AutoML–RvS: More automated offline RL training

Ramanan Abeyakaran ram.abey2001@berkeley.edu
Yuxi Liu yuxi_liu@berkeley.edu

Abstract

With an aim to further automate offline RL, we applied AutoGluon, an AutoML package, to create a pipeline for training offline RL policies with minimal human tuning of hyper-parameters and model choice whatsoever. We then benchmarked its performance against a previous algorithm, RvS-R, on three Gym MuJoCo environments `HalfCheetah`, `Hopper`, and `Walker2d`. We found that

env name	`HalfCheetah`	`Hopper`	`Walker2d`
policy stability	stable	unstable	unstable
improves with higher quality dataset?	yes	no	yes
maximal normalized score	45	25	70
average normalized score	2–42	3–27	4–28

Figure 3 shows the results synoptically.

The results are encouraging, although it plateaus at only 25% of the performance of expert policy, and does not reliably scale with quality of offline dataset. We believe that more finetuning would help with the performance, and encourage more research into this AutoML–RvS pipeline.

1. Introduction

1.1 Contributions

Ramanan Abeyakaran wrote some parts of the report introduction, ran some experiments with his Google Colab pro account, and made empty promises.

Yuxi Liu wrote most of the report, went to lots of office hours, reviewed the literature, and solved several dependency hells. He apologizes for being unable to make this report less funny.

1.2 The promise and struggle of RL

AI may be divided into two parts: the "what" and the "how": What kind of program to run, and how to find that program? For the "what", the main answers are: real-time feedback mechanisms, symbolic programs, numerical functions, and their hybrids. For the "how", the main answers are manual programming, automated search, supervised learning (SL), reinforcement learning (RL), and meta-programming. Table 1 shows the $3 \times 5 = 15$ possibilities.

RL has a long history, but only in the past decade has there been successful applications in fields such as strategic game playing, robotic control and autonomous vehicle navigation.

	real-time feedback mechanism	symbolic program	numerical program
human design	classical control heory: Watt governor, PID controller, Kalman filter	typical Python programs	typical FORTRAN programs
automatic search	Darwinian evolution, evolutionary hardware design, Ashby's homeostat	proof search by SAT solvers[b], Logic Theorist	convex programming, dynamic programming, evolutionary search
supervised learning	Pavlovian conditioning, system identification	program synthesis by large language models: GPT-f, GitHub Copilot	statistical regression, support vector machine, neural network
reinforce-ment learning	operant conditioning, adaptive filters	program synthesis by RL: AlphaTensor	typical RL systems
meta programming	self-improving robotic systems, the Autofac	Eurisko, Gödel machine	AIXI, self-referential weight matrix, learning to reinforcement learn

Table 1: The 15 possible ways to combine the "what" and "how" of AI. The columns are the "what", and the rows are the "how". Each box contains the key examples for the particular combination of "what" and "how".

a. By the Curry–Howard correspondence, proof search is equivalent to program search.
b. By the Curry–Howard correspondence, proof search is equivalent to program search.

Advances in deep neural network architectures and ground breaking chip technologies popularized the sub-field of Deep Reinforcement Learning (DRL) and is responsible for these recent breakthroughs.

The main difference between SL and RL is that SL is "linear" while RL is "loopy", fortuitously illustrated by their origins. SL grew out of statistical regression, with its enlightenment ideal of "pure observation" exemplified by Gauss's estimation the orbit of Ceres by the least-squares method. The orbit of Ceres is an observable, and no action by Gauss could possibly affect it. RL grew out of sequential decision theory, with its Cold War ideal of "fight and win wars" exemplified by Skinner's pigeon-guided missiles. Victory is not an observable – if it is observable, the game is already over – and only actions that could affect the outcome are considered. Unlike SL, the distribution of observations encountered by a RL agent depends on what the RL agent is. Consequently, RL is inherently loopy.

Positive feedback is the root of all instability. Like a screaming microphone placed too close to the speaker, the loopiness of RL is both its power and bane. Whereas in SL, convergence is easy to prove in theory and observe in practice, in RL, convergence is often theoretically impossible (such as in fitted value iteration) and not observable in practice. This extreme instability explains why in RL research, reproducing results often requires exactly replicating the random seed used, learning rate schedules, the software version number, and other "implementation details" (ARS+20; EIS+20).

