Trustworthy and Explainable Offline Reinforcement Learning by Inferring a Discrete-State Discrete-Action MDP from a Continous-State Continuous-Action Dataset

Denis Steckelmacher[1][0000−0003−1521−8494] and Ann Nowé[1][0000−0001−6346−4564]

Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussels
`ann.nowe@vub.be`

Abstract. Offline Reinforcement Learning allows to learn a controller for a system from a history of states, actions and rewards, without requiring to interact with the system or a simulator of it. Current Offline RL approaches mainly build on Off-policy RL, such as Q-Learning or TD3, with small extensions to prevent the algorithm from diverging due to the inability to try actions in real time. In this paper, we observe that these incremental approaches mostly lead to low-quality and untrustable policies. We then propose an Offline RL method built from the ground up, based on inferring a discrete-state and discrete-action MDP from the continuous states and actions in the dataset, and then solving the discrete MDP with Value Iteration. Our empirical evaluation shows the promises of our approach, and calls for more research in Offline RL with dedicated algorithms.

Code: https://github.com/vub-ai-lab/offline-rl

Keywords: Offline Reinforcement Learning · Evolutionary Strategies · Discrete Variational Autoencoder

1 Introduction and Related Work

Offline Reinforcement Learning is the research area that considers the problem of learning a good policy from a dataset of states, actions and rewards (16). The dataset is assumed to have been produced by one or more behavior policies, and the aim of the algorithm is to learn a policy that achieves an expected discounted sum of rewards above the one of the behavior policies. Intuitively, we aim at learning, from historical data, a policy that is better than any of the policies used to generate that data.

The main problem of Offline Reinforcement Learning is the inability for the algorithm to perform counterfactual reasoning. There is no way, for the RL agent, to know the quality of some action in some state if this combination of state and action does not appear in the dataset.

Most current Offline Reinforcement Learning build on Off-policy RL algorithms, such as Q-Learning variants, with a special loss or special techniques that prevent the learned policies from being too different from the policies implied by the dataset. This aims at keeping the learned policy in an area where the dataset is usable to compute the quality of actions. Batch-Constrained Q-Learning (7), one of the foundational Offline RL works, modifies the Q-Learning algorithm to only look at actions that have support in the dataset when computing its max operation. TD3-BC (5) builds on the state-of-the-art deterministic-policy TD3 algorithm (6), with a behavior cloning loss to prevent the actor from being too different from the policies in the dataset. Conservative Q-Learning (15) modifies the Q-Learning equations to learn a lower-bound (according to the dataset) of the quality of the actions (produced by the actor). The aim is to ensure that actions not represented well in the dataset have low Q-Values, and are thus not selected by the actor. Implicit Q-Learning (13) uses an expectile regression loss to learn Q-Values with a SARSA-like equation (that requires no explicit policy, and hence looks only at the dataset by definition), then proposes a method to extract a policy from the learned Q-Values. OneStep (3) reviews several policy evaluation (learning Q-Values) and policy improvement algorithms, and shows that, with most combinations, doing a single iteration of evaluation and then improvement outperforms conventional Offline RL algorithms that perform several learning iterations.

The literature on Offline RL is vast, and the above paragraph only scratches the surface. Most of the algorithms mentioned above are reviewed and compared in (10), from which we make the following observations:

- Much research aims at Offline Reinforcement Learning;
- No algorithm is clearly better than the other ones, which one is best varies greatly depending on the task being used for evaluation;
- All the algorithms follow the same approach: take an Off-policy (but online) RL algorithm, extend it in some incremental way to make it amenable to an offline setting.

In this paper, we propose an Offline RL algorithm with a completely original architecture, designed with applicability in industrial settings in mind.

2 Overview of the Contribution

Before presenting background on AI techniques not related to Reinforcement Learning, we briefly present the general idea of our contribution, and thus motivate such diverse background.

