumed to be differentiable neural network models, it is possible to optimize the policy parameters θ by means of gradient descent, however in this thesis we will also explore gradient-free options.

$$\theta^* = \arg\min_\theta -R(\pi_\theta, \hat{T}) \tag{2.37}$$

Using Eqs. 2.35 or 2.36 to assess policy performance and then directly optimize it with Eq. 2.37 can be considered a special case of model-based RL methods that do not use any value function. Often, when an algorithm is described as being model-based, there is still a state- or action-value function involved, a paradigm that will be briefly introduced in the next section.

2.2.3 Hybrid Approaches

Many works that propose algorithms in the model-based domain do not directly use the trained transition models to optimize the policy. Instead, a value function is additionally trained to assess the policy's performance. The policy is again optimized with respect to the value function (like in the model-free setting), however the data that is collected by the policy does not necessarily stem from the real environment. Instead, a percentage of the interactions come from the transition model (up to 100%, i.e. no real environment interaction any more). Throughout this thesis, such approaches will be called *hybrid* since they apply methods from both worlds. One of the classic hybrid algorithms is the so-called dyna architecture, in which the agent in each episode collects some experience from the real environment, then performs a value function update with the collected data, trains the transition model, and finally updates the value function for a number of steps, but this time with simulated data from the transition model. Other, more recent examples of hybrid approaches include [41, 42, 43].
Whether hybrid or not, model-based methods are generally regarded to be more data-efficient, i.e. they need to collect fewer interactions with the true dynamics to arrive at the same policy performance as model-free ones. This makes sense intuitively since the same interaction is used multiple times (in each epoch of model training) and since the information it represents is saved in the transition model, where it is re-used when new data is generated. Another theory about why model-based methods are more efficient concerns learned representation: Since the transition model needs to predict all state dimensions as well as the reward, the supervision signal provided can be much more helpful than when just trying to predict the value. Since offline RL is an inherently data-scarce task, data efficiency is critical, and we are thus primarily concerned with developing model-based algorithms throughout this thesis.

2.3 Batch and Offline Reinforcement Learning

While reinforcement learning has classically emerged from the assumption that the agent is allowed to directly interact with its environment, some works have always explored alternative settings, in which e.g. only a limited number of policy deployments was possible (see [9, 44, 45]). These methods have been called semi-batch, or growing batch setting, since the dataset the agent learns from is in this case not entirely static, but instead the agent can collect additional data a certain number of times. At the same time, the setting is also not entirely online any more, because the agent cannot change the collecting policy arbitrarily often (in the extreme case after every update).
In the “batch” or “offline” setting, interaction with the real environment is entirely prohibited until a final policy has been trained and is being deployed. The setting became more popular recently, since it promises to enable learning control strategies purely from static, previously collected datasets, that have been obtained via logging during ordinary operation, which would alleviate many issues relating to safety: Most online methods at least initially don't act very goal oriented and need to take many random exploratory actions that could in real-world scenarios result in damaged equipment or even be harm-

ful for humans (e.g. nobody would want to train an RL policy for self driving cars from scratch). However, most of the early batch RL methods, such as [46, 47, 11] assume that the dataset was collected under an extremely exploring policy, e.g. uniformly random actions at every time step. While this leads to models that generalize well everywhere in the state-action space, it is not generally a realistic assumption in offline RL. Many datasets contain only very limited exploration since most of the data is collected during productive operation, where many times, similar events occur over and over.

In the most recent wave of offline RL works (e.g. [48, 49, 36]), datasets have been considered that contain very narrow data distributions up to no exploration at all. Corresponding algorithms need to balance carefully between optimizing their policy for performance and staying within the support of the dataset, since otherwise the trained policy performs poorly once deployed on the real system, an issue that unadapted online algorithms face, as will be shown in the next section. See Fig. 2.2 for a simplified comparison between offline and online learning.

2.4 Fail Cases of Prior Methods on Offline Data

Prior works have shown that algorithms that are designed for the online setting cannot directly be used for the offline setting, even if they are designed to learn from off-policy data, i.e. interactions that weren't collected under the same policy. In [8], the authors train a DDPG (deep deterministic policy gradient, a widely used model-free algorithm [4]) agent online in a simulated environment until it reaches a certain performance. DDPG uses a so called replay buffer, from which it samples interactions to learn from, so in principle it should be possible to learn from offline datasets with DDPG, simply by putting all interactions from the static dataset into the replay buffer and not allowing any further collection. Their study however shows that this is not possible: They collect different datasets, either by just evaluating the online trained agent in the environment and recording the result or by recording all interactions during training of the online agent (both can be augmented with action space noise). The DDPG agents then trained

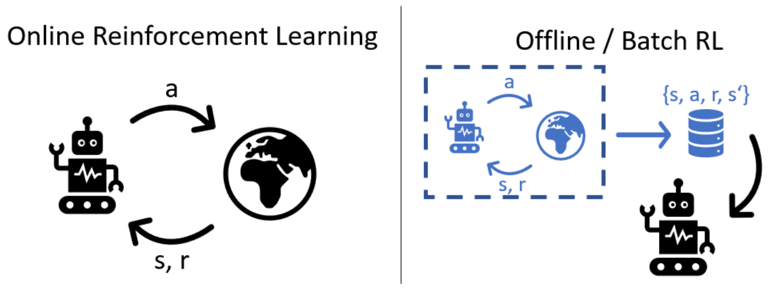


Figure 2.2: Difference between commonly assumed online RL, and offline RL, where interaction with the real environment is not possible for the agent, and behavior needs to be learned only from recorded past interactions.

on the resulting datasets however are unable to remotely replicate the performance, even though some of them are trained on the exact same data. See Fig. 2.3 for a visualization of the resulting training curves of online and offline trained DDPG agents. The authors attribute the discrepancy between the performances of the two agents mostly to a phenomenon they term *extrapolation error*, which occurs when during optimization of the policy, the value function is evaluated for state-action pairs that it hasn't seen in the training data. In online algorithms, this would be corrected by collecting new data, but in the offline case, there is no recovery.

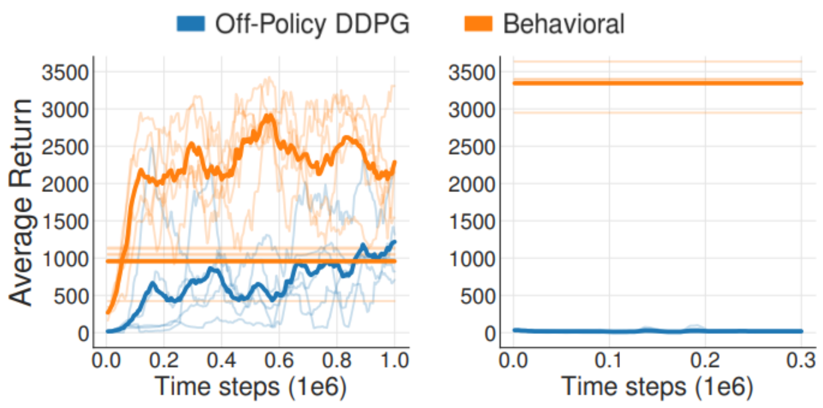


Figure 2.3: Performance curves of online and offline learning DDPG agents. On the left, the online agent directly trains in the environment with some additional action noise, while the offline agent trains on the final dataset containing all transitions that occurred during training of the online agent. On the right, the final trained online agent is only being evaluated, and the recorded data is used for training of the offline agent. Figure adapted from [8].

While DDPG is a model-free algorithm that fails in the offline context, we briefly point out in this section that model-based approaches may suffer from the very same phenomenon, since the transition models are also evaluated for state-action pairs that haven't been seen before if the trained policy deviates from the policy that collected the data. In Fig. 2.4, we show that a naive model-based setup, such as the one discussed in Section 2.2.2 fails to reproduce even the generating policy's performance on a subset of the industrial benchmark datasets (which we will introduce later in Chapter 3), which exhibit little to no exploration. Throughout this thesis, it will be our goal to close this gap, and find policies that perform better than the generating policy, by deriving algorithms that can effectively learn from datasets even when they contain very little exploration.

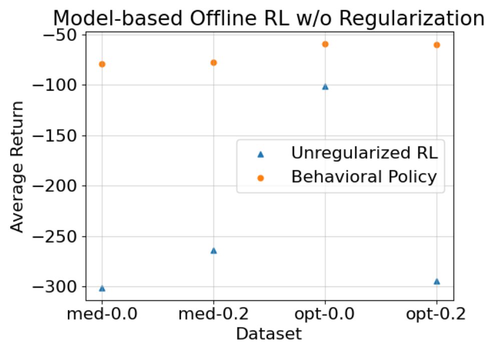


Figure 2.4: A simple model-based policy optimization scheme exhibits similar issues as found in the model-free setting. The trained policies evaluate the transition models in areas of the state-action space that haven't been seen in the datasets, causing high extrapolation error and ultimately a performance below the generating policy.

In the wind turbine example, if an operator would like to train an RL policy, because they hope to improve long term performance and thus revenue of their system, they are faced with a variety of issues when trying to do so: Classical online algorithms would need to learn on the turbine directly, which would due to many exploratory actions mean at least a likely much lower performance in the beginning, possibly even damage of equipment due to actions that lead the system to undesired states (e.g. high vibrations, too high rotational speeds, etc.). Also, prior off-policy approaches cannot deal with just a batch of past operational data without new system interaction, meaning a high likelihood of under-performing the previous controller with no hope of improvement. The operator thus has a natural need for algorithms that can learn a policy purely from historic operational data, i.e. offline RL algorithms.

2.5 Concurrently Proposed Offline RL Algorithms

Before this thesis was started, the offline RL scenario was rather uncommonly encountered in academic literature. While a few batch / offline RL works existed, most of them did not explicitly take into account the problem of the distribution shift, that occurs when training policies that visit states and actions that have not been seen in the dataset. Instead, many papers assumed a very well explored dataset, so that the extrapolation error could be ignored. A notable exception to this was the paper "Off-Policy Deep Reinforcement Learning without Exploration" [8], which has been claimed to propose the first batch deep reinforcement learning algorithm. In it, the authors show that classic RL algorithms fail when they are not allowed to interact directly with the environment, but instead have to learn just from a previously collected dataset. During the preparation of this thesis however, a plethora of methods have been proposed for offline learning for continuous control. In this section, we aim to give a representative overview and details on the most popular methods, which will serve as baselines later on in the experimental section. To this end, we mostly follow the summary provided in contribution [D]. The overview however does not make any claims regarding completeness - since the field of offline RL has taken on a huge amount of interest from academic and industrial players, the amount of methods proposed is growing rapidly, rendering any ambitions for an exhaustive review extremely challenging.

The **model-free**, value function based paradigm is employed by the majority of offline RL algorithms in the corresponding literature. One of the first methods to consider policy learning without environment interaction, only from a static pre-collected dataset with potentially insufficient amounts of exploration was Safe Policy Improvement under Baseline Bootstrapping (SPIBB) [50]. It relies on the assumption that the dataset generating policy is known and made available to the algorithm in order to provide a safe fallback option. Batch Constrained Q-learning (BCQ) [8] is the approach developed in the above mentioned "Off-Policy Deep Reinforcement Learning without Exploration". In contrast to SPIBB, it does not rely on the availability of the generating policy (none of the following approaches do), but instead learns a variational autoencoder (VAE) based representation of it from the dataset. Despite the learned perturbation model in BCQ, which tweaks sampled actions for better value in a predefined range, and could be viewed as a parametric policy model, the authors chose to give it the "Q-learning" stamp since it uses the Bellman optimality operator by taking the value-maximizing action among a set of sampled actions from VAE model. Further, BCQ is likely the first offline RL method designed for continuous state and action spaces. Expected Max Q-learning (EMaQ) [51] considers a simplification of the BCQ approach by getting rid of the perturbation model and only choosing the value maximizing action among a (larger) batch of sampled actions. Bootstrapping Error Accumulation Reduction (BEAR) [52] is one of the first offline methods to employ a closed-form policy: It learns by imposing a penalty based on the maximum mean discrepancy (MMD) [53] between target and (learned) generating policy actions. Behavior Regularized Actor Critic (BRAC) [54] is most similar to BEAR, however pioneered the idea that not only the policy, but also the value function should be penalized during training, meaning that the algorithm tries to assign lower values to state-action pairs outside the known data distribution.
The Advantage weighted Behavior Model (ABM) [55] simply "keeps doing what worked". Basically, the algorithm can be seen as a sort of weighted behavior cloning, where behavior that yielded high returns is weighted higher and is thus more likely to be replicated by the trained policy. Similar approaches have been presented in Advantage Weighted Regression (AWR) as well as Reinforcement Learning via Supervised Learning (RvS)[56, 57], where a policy is conditioned on the expected return-to-go in an episode (RvS does not even contain a value function). Best Action Imitation Learning (BAIL) [58], Critic Regularized Regression (CRR) [59], and Curriculum Offline Imitation Learning (COIL) [60] also build on the idea of weighted behavior cloning: BAIL uses a state-value function to imitate only well performing actions, CRR employs filtered policy gradients to only imitate behavior with high Q-value actions, and COIL imitates only behavior of policies adjacent to the currently trained one which exhibit higher estimated returns, since these

can be more easily learned.
Many developed offline RL approaches build on the DICE (DIstribution Correction Estimation) framework introduced in [61]: AlgaeDICE [62], GenDICE [63], and GradientDICE [64] all estimate the proportional discrepancy between the state-action visitation distributions of the newly trained and the generating policy in order to correct the return estimation. Similarly, in Policy Sampling Error Corrected TD-0 (PSEC-TD-0) [65] the actions are re-weighted using importance sampling according to how likely the actions are under the trained policy, instead of simply assuming the visitation frequency induced by the dataset for value function training.
The algorithms REM (Random Ensemble Mixtures) [66], PEBL (PEssimistic ensemBLes for offline deep reinforcement learning) [67], and O-RAAC (Offline Risk Averse Actor Critc) [68] rely on learning to minimize some form of risk instead of directly regularizing the trained policy towards the generating one: REM requires the policy to perform well on randomly selected members of the learned value function ensemble without an explicitly added penalty term. By contrast, PEBL builds on the Double Deep Q-learning (DDQ) algorithm as well as Soft Actor-Critic (SAC) [69, 7] to impose a value penalty based on the prediction discrepancies among the bootstrap ensemble members. O-RAAC also learns to be risk averse by employing pessimistic value estimates, which under some predefined distributional shifts give the algorithm theoretical performance guarantees. O-RAAC is shown to visit much fewer high risk states than prior offline methods.
The works presented in [70] and [71] (CQL - Conservative Q-Learning) both modify the bellman backup to train a conservative action-value function (i.e. one that lower bounds the true value of the corresponding action) and find an approximately optimal policy in the space of policies explored by the dataset.
Other proposed algorithms include OPAL (Offline Primitive discovery for Accelerating offline reinforcement Learning) [48], which aims to alleviate problems originating from sparse rewards (a classically hard problem since credit assignment becomes cumbersome). The algorithm builds a set of skills by imitating parts of trajectories, and then uses them as a set of actions for a higher level policy which only chooses the skill to execute (the step-wise skill execution is then performed by a lower level policy). This effectively shortens the horizon from the point of view of the algorithm and it is shown empirically to greatly improve learning in sparse reward settings. [72] presents Deep Averagers with Costs MDPs (DAC-MDPs), which introduce costs for exploiting under-represented parts of the state-action space in the initial dataset. [73] introduces TD3+BC (Twin Delayed Deep Deterministic policy gradient + Behavior Cloning), which is an adaptation of the TD3 algorithm to the offline setting by only introducing minimal changes. Despite not even modeling the generating policy, it still performs well on MuJoCo datasets.

Model-based methods have previously lead a niche existance in offline RL literature, but quickly catch on in popularity. Generally, model-based algorithms have been attributed better sample efficiency than model-free algorithms (i.e., they achieve the same returns with fewer environment interactions), while at the same time yielding lower asymptotic performance due to model bias [15, 16, 17]. In the offline setting, the former seems much more important, and model-based methods are thus often motivated by

highlighting their efficiency in limited data scenarios. As opposed to model-free methods, these algorithms learn a model of the transition dynamics of the underlying MDP, by predicting future states and rewards using prior states and actions as they were recorded in the initial dataset. In which capacity this model is then used however wildly differs among proposed algorithms. We will in the following differentiate methods by whether they use the models for additional data generation and whether they additionally make use of a value function. Basically, we identify two types of methods in the (offline) model-based RL iterature:
The first type uses the learned transition model directly for policy training without any extra models or data generation steps [11, 46, 74]. Usually, algorithms of this type perform imagined trajectories using the trained policy and the transition model instead of the true environment. They then search for policies that maximize the resulting estimated return in the imagined trajectories, possibly using gradients that are backpropagated throughout all interaction steps.
MOReL (Model-based Offline Reinforcement Learning), MOPO (Model-based Offline Policy Optimization), and COMBO (Conservative Offline Model-Based policy Optimization) [42, 43, 41] belong to the second type of model-based methods. This type uses the transition models not directly for policy training, but instead to generate additional interaction samples to enrich the initial dataset. This larger dataset is then used for offline value function learning and the policy is then still optimized with respect to the value function. Hence, we will refer to these algorithms as being **hybrid** methods in this thesis, as already outlined in Section 2.2.3. MOReL and MOPO both use their transition models to derive an uncertainty estimate, which is either used to penalize synthetic rewards, or to stop data generation in areas of the state-action space that the models are too uncertain about. The trained policy should thus be unable to exploit the transition models in regions of the state-action space in which they are not sufficiently accurate. COMBO also uses transition models for additional data generation. As the authors however observe that the accuracy of the generated data samples degrades the further away from the true data points they are collected, they simply penalize distance to the initial datapoints measured in time steps.

Baselines considered in this thesis: Among the above mentioned algorithms, we select a subset to compare against the new contributions we make in Chapters 5 and 6. Among them are the five model-free algorithms BCQ, BEAR, BRAC, CQL, and TD3+BC, as well as the model-based hybrid methods MOPO and MOReL. On top of that, we include a comparison with the popular (model-free) DDPG algorithm, which does not contain any adaptations to the offline RL setting. In the following, we briefly present the key characteristics of each algorithm.

Deep Deterministic Policy Gradient (DDPG) [4] trains a deterministic policy to maximize its value function. As it is not specifically designed to work in an offline setting, it samples interactions for its updates from a replay buffer $\mathcal{B}$, which is assumed to be regularly updated and filled with new samples that were collected under the currently trained policy by interacting with the real environment:

$$\hat{Q}^{k+1} = \arg\min_{Q} \mathrm{E}_{s,a,s',r\sim\mathcal{B}} \left[\left(r + \gamma \hat{Q}^{k}(s', \pi^{k}(s')) - Q(s,a) \right)^2 \right] \tag{2.38}$$

$$\pi^{k+1} = \arg\max_{\pi} \mathrm{E}_{s\sim\mathcal{D}} \left[\hat{Q}^{k+1}(s, \pi(s)) \right] \tag{2.39}$$

Batch Constrained Q-Learning (BCQ) [8] can be considered a special case on the edge between Q-learning and actor-critic approaches: While Q-learning approaches commonly use the Bellman optimality operator (i.e. the maximum over all actions) to update the value function and then classically select the action greedily without an explicit policy representation (originally, Q-learning is defined for discrete actions), actor-critic approaches usually learn an explicit representation of the policy and alternatingly use the ordinary Bellman operator (i.e. the expectation over actions) to estimate the Q-function for that policy and then optimize the policy with respect to the Q-function. BCQ trains a variational autoencoder (VAE) based representation G of the policy that generated the dataset and then uses it during value function training (and later during deployment) to sample actions that the generating policy would have taken given the current state. It then considers the action that yielded the best value going forward. However, the algorithm also learns a so called perturbation model $\xi(s, a, \Phi)$, which is allowed to change the chosen action up to a value of Φ, so the method features parts of both paradigms. Note that BCQ, as well as most other proposed algorithms, consider the reward r at a certain time step as an explicitly saved and sampled feature from the dataset:

$$\begin{aligned} \hat{Q}^{k+1} &= \arg\min_{Q} \mathrm{E}_{s,a,s',r\sim\mathcal{D}} \left[\left(r + \gamma \max_{a'\sim G(s')} \left[\hat{Q}^{k}(s', a' + \xi^{k}(s', a', \Phi)) \right] - Q(s,a) \right)^2 \right] \\ \xi^{k+1} &= \arg\max_{\xi} \mathrm{E}_{s\sim\mathcal{D}; a\sim G(s)} \left[\hat{Q}^{k+1}(s, a + \xi(s, a, \Phi)) \right] \end{aligned} \tag{2.40}$$

The final BCQ policy can then be written as:

$$\pi(s) = \arg\max_{a\sim G(s)+\xi(s,a,\Phi)} Q(s, a + \xi(s, a, \Phi)) \tag{2.41}$$

Bootstrapping Error Accumulation Reduction (BEAR) [52] can be much more clearly categorized as being an actor-critic algorithm. It alternates between learning an action-value function based on the dataset and the current policy and then updates its explicit policy representation to maximize the value function. Since BEAR is not constrained to only sample actions that were likely under the generating policy, it needs to regularize the policy in a different way: In addition to maximizing it's Q-value, the policy is trained to minimize the maximum mean discrepancy (MMD) between the trained policy and a similar VAE model G of the generating policy. In order to be able to do so, the policy π is not a deterministic feed-forward neural network like the perturbation model in BCQ, but instead a Gaussian policy, i.e. a network that predicts mean and standard deviation of a Gaussian distribution from which actions are then sampled.

$$\hat{Q}^{k+1} = \arg\min_Q \mathrm{E}_{s,a,s',r\sim\mathcal{D};a'\sim\pi^k(s')} \left[\left(r + \gamma \hat{Q}^k(s',a') - Q(s,a) \right)^2 \right] \tag{2.42}$$

$$\pi^{k+1} = \arg\max_\pi \mathrm{E}_{s\sim\mathcal{D}} \left[\hat{Q}^{k+1}(s,\pi(s)) - \lambda \mathrm{MMD}(\pi(s), G(s)) \right] \tag{2.43}$$

Behavior Regularized Actor Critic (BRAC) [54] is a generalization of BEAR: Instead of using a specific divergence metric, BRAC leaves open the choice of divergence to use for regularizing the policy to stay close to the generating policy. Also, BRAC does not use a VAE model of the generating policy, and instead models the generating policy β with the same architecture as the trained policy π (which is a Gaussian feed-forward network, like in BEAR). Further, BRAC introduces the idea that policy regularization is not the only way to keep the policy close to the generating policy in actor-critic approaches: In BRAC-v, not only the policy, but also the action-value function is penalized for state-action pairs unlikely under the generating policy. That means that Q-values for state-action pairs that haven't been observed are naturally lower and policy optimization should thus prioritize behavior closer to the generating policy even when purely maximizing value. The penalty coefficients α can of course be adjusted:

$$\begin{aligned} \hat{Q}^{k+1} &= \arg\min_Q \mathrm{E}_{s,a,s',r\sim\mathcal{D};a'\sim\pi^k(s')} \left[\left(r - Q(s,a) + \gamma \left[\hat{Q}^k(s',a') - \alpha \mathrm{D}(\pi^k(s'), \beta(s')) \right] \right)^2 \right] \\ \pi^{k+1} &= \arg\max_\pi \mathrm{E}_{s\sim\mathcal{D}} \left[\hat{Q}^{k+1}(s,\pi(s)) - \alpha \mathrm{D}(\pi(s), \beta(s)) \right] \end{aligned} \tag{2.44}$$

The authors empirically find the KL-Divergence to perform best (i.e., in the above, replace D with KL). The standard values for α are 1. A variant which only penalizes divergence during policy update is called BRAC-p and obtained by setting $\alpha = 0$ in the value update. BRAC-v is however found to perform better and thus used as a baseline when we refer to BRAC.

Conservative Q-Learning (CQL) [71] picks up on the idea introduced in BRAC to regularize the value function instead of the policy. The method aims to learn a Q-function that lower-bounds the true action-value function, so that unregularized policy updates to maximize the policy performance become feasible as they do not threaten to cause extrapolation error any more. To do so, the Q-function backup is modified, so that the Q-values are minimized for state-action pairs that have not been seen in the data:

$$\begin{aligned} \hat{Q}^{k+1} = \arg\min_Q \Big[& \alpha \cdot \left(\mathrm{E}_{s\sim\mathcal{D};a\sim\pi^k(s)}[Q(s,a)] - \mathrm{E}_{s,a\sim\mathcal{D}}[Q(s,a)] \right) \\ & + \frac{1}{2} \mathrm{E}_{s,a,s',r\sim\mathcal{D};a'\sim\pi^k(s')} \left[\left(r + \gamma \hat{Q}^k(s',a') - Q(s,a) \right)^2 \right] \Big] \end{aligned} \tag{2.45}$$

The terms multiplied by the regularization coefficient α are meant to minimize the value of actions proposed by the policy if they are not sufficiently similar to the actions seen in the dataset - otherwise, the two terms offset. Policy updates are then performed by simply maximizing the corresponding Q-values:

$$\pi^{k+1} = \arg\max_{\pi} \mathrm{E}_{s\sim\mathcal{D}} \left[\hat{Q}^{k+1}(s, \pi(s))\right] \tag{2.46}$$

Twin Delayed Deep Deterministic policy gradient + Behavior Cloning [73] features a more simplistic approach towards offline RL: The authors start with an existing RL algorithm, in this case TD3, and try to make only absolutely minimal changes in order to make it work in the offline case. To this end, the algorithm contains an additional behavior cloning term in the policy update, which penalizes squared differences between policy and data actions:

$$\pi^{k+1} = \arg\max_{\pi} \mathrm{E}_{s,a\sim\mathcal{D}} \left[\lambda\hat{Q}^{k+1}(s, \pi(s)) - (\pi(s) - a)^2\right] \tag{2.47}$$

Interestingly, even though the adaptation seems so simple, TD3+BC was able to perform well on the D4RL benchmark datasets. Together with CQL, it is among the few offline algorithms that do not need to train a model of the generating policy, reducing the amount of screws and bolts involved, making the method less error prone. The Q-function update is performed in the standard way, using the ordinary Bellman backup:

$$\hat{Q}^{k+1} = \arg\min_{Q} \mathrm{E}_{s,a,s',r\sim\mathcal{D}} \left[\left(r + \gamma\hat{Q}^{k}(s', \pi^k(s')) - Q(s,a)\right)^2\right] \tag{2.48}$$

Model-based Offline Policy Optimization (MOPO) [43] was one of the first model-based offline RL methods proposed. As such, it trains stochastic transition models $\hat{T}(s'|s,a)$ to estimate the true transition dynamics $T(s'|s,a)$. The networks are used to predict mean and standard deviation of a Gaussian distribution for the future states. Such a model can be trained by using the loss derived in Eq. 2.29. MOPO uses the standard deviations predicted by the transition models to derive an uncertainty penalty which is added on top of the reward for synthetic samples:

$$\hat{r}(s,a) = r(\hat{T}(s,a)) - \lambda \max_{i=1..M} ||\Sigma^i(s,a)||_F \tag{2.49}$$

Where $r(\hat{T}(s,a))$ denotes the reward prediction of the dynamics model when executing action a in state s, Σ^i denotes the predicted standard deviation of the future state distribution by the ith transition model, and $||.||_F$ is the frobenius norm. MOPO thus uses the dynamics ensemble with M members to generate new synthetic data with conservatively estimated rewards. A regular online, model-free algorithm can then be used on top of the ensemble as if it were the true environment. MOPO uses well known Soft Actor-Critic (SAC) [7] as an instance of such an algorithm.

Model-based Offline Reinforcement Learning (MOReL) [42] is conceptually very similar to MOPO: The algorithm also trains an ensemble of transition models to learn the dynamics from the interactions provided in the dataset. However, the transition models in MOReL are deterministic, so they are obtained using Eq. 2.31. Instead of directly penalizing synthetic rewards, the algorithm then defines a so called Uncertain State Detector (USD) by means of a model-disagreement approach:

$$\mathrm{USD}(s,a) = \left(\max_{i,j} \left[s(\hat{T}_i(s,a)) - s(\hat{T}_j(s,a)) \right]^2 \right) < \epsilon \tag{2.50}$$

where $s(\hat{T}_i(\cdot))$ denotes the future state prediction of the ith ensemble member and ϵ is a threshold parameter. The USD is used to identify when synthetically generated trajectories are reaching a part of the state-action space in which the transition models disagree too strongly. MOReL simply stops trajectories once an uncertain state is detected, making sure that synthetically generated data stays realistic. The algorithm uses TRPO [5] or NPG [75] to solve the approximated MDP as if it were the true environment. As outlined in Section 2.2.3 We thus call MOPO and MOReL *hybrid* methods instead of simply model-based.

While this section provides an overview over the methods used as baselines in this thesis, not all details are included. For example, most of the model-free algorithms presented use ensembling as well as target networks for value functions and policies to stabilize learning, MOPO and MOReL use model-free algorithms for which we didn't provide additional details, and we also left out some bells and whistles related to data handling and normalization. For more details on all the algorithms, we refer to the original papers.

2.6 Particle Swarm Optimization

Particle Swarm Optimization (PSO) is a method to optimize non-linear continuous functions. The algorithm has been proposed specifically to optimize weights of artificial neural networks and has been found through experiments with a simplified social behavior model: The authors of [76] were originally intending to model the social behavior of groups of animals, such as birds in a flock or fish in a swarm, when they discovered the algorithm was performing optimization. PSO is a gradient-free optimization algorithm, and is in Chapter 5 thus used as an alternative to the much more prominently used gradient-based optimization procedures in most machine learning algorithms. Instead of only maintaining a single candidate solution that is iteratively improved like in most gradient based algorithms, PSO maintains a population of candidates, which are also called particles. To substitute for the missing gradient information, PSO particles use information from their neighbouring particles to effectively search the parameter space. In our case, the parameter space will be the space of policies that can be expressed by a feed-forward neural network with a certain architecture.

More formally, PSO is aiming to optimize a fitness function $f : \mathcal{X} \to \mathbb{R}$, which maps solution candidates from the parameter space $x \in \mathcal{X}$ to a scalar value. The parameter space itself is a real-valued vector space of dimensionality D, i.e. $\mathcal{X} = \mathbb{R}^D$. For optimization, PSO sustains a population of N candidates $n = 0, ..., N-1$, each of which has a position x_n in the parameter space as well as a velocity v_n that determines how the particle moves in the parameter space. Through iteratively applying the velocity term to the position, and then updating the velocity term, the particles move through the parameter space in search of better fitness values. To denote the positions and velocities at a specific iteration i of the algorithm, we use x_n^i and v_n^i. A particles fitness value y_n^i at a certain iteration i is obtained by evaluating the fitness function for its current position, i.e. $y_n^i = f(x_n^i)$. Each particle also knows the position b_n^i, where it has received the best fitness so far, as well as the best fitness that any of its neighbours, $m \in \mathcal{N}(n)$, have been able to achieve so far: $\hat{b}_n^i$.

$$b_n^i = \underset{\{x_n^j | j \in 0,...,i\}}{\arg\max} \; f(x_n^j) \tag{2.51}$$

$$\hat{b}_n^i = \underset{\{b_m^i | m \in \mathcal{N}(n)\}}{\arg\max} \; f(b_m^i) \tag{2.52}$$

Note that not all past $\{x_n^j | j \in 0, ..., i\}$ need to be kept in memory to compute Eq. 2.51. Instead, one simply updates b_n^{i+1} using:

$$b_n^{i+1} = \begin{cases} b_n^i & \text{if } f(b_n^i) >= f(x_n^{i+1}) \\ x_n^{i+1} & \text{otherwise} \end{cases} \tag{2.53}$$

The velocities and the positions of the particles are initialized using $v_n^0 = 0$ and $x_n^0 \sim \mathcal{U}(x_{\min}, x_{\max})$ [77], and are updated using the following Equations:

$$v_n^{i+1} = v_n^i + c_1 \cdot r_1 \cdot [b_n^i - x_n^i] + c_2 \cdot r_2 \cdot [\hat{b}_n^i - x_n^i] \tag{2.54}$$

$$x_n^{i+1} = x_n^i + v_n^{i+1} \tag{2.55}$$

Where c_1 & c_2 are acceleration constants and r_1 & r_2 are randomly drawn during each iteration in order to add a stochastic element to the optimization process. Note that since the parameter space is D-dimensional, most components of the Eqs. 2.54 & 2.55, i.e. v_n^i, x_n^i, b_n^i, r_1, and r_2 are all D-dimensional vectors. Consequently, the factors in r_1 & r_2 are drawn independently for each of the d dimensions, i.e. $\forall d \in 0, ..., D-1 : r_1^d, r_2^d \sim \mathcal{U}(0, 1)$.

Neighbourhoods The neighbourhood of each particle n: $\mathcal{N}(n)$ is defined a priori and kept fixed throughout the optimization process, i.e. the neighbourhood doesn't actually have anything to do with low distances in the parameter space. Different topologies of neighbourhoods affect the behavior of the optimization process: Broadly speaking, more densely connected topologies (in the extreme case every particle is the neighbour of every other particle) lead to faster convergence at the expense of diversity and with the potential of prematurely converging to a local optimum [78]. On the other hand, more sparsely connected topologies lead to slower convergence, but feature more diverse solution candidates, which can help to find better performing particles in the end.

Velocity clamping One problem that arises during PSO optimization as defined above is that some particles may stray far away from their own as well as the neighbourhoods best observed positions (b_n & $\hat{b}_n$). Consequently, these particles will obtain large velocities, which causes erratic and sometimes diverging behavior since the particle gets pulled back and forth by ever larger velocities. A simple measure to prevent this from happening is velocity clamping [79]: One defines a maximum velocity magnitude $v_{\max}$ and then clips all velocity values v_n^i at any point to not exceed it, i.e.:

$$v_n^i = \begin{cases} v_{\max} & \text{if } v_n^i > v_{\max} \\ -v_{\max} & \text{if } v_n^i < -v_{\max} \\ v_n^i & \text{otherwise} \end{cases} \tag{2.56}$$

Inertia Weighting Another adaptation to the PSO algorithms is discussed in [80]: In order to obtain more robustly converging optimizations to well performing particles, the authors introduce a weighting factor w to control how strongly the prior velocity should be weighted when calculating the new one:

$$v_n^{i+1} = w \cdot v_n^i + c_1 \cdot r_1 \cdot [b_n^i - x_n^i] + c_2 \cdot r_2 \cdot [\hat{b}_n^i - x_n^i] \tag{2.57}$$

While the hyperparameters for PSO, i.e. c_1 & c_2, $x_{\min}$ & $x_{\max}$, $v_{\max}$, and w are of course problem dependent, the authors of [81] offer default values for some of them, proposing to set $c_1 = c_2 = 1.49618$ and $w = 0.7298$.