1.3 Offline RL

The main paradigm in RL is online policy iteration, which is schematically an {observation–action–reward–policy improvement} loop. This loop leads to two problems: high sample complexity and instability. It has high sample complexity, since every time the policy improves, previously collected data is obsoleted.

A common paradigm to solve high sample complexity is model-based RL (MBRL), where a model of the environment is built from real data, and the policy is trained inside the model. The problem with this paradigm is that now there are two loops: an {(environment, policy)–model} slow loop, and a {policy–model} fast loop. This paradigm has a usual failure mode where the policy exploits the model to get fake high rewards. Successful MBRL generally requires GAN-like optimization methods instead of vanilla gradient descent methods (HRU+17; RMK20).

Offline RL is an alternative new paradigm. In offline RL, the agent learns from a fixed dataset of collected prior interactions, without the need for additional online interactions. This allows the agent to learn from experience that has been collected by other agents, simulators, or even humans, and to reuse this experience to solve new tasks or adapt to changing environments.

The key idea is to break the loop, making RL more like SL. Schematically, the process is {(environment, collection policy) $\rightarrow$ dataset $\rightarrow$ trained policy}.

1.4 RL via SL (RvS)

Despite breaking the online RL loop, many offline RL algorithms have another loop which remains unbroken, and that is in Bellman updates. For example, the Q-learning algorithm

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_t + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t))$$

leads to the optimism bias (VHGS16), since if at some point in the training, the Q-function overvalues a certain state-action pair (s_t, a_t), this overvalue would propagate to state-action pairs leading to that. If Q-function is not tabular, but fitted (as in all DRL cases), then all state-action pairs nearby would also be overvalued. Similar worries apply for offline actor-critic algorithms like AWAC (NGDL20), since they also use Bellman updates.

On the other extreme are various imitation learning algorithms, either plain or filtered. They perform no Bellman update, and so they lose temporal compositionality.

(EEKL21) proposes to organize these methods under the umbrella term "RL via SL" (RvS), and shows that two simple RvS algorithms can perform at SOTA level among offline RL algorithms, using just two-layered MLP trained with standard SL methods.

1.5 AutoML

Despite the ideal of pure observation, even SL is not immune from the loop. Typically, a SL system is built by a {design–test–tweak hyperparameters} loop, with the "tweak hyperparameters" part done by "graduate student descent".

The productivity of hardware makers have increased at a much faster rate than the productivity of programmers. As a consequence, we have Baumol's cost disease in information technology: hardware costs have crashed just as programmer salary has soared[1]. Recalling Amdahl's law, this means that for an AI system with a human in the loop to run faster, it would eventually have to paralellize the human labor, change the type and proportion of human labor in the loop, or somehow make Moore's law apply for humans too.[2] The bitter lesson (Sut19) is bitter because human labor stands still while computer labor races forward.

AutoML is just using computers to perform the "tweak hyperparameters" step. The main methods of AutoML are blackbox search, multi-armed bandit, gradient-based search, and metalearning. There have been a few attempts at automating hyperparameter search in RL algorithms, notably (ZRP$^+$21) where they performed a hyperparameter search for a MBRL algorithm, resulting in a hyperparameter setting that broke the MuJoCo simulator. However, to the extent of our knowledge, there has been no publication where AutoML is combined with RvS. This final project is the first attempt at combining an off-the-shelf AutoML package with RvS.

1. Over 40% undergrads of UC Berkeley are in the EECS department!
2. This is not "intelligence amplification", which does not make humans run faster, but rather replaces human labor from certain areas and letting them specialize more. By "make Moore's law apply for humans", we do not mean creating Python so that humans don't have to write assembly code by hand, but making humans write assembly code faster. Healthy humans can merely run as fast as healthy humans 100 years ago, except now they mostly drive cars.

2. Theory

2.1 Striving for simplicity in RL

The phrase "striving for simplicity" is a snowclone in machine learning literature, used to describe situations where a simple method does just as well as the more complex methods currently in use. Its first use was (SDBR14) for supervised learning of computer vision.

Complex RL algorithm being unstable, unreliable, and ununderstood theoretically, there has been several attempts in striving for simplicity in RL.