The overall intuition of our contribution is that Offline RL is usually desirable in high-cost high-risk environments, such as large industrial plants. Other environments can be interacted with when learning, or can be used to design a simulator, alleviating the need for Offline RL. Considering the high-risk nature of the environment, a policy learned with Offline RL should then never execute an action that none of the behavior policies would execute. This is completely opposite to the current approach at Offline RL (learn a completely fresh policy, but keep it close to the behavior policies), and leads to the following approach:

1. We assume that a finite number of distinct policies was used to generate the dataset. We use a Conditional Variational Autoencoder to identify these policies.
2. We consider that what the agent should learn is not what action to perform in which states, but which policy from the dataset to execute in which state.
3. We assume that which policy from the dataset is the best one changes only at *decision boundaries*, lines/planes in the state space that separate contiguous regions in which a single policy is optimal.
4. We therefore learn a state-space discretization that optimizes the placement of these decision boundaries by:
   (a) Considering a discrete MDP where the finite number of regions are states, and the finite number of policies in the dataset are actions. We detail later what the reward function is.
   (b) Learning Q-Values with Value Iteration.
   (c) Using the average Q-Value as a measure of quality of the MDP.
   (d) Iterating on this process with a Genetic Algorithm to find the state-space discretization that maximizes the quality of the resulting discrete MDP.

The outcome of our algorithm is an identification and approximation of the policies in the dataset, a state-space partitioning, and a policy that maps regions of the state-space to what policy should be executed in it. Our empirical evaluation validates our approach, and illustrate its inherent explainability and trustability aspects.

3 Background

Our contribution being at the intersection of Reinforcement Learning, Genetic Algorithms and Autoencoders, we now briefly introduce these concepts.

3.1 Markov Decision Process

A Markov Decision Process (1) is a discrete-time sequential decision process, with t the time variable that measures the current time-step. The state $s_t \in S$ is an input to the agent at time t, the action $a_t \in A$ is a control signal produced by the agent, the reward $r_t = R(s_t, a_t, s_{t+1})$ measures the quality of an action in a state, the stochastic transition function encodes the dynamics of the environment as $s_{t+1} \sim T(s|s_t, a_t)$, and the initial state distribution desribes how the environment is reset to some initial state at the end of an episode, with $s_1 \sim \mu_0(s)$.

The objective of a Reinforcement Learning agent is to learn an optimal policy $\pi(a|s)$ that maximizes $\mathbb{E}_\pi \sum_t \gamma^t r_t$, the expected sum of discounted rewards obtained by following the policy in the environment. The Reinforcement Learning agent is not allowed to observe T nor the reward function R. This means that Reinforcement Learning is a model-free approach, as opposed to planning that assumes access to a model of the environment.

Conventional *online* Reinforcement Learning algorithms, reviewed in the excellent Sutton and Barto book (21), use a variety of methods that require direct interactions with the Markov Decision Process to learn the optimal policy. *Offline* Reinforcement Learning only has access to historical data of past states, actions and rewards.

3.2 Discrete Variational Autoencoders

Unrelated to Reinforcement Learning, but used in this paper, Variational Autoencoders are a neural network design that learns an identity mapping from an input to itself, but going through an information bottleneck (12). The information bottleneck can be a small number of real numbers, or an integer in some finite range in the case of Discrete Variational Autoencoders (19). An autoencoder can therefore be seen a having two halves. The first one, the encoder, maps an input to a *latent representation*, and the second one, the decoder, maps the latent representation to some output, trained to look similar to the original input.

Variational Autoencoders are primarily used in two ways:

1. Focusing on the *encoder*, to map inputs to a lower-dimensional representation of them, as a form of dimensionality reduction. No major seminal paper appears to introduce this use, but numerous works use Autoencoders like this, for instance to detect faults in ball-bearings (20). This is also how we use Autoencoders in this work.
2. Focusing on the *decoder*, to map samples of latent representations back to full outputs. This allows to train an Autoencoder to learn some complex probability distribution, from which it is then possible to sample by feeding normal-distributed latent representations and obtaining in-distribution outputs from the decoder (18). Such *generative* use is also commonly performed using the well-known Generative Adversarial Networks (9).

Conditional Variational Autoencoders extend the Variational Autoencoders to take an additional input, a context (11). The CVAE therefore learns the distribution $p(x|s)$, with s the context. In this work, we use a Discrete CVAE to learn $p(a|s, \pi_k)$, the distribution of actions as seen in the dataset (thus conditioned on states), with, as a latent representation, simply the index of some policy.

3.3 Genetic Algorithms

Genetic Algorithms, reviewed in the excellent book by Kramer (14), are a gradient-free black-box function optimization technique. Genetic Algorithms are used when some parametric function has to be maximized, the function has no computable gradient (otherwise, Gradient Descent would often be preferred), and is not known to the optimization algorithm.