2.7 Variational Autoencoders

Variational Autoencoders have been proposed in [82]. The assumed problem setting is that N datapoints x_i from a dataset $\mathcal{D}$ are given, and have been generated by a stochastic process in which at first, some hidden latent variable z_i is sampled from a prior distribution $p^*_\omega(z)$, and then the actual sample x_i is produced by sampling from a conditional distribution $p^*_\omega(x|z)$. It is further assumed that both come from parametric families of distributions $p_\omega(z)$ & $p_\omega(x|z)$. VAEs propose a solution to three problems derived from this setting:

- approximating the parameters ω, which is essentially modeling the data generating process, enabling the generation of synthetic data samples $\hat{x}$ which come from the same distribution as the samples from the dataset.
- approximate posterior inference of z given a data sample x, enabling compression and representation learning.
- approximate marginal inference of x, i.e. the likelihood of datapoint x

VAEs consist of two networks: The encoder $q_\psi(z|x)$ as well as the decoder $p_\omega(x|z)$. Typically, the latent representation z has a much lower dimensionality than the actual data samples x, so that a bottleneck network architecture is constructed, as shown in Fig.

2.5. Ideally, one would like to maximize the marginal log likelihood $\log p_\omega(x_i)$ directly, however it is intractable. Instead, one can derive a lower bound on the marginal, as shown in [83] - we loosely follow the author's derivation in the following. One starts with the KL divergence (which is always non-negative) of the estimated and the true posterior distribution:

$$\mathrm{KL}(q_\psi(z|x), p(z|x)) = -\int p(z|x) \left(\frac{p(z|x)}{q_\psi(z|x)}\right) dz \geq 0 \tag{2.58}$$

By applying Bayes law, the posterior distribution $p(z|x)$ can be written using the marginal likelihood $p_\omega(x|z)$, as well as the probabilities of the latent as well as the sample $p(z)$ & $p(x)$, ultimately enabling to write down an inequality with the marginalized log likelihood on one side:

$$-\int q_\psi(z|x) \log \left(\frac{p(z|x)}{q_\psi(z|x)}\right) dz \geq 0 \tag{2.59}$$

$$-\int q_\psi(z|x) \log \left(\frac{p_\omega(x|z)p(z)}{q_\psi(z|x)p(x)}\right) dz \geq 0 \tag{2.60}$$

$$-\int q_\psi(z|x) \left[\log \left(\frac{p_\omega(x|z)p(z)}{q_\psi(z|x)}\right) - \log p(x)\right] dz \geq 0 \tag{2.61}$$

$$-\int q_\psi(z|x) \log \left(\frac{p_\omega(x|z)p(z)}{q_\psi(z|x)}\right) dz + \int q_\psi(z|x) \log p(x) dz \geq 0 \tag{2.62}$$

$$-\int q_\psi(z|x) \log \left(\frac{p_\omega(x|z)p(z)}{q_\psi(z|x)}\right) dz + \log p(x) \int q_\psi(z|x) dz \geq 0 \tag{2.63}$$

$$-\int q_\psi(z|x) \log \left(\frac{p_\omega(x|z)p(z)}{q_\psi(z|x)}\right) dz + \log p(x) \geq 0 \tag{2.64}$$

$$\log p(x) \geq \int q_\psi(z|x) \log \left(\frac{p_\omega(x|z)p(z)}{q_\psi(z|x)}\right) dz \tag{2.65}$$

Rewriting Eq. 2.65 yields the well known evidence lower bound (ELBO), which can be optimized as a tractable approximation to the marginalized $\log p(x)$:

$$\log p(x) \geq \int q_\psi(z|x) \log \left(\frac{p(z)}{q_\psi(z|x)}\right) dz + \int q_\psi(z|x) \log p_\omega(x|z) dz \tag{2.66}$$

$$\log p(x) \geq -\mathrm{KL}(q_\psi(z|x), p(z)) + \mathbb{E}_{z \sim q_\psi(z|x)} [\log p_\omega(x|z)] \tag{2.67}$$

The KL term can here be seen as a regularizing prior on $q_\psi(z|x)$, while the expectation of the log likelihood is a reconstruction term. When Gaussian priors are chosen for the latent variables $p(z)$ as well as for the approximate distribution $q_\psi(z|x)$, it is possible to find a closed form solution for the ELBO:

$$\mathrm{ELBO} = \frac{1}{2}\left[1 + \log \sigma^2 - \sigma^2 - \mu^2\right] + \mathbb{E}_{z \sim q_\psi(z|x)} [\log p_\omega(x|z)] \tag{2.68}$$

To optimize the VAE parameters ψ, ω, one can perform gradient descent on the negative ELBO.

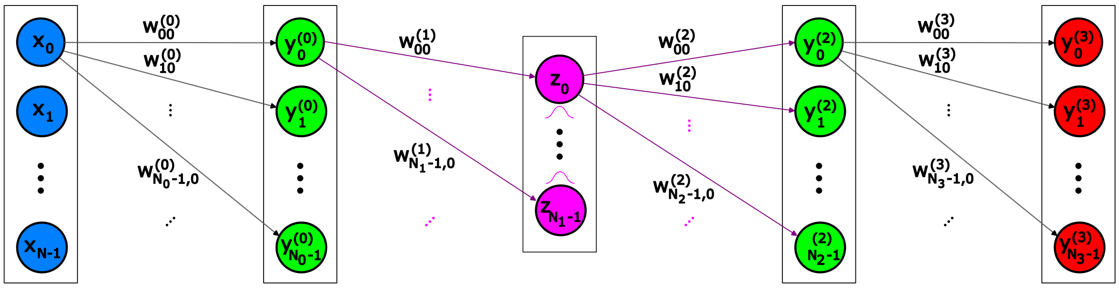


Figure 2.5: Schematic of a variational autoencoder with encoder parameters $\psi = \{\mathbf{w}^0, \mathbf{w}^1\}$ and decoder parameters $\omega = \{\mathbf{w}^2, \mathbf{w}^3\}$. $y_i^{(l)}$ denotes the output of the i^{th} neuron in layer l. z are the latent space parameters. It is assumed that N_1, the number of parameters in the latent layer is much smaller than the dimensionality of the in- and output or any other layer, and $N_3 = N$ since the output shape is the same as the input shape.

Conditional Variational Autoencoders In order to obtain conditional VAEs, one needs access to labels y_i corresponding to the data points x_i from the dataset. If available, one may concatenate these to the latent space for the reconstruction model, i.e. the decoder network now conditions not only on the latent encoding z_i, but also on the label y_i: $p_\omega(x_i|z_i, y_i)$, which enables the decoder model to later generate samples that belong to a previously specified class. E.g., if the dataset consisted of images of cats and dogs, with a regular VAE, one would be able to generate images of cats and dogs, but could not specify which of the two, which becomes possible with a conditional VAE.

3 Environments and Transition Models

3.1 Industrial Benchmark

The industrial benchmark (IB), introduced in [84], is a reinforcement learning environment motivated by industrial control problems, such as control of wind or gas turbines, chemical reactors, manufacturing plants, as well as applications from process industry, such as steel and paper production. Over the years, many academic benchmarks, such as the classic cartpole balance and swingup, mountain car, and pendulum benchmarks, as well as more advanced environments like Atari47 for video games, and MuJoCo for robotic locomotion, have been proposed. While they have greatly contributed to the advancement of the scientific field, they lack some of the characteristics commonly observed in real-world industrial applications like the ones listed above. For instance, the classical environments like cartpole, mountaincar, and pendulum have very low-dimensional state and action spaces as well as deterministic state transitions. While the robotic MuJoCo environments are much more complex since their state and action dimensionality is larger, their transition dynamics are still deterministic. The video games in the Atari47 suite feature even higher dimensional states, since policies need to be learned directly on pixel level, but their action space is discrete, and it is debatable how well results transfer from a video game to real-world applications.
Since the goal of RL research is to find methods that generalize well across all sorts of application domains, the IB has been proposed to bridge the gap between industrial data, that is often unavailable to academic research teams, and the already existing benchmarks that have been proposed in the literature. The industrial benchmark is not designed to replicate the behavior of a single known system, but is instead inspired by many different systems, and is designed to exhibit difficulties commonly encountered in many such applications. It thus features multiple delayed reward components, partially observable states, high dimensional as well as continuous state and action spaces, and state-dependent, multi-modal as well as heteroscedastic stochasticity. In the following, we provide a brief summary over the state and action space, as well as the transition dynamics as they were presented in [84].

The observable state of the IB is comprised of 6 variables: The setpoint (p), the three steerings velocity (v), gain (g), and shift (h), as well as the two conflicting reward components fatigue (f) and consumption (c). The three-dimensional actions can be seen as proposed changes to the steerings velocity, gain, and shift, i.e. $a = (\Delta v, \Delta g, \Delta h)$. Each action dimension is limited in $[-1, 1]$ and the corresponding state variables are limited in $[0, 100]$. The actions translate to updated steering values in the following way:

$$\begin{aligned} v_{t+1} &= \max(\min(v_t + d^v \Delta v, 100), 0) \\ g_{t+1} &= \max(\min(g_t + d^g \Delta g, 100), 0) \\ h_{t+1} &= \max(\min(h_t + d^h \Delta h, 100), 0) \end{aligned} \tag{3.1}$$

where $d^v = 1$, $d^g = 10$, and $d^h = 20\sin(15^0)/0.9 \approx 5.75$. The IB's reward is defined as a simple weighted combination of the two components fatigue and consumption:

$$r_t = -3f_t - c_t \tag{3.2}$$

The benchmark exhibits three sub-dynamics that are not directly visible to the RL agent since the corresponding state variables are not part of the observable state. These sub-dynamics are: Mis-calibration, operational cost, and fatigue.

Operational cost: The first sub-dynamic is called operational cost θ_t, and is at each time step t determined by the current values of velocity and gain, as well as the setpoint:

$$\theta_t = \exp\left(\frac{2p_t + 4v_t + 2.5g_t}{100}\right) \tag{3.3}$$

the operational cost is however not used further directly. Instead, it is meant to bring a delayed as well as blurred component into the reward calculation, so the convolved operational cost θ_t^c is defined as:

$$\theta_t^c = \frac{1}{9}\theta_{t-5} + \frac{2}{9}\theta_{t-6} + \frac{3}{9}\theta_{t-7} + \frac{2}{9}\theta_{t-8} + \frac{1}{9}\theta_{t-9} \tag{3.4}$$

Mis-calibration is the second sub-dynamic, and its behavior is influenced by the setpoint p and the shift h. The mis-calibration dynamics are meant to make the problem harder by requiring the optimal policy to oscillate in a specific frequency around a value of h influenced by p. The effective shift h^e is calculated using:

$$h_t^e = \max\left(-1.5, \min\left(1.5, \frac{h_t}{20} - \frac{p_t}{50} - 1.5\right)\right) \tag{3.5}$$

The effective shift in turn influences the three latent variables domain $\delta \in \{\textit{negative}, \textit{positive}\}$, response $\psi \in \{\textit{disadvantageous}, \textit{advntageous}\}$, and direction $\phi \in \{-6, -5, ..., 5, 6\}$, which also influence each other. The mis-calibration m is then calculated using a linearly biased Goldstone potential (we refer to [84] for a definition of ω):

$$m_t = -\alpha\omega(\rho_t^s, h_t^e)^2 + \beta\omega(\rho_t^s, h_t^e)^4 + \kappa\rho_t^s\omega(\rho_t^s, h_t^e) \quad \text{with} \quad \rho_t^s = \sin\left(\frac{\pi}{12}\phi_t\right) \tag{3.6}$$

The mis-calibration m_t, like the convolved operational cost θ_t^c, is however not directly observable. Instead, both are combined to yield the modified operational cost $\hat{c}_t$:

$$\hat{c}_t = \theta_t^c + 25m_t \tag{3.7}$$

Finally, the consumption c_t (which is an observable state variable as well as a direct influence on reward) is calculated by applying heteroscedastic observation noise to the modified operational cost:

$$c_t = \hat{c}_t + \mathcal{N}(0, 1 + 0.02\hat{c}_t) \tag{3.8}$$

The update rules for the latent variables domain, response, and direction are a bit more involved and provided in the following. We start by calculating potential values for domain and response:

$$\hat{\delta}_{t+1} = \begin{cases} \delta_t & \text{if } |h^e| \leq z \\ \operatorname{sgn}(h^e) & \text{else} \end{cases} \tag{3.9}$$

$$\hat{\psi}_{t+1} = \begin{cases} 1 & \text{if } \delta_t \neq \delta_{t+1} \\ \psi_t & \text{else} \end{cases} \tag{3.10}$$

I.e., the domain δ is *positive* (which is also the default initial value) if the effective shift h^e is a positive number, and *negative* if h^e is a negative number, except when h^e is in the so called safe zone, i.e. $h^e \in [-z, z]$, where $z = \sin(\pi \cdot 15/180)/2 \approx 0.1309$. The response ψ is *advantageous* if the domain has changed in the last time step, and otherwise stays the same as before. The potential value for direction is defined as:

$$\hat{\phi}_{t+1} = \phi_t + \Delta\phi_{t+1} \tag{3.11}$$

$$\text{with} \quad \Delta\phi_{t+1} = \begin{cases} -\operatorname{sgn}(\phi_t) & \text{if } |h^e| \leq z \\ 0 & \text{if } |h^e| > z \quad \text{and} \quad \phi_t = -6\hat{\delta}_{t+1} \\ \hat{\psi}_{t+1} \cdot \operatorname{sgn}(h^e) & \text{else} \end{cases} \tag{3.12}$$

This means that the direction ϕ will always take a step towards zero if the effective shift is in the safe zone. Otherwise it will take a step according to the product of the values of domain and response, except when direction is at the negative edge of its defined bounds ($\phi_t = -6\hat{\delta}_{t+1}$), where it will simply stay.

If the value for direction is after this update below -6 or above 6, i.e. would exceed its defined bounds, the response enters the *disadvantageous* state and the direction is turned towards zero:

$$\hat{\hat{\psi}} = \begin{cases} -1 & \text{if } |\hat{\phi}_{t+1}| \geq 6 \\ \hat{\psi}_{t+1} & \text{else} \end{cases} \tag{3.13}$$

$$\phi_{t+1} = \begin{cases} 12 - ((\hat{\phi}_{t+1} + 24) \bmod 24) & \text{if } |\hat{\phi}_{t+1} \geq 6 \\ \hat{\phi}_{t+1} & \text{else} \end{cases} \tag{3.14}$$

Finally, if effective shift h^e has returned to the safe zone and direction ϕ has completed a full circle back to zero, domain δ and response ψ are reset to their initial values *positive*

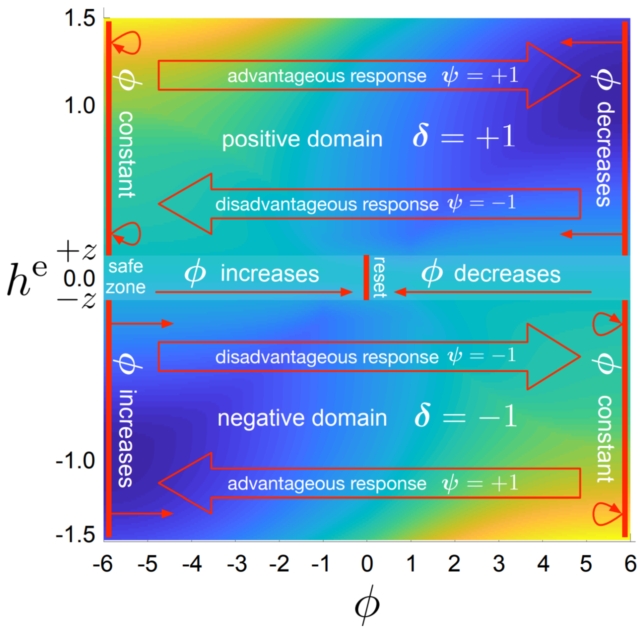

advantageous response $\psi = +1$
positive domain $\delta = +1$
disadvantageous response $\psi = -1$
ϕ constant
ϕ decreases
safe zone
ϕ increases
reset
ϕ decreases
disadvantageous response $\psi = -1$
negative domain $\delta = -1$
advantageous response $\psi = +1$
ϕ increases
ϕ constant
h^e
1.5
1.0
$+z$
0.0
$-z$
-1.0
-1.5
-6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6
ϕ

Figure 3.1: Visualization of the mis-calibration penalty (blue=low, yellow=high) depending on the value of the latent variables domain δ, response ψ, and direction ϕ, as well as effective shift h^e. Figure from [84].

and *advantageous*:

$$\delta_{t+1} = \begin{cases} 1 & \text{if } \phi_{t+1} = 0 \quad \text{and} \quad |h^e| \leq z \\ \hat{\delta}_{t+1} & \text{else} \end{cases} \tag{3.15}$$

$$\psi_{t+1} = \begin{cases} 1 & \text{if } \phi_{t+1} = 0 \quad \text{and} \quad |h^e| \leq z \\ \hat{\hat{\psi}}_{t+1} & \text{else} \end{cases} \tag{3.16}$$

An overview over the space of the latent variables and their influences is given in Figure 3.1.

Fatigue f is influenced by the same input variables as operational cost, namely the velocity v, the gain g, and the setpoint p. The idea of the fatigue sub-dynamics is to increase the penalty when the steerings velocity and gain are controlled to minimize the operational cost, thereby creating two opposing reward components that need to be carefully traded against each other. Fatigue is calculated as:

$$f = f^b(1 + 2\alpha)/2 \tag{3.17}$$

$$\text{with} \quad f^b = \max\left(0, \frac{3000}{5v + 100} - 0.01g^2\right) \tag{3.18}$$

α depends on two latent variables μ^v and μ^g as well as two noise terms η^v and η^g:

$$\alpha = \begin{cases} \frac{1}{1+e^{-\mathcal{N}(2.4,0.4)}} & \text{if } \max(\mu^v, \mu^g) = 1.2 \\ \max(\eta^v, \eta^g) & \text{else} \end{cases} \tag{3.19}$$

Both the latents as well as the noise terms depend on the effective velocity v^e and effective gain g^e, which are calculated by:

$$v^e = \frac{T^v(v,g,p) - T^v(0,100,p)}{T^v(100,0,p) - T^v(0,100,p)} \tag{3.20}$$

$$g^e = \frac{T^g(g,p) - T^g(100,p)}{T^g(0,p) - T^g(100,p)} \tag{3.21}$$

$$\text{with} \quad T^v(v,g,p) = \frac{g+p+2}{v-p+101} \tag{3.22}$$

$$T^g(g,p) = \frac{1}{g+p+1} \tag{3.23}$$

The latents μ^v and μ^g as well as the noise components η^v and η^g can then be calculated using:

$$\eta^v = \eta^{ve} + (1-\eta^{ve})\eta^{vu}\eta^{vb}v^e \tag{3.24}$$

$$\eta^g = \eta^{ge} + (1-\eta^{ge})\eta^{gu}\eta^{gb}g^e \tag{3.25}$$

$$\mu^v_t = \begin{cases} v^e & \text{if } v^e \leq 0.05 \\ \min(5, 1.1\mu^v_{t-1}) & \text{if } v^e > 0.05 \quad \text{and} \quad \mu^v_{t-1} \geq 1.2 \\ 0.9\mu^v_{t-1} + \frac{\eta^v}{3} & \text{else} \end{cases} \tag{3.26}$$

$$\mu^g_t = \begin{cases} g^e & \text{if } g^e \leq 0.05 \\ \min(5, 1.1\mu^g_{t-1}) & \text{if } g^e > 0.05 \quad \text{and} \quad \mu^g_{t-1} \geq 1.2 \\ 0.9\mu^g_{t-1} + \frac{\eta^g}{3} & \text{else} \end{cases} \tag{3.27}$$

where $\eta^{ve}, \eta^{vb} \sim \mathrm{Binom}(1, \mathrm{v^e})$ and $\eta^{ge}, \eta^{gb} \sim \mathrm{Binom}(1, \mathrm{g^e})$, while $\eta^{vu}, \eta^{gu} \sim \mathcal{U}(0,1)$.

3.1.1 Datasets

The datasets of the industrial benchmark that will be used throughout this thesis have initially been proposed in contribution [B]. They are generated by three different baseline policies $\pi \in \{\pi_{\text{bad}}, \pi_{\text{mediocre}}, \pi_{\text{optimized}}\}$ mixed with varying degrees of ε-greedy exploration, with $\varepsilon \in \{0.0, 0.2, 0.4, 0.6, 0.8, 1.0\}$. This means, that the effective generating policy π_β that generated a dataset is given by:

$$\pi_\beta(s) = \begin{cases} \mathcal{U}(-1,1)^3 & \text{in } (\varepsilon \times 100)\% \text{ of cases} \\ \pi_{\text{baseline}}(s) & \text{otherwise} \end{cases} \tag{3.28}$$

The three baselines are provided in the following formulae:

$$\pi_{\text{bad}} = \begin{cases} 100 - v_t \\ 100 - g_t \\ 100 - h_t \end{cases} \quad \pi_{\text{med}} = \begin{cases} 25 - v_t \\ 25 - g_t \\ 25 - h_t \end{cases} \quad \pi_{\text{opt}} = \begin{cases} -\tilde{v}_{t-5} - 0.91 \\ 2\tilde{f}_{t-3} - \tilde{p} + 1.43 \\ -3.48\tilde{h}_{t-3} - \tilde{h}_{t-4} + 2\tilde{p} + 0.81 \end{cases}$$

The optimized baseline was in this case obtained by a Reinforcement Learning algorithm called GPRL [13], which produces interpretable policies. The mediocre policy tries to move to a fixed point in the space of the three steerings, however ends up alternating around it in the dimensions gain and shift, since the factors d^g & d^h are not equal to one. It can be seen as a simple controller that is standardly deployed in many systems. The bad baseline deliberately steers the benchmark to an edge of the state space at which the reward is particularly bad - it is a rather unrealistic generating policy in reality, however it will be interesting to see how different algorithms cope when being exposed to such a controller.

Each of the 16 resulting datasets (18 - 2, since the three 100% exploration datasets would constitute the same setting) is comprised of 100 trajectories of length 1000, where each state is a concatenation of the past 30 observable variables of the IB (see how $\pi_{\text{optimized}}$ uses observations that lie in the past). The setpoint p is never altered and remains at a constant 70 throughout all episodes and datasets. Together, the datasets allow a comprehensive overview over how an offline RL algorithm behaves in the face of various encountered settings, such as narrow data distributions (0% exploration datasets), classic controllers (mediocre), RL policies (optimized), as well as undirected data (bad) and also larger amounts of exploration. The datasets have been made publicly available at `https://github.com/siemens/industrialbenchmark/tree/offline_datasets/datasets` under the Apache License 2.0.

3.2 MuJoCo

MuJoCo is a simulation environment for **Mu**lti-**Jo**int Dynamics with **Co**ntact [85], that is tailored specifically towards robotic simulation. A variety of robotic problems have been proposed and implemented in OpenAI gym [86], facilitating easy access for RL researchers. In this thesis, we will evaluate algorithms on a subset of the most popularly used MuJoCo benchmarks in RL literature: Swimmer, Hopper, and Walker2D. In comparison with the industrial benchmark, the MuJoCo environments feature no noise in their transitions, however they are still very high-dimensional, complex, and provide a different domain to evaluate algorithms in. See Figure 3.2 for a visualization of the three robots. In the following we briefly describe each environment and provide an overview over their action and observation spaces.

Swimmer The Swimmer has initially been introduced in [87]. The robot consists of three segments that are connected with two rotor joints to form a linear chain. It is placed in a 2D pool environment and its goal is to maximize the speed along the

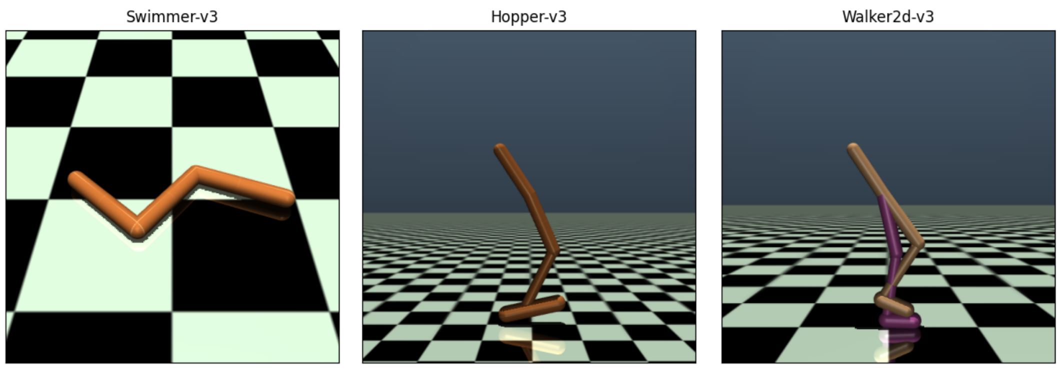


Figure 3.2: The three used MuJoCo environments, Swimmer, Hopper, and Walker2D.

x-axis by applying torques to the rotor joints and exploiting the fluids friction. The robots initial position is roughly the same for each environment start, just slight random deviations are applied. The reward consists of two components: (i) *forward_reward*, which is basically a reward for moving to the right as fast as possible, and is calculated as *forward_reward_weight* $\times$ (*x-position before action* - *x-position after action*)/*dt*, and (ii) *ctrl_cost*, which penalizes the robot for taking extreme actions: *ctrl_cost* = *ctrl_cost_weight* $\times sum(action^2)$. The final reward for each step is then calculated as **reward** = *forward_reward* - *ctrl_cost*. In the standard setting, the *forward_reward_weight* is equal to 1, while the *ctrl_cost_weight* is 1×10^{-4}. The default *dt* is 0.04, due to the default values for *frame_skip* (4) and *frametime* (0.01). Variations of the environment exist with N segments and $N-1$ rotors, however we will only consider the standard variant. The observation and action spaces are described in Tables 3.1 and 3.2.

Index	Description	Joint Type	Unit
0	Front tip angle	Hinge	radiants
1	First rotor angle	Hinge	radiants
2	Second rotor angle	Hinge	radiants
3	Tip velocity in x-direction	Slide	m/s
4	Tip velocity in y-direction	Slide	m/s
5	Front tip angular velocity	Hinge	rad/s
6	First rotor angular velocity	Hinge	rad/s
7	Second rotor angular velocity	Hinge	rad/s

Table 3.1: Observation space of the MuJoCo Swimmer environment.

Hopper The Hopper environment was introduced in [88], aiming to increase the complexity and number of state and action dimensions compared to classic control benchmarks like cartpole swingup or mountain car. The robot consists of four body parts: The torso (top), the thigh, the leg, and the foot. Its goal is to make hops forward

Index	Description	Joint Type	Unit
0	First rotor torque	Hinge	N/m
1	Second rotor torque	Hinge	N/m

Table 3.2: Action space of the MuJoCo Swimmer environment. Both actions are constraint to be between -1 and 1.

(along the x-axis) as fast as possible, and without falling over. The reward function is similar to the one of the Swimmer, with the *forward_reward* and *ctrl_cost* components designed exactly the same. Additionally, the Hopper is rewarded for every step that it is "healthy", i.e. has not fallen over, so that the final reward is calculated as: **reward** = *forward_reward* + *healthy_reward* - *ctrl_cost*. The *healthy_reward* is given whenever the Hopper height does not exceed the *healthy_z_range*, the torso angle does not exceed the *healthy_angle_range*, and none of the other state variables exceed the *healthy_state_range*. The default values for these ranges are $(0.7, \infty)$, $(-0.2, 0.2)$, and $(-100, 100)$. The default *healthy_reward* is 1, the default *ctrl_cost_weight* is 0.001, and the default *forward_reward_weight* is 1. The default *dt* is in this case 0.008, due to the default values for *frame_skip* (4) and *frametime* (0.002). An overview over the state and action space of the environment is provided in tables 3.3 and 3.4.

Index	Description	Joint Type	Unit
0	Torso z-coordinate (Hopper height)	Slide	meters
1	Torso (top) angle	Hinge	radiants
2	Thigh joint angle	Hinge	radiants
3	Leg joint angle	Hinde	radiants
4	Foot joint angle	Hinge	radiants
5	Torso x-axis velocity	Slide	m/s
6	Torso z-axis velocity	Slide	m/s
7	Torso angular velocity	Hinge	rad/s
8	Thigh joint angular velocity	Hinge	rad/s
9	Leg joint angular velocity	Hinge	rad/s
10	Foot joint angular velocity	Hinge	rad/s

Table 3.3: Observation space of the MuJoCo Hopper environment.

Walker2D The Walker2D environment extends the Hopper environment by adding a second leg. The environment thus consists of seven parts: One torso, and two instances each of thigh, leg, and foot. Consequently there are also six rotor joints connecting the components, doubling the action space dimensionality. Again, the goal is to move as fast as possible along the x-axis, without falling over and thus prematurely ending the episode. The reward is calculated the same as for the Hopper environment, i.e. **reward** = *forward_reward* + *healthy_reward* - *ctrl_cost*. Also most of the de-

Index	Description	Joint Type	Unit
0	Thigh rotor torque	Hinge	N/m
1	Leg rotor torque	Hinge	N/m
2	Foot rotor torque	Hinge	N/m

Table 3.4: Action space of the MuJoCo Hopper environment. Both actions are constraint to be between -1 and 1.

fault values are the same, except for the ranges in which the Walker is considered to be "healthy": The *healthy_z_range* is $(0.8, 2)$, the *healthy_angle_range* is $(-1, 1)$, and the *healthy_state_range* for the remaining observation dimensions is $(-\infty, \infty)$. The state and action spaces of the Walker2D environment are provided in Tables 3.5 and 3.6.

Index	Description	Joint Type	Unit
0	Torso z-coordinate (Walker height)	Slide	meters
1	Torso (top) angle	Hinge	radiants
2	Right thigh joint angle	Hinge	radiants
3	Right leg joint angle	Hinde	radiants
4	Right foot joint angle	Hinge	radiants
5	Left thigh joint angle	Hinge	radiants
6	Left leg joint angle	Hinde	radiants
7	Left foot joint angle	Hinge	radiants
8	Torso x-axis velocity	Slide	m/s
9	Torso z-axis velocity	Slide	m/s
10	Torso angular velocity	Hinge	rad/s
11	Right thigh joint angular velocity	Hinge	rad/s
12	Right leg joint angular velocity	Hinge	rad/s
13	Right foot joint angular velocity	Hinge	rad/s
14	Left thigh joint angular velocity	Hinge	rad/s
15	Left leg joint angular velocity	Hinge	rad/s
16	Left foot joint angular velocity	Hinge	rad/s

Table 3.5: Observation space of the MuJoCo Walker2D environment.

For reproducibility, note that we use the third version ("-v3") of all of the environments.

3.2.1 Datasets

To extract datasets from the three MuJoCo benchmarks, we use the "imperfect demonstrations" setup introduced in [8]. For each of the environments, we train a DDPG agent [4] through online interaction for 1,000,000 time steps, so that it can then act as an "expert" policy π_{DDPG}. To collect the datasets, we then evaluate the expert agent for

Index	Description	Joint Type	Unit
0	Right thigh rotor torque	Hinge	N/m
1	Right leg rotor torque	Hinge	N/m
2	Right foot rotor torque	Hinge	N/m
3	Left thigh rotor torque	Hinge	N/m
4	Left leg rotor torque	Hinge	N/m
5	Left foot rotor torque	Hinge	N/m

Table 3.6: Action space of the MuJoCo Walker2D environment. Both actions are constraint to be between -1 and 1.

another 1,000,000 time steps and add two different sources of noise: (i) In 70% of cases, we add Gaussian noise with a standard deviation of 0.3 on top of the action proposed by the expert before it is executed. (ii) In the other 30%, we pick an action uniformly at random. In the resulting (state, action, reward, future state)-tuples, the expert policy is thus effectively buried under a lot of noise, which should make it hard for an offline algorithm to recover its original performance.

$$\pi_\beta = \begin{cases} \pi_{DDPG} + \mathcal{N}(0, 0.3)^d & \text{in 70\% of cases} \\ \mathcal{U}(-1, 1)^d & \text{else} \end{cases} \tag{3.29}$$

Equation 3.29 formally shows how the generating policy π_β to collect the datasets is formed. Here, d denotes the action space dimensionality, i.e. 2 for Swimmer, 3 for Hopper, and 6 for Walker2D. During both random sampling phases, each action dimension is sampled independently from the others.

3.3 Transition Models

Throughout this thesis, we will use transition models $\hat{T}$ to imitate the true dynamics T of the environment we are trying to solve. To this end, we use different kinds of neural network models to predict both the future state and the reward, given the current state of the environment as well as the control action provided by the policy. While in tabular datasets, boosted decision trees still play a large role, neural networks are known to perform well in the sorts of environments that we aim to solve: Continuous, smooth, and in the case of MuJoCo also deterministic system dynamics. While different problems may require other model classes, like Gaussian Processes (GPs) [15, 89, 90] or Bayesian neural networks (BNNs) [11, 91], that would additionally to the prediction also provide a quantification of the associated uncertainty, we will not consider them in our experiments, since our focus lies on the regularization of the policy, and not on the modeling of the environment. Further, the environments considered can already be solved with the multi-layer perceptron (MLP) and recurrent neural network (RNN) models presented in this section.

3.3.1 Feedforward MLPs

Artificial neural networks (ANNs) are universal function approximators [92, 93], meaning that given enough capacity, they are able to approximate any given function arbitrarily accurate. The most common form of ANN is the multi layer perceptron (MLP), that was proposed in [94]. In an MLP, multiple layers, each consisting of multiple neurons are hierarchically connected in a directed acyclic graph, where a connection exists from any neuron of one layer to every neuron of the immediately following layer. Each layer only has one following layer and there are no skip connections. See Figure 3.3 for a visualization. Each neuron effectively realizes a logistic regression, i.e.:

$$y = \sigma\left(\sum_{i=0}^{N-1} x_i w_i + b\right) \tag{3.30}$$

Were $\mathbf{x}$ is an N-dimensional input vector with components $x_0, ..., x_{N-1}$, σ is the sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$, $\mathbf{w}$ is the weight vector of the logistic regression model with components $w_0, ..., w_{N-1}$ (one for each input dimension), and b is the bias.