(RLTK17) studied training RL agents for continuous control problems in MuJoCo. They trained two kinds of policies: "linear" and "RBF". The linear policies were of form $a_t \sim N(Ws_t + b, \Sigma)$, where $N(\mu, \Sigma)$ is the normal distribution of mean μ and covariance Σ, and W, b, Σ are trained. The RBF policies were of form $a_t \sim N(Wy_t + b, \Sigma)$, where $y_t = \sin\left(\nu^{-1}Ms + \phi\right)$ is the random feature vector, with M, ϕ normally sampled at initialization (not trained). They found that such simple policies were competitive with TRPO (SLA+15), the most advanced algorithm around 2016.

(MGR18) proposes Augmented Random Search (ARS), a derivative-free optimization algorithm that produces deterministic, linear policies of the form $\pi_{(M,\mu,\Sigma)}$, defined by trainable parameters (M, μ, Σ). The policy take in observation s_t, normalize it to $x_t = \Sigma^{-1/2}(s_t - \mu)$, then take action $a_t = Mx_t$. It exceeds or matches state-of-the-art performance on the MuJoCo Gym environments `Hopper-v1`, `Walker2d-v1`, `Ant-v1`, and `Humanoid-v1`, taking less than 100 CPU-hours each.

2.2 RvS

In RvS, the policy is of form $\pi(a_t|s_t, \omega_t)$, where ω_t is a conditioning variable. For RvS-G, ω_t symbolizes "goal state", that is, a state sampled uniformly at random from the rest of the episode, as in $\omega_t \sim \text{Uniform}(s_{t+1}, ..., s_H)$ where H is the end of the episode. For RvS-R, it symbolizes "reward-to-go", that is, the average reward per step of the rest of the episode, as in $\omega_t = \frac{1}{H-t+1}\sum_{t'=t}^{H} r_{t'}$.

2.3 D4RL

Datasets for Deep Data-Driven Reinforcement Learning (D4RL) is a benchmark for evaluating offline RL algorithms (FKN+20). It consists of dataset from over 40 tasks across 7 qualitatively distinct domains, each collected with various forms of policies (random, medium, a mixture of medium and expert, etc). It is used by the RvS paper, and so we chose it as well to have an apples-to-apples comparison.

2.4 AutoGluon

AutoGluon (EMS+20) is a SOTA software package that could perform SL at the top human level[3]. It performs SL by training a sequence of models, then combining them into an ensemble model. To use the training time (specifiable by the user) economically, it trains

3. "In two popular Kaggle competitions, AutoGluon beat 99% of the participating data scientists after merely 4h of training on the raw data."

the cheap and low-capacity models (such as random forests) earlier, and the expensive and high-capacity models (such as shallow neural networks) later.

3. Experiments

Due to many issues (see Section A), we ran very few experiments worth reporting.[4]

We performed three kinds of experiments: Partially reproducing the RvS paper; using AutoGluon to produce policies from the same data used for the RvS paper; testing the policies produced by AutoGluon by the same benchmarks for the RvS paper.

3.1 Partially reproducing RvS

In order to benchmark the AutoGluon policies by the same standard, we installed the RvS codebase. Due to many technical difficulties, we only got a part of it running, enough to run the benchmarks. Details may be found in Section B.

In short, out of all the algorithms and tasks benchmarked in the RvS paper, we set up the three MuJoCo Gym environments (`HalfCheetah`, `Walker2d`, `Hopper`). Out of all the policies, we set up the RvS-R policy.

3.2 Using AutoGluon to produce policies

We downloaded the D4RL datasets for 3 MuJoCo environments (`HalfCheetah`, `Walker2d`, `Hopper`) collected by 4 collection policies (`random`, `medium-replay`, `medium`, `medium-expert`). Their meanings are as follows:

- `random`: a random policy defined as $a_t = \tanh(x)$ where $x \sim N(0, 1)$.
- `medium`: a SAC policy, but not trained until convergence;
- `medium-replay`: a dataset sampled from the states encountered during the training of `medium`;
- `expert`: a SAC policy, trained to convergence;
- `medium-expert`: a 50-50 mixture of `medium` and `expert`.

As shown in Figure 3, the distribution of reward-to-go in the datasets has a clear progression as the dataset progresses from `random` to `medium-expert` for each environment.

We found that k-nearest-neighbor policy took a long time to validate, and also had very high L2 loss, so we excluded it from training.