Genetic Algorithms consider a population of individuals, represented by a computer form of DNA. In the framework we use for our experiments (8), the DNA is a list of real numbers, the same length for every individual. Generation after generation, the individuals are mutated, the best ones are paired to produce offsprings (using a cross-over operator), and the worst individuals are removed from the population. The quality of the individuals is provided to the algorithm by an objective function, that maps DNA to a real value to be maximized. Over time, the quality of the individuals increases, until a global maximum is found (given an infinite amount of time).

4 Contribution

We propose an Offline Reinforcement Learning algorithm, outlined in Section 2, that consists of the following steps:

1. Learn, with a Conditional Discrete Variational Autoencoder, what behavior policies exist in the dataset of states, actions and rewards. Choosing which policy to execute in which states becomes the *action* in a new Markov Decision Process.
2. Optimize a state partitioning neural network, that maps states to clusters. These clusters form *states* in a new Markov Decision Process.
3. The state partitioning is optmized with a Genetic Algorithm to produce Markov Decision Processes that have the highest-possible average value over states.

4.1 Identifying policies

We assume that the dataset available for learning originates from a system in which several behavior policies have been used to select actions. In an industrial setting, this happens often when the engineers vary some parameters of a controller through trial and error, or thanks to gained experience. We assume that there may be no policy that is the best one in every state, and that the Offline RL task to solve is finding, in which state, what policy is the best one.

For this, we propose to use a Conditional Discrete Variational Autoencoder that maps states (the context) and actions to a discrete internal representation, and then states and the internal representation back to actions:

$$z \sim f(z|s,a) \tag{1}$$
$$a' = g(s,z) \tag{2}$$

with f the encoder that maps a state s and action a to a discrete probability distribution, from which an integer *latent variable* z is sampled. g is the decoder, that maps a state and latent variable to a reconstruction of the action a'. Both f and g are implemented using feed-forward neural networks of 1 hidden layer of 128 neurons. The sampling of z from a discrete distribution is not differentiable in its naive form, but we use the straight-through estimator (2) to still attach a gradient to z, so that minimizing the reconstruction loss between a' and a causes gradients to flow into both g and f.

We observe that our Discrete VAE is much simpler than the current state of the art, that uses advanced formulas and non-gradient-descent training schemes (23; 17). In practice, our approach works well for small numbers of policies and state spaces of moderate dimensions. We leave the study of more advanced Discrete VAE architectures to future work.

We propose to call the latent variable z the *policy index*, which allows us to interpret the Autoencoder as learning, in its encoder, to recognize from which policy (for some arbitrary learning-dependent numbering) an action in a state belongs. The decoder is able to map a state and a policy index to the action that the policy would execute in the state.

Considering an Offline RL dataset of (s, a, r, d) tuples of states, actions, rewards, and done signals, we train the Discrete Conditional Variational Autoencoder to minimize the mean squared error between a and $g(s, z \sim f(z|s, a))$, the reconstruction loss (12). We observed that no other loss is required. In particular, there does not seem to be necessary to encourage f to learn a highly stochastic probability distribution, or any other form of regularization.

4.2 Partitioning the state

We assume an Offline RL dataset with continuous states. We make the observation that *which policy is best to execute in which state* usually does not change for every small epsilon variation of the state variables, but instead changes along big *decision boundaries*, occasional large lines that traverse the state space.

The regions inside these lines form clusters of states. The clusters may not have straight edges, and as such, we choose to use a feed-forward neural network $c_\theta(s)$ to map a state s (from the environment) to a cluster index. We use a neural network with one input per state variable, a single hidden layer followed by a tanh activation function, and one output per possible cluster index (the user then has to configure C, how many clusters to produce). Which cluster a state belongs to is then the arg-maximum of these outputs: $c(s) = \text{argmax}_i\, c_\theta(s, i)$.

The number of neurons in the single hidden layer can be quite low (down to 8 still provides meaningful partitions in our Table experiment) mainly because we expect the state partitioning to be relatively simple, made of a small number of clusters of simple shapes. In the interest of reducing the number of hyper-parameters of our approach, we choose in our experiments to set that number to the same number as neurons in the Variational Autoencoder hidden layers (128).

Now that we introduced the feed-forward network used to partition the state-space, we can turn to how to train it. We want to learn a state partitioning that maximizes the average value of a policy over policies (which policy from the dataset to execute in which state), with the decision boundaries defined by the neural network. This problem is not differentiable, so no gradient descent approach can be used. We instead turn to Genetic Algorithms, but first, in the next section, detail how to evaluate a state partitioning.

