Neural Stochastic Differential Equations for Uncertainty-Aware Offline RL (ICLR 2025): An attempted Reliable Jax Implementation

Matthew Lowery
Department of Computer Science
University of Utah
mlowery@cs.utah.edu

Abstract

Reproducibility is a very important pillar of scientific research. Without it, how are we able to expand upon another's work? With the expectation to publish at warp-speed in order to advance along in the field, it's value becomes pushed lower and lower on the totem pole. It is not that researcher's are trying to be dishonest! It is just that there is less of a weight placed upon it by the broader community. For instance, in an ICLR submission, a world-class ML venue, it is merely optional to submit a code base alongside a paper. Accordingly, we attempt to reproduce a subset of the results of an interesting, very recent paper: Neural Stochastic Differential Equations for Uncertainty-Aware Offline RL [5] accepted into ICLR 2025. In such an attempt, we note missing hyperparameter mentionings in the paper, a code base in which one has to dig through eight different files to see what something is doing, and versioning issues/ bugs with the code base requirements (which meant that it could not be run trivially, nor could the datasets be downloaded). Thus, the math was all we could rely upon in order to make a working implementation. In that regard we coded an auto-differentiable sde solver using a Jax library called diffrax [3] and made our own inverted pendulum dataset for training it to predict a next state distribution. We also wrote an A2C policy optimization strategy in Jax, which

1 Introduction

This paper discusses offline model-based reinforcement learning. The 'offline' descriptor is to say that the agent we are training has no ability to interact with the environment beyond a fixed dataset of previous interactions with the world. This is relevant in contexts where interacting with the environment, particularly when having a novice skill set, is costly. It could be costly in the sense of: an agent spending one's actual money because its task is to reinvest your assets; or costly because the agents task is to prescribe medicine and thus there are steaks involved.

Moving towards the second descriptor, 'model-based,' this implies that our goal is to learn a representation of the environment, i.e. $P(s_{t+1}|s_t, a_t)$. Versus model-free RL, where an agent is only able to react to the environment, in this case the agent now try to develop an intuitive understanding of the environment itself. It can now make an educated guess for the question: How will the environment respond given the action that I choose?

For more concrete reasons why model-based RL is helpful, if one has a model of the world, one can also simulate the world, which could help with small dataset sizes. Moreover, model-based RL enables us to be task agnostic, as we could use the learned world model to train an agent to, say, build a fire OR to build a shelter.

Class Project for Advanced AI, Spring 2025.

The phrasing of learning a model of the environment is precisely, 'learn the dynamics of the environment.' This has a precise meaning. This is to say one now focuses on the continuous perspective of state, just as in class we have focused on the continuous perspective of actions. Now we view $\frac{ds(t)}{dt} = f(t, s(t), a(t))$, where there is a function describing the way the environment changes given an action and current state. To get the current state, we must integrate in time, or solve this ordinary differential equation (ODE)! If we want to parameterize this, we look to neural ODEs (NODEs) [2] (add θ in the subscript of f). In order to get gradients with respect to a loss function using various $s(t_n)$'s we have integrated, it is required to either solve ODE backwards in time, called an adjoint equation or (less efficiently) to backpropagate through the forward solvers steps.

To now view state as a probability distribution, one has to change this NODE to a neural stochastic differential equation (NSDE) [7], [8], [4].

2 Problem Statement

The goal of model-based reinforcement learning is to attain a parameterized model, which predicts a distribution over the next possible state, and potentially the reward signal as well, using the current state and action, that mimics the real environment as closely as possible. In the offline setting, one hopes to use this model generatively to supplement a fixed dataset so that a better policy could be learned than could have been previously. An unfortunate pitfall to this aim is model exploitation, wherein an agent trains it's policy using the model and exploits the model's lack of knowledge on the outskirts of the data distribution, where it might, for example, predict extraneous rewards. Thus countermeasures to letting model make predictions it is highly uncertain about have ensued in research.