These MuJoCo Gym environments all have action space of form $[-1, +1]^n$ where n is the dimension of action space. However, AutoGluon could not automatically perform regression for variables in a bounded range, so we manually added a preprocessing step to the actions of the datasets by `x' = np.arctanh(0.995 * np.clip(x, -1.0, 1.0))`. We refer to this as the "arctanh action" in the figure captions.

4. The final project guidelines said that "Successful reports will have a main body that is about eight pages in length.", so we had to write a very long introduction to get to eight pages.

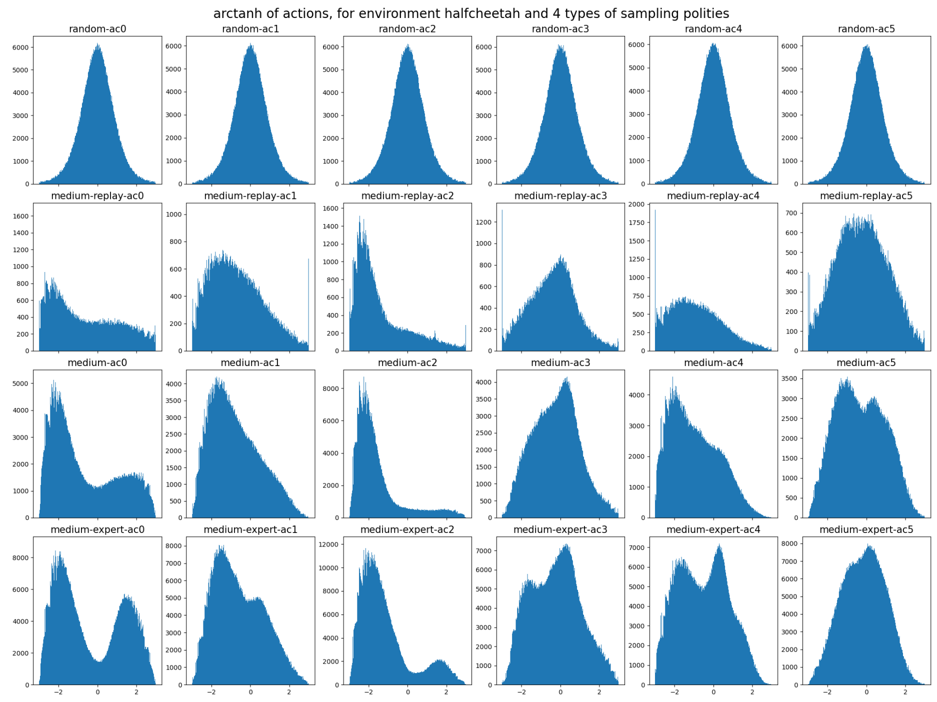

Figure 1: The histograms of arctanh-actions for the `HalfCheetah-medium-expert`, compared to the predictions by the policies produced by 4 AutoGluon quality settings. Eyeballing, the medium quality setting qualitatively gives the best match.

AutoGluon has 4 quality settings: medium, good, high, best. We trained one of each for the `HalfCheetah-medium-expert` dataset, each for 2 hours, in order to see which one performs the best under the given time constraint. The best was medium quality, as shown by Figure 2 and Table 2. After that, we trained 12 medium-quality policies for the 3 environments $\times$ 4 collection policies, taking 24 hours in total.

	`ac0`	`ac1`	`ac2`	`ac3`	`ac4`	`ac5`
`medium`	0.203319	0.174317	0.201113	0.152283	0.163099	0.189098
`good`	0.220616	0.188009	0.213779	0.167066	0.177785	0.201476
`high`	0.219693	0.187197	0.214508	0.167088	0.177537	0.201598
`best`	0.21679	0.184454	0.210596	0.16318	0.174045	0.199378

Table 2: The average L2 loss for each action dimension for `HalfCheetah-medium-expert` with 4 AutoGluon quality settings. The medium quality setting achieves the lowest loss.