4.3 Discrete Markov Decision Process

We temporarily assume that a state partitioning is provided (in the next subsection, we explain how to learn it), and define how we can build a discrete-state and discrete-action Markov Decision Process from that state partitioning, the Conditional Discrete Variational Autoencoder described above, and the Offline RL dataset provided to the agent:

States
States in the discrete MDP are clusters. We can therefore map a continuous state from the environment to a discrete cluster index, to be used as state index in the discrete MDP.

Actions
Actions in the discrete MDP are policies from the original dataset, as detected by the Conditional Discrete Variational Autoencoder. Executing an action in the discrete MDP therefore means *executing the a-th policy*. The decoder of the VAE is then able to map a continuous state and a policy index to a continuous action.

Rewards
The reward for executing a policy in some cluster is defined to be the sum of rewards, from the dataset, for any state that belongs to the cluster and an action that belongs to the policy; divided by how many times trajectories *leave* the cluster, so how many experiences in the buffer have their state in the cluster and next-state outside the cluster.

Transition Function
We simply map the states and next states from the dataset to clusters and next clusters. We count how many times a cluster transitions to another cluster, and then normalize these counts to obtain probabilities.

Termination Function
We introduce a special absorbing cluster number 0 to which any experience from the Offline RL dataset transitions when its *done* flag is true.

Initial State Distribution
We count how many times the initial state of every trajectory belongs to each of the clusters, and normalize these counts.

Inferring an MDP from another MDP is also sometimes referred as building a latent-space model (4) and is usually done to be able to prove properties on the inferred MDP, that still echo to the original MDP. It is possible to measure how closely the inferred MDP matches the original MDP by using metrics such as bisimulation (22).

From this discrete MDP, of which the reward and transition functions are known, a value function V can be computed with Value Iteration (21).

4.4 Optimizing the state partitioning

The above section assumes a state partitioning and uses it to infer a discrete MDP, and solve it with Value Iteration. We propose to learn the state partitioning using a genetic algorithm:

Population

The individuals in the population are weights for the feed-forward neural network that maps states to clusters. In our experiments, we use populations of 100 individuals.

Objective Function

Given an individual, the weights that define a state-space clustering, the objective function builds the according discrete MDP as described in the previous section, solves it with Value Iteration, and uses the average value $V(s)$ over s μ_0 as the quality of the individual. μ_0 is the initial state distribution.

Implementation Details

The actual algorithm relies on PyGAD (8), a Python library for genetic algorithms. We use the default mutation and cross-over operators. PyGAD's TorchGA module is used to map individuals to and from PyTorch neural networks, used to perform the actual state clustering. The DNA of the individuals is simply the list of weights of the neural network.

After several generations (500 in our experiments), we obtain weights for the partitioning neural network that lead to discrete MDPs that have the highest-possible average value across states. This partitioning, combined with the learned Q-Values for the resulting discrete MDP, and the Discrete Conditional Variational Autoencoder, form the learning outcomes of our algorithm.

4.5 Running the optimized policy

After the genetic algorithm produces its final state partitioning, we can either look at the partitioning for explainability purposes (we do that in our experiments), and/or use the learning outcomes mentioned in the previous section to control the system. For this, we need to define $\pi(s)$, the policy that maps a state from the original environment to an action to perform in that environment. We do this according to this procedure:

1. Map s to a cluster index using partitioning feed-forward neural network $c(s)$.
2. Use the Q-Table produced by Value Iteration to find which policy index is best in that cluster: $z = \operatorname{argmax}_k Q(c, k)$.
3. Use the decoder of the Conditional Variational Autoencoder, along with the environment state, to produce the action to execute: $a = g(s, z)$

Assuming that the VAE learned a good model of the policies in the dataset, we observe that the actions executed in the environment will always be very similar to what the behavior policies would execute, with the only learned aspect being what policy is best in which state.

5 Evaluation

Evaluating Offline RL approaches is fundamentally challenging because, in their real application domains, there is no supervised dataset nor environment available for querying, and thus the policies learned through Offline RL have no way to be evaluated. However, we still propose an evaluation that aims at answering the following questions about our approach:

1. Does it run and produce outputs in a reasonable amount of time on several datasets?
2. Do the results (policies identified, state partitioning, learned optimal policy) make sense? Would they lead to good performance in the real system?