Similarly, in a neural network, the output of a single layer l, $y^{(l)}$ can be computed based on the output of the previous layer $y^{(l-1)}$ as follows:

$$y_i^{(l)} = f\left(\sum_{j=0}^{N^{(l-1)}} w_{ij}^{(l)} y_j^{(l-1)} + b_i^{(l)}\right) \tag{3.31}$$

where $N^{(l)}$ is the number of neurons in layer l, $w_{ij}^{(l)}$ is the weight connecting neuron j in layer $l-1$ with neuron i in layer l, $b_i^{(l)}$ is the bias of neuron i in layer l, and f is a nonlinear function. While the sigmoid σ could be used, modern neural networks usually use the ReLU nonlinearity [95], or one of its variants, e.g. Leaky ReLU[96], ELU[97], etc. due to their mitigation capabilities regarding the exploding and vanishing gradient problems [98, 99]. For a visualization of a simple feed-forward MLP, see Fig. 3.3

Deterministic neural networks for state predictions are usually optimized by minimizing the mean squared error (see Eq. 2.31). The set of all weights $w_{ij}^{(l)}$ and biases $b_i^{(l)}$ is usually represented by the set of all trainable parameters $\phi = \{w_{ij}^{(l)} | \forall l, i, j\} \cap \{b_i^{(l)} | \forall l, i\}$. The MSE can be calculated over all transition samples that have been collected in the dataset $\mathcal{D}$:

$$\mathcal{L}(\phi) = \frac{1}{K} \sum_{k=0}^{K-1} \sum_{i=0}^{d-1} (s_k'^i - y_i^{(L-1)})^2 \tag{3.32}$$

where K is the number of samples contained in $\mathcal{D}$, d is the dimensionality of a state, $s_k'^i$ is the future state dimension i as it was recorded in the dataset interaction with the index k, and $y_i^{(L-1)}$ is the output of the final layer of the neural network in the same dimension. We assume that the input of the neural network, i.e. the nodes going into the first layer,

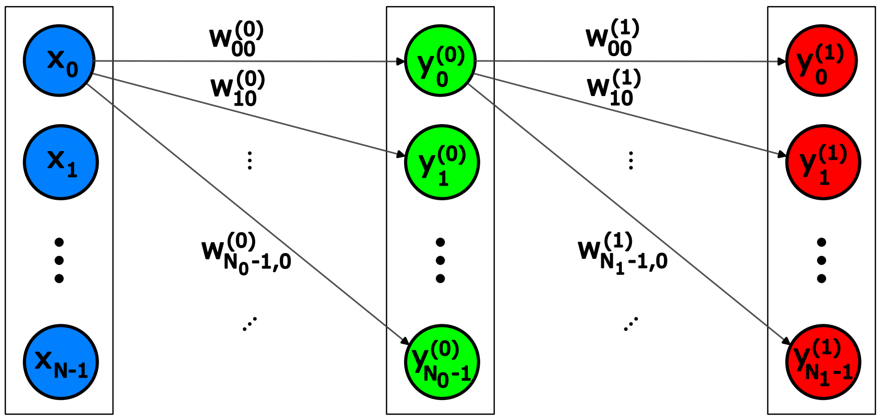


Figure 3.3: Visualization of a standard feed-forward MLP with an input layer (blue), a single hidden layer (green), and an output layer (red). N_l denotes the number of neurons in layer l.

were the concatenation of the recorded prior state and the associated action: $[s_k, a_k]$.

Since the loss function and every component of the neural network is differentiable, we can compute the analytical gradient of the loss w.r.t. the network parameters $\frac{\partial L(\phi)}{\partial \phi}$ and use gradient descent to optimize the parameters ϕ to minimize the loss function $\mathcal{L}(\phi)$. In the experiments in the next Chapters, we always use the Adam optimizer [100] to do so, i.e. we use the following update rules to compute the next set of network parameters based on the current ones and their gradients:

$$\phi_{t+1} = \phi_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon} \tag{3.33}$$

where η is the learning rate. Exponentially decaying estimates of the mean and squared gradient maintained in the variables m_t and v_t:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \frac{\partial L(\phi)}{\partial \phi} \tag{3.34}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) \frac{\partial L(\phi)}{\partial \phi} \tag{3.35}$$

Since m_0 and v_0 are initialized with zeros, unbiased estimates are computed using:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \qquad \hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{3.36}$$

Note that m_t, $\hat{m}_t$, v_t, and $\hat{v}_t$ are vectors, containing one element for each individual parameter in ϕ, while β_1, β_2, and ε are scalars, with default values of 0.9, 0.999, and 10^{-8}.

We take additional measures to improve the accuracy of the transition models. **Weight normalization** [101] is a technique that decouples the neural weight directions from their norms to speed up convergence. To this end, the weight vector $\mathbf{w}$ of a single neuron is reparameterized by a parameter vector $\mathbf{v}$ and a scaling factor g:

$$\mathbf{w} = \frac{g}{||\mathbf{v}||}\mathbf{v} \tag{3.37}$$

This reparameterization fixes the norm of $\mathbf{w}$ independently of $\mathbf{v}$, i.e. $\mathbf{w} = g$. Instead of training $\mathbf{w}$, the new parameters $\mathbf{v}$ and g are trained directly via gradient descent.

3.3.2 Recurrent Neural Networks

In sequence modeling, such as natural language in texts or state transitions in MDPs, the next sequence element often does not only depend on the current one, but can also depend on one or multiple elements further in the past. For example, the industrial benchmark features the state dimension operational cost, which includes heavily delayed components and influences the reward. In order to address this issue, different measures can be taken: (i) One may concatenate the G past states together and provide this history of G time steps to the MLP transition model as an input instead of just the last one. (ii) Alternatively, one may train a recurrent neural network, only provide the latest state (and action), but have the network learn a hidden state that is carried over in every forward pass and contains the information about the past that is necessary and helpful to predict the future ([102] is one of the central works analyzing RNNs for system identification). Both are valid approaches, however (i) has some shortcomings: Augmenting the input space of the MLP requires knowledge about the environment, i.e. how many steps into the past and which state dimensions could be important. Further, it severely increases amount of parameters in the first layer (depending on how many steps are added), while at the same time not benefiting from the underlying symmetry - weights concerning the same state dimensions in different time steps should be similar, but each weight needs to be learned separately. In (ii), the amount of parameters is also larger than in the single-state-MLP, but the number is likely smaller than in (i), and it may benefit from weight symmetry since for each state embedding, the same weights are used. RNNs have consequently been used for reinforcement learning as well as system identification [103, 104, 105]. See Fig. 3.4 for a visualization of a simple RNN.

A single recurrent layer in its simplest form works as follows: Instead of having only a single input (usually the output of the prior layer), a recurrent layer has two inputs. One is the output of the prior layer (or the actual input, i.e. the current state and action), and one is its own output that was computed in the prior step, which was simply saved for the next one. When we denote by $h_t^{(i)}$ the i^{th} dimension of the hidden state (i.e. output of the recurrent layer), the computation of the next hidden state can be written as follows:

$$h_{t+1}^{(i)} = f\left(\sum_j w_{ij}^{\text{hid}} h_t^{(j)} + b_i^{\text{hid}} + \sum_j w_{ij}^{(l)} y_j^{(l-1)} + b_i^{(l)}\right) \tag{3.38}$$

where w^{hid}, b^{hid}, $w^{(l)}$, and $b^{(l)}$ are the two sets of weights and biases for the hidden and embedding parts of the recurrent layer. Further we assume the previous layer to be the

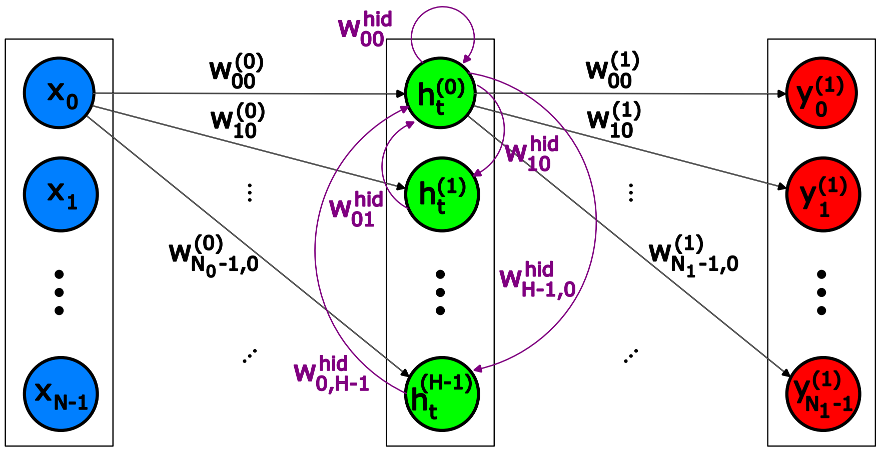


Figure 3.4: Simple recurrent neural network schematic with a hidden state of size H. The additional weights compared to the MLP in Fig. 3.3 are highlighted in purple.

one with the index $l-1$, so that $y^{(l-1)}$ is the result vector from the previous layer that is going into the recurrent layer. The above may be rewritten in vectorized notation as:

$$h_{t+1} = f(w^{\text{hid}} h_t + b^{\text{hid}} + w^{(l)} y^{(l-1)} + b^{(l)}) \tag{3.39}$$

h_0 is usually initialized with zeros and $f(\cdot)$ is a nonlinearity (usually tanh, sometimes ReLU). A prediction can then be made by building up the hidden state over a number G of history steps, and then using the resulting hidden state h_G to predict the next state s_{G+1}, i.e.:

$$s_{G+1} = f(w^{(l+1)} h_G + b^{(l+1)}) \tag{3.40}$$

The prediction can again be optimized via gradient descent on the mean squared error and may as well reflect the delta state instead of the state directly.

In the context of RNNs, it has been shown that it is beneficial to train the transition models on multiple future steps instead of just a single one. It appears that models that are trained for just a single step often generalize poorly when they are rolled out for longer into the future, so the prediction error may be accumulated over F future steps:

$$L(\phi) = \sum_{i=0}^{F-1} (s'_{t+i+1} - \hat{T}_\phi(\hat{s_{t+i}}, a_{t+i}, h_{t+i-1}))^2 \tag{3.41}$$

where $\phi = \{w^{\text{hid}}, b^{\text{hid}}\} \cap \{w^{(l)}, b^{(l)} | \forall l\}$, $\hat{T}$ has two output components $h_{\text{new}} = h(\hat{T}^h_\phi(s, a, h_{\text{old}}))$ (by means of Eq. 3.39) and $\hat{s}' = s(\hat{T}_\phi(s, a, h_{\text{old}}))$ (Eqs. 3.39 and 3.40). The hidden state is built up over the first G steps using the real observed states, i.e. $h_t = h(\hat{T}(s_t, a_t, h_{t-1}))$, and the predicted states $\hat{s}$ are used afterwards for prediction as well as maintaining the hidden state.

4 Dataset Quality Estimation

Offline RL literature is mostly concerned with finding generally applicable algorithms that can train well performing policies given a single static dataset of past interactions. This Chapter takes a different perspective, and instead discusses different approaches how to select promising datasets for offline RL from a pool of provided datasets.

4.1 Problem Setting

Offline RL algorithms have over the past couple of years significantly improved the applicability of reinforcement learning techniques, as they make learning of control strategies possible without any interaction with the actual environment. Before, online RL algorithms had shown tremendous success on many simulated tasks or video games [2, 3, 106], however application to physical systems was always limited due to safety concerns (most online algorithms use some form of randomness to perform exploration), scalability issues (online algorithms often train on many millions of interactions, and use massive parallelism in order to make learning possible in a reasonable time, which is hardly possible to come by in physical systems), and opportunity cost (during training, the system can likely not be productively used). Now, with offline RL algorithms, which regularize the policy to stay in the support of the previously collected data, all three problems have tremendously improved: Offline RL features no exploration, and algorithms exist that derive deterministic policies, so that practitioners can closely examine a trained policy before deployment, greatly improving the safety situation. Since the environment is not used during training, both the scalability and the opportunity cost problems are practically mitigated. Offline RL thus has the potential to actually bring reinforcement learning to the real world.

In the offline RL setting, a new dilemma arises: Many companies have massive amounts of data about many of their systems, for which they would now like to find better control strategies with offline RL. However, how do they choose which dataset, i.e. which system to start with? The easy solution would be to just do all of them, however in practice there exist obvious limitations on the computational as well as the human resources side - data needs to be curated, models need to be trained, hyperparameters need tuning, sanity checks need to be performed, etc. On top of that, there are likely at least a few test runs necessary on the real system before a policy can be used in production. Since machine learning experts, domain experts, production environment stops for testing, potentially damaged equipment, and compute cost together constitute a significant up front investment, it would be great to be able to select the most promising datasets and applications among the available ones in order to increase chances of earning the investment

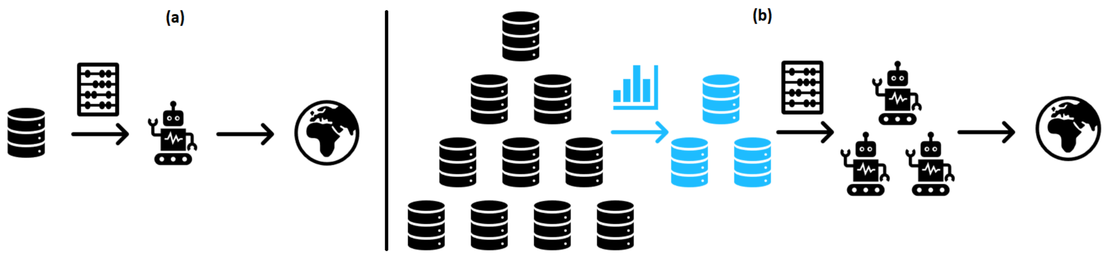


Figure 4.1: Schematic of the newly proposed dataset selection problem in offline RL: In Figure (a), the standard offline RL process is shown, where a single dataset is provided and an agent is distilled from it via some learning algorithm, before it is finally deployed in the real environment. In Figure (b) we are provided with many datasets from different systems, introducing the problem of how to select the most promising ones for offline RL. In a new step, a subset of the datasets is selected by calculating quality metrics for all of them. Finally, the selected datasets are again used to train policies which can be deployed.

back. Furthermore, offline RL algorithms for continuous control do not come with any meaningful guarantees - no algorithm can with certainty say that it will improve over the controller that was running during generation of the dataset by any percentage - they cannot even guarantee that they will not become worse. This makes it even more important to select the most promising datasets to make success as likely as possible: In real projects, it is usually crucial when a new technology should be applied, that an initial success story can be generated. If the first few datasets / systems that are chosen to be tackled by offline RL turn out to perform worse than the previous controller, trust in the technology might be lost, making future application of it much less likely.

For the example to work in this context, we assume now that the operator from before not only has a wind turbine, but also a hydroelectric power station, a nuclear power plant, a gas power plant, and possibly other kinds of power generating facilities. While the operating company has datasets from all of the power plants it owns, it only has a tiny data science department consisting of a single person. Also, developing and deploying an offline RL based control solutions for any of the plants requires an up front investment, due to compute and expert time as well as some time during which the plant cannot be (as) productively used. Due to the costs and since RL technology is new to the company, they want to start optimizing only the power plant that has the highest chances of improving its yield and thus the revenue of the company. The question thus becomes: which dataset among the available ones is most suitable for offline RL, i.e. which has the highest data quality?

A little more formally, we expect to be provided with N datasets of lengths $L_i,\ i \in 0, ..., N-1$, each containing state, action, reward, and next state tuples resulting from past interactions with some system: $D_i = \{s_j, a_j, r_j, s'_j\},\ j \in 0, ..., L_i - 1$. Furthermore, each task is associated with a specific cost of deployment C_i reflecting opportunity cost when the system is not in productive use, domain experts time, and materials used. All tasks also share a fixed cost F, reflecting compute cost and ML expert time. When we select a dataset to train a policy and then test and deploy it, we need to pay the corresponding $C_i + F$ and observe a change in return $\Delta R = R^{\text{data}} - R^{\text{new}}$. If this change in return in positive, it has the potential to offset the investment over some horizon H: $R^{\text{meta}} = \sum_{t=0}^{H} \gamma^t \Delta R - (C_i + F) \geq 0$. Ideally we would like to select the datasets so that R^{meta} is maximized, however we do not want to pay $C_i + F$ for all of the datasets, i.e. simply try them all out. We term this the *dataset selection problem*.

4.2 Related Work

In supervised machine learning projects, researchers and practitioners have already recognized that data quality is crucially important to obtain the best possible results. Even though most of the corresponding literature focuses on improving over existing or proposing new algorithms, and earlier steps in the machine learning pipeline are traditionally getting less attention, a variety of methods have been developed to define, assess and improve data quality. Quality of datasets is classically defined along the dimensions accuracy, completeness, consistency, and timeliness [107, 108]. When examining accuracy, the concept of source trustworthiness [109] is important: [110] predict the same class labels with features from two different sets of sensors that represent the same abstract concepts, however the prediction accuracy is much lower with the low quality sensor features. Similarly [111] use different camera systems to predict water turbidity and find large gains in accuracy when the feature quality is improved. Finally, [112] find large prediction quality differences when using different sensors for depth estimation. Even when trustworthy sources are unavailable, techniques exist that combine many untrustworthy sources together to form a dataset which achieves better predictions [113]. Aside from accuracy, data validation tools such as Deequ, DaQL, DuckDQ, and TFX [114, 115, 116, 117, 118] have been developed to address issues regarding completeness, consistency, and timeliness by having practitioners express their beliefs about how the data could possibly look like. Data cleaning techniques based on anomaly detection may even improve data consistency without requiring any expert knowledge [119, 120, 121]. In the following we investigate how to measure data quality for offline RL, since it appears likely that the results in e.g. [110, 111, 112] extend to the reinforcement learning domain and that data quality would likely also improve the performance of the policies that can be found using offline RL algorithms. In order to extract information about the usability for offline RL, we mainly focus on two very basic quality dimensions of the datasets:

- the performance reached by the generating policy during the best trajectories compared to the average

- a rather classical notion of completeness often used in databases literature, where it is defined as "ability of an information system to represent every meaningful state of a real-world system" [107, 122]. For RL algorithms, this means that the performance estimators will likely be able to evaluate the policy's return accurately anywhere in the state-action space

4.3 Relevant Dataset Features

In this section, we aim to find generally applicable indicators to assess dataset quality in terms of suitability for use in offline RL to derive policies with as large as possible R^{meta}. Essentially, we deem two different basic scenarios as most beneficial for large performance gains in offline RL:

- **High return deltas:** If the dataset contains trajectories with very high returns, but the average returns are comparatively low, we would expect from an offline RL algorithm that it can at least copy most of the good behaviors and avoid those that yielded lower returns. This scenario essentially connects to the behavior cloning portion contained in most offline RL algorithms. An exception to this assumption would arise if the difference in returns is entirely due to environment stochasticity.
- **Large action stochasticity:** While one component of offline RL loss functions usually contains a behavior cloning style regularizer, the other one aims to estimate the performance of the trained policy. The question is how accurately this component, no matter whether value function or transition model, can assess the returns. Intuitively, this becomes much easier, the more exploration is contained in the dataset: Essentially, the question is in this case how stochastic the action distribution in the dataset is. The closer the action distribution is to uniform random, the higher the entropy and the more information can be assumed to have been gathered and learned by the performance estimating component.

We aim to find simple indicators that are generic and applicable not only to datasets with specific characteristics, but rather anything from high dimensional and continuous state and action spaces to low dimensional and discrete ones.

4.3.1 Estimating selective behavior cloning improvement

Identifying whether high return trajectories are present in the datasets is relatively simple: We can just sum up the discounted rewards of each trajectory and inspect the resulting distribution:

$$R^{\text{data}} = \left\{ \sum_{t=0}^{H_i} \gamma^t r_t^{(i)} \mid i \in 0, ..., I-1 \right\} \tag{4.1}$$

where the dataset consists of I trajectories, H_i is the horizon of the i^{th} trajectory and $r_t^{(i)}$ is the reward at time step t in trajectory i. For practitioners it could be interesting to inspect the entire distribution of R^{data}, however for the sake of simplicity we will limit ourselves to using the mean and maximum to come up with a quality metric. Intuitively, if the offline RL algorithm copies more of the good behavior than of the bad, it should easily be able to outperform the mean return present in the data. The hope would be to get close to the maximum performance or even exceed it. As different datasets and tasks have different reward scales, we propose in indicator that normalizes the *expected return improvement* (ERI) as given by the difference between mean and max performance:

$$\text{ERI} = \frac{\max(R^{\text{data}}) - \text{mean}(R^{\text{data}})}{\text{mean}(R^{\text{data}}) + \varepsilon} \tag{4.2}$$

In order for the ERI indicator to yield consistent values, we need to assume returns to be bounded by zero from below. If this is not the case, we need to normalize the values in R^{data} by adding the minimum return first. We assume that this quantity is either basic knowledge about the environment that can be provided by a domain expert or can be estimated from the data.

Looking at the ERI indicator as defined above, one might wonder whether it is not overly simplistic, since offline RL algorithms should of course be able to do more than just copy well performing trajectories and ignore others. Instead, they should be able to intelligently combine skills, i.e. behaviors on a sub-trajectory level, that were observed in different trajectories. ERI cannot account for such smart "stitching" together of trajectory parts, but an extension would of course be possible by considering to define R^{data} over smaller horizons H than the full trajectory length. Since it is difficult to predict how offline algorithms will combine sub-trajectory level skills, and since finding the correct H is a hard to tackle hyperparameter problem, we will however not consider this extension here.

4.3.2 Estimating the Amount of Exploration Present in the Data

Estimating the amount of exploration in the dataset is not quite as straightforward as the reward improvement estimation before. Essentially, we would like to know how well the state-action space has been explored by the dataset-generating policy. The idea is that if it has been satisfactorily explored, the performance estimation method of choice will likely be able to assess the performance of more policy candidates accurately, which will help to find a well performing policy. One could try to estimate e.g. the density of the states that have been visited in the dataset, work with pseudo-counts [123], or try to calculate the convex hull of the visited states in order to answer the question of how well the state space has been explored, however all of these options are rather cumbersome and especially hard to scale to high-dimensional state spaces. Since exploration is usually induced by an algorithm by taking random actions instead of the currently thought best option in terms of return, we propose to estimate how random the action distribution of the generating policy was. That way, the indicator for exploration will remain oblivious of the state space, which makes it much easier to transfer it to a range of tasks & datasets. We do not assume the generating policy to be known, since we often do not have a closed form representation of it in practice. Instead, human interactions with the system might be mixed together with a variety of deterministic controllers that changed slightly over time, noisy controllers, etc. Many sources of stochasticity and thus exploration exist, including deterministic policies that live in a different state space than the one we can observe (e.g. a different policy is used in the morning than in the evening, and time of day is not a state dimension). In order to estimate the action stochasticity, we train a tanh transformed Gaussian policy, whose mean and standard deviation are given by neural networks f_μ and f_σ, both operating on the same features provided by another neural network f_z:

$$a \sim \tanh(\mathcal{N}(f_\mu(f_z(s)), f_\sigma(f_z(s))^2)) \tag{4.3}$$

$$\theta^* = \arg\min_\theta - \sum_i \sum_t \log\ p(a_t^{(i)} | \theta, s_t^{(i)}) \tag{4.4}$$

where θ is the joint set of parameters of f_μ, f_σ, and f_z. The optimization of the Gaussian policy can be performed with gradient descent, similarly to the Gaussian model via the negative log likelihood as in Eq. 2.29. After training has conceded we define the action stochasticities AS using the predicted standard deviations on the dataset:

$$\mathrm{AS}_f = \{f_\sigma(f_z(s)) | s \in \mathcal{D}\} \tag{4.5}$$

Low values in AS_f correspond to rather deterministic behavior, while higher values correspond to good exploration. Similarly to the ERI feature, it can be beneficial to inspect the entire distribution of AS_f, however for simplicity, we will derive the *expected action stochasticity* (EAS) indicator by taking the mean over AS_f. This might overestimate the exploredness in some regions of the state space, but as a heuristic it should suffice:

$$\mathrm{EAS} = \mathrm{mean}(\mathrm{AS}_f) \tag{4.6}$$

We further experimented with other ways to quantify exploration contained in the dataset. [124] claim that environment stochasticity could be sufficient to explore the state space, so an option could be to measure stochasticity in a trained transition model instead of a policy model. We were however unable to show a meaningful connection between the state transition stochasticity and the performance of an offline RL algorithm and would also argue that it makes little sense intuitively: If a deterministic policy runs on a stochastic environment and consequently sees a plethora of states, then a learner cannot really benefit from this, since it cannot infer how to get to the different states (it was purely chance) and it has only seen a single action per state, so it doesn't know about different options to take. We thus argue that to measure exploration it is mostly important to look at action space stochasticity.

4.4 Estimating Suitability for Offline Learning

We analyze how the derived indicators ERI and EAS can help to select promising datasets by evaluating them on the industrial benchmark datasets proposed in Chapter 3 and

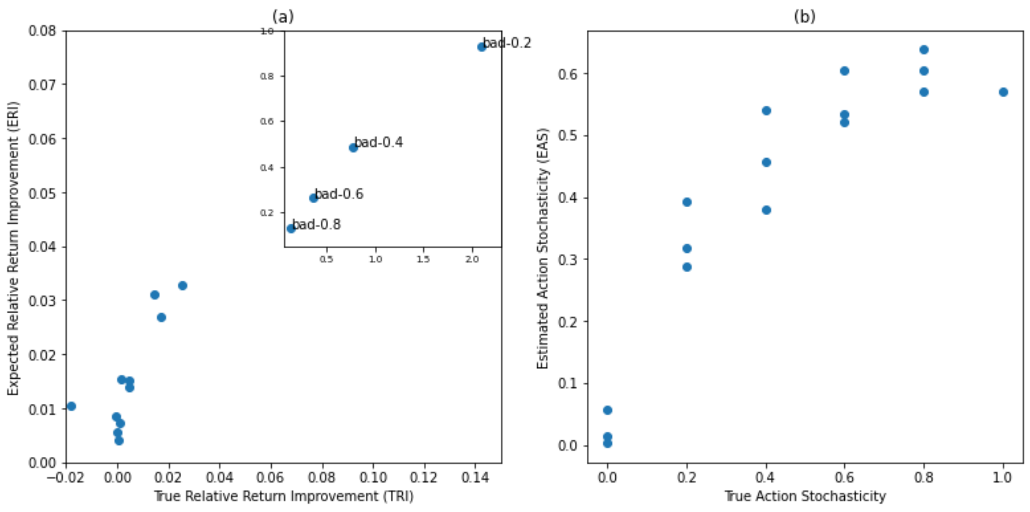


Figure 4.2: Data quality dimensions: Comparison of ground truth values and derived estimators for return improvement and action stochasticity. Figure (a) shows expected relative return improvement (ERI) over the true relative return improvement (TRI). Figure (b) shows estimated action stochasticity (EAS) over true actions stochasticity. Rank correlations are provided in Table 4.1

.

comparing how they align with actual offline RL algorithm performances as published in contribution [B]. For the sake of these experiments, we pretend that each dataset represents a completely different environment, since otherwise one could simply pool all of the data together and obtain the obviously best dataset for offline RL since it contains all the information. We analyze how well the ERI and EAS features help predict the true relative return improvement (TRI):

$$\mathrm{TRI} = \frac{R^{\mathrm{algo}} - \mathrm{mean}(R^{\mathrm{data}})}{\mathrm{mean}(R^{\mathrm{data}})} \tag{4.7}$$

where R^{data} is defined as for ERI and R^{algo} is the reported performance achieved by the offline algorithm based on the dataset. As mentioned in Section 4.3.1, we need to lower bound the return values by zero. Since all returns on the IB are negative, we add 350 to each of the returns to move them all to the positive domain (-327 is the lowest observed return in any of the datasets, but potentially it can be even lower).

In Figure 4.2 we plot EAS and ERI against their ground truth values (for the true action stochasticity, we use the percentage of actions that were chosen at random). While the datasets do not reside on a line (which would not be expected), their ranks in the ground truth values can be rather well estimated using the derived indicators. The ranks and rank correlations of EAS and ERI with TRI are presented in Table 4.1. Generally we can observe that the EAS feature has a lower rank correlation with the true performance gain than the ERI feature. This is unsurprising since the effect of exploration on downstream RL tasks is a little less direct: Even if a good amount of exploration is contained in the data, it is unclear whether the most promising regions of the state-action space have been seen. We also observe that for the datasets generated with the bad baseline, EAS is actually inversely correlated with performance gains. This is due to the fact that the bad baseline is so bad that random actions perform much better, which leaves less room for improvement by the RL algorithm. We thus also examine the rank correlations when the bad datasets are taken out of the equation - then, EAS is similarly predictive of performance increases as ERI is (rank correlation improves from 0.13 to 0.83).

In order to combine both features together, we derive a *combined offline indicator* (COI) by combining the ranks that the two features yield 2:1 (i.e. for the dataset bad-0.8, the ERI rank is 12 and the EAS score is 14, so the COI score is $2 \times 12 + 14 = 36$, which is the highest among any of the datasets, making it rank 15). With a rank correlation of 0.75 for all, and 0.86 for the more realistic set of datasets without the bad ones, we find that COI can be an effective tool for selecting datasets for offline RL with minimal effort / up-front investment. While COI does not select the best dataset first, it almost completely correctly identifies the top half of datasets, with the exception of bad-0.0, however this dataset can be considered an outlier - its TRI is only so good because anything is better than sticking to the bad policy. So in a fictitious example where we would have the resources to work on half of the datasets, COI would almost correctly identify all datasets we would like to work on. Other selections like top-3 or top-1 would be less close to the optimal solution, however it is also unrealistic to expect anything more from such a basic heuristic to predict the algorithms outcome much better: After all, COI does not know anything about the algorithm at hand, so the algorithm may have simply failed to realize the potential that was actually present in the data. Also, both ERI and EAS have a potential to be misleading: A dataset that contains a large span of returns could also be collected by a deterministic policy and a very stochastic environment, which means simply imitating the good trajectories will not help. Similarly, a well explored dataset according to EAS could simply not include the most important state-action space regions (random actions don't necessarily bring the agent to the most interesting parts). Apart from these special cases however, the proposed indicators should be helpful in practice, especially when the available resources include working on more than just a single dataset.

Concluding, we propose a new view on offline RL which has so far not been discussed by the corresponding literature: We assume multiple static datasets to be available for offline learning instead of just a single one, and introduce the *dataset selection problem* in which we need to assess which of the available datasets are likely most suitable for offline learning in terms of the return improvement that can be expected. We propose two very cheap indicators, EAS & ERI, as well as their combination COI, in order to do so and evaluate them on the industrial benchmark datasets successfully. In the future, we expect more efforts towards data quality in offline RL (which might include more sophisticated indicators), as this has become a factor in other machine learning fields already.

Dataset	ERI	EAS	COI	TRI
bad-0.2	**15**	3	**10**	15
bad-0.4	**14**	5	**11**	14
bad-0.6	**13**	**8**	**12**	13
bad-0.0	**8**	0	5	12
bad-0.8	**12**	**14**	**15**	11
mediocre-1.0	**11**	**12**	**13**	10
mediocre-0.6	**9**	**13**	**9**	9
mediocre-0.8	**10**	**15**	**14**	8
optimized-0.8	**6**	11	8	7
mediocre-0.4	**5**	10	**7**	6
mediocre-0.2	**7**	**6**	**6**	5
optimized-0.4	**2**	**7**	**3**	4
optimized-0.0	**0**	**1**	**0**	3
optimized-0.2	**1**	**4**	**1**	2
optimized-0.6	**3**	9	**4**	1
mediocre-0.0	**4**	**2**	**2**	0
Spearman's ρ to TRI	0.91	0.13	0.75	
ρ w/o 'bad-*' datasets	0.81	0.83	0.86	

Table 4.1: Datasets together with their values for the derived indicators ERI, EAS, and COI (=Combined Offline Indicator), and sorted by their ground truth return improvement (TRI) from best to worst. The extra horizontal line in columns 2-5 separates the top and bottom half of datasets in terms of their TRI. If we had a budget to work on eight of the sixteen datasets, we would ideally like to select the top half. Bold values indicate that corresponding feature would have placed the dataset in the correct half.

5 Policy Regularization for Model-based Offline Reinforcement Learning

In this Chapter, we present the offline reinforcement learning algorithms MOOSE & WSBC, that were initially published in contributions [B & C]. As pointed out in sections 2.3 & 2.4, offline RL algorithms need to regularize the trained policy in such a fashion that it only visits states and performs actions that are supported by the distribution induced by the given dataset, so that the estimated policy performance is an accurate reflection of what can be expected once it is deployed in the real environment. Both algorithms are model-based, i.e. they train transition models based on the dynamics data observed in the provided dataset, and use the models to evaluate the trained policy's performance. The main difference among the two is the method used for policy regularization: MOOSE regularizes the policy in the action space, by penalizing high reconstruction losses of actions under a conditional variational autoencoder that was trained to mimic the generating policy that generated the dataset. In contrast, WSBC uses an approach that regularizes the policy in parameter space directly.

5.1 Regularization in Action Space by Penalizing Reconstruction Error

Offline RL is a problem setting that is often encountered in practice: The only asset upon project start is often just a batch of historic data that was passively collected via logging sensor data and control signals during ordinary operation of some system. The enormous successes that (online) deep reinforcement learning algorithms were able to celebrate starting a few years ago [3, 4, 106] have so far not changed the world in the way that was largely anticipated by machine learning researchers, mostly due to the sim2real gap [125, 24, 126], meaning that policies trained in a simulation environment were unable to transfer their performance to the real problem, since the simulation turned out to not be as accurate of a reflection of reality as expected. On the other hand, in many domains (e.g. autonomous vehicles, turbine control, factories, or reactors) there exist large quantities of data that have been collected over time, but both learning online as well as building a simulation would be prohibitively expensive (either due to safety or due to cost). Since we lack sufficiently accurate simulations of real-world problems for policy training in many (some would argue in most of the interesting) problems, however we most of the time do possess significant amounts of past interaction data, performing policy optimization directly from the historic data has become a very promising direction of reinforcement learning research.

In online RL algorithms, a big research question is how to balance exploration (learning about your environment) and exploitation (using what you know to get the best possible outcome). In a sense, this dilemma does not transfer to offline RL, since we can by definition not learn anything new about the environment. Offline RL may thus be seen as purely exploiting algorithms. A major question in offline RL is instead how to regularize algorithms so that they actually exploit only what they really know. Many of the early works in offline RL, such as BCQ, BEAR & BRAC, but also more recent algorithms such as TD3+BC rely on regularizing the trained policy in the action space, meaning that they penalize the trained policy for performing actions that differ from actions that the generating policy would have taken when in the same state. More formally, when in state s, the trained policy π needs to choose an action $a \sim \pi(s)$, which is supported by the distribution of actions induced by the generating policy β when in the same state, so $\beta(\cdot|s)$. The key choices in this context are how to represent and learn β, since it cannot generally be assumed to be given (especially when data was collected by human interactions, like in cars or turbine operators, we usually cannot represent the policy in closed form), and how to define "being supported by". The previously mentioned algorithms all propose different solutions to these two problems:
BCQ learns a conditional variational autoencoder model of the generating policy, which enables it to sample likely actions from it, i.e. it learns $\beta(\cdot|s)$, and samples a set of candidate actions from it: $a_0, ..., a_N \sim \beta(\cdot|s)$. Among candidates, BCQ then selects the most promising one according to its Q-value, i.e. $a = \arg\max_{a_i} Q(s, a_i)$. Since the actions were samples from the generating policy, it can be assumed that the final action is supported by the generating policy (BCQ slightly perturbs the actions afterwards, but the concept remains the same). BEAR uses the same generating policy representation, but it trains a closed form policy π, by penalizing the maximum mean discrepancy between π and β, i.e. $MMD[\pi(\cdot|s), \beta(\cdot|s)]$, meaning that actions sampled from π can be assumed to also be supported by β. BRAC changes the representation of β to a regular Gaussian policy and chooses the KL divergence instead of the MMD. TD3+BC comes without a model of the generating policy and directly penalizes the squared distance in the action space, i.e. $(\pi(s) - a)^2 \quad s, a \sim \mathcal{D}$.