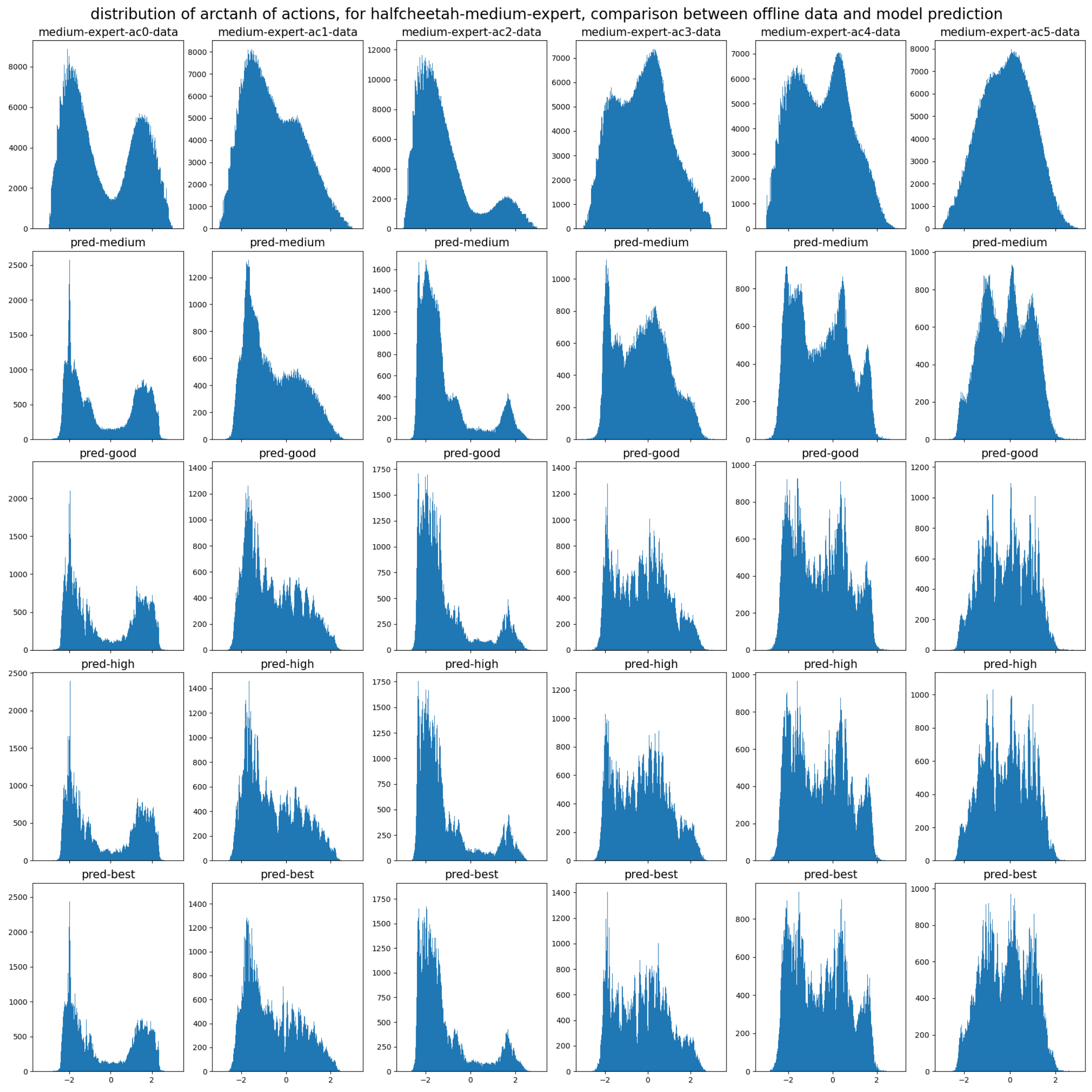

Figure 2: The histograms of arctanh-actions for the `HalfCheetah-medium-expert`, compared to the predictions by the policies produced by 4 AutoGluon quality settings. Eyeballing, the medium quality setting qualitatively gives the best match.

3.3 Benchmarking the policies generated by AutoGluon

After training 12 policies by AutoGluon, we benchmarked them using the script from the RvS codebase.

Benchmarking RvS-R-like policies means rolling out such policies conditioned on different target reward-to-go. The data is processed in two ways: One, the actually achieved average episodic reward is plotted against the target reward-to-go as violin plots. Two, the best average episodic reward is normalized against the rewards achievable by expert policy and random policy by the formula

$$100 \times \frac{\text{policy episodic reward} - \text{random policy episodic reward}}{\text{expert policy episodic reward} - \text{random policy episodic reward}}.$$

Table 3 for the relevant benchmark scores against which we would compare AutoGluon policies.

	random return	expert return	RvS-R, normalized score
HalfCheetah	-285	12330	92.2
Hopper	20*	3624	101.7
Walker2d	2*	4005	106.0

Table 3: Data from D4RL GitHub Wiki and RvS paper (EEKL21). *Re-estimated, because original data is grossly in error.