We evaluate our proposed Offline RL method on two datasets. The first one is produced from a Gymnasium environment, Table, a simple 2D environment. On Table, we are able to run the learned policy and compute its actual return in the true environment. The second dataset is generated by the MATLAB Brayton Cycle demo, in which fuel is combusted and turned into mechanical energy thanks to gas turbines. This second dataset has more dimensions, more complex policies (stacked PID controllers) and better illustrates how our method can be used in an industrial setting.

We note that Offline RL is usually evaluated on a standard set of benchmarks, usually using the Minari framework,[1] but we observed that these benchmarks do not represent industrial settings. Industrial tasks are lower-dimensional but have a more dynamical nature. The Offline RL dataset they come with usually contains vast amounts of data, but from only a small number of distinct policies.

5.1 Table

The agent is a 2D point on a 1-by-1 table. The state is the (x, y) coordinates of the agent. The action is a 2D vector (dx, dy). Every time-step, the agent moves to $(x + 0.01dx, y + 0.01dy)$ on the table.

If the agent falls of the table (either x or y outside of the $[0, 1]$ range), the episode terminates with a reward of -50. If the agent enters the region $(0.5 \pm 0.05, 0.5 \pm 0.05)$, the episode terminates with a reward of 100. Otherwise, the episode continues with a reward of 0. Episodes are truncated after 100 time-steps.

The **offline dataset** is generated by repeatedly executing either $(1, 1)$, $(1, -1)$, $(-1, 1)$ or $(-1, -1)$. Which action to execute is selected randomly every 300 time-steps, and then maintained during 300 time-steps. The action selection ignores the state and the termination, which causes some episodes in the dataset to "change direction" in the middle of the episode.

The environment and the behavior policies are extremely simple, leading to easily interpretable results. A figure of the environment will be shown along with the results of applying our method to it.

5.2 Brayton Cycle

The Brayton Cycle, known for its use in jet engines, consists of two back to back gas turbines, in between whose air is heated up by burning some fuel. The "front"

[1] https://minari.farama.org/

turbine spends energy taking in air and compressing it. The "back" turbine uses the compressed and heated up air to produce mechanical energy, some of which sent to the front turbine, some of which made available to an external energy consumer.

The Brayton Cycle represents a simple-enough real-world multi-physics setting, that is usually controlled using PID or cascaded PID controllers. In this paper, we use the Mathworks Simulink Brayton Cycle example,[2] with only a small wrapper script around it. We hope that this example is accessible easily enough for the research community to ensure ease of reproduction of our research.

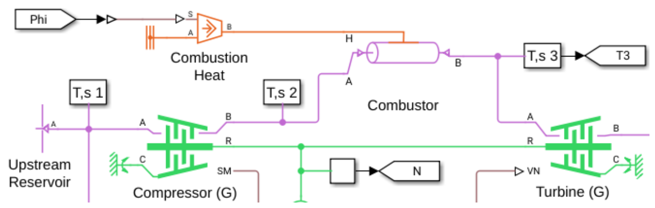

Fig. 1. Simplified view of the Simulink simulation of the Brayton Cycle, with one turbine taking in air (purple), a gas (purple) heater that heats the air between the two turbines, and a second turbine mechanically tied to the first one (green) extracting energy from the hotter air.

The system is controlled by setting Phi, how much fuel is injected, and VN, the nozzle opening of the second turbine. The inputs to the controller are the rotational speed of the turbines, the temperature of the air after combustion, and the surge margin of the first turbine (SM in Figure 1). Over 1000 seconds, two setpoints move in a stepwise fashion: the desired rotational speed, and the desired surge margin. We do not control nor changed the setpoints and use the ones provided in the Simulink example.

The **offline dataset** is generated by using 6 variations of the PID controller that control the second turbine's nozzle opening. The P and I gains are $(1,1), (2,1), (1,2), (0.5,2), (2,0.5), (2,2)$. We then log the inputs to the controller as states, and the P and I gains of that PID as action (actions are therefore constant for an entire variation in the dataset). We log the efficiency of the system as reward. Such data collection is common in the industry and represents the setting of *we don't know which PID gains are best in which situation and would like an Offline RL agent to tell us.*

5.3 Results on Table

We run our algorithm on Table with the following parameters:

[2] https://nl.mathworks.com/help/hydro/ug/sscfluids_brayton_cycle.html

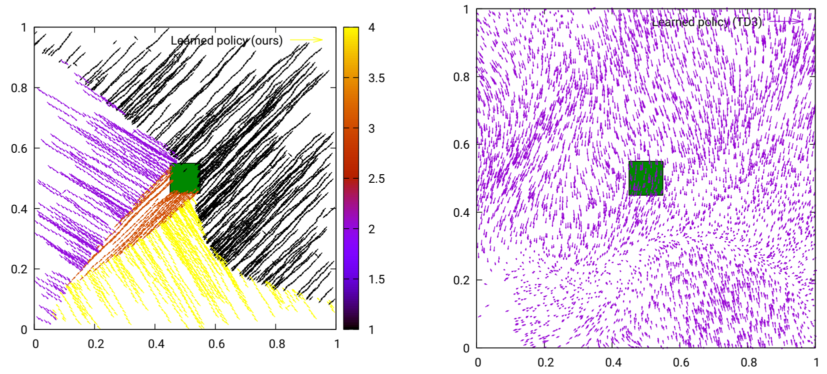

Fig. 2. On Table. *Left:* learned state-space partitioning (colors) and resulting learned policy (arrows) with our Offline RL algorithm. The arrows point towards the goal and represent a high-quality policy. *Right:* same for TD3+BC. The arrows point towards some attractor state, that is not the true goal state. TD3+BC failed to learn a good policy.

Identifying policies 6 policies to identify, even though we know that the dataset only contains 4, this is to show that this parameter can be over-estimated. The encoder and the decoder of the VAE are both single-hidden-layer networks with 128 neurons in the hidden layer.

Partitioning the state 5 clusters, even though we know that 4 would be optimal. This is again to show that this parameter can be over-estimated.

Value Iteration 50 Value Iteration steps are performed before the Q-Values are used to evaluate a state partitioning.

After about 30 minutes of run-time on an AMD Ryzen 5 3500U laptop (4 cores, 2.1 to 3.7 Ghz, 16 GB of RAM), a state clustering that uses 4 clusters is found, along with 4 policies (that correspond to the policies used for the dataset generation). The clustering is shown in Figure 2 and allows to reach the goal from almost any state. A small region on the bottom-left of the goal has arrows pointing away from the goal, but the average return obtained by this policy in the environment is still 91.3. We compare how our method evaluates the quality of the policies (using Q-Values) to actual Monte-Carlo returns in Figure 3.

We also ran TD3+BC (5) on this dataset. We choose TD3+BC because this algorithm is very close to state of the art in Offline RL, and has an official open-source implementation.[3] After about 30 minutes, TD3+BC learns a policy that achieves a return of 20, which is significantly worse than what our approach learned. Interestingly, the average Q-Values produced by the TD3 critics is 90 after several minutes, and then explore to 300 or more. TD3 therefore seems to over-estimate the Q-Values (the optimal policy for this environment achieves a return of 100). We can observe the effect of this discrepancy in Figure 2:

[3] https://github.com/sfujim/TD3_BC

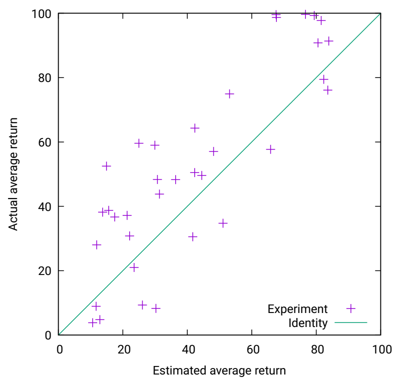

Fig. 3. On Table: comparison of the estimated average return of the Offline RL policy (horizontal axis), for several policies produced through the optimization procedure, with the actual average return obtained by the policy in the environment (vertical axis). We observe a good correlation ($R^2 = 0.73$), and thus our method can be trusted to evaluate the quality of the policies it produces.

TD3+BC learns a policy that goes towards some goal location, but the location is not the actual goal location.

5.4 Results on the Brayton Cycle

We run our algorithm on the Brayton Cycle example with the following parameters:

Identifying policies 10 policies to identify, even though we know that the dataset only contains 6, this is to show that this parameter can be over-estimated.
Partitioning the state 10 clusters, to be sure to over-estimate this parameter.
Value Iteration 50 Value Iteration steps are performed before the Q-Values are used to evaluate a state partitioning.

After about 9 minutes on the same computer as Table, our method finds an interesting policy for the Brayton Cycle, shown in Figure 4. The P gain of the PI controller is chosen to be 2 in every state. The I gain is chosen to be 0.5 in almost every state, except at the beginning where it is chosen to be 2. In Figure 4, we see that the system is tightly controlled by the PI controllers and follows a single trajectory, with almost no points for making decisions. The trajectories start at the bottom (low temperature, 5K RPM) and then move upwards. The small yellow bit indicates the region in which I is chosen to be 2.