All four algorithms are model-free, i.e. use an action-value function to estimate the return of the trained policy. The goal of the here presented algorithm - Model-based Offline Policy Search with Ensembles (MOOSE), is to bring the action space policy regularization paradigm for offline learning to the model-based domain. This endeavour is promising, since model-based RL algorithms have long been attributed superior sample efficiency compared with their model-free counterparts [16, 11, 18, 15]. The effect is usually explained by the fact that model-based algorithms use more of the available information for training (future state as a target for the dynamics model versus return only for the value function) and that they thus learn latent representations that generalize better. Since offline RL is an innately data-scarce problem due to the lack of additional data collection, it seems only natural to extend the action space regularization for offline

RL to the model-based domain. An additional upside of transition models in offline RL is that they are static (i.e. a change in policy does not mean that the model has to be retrained) and that they can be examined by expert practitioners (do the state transitions look meaningful / correct). Note that at the time of development of MOOSE, only BCQ and BEAR were published, so MOOSE can be considered one of the first model-based offline RL algorithms. MOPO and MOReL were proposed concurrently, they however did not serve as inspiration for MOOSE and we were only made aware of them after [B] was submitted for publication. Nevertheless, we include experimental comparisons with them in Section 5.1.2.

5.1.1 Model-based Offline Policy Search with Ensembles (MOOSE)

The MOOSE algorithm builds on top of the model-based RL framework presented in Section 2.2.2. Specifically, the algorithm trains an ensemble of K transition models $\{\hat{T}^k | k = 0, ..., K-1\}$ with parameters $\phi = \{\phi_k | k = 0, ..., K-1\}$. The transition models are deterministic, i.e. they each predict a single estimate of the future state $\hat{s}_{t+1}$ given the current state s_t and action a_t. Depending on the environment, MOOSE either trains the ensemble by minimizing the mean squared error between the future state and the model output (i.e. Equation 2.31), or between the difference in states and the model output (i.e. $\Delta s_t = s_{t+1} - s_t$).

$$\begin{aligned} L(\phi_k) &= \sum_t ||\hat{T}_{\phi_k}(s_t, a_t) - \frac{(s_{t+1} - s_t) - \mu^{\mathbf{\Delta s}}}{\sigma^{\mathbf{\Delta s}}}||_2 && (5.1) \\ &\text{or} \\ L(\phi_k) &= \sum_t ||\hat{T}_{\phi_k}(\frac{s_t - \mu^{\mathbf{s}}}{\sigma^{\mathbf{s}}}, a_t) - \frac{s_{t+1} - \mu^{\mathbf{s}}}{\sigma^{\mathbf{s}}}||_2 && (5.2) \end{aligned}$$

For the model training, 10% of the data is left out as a validation set in order to assess the models' performance on unseen data. This allows choosing which of the two losses yield the better result, as well as potentially other hyperparameter choices, however MOOSE does not explicitly contain any hyperparameter tuning stage. MOOSE generally expects a dataset of T transition tuples $\{(s_t, a_t, r_t, s_{t+1}) | t = 0, ..., T-1\}$, meaning that normally the reward function is also learned by a separate set of reward models $\{\hat{r}^k | k = 0, ..., K-1\}$ parameterized by $\{\psi_k | k = 0, ..., K-1\}$, which each predict a single reward estimate $\hat{r}_t$ based on the current state and action, i.e. $\hat{r}_t^{(k)} = \hat{r}_{\psi_k}(s_t, a_t)$. The training is in this case analogous to the dynamics model training in Eq. 5.2 - even though generally possible, MOOSE does not train on the difference in rewards:

$$L(\psi_k) = \sum_t ||\hat{r}_{\psi_k}(\frac{s_t - \mu^{\mathbf{s}}}{\sigma^{\mathbf{s}}}, a_t) - \frac{r_t - \mu^{\mathbf{r}}}{\sigma^{\mathbf{r}}}||_2 \qquad (5.3)$$

As can be seen in Equations 5.1 - 5.3, MOOSE normalizes inputs as well as outputs to have zero mean and unit variance for better accuracy. Further, as outlined in Chapter 3, MOOSE uses weight normalization for faster convergence and employs Adam as an optimizer for its model parameters.

Since MOOSE was designed with real-world applications like turbine control or factory solutions in mind, it trains deterministic policies π parameterized by θ in order to avoid issues relating to safety that would arise from stochastic policies, such as those proposed by common online algorithms [7], but even by recent offline algorithms such as BCQ or BEAR [8, 52]. Policy evaluation is performed as outlined in Section 2.2.2, except that MOOSE uses an ensemble of transition models to perform the imagined trajectories instead of just a single one, i.e. multiple models predict the reward and the future state in order to ultimately estimate the return R:

$$\begin{aligned} L(\theta) &= -\frac{1}{K}\sum_{k=0}^{K-1}\sum_{t=0}^{H-1}\gamma^t \hat{r}_{\psi_k}(\hat{s}_t, \pi_\theta(\hat{s}_t)) \\ &= -\,\mathbb{E}_{\pi,\hat{T}^{1..K},\hat{r}^{1..K}}[R] \end{aligned} \tag{5.4}$$

Where γ is the discount factor of the respective MDP, H is the length of the considered horizon, $\hat{s}_{t+1} = \hat{T}(\hat{s}_t, \pi(\hat{s}_t))$ and $\hat{s}_0 = s_0$ is sampled from the dataset $\mathcal{D}$. In Equation 5.4, the mean of the reward predictions is used to form the return estimate. In principle however, different weighting schemes for the ensemble are possible. Similarly, different methods exist for carrying forward the estimated future state: Each ensemble member can use its own prediction for the next step "in isolation", the mean of the predicted states can be used for each of the members, or a random one can be chosen among the predictions and then used in the next step for all members, etc.

If we could assume that the dataset sufficiently covers all possible state transitions accurately, the above Equation could directly be used for policy optimization. Since however, we cannot realistically make this assumption in the offline RL context, we now move to the regularization part of the MOOSE algorithm, which will keep the policy in the area of the state-action space sufficiently supported by the dataset, so that the transition models can accurately assess the policy performance. MOOSE aims to minimize the expected model bias in step t, b_t, which is defined as the error in state transitions when comparing trajectories that would be performed in the real environment e with imagined trajectories through the ensemble members $\hat{T}^k$:

$$b_t = ||s_{t+1} - \hat{s}_{t+1}||_2 = ||e(s_{t+1}|s_t, a_t) - \hat{T}(\hat{s}_t, \hat{a}_t)||_2 \tag{5.5}$$

where the transition (s_t, a_t, r_t, s_{t+1}) stems from the real environment and $(\hat{s}_t, \hat{a}_t, \hat{r}_t, \hat{s}_{t+1})$ stems from the imagined trajectory through the transition model(s), where both trajectories started in the same starting state $\hat{s}_0 = s_0$, and where both trajectories were collected under the same policy π. The sum of biases in a trajectory is referred to as B:

$$B = \sum_t b_t \tag{5.6}$$

MOOSE aims to minimize the expected value of B, for which we need to know the state visitation distribution for the policy π under the real environment e, μ_π^e:

$$\mathbb{E}_{s\sim\mu_\pi^e, a\sim\pi(s), \hat{s}\sim\mu_\pi^{\hat{T}}, \hat{a}\sim\pi(\hat{s})}[B] \tag{5.7}$$

where $\mu_\pi^{\hat{T}}$ is the state visitation distribution under the transition model(s). While we can estimate $\mu_\pi^{\hat{T}}$ by performing rollouts through $\hat{T}$, we cannot say much about μ_π^e since we do not have access to the real environment transition $e(\cdot|s,a)$. The only policy for which we can estimate it, is the generating policy β, i.e. we can sample from μ_β^e by sampling from the dataset $\mathcal{D}$. MOOSE thus aims to minimize the model bias under μ_β^e and $\mu_\pi^{\hat{T}}$, which automatically moves π closer to β, as this is necessary for the bias to decrease:

$$\theta^* = \arg\min_\theta \mathbb{E}_{s\sim\mu_\beta^e, a\sim\beta(s), \hat{s}\sim\mu_\pi^{\hat{T}}, \hat{a}\sim\pi(\hat{s})}[B] \tag{5.8}$$

Intuitively, it is clear that if we are in some imagined state $\hat{s}$ and perform an action $\hat{a}$ given by the trained policy π, we want the following two things in order to have low bias:

- The probability of the state $\hat{s}$ under the true environment and the generating policy, μ_β^e, should be as high as possible. Otherwise we will likely not have observed a similar transition in the dataset before and cannot hope to have modeled it accurately with $\hat{T}$.
- The probability of the action $\hat{a}$ under the given state $\hat{s}$ should be likely under the generating policy, since otherwise, similarly, we will likely not have encountered such a transition in the dataset before.

Since we are only able to change the parameters θ of the policy π, we aim to allow a large $\mu_\pi^{\hat{T}}(\hat{s})\pi(\hat{a}|\hat{s})$ only if the probability $\mu_\beta^e(\hat{s})\beta(\hat{a}|\hat{s})$ is also large. In order to reach this goal of minimizing bias, MOOSE makes an assumption about how the probability of the generating policy to have generated some sample (s,a) in the real environment influences the error we can expect the transition models to make while predicting the successor state of this sample (i.e. how large the bias is).

Assumption 1 *The distribution of model errors $e(s,a) - \hat{T}(s,a)$ has a variance that is monotonically decreasing with increasing probability of having seen the imagined data in reality, i.e., that the data sample (s,a) was generated under the original environment dynamics e and the generating policy β:*

$$e(s,a) - \hat{T}(s,a) \sim \mathcal{N}(0, -\log p_{e,\beta}(s,a)) \tag{5.9}$$

Under this assumption, minimizing bias immediately translates to maximizing the probability of the state and action probabilities under the real environment and the generating policy, since the expectation of a squared Gaussian variable is its variance. Starting out with the objective to minimize bias:

$$\begin{aligned}
\theta^* = & \quad \underset{\theta}{\arg\min} \quad \mathbb{E}\left[B\right] \\
= & \quad \underset{\theta}{\arg\min} \quad \mathbb{E}\left[\sum_t b_t\right] \\
= & \quad \underset{\theta}{\arg\min} \quad \mathbb{E}\left[\sum_t (e(\hat{s}_t, \hat{a}_t) - \hat{T}(\hat{s}_t, \hat{a}_t))^2\right] \\
= & \quad \underset{\theta}{\arg\min} \quad \mathbb{E}_t\left[-\log p_{e,\beta}(\hat{s}, \hat{a})\right] \\
= & \quad \underset{\theta}{\arg\max} \quad \mathbb{E}_t\left[\log p_{e,\beta}(\hat{s}, \hat{a})\right] \\
= & \quad \underset{\theta}{\arg\max} \quad \mathbb{E}_t\left[p_{e,\beta}(\hat{s}, \hat{a})\right] \\
= & \quad \underset{\theta}{\arg\max} \quad \mathbb{E}_t\left[\mu^e_\beta(\hat{s})\beta(\hat{a}|\hat{s})\right]
\end{aligned} \tag{5.10}$$

we quickly arrive at the desired maximization of the probability of state-action pairs under the generating policy and the real environment.
It should be noted, that the MOOSE approach thus not only penalizes actions that were unlikely under the generating policy (something that prior works, such as KL control [127] have done in other settings before), but also penalizes states that were unlikely to be visited under the true dynamics and the generating policy.

In order to derive a practical algorithm from the approach, MOOSE needs to find a way to model $\mu^e_\beta(\hat{s})\beta(\hat{a}|\hat{s})$, which is in this case achieved using a conditional variational autoencoder model (VAE, see Section 2.7) [82]. The VAE v with parameters ω is trained to reconstruct actions based on the state-action pairs provided in the dataset, which means that it will reconstruct actions better if the corresponding input state-action pair has been seen in the provided dataset. By proxy, it thus models the likelihood of $\mu^e_\beta(\hat{s})\beta(\hat{a}|\hat{s})$, since both out-of-distribution actions given an observed state, as well as out-of-distribution states, will lead to worse reconstructions. Since we only really measure the action reconstruction error though, MOOSE is said to employ an action-space regularization. In order to optimize v, we optimize the evidence lower bound (ELBO) and place a Gaussian prior on the latent space:

$$\begin{aligned}
L(\omega) = \mathbb{E}_{q_\omega(z|s,a)}[-\log p_\omega(a|s,z)] + D_{KL}(q_\omega(z|s,a)||p(z)) \\
p(z) \sim \mathcal{N}(0,1)
\end{aligned} \tag{5.11}$$

Under standard assumptions (normally distributed modeling errors in Euclidean space), the ELBO is minimized using a mean squared error reconstruction loss. In order to minimize the bias during policy training, we will thus use it again to assess the probability of a state-action pair: Low reconstruction errors correspond to high probabilities since the model has learned to accurately reproduce the input, and vice versa. Similarly to

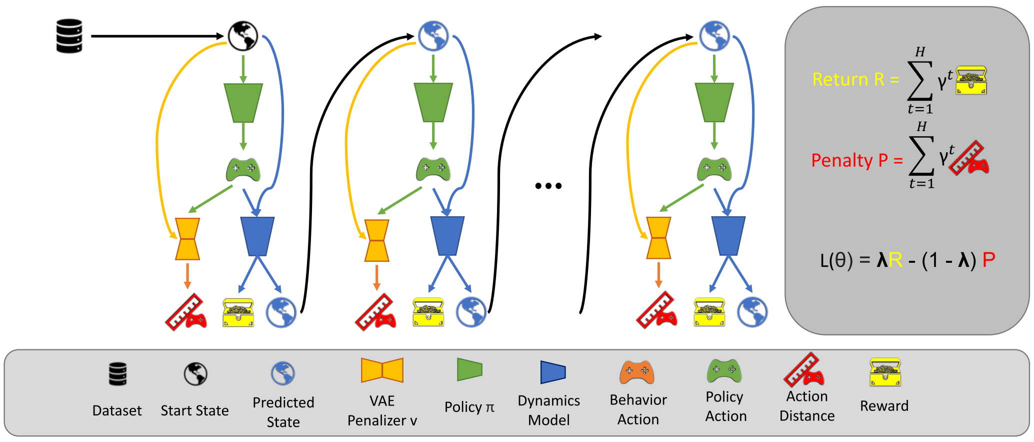


Figure 5.1: Visualization of the MOOSE training procedure. First, a start state is sampled from the dataset (black, top-left), and provided to the target policy (green) to propose an action. This action is then given to the VAE (orange) together with the current state to assess how well the action is represented in the data and calculate the penalty. The target action and current state are then also provided to the transition ensemble (blue) in order to predict the reward as well as the future state, with which the procedure starts anew until the optimization horizon is reached.

the rewards, we accumulate the reconstruction errors to form a penalty term:

$$\mathbb{E}[P] = \sum_t \mathbb{E}_{q_\omega(z|s,a),(s,a)\sim\pi,\hat{T}}[-\log p_\omega(a|s,z)] \tag{5.12}$$

$$= \sum_t \mathbb{E}_{p_\omega(\hat{a}|\hat{s},z),q_\omega(z|s,a),(s,a)\sim\pi,\hat{T}}[(a-\hat{a})^2] \tag{5.13}$$

MOOSE then optimizes the policy π by extending the loss function in Equation 5.4, by using a convex combination of the expected return estimate and the derived penalty:

$$L(\theta) = -\lambda\mathbb{E}[R] + (1-\lambda)\mathbb{E}[P] \tag{5.14}$$

See a visualization of the policy training procedure in Fig. 5.1. Additionally, MOOSE biases the return estimate towards the minimum of the ensemble instead of the mean, in order to incorporate a form of conservatism induced by the transition models. This methodology is a generalization of a technique inspired by double Q-learning [8], where the authors use two value functions and mix it ($\frac{1}{4}\max + \frac{3}{4}\min$) in order to obtain a more conservative value estimate. Since the MOOSE ensemble may consist of more than two members, the new return estimate is defined over the mean and the min:

$$\mathbb{E}[R] = \eta \min_k \left\{ \sum_t \gamma^t r(s_t, \pi_\theta(s_t), f_k(s_t, \pi_\theta(s_t))) \right\} \tag{5.15}$$

$$+ (1-\eta)\frac{1}{K}\sum_k \left[\sum_t \gamma^t r(s_t, \pi_\theta(s_t), f_k(s_t, \pi_\theta(s_t))) \right]$$

where η, just like λ needs to be between 0 and 1. An overview over the MOOSE algorithm is again provided in pseudocode in Algorithm 1. This concludes the formalization of the idea as well as the practical derivation of MOOSE - one of the first model-based offline RL algorithms, which aims to regularize the trained policy π by means of a VAE induced reconstruction penalty in the action space. In the next section, we will show experimental results of the approach on a variety of benchmark datasets, and compare the algorithm's performance with other state of the art offline RL baselines.

Algorithm 1 MOOSE

```
1:  Require Dataset $D$, randomly initialized parameters $\theta$, $\phi$, $\omega$, horizon $H$, number of
    policy updates $U$
2:  // dynamics and VAE models can be trained supervised and independently of other
    components
3:  train original policy model $\upsilon_\omega$ using $D$ and Equation 5.11
4:  train dynamics models $\hat{T}^k_{\phi^k}$ with $D$ and Equation 5.1
5:  for j in 1..U do
6:      sample start states $S_0 \sim D$
7:      estimate $\mathbb{E}[R]$ using $\hat{T}^k_{\phi^k}$ and Equation 5.15
8:      estimate $\mathbb{E}[P]$ using $\upsilon$ and Equation 5.12
9:      $\theta_j \leftarrow \theta_{j-1} - \alpha\nabla_{\theta_{j-1}}[-\lambda\mathbb{E}[R] + (1-\lambda)\mathbb{E}[P]]$
10: return $\pi_\theta$;
```

5.1.2 Experimental Comparison of MOOSE with Prior Methods on the Industrial Benchmark & MuJoCo

In order to empirically verify our approach, we evaluate the MOOSE algorithm on 19 datasets from the MuJoCo and the industrial benchmark domain. The 16 industrial benchmark datasets have been collected under three different baseline policies (bad, mediocre, optimized) mixed with varying degrees of ε-greedy exploration (0%, 20%, 40%, 60%, 80%, and one dataset with entirely random data). The MuJoCo datasets have been collected from three different robotic locomotion environments (Swimmer, Hopper, Walker2D), by adding Gaussian noise to a pretrained expert policy in 70% of cases mixed with entirely random actions in 30% of cases. For a detailed description of the datasets and environments, we refer to sections 3.2 & 3.1.

In the experiments, we aim to analyze the following points:

- **Environment Generalization** We aim to derive an algorithm that is generally applicable across different domains. Thus, we test it on different robotic environments as well as the IB, which is inspired by applications such as turbine or reactor control, in order to see whether MOOSE can derive effective offline policies across a range of tasks.

	Swimmer	Hopper	Walker2D
DDPG	-10.2 (1.8)	0.4 (4.5)	-12.7 (3.9)
BRAC-v	**25.9 (0.1)**	134.0 (0.2)	17.1 (0.9)
BEAR	22.9 (0.1)	117.4 (1.3)	-7.0 (3.8)
BCQ	24.8 (0.1)	132.0 (0.2)	91.9 (1.0)
TD3+BC	13.9 (0.1)	56.7 (0.1)	28.9 (0.5)
CQL	8.5 (0.2)	56.1 (0.1)	42.9 (0.3)
MOPO	8.3 (0.1)	4.2 (0.1)	-8.8 (0.1)
MOReL	4.1 (0.1)	5.8 (0.1)	-8.3 (0.1)
MOOSE	24.2 (0.1)	**147.2 (0.2)**	**113.7 (0.6)**

Table 5.1: Robust performance of the algorithms in the MuJoCo experiments. To assess robustness, 10th percentile performance is shown together with its standard error. Final 10% of performance values across five seeds taken into account. MOOSE outperforms its model-free counter parts on Hopper and Walker, while BRAC performs better on the Swimmer task.

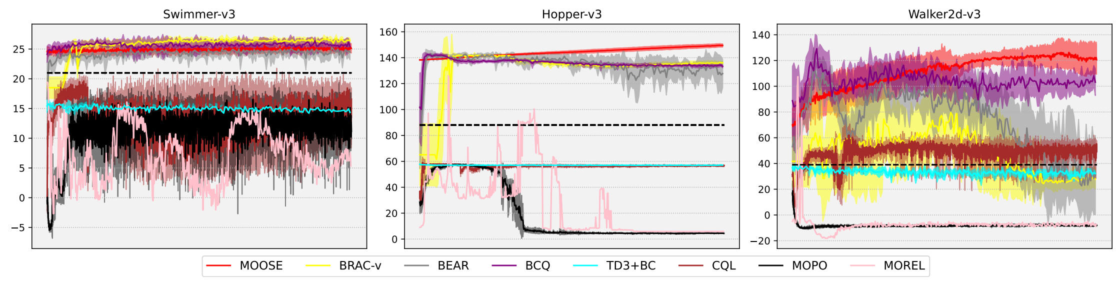
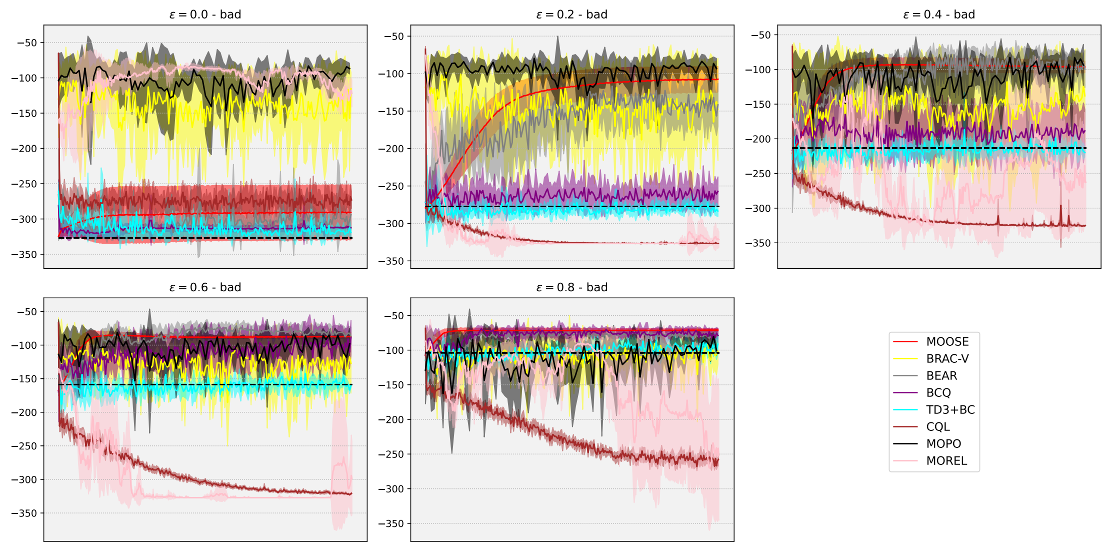


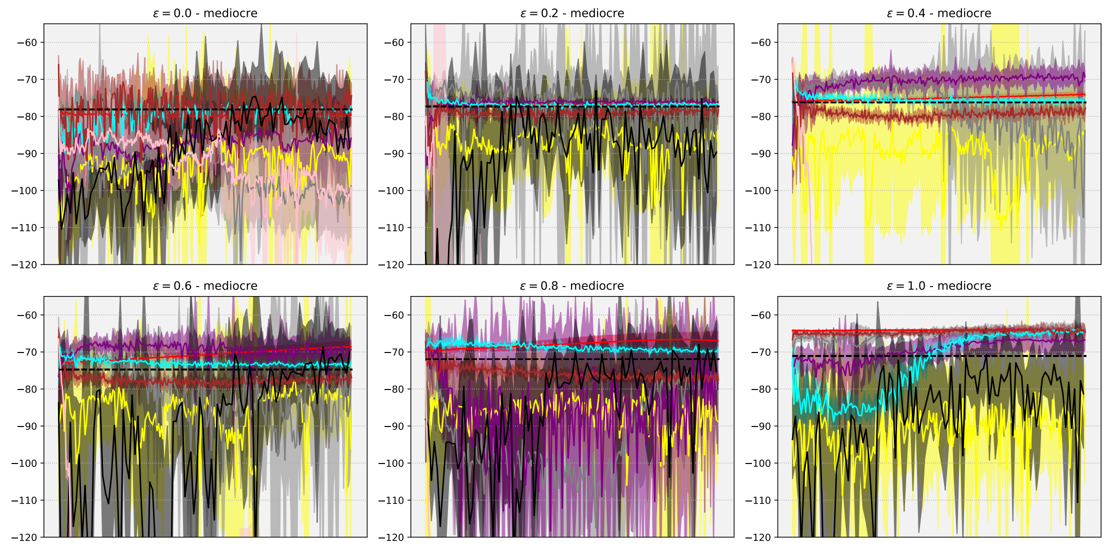
Figure 5.2: Mean performance ± standard deviation in MuJoCo experiments. Dashed line represents original batch performance.

- **Dataset Generalization** MOOSE is supposed to work well, no matter how the provided dataset was generated. The datasets thus represent a diverse set of baseline policies (expert data, undirected data, mediocre attempts) and exploration amounts (from none at all to pure exploration), so that we can test how the algorithm performs when faced with various different settings.
- **Performance Comparison** Of course, we aim to answer the question whether MOOSE can beat state of the art offline RL algorithms on the benchmark datasets, i.e. whether the design choices including the model-based value estimation and the VAE based action space regularization pay off.
- **Learning Stability** Last but not least, we care about the stability of the learning process: In offline RL more so than in online RL, it is crucially important that the algorithm produces a well performing policy reliably, and not only every 100 seeds, since we cannot reliably evaluate the performance before deployment.

Training curves for the experiments are provided in Figures 5.3 & 5.2, while final performance values are given in Tables 5.2 & 5.1. The performance values are obtained as follows: Since we do not know the true performance of a policy in the real environment until actual deployment (can consequently cannot perform policy selection), we report 10$^{\text{th}}$ percentile performance of the final 10% of policies in order to get an understanding of how an algorithm will do from a perspective that is closer to the worst case. Often, practitioners will care more about not loosing performance than about gaining it (a bias common in human nature), which is why this perspective is much more important for real world deployments than average performance. As we work with limited computational resources, we cannot repeat all experiments another J times. Instead, we train five seeds for each algorithm, and then collect the final 10% of policies obtained in these trainings to add them to a pool of final policies. We then calculate the 10$^{\text{th}}$ percentile performance on this larger policy pool. Strictly speaking, the assumption necessary for this methodology to obtain correct results (independence of the policies) is not fulfilled, however it is the best we can do with the limited amount of compute available. Also, considering the final 10% of policies actually well reflects the early stopping problem in offline RL: We do not know when to stop training, so ideally all policies close to the "end" should be admissible. In order to provide an uncertainty estimate of the 10$^{\text{th}}$ percentile, we assume policy performances to be normally distributed, calculate the standard error of the mean, and multiply it by 1.7, since we find through Monte Carlo experiments that the 10$^{\text{th}}$ percentile is about 1.7 times as uncertain as the mean for normally distributed variables.

Experimental Details In accordance with prior literature we set the discount factor γ to 0.99 for the MuJoCo and to 0.97 for the industrial benchmark datasets. We perform virtual rollouts of length 100 and evaluate on 10 trajectories of the same length. MOOSE's transition models and policies have two layers of 400 and 300 neurons and use ReLU nonlinearities, except for the final layer, where the transition models do not employ any nonlinearity, and the policies use a tanh nonlinearity to normalize the outputs between $(-1, 1)$. The variational autoencoder models have two encoding layers of size 750, as well as two decoding layers of the same size. They also use ReLU nonlinearities, except for the final layer. The latent space dimension is 2×action_dimensionality, since it learns mean and variance separately. While the policies are trained with a batchsize of 100, the transition models and VAEs use batches of size 500. We use the Adam optimizer [100] with standard parameters and learning rate 10^{-4} for transition and VAE models. For the MuJoCo policies we decrease this value to 10^{-5} and for the IB policies we even switch to vanilla SGD with a learning rate of 10^{-4}, since we found that the momentum style components of Adam hurt the rollout based optimization and induced too much variance. Transition model outputs are furthermore clipped to stay inside the known range of values to avoid extreme extrapolation mistakes.
We train MOOSE's tranistion and VAE models for 50 full epochs, i.e. until each sample has passed 50 times, and the policy for 5000 steps, except in the Hopper environment, since the transition model predicts premature falling over of the robot after about 1000 steps. For the other algorithms, we stick to the standard set of hyperparameters, ex-





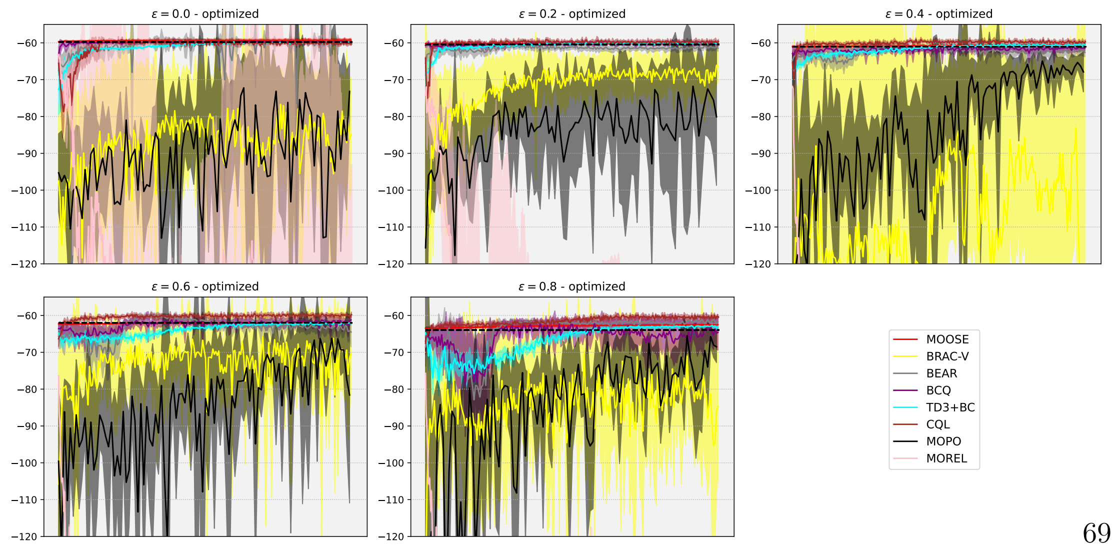


Figure 5.3: Mean performance ± standard deviation in IB experiments. Dashed line represents original batch performance.

cept that we reduce the amount of compute since the amount of data is also lower, i.e. we train BCQ, BEAR, BRAC, and TD3+BC for 10,000 steps on the IB datasets, and similarly train 300 epochs instead of 3000 for CQL, and 100 epochs instead of 1000 for MOPO. MOOSE is decreased from 5000 to 1000 steps. We choose the penalty-reward mixture parameter λ at a very conservative 0.01, since we have no way of tuning it. The ensemble size is always $K = 4$ and we use $\eta = 0.5$ to trade of between mean and min estimated performances.

Table 5.2: Tenth percentile performances of evaluated Offline RL algorithms and their standard error. Final 10% of performance values across five seeds taken into account. MOOSE performs best on 9 out of 16 datasets, while CQL is best in three, MOPO and BCQ in two, and TD3+BC takes the top spot on a single dataset.

	$\varepsilon =$	0.1	0.2	0.4	0.6	0.8	1.0
Bad	DDPG	-326 (25)	-383 (5)	-383 (7)	-383 (7)	-383 (7)	
	BRAC	-266 (8)	-248 (8)	-265 (8)	-190 (5)	-132 (4)	
	BEAR	-322 (3)	-180 (4)	-129 (3)	-98.3 (1.3)	-88.7 (0.9)	
	BCQ	-313 (1)	-286 (2)	-234 (5)	-137 (4)	-87.1 (1.2)	
	TD3+BC	-326 (2)	-289 (2)	-230 (2)	-173 (2)	-113 (2)	
	CQL	-292 (2)	-327 (1)	-326 (2)	-323 (1)	-271 (1)	
	MOPO	**-124 (5)**	**-110 (4)**	-140 (8)	-131 (5)	-119 (5)	
	MOREL	-144 (2)	-327 (2)	-327 (8)	-327 (7)	-327 (11)	
	MOOSE	-322 (5)	-126 (2)	**-109 (1)**	**-90.3 (0.3)**	**-73.3 (0.3)**	
Mediocre	DDPG	-960 (34)	-960 (36)	-960 (37)	-961 (24)	-960 (48)	-178 (3)
	BRAC	-114 (2)	-103 (2)	-112 (2)	-99.0 (2.6)	-101 (3)	-107 (2)
	BEAR	-112 (1)	-103 (5)	-124 (3)	-105 (3)	-99.7 (1.6)	-66.3 (0.2)
	BCQ	-103 (1)	**-77.1 (0.1)**	**-72.7 (0.3)**	-75.9 (0.7)	-107 (2)	-69.2 (0.4)
	TD3+BC	**-79.7 (0.6)**	-77.7 (0.1)	-76.8 (0.1)	-74.3 (0.1)	-70.7 (0.1)	-65.4 (0.1)
	CQL	-89.5 (1.2)	-80.5 (0.2)	-80.8 (0.2)	-79.5 (0.2)	-78.9 (0.2)	-66.0 (0.2)
	MOPO	-102 (4)	-119 (5)	-81.4 (2.3)	-86.1 (2.5)	-92.5 (4.5)	-105 (7)
	MOREL	-122 (2)	-327 (6)	-327 (4)	-327 (7)	-327 (1)	-327 (12)
	MOOSE	-83.0 (0.3)	**-77.1 (0.1)**	-75.1 (0.1)	**-70.5 (0.2)**	**-68.7 (0.3)**	**-64.3 (0.1)**
Optimized	DDPG	-416 (24)	-382 (11)	-257 (21)	-312 (30)	-168 (3)	
	BRAC	-113 (3)	-77.3 (0.8)	-158 (6)	-89.2 (1.7)	-116 (3)	
	BEAR	-60.5 (0.3)	-62.5 (0.1)	-64.4 (0.2)	-66.0 (0.3)	-63.0 (0.1)	
	BCQ	-60.1 (0.1)	-60.9 (0.1)	-62.7 (0.1)	-63.6 (0.2)	-72.4 (0.5)	
	TD3+BC	-60.3 (0.1)	-60.6 (0.1)	-61.1 (0.1)	-62.7 (0.1)	-63.6 (0.1)	
	CQL	-60.9 (0.1)	-60.6 (0.1)	**-60.6 (0.1)**	**-60.6 (0.1)**	**-61.3 (0.1)**	
	MOPO	-126 (9)	-102 (4)	-72.0 (0.9)	-80.6 (3.3)	-90.3 (2.8)	
	MOREL	-279 (8)	-327 (8)	-327 (1)	-327 (1)	-327 (7)	
	MOOSE	**-59.7 (0.1)**	**-60.3 (0.1)**	-60.8 (0.1)	-62.1 (0.1)	-62.7 (0.1)	

5.1.3 Discussion

In this section we develop the idea of the MOOSE algorithm: A purely model-based offline RL algorithm that estimates policy returns based on virtual rollouts and does not employ a value function. MOOSE instead adapts the idea of action space policy regularization and is the first algorithm to employ it in a model-based context in order to facilitate learning only from a single pre-collected static dataset.