Figure 3 shows the violin plots of achieved reward-to-go of the policies, when conditioning on different target reward-to-go. This may be compared with Figure 6 of (EEKL21). Table 4 shows the normalized scores of the AutoGluon policies.

The results show that the AutoGluon policies can achieve higher rewards than the offline dataset averages, and so it is a viable strategy. It performs consistently for `HalfCheetah` environment as the offline dataset quality improves. However, the performance is highly unstable for `Hopper` and `Walker2d` environments.

We hypothesize that it is because the environments `Hopper` and `Walker2d` both allow falling down and early termination, whereas `HalfCheetah` has no possibility of early termination. Studying the exact failure mode that led to the model falling down in some episodes, but walking straight for other episodes, is left for future studies.

4. Conclusion

We used AutoGluon, an AutoML package, to further simplify the pipeline for training offline RL policies. We benchmarked its performance against a previous algorithm, RvS-R.

The results are encouraging, although it plateaus at only 25% of the performance of expert policy, and does not reliably scale with qulity of offline dataset. We believe that more finetuning would help with the performance, but we ran out of computing budget as well as patience.

We invite more researchers to explore this application of AutoML to offline RL, and wish them more luck and computing power than we did.

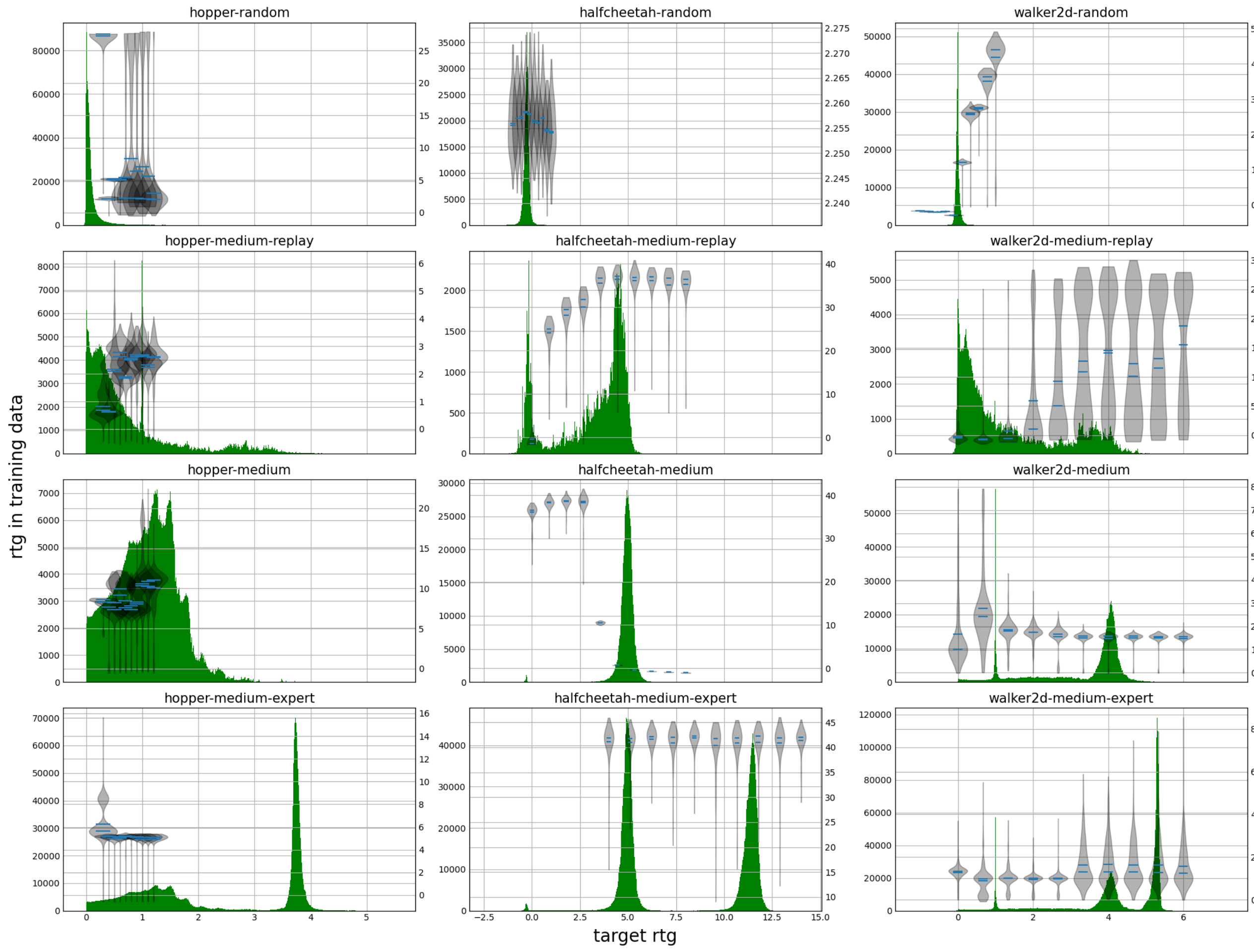

Figure 3: The violin plots of achieved reward-to-go of the policies when conditioning on different target reward-to-go. Compare with Figure 6 of (EEKL21).

	RvS-R	AutoGluon	AutoGluon (unnormalized)
halfcheetah-random-v2	3.9	2.3	0
hopper-random-v2	7.7	27.3	1003
walker2d-random-v2	-0.2	4.3	170
random-v2 average	3.8	11.3	
halfcheetah-medium-replay-v2	38.0	36.4	4310
hopper-medium-replay-v2	73.5	2.6	115
walker2d-medium-replay-v2	60.6	15.5	621
medium-replay-v2 average	57.4	18.2	
halfcheetah-medium-v2	41.6	38.5	4574
hopper-medium-v2	60.2	10.5	400
walker2d-medium-v2	71.7	27.6	1108
medium-v2 average	57.8	25.5	
halfcheetah-medium-expert-v2	92.2	41.9	4996
hopper-medium-expert-v2	101.7	6.2	245