We note that the Brayton Cycle Simulink demonstration is not made in a way that allows to change the PID gains mid-simulation. Thus, it is not possible to run our learned policy on the system and compute an expected return. We can only evaluate the policy from a qualitative perspective. These results are still very encouraging for several reasons:

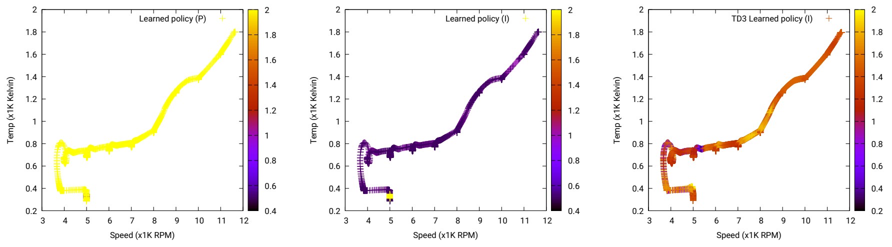

Fig. 4. Learned policy for the Brayton cycle, the policy makes sense for this task. The dataset contains 6 almost-identical trajectories, that start at the bottom (low temperature, 5K RPM) and go to the top. *Left:* P gain learned to be always 2. *Middle:* I gain learned to be 0.5 in almost every state except when starting up (then it has to be 2). Some chatter in the color exists for the I gain because the VAE is slightly stochastic in reconstructing the I gain from a policy index. *Right:* TD3 learns a policy that continuously changes the PID gains and has them larger (here I is shown).

1. The dataset is very challenging with its narrow trajectory, yet both the VAE and the state clustering managed to learn something meaningful: not zeros everywhere, or a constant action, and the learned policy makes sense from a control perspective.
2. In this system, it actually makes sense to control it with a higher I (integral) gain at the beginning, when errors are larger and may prevent the system from starting. When the system is running in steady state, almost no control has to be exerted on it, hence the low I gain. Low I gains are possible when P is high, which is exactly what the agent learned (P is chosen to be 2 in a choice of 0.5, 1 or 2).
3. While interpreting these results requires knowledge about the Brayton cycle, we note that such interpretation is possible. The learned state partitioning (Figure 5) shows the importance of good decisions at startup (many clusters at the bottom of the figure) and matches the human partition of Brayton Cycle regimes: startup, spinning up, and then steady state (the last state, on top of the figure).

We also ran TD3+BC on the Brayton Cycle dataset and obtained a policy shown in Figure 4. TD3+BC learns Q-Values that are significantly overestimated (600+, with episodes in the dataset having a return of at most 400), and a policy that changes the P and I gains often, in a continuous way. We observe no difference in the learned TD3 policy between the startup, spinning-up and stady-state regimes.

6 Conclusion

We introduced an Offline RL method built from the ground up with the specific needs of Offline RL in an industrial setting in mind. Our algorithm ensures that

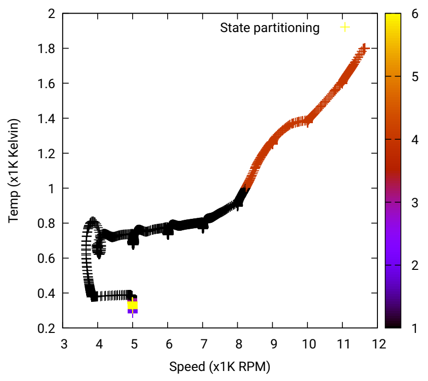

Fig. 5. State space partitioning learned for the Brayton cycle. Many states in the beginning of the trajectories (correctly) indicate the importance of good control when starting the system. Then, one state represents the spinning-up regime, and a final state considers the steady-state system (towards the top). Not every state has a different optimal action, hence this Figure shows more detail than Figure 4.

the learned policy does not execute actions not present in the dataset, and does so by learning which behavior policy (auto-detected from the dataset) to execute in which state, with changes in behavior policy only happening along a small number of decision boundaries (for explainability).

Our experiments show that our method works with two easy-to-visualize datasets of industrial relevance. One of them, Table, illustrates that very low-quality behavior policy can be combined in an high-quality learned policy. The Brayton cycle demonstrates that our method works even for dataset in which only a small amount of states, in a 3D space (only 2 dimensions shown in our figures), are visited. In both cases, our method learned a better policy than what TD3+BC would.