In our comparison, MOOSE performs best on nine of the sixteen industrial benchmark datasets in terms of robust (10th percentile) performance. Note that CQL & TD3+B were proposed much later than MOOSE, so that at the time of publication MOOSE would even outperform the remaining algorithms on 13 out of the 16 IB datasets. On the MuJoCo tasks, MOOSE performs best on the more complex Hopper and Walker tasks, but is outperformed by BRAC and BCQ in the Swimmer environment. Overall, this highlights MOOSE's capability to produce well performing policies reliably, over a range of different tasks and datasets, as well as in high dimensional state and action spaces. Compared to most other algorithms in Figures 5.3 & 5.2, its performance curves are much smoother, i.e. exhibit much lower variance, which is an important factor in offline RL due to the lack of early stopping / policy evaluation possibility. We attribute most of this gain to the fact that MOOSE is the only algorithm in the comparison which does not use a value function for planning - an element which has in the past been known for being unstable and hard to train. By instead optimizing directly on imagined trajectories, combined with the novel action space penalty based on VAE reconstruction error, MOOSE appears better suited for real world offline RL deployments. Further, MOOSE, DDPG, and TD3+BC are the only algorithms in the comparison able to derive deterministic policies (a requirement likely in real world systems, especially those of mechanical nature due to safety concerns), and MOOSE far outperforms the other two.

It is no surprise that DDPG does not manage to train any decent policies since even though it is off-policy it does not contain any adaptations for the offline setting, and it was already shown in Section 2.4 that these algorithms fail when they are not trained with at least close to on-policy data. Some of the other algorithm results however are a little more surprising: BRAC-v performs well on the MuJoCo tasks, but badly on the IB datasets. We hypothesize this to be due to (a) the Gaussian policy representation, which suits the way the data was collected in the MuJoCo experiments much better, and (b) the divergence estimation based on relatively few samples, leading to large variance. MOReL is also performing rather badly on the IB datasets - in our experiments we observed that the virtual rollouts were likely far too long: The uncertain state detector (USD) was ending the episodes too late, and the value functions were thus trained on synthetic data too far away from the training distribution (i.e. likely too inaccurate). MOPO seemed to have similar issues, albeit on a much lower scale, since it only performs very short rollouts - however since it performs new rollouts starting from the already imagined states, the quality of data likely degrades over time. BCQ, TD3+BC, and CQL appear to be the most promising alternatives to MOOSE - they are a bit conservative and thus lead to lower performances especially when the baseline policy was bad, however that is part of the concept.

5.2 Regularization in Weight Space with Population based Policy Search

A key issue with MOOSE is the gradient based optimization of the policy by means of imagined rollouts: Since MOOSE does not rely on any value function, the gradient of the estimated return needs to be calculated with respect to the sum of individual step rewards, which can be an issue due to two problems: (i) The gradient needs to be propagated back through many steps, leading to potentially exploding or vanishing gradients, a problem well known from very deep networks in language or vision models [98, 99]. (ii) It is unclear whether the individual step gradients are the best way to optimize the cumulative return, since they mutually influence each others - e.g. in the Hopper environment, if one step yields a higher reward since the robot moved faster, the policy might loose control over it in the next step, the robot falls over and the trajectory ends. We thus seek to develop an alternative rollout based algorithm that can optimize the policy without relying on the gradient information of the return estimated by the transition model.

5.2.1 Weight Space Behavior Constraining (WSBC)

The key novel ingredient for the WSBC algorithm needs to be a new way to optimize the policy with respect to the estimated performance and the penalty for diverging from the known data distribution, since the optimization of Equation 5.14 in MOOSE is performed using Adam, which is an advanced gradient descent method. Since we do not fully trust the gradient to do exactly what we aim for (individual step rewards versus global return), and since numerical stability issues may arise in the gradient setting due to the long rollouts, we will replace the gradient based optimization with a gradient-free, population based search algorithm.
Gradient-free optimization techniques, such as CMA-ES (Covariance Matrix Adaptation Evolution Strategy) or PSO (Particle Swarm Optimization) have been used in prior work to tackle control problems [128, 129, 12]. [128, 130] found that training a policy gradient-free required more environment interactions, since they were at the same time neither using a value function nor a transition model in order to estimate policy performance. In the offline setting with model-based value estimation, this result does not concern us too much, since the limiting factor is usually not how often we can evaluate the transition model to estimate the performance of the current policy. [13] developed an approach to train neural policies in a gradient-free fashion for the industrial benchmark directly from static datasets. In that case, the datasets contained 100% uniform random actions, i.e. a lot of exploration, which didn't require the authors to make any adjustments to regularize the policy towards the data distribution. Instead, since the transition model was able to learn properly about most parts of the state-action space, the PSONN (particle swarm optimization neural network) algorithm searches directly for neural policies by assessing a particles fitness via trained transition models. Since the assumption of uniformly random actions is not particularly realistic, we will in the following develop a new offline RL algorithm by bringing together ideas from PSONN and MOOSE: The

policy optimization needs to be performed gradient-free, but it also needs to be regularized towards the data distribution.

Transition Models First, we introduce the transition models used in our algorithm, *Weight Space Behavior Regularization* (WSBC). We take inspiration from the recurrent networks developed for the industrial benchmark in [46]: The RNNs have a single RNN cell with tanh nonlinearity, as introduced in Section 2.2.2, i.e. the hidden state is calculated using Equation 3.39, by using the prior hidden state as well as the new input, in this case the new state and action. The next environment state (and/or reward) is then calculated based on the updated hidden state using Eq. 3.40. Since [46] found it beneficial for the prediction error on a held out evaluation dataset, we train two separate RNNs for consumption and fatigue on the IB. Further, the model for consumption does not receive its past prediction for the consumption back as an input (see Section 3.3). For the MuJoCo environments, we only use one RNN to predict both state and reward. The loss function for training the transition models based on the initial dataset $\mathcal{D}$ thus looks as follows (we omit explicit notation of modeling deltas instead of states):

$$L(\phi) = \mathbb{E}_{s_0 \sim D}\left[\sum_{t=G}^{G+F} [s(\hat{T}_\phi(\hat{s}_t, a_t, h_{t-1})) - s_{t+1}]^2\right] \tag{5.16}$$

$$\text{with} \quad \hat{s}_{t+1} = \begin{cases} s_{t+1} & \text{if} \quad t < G \\ s(\hat{T}_\phi(\hat{s}_t, a_t, h_{t-1})) & \text{else} \end{cases} \quad \text{and} \quad h_t = h(\hat{T}_\phi(\hat{s}_t, a_t, h_{t-1}))$$

where $s(\cdot)$ describes the output portion of the transition model $\hat{T}$ that contains the prediction of the future state and $h(\cdot)$ describes the corresponding portion containing the updated hidden state. For G steps, the model is fed the correct input states (and actions) that were observed during the trajectory in the dataset in order to build the hidden state. Afterwards, the model receives its own past prediction (if applicable) as "self input" and still needs to predict the true labels for the next F steps. That way, the transition model needs to stay accurate for many steps into the future instead of just a single one. We again train an ensemble of K transition models.

Policy Search Similarly to MOOSE / PSONN, the return or fitness value of a policy is estimated by performing imagined rollouts through the trained ensemble of transition models. We search in the space of neural network policies π with parameters θ, that provide an action based on the current state s_t and possibly more past states: $a_t = \pi_\theta(s_t, [s_{t-1}, ..., s_{t-G}])$ (The IB is partially observable and benefits from a collection of past timesteps taken into consideration, while this does not play a role in the MuJoCo environments). We limit the dimensionality of the parameters θ by choosing low numbers of neurons and only a single hidden layer, since every extra dimension exponentially increases the search space that needs to be covered by the gradient-free optimizer.

$$L(\theta) = \mathbb{E}_\pi[R] = \frac{1}{|S|} \sum_{s_0 \in S} \sum_{t=G}^{G+F} \gamma^t r(\hat{T}_\phi^{k_t}(\hat{s}_t, \pi_\theta(\hat{s}_t[\hat{s}_{t-1}, ..., \hat{s}_{t-G}]), h_{t-1}^{k_t})) \tag{5.17}$$
$$\text{s.t.} \quad k_t = \arg\min_k r(\hat{T}_\phi^k(\hat{s}_t, \pi_\theta(\hat{s}_t[\hat{s}_{t-1}, ..., \hat{s}_{t-G}]), h_{t-1})) \quad k \in \{0, ..., K-1\}$$
$$\text{with} \quad \hat{s}_{t+1} = \begin{cases} s(\hat{T}_\phi^{k_t}(\hat{s}_t, a_t, h_{t-1})) & \text{if} \quad t < G \\ s(\hat{T}_\phi^{k_t}(\hat{s}_t, \pi_\theta(\hat{s}_t[\hat{s}_{t-1}, ..., \hat{s}_{t-G}]), h_{t-1}^{k_t})) & \text{else} \end{cases}$$
$$\text{and} \quad h_t^k = \begin{cases} h(\hat{T}_\phi^k(\hat{s}_t, a_t, h_{t-1}^k)) & \text{if} \quad t < G \\ h(\hat{T}_\phi^k(\hat{s}_t, \pi_\theta(\hat{s}_t[\hat{s}_{t-1}, ..., \hat{s}_{t-G}]), h_{t-1}^k)) & \text{else.} \end{cases}$$

where $r(\cdot)$ describes the portion of the model's output that corresponds to the reward prediction, and k_t denotes the index of the transition model that currently predicts the lowest reward. During rollouts, the predicted state of that model is used to carry on with the trajectory. Compared to MOOSE, this means a little more conservatism is embedded into the reward estimation.

Instead of optimizing Eq. 5.17 w.r.t. the policy parameters θ by gradient descend, we choose PSO [76, 80] to search for well performing policies. We vow for PSO over other gradient-free algorithms due to its attributed robustness and the prior experiences on the industrial benchmark published in related works. The algorithm maintains a population of policies with their positions (parameter values) as well as velocities (changes in the parameter space). In every iteration, all policies are evaluated using Eq. 5.17, and afterwards their positions and velocities are updated. The velocity update contains a social and a local component, leading policies to be pulled towards their personal best as well as the best parameter values of their neighbours. See Section 2.6 for a more in-depth introduction to PSO.

Behavior Constraining What remains is to constrain the policy search, such that only policies that do not deviate too strongly from the dataset distribution can be found. Naively, one might transfer the VAE based action space penalty from MOOSE over to WSBC by augmenting Eq. 5.17 accordingly. However, contribution [C] found that it is much more effective to penalize the search of the policy directly in neural weight space. To this end, we train a model of the generating policy $\beta_\psi(\cdot)$ which has the exact same architecture, and consequently the same parameters, as the policy π_θ which we aim to train. β is trained by minimizing the mean squared error on the dataset $\mathcal{D}$ with N interactions:

$$L(\psi) = \frac{1}{N} \sum_t [\beta_\psi(s_t[st-1, ..., s_{t-G}]) - a_t]^2 \tag{5.18}$$

We then use the found parameters of the generating policy to constrain the search space of PSO to a small region around the initial parameters ψ:

$$\theta^* = \max_\theta \mathbb{E}_\pi[R] \quad \text{s.t.} \quad \forall i : ||\theta_i - \psi_i|| < d. \tag{5.19}$$

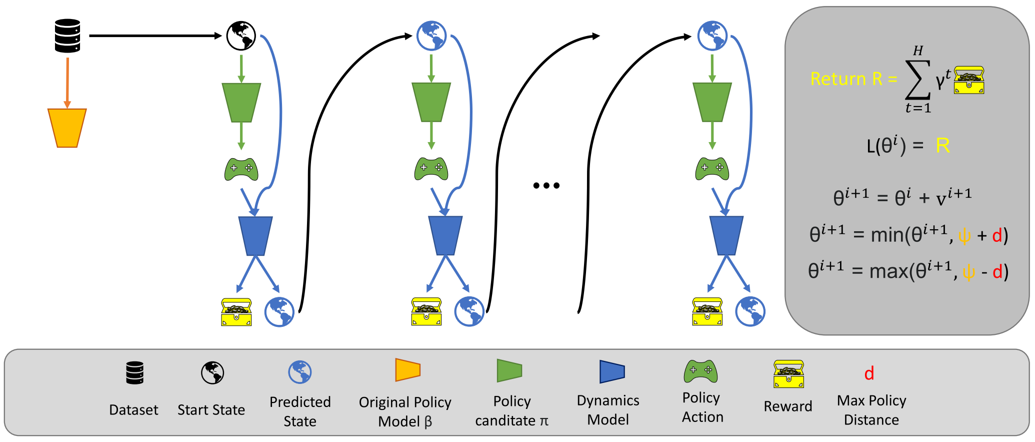


Figure 5.4: Visualization of the WSBC training procedure. First, a fixed set of start states is sampled from the dataset (black, top-left), and provided to the target policy (green) to propose an action. This action is then provided to the transition ensemble (blue) in order to predict the reward as well as the future state. The future state is then used to start the procedure anew. In the end, the policy is regularized by only searching in the space around the trained generating policy (orange+red).

Where d is the L1 distance threshold within which PSO is allowed to search. In order to fulfill the constraint, WSBC augments the update rule in Equation 2.55, clipping particle positions to lie within the allowed range:

$$\theta_n^{i+1} = \min(\max(\theta_n^i + v_n^{i+1}, \psi - d), \psi + d) \tag{5.20}$$

with

$$v_n^{i+1} = v_n^i + c_1 \cdot r_1 \cdot [b_n^i - \theta_n^i] + c_2 \cdot r_2 \cdot [\hat{b}_n^i - \theta_n^i] \tag{5.21}$$

Note that $\psi - d$ & $\psi + d$ are vectors and the min and max operations are applied element-wise. See Fig. 5.4 for a visualization of the WSBC policy training and algorithm 2 for pseudocode of the WSBC algorithm.

The idea of PSO and other population based optimization algorithms is to model social behavior in the search for better solutions. Particles are in one way or the other pulled towards solutions that have previously been found to perform well either by other particles or by themselves. On the other hand, such algorithms need to feature some sort of exploratory behavior, so that they do not immediately converge to the best particle in the initial distribution. In PSO, velocities during the first few iterations build up to become rather large, since the particles are spread throughout the entire search space. Throughout the course of training, the exploratory behavior becomes less, i.e. in PSO the velocities slowly degrade over the iterations, since particles focus more and more on the regions around the previously best found solutions. The underlying assumption that makes the search work is thus that good solutions lie in close proximity to other good solutions in the parameter space - otherwise it would not make sense to pull particles towards them.

In WSBC, we employ a similar assumption to perform the needed policy regularization to make the algorithm work in the offline setting: By simply constraining the parameter space to a small region around the parameters of the generating policy, the algorithm effectively only searches for solutions sufficiently close to the generating policy, so that it can reasonably be assumed that the transition models are able to accurately predict trajectories and returns resulting from applying the found solutions to the environment. The assumption WSBC makes with regard to the search space is thus that particles with similar parameters (i.e. those that lie within the proximity defined by d), also exhibit similar behavior.
While prior algorithms like MOOSE, MOPO, BRAC, BEAR, etc. need to augment their loss function / return computation and add a penalty term based on the action space divergence between the policies, or the uncertainty in the transition models, WSBC can effectively optimize the same objective as the corresponding online algorithm. Instead of adding a regularization term to the loss, we have moved this critical part of offline RL algorithms to be a constraint under which the objective can be optimized. The changes necessary to adapt the online algorithm to become offline compatible can be considered minimal: The loss function stays the same, and we only add a very simple to implement (see Eq. 5.20) constraint.

Algorithm 2 WSBC

```
1: Require Dataset $D$, randomly initialized parameters $\theta$, $\phi$, $\psi$, horizon $H$, number of
   PSO update epochs $U$, number of particles $P$, neighbourhood architecture
2: // dynamics and original policy models can be trained supervised and independently
   of other components
3: train original policy model $\beta_\psi$ using $D$ and Equation 5.18
4: train dynamics models $\hat{T}^k_{\phi^k}$ with $D$ and Equation 5.16
5: sample start states $S_0 \sim D$
6: initialize particle positions $\theta^0_p$ and velocities $v^0_p$, $p \in 1, ..., P$
7: for j in 1..U do
8:     for p in 1..P do
9:         update velocity $v^j_p$ according to Eq. 5.21
10:        update position $\theta^j_p$ according to Eq. 5.20
11:        init $L(\theta^j_p) = 0$
12:        for t in 1..H do
13:            calculate policy actions $a_t = \pi_{\theta^j_p}(s_t, \lambda)$
14:            $r_t, s_{t+1} = \hat{T}^k_{\phi^k}(s_t, a_t) \quad s.t. \quad k = \arg\min_k \{r(\hat{T}^k_{\phi^k}(s_t, a_t))\}$
15:            $L(\theta^j_p) += \gamma^t \lambda r_t$
16:            update $b^j_p$ and $\hat{b}^j_p$
17: return particle $\theta$ with best overall observed return $L(\theta)$;
```

5.2.2 Experimental Comparison of WSBC with Prior Methods on the Industrial Benchmark & MuJoCo

Similarly to the experiments conducted for MOOSE, we examine the performance of WSBC on the 16 industrial benchmark datasets as well as on the three MuJoCo tasks. In Tables 5.4 and 5.3, we again report 10th percentile performance, and in Figures 5.6 and 5.5 we show the corresponding training curves.

The different regularization technique, constraining the neural weights of the target policy to be close to those of the generating policy, appears to work rather well and robustly: On the industrial benchmark datasets, WSBC outperforms all other algorithms, including MOOSE, on 11 out of the 16 datasets, mostly by significant margins. In the MuJoCo tasks, WSBC does not perform as well, however still takes second place on the Hopper and Walker benchmarks, outperforming all prior algorithms except for MOOSE. In the Swimmer environment, WSBC also takes the rank behind MOOSE, meaning it is also outperformed by BRAC & BCQ.

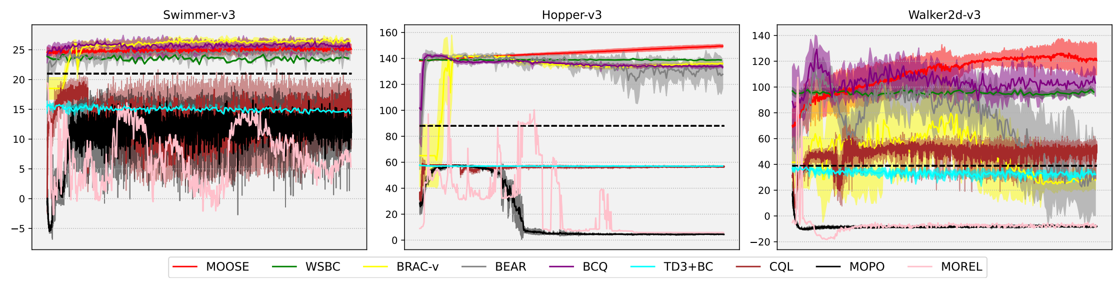


Figure 5.5: Mean performance ± standard deviation in MuJoCo experiments. Dashed line represents original batch performance.

Regarding the four main investigation issues, we find that WSBC appears to generalize both over (i) different environments as well as (ii) different dataset modalities, since it produces well performing policies in different tasks / environments as well as with vastly different amounts of exploration contained in the corresponding datasets. While the (iii) absolute performance seems to even surpass that of MOOSE, we note it goes hand in hand with (iv) slightly worse stability of the learning process, exhibiting higher uncertainties especially on the bad-0.0 & bad-0.2 datasets as well as an interesting underperforming outlier on mediocre-0.6. Overall however we still assess WSBC's stability as better than that of most prior algorithms, likely due to the fact that like MOOSE, it does not employ a value function for the return estimation.

Experimental Details The transition models used in WSBC are all simple RNNs with a hidden dimensionality of 30 for the industrial benchmark experiments and 100 for the MuJoCo tasks. The models were trained for up to 3000 epochs with a learning rate of 1.25×10^{-5}, a learning rate decay of 0.95, a batchsize of 1024, and a patience of 128,

	Swimmer	Hopper	Walker2D
DDPG	-10.2 (1.8)	0.4 (4.5)	-12.7 (3.9)
BRAC-v	**25.9 (0.1)**	134.0 (0.2)	17.1 (0.9)
BEAR	22.9 (0.1)	117.4 (1.3)	-7.0 (3.8)
BCQ	24.8 (0.1)	132.0 (0.2)	91.9 (1.0)
TD3+BC	13.9 (0.1)	56.7 (0.1)	28.9 (0.5)
CQL	8.5 (0.2)	56.1 (0.1)	42.9 (0.3)
MOPO	8.3 (0.1)	4.2 (0.1)	-8.8 (0.1)
MOReL	4.1 (0.1)	5.8 (0.1)	-8.3 (0.1)
MOOSE	24.2 (0.1)	**147.2 (0.2)**	**113.7 (0.6)**
WSBC	23.5 (0.1)	138.5 (0.2)	95.7 (1.5)

Table 5.3: Robust performance of the algorithms in the MuJoCo experiments. To assess robustness, 10th percentile performance is shown together with its standard error. Final 10% of performance values across five seeds taken into account. While WSBC outperforms the prior baselines on Hopper and Walker, it does not perform as well in them as MOOSE. In the Swimmer environment, WSBC still performs well, but similarly to MOOSE it is outperformed by BCQ and BRAC.

using the Adam optimizer. $G = 30$ history steps as well as $F = 50$ future steps were used for the industrial benchmark, while $G = 3$ was used for MuJoCo. Even though the latter environments are deterministic, models with a few history steps obtained higher validation returns on the future predictions than those that only used the current state.

The generating policy model was a simple MLP model with a single hidden layer of size 20 for the industrial benchmark and size 40 for the MuJoCo datasets. The policies needed to have a lower amount of parameters since it makes learning without gradients much easier (covering the parameter space needs exponentially more samples with each additional dimension). However, we didn't observe issues with this since both the reproduction of the generating policy was still sufficiently accurate and the target policies of the same architecture were able to perform well despite their lower capacity. The target policies were searched for using PSO with a population size of 200 particles for 100 epochs in the limited search space $\psi \pm 1.25 \times 10^{-2}$ for the IB and $\psi \pm 6.25 \times 10^{-3}$ for MuJoCo (the distance needs to be lower in MuJoCo due to the higher complexity policies). During optimization, ring neighbourhoods of size 31 were employed, so that each particle had 30 neighbours. Each particle used the same 200 starting states, since sampling would expose PSO to noise, leading to "lucky" particles taking the top spots, but not actually performing well in the real environment. The rollouts as well as the evaluations were conducted for 100 steps with a γ of 0.97 for the industrial benchmark and 0.99 for the MuJoCo datasets. ReLU activations were used everywhere except for the hidden layer of the RNN, which employed a tanh function.

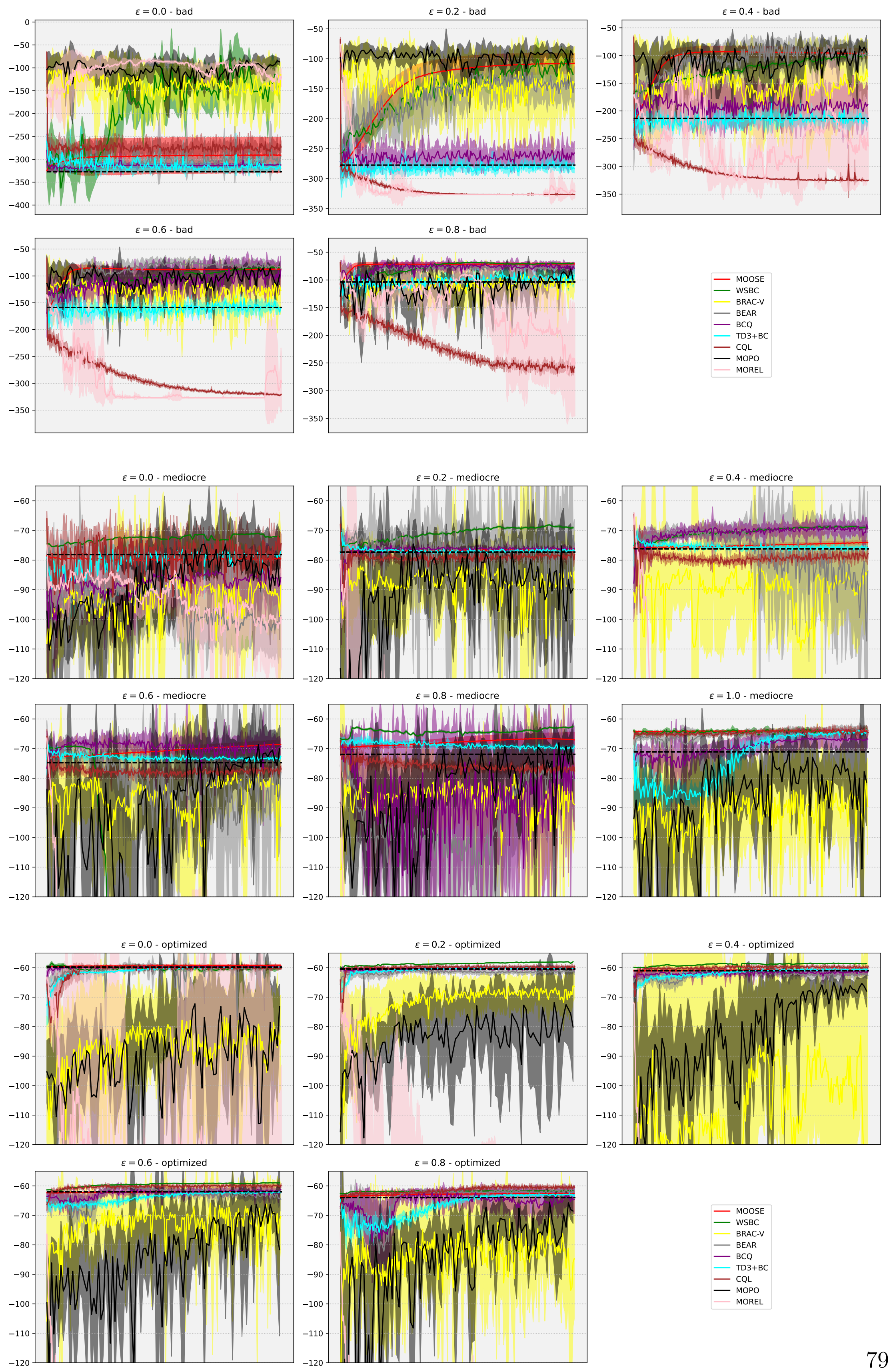


Figure 5.6: Mean performance ± standard deviation in IB experiments. Dashed line represents original batch performance.

Table 5.4: Tenth percentile performances of evaluated Offline RL algorithms and their standard error. Final 10% of performance values across five seeds taken into account. WSBC outperforms all other algorithms including MOOSE on 11 out of 16 datasets, while MOOSE & MOPO each take two, and CQL takes the top spot on a single dataset.

	$\varepsilon =$	0.0	0.2	0.4	0.6	0.8	1.0
Bad	DDPG	-326 (25)	-383 (5)	-383 (7)	-383 (7)	-383 (7)	
	BRAC	-266 (8)	-248 (8)	-265 (8)	-190 (5)	-132 (4)	
	BEAR	-322 (3)	-180 (4)	-129 (3)	-98.3 (1.3)	-88.7 (0.9)	
	BCQ	-313 (1)	-286 (2)	-234 (5)	-137 (4)	-87.1 (1.2)	
	TD3+BC	-326 (2)	-289 (2)	-230 (2)	-173 (2)	-113 (2)	
	CQL	-292 (2)	-327 (1)	-326 (2)	-323 (1)	-271 (1)	
	MOPO	**-124 (5)**	**-110 (4)**	-140 (8)	-131 (5)	-119 (5)	
	MOREL	-144 (2)	-327 (2)	-327 (8)	-327 (7)	-327 (11)	
	MOOSE	-322 (5)	-126 (2)	-109 (1)	-90.3 (0.3)	-73.3 (0.3)	
	WSBC	-134 (35)	-118 (12)	**-102 (3)**	**-84.9 (0.1)**	**-70.0 (0.3)**	
Mediocre	DDPG	-960 (34)	-960 (36)	-960 (37)	-961 (24)	-960 (48)	-178 (3)
	BRAC	-114 (2)	-103 (2)	-112 (2)	-99.0 (2.6)	-101 (3)	-107 (2)
	BEAR	-112 (1)	-103 (5)	-124 (3)	-105 (3)	-99.7 (1.6)	-66.3 (0.2)
	BCQ	-103 (1)	-77.1 (0.1)	-72.7 (0.3)	-75.9 (0.7)	-107 (2)	-69.2 (0.4)
	TD3+BC	-79.7 (0.6)	-77.7 (0.1)	-76.8 (0.1)	-74.3 (0.1)	-70.7 (0.1)	-65.4 (0.1)
	CQL	-89.5 (1.2)	-80.5 (0.2)	-80.8 (0.2)	-79.5 (0.2)	-78.9 (0.2)	-66.0 (0.2)
	MOPO	-102 (4)	-119 (5)	-81.4 (2.3)	-86.1 (2.5)	-92.5 (4.5)	-105 (7)
	MOREL	-122 (2)	-327 (6)	-327 (4)	-327 (7)	-327 (1)	-327 (12)
	MOOSE	-83.0 (0.3)	-77.1 (0.1)	-75.1 (0.1)	**-70.5 (0.2)**	-68.7 (0.3)	-64.3 (0.1)
	WSBC	**-71.1 (0.1)**	**-68.5 (0.2)**	**-68.8 (0.2)**	-243.1 (0.1)	**-62.9 (0.1)**	**-63.8 (0.1)**
Optimized	DDPG	-416 (24)	-382 (11)	-257 (21)	-312 (30)	-168 (3)	
	BRAC	-113 (3)	-77.3 (0.8)	-158 (6)	-89.2 (1.7)	-116 (3)	
	BEAR	-60.5 (0.3)	-62.5 (0.1)	-64.4 (0.2)	-66.0 (0.3)	-63.0 (0.1)	
	BCQ	-60.1 (0.1)	-60.9 (0.1)	-62.7 (0.1)	-63.6 (0.2)	-72.4 (0.5)	
	TD3+BC	-60.3 (0.1)	-60.6 (0.1)	-61.1 (0.1)	-62.7 (0.1)	-63.6 (0.1)	
	CQL	-60.9 (0.1)	-60.6 (0.1)	-60.6 (0.1)	-60.6 (0.1)	**-61.3 (0.1)**	
	MOPO	-126 (9)	-102 (4)	-72.0 (0.9)	-80.6 (3.3)	-90.3 (2.8)	
	MOREL	-279 (8)	-327 (8)	-327 (1)	-327 (1)	-327 (7)	
	MOOSE	**-59.7 (0.1)**	-60.3 (0.1)	-60.8 (0.1)	-62.1 (0.1)	-62.7 (0.1)	
	WSBC	-60.2 (0.3)	**-58.2 (0.1)**	**-58.6 (0.1)**	**-59.0 (0.1)**	-61.7 (0.1)	

5.2.3 Discussion

In this section we introduced the WSBC algorithm and compared its performance, stability, and generalization capabilities with prior offline RL algorithms as well as the newly proposed MOOSE algorithm from the previous section. WSBC outperforms all other algorithms considered in 11 out of the 16 industrial benchmark datasets, while exhibiting slightly larger uncertainties than MOOSE, who takes the second place in most of these datasets. Since WSBC also performs well in the MuJoCo domain, we assess it good generalization capabilities when faced with different policy baseline and exploration scenarios as well as different environments and tasks. Overall, we thus deem WSBC to be a good choice for offline RL in practice.

Looking at both algorithms proposed in this Chapter together, they perform best in 15 out of 19 datasets and take second place in the remaining four, making a strong case for usability in real world offline tasks. While the differences in regularization techniques appear to be significant and have their strengths in different environments, datasets or metrics (e.g. higher variance of WSBC in bad-0.0, bad-0.2 and underperformance in mediocre-0.6, but overall higher performance on the other datasets; MOOSE with better performance in the MuJoCo environments), we would like to point out that overall it seems to be beneficial to use transition models in offline RL, and at the same time abstain from employing a value function. In the experimental comparison, both the purely value function based, as well as the hybrid approaches yielded higher variance and overall lower 10th percentile performance, likely in part due to the associated training instabilities. While there exist environments / tasks e.g. with very long horizons for which model-rollout based return estimation can become impractical, we find that for environments in which they are feasible, they bring key advantages to the table.

For the wind turbine operator, MOOSE and WSBC constitute a large step forward: Using these algorithms, no online learning phase is required, meaning lower risk of damaged equipment and much lower risk of underperforming the prior controler due to the lack of exploratory actions. Looking at the results of MOOSE and WSBC, the algorithms rarely ever perform below the generating policy (MOOSE slightly on mediocre-0.0, WSBC on mediocre-0.6), meaning that the operator can confidently use them in order to improve the systems electricity yield - in contrast, most other tested offline RL algorithms underperform on some datasets. While real world datasets can look differently and the results are no guarantee, the algorithms still significantly improve usability in real world scenarios.

6 Proximity Conditioned Policies

In this Chapter, we motivate and develop the idea of proximity conditioned policies, i.e. policies that condition on the proximity to the generating policy. We devote two separate sections 6.1 & 6.2 to the motivation of the concept, because it can be derived in two completely separate ways. We will further examine the suitability of the resulting policies for offline RL on the industrial benchmark and MuJoCo datasets presented in Chapter 3 and compare their performances with those of other state of the art offline RL algorithms presented in Chapters 2.5 & 5. Finally, we discuss potential use cases for proximity conditioned policies and how to find good trade-offs between proximity and performance. The material presented in this Chapter corresponds to the contributions [E & F].

6.1 The Difficulty of Finding the Correct Hyperparameters in Offline RL

Rather recently, offline RL algorithms have shown that it is possible to learn policies for continuous control problems purely from static datasets and without any environment interaction [8, 54, 52, 36, 43]. In doing so, they have already alleviated a large obstacle on the path towards more deployments of RL policies in real-world (physical) systems, because environment interaction is often an issue due to safety concerns, opportunity cost, and scalability (many online RL algorithms simply require too many interactions to be realistically collected by a single physical system).