3 Related Work

3.1 Preventing Model Exploitation

A main focus of this paper is to prevent an agent exploiting the learned environment dynamics. There are previously introduced ways to mitigate this, such as if the uncertainty at each step in a rollout (a way to add more data to train an agent's policy using the dynamics model) accumulates and surpasses some threshold, the rollout is nullified [13]. Another is to directly penalize the agent, via the value function, from doing things it has not seen in the dataset [10]. This paper uses a method which punishes any state action pair chosen by the agent which is 'far' in an l_2 norm sense from the state action pairs in the dataset. Previously, a k-nearest neighbors approach [12] was taken in this regard, which required a costly search to find the closest neighbor in the dataset of the proposed pair. The authors instead train a separate model to estimate this value, less expensively.

Other model-based offline RL algorithms such as [11] (MOPO) and [9] (MOBILE) learn probabilistic ensemble dynamics and mitigate model exploitation through conservative mechanisms. MOPO penalizes rewards based on model uncertainty. In contrast, this method models probabalistic model dynamics with an NSDE where uncertainty is captured continuously through a learned diffusion term. This yields a principled, data-driven form of regularization that discourages policy exploitation in out-of-distribution regions without relying on explicit reward penalties.

3.2 Neural SDEs

Multiple styles of NSDE's have been realized, but this paper assumes Gaussianity, where each stochastic transition is assumed to be unimodal. Although a next state distribution could certainly be multimodal, and there are established ways to train NSDEs to predict arbitrarily shaped distributions over its stochastic transitions. For instance, a GAN style training (2021) as in [4] or diffusion style training as in [8].

In addition, to compute gradients there are more optimal methods [6], both in accuracy and temporal/memory complexity, than autodifferentiating through the solver's steps, which given that the authors did not mention, were likely not employed.

4 Method

4.1 Tools

Neural differential equation ideology is relatively new, and by consequence training associated models is not as fleshed out as is training standard MLPs or ConvNets. We must both have a solver which is robust to nonlinear terms, and we must also be able to execute gradients with respect to the parameters underlying the model. Accordingly, we employ a library which was written by a student who did his thesis on Neural ODES, coined Diffrax [3]. Here robust SDE solvers are available, and various methods for computing gradients are accessible.

This library exists on top of Jax [1], a python library which makes it relatively easy to write high level linear algebra code to run on a GPU. It exposes a numpy-esque api to the user so that vectorized operations are straightforward. Those operations are light-weighted wrappers around Accelerated Linear Algebra (XLA) commands, which is the powerhouse open source compiler of Jax. It boils down the code in two stages of optimizations: first target-agnostic optimizations and then target-dependent optimizations. In the former optimizations such as operation fusion, algebraic simplifications, constant folding, control flow normalization and dead code elimination might occur. In the latter, for example, the Cuda specific optimizations would occur, such as memory management optimization, tiling and threading, unrolling, etc.

Now, in the Jax-numpy functions a user writes, they are compiled for a fixed datatype as well as data size. For a change in either, a recompilation step occurs. For no change, the compiled function is repeatedly used. This is very much amenable to machine learning pipelines, which resulted in many convenient libraries for this being written on top of Jax. This is all to say it is better than PyTorch. Except with our little experience in RL we notice PyTorch's statefulness is quite convenient and one is not as partial to fixed batch sizes, which is helpful, for instance, in policy gradients: if we continuously do gradient updates with respect to a new episode from that environment, the episode's length is of course not fixed.

4.2 Math

Here, as previously mentioned, the authors attempt to learn transition dynamics:

$$s_{t+1} \sim p_\theta(s_{t+1} \mid s_t,\ a_t)$$

where:

- s_t is the current state,
- a_t is the action,
- s_{t+1} is the next state,
- p_θ is the learned dynamics model parameterized by θ.

By minimizing a negative log-likelihood between the predicted distribution over the next state and the true next state given data:

$$L(\theta) = -\mathbb{E}_{(s_t,a_t,s_{t+1})\sim D}\left[\log p_\theta(s_{t+1}|s_t,a_t)\right]$$

$$\mathcal{D} = \{(s_t, a_t, s_{t+1})\}_{t=1}^{N}, \quad