These results show the promise of our from-the-ground-up method, and while the method is not designed for, nor evaluated on the robotic Minari tasks, it shows promise on closer-to-real-world problems, with existing control already available on it, and a need for explainable learning outcomes.

Acknowledgments

We acknowledge financial support from the CTRxAI strategic basic research project, founded by Flanders Research Foundation (FWO S007723N), and from the Flanders AI Research Program.

Bibliography

[1] Bellman, R.: A markovian decision process. Journal of mathematics and mechanics pp. 679–684 (1957)

[2] Bengio, Y., Léonard, N., Courville, A.: Estimating or propagating gradients through stochastic neurons for conditional computation. arXiv preprint arXiv:1308.3432 (2013)

[3] Brandfonbrener, D., Whitney, W., Ranganath, R., Bruna, J.: Offline rl without off-policy evaluation. Advances in neural information processing systems **34**, 4933–4946 (2021)

[4] Delgrange, F., Nowé, A., Pérez, G.A.: Distillation of rl policies with formal guarantees via variational abstraction of markov decision processes. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 36, pp. 6497–6505 (2022)

[5] Fujimoto, S., Gu, S.S.: A minimalist approach to offline reinforcement learning. Advances in neural information processing systems **34**, 20132–20145 (2021)

[6] Fujimoto, S., Hoof, H., Meger, D.: Addressing function approximation error in actor-critic methods. In: International conference on machine learning. pp. 1587–1596. PMLR (2018)

[7] Fujimoto, S., Meger, D., Precup, D.: Off-policy deep reinforcement learning without exploration. In: International conference on machine learning. pp. 2052–2062. PMLR (2019)

[8] Gad, A.F.: Pygad: An intuitive genetic algorithm python library. Multimedia Tools and Applications pp. 1–14 (2023)

[9] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., Bengio, Y.: Generative adversarial networks. Communications of the ACM **63**(11), 139–144 (2020)

[10] Kang, B., Ma, X., Wang, Y., Yue, Y., Yan, S.: Improving and benchmarking offline reinforcement learning algorithms. arXiv preprint arXiv:2306.00972 (2023)

[11] Kim, J., Kong, J., Son, J.: Conditional variational autoencoder with adversarial learning for end-to-end text-to-speech. In: International Conference on Machine Learning. pp. 5530–5540. PMLR (2021)

[12] Kingma, D.P., Welling, M.: Auto-encoding variational bayes. arXiv preprint arXiv:1312.6114 (2013)

[13] Kostrikov, I., Nair, A., Levine, S.: Offline reinforcement learning with implicit q-learning. arXiv preprint arXiv:2110.06169 (2021)

[14] Kramer, O.: Genetic algorithms. Springer (2017)

[15] Kumar, A., Zhou, A., Tucker, G., Levine, S.: Conservative q-learning for offline reinforcement learning. Advances in Neural Information Processing Systems **33**, 1179–1191 (2020)

[16] Levine, S., Kumar, A., Tucker, G., Fu, J.: Offline reinforcement learning: Tutorial, review, and perspectives on open problems. arXiv preprint arXiv:2005.01643 (2020)

[17] Lorberbom, G., Gane, A., Jaakkola, T., Hazan, T.: Direct optimization through argmax for discrete variational auto-encoder. Advances in neural information processing systems **32** (2019)

[18] Ribeiro, J.M.L., Bravo, P., Wang, Y., Tiwary, P.: Reweighted autoencoded variational bayes for enhanced sampling (rave). The Journal of chemical physics **149**(7) (2018)

[19] Rolfe, J.T.: Discrete variational autoencoders. arXiv preprint arXiv:1609.02200 (2016)

[20] San Martin, G., López Droguett, E., Meruane, V., das Chagas Moura, M.: Deep variational auto-encoders: A promising tool for dimensionality reduction and ball bearing elements fault diagnosis. Structural Health Monitoring **18**(4), 1092–1128 (2019)

[21] Sutton, R.S., Barto, A.G.: Reinforcement learning: An introduction. MIT press (2018)

[22] Taylor, J., Precup, D., Panagaden, P.: Bounding performance loss in approximate mdp homomorphisms. Advances in Neural Information Processing Systems **21** (2008)

[23] Vahdat, A., Andriyash, E., Macready, W.: Dvae#: Discrete variational autoencoders with relaxed boltzmann priors. Advances in Neural Information Processing Systems **31** (2018)