However, offline RL algorithms also still face a major issue: They are generally unable to perform hyperparameter tuning for most of their hyperparameters since no environment interaction is possible until final deployment. While model-based / hybrid algorithms can at least tune the hyperparameters of their transition models since it is learned entirely supervisedly, model-free approaches do not have any option to tune them. While for some parameters reasonable guesses can be made, such as the policy architecture or the gradient descent algorithm, one of the arguably most important hyperparameters in offline RL is the regularization term that moves the trained policy closer to the generating policy and in doing so enables offline learning. Different techniques exist for this regularization, but all offline RL algorithms have such a parameter: In BRAC it is the α parameter that controls how large the penalty for high KL-divergences between generating and trained policy is, in BEAR, λ controls the magnitude of the MMD-based penalty, in TD3+BC, α determines the strictness of the behavior cloning regularization, in CQL, α decides how much to devalue state-action pairs that have not been seen in the dataset, in MOPO, λ settles how much the reward is decreased by the uncertainty estimate, in MOReL, if the model-disagreement exceeds the value of ϵ the trajectories will be terminated, in MOOSE, λ controls the influence of the VAE penalty, and in WSBC, d is the maximum distance between generating policy and any policy candidate. A small exception is BCQ, which features three immediate regularizing hyperparameters: (1) Φ, which is the maximum value by which the perturbation model ξ is allowed to change the selected action, (2) The number of actions sampled from the generating policy model, and (3) the test-time maximum values of the standard deviation of the VAE latent space parameters. The bottom line is that each offline RL algorithm features at least one central hyperparameter that determines how strictly to regularize the policy towards the generating policy. Many different names have been proposed for this, such as conservatism, pessimism, risk-awareness, uncertainty-avoidance, etc. In the following, we will refer to this concept as proximity, as in the proximity of the trained policy to the generating policy.

Since prior work has found that offline RL algorithms can be rather sensitive with respect to their hyperparameters [73], particularly the proximity regularizing parameter, practitioners are in practical applications faced with a dilemma: How should they choose the α / λ / ϵ / d / Φ, etc.? The standard hyperparameters may have worked on a set of academic benchmark datasets, however the specific task / dataset at hand might exhibit very different characteristics. The practitioners thus risk being either overly conservative or too adventurous in their choice, both of which would be a problem: Overly liberal hyperparameter choice would lead to the policy exploiting the return estimator, visiting state-action pairs that cannot be accurately accounted for and ultimately performing much worse in the real environment than expected. But also too conservative policies are a problem since they cannot lead to significant improvements, which would in practice likely result in discontinuation of the project due to lack of meaningful gains.

There is thus an obvious need for a way to tune hyperparameters in the offline RL context, specifically the proximity of the new policy to the generating policy. One way of addressing this need is by offline policy evaluation or offline policy selection (OPE / OPS): The idea is that not only the training process, but also the evaluation can be performed entirely without environment interaction, by e.g. using a different kind of transition model than the one used in training or by training a Q-function specifically for the policy to be evaluated and correct for the discrepancy in state-action visitation via importance sampling [131, 132, 133, 134, 135]. The main problem with these techniques is that the issue of not knowing what they don't know always remains: Some parts of the state-action space are simply not (well) explored - if a policy enters this region then the offline policy evaluation is likely equally inaccurate as the return estimator used during training. Of course there exists something between unexplored and completely covered by the generating policy, however it is unclear which general properties a return estimator during evaluation should have that one during training cannot have. While entirely offline tuning thus appears problematic, works such as [136, 137] can make OPE much more viable once a budget for online evaluations exists. Similarly, works exist that adaptively train the policy online after an offline pre-training phase [138, 139, 140]. Among other ideas, meta reinforcement learning [141, 142, 143], which is mostly associated with adapting quickly to new tasks, can in this context be used to quickly adapt the offline policy to the online environment.

Since we assume that no actual online learning is admissible (domain experts should have the ability to examine the policy before deployment in order to mitigate the associated risks), we propose to learn policies that condition on the proximity hyperparameter: Instead of learning only a single trade-off between proximity and estimated return, which might be the wrong one since regularization should be more or less strict, we aim to learn a continuous spectrum of all possible trade-offs.

6.2 The Role of the User

One of the greatest obstacles on the path to more RL policy deployments in real (physical) systems besides learning without interaction is the fact that offline as well as online RL algorithms for continuous control mostly ignore the presence of domain experts, or operators, of the system they seek to optimize. These operators can be seen as users of the final policy, so a solution should be appealing to them. Instead, most of (offline) RL literature can be seen as trying to automate away domain experts. We believe that it is the obligation of an AI system to provide utility and control to the user, not only because they can be seen as gatekeepers in this scenario, but also because it is likely that the combination of human expert together with the AI system can make better choices than either of them alone. In other machine learning disciplines, it has already been established that users are crucially important to solving a task [144, 145, 20, 19, 146]: For example in medical diagnoses, even though many methods have shown to outperform human experts on benchmark datasets for e.g. skin cancer recognition [147], few people want to be diagnosed by a completely automated AI solution. As a consequence, methods to team up human and AI have been developed, where medical images from past patients together with their diagnoses can be retrieved that are similar to the current patient's image, thereby aiding the medical professional in their diagnosis [145]. Similarly, the multi-target reinforcement learning framework can be used to provide users with control about the goal the agent should be fulfilling at runtime [21, 22, 23]. Methods like these are a great example of AI systems that behave more like an intelligent tool, and less like a replacement for the user, which is a general theme that researchers have called for in future AI systems [29, 30]. Ideally, human-AI teams should benefit from each others, by combining the high level of automation that is provided by the AI system with the high level of domain knowledge and control that the users can bring to the table. Resulting methods should be more safe, transparent, and trustworthy than the ones we currently have. Ideally, they will also lead to improved performance.

A whole other reason to adopt a more user centric approach for AI systems is that it may in the future be required by law. The EU high level expert group on AI has already recognized human autonomy and oversight as key concepts in their ethics guidelines for trustworthy AI [28]. Users of an AI system should thus have a right to act autonomously and control the AI system, not vice versa. As the use of AI technologies in industry and society grows, these concepts might be enforced by law, requiring change in perspective and enabling the users to stay in control.

As a first step to providing the user with more control in the offline RL setting, we propose to train adaptive policies that condition on the proximity hyperparameter of the algorithm, which regularizes the trained policy towards the generating one. That way, users can quickly adapt the behavior of the policy if it is not acting in a way that the user would like to endorse. This "disliked" behavior does not need to manifest itself in a lower return, but could simply be due to unfamiliarity or the concern that the policy might enter a potentially dangerous state, which the user can recognize before. When the proximity conditioned policy is trained, users can easily and quickly alter their degree of conservatism based on what they observe, since no retraining of the policy is required and any continuous step size is supported.

In the wind turbine example: While the operator already has good chances of successfully deploying an offline RL policy built with MOOSE or WSBC, a few pain points remain. While the operator is keen on using the new technology, they do not really trust the fixed nature of the derived policies - after all, there is no guarantee that the trained policy will outperform the prior controller. Ideally, the operator would still like to be able to influence the behavior of the policy after training if they don't like it. With the idea of proximity conditioned policies, this becomes possible: The operator can safely deploy the policy and condition it so that it closely resembles the prior controller, and then gradually give it more freedom to move away from it in order to improve the yield - if at any point the operator sees behavior that is undesired (e.g. vibrations in the turbine, risky high rotational speeds, or other things that are not necessarily directly reflected in the reward function), they may immediately go back to a more constrained policy without retraining.

6.3 Training a Discrete Number of Offline Policies to Cover the Trade-off Space

At first, we look at a trivial solution to achieve the same goal as proximity conditioning: Using an existing offline RL algorithm, and simply training not only with a single value for the proximity hyperparameter, but instead with a discrete number of different parameters to span the space of possible trade-offs between proximity and performance. As a well known offline RL algorithm, we choose MOPO to perform this initial experiment: We choose a subset of three datasets from the industrial benchmark datasets and train MOPO with uncertainty penalizing parameters $\lambda = \{0, 0.1, 0.2, ..., 2.5\}$, to see whether this method could serve as a replacement for the proximity conditioned policies. Users

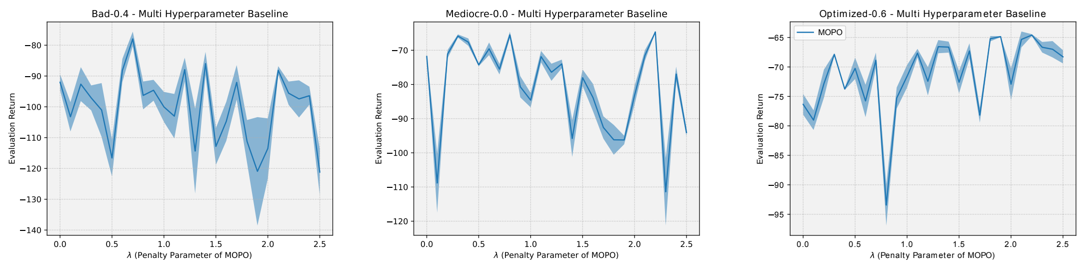


Figure 6.1: Prior offline RL algorithms, such as MOPO do not necessarily behave consistently when trained across a range of penalizing hyperparameters.

could be provided with the resulting collection of policies and switch between them in order to stay closer to or stray further away from the generating policy. Two downsides of the approach are immediately clear: (1) The computational complexity grows linearly with the number of trade-off choices which we want to provide to the user because we need to train an individual policy for each trade-off, and (2) since we only train a discrete amount of policies we cannot be sure whether the desired trade-off by the user is contained in the set of policies that we provide. Particularly the combination of both issues becomes tricky because one could try to solve (2) by simply training more policies, but then the complexity issue (1) becomes even worse and vice versa.

The result of our experiment can be seen in Figure 6.1: For each penalty factor λ, we plot the mean and standard deviation of the resulting performance in the real environment. Note that low λ values correspond to unconstrained policies, while high λ values correspond to policies more regularized towards the generating policy. The default value in MOPO is $\lambda = 1$. Interestingly, we observe that the trade-offs are not at all consistent: One would expect that the policies would all start rather low on the right side (i.e. the constrained end), since they can pretty much only imitate the generating policy. Then, one would at least in the bad-0.4 and optimized-0.6 datasets expect an increase in performance the further the penalty is decreased, since enough exploration is contained in these datasets, so that the return estimators should be able to improve on the baseline. Then, more towards the left end of the low contraint values, one would expect degrading or rapidly changing behavior since the return estimators cannot assess the performance of the unconstrained policies accurately enough. Instead, what we observe is a constant zigzag of significant magnitude across the range of trade-off parameters for all three datasets, even with small changes in the parameter. We thus note a third downside to this naive approach: When the policies are trained in isolation instead of jointly as a single network that forces a smooth interpolation among them, the resulting collection can be very inconsistent and thus not particularly useful for a user searching for a decent trade-off.

6.4 Enabling Users via Proximity Conditioning

The simplest method to find a likely very familiar policy for the user is to perform behavior cloning on the interactions in the dataset, however since there is no improvement available in simply copying the behavior in the data, this method alone is of very limited use. On the other extreme end would be unregularized reinforcement learning from data, however as we have shown in Section 2.4, algorithms not explicitly built to regularize policies to enable learning from static datasets usually fail when deployed in the real environment. In between the two extremes lies offline reinforcement learning, which combines RL with a regularization inspired by behavior cloning, moving the policy into a regime where return estimators can (hopefully) accurately predict performance. However, the problem still remains that there exists no general way to correctly choose the parameter balancing the two components, because we are not allowed to evaluate on the real system and because offline policy evaluation is still an open issue [131, 132, 133, 134, 135, 148]. We thus aim to develop a solution that can adaptively change the proximity hyperparameter at runtime in order to enable users to quickly find a well performing value.

As we have shown in the previous section, the simple solution of training many policies with a prior offline RL algorithm on a discrete number of proximity hyperparameters has at least significant downsides. We thus aim to develop a method that mitigates the examined disadvantages: Our solution should enable a continuous choice of hyperparameters, should be computationally feasible, and most importantly it should be consistent over the range of proximity values in order to enable a user to navigate the hyperparameter landscape sufficiently. We term the approach LION - Learning in Interactive Offline eNvironments.

6.4.1 Learning in Interactive Offline Environments (LION)

In LION, we train three components:

- A model of the generating policy $\beta_\psi(s)$
- An ensemble of K transition models $\{\hat{T}^k_{\psi^k}(s,a)|k=0,...,K-1\}$
- A proximity conditioned policy $\pi_\theta(s,\lambda)$, where lambda represents the desired proximity of the target policy to the generating one.

The first two, the generating policy model β_ψ and the transition ensemble $\{\hat{T}^k\}$, are trained entirely in isolation and fully supervised. The generating policy β_ψ as well as the target policy π_θ are both deterministic MLPs, while the transition model architecture is chosen according to the requirements of the environment, either as a recurrent network or also as an MLP. The setup is thus in the beginning very similar to WSBC, i.e. the corresponding losses are given by:

$$L(\psi) = \frac{1}{N} \sum_{s_t, a_t \sim \mathcal{D}} [a_t - \beta_\psi(s_t)]^2 \tag{6.1}$$
$$L(\psi) = \frac{1}{N} \sum_{s_t, ..., s_{t+G},\ a_{t+G+1} \sim \mathcal{D}} [a_{t+G+1} - \beta_\psi(s_t, ..., s_{t+G})]^2$$

and

$$L(\phi_k) = \frac{1}{N} \sum_{s_t, a_t, s_{t+1} \sim \mathcal{D}} \left[s_{t+1} - \hat{T}^k_{\phi_k}(s_t, a_t)\right]^2 \tag{6.2}$$
$$L(\phi_k) = \frac{1}{N} \sum_{t \sim \mathcal{D}} \sum_{f=1}^{F} [s_{t+G+f+1} - \hat{T}^k_{\phi_k}(s_t, a_t, \ldots s_{t+G}, a_{t+G}, \ldots \hat{s}_{t+G+f}, a_{t+G+f})]^2$$

where the first formulations in Equations 6.1 & 6.2 represent the case when no history of states is considered and the latter ones the case where it is needed to estimate the hidden state of the environment. We do not explicitly denote recurrence and the hidden state of the RNN as in Eq. 5.16 in order to remove clutter, G is number of history steps used to build the hidden state of the RNN, while F is the number of steps it needs to predict into the future. During this stage of training, architecture and other components of the models could be optimized by holding out a validation set from the original training data.

When generating policy model and transition ensemble are trained satisfactorily, the algorithm moves on to train the adaptive policy: As before, the return estimation is performed by sampling start states from the dataset $\mathcal{D}$ and then using the transition model and the target policy to generate virtual rollouts and summing up the predicted rewards by the transition models. The regularization part of the algorithm is more closely related to MOOSE than to WSBC, since it is performed in the action space again, by comparing the actions the target policy took with the ones the generating policy model would have taken in the same situation and penalizing the mean squared distance between the two, i.e. we define a penalty $p(s, a)$ as:

$$p(s, a) = \frac{1}{D} \sum_{d=0}^{D} \left(a^{(d)} - \beta_\psi(s)^{(d)}\right)^2 \tag{6.3}$$

where $s^{(d)}$ is the d^{th} of D state dimensions. We will use the penalty to regularize the policy towards the generating one. Further, we define the ensemble prediction $e(s, a)$ as the minimum reward predicted among the transition ensemble members in order to estimate the trajectory return conservatively:

$$e(s, a) = \arg\min_k r(\hat{T}^k_{\phi^k}(s, a)) \tag{6.4}$$

where $r(\cdot)$ extracts the reward prediction from the transition model. Here, explicit notation of recurrence or state history is omitted - if required by the environment, Equations

6.3 & 6.4 need to be augmented similarly to Equations 6.1 & 6.2. Once $p()$ & $e()$ are defined, we can write the policy loss function as:

$$L(\theta) = -\sum_{s_0 \sim \mathcal{D}} \sum_{s_t \sim \hat{T},\ a_t \sim \pi} \gamma^t [\lambda e(s_t, a_t) - (1-\lambda) p(s_t, a_t)] \qquad a_t = \pi_\theta(s_t, \lambda) \tag{6.5}$$

The crux of the matter is that here, λ is the hyperparameter that balances the two components reward and penalty in the loss function, but is at the same time also an input of the target policy that is being trained. This way, the policy is enabled to learn the connection between desired proximity on the input side and the accordingly correct behavior that optimizes the loss function for that specific value of λ. What is left is to provide the policy with many different λ during training: During each policy update, we sample not only the start states from $\mathcal{D}$, but also randomly sample a different λ for each individual trajectory from a beta distribution:

$$\lambda \sim \mathrm{B}(\alpha, \beta) \tag{6.6}$$

where $\alpha = \beta = 0.1$ and consequently $\lambda \in (0, 1)$. Similarly to [149], we find that the weighting parameter should not be sampled from a uniform random distribution but rather from one that puts more probability mass on the edges of the distribution. It appears that emphasizing the edge cases of the desired behavior range is more important and that interpolation between them is easier than learning the "extreme" behavior itself. See Fig. 6.10 and Section 6.6.2 for a brief comparison of distribution parameters. Due to this procedure, LION is optimized in a gradient based fashion similarly to MOOSE, since if we were to adopt WSBC's gradient-free optimization, we would either face severe computational overhead or could only select a fixed, discrete set of λ values to train for. See Fig. 6.2 for a visualization of the training process.

An advantage of formulating the return estimation in the proposed purely model-based fashion without involving a value function is the stability during training: Since the transition model and the generating policy can be trained supervisedly in isolation, only the policy parameters θ are trained in an unsupervised regime, meaning fewer moving parts during this part of the optimization. If LION would instead use a value function to train the policy, this value function would also need to be condition on λ, which would lead to two conditional components that train in an intertwined process together, which seems to lead to instabilities (see Section 6.6.2).

When the policy optimization is performed as so far proposed, we observe that the generating policy is often not entirely accurately recovered when the final policy is evaluated with $\lambda = 0$. This is likely due to the inaccuracies compounding in the transition and policy models during the virtual rollouts. In order to recover the generating policy accurately, we add a penalty term that regularizes the policy towards the dataset interactions only at $\lambda = 0$:

$$L(\theta) = -\sum_{s_0 \sim \mathcal{D}} \sum_{s_t \sim \hat{T},\ a_t \sim \pi} \gamma^t [\lambda e(s_t, a_t) - (1-\lambda) p(a_t)] \quad + \quad \eta \sum_{s, a \sim \mathcal{D}} [a - \pi(s, \lambda = 0)]^2 \tag{6.7}$$

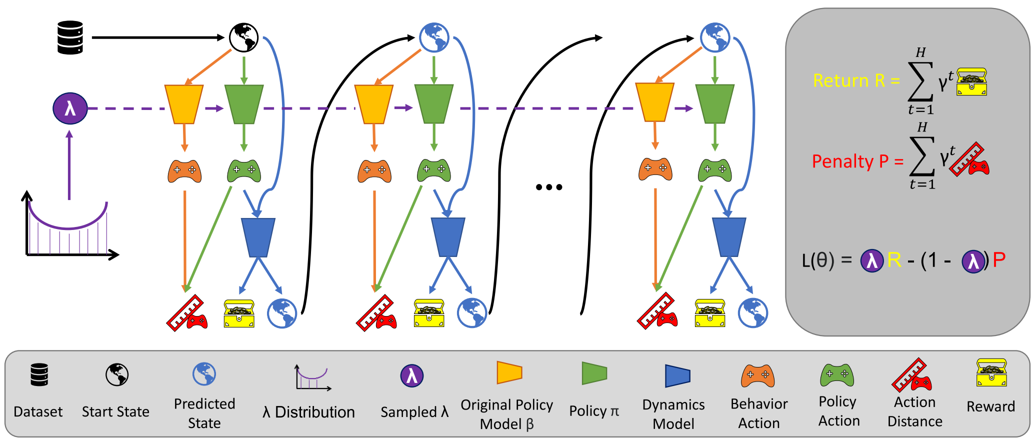


Figure 6.2: Schematic of the LION policy training: The generating policy (orange), as well as the transition model (blue) have already been trained at this point. Now, the target policy (green) is trained: The initial start states are chosen from the provided dataset (black), and used to generate an action from both the generating as well as the target policy. The target action together with the state is used to predict the step reward (yellow) as well as the future state (blue), and the generating and target policy actions are used to compute the step penalty (red). This procedure is repeated with the predicted future state as next starting point until the trajectory horizon is reached. The proximity conditioning parameter λ (purple), which controls the weighting between return and penalty, is in each trajectory sampled from a Beta distribution and provided to the target policy, so that it can learn the connection between desired proximity value and the respective behavior that optimizes the loss.

Finally, an important factor in this context is normalization: While normalizing the states to have zero mean and unit variance by employing the dataset mean and standard deviation can be considered standard, we also need to normalize the rewards, so that they live in the same magnitude as the penalties, in order to effectively enable weighting of the two components. We assume actions to be in $[-1,1]^E$, with E being the dimensionality of the actions, so that the penalty can be in $[0,4]$. Summarizing this section, we provide pseudocode for the LION algorithm in algorithm 3.

6.4.2 Deployment

At inference time, the operator who would have otherwise been in direct control of the system, may now at each time observe the behavior of the system and choose to give the policy more freedom to optimize or less freedom so that the behavior more closely resembles the known controller that was in place before the LION policy deployment. The control actions are then provided by the policy based on the current state and the chosen proximity parameter:

$$a_t = \pi_\theta(s_t, \lambda) \quad \lambda \in \text{User}(s_t). \tag{6.8}$$

Algorithm 3 LION

```
1: Require Dataset $D$, randomly initialized parameters $\theta$, $\phi$, $\psi$, lambda distribution
   parameters Beta$(\alpha, \beta)$, horizon $H$, number of policy updates $U$
2: // dynamics and original policy models can be trained supervised and independently
   of other components
3: train original policy model $\beta_\psi$ using $D$ and Equation 6.1
4: train dynamics models $\hat{T}^k_{\phi^k}$ with $D$ and Equation 6.2
5: for j in 1..U do
6:     sample start states $S_0 \sim D$
7:     sample lambda values $\lambda \sim$ Beta$(\alpha, \beta)$
8:     initialize policy loss $L(\theta) = 0$
9:     for t in 0..H do
10:        calculate policy actions $a_t = \pi_\theta(s_t, \lambda)$
11:        calculate penalty term $p(s_t, a_t) = [\beta_\psi(s_t) - a_t]^2$
12:        $r_t, s_{t+1} = \hat{T}^k_{\phi^k}(s_t, a_t) \quad s.t. \quad k = \arg\min_k\{r(\hat{T}^k_{\phi^k}(s_t, a_t))\}$
13:        $L(\theta) += -\gamma^t[\lambda r_t - (1-\lambda)p(s_t, a_t)]$
14:    update $\pi_\theta$ using gradient $\nabla_\theta L(\theta)$ and Adam
15: return $\pi_\theta$;
```

The user may observe the feedback after choosing λ and immediately change it if they deem the current behavior as too conservative or too adventurous. Crucially, no time is lost due to training a new policy from scratch or due to deployment related issues. Instead, the policy can change immediately at runtime and apply not just pre-specified values of λ, but rather any value on the continuous spectrum between 0 & 1. This way, the user can tune the policy behavior to their personal taste even though these preferences weren't known at training time.
In real world deployments, it is likely best to start by setting $\lambda = 0$, in order to check whether the policy is able to accurately recover the generating policy and since it poses the lowest risk of underperforming or otherwise violating any constraints the users might have. Then, depending on how much time the user wants to invest, how safety critical the task is, or simply how quickly the user desires, they can slowly increase λ in order to deviate more from the generating policy and let the policy try to actually optimize the performance. The incline can be continued until the user starts to see undesired behavior, and then immediately return to the last λ value where the behavior was still suitable.

6.5 Evaluating LION on a toy 2D World

Before we move on to more interesting benchmarks, we evaluate LION on a very simple 2D world in order to perform a sanity check: Do LION policies behave as expected, in the sense that they stick closely to the generating policy for $\lambda = 0$ and then deviate more

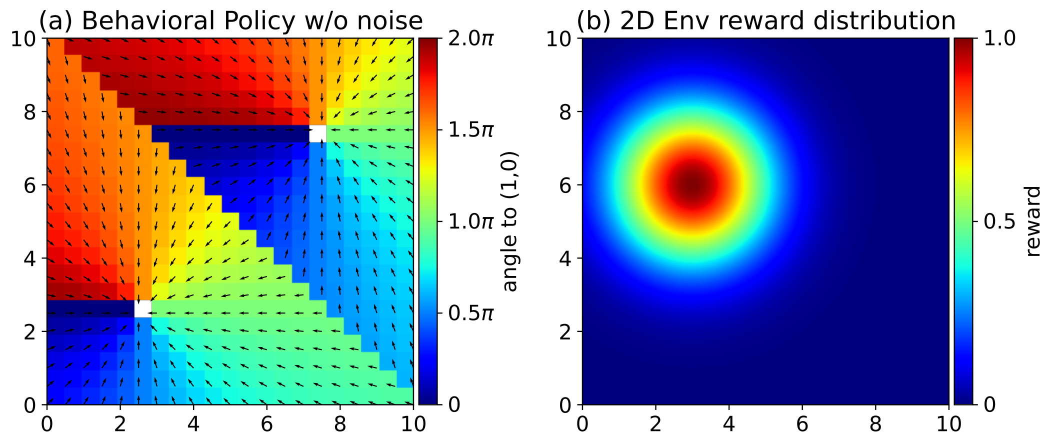


Figure 6.3: (a) Original policy for data collection - color represents action direction (b) reward distribution in the 2D environment - color represents reward value

and more from it in order to optimize return when λ is increased?

To this end, we propose the following 2D benchmark: The states of the environment represent the x & y coordinates of the agent that is placed in it. The rewards are given based on the distance of the agent to some fixed point in the 2D world: The closer the agent is to $[3, 6]^T$, the higher the reward obtained. The reward value decreases from there based on the Gaussian density function, i.e. $r(s_t) = \frac{1}{\sigma\sqrt{2\pi}} e^{-0.5((s_t-\mu)/\sigma)^2}$, where $\mu = [3, 6]^T$ and $\sigma = [1.5, 1.5]^T$. As a generating policy, we simply use an agent that either moves to $[2.5, 2.5]^T$ or $[7.5, 7.5]^T$, depending on which of the two points is closest upon randomly spawning somewhere in the $[0, 10]^2$ world. The actions correspond to proposed changes in the agents position, with a limit of 1 in any direction, i.e. $a_t \in [-1, 1]^2$. Additionally, we augment the policy with 10% random actions in order to have some exploration contained in the dataset so that the task becomes very easy for the transition models, since we only care about whether the policy behavior actually exhibits the expected variety over the range of λ values. We collect 40 trajectories with a horizon of 25, i.e. 1,000 interactions. Figure 6.3 shows the generating policy without noise together with the environment's reward distribution.

We show the behavior of the policy for each state in the 2D world, similar to Fig. 6.3(a), over the course of different λ values in Fig. 6.4. Note that the color indicates direction of the proposed policy action. In the policy maps, we can see that for $\lambda = 0$, the policy approximately recovers the generating policy (note that 10% exploration were contained in the data, so entirely accurate recovery is unlikely). In the first row, i.e. up to $\lambda = 0.3$ we do not see much of a behavior change. Starting in row two however, we can observe that in the region south-west of the $[7.5, 7.5]^T$ goal, the policy doesn't direct the agent to that goal any more, but instead has the agent move more often to the $[2.5, 2.5]^T$ goal of the generating policy. This happens because the policy has more

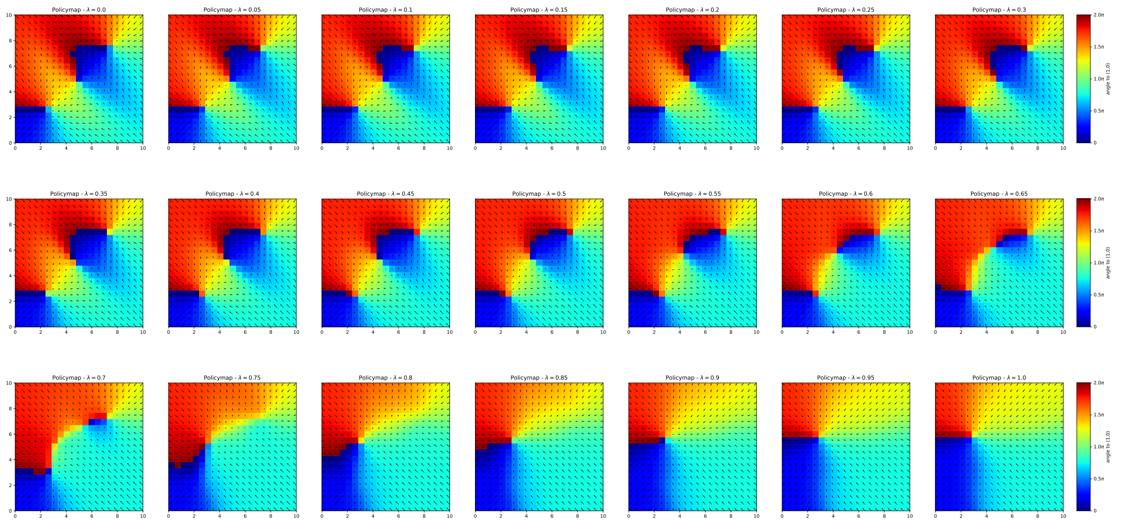


Figure 6.4: Policy maps for different λ values in the 2D environment: We move from $\lambda = 0$ (only behavior cloning) up to $\lambda = 1$ (entirely unregularized policy optimization) in steps of 0.05. The policy slowly changes its behavior by first ignoring the top right generating policy goal and rather moving the agent to the bottom left goal since it is closer to the center of the reward distribution, achieving higher returns while still maintaining some proximity to the generating policy. When λ is increased even further, the policy moves the agent closer and closer to the reward center, until it leads the agent straight on top of it at $\lambda = 1$. Note that the colors represent action directions. The best solution is only obtained at $\lambda = 1$ since the environment and dataset made it so simple - generally one expects the best performance somewhere in between the extreme λ values.

freedom to deviate from the generating policy and wants to use it to increase returns, which works because $[2.5, 2.5]^T$ is closer to the center of the reward distributing Gaussian at $[3, 6]^T$, than the other goal at $[7.5, 7.5]^T$. Yet at the same time, the policy still needs to adhere to the generating policy in a way, which is why it cannot yet move the agent directly to $[3, 6]^T$. This procedure of removing the $[7.5, 7.5]^T$ goal and only moving to $[2.5, 2.5]$ finishes around the $\lambda = 0.7$ / $\lambda = 0.75$ mark, where the policy is not entirely free to optimize yet. Over the last few steps up to $\lambda = 1$, the policy moves the goal destination slowly north, gradually improving the return as it moves from the generating policy $[2.5, 2.5]^T$ goal to the reward center at $[3, 6]^T$, where it achieves the highest possible return.

Of course, we do not always expect the policy to perform best at $\lambda = 1$, since this represents entirely unregularized policy optimization. In this case however, the environment was chosen to be sufficiently easy mainly for illustrative purposes. The simple state transition and reward distribution as well as the exploration that spanned most of the environment lead to the transition models well understanding the task and thus providing the best solution when the policy is not regularized. This will of course change in the next section, where we move to more interesting benchmarks and datasets that may not cover the state-action space sufficiently.

6.6 Experimental Comparison of LION with Prior Methods on the Industrial Benchmark & MuJoCo

Since the sanity checks show that the LION policies generally seem to behave the way one would expect, we now evaluate them on the industrial benchmark and MuJoCo datasets introduced in Chapter 3. During the evaluation of LION on these benchmarks, we are interested in examining the following issues:

- Do the LION policies outperform or at least match prior state of the art algorithms on the datasets somewhere in the λ range?
- Is it possible for practitioners to find these well performing λ values easily? That is, are the performance curves smooth and exhibit only a single global optimum or are they as erratic as the discrete baseline with many ups and downs and corresponding local mini and maxima?
- Is it possible for practitioners to tune λ so that the policy only exhibits behavior that is desired, in the sense that it is for their taste not too far away from the familiar solution? Ideally, the distance to the generating policy as a function of λ should be monotonously increasing.

We train the policies as outlined in Section 6.4.1 and evaluate them on the real environments in steps of 0.05 for λ from 0 to 1 and plot mean and standard deviation over five seeds in Figs. 6.5 - 6.7. To compare the results with prior state of the art offline RL algorithms, the figures include the respective performances as dashed horizontal lines: We compare LION to the model-free offline algorithms BCQ, BEAR, BRAC, TD3+BC, and CQL, to the hybrid approaches MOPO and MOReL, as well as to the purely model-based approaches so far presented in this thesis: MOOSE and WSBC. Since the considered prior works do not train adaptive policies, their behavior and thus performance remains the same everywhere on the spectrum. In principle, it is of course possible to train different discrete policies with the baselines for many of their respective proximity hyperparameters, however as we have shown in Section 6.3, this methodology has significant downsides compared to the LION approach (computational effort would be one of them: Training each baseline for each dataset for multiple seeds and for many hyperparameter values would be infeasible here). Further it is not possible to map the baselines' respective proximity hyperparameters to the correct λ value, since they all feature different regularization approaches. Thus, we stick to the baselines being portrayed as constant in Figs. 6.5 - 6.7. We further plot the distance of the evaluated target policy to the generating policy.

Examining the shown performance curves we can see that LION policies behave very much as one would expect them to: Starting with the 0% exploration datasets on the

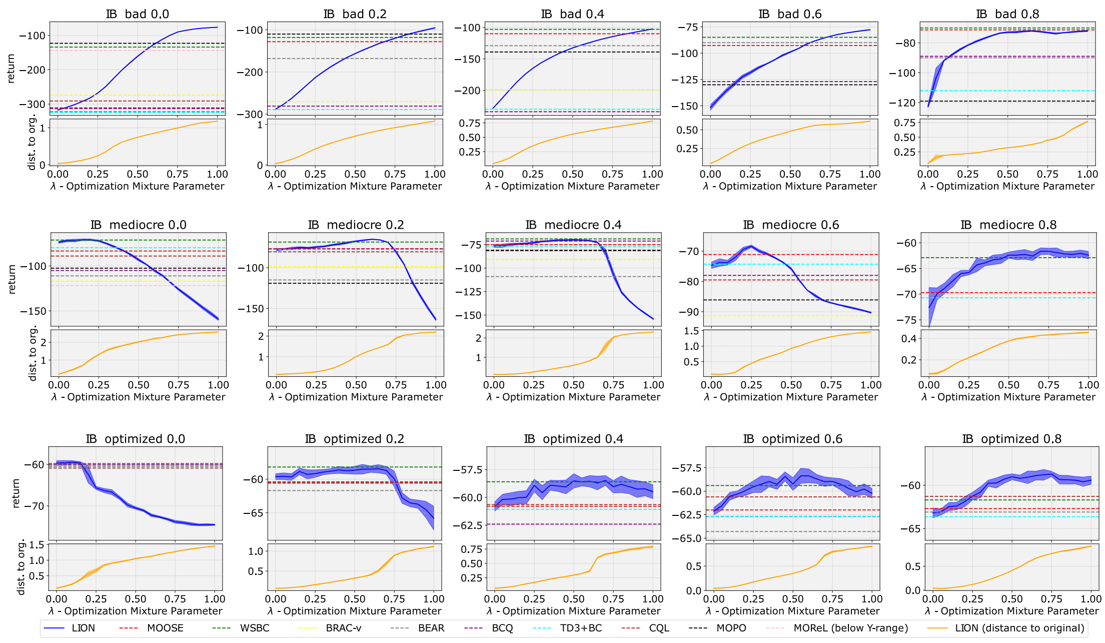


Figure 6.5: LION evaluation performance (top part of each graph) and distance to the generating policy (lower part of each graph) over the course of λ for each of the fifteen industrial benchmark datasets with the generating baselines bad, mediocre, and optimized. See Fig. 6.6 for the corresponding visualization on the 100% exploration dataset. The offline RL baselines' evaluation performances are represented as dashed lines and appear constant over the course of λ since (a) the corresponding policies are not adaptive, and (b) it is not possible to map their respective proximity hyperparameter to a single value of λ.

industrial benchmark, the policy performs only well close to $\lambda = 0$ in the mediocre and optimized settings, which makes sense since the transition models have not learned much about any other part of the state-action space, causing the policy return to be estimated inaccurately when more freedom in the form of higher λ values is granted, ultimately leading to steeply decreasing performances on the real environment. On the dataset with the bad baseline, the policy still improves with increased freedom since almost anything is better than copying the generating policy. Once more exploration is added to the datasets, the λ range in which LION performs well extends further and further towards the higher λ values, until in the 80% exploration datasets, the best performance is found close to $\lambda = 1$ since the transition models in these cases have much better knowledge about a larger part of the environment.
In thirteen of the sixteen industrial benchmark datasets, there exist parts of the λ spectrum where the evaluation return of LION is higher than that of any baseline, and on the remaining three, LION can still achieve close to on par performance. Crucially, the performance curves are very smooth and rise mostly monotonously towards their maximum value, making it as easy as possible for a potential user to navigate the λ landscape. Furthermore, the distance to the generating policy curves are all monotonously increas-

ing, making also the selection in terms of proximity feasible for a user wishing to tune the policy behavior towards the edge of what they are willing to endure in terms of unfamiliarity of the new solution.