walker2d-medium-expert-v2	106.0	16.5	663
medium-expert-v2 average	100.0	21.5	
gym-v2 average	54.7	19.1	

Table 4: Normalized scores for the rewards achieved by the RvS-G and AutoGluon policies. RvS-G data taken from Table 1 of (EEKL21).

References

[ARS+20] Marcin Andrychowicz, Anton Raichuk, Piotr Stańczyk, Manu Orsini, Sertan Girgin, Raphael Marinier, Léonard Hussenot, Matthieu Geist, Olivier Pietquin, Marcin Michalski, et al. What matters in on-policy reinforcement learning? a large-scale empirical study. *arXiv preprint arXiv:2006.05990*, 2020.

[EEKL21] 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.

[EIS+20] Logan Engstrom, Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Firdaus Janoos, Larry Rudolph, and Aleksander Madry. Implementation matters in deep policy gradients: A case study on PPO and TRPO. *arXiv preprint arXiv:2005.12729*, 2020.

[EMS+20] Nick Erickson, Jonas Mueller, Alexander Shirkov, Hang Zhang, Pedro Larroy, Mu Li, and Alexander Smola. Autogluon-tabular: Robust and accurate automl for structured data. *arXiv preprint arXiv:2003.06505*, 2020.

[FKN+20] Justin Fu, Aviral Kumar, Ofir Nachum, George Tucker, and Sergey Levine. D4RL: Datasets for deep data-driven reinforcement learning. *arXiv preprint arXiv:2004.07219*, 2020.

[HRU$^+$17] Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. Gans trained by a two time-scale update rule converge to a local nash equilibrium. *Advances in neural information processing systems*, 30, 2017.

[MGR18] Horia Mania, Aurelia Guy, and Benjamin Recht. Simple random search provides a competitive approach to reinforcement learning. *arXiv preprint arXiv:1803.07055*, 2018.

[NGDL20] Ashvin Nair, Abhishek Gupta, Murtaza Dalal, and Sergey Levine. Awac: Accelerating online reinforcement learning with offline datasets. *arXiv preprint arXiv:2006.09359*, 2020.

[RLTK17] Aravind Rajeswaran, Kendall Lowrey, Emanuel V Todorov, and Sham M Kakade. Towards generalization and simplicity in continuous control. *Advances in Neural Information Processing Systems*, 30, 2017.

[RMK20] Aravind Rajeswaran, Igor Mordatch, and Vikash Kumar. A game theoretic framework for model based reinforcement learning. In *International conference on machine learning*, pages 7953–7963. PMLR, 2020.