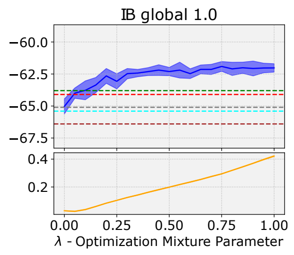


Figure 6.6: LION results as presented in Fig. 6.5 for the 100% exploration dataset.

Looking at the MuJoCo performances in Fig. 6.7, the smoothness of the curves becomes a bit of an issue: Especially in the Walker task, the LION policy clearly exhibits a local maximum after which the performance drastically declines. Between global and local maximum, the policy degrades by roughly 25%. The reasons for this may be plentiful: We can see from the transition model evaluation errors that the Walker environment is harder to model from a dynamics perspective. Also, the environment features a higher dimensional action space compared to the other environments, yet the dataset provides the same amount of interactions for learning. We hypothesize that due to the less accurate model in combination with the more complex action space, which makes it harder to interpolate between different λ values for the LION policy with limited capacity, the LION policy in this environment was not able to produce as smooth of a performance curve as the others. In the Swimmer environment, the local maxima are likely a product of randomness, since for each λ, five random seeds are used to evaluate the policy in the environment.

Putting smoothness aside, we find that the LION policies still achieve state of the art performance in parts of the λ spectrum or achieve on par performance. Also, the familiarity curves are monotonously rising, making it still likely that a practitioner could find a well performing policy that adheres to their maximum tolerated distance to the generating policy.

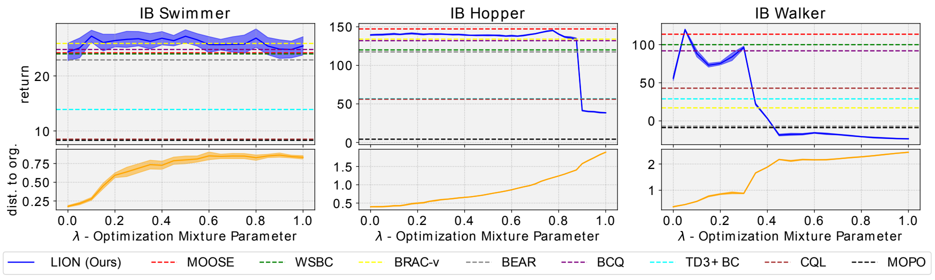


Figure 6.7: LION results as presented in Fig. 6.5 for the MuJoCo datasets. In this case, the performance curves are less smooth (most notably for the Walker benchmark), likely due to the higher dimensional and more complex action space. Nevertheless, The LION policies achieve state of the art or close to on par performance somewhere in the λ range and the distance to generating policy curves are still monotonously rising. A practitioner would still likely be able to pick a policy that is both well performing and suitable in terms of familiarity.

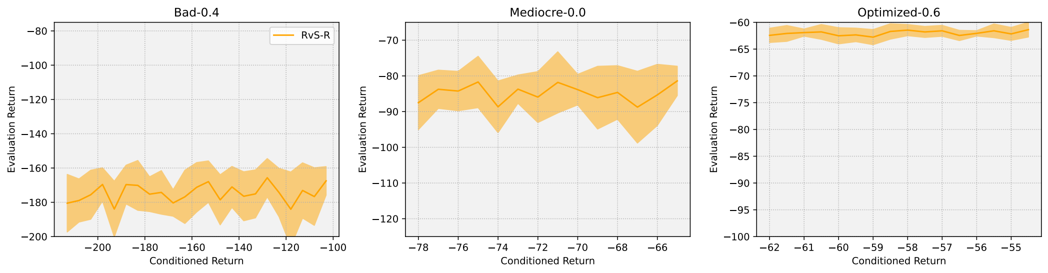


Figure 6.8: Mean evaluation performances together with their standard deviation of the RvS algorithm over a range of conditional return-to-go values for the same datasets as in Fig. 6.1.

6.6.1 Finding Good Trade-offs with LION

In order to find well performing policies, we have already previously proposed to start with $\lambda = 0$ and then slowly move towards higher values of λ. In this section, we formalize and evaluate this strategy for illustrative purposes - however we emphasize that it is not the goal of LION to find an automated strategy to find the "correct" λ value. Instead, providing control and utility ot the user remains a goal in itself. The proposed strategy is merely an example of how a user might choose to employ LION in reality. In order to compare how this strategy performs, we choose two baselines:

1. The discrete baseline from Section 6.3: If instead of LION, a discrete "adaptive" solution were built based on a prior algorithm such as MOPO, how would the hyperparameter search by the user compare?

2. In another line of work, researchers have come up with the idea of return conditioned policies. Algorithms such as Reinforcement Learning via Supervised Learning (RvS), or Decision Transformer (DT) [57, 150] condition on the return to go during trajectories and employ neither value function nor transition model. Instead, they model the policy for different targeted values of performance. We choose RvS as one of the latest algorithms of this family for our comparison.

Each of the three methods has a hyperparameter that influences proximity to the generating policy and expected performance: For LION it is λ, the mixture parameter between penalty and return estimate, for MOPO it is also called λ, just that here it means the scale of the uncertainty based penalty and that we only have a discrete number of choices, and for RvS, it is $\hat{R}$, the conditioned return-to-go which the policy should aim to achieve. In the last case, the connection to desired proximity is only implicit - one would however expect that if the return value of the generating policy is chosen, that it is sufficiently well recovered and thus the resulting distance to the generating policy would be low. See a visualization of the RvS performances on the same datasets as the discrete MOPO basline in Fig. 6.8.

The proposed strategy is to:

1. start with very constrained policies: $\lambda = 0$ for LION, $\lambda = 2.5$ for MOPO, and $\hat{R} = \text{mean}(R^{\text{data}})$ for RvS. Record the initial performance as P^{best}.

2. increase (decrease) the trade-off parameter in small steps: 0.05 for LION, 0.1 for MOPO, and 5/1/0.5 (due to the different return ranges) for RvS, since the hyperparameter ranges of MOPO and RvS are very different. Observe new performance P^{new}

3. If the observed performance P^{new} exceeds P^{best}: Set $P^{\text{best}} = P^{\text{new}}$ and go back to step 2 to grant further optimization freedom. If $P^{\text{new}} < P^{\text{best}}$: Immediately stop and go back to the last hyperparameter which generated P^{best}.

When it is applied to the datasets bad-0.4, mediocre-0.0, and optimized-0.6 for which we have also calculated the discrete baseline, the final hyperparameter values and corresponding evaluation returns of the three methods can be seen in Table 6.1. Due to the more erratic behavior of the discrete MOPO and RvS baselines compared to LION, the strategy is unable to identify the well performing hyperparameters, and consequently LION performs best in the three scenarios with the given strategy. For example in the bad-0.4 and mediocre-0.0 datasets, multiple hyperparameter values exist for MOPO that perform better than any hyperparameter in LION, however due to the zigzag of the performances, the strategy cannot find them. While users don't necessarily employ this strategy, we find it likely that similar ways of finding a well performing hyperparameter will be rather common among practitioners in practice, and we thus deem LION as likely more suitable for real world applications.

Dataset	LION Final λ	MOPO Final λ	RvS Final $\hat{R}$	LION Return	MOPO Return	RvS Return
Bad-0.4	1.0	2.4	-198	**-102.4**	-107.2	-169.6
Mediocre-0.0	0.2	2.4	-75	**-70.8**	-87.8	-81.7
Optimized-0.6	0.35	2.5	-60.5	**-58.9**	-72.4	-61.8

Table 6.1: If a user adopts the simple strategy of moving in small steps (0.05 for LION, 0.1 for MOPO , 5/1/0.5 for RvS) from conservative towards better solutions, immediately stopping when a performance drop is observed, LION finds much better solutions due to the consistent interpolation between trade-offs. Note that in MOPO, we start with large $\lambda = 2.5$ since there it controls the penalty, while we start with $\lambda = 0$ in LION, where it controls the return.

6.6.2 Additional Experiments

Hyperparameters Like any other newly proposed RL algorithm, our method introduces hyperparameters implicitly or explicitly. Due to the purely model-based nature of the LION algorithm, many of these can already be well tuned due to the supervised nature of the training of the first two components: The model of the dataset generating policy and the ensemble of transition dynamics models. Any hyperparameter related to these two, such as the model architecture, number of neurons per layer, optimizer, learning rate, etc can be tuned by holding out a validation set from the original data, and checking how well the trained model predicts actions or future states on the validation set.

Hyperparameters during offline policy training are notoriously hard to tune: Due to the nature of the offline setting, we can by definition not evaluate the performance of our hyperparameter choices. While offline policy evaluation approaches have been proposed to mitigate the problem, they need to be handled with extreme care, and overall do not seem to perform reliably yet. Especially the arguably most important hyperparameter, the desired proximity of the target policy to the known generating policy, cannot really be tuned with them since it involves knowing when the return evaluation method itself cannot be trusted any more. This is one reason why we propose to use LION to find well performing proximity hyperparameters instead of using OPE methods in practical applications.

On top of the proximity hyperparameter, LION exposes other hyperparameters during policy training, such as η, which controls how strictly the policy needs to stick to the dataset actions when evaluated for $\lambda = 0$, and α & β, the parameters of the Beta distribution from which λ is sampled during training. These parameters were not tuned, since we use the same parameters as [149] for α & β, by sampling $\lambda \sim \text{Beta}(0.1, 0.1)$. Similarly we choose $\eta = 0.1$ once and stick with it. However, these parameters of course have an effect on the final policy, and one could argue that just choosing it according to some literature could be a lucky guess and there should be a more principled way of choosing them. While we of course cannot assess performance related impacts, these two hyperparameters aren't directly responsible for performance, which provides us with an option to tune them at least to a degree even in the offline setting: η controls how strictly π adheres to β at $\lambda = 0$, which is something we can measure without the real environment: We can simply check how well $\pi(\cdot, \lambda = 0)$ reproduces dataset actions on a held out validation set after training. Fig. 6.9(a) shows how this quantity changes over the course of different η values. It appears that as long as a nonzero value is chosen, it does not make much of a difference, which is why we remain 0.1 as a reasonable value. Similarly, the α, β parameters determine how much weight to put on the edges of the distribution, i.e. how often which λ values get seen in training. At least at the edge that aims to learn the generating policy (i.e. $\lambda = 0$), we can again check whether the endeavour has worked out fully offline by evaluating against the generating policy: Fig 6.9(b) thus shows a similar evaluation for the $\alpha = \beta$ parameters, showing that by putting less weight on the edges, the generating policy is less accurately learned at $\lambda = 0$. $\alpha = \beta = 0.1$ thus seems a sensible choice, since even lower values may result in numerical instabilities.

Sampling λ A little counter-intuitively, λ is not sampled from a uniform distribution, even though in theory we would like to put equal weight on learning every value in the spectrum from generating policy to entirely free optimization. Interestingly however, the edges of the spectrum appear harder to learn, since they are less accurately recovered

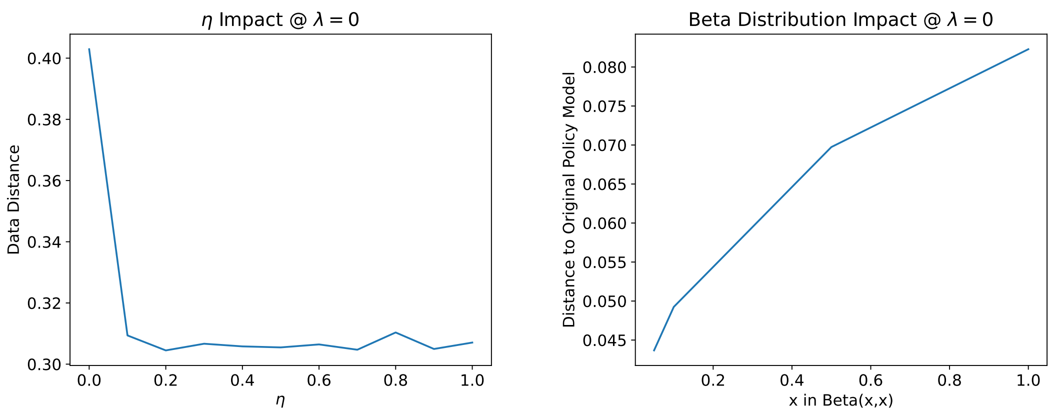


Figure 6.9: Impact of η, α, and β parameters on learning the generating policy accurately. This evaluation can be done entirely offline and requires no environment interaction, making it possible to make sensible hyperparameter choices.

during evaluation. We show performance curves over the λ spectrum for different parameters of the Beta distribution in Fig. 6.10 in order to highlight the phenomenon. We hypothesize that this is due to the averaging effect in neural networks: Since many values around the mean of the distribution are seen, the network is biased towards it. We observe this behavior also if the capacity is greatly increased and even in the simple 2D environment.

Effect of the transition ensemble We briefly analyze the effect the ensemble of transition models has on the policy. In the LION experiments, four ensemble members are used to ensure that the policy cannot simply exploit a single model with a potentially bad seed. Additionally, the minimum among the members is used for future state predictions, adding an additional source of conservatism which is usually beneficial in the offline setting. In Fig. 6.11, we show how the policy behavior over the range of λ changes when we use the mean instead of the minimum as aggregation and what happens if just a single model is used to perform the virtual rollouts instead of four.

The observable effect of the experiments is mostly as expected: While not much changes in the bad baseline (probably since any policy improves upon the bad policy), on the mediocre dataset with only a single model, the policy performance degrades sooner, i.e. for lower values of λ than in the standard case, suggesting that the policy is exploiting the model which is possible in the absence of corrections by other ensemble members. The effect is even more significant in the optimized dataset: While the policy performance only slightly degrades in performance towards the higher end of the λ spectrum in the LION performances in Fig. 6.5, there is a significant performance drop roughly at $\lambda = 0.5$ in the mean aggregation experiment, showing that the extra measure of conservatism by using the minimum aggregation helps in contexts where transition models are predicting outside of their region of confidence. Similarly to the mediocre setting, the performance

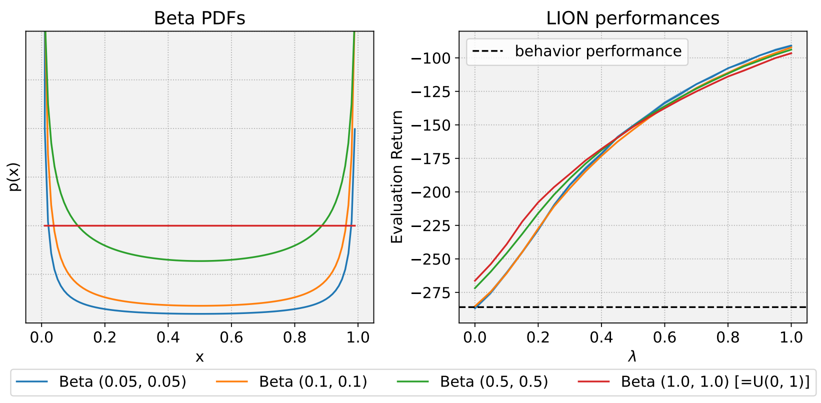


Figure 6.10: Beta distributions and resulting performances on the bad-0.2 dataset. Edge cases become more accurate with increased probability mass, likely due to neural networks being biased towards the mean of the λ distribution.

drop is then moved towards even lower λ values when only a single ensemble member is used ($\lambda \approx 0.3$).

Model-free return estimation While our purely model-based approach poses a plethora of advantages over model-free or hybrid approaches, such as increased stability of the learning process, better data efficiency, and the possibility to tune corresponding hyperparameters due to the supervised learning, there are reasons to extend the approach to the model-free domain: E.g. depending on the horizon of the environment, purely rollout based approaches might not be feasible / not obtain the best returns. This is due to error accumulation during the state prediction: When the future state predictions of the transition models are used in the next step again to predict the future state afterwards (and so forth), each time, small errors are made, which exponentially increase over the prediction horizon. In settings of long horizons, model-free approaches thus can have an edge over the rollout based approach since they only need to predict the value of a state-action combination instead of all state dimensions.

Even though the intertwined training of policy and value function that are both conditional on the λ parameter is obviously more complex and probably lacks stability, a model-free variant of LION should be attainable. To this end, we choose TD3+BC as an existing model-free offline RL algorithm and augment it by making both the Q-function and the policy conditional on λ, sampling λ as in LION from a Beta distribution during training, and altering the loss function so that λ influences the trade-off between Q-value and action distance penalty. We choose TD3+BC for this endeavour since it is a rather simple algorithm, making only minimal changes to move from the online TD3 to an offline algorithm with fewer models and implementational details than most other offline RL algorithms. The original Q-function and policy updates are defined as follows in TD3+BC:

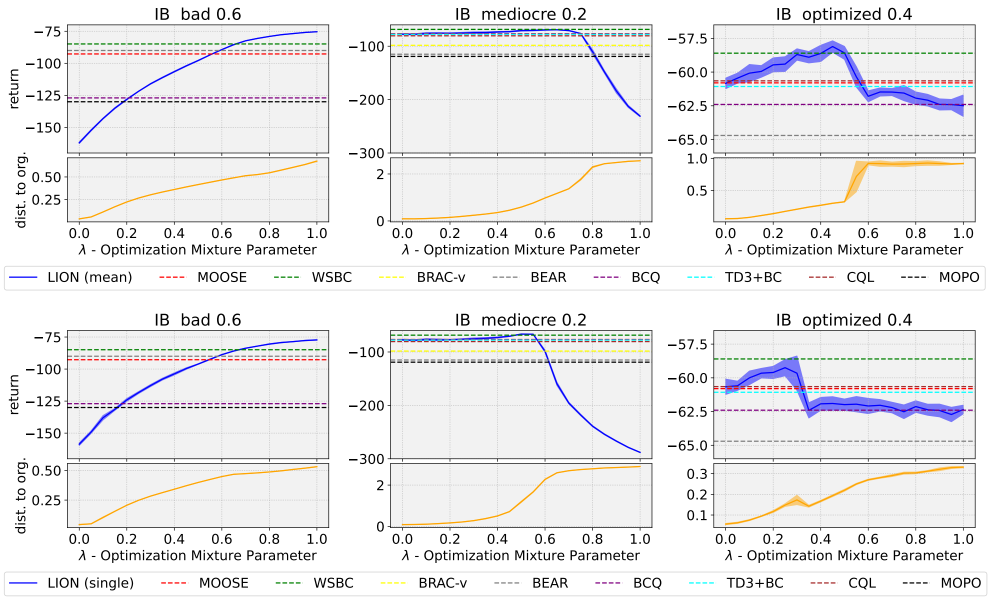


Figure 6.11: Results as presented in Fig. 6.5, but with mean aggregation over the ensemble members (top row), or with only a single instead of four models used in the reward estimation (bottom row).

$$Q = \arg\min_{Q} \mathbb{E}_{s,a,s'\sim\mathcal{D}} \left[[Q(s,a) - (r + \gamma Q^t(s', \pi(s))]^2) \right],$$

where Q^t denotes the slow moving target Q-function used for computing the Q-targets. And:

$$\pi = \arg\max_{\pi} \mathbb{E}_{s,a\sim\mathcal{D}} \left[\frac{\alpha}{\frac{1}{N}\sum_s |Q(s,\pi(s))|} Q(s,\pi(s)) - (\pi(s) - a)^2 \right]$$

In our extension, which we denote by λ-TD3+BC, we need to add λ as an input to the policy as well as the Q-function and use the parameter to balance the two optimization terms, similarly to the LION approach. The new objectives are then:

$$Q = \arg\min_{Q} \mathbb{E}_{s,a,s'\sim\mathcal{D}} \left[[Q(s,a,\lambda) - (r + \gamma Q^t(s', \pi(s,\lambda),\lambda)]^2) \right],$$

and:

$$\pi = \arg\max_{\pi} \mathbb{E}_{s,a\sim\mathcal{D}} \Bigg[\lambda \frac{\alpha}{\frac{1}{N}\sum_{s,a} |Q(s,\pi(s,\lambda),\lambda)|} Q(s,\pi(s,\lambda),\lambda) - (1-\lambda)(\pi(s,\lambda) - a)^2 \Bigg] \tag{6.9}$$

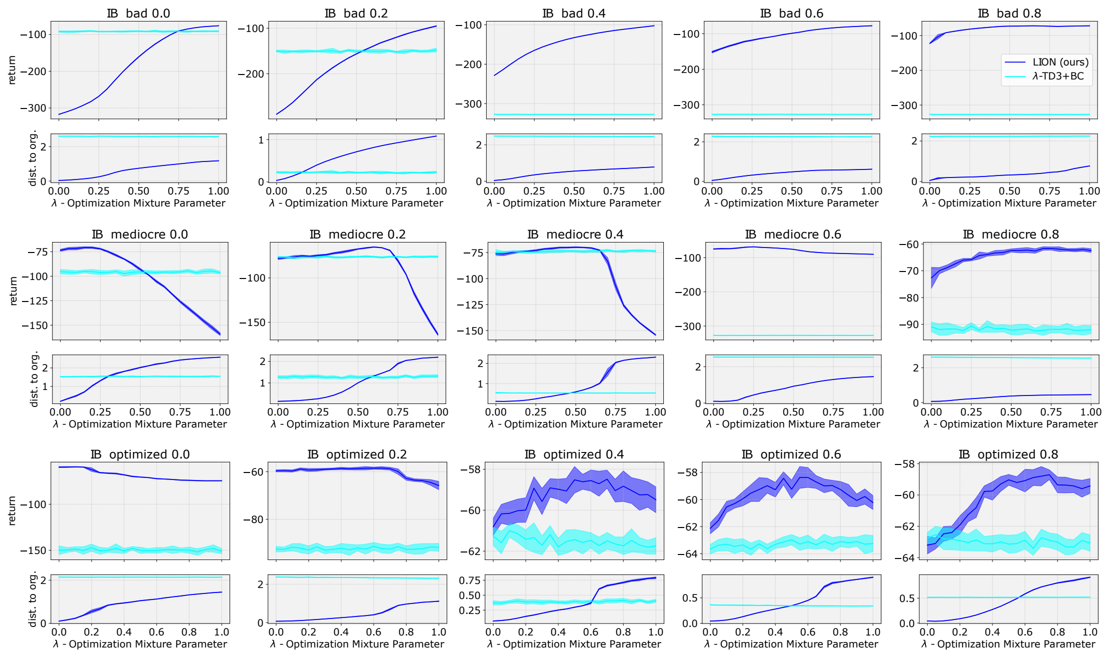


Figure 6.12: Model-free experiments incorporating the LION approach. The trained policies are unable to alter their behavior for different inputs of λ.

In Figs. 6.12 - 6.13, the results of the model-free variant on the industrial benchmark and MuJoCo benchmark datasets are presented together with the LION results as a comparison, similarly to the visualizations in Figs. 6.5 - 6.7. Interestingly, on the industrial benchmark datasets, the final policy is barely able to alter its behavior anywhere in the λ range - instead, it appears that the training process is unstable and the policy collapses to a single behavior no matter which λ is provided as input. While some of the resulting performances are quite good, the solution is missing the entire point of the approach, which is quite surprising. Even more intriguing, the same technique appears to work much more according to the expectations on the MuJoCo datasets: While the proximity curves on the bottom of Fig. 6.13 cannot be characterized as monotonous, there clearly is a tendency of policies closer to the generating policy (thus better performing) for lower λ values, and a tendency for larger deviations and consequently lower performance for higher λ values, at least in the Hopper and Walker2D tasks. The performances for some of the λ values are also much better than the non-adaptive basic TD3+BC algorithm (see Fig. 6.7), suggesting that the standard parameters are likely not the best ones for these datasets, and that users might be able to find better ones with our augmented version. The question why λ-TD3+BC does not expose any significant behavior changes on the industrial benchmark datasets yet remains - we hypothesize that the λ conditioned value function based training might be too unstable in the presence of such a highly stochastic benchmark, but ultimately it remains unclear and we leave investigations to future work.

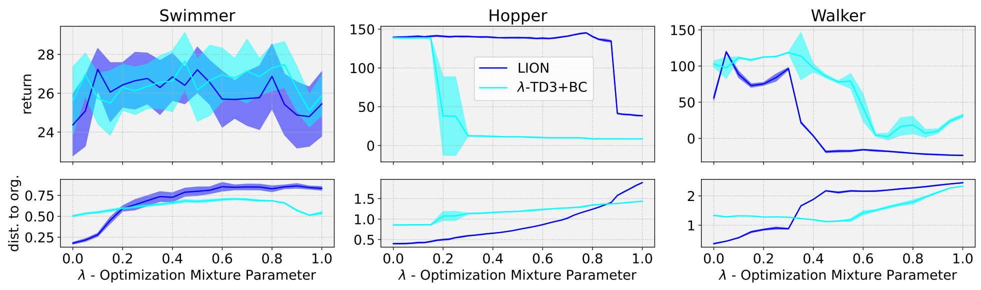


Figure 6.13: Model-free experiments incorporating the LION approach. Interestingly, the λ-TD3+BC approach appears to work relatively well in the MuJoCo tasks even though it fails to display significant behavior changes over the course of λ in the industrial benchmark datasets.

6.6.3 Experimental Details

In LION, we use the same transition models as in WSBC, i.e. for the industrial benchmark datasets, we used recurrent neural networks with a hidden dimensionality of 30 for each of the two reward components. The batchsize was 1024, the learning rate 1.25×10^{-5} and the employed optimizer was Adam. We trained for a maximum of 3000 epochs, while stopping early if the performance has not improved on a 10% held out validation set for 128 epochs. We employed an exponentially decaying learning rate by multiplying the current rate by 0.95 in every epoch. As outlined in [46], the output of the fatigue model was fed back in as new input during prediction, while this was not the case for the consumption model. The number of history steps G was 30 and the number of future steps F was 50. For the MuJoCo tasks, we only have a single model for both state and reward predictions with 100 neurons in the hidden RNN layer, and we decrease the number of history steps G to 3, while F remains at 50. We use a lower amount of history steps because the MuJoCo environments have deterministic transitions and are fully observable, however we still observe better evaluation performance if a low number of history steps is considered instead of just using the current state.

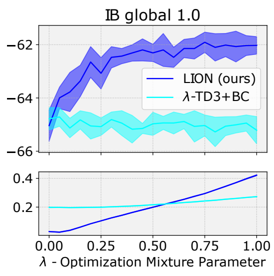


Figure 6.14: Model-free experiment on the $\varepsilon = 100\%$ IB dataset.

For the simple 2D environment, we trained simple MLP dynamics models with a batchsize of 512, 24 neurons in the hidden layer, a learning rate of 0.01, a patience of 10, and a decay factor of 0.99 for up to 300 epochs. While the models for the industrial benchmark and MuJoCo predict the state delta, the 2D environment models predict the next state directly. For the generating policy model, we employ again a simple MLP with a single hidden layer of 20 neurons for the industrial benchmark datasets and the 2D environment, and a two-layer size 128 MLP for the MuJoCo tasks. Here we employ no weight decay and the learning rate is set to 3×10^{-4} for 50 epochs. The target policy needs to have a higher capacity since it needs to be able to represent many different behaviors based on the conditioned λ value, so the target policy is a two layer MLP with 256 neurons in each layer. The final policy is trained for 1000 policy updates, in each of which we sample 200 starting states, build up the hidden state of the dynamics model for G history steps and then perform virtual rollouts for 100 steps with a γ of 0.97 for the industrial benchmark and 0.99 for the MuJoCo tasks and the 2D environment. As previously mentioned, η is always set to 0.1 as well as the parameters of the Beta distribution α & β. For the model-free experiments we use the standard parameters of TD3+BC and the same sampling parameters for λ as in LION. We use ReLU nonlinearities everywhere except for the hidden layer of the RNNs, which employ a tanh function.

6.7 Discussion

In this Chapter, we introduced the notion of proximity conditioned policies in order to provide utility value to the operator of the system to be controlled and in order to find well performing proximity hyperparameters. We develop the LION algorithm which can be seen as a natural extension of MOOSE and WSBC with the ability to condition the policy on a desired parameter of proximity to the generating policy, by exposing the target policy to various different loss trade-offs between proximity and return. The result is a broadly applicable method, which instead of optimizing away precious domain experts, compliments them to form a human-AI team which is enabled to find better performing policies on most of the considered datasets than would be possible with prior offline RL algorithms. We find that the solution also facilitates trust due to the option to always reduce the distance to the known generating policy for the expert, which should decrease deployment risks & costs, and consequently lead to more deployments of RL policies in real world systems.

Additionally, we show that the solution cannot be attained by simply training a prior offline algorithm for many different discrete hyperparameter choices due to inconsistent behavior as well as computational overhead. We provide an example strategy that could be employed by a user in practice and evaluate how well it serves to find good hyperparameters in practice, where we find that due to LION's smooth and consistent interpolation of behaviors over the λ spectrum, the operator has the best chances compared to the baselines. Moreover, we develop and evaluate a model-free variant of the proximity conditioned policy training, for which we augment TD3+BC with the same proximity-return trade-off balancing as in LION: While the approach appears to be technically valid and produces promising results on the MuJoCo datasets, it fails to exhibit significantly diverse behavior when evaluated on the industrial benchmark datasets.

7 Conclusion & Outlook

7.1 Conclusion

In this thesis, we considered the problem of learning reinforcement learning policies purely from static datasets, where the algorithm has no control over the collection process which generated the data, also known as offline reinforcement learning. We examined how suitable datasets can be selected for offline RL when many different datasets from different systems are available, as well as how to derive policies that on top of being offline, enable users to still influence and correct desired behavior after deployment. To this end, three novel model-based offline RL algorithms, as well as one method to assess offline data quality, were proposed.

The need for offline RL approaches naturally arises in industrial use cases that often involve physical systems: Due to safety concerns as well as large opportunity costs, online learning is almost always impractical. While model-free as well as hybrid offline approaches have been proposed before, the algorithms presented in this thesis are all purely model-based, i.e. they do not involve training a value function. We favour this setting since transition models have been observed to be more data efficient as well as leading to improved stability during policy training, both of which can be considered crucial elements in the offline setting. The common denominator of the three proposed algorithms is thus that they (i) train a transition ensemble from the given dataset, (ii) train another model which represents the data generating policy, (iii) perform policy training by estimating returns using virtual rollouts through the transition models and at the same time regularizing the policy to stay in some way close to the data generating policy.

The algorithms are evaluated on 19 challenging benchmark datasets: Three different generating policies of varying quality are combined with six levels of exploration on the industrial benchmark, in order to assess whether policies are able to cope with different degrees of knowledge available over the environment as well as different behaviors from which they can learn. The industrial benchmark features high dimensional as well as continuous state and action spaces, heteroscedastic as well as multimodal noise in the transition dynamics, is only partially observable, and exhibits delayed as well as contradicting reward components that require trading-off. Additionally, datasets from the three very well known MuJoCo domains Swimmer, Hopper, and Walker2D are used to examine how the algorithms cope with more complex transition dynamics.

The first proposed method examines the suitability of datasets for offline reinforcement learning by means of two simple indicators: The expected return improvement (ERI) and the estimated action stochasticity (EAS). The first basically measures how well the top performing trajectories in the dataset did compared with the average, so that it can be assessed how well an approach would do that simply copied more of the good behavior and would be able to discard most of the rest. The latter on the other hand aims to assess exploration, i.e. the amount of knowledge about alternative behavior that is contained in the dataset. The better an environment was explored, the more likely it is that return estimators are able to provide accurate predictions for policies that deviate from the baseline, making return improvements more likely. We find that the combination of ERI and EAS, called COI (combined offline indicator) is able to identify more promising datasets, albeit not perfectly, very well.

The first proposed offline RL algorithm to actually derive a policy from a dataset is called MOOSE (MOdel-based Offline policy Search with Ensembles), and aims to bring action-space based regularization penalties from the model-free world to the purely model-based setting in order to improve data efficiency and stability of the learning process. The performance is assessed in virtual rollouts through the transition models, and the target policy regularization towards the data generating one is performed by means of an additional variational autoencoder, which is trained to reconstruct actions based on a provided state-action pair from the dataset. Since it reconstructs actions from known state-action pairs better than unknown ones, it can be used to build a penalty term which is added to the policy loss. Experiments show that MOOSE almost never underperforms the data generating policy and provides better returns than prior algorithms on most datasets, thus making learning from static datasets practical.

The second proposed algorithm examines whether regularization could actually be performed in an entirely different way: Instead of regularizing the actions towards those the data generating policy would have taken, WSBC (Weight Space Behavior Constraining) constrains the target policy's parameters directly in the neural weight space to remain close to those of the generating one. To this end, WSBC trains a model of the generating policy of the same architecture as the target policy, and searches the policy space by means of gradient-free optimization using PSO (particle swarm optimization). WSBC appears to perform better in the industrial benchmark domain, where it outperforms the results of MOOSE, than in the MuJoCo domain, where it takes a second place among the considered algorithms. This is likely related to the lower dimensional policies that are sufficient for the industrial benchmark, but not in the three MuJoCo tasks.

Finally, the LION algorithm together with the idea of proximity conditioned policies is proposed in Chapter 6. Instead of only choosing a single hyperparameter value for the weight of the policy regularization towards the generating policy, LION learns policies that condition on it and leave the concrete choice of value open until runtime. This enables effective hyperparameter tuning of the most important hyperparameter in offline reinforcement learning, as well as provides expert users with a utility, since they remain in control of the system and can immediately correct undesired behavior by constraining the policy more heavily. We show that the best hyperparameter is a different one for each considered dataset, highlighting the subobtimality of prior approaches that use a single works-for-it-all solution. Using LION, we show that due to the smooth interpolation it is possible to find close to optimal parameters for most datasets using simple strategies and thus improve over the results in MOOSE and WSBC.

In summary, we can see that purely model-based offline RL methods, despite their niche existence in literature, perform better and train policies more stably than model-free or hybrid methods due to their improved data efficiency over model-free approaches and their simplicity and stability compared to hybrid approaches. Further, we show that for each conceivable dataset, different hyperparameters in terms of proximity to the generating policy are required to train well performing policies, contrary to popular belief with no one-works-for-it-all solution in sight. This makes policies that remain in some way adaptive in this hyperparameter a necessity, and we expect future works to adapt policy behavior using a combination of observed data quality dimensions and feedback from the first few evaluations.

7.2 Future Work

One of the most straightforward extensions of the work presented in this thesis is to transfer the idea of proximity conditioned policies to the model-free algorithm domain, as outlined in Section 6.6.2. While we were unable to make the λ-TD3+BC approach work properly on all of the considered benchmarks, we believe that it should generally be possible to implement proximity conditioned policies in the model-free setting as well, even though some additional measures in order to combat training instabilities of the two intertwined models need to be developed. In environments in which horizons are even longer than those considered in this thesis, this extension should lead to improved performance over the purely model-based variant, since in very long horizons, model errors compound and thus impair the policy performance through inaccurate return estimations.

Another interesting line of work would consist of using the proximity conditioned policies to automatically finding well performing or otherwise desirable parameters. While we very briefly touched on this subject in Section 6.6.2, we explicitly leave hyperparameter choice and tuning strategy to the expert user, since we assume they exist, are to be provided with a utility, and possibly have an insider edge over any automated system. While this is true in many industrial contexts, other systems may benefit from a more automated solution since such an expert does not exist or has limited time for each individual system and would thus also welcome more automation. Also, in systems where opportunity cost and safety concerns are low, automation might just be less expensive than the expert's time. In this case, selection strategies could be developed that based on an online evaluation budget try to find the best proximity value with as few tries as possible. For this, many prior works on hyperparameter search exist, such as Bayesian

Optimization, Population Based Training, or Hyperband Optimization [151, 152, 153]. In order to derive interesting priors or heuristics for the best proximity value, the dataset quality measurement techniques from Chapter 4 could be adapted in order to provide an initial educated guess: E.g. if the EAS feature assesses low amounts of exploration contained in the data, it is likely that low values of λ will lead to the best performing policies.

While three different types of regularizations for offline RL have been proposed in this thesis, even though they all work well and produce useful policies, it remains unclear which of them can be considered best. Of course, the best regularization could also be another one proposed elsewhere, since every year, hundreds of papers related to offline reinforcement learning are currently proposed. Likely, there exists not even just a single best method, instead it could be a different one for any given dataset, similarly to the results in Chapter 6 which show that each dataset has a different optimal value of proximity to the generating policy. Since each regularization technique exhibits a different proximity-like hyperparameter that has a different value range and offers a different interpretation, it is quite hard to compare them. Additionally, as shown in Section 6.3, many algorithms are proposed with a single default value of their proximity parameter, however are not consistent or stable when evaluated over a broader range of values, making the problem worse. The proximity conditioned policies framework however could provide a unique opportunity to compare regularizations: When for each regularization, a regularization parameter conditional policy is trained, it becomes possible to compare the performance curves of the techniques and evaluate which finds the single best policy (peak), which is least hyperparameter sensitive and provides the best policy for most concrete hyperparameter values (area under curve), or which is best suitable for hyperparameter search (smoothness of the curve). The main condition for this procedure would be a way to align the ranges of different hyperparameters, which could be challenging since some methods exhibit unlimited ranges while others have limits. However, it is likely possible to produce limited versions in these cases by employing a similar convex combination as proposed in the LION algorithm.

Even though one of the benchmarks considered in this thesis exhibits high amounts of stochasticity, the corresponding models where simple RNNs, without explicit uncertainty quantification as would be available in Gaussian Processes or Bayesian Neural Networks, or even simple Gaussain output networks such as those employed in MOPO. Only slight adaptations to the multimodal and heteroscedastic noise are performed, by employing ensembles of models and biasing the predictions towards the minimum of the members. In future works, it could be evaluated which model class is actually best suitable to deal with the stochastic element. Compared with the other future research paths proposed in this section, this one should be comparatively simple, since training of the models and comparison of the resulting uncertainty estimates should be possible in a supervised fashion.

Bibliography

[1] Richard S Sutton and Andrew G Barto. *Introduction to reinforcement learning*, volume 135. MIT press Cambridge, 1998.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

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

[4] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *International conference on machine learning*, pages 387–395. PMLR, 2014.

[5] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897, 2015.

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

[7] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv preprint arXiv:1801.01290*, 2018.

[8] Scott Fujimoto, David Meger, and Doina Precup. Off-policy deep reinforcement learning without exploration. *arXiv preprint arXiv:1812.02900*, 2018.

[9] Sascha Lange, Thomas Gabel, and Martin Riedmiller. Batch reinforcement learning. In *Reinforcement Learning*, pages 45–73. Springer, 2012.

[10] Damien Ernst, Mevludin Glavic, Pierre Geurts, and Louis Wehenkel. Approximate value iteration in the reinforcement learning context. application to electrical power system control. *International Journal of Emerging Electric Power Systems*, 3(1), 2005.

[11] Stefan Depeweg, José Miguel Hernández-Lobato, Finale Doshi-Velez, and Steffen Udluft. Learning and policy search in stochastic dynamical systems with Bayesian neural networks. *arXiv preprint arXiv:1605.07127*, 2016.

[12] Daniel Hein, Alexander Hentschel, Thomas A Runkler, and Steffen Udluft. Reinforcement learning with particle swarm optimization policy (PSO-P) in continuous state and action spaces. *International Journal of Swarm Intelligence Research (IJSIR)*, 7(3):23–42, 2016.

[13] Daniel Hein, Steffen Udluft, and Thomas A Runkler. Interpretable policies for reinforcement learning by genetic programming. *Engineering Applications of Artificial Intelligence*, 76:158–169, 2018.

[14] Branka Mirchevska, Christian Pek, Moritz Werling, Matthias Althoff, and Joschka Boedecker. High-level decision making for safe and reasonable autonomous lane changing using reinforcement learning. In *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*, pages 2156–2162. IEEE, 2018.

[15] Marc Deisenroth and Carl E Rasmussen. PILCO: A model-based and data-efficient approach to policy search. In *28th International Conference on Machine Learning (ICML-11)*, pages 465–472, 2011.

[16] Thanard Kurutach, Ignasi Clavera, Yan Duan, Aviv Tamar, and Pieter Abbeel. Model-ensemble trust-region policy optimization. *arXiv preprint arXiv:1802.10592*, 2018.

[17] Anusha Nagabandi, Gregory Kahn, Ronald S Fearing, and Sergey Levine. Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 7559–7566. IEEE, 2018.

[18] Daniel Hein. *Interpretable Reinforcement Learning Policies by Evolutionary Computation*. PhD thesis, Technische Universität München, 2019.

[19] M Milani Fard and Joelle Pineau. Non-deterministic policies in Markovian decision processes. *Journal of Artificial Intelligence Research*, 40:1–24, 2011.

[20] Maria De-Arteaga, Artur Dubrawski, and Alexandra Chouldechova. Leveraging expert consistency to improve algorithmic decision support. *arXiv preprint arXiv:2101.09648*, 2021.

[21] Marc Peter Deisenroth and Dieter Fox. Multiple-target reinforcement learning with a single policy. 2011.

[22] Bastian Bischoff, Duy Nguyen-Tuong, Torsten Koller, Heiner Markert, and Alois Knoll. Learning throttle valve control using policy search. In *Joint European conference on machine learning and knowledge discovery in databases*, pages 49–64. Springer, 2013.

[23] Vitchyr Pong, Shixiang Gu, Murtaza Dalal, and Sergey Levine. Temporal difference models: Model-free deep RL for model-based control. *arXiv preprint arXiv:1802.09081*, 2018.

[24] Gabriel Dulac-Arnold, Daniel Mankowitz, and Todd Hester. Challenges of real-world reinforcement learning, 2019. URL: `https://arxiv.org/abs/1904.12901`, `doi:10.48550/ARXIV.1904.12901`.

[25] Tatsuya Matsushima, Hiroki Furuta, Yutaka Matsuo, Ofir Nachum, and Shixiang Gu. Deployment-efficient reinforcement learning via model-based offline optimization. *arXiv preprint arXiv:2006.03647*, 2020.

[26] Sergey Levine, Aviral Kumar, George Tucker, and Justin Fu. Offline reinforcement learning: Tutorial, review, and perspectives on open problems. *arXiv preprint arXiv:2005.01643*, 2020.

[27] Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. Programmatically interpretable reinforcement learning. In *International Conference on Machine Learning*, pages 5045–5054. PMLR, 2018.

[28] Nathalie Smuha. Ethics guidelines for trustworthy AI. *European Commission, Directorate-General for Communications Networks, Content and Technology Publications Office*, 2019.

[29] Ben Shneiderman. Human-centered artificial intelligence: three fresh ideas. *AIS Transactions on Human-Computer Interaction*, 12(3):109–124, 2020.

[30] Albrecht Schmidt, Fosca Giannotti, Wendy Mackay, Ben Shneiderman, and Kaisa Väänänen. Artificial intelligence for humankind: A panel on how to create truly interactive and human-centered AI for the benefit of individuals and society. In *IFIP Conference on Human-Computer Interaction*, pages 335–339. Springer, 2021.

[31] Matthias Althoff, Olaf Stursberg, and Martin Buss. Model-based probabilistic collision detection in autonomous driving. *IEEE Transactions on Intelligent Transportation Systems*, 10(2):299–310, 2009.

[32] Hanna Krasowski, Prithvi Akella, Aaron Ames, and Matthias Althoff. Verifiably safe reinforcement learning with probabilistic guarantees via temporal logic. *arXiv preprint arXiv:2212.06129*, 2022.

[33] Niklas Kochdumper, Hanna Krasowski, Xiao Wang, Stanley Bak, and Matthias Althoff. Provably safe reinforcement learning via action projection using reachability analysis and polynomial zonotopes. *arXiv preprint arXiv:2210.10691*, 2022.

[34] Hanna Krasowski, Yinqiang Zhang, and Matthias Althoff. Safe reinforcement learning for urban driving using invariably safe braking sets. In *2022 IEEE 25th International Conference on Intelligent Transportation Systems (ITSC)*, pages 2407–2414. IEEE, 2022.

[35] Phillip Swazinna, Steffen Udluft, and Thomas Runkler. Measuring data quality for dataset selection in offline reinforcement learning. In *2021 IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE, 2021.

[36] Phillip Swazinna, Steffen Udluft, and Thomas Runkler. Overcoming model bias for robust offline deep reinforcement learning. *Engineering Applications of Artificial Intelligence*, 104:104366, 2021.

[37] Phillip Swazinna, Steffen Udluft, Daniel Hein, and Thomas Runkler. Behavior constraining in weight space for offline reinforcement learning. *European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning*, 2021.

[38] Phillip Swazinna, Steffen Udluft, Daniel Hein, and Thomas Runkler. Comparing model-free and model-based algorithms for offline reinforcement learning. *IFAC Conference on Intelligent Control and Automation Sciences*, 2022.

[39] Phillip Swazinna, Steffen Udluft, and Thomas Runkler. Towards user-interactive offline reinforcement learning. *NeurIPS 3rd Offline RL Workshop: Offline RL as a "Launchpad"*, 2022.

[40] Phillip Swazinna, Steffen Udluft, and Thomas Runkler. User-interactive offline reinforcement learning. *Accepted at ICLR 2023 - 11th International Conference on Learning Representations*, 2023.

[41] Tianhe Yu, Aviral Kumar, Rafael Rafailov, Aravind Rajeswaran, Sergey Levine, and Chelsea Finn. Combo: Conservative offline model-based policy optimization. *arXiv preprint arXiv:2102.08363*, 2021.

[42] Rahul Kidambi, Aravind Rajeswaran, Praneeth Netrapalli, and Thorsten Joachims. MOReL: Model-based offline reinforcement learning. *arXiv preprint arXiv:2005.05951*, 2020.

[43] Tianhe Yu, Garrett Thomas, Lantao Yu, Stefano Ermon, James Y Zou, Sergey Levine, Chelsea Finn, and Tengyu Ma. MOPO: Model-based offline policy optimization. In *Advances in Neural Information Processing Systems*, volume 33, pages 14129–14142, 2020.

[44] Martin Riedmiller. Neural fitted Q iteration–first experiences with a data efficient neural reinforcement learning method. In *European Conference on Machine Learning*, pages 317–328. Springer, 2005.

[45] Martin Riedmiller, Thomas Gabel, Roland Hafner, and Sascha Lange. Reinforcement learning for robot soccer. *Autonomous Robots*, 27(1):55–73, 2009.

[46] Daniel Hein, Steffen Udluft, Michel Tokic, Alexander Hentschel, Thomas A Runkler, and Volkmar Sterzing. Batch reinforcement learning on the industrial benchmark: First experiences. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 4214–4221. IEEE, 2017.

[47] Markus Kaiser, Clemens Otte, Thomas Runkler, and Carl Henrik Ek. Interpretable dynamics models for data-efficient reinforcement learning. *arXiv preprint arXiv:1907.04902*, 2019.

[48] Anurag Ajay, Aviral Kumar, Pulkit Agrawal, Sergey Levine, and Ofir Nachum. Opal: Offline primitive discovery for accelerating offline reinforcement learning. *arXiv preprint arXiv:2010.13611*, 2020.

[49] Justin Fu, Aviral Kumar, Ofir Nachum, George Tucker, and Sergey Levine. Datasets for data-driven reinforcement learning. *arXiv preprint arXiv:2004.07219*, 2020.

[50] Romain Laroche, Paul Trichelair, and Rémi Tachet des Combes. Safe policy improvement with baseline bootstrapping. *arXiv preprint arXiv:1712.06924*, 2017.

[51] Seyed Kamyar Seyed Ghasemipour, Dale Schuurmans, and Shixiang Shane Gu. Emaq: Expected-max Q-learning operator for simple yet effective offline and online RL. In *International Conference on Machine Learning*, pages 3682–3691. PMLR, 2021.

[52] Aviral Kumar, Justin Fu, Matthew Soh, George Tucker, and Sergey Levine. Stabilizing off-policy Q-learning via bootstrapping error reduction. In *Advances in Neural Information Processing Systems*, pages 11761–11771, 2019.

[53] Arthur Gretton, Karsten M Borgwardt, Malte J Rasch, Bernhard Schölkopf, and Alexander Smola. A kernel two-sample test. *Journal of Machine Learning Research*, 13(Mar):723–773, 2012.

[54] Yifan Wu, George Tucker, and Ofir Nachum. Behavior regularized offline reinforcement learning. *arXiv preprint arXiv:1911.11361*, 2019.

[55] Noah Y Siegel, Jost Tobias Springenberg, Felix Berkenkamp, Abbas Abdolmaleki, Michael Neunert, Thomas Lampe, Roland Hafner, and Martin Riedmiller. Keep doing what worked: Behavioral modelling priors for offline reinforcement learning. *arXiv preprint arXiv:2002.08396*, 2020.

[56] Xue Bin Peng, Aviral Kumar, Grace Zhang, and Sergey Levine. Advantage-weighted regression: Simple and scalable off-policy reinforcement learning. *arXiv preprint arXiv:1910.00177*, 2019.

[57] Scott Emmons, Benjamin Eysenbach, Ilya Kostrikov, and Sergey Levine. RvS: What is essential for offline RL via supervised learning? *arXiv preprint arXiv:2112.10751*, 2021.

[58] Xinyue Chen, Zijian Zhou, Zheng Wang, Che Wang, Yanqiu Wu, and Keith Ross. BAIL: Best-action imitation learning for batch deep reinforcement learning. *arXiv preprint arXiv:1910.12179*, 2019.

[59] Ziyu Wang, Alexander Novikov, Konrad Zolna, Jost Tobias Springenberg, Scott Reed, Bobak Shahriari, Noah Siegel, Josh Merel, Caglar Gulcehre, Nicolas Heess, et al. Critic regularized regression. *arXiv preprint arXiv:2006.15134*, 2020.

[60] Minghuan Liu, Hanye Zhao, Zhengyu Yang, Jian Shen, Weinan Zhang, Li Zhao, and Tie-Yan Liu. Curriculum offline imitating learning. In *Thirty-Fifth Conference on Neural Information Processing Systems*, 2021.

[61] Ofir Nachum, Yinlam Chow, Bo Dai, and Lihong Li. Dualdice: Behavior-agnostic estimation of discounted stationary distribution corrections. *arXiv preprint arXiv:1906.04733*, 2019.

[62] Ofir Nachum, Bo Dai, Ilya Kostrikov, Yinlam Chow, Lihong Li, and Dale Schuurmans. Algaedice: Policy gradient from arbitrary experience. *arXiv preprint arXiv:1912.02074*, 2019.

[63] Ruiyi Zhang, Bo Dai, Lihong Li, and Dale Schuurmans. Gendice: Generalized offline estimation of stationary values. *arXiv preprint arXiv:2002.09072*, 2020.

[64] Shangtong Zhang, Bo Liu, and Shimon Whiteson. Gradientdice: Rethinking generalized offline estimation of stationary values. In *International Conference on Machine Learning*, pages 11194–11203. PMLR, 2020.

[65] Brahma Pavse, Ishan Durugkar, Josiah Hanna, and Peter Stone. Reducing sampling error in batch temporal difference learning. In *International Conference on Machine Learning*, pages 7543–7552. PMLR, 2020.

[66] Rishabh Agarwal, Dale Schuurmans, and Mohammad Norouzi. An optimistic perspective on offline reinforcement learning. In *International Conference on Machine Learning*, pages 104–114. PMLR, 2020.

[67] Jordi Smit, Canmanie T Ponnambalam, Matthijs TJ Spaan, and Frans A Oliehoek. PEBL: Pessimistic ensembles for offline deep reinforcement learning. In *Robust and Reliable Autonomy in the Wild Workshop at the 30th International Joint Conference of Artificial Intelligence*, 2021.

[68] Núria Armengol Urpí, Sebastian Curi, and Andreas Krause. Risk-averse offline reinforcement learning. *arXiv preprint arXiv:2102.05371*, 2021.

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

[70] Yao Liu, Adith Swaminathan, Alekh Agarwal, and Emma Brunskill. Provably good batch reinforcement learning without great exploration. *arXiv preprint arXiv:2007.08202*, 2020.

[71] Aviral Kumar, Aurick Zhou, George Tucker, and Sergey Levine. Conservative Q-learning for offline reinforcement learning. *arXiv preprint arXiv:2006.04779*, 2020.

[72] Aayam Shrestha, Stefan Lee, Prasad Tadepalli, and Alan Fern. Deepaveragers: Offline reinforcement learning by solving derived non-parametric MDPs. *arXiv preprint arXiv:2010.08891*, 2020.

[73] Scott Fujimoto and Shixiang Shane Gu. A minimalist approach to offline reinforcement learning. *arXiv preprint arXiv:2106.06860*, 2021.

[74] Anton Maximilian Schaefer, Steffen Udluft, and Hans-Georg Zimmermann. A recurrent control neural network for data efficient reinforcement learning. In *2007 IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning*, pages 151–157, 2007. `doi:10.1109/ADPRL.2007.368182`.

[75] Sham M Kakade. A natural policy gradient. *Advances in neural information processing systems*, 14, 2001.

[76] James Kennedy and Russell Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95-international conference on neural networks*, volume 4, pages 1942–1948. IEEE, 1995.

[77] Andries Engelbrecht. Particle swarm optimization: Velocity initialization. In *2012 IEEE congress on evolutionary computation*, pages 1–8. IEEE, 2012.

[78] Andries P. Engelbrecht. *Fundamentals of computational swarm intelligence.* Wiley, 2005.

[79] Russ Eberhart, Pat Simpson, and Roy Dobbins. *Computational intelligence PC tools*. Academic Press Professional, Inc., 1996.

[80] Yuhui Shi and Russell Eberhart. A modified particle swarm optimizer. In *1998 IEEE international conference on evolutionary computation proceedings. IEEE world congress on computational intelligence (Cat. No. 98TH8360)*, pages 69–73. IEEE, 1998.

[81] Russ C Eberhart and Yuhui Shi. Comparing inertia weights and constriction factors in particle swarm optimization. In *Proceedings of the 2000 congress on evolutionary computation. CEC00 (Cat. No. 00TH8512)*, volume 1, pages 84–88. IEEE, 2000.

[82] Diederik P Kingma and Max Welling. Auto-encoding variational Bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[83] Stephen Odaibo. Tutorial: Deriving the standard variational autoencoder (vae) loss function. *arXiv preprint arXiv:1907.08956*, 2019.

[84] Daniel Hein, Stefan Depeweg, Michel Tokic, Steffen Udluft, Alexander Hentschel, Thomas A Runkler, and Volkmar Sterzing. A benchmark environment motivated by industrial control problems. In *2017 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–8. IEEE, 2017.

[85] Emanuel Todorov, Tom Erez, and Yuval Tassa. MuJoCo: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012.

[86] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[87] Rémi Coulom. *Reinforcement learning using neural networks, with applications to motor control.* PhD thesis, Institut National Polytechnique de Grenoble-INPG, 2002.

[88] Tom Erez, Yuval Tassa, and Emanuel Todorov. Infinite-horizon model predictive control for periodic tasks with contacts. *Robotics: Science and systems VII*, 73, 2012.

[89] Markus Kaiser, Clemens Otte, Thomas A Runkler, and Carl Henrik Ek. Bayesian decomposition of multi-modal dynamical systems for reinforcement learning. *Neurocomputing*, 2020.

[90] Markus Kaiser, Clemens Otte, Thomas Runkler, and Carl Henrik Ek. Bayesian alignments of warped multi-output Gaussian processes. *arXiv preprint arXiv:1710.02766*, 2018.

[91] Stefan Depeweg, José Miguel Hernández-Lobato, Finale Doshi-Velez, and Steffen Udluft. Decomposition of uncertainty in Bayesian deep learning for efficient and risk-sensitive learning. *arXiv preprint arXiv:1710.07283*, 2017.

[92] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[93] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257, 1991.

[94] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[95] Kunihiko Fukushima. Cognitron: A self-organizing multilayered neural network. *Biological cybernetics*, 20(3):121–136, 1975.

[96] Andrew L Maas, Awni Y Hannun, Andrew Y Ng, et al. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, page 3. Atlanta, Georgia, USA, 2013.

[97] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015.

[98] Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6(02):107–116, 1998.

[99] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. LSTM neural networks for language modeling. In *Thirteenth annual conference of the international speech communication association*, 2012.

[100] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[101] Tim Salimans and Durk P Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. In *Advances in neural information processing systems*, pages 901–909, 2016.

[102] Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[103] Anton Maximilian Schäfer. *Reinforcement learning with recurrent neural networks.* PhD thesis, Osnabrück, Univ., Diss., 2008, 2008.

[104] Daniel Schneegaß. *Steigerung der Informationseffizienz im Reinforcement-Learning.* PhD thesis, Lübeck, Univ., Diss., 2008, 2008.

[105] Anton Maximilian Schaefer, Daniel Schneegass, Volkmar Sterzing, and Steffen Udluft. A neural reinforcement learning approach to gas turbine control. In *2007 International Joint Conference on Neural Networks*, pages 1691–1696. IEEE, 2007.

[106] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[107] Carlo Batini, Cinzia Cappiello, Chiara Francalanci, and Andrea Maurino. Methodologies for data quality assessment and improvement. *ACM Computing Surveys (CSUR)*, 41(3):1–52, 2009.

[108] Venkat Gudivada, Amy Apon, and Junhua Ding. Data quality considerations for big data and machine learning: Going beyond data cleaning and transformations. *International Journal on Advances in Software*, 10(1):1–20, 2017.

[109] Li Li, Huazhu Fu, Bo Han, Cheng-Zhong Xu, and Ling Shao. Federated noisy client learning. *arXiv preprint arXiv:2106.13239*, 2021.

[110] Angelica Poli, Gloria Cosoli, Lorenzo Scalise, and Susanna Spinsante. Impact of wearable measurement properties and data quality on adls classification accuracy. *IEEE Sensors Journal*, 2020.

[111] Jorge Villamil, Jorge Victorino, and Francisco Gómez. The effect of mobile camera selection on the capacity to predict water turbidity. *Water Science and Technology*, 2021.

[112] Vito F. Chiarella, Thiago Rateke, Karla A. Justen, Antonio C. Sobieranski, Sylvio L. Mantelli, Eros Comunello, and Aldo von Wangenheim. Comparison between low-cost passive and active vision for obstacle depth. *Revista de Ciência e Tecnologia (RCT)*, 6, 2020.

[113] Victor S Sheng, Foster Provost, and Panagiotis G Ipeirotis. Get another label? improving data quality and data mining using multiple, noisy labelers. In *14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 614–622, 2008.

[114] Eric Breck, Neoklis Polyzotis, Sudip Roy, Steven Whang, and Martin Zinkevich. Data validation for machine learning. In *MLSys*, 2019.

[115] Sebastian Schelter, Stefan Grafberger, Philipp Schmidt, Tammo Rukat, Mario Kiessling, Andrey Taptunov, Felix Biessmann, and Dustin Lange. Deequ-data quality validation for machine learning pipelines. In *Machine Learning Systems Workshop at the Conference on Neural Information Processing Systems (NeurIPS)*, 2018.

[116] Till Doehmen, Mark Raasveldt, Hannes Mühleisen, and Sebastian Schelter. Duckdq: Data quality assertions for machine learning pipelines. In *Workshop on Challenges in Deploying and Monitoring ML Systems at the International Conference on Machine Learning (ICML), 2021*, 2021.

[117] Lisa Ehrlinger, Verena Haunschmid, Davide Palazzini, and Christian Lettner. A daql to monitor data quality in machine learning applications. In *International Conference on Database and Expert Systems Applications*, pages 227–237. Springer, 2019.

[118] Tammo Rukat, Dustin Lange, Sebastian Schelter, and Felix Biessmann. Towards automated data quality management for machine learning. In *ML Ops Workshop at the Conference on ML and Systems (MLSys)*, 2020.

[119] Sergey Redyuk, Zoi Kaoudi, Volker Markl, and Sebastian Schelter. Automating data quality validation for dynamic data ingestion. In *EDBT*, pages 61–72, 2021.

[120] Xu Chu, Ihab F Ilyas, Sanjay Krishnan, and Jiannan Wang. Data cleaning: Overview and emerging challenges. In *2016 International Conference on Management of Data*, pages 2201–2206, 2016.

[121] Stefan Larson, Anish Mahendran, Andrew Lee, Jonathan K Kummerfeld, Parker Hill, Michael A Laurenzano, Johann Hauswald, Lingjia Tang, and Jason Mars. Outlier detection for improved data quality and diversity in dialog systems. *arXiv preprint arXiv:1904.03122*, 2019.

[122] Yair Wand and Richard Y Wang. Anchoring data quality dimensions in ontological foundations. *Communications of the ACM*, 39(11):86–95, 1996.

[123] Marc Bellemare, Sriram Srinivasan, Georg Ostrovski, Tom Schaul, David Saxton, and Remi Munos. Unifying count-based exploration and intrinsic motivation. *Advances in neural information processing systems*, 29, 2016.

[124] Rongjun Qin, Songyi Gao, Xingyuan Zhang, Zhen Xu, Shengkai Huang, Zewen Li, Weinan Zhang, and Yang Yu. Neorl: A near real-world benchmark for offline reinforcement learning. *arXiv preprint arXiv:2102.00714*, 2021.

[125] Wenshuai Zhao, Jorge Peña Queralta, and Tomi Westerlund. Sim-to-real transfer in deep reinforcement learning for robotics: a survey. In *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 737–744. IEEE, 2020.

[126] Josip Josifovski, Mohammadhossein Malmir, Noah Klarmann, Bare Luka Žagar, Nicolás Navarro-Guerrero, and Alois Knoll. Analysis of randomization effects on sim2real transfer in reinforcement learning for robotic manipulation tasks. In *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 10193–10200. IEEE, 2022.

[127] Natasha Jaques, Asma Ghandeharioun, Judy Hanwen Shen, Craig Ferguson, Agata Lapedriza, Noah Jones, Shixiang Gu, and Rosalind Picard. Way off-policy batch deep reinforcement learning of implicit human preferences in dialog. *arXiv preprint arXiv:1907.00456*, 2019.

[128] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017.

[129] David Ha and Jürgen Schmidhuber. World models. *arXiv preprint arXiv:1803.10122*, 2018.

[130] Patryk Chrabaszcz, Ilya Loshchilov, and Frank Hutter. Back to basics: Benchmarking canonical evolution strategies for playing atari. *arXiv preprint arXiv:1802.08842*, 2018.

[131] Alexander Hans, Siegmund Duell, and Steffen Udluft. Agent self-assessment: Determining policy quality without execution. In *2011 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*, pages 84–90. IEEE, 2011.

[132] Tom Le Paine, Cosmin Paduraru, Andrea Michi, Caglar Gulcehre, Konrad Zolna, Alexander Novikov, Ziyu Wang, and Nando de Freitas. Hyperparameter selection for offline reinforcement learning. *arXiv preprint arXiv:2007.09055*, 2020.

[133] Ksenia Konyushova, Yutian Chen, Thomas Paine, Caglar Gulcehre, Cosmin Paduraru, Daniel J Mankowitz, Misha Denil, and Nando de Freitas. Active offline policy selection. *Advances in Neural Information Processing Systems*, 34, 2021.

[134] Michael R Zhang, Tom Le Paine, Ofir Nachum, Cosmin Paduraru, George Tucker, Ziyu Wang, and Mohammad Norouzi. Autoregressive dynamics models for offline policy evaluation and optimization. *arXiv preprint arXiv:2104.13877*, 2021.

[135] Justin Fu, Mohammad Norouzi, Ofir Nachum, George Tucker, Ziyu Wang, Alexander Novikov, Mengjiao Yang, Michael R Zhang, Yutian Chen, Aviral Kumar, et al. Benchmarks for deep off-policy evaluation. *arXiv preprint arXiv:2103.16596*, 2021.

[136] Baohe Zhang, Raghu Rajan, Luis Pineda, Nathan Lambert, André Biedenkapp, Kurtland Chua, Frank Hutter, and Roberto Calandra. On the importance of hyperparameter optimization for model-based reinforcement learning. In *International Conference on Artificial Intelligence and Statistics*, pages 4015–4023. PMLR, 2021.

[137] Cong Lu, Philip Ball, Jack Parker-Holder, Michael Osborne, and Stephen J Roberts. Revisiting design choices in offline model based reinforcement learning. In *International Conference on Learning Representations*, 2021.

[138] Vladislav Kurenkov and Sergey Kolesnikov. Showing your offline reinforcement learning work: Online evaluation budget matters. *arXiv preprint arXiv:2110.04156*, 2021.

[139] Ashvin Nair, Abhishek Gupta, Murtaza Dalal, and Sergey Levine. Awac: Accelerating online reinforcement learning with offline datasets. *arXiv preprint arXiv:2006.09359*, 2020.

[140] Yi Zhao, Rinu Boney, Alexander Ilin, Juho Kannala, and Joni Pajarinen. Adaptive behavior cloning regularization for stable offline-to-online reinforcement learning. 2021.

[141] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *International conference on machine learning*, pages 1126–1135. PMLR, 2017.

[142] Vitchyr H Pong, Ashvin Nair, Laura Smith, Catherine Huang, and Sergey Levine. Offline meta-reinforcement learning with online self-supervision. *arXiv preprint arXiv:2107.03974*, 2021.

[143] Zhenshan Bing, David Lerch, Kai Huang, and Alois Knoll. Meta-reinforcement learning in non-stationary and dynamic environments. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2022.

[144] Varun Babbar, Umang Bhatt, and Adrian Weller. On the utility of prediction sets in human-AI teams. *arXiv preprint arXiv:2205.01411*, 2022.

[145] Carrie J Cai, Emily Reif, Narayan Hegde, Jason Hipp, Been Kim, Daniel Smilkov, Martin Wattenberg, Fernanda Viegas, Greg S Corrado, Martin C Stumpe, et al. Human-centered tools for coping with imperfect algorithms during medical decision-making. In *Proceedings of the 2019 chi conference on human factors in computing systems*, pages 1–14, 2019.

[146] Shengpu Tang, Aditya Modi, Michael Sjoding, and Jenna Wiens. Clinician-in-the-loop decision making: Reinforcement learning with near-optimal set-valued policies. In *International Conference on Machine Learning*, pages 9387–9396. PMLR, 2020.

[147] Mehwish Dildar, Shumaila Akram, Muhammad Irfan, Hikmat Ullah Khan, Muhammad Ramzan, Abdur Rehman Mahmood, Soliman Ayed Alsaiari, Abdul Hakeem M Saeed, Mohammed Olaythah Alraddadi, and Mater Hussen Mahnashi. Skin cancer detection: a review using deep learning techniques. *International journal of environmental research and public health*, 18(10):5479, 2021.

[148] Han Wang, Archit Sakhadeo, Adam White, James Bell, Vincent Liu, Xutong Zhao, Puer Liu, Tadashi Kozuno, Alona Fyshe, and Martha White. No more pesky hyperparameters: Offline hyperparameter tuning for rl. *arXiv preprint arXiv:2205.08716*, 2022.

[149] Sungyong Seo, Sercan Arik, Jinsung Yoon, Xiang Zhang, Kihyuk Sohn, and Tomas Pfister. Controlling neural networks with rule representations. *Advances in Neural Information Processing Systems*, 34, 2021.

[150] Lili Chen, Kevin Lu, Aravind Rajeswaran, Kimin Lee, Aditya Grover, Misha Laskin, Pieter Abbeel, Aravind Srinivas, and Igor Mordatch. Decision transformer: Reinforcement learning via sequence modeling. *Advances in neural information processing systems*, 34:15084–15097, 2021.

[151] Ryan Turner, David Eriksson, Michael McCourt, Juha Kiili, Eero Laaksonen, Zhen Xu, and Isabelle Guyon. Bayesian optimization is superior to random search for machine learning hyperparameter tuning: Analysis of the black-box optimization challenge 2020. In *NeurIPS 2020 Competition and Demonstration Track*, pages 3–26. PMLR, 2021.

[152] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, et al. Population based training of neural networks. *arXiv preprint arXiv:1711.09846*, 2017.

[153] Jiazhuo Wang, Jason Xu, and Xuejun Wang. Combination of hyperband and Bayesian optimization for hyperparameter optimization in deep learning. *arXiv preprint arXiv:1801.01596*, 2018.