[SDBR14] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806*, 2014.

[SLA$^+$15] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR, 2015.

[Sut19] Richard Sutton. The bitter lesson. *Incomplete Ideas (blog)*, 13:12, 2019.

[VHGS16] 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.

[ZRP$^+$21] 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.

Appendix A. The process of the final project

The project was roughly divided into four phases, all of which were horrible.

In the original plan, drawn up at the formation of the project group, Yuxi Liu (YL) would perform literature survey, theoretical analysis, and report-writing, and Ramanan Abeyakaran (RA) would perform programming. This plan did not survive.

In phase 1, they wrote a project proposal describing in vague words about studying exploration in RL. Unsurprisingly, it got a 8/10 for being too vague. YL then performed a general survey of all the main methods of exploration in RL and wrote a document summarizing the methods.

In phase 2, RA disappeared for a month due to not checking the email. YL wrote the milestone report assuming that RA dropped the course and that he would do the project alone. YL realized that he was good at learning and explaining, but bad at innovating. After meeting tutors for a few times YL thought up a new topic: studying the effect of coding observation space in RL exploration. He learned to use `Ray/RLLib` expecting to use that for the final project.

In phase 3, YL realized that he still had no idea how to do that topic, so he went to more office hours searching for a new topic. RA suddenly appeared after checking the email. They finally met at an office hour and had a discussion which was fun, but had no utility for actually writing the final report or its experiments.

In phase 4, they gave up all illusions of writing anything important and simply aimed to not fail the course. YL finally decided on a project idea that seemed like it would require no creativity whatsoever and would take about a week of concentrated work (he was wrong on both accounts), wrote a programming plan and asked RA to start working through it. He was then absorbed by Chinese politics during the turbulent week of November 24–30. 10 days later, RA have not even finished the first step in the plan (that is, getting the RvS codebase running – see Section B), so YL took three days to do it. The plan did not survive dependency hell intact, and they downscaled it again to just what they could manage to run – just RvS-R on 3 MuJoCo Gym environments.

Communication was dysfunctional. They planned to meet weekly, but it never actually happened. Face-to-face meeting happened exactly once during an office hour. YL insisted on using email but RA insisted on using SMS messaging. This ended in an equilibrium state where YL mainly used email and RA mainly used SMS, like how in some Chinese immigrant families, the parents would talk to children in Chinese, and the children would reply in English.

Appendix B. Getting RvS to run

Though written just a year ago, the RvS codebase is already a dependency hell to run. As a brief sketch of the dependencies involved, it requires both `MuJoCo 2.1.0` (for `mujoco-py`) and `MuJoCo 2.1.1` (for `dm-control`). Attempting to install it on Windows does not work, since there is no way to make `MuJoCo 2.1.0` to run on Windows. It also requires `pytorch <= 1.8` (for GCSL), but `pytorch 1.7.1` does not support the GPU units we had access to (CUDA architecture 8.6), and running the experiments without GPU takes 4 times as long. We eventually decided to comment out all the code requiring GCSL, reasoning that replicating the RvS-R experiments would be enough.

When it was finally running, the code takes about an hour to run per experiment on an A5500 GPU. Merely recreating the D4RL benchmarks (3 environments, 4 collection policies, and 5 random seeds) for RvS-R would require 60 hours of computing. Consequently we gave up on replicating the work.

Complicating matters were the various "frictions".[5] The Windows Subsystem for Linux 2 uses `winsock`, which is broken by using VPN. This was eventually solved by simply deleting VPN. While MuJoCo 1.5 could be installed on Windows, `mujoco_py 1.5` cannot despite it being a "known good past version". The bug only appears when it is first imported and compiled by `Cython` – a temporary file generated during compilation has a filename longer than 260 characters, which Python refused to process further. Jupyter Notebook spontaneously broke twice and could not be fixed except by deleting the entire `miniconda3` installation and reinstalling from scratch. Another time, RvS started mysteriously crashing upon a certain command, with not diagnosis information. It was again fixed by reinstalling the entire `miniconda3`. One overnight AutoGluon run was cut short by a blue screen of death.

5. Clausewitz defined it as "the force that makes the apparently easy so difficult", such that "I doubt if a commander ever launched an operation of any magnitude without being forced to repress new misgivings from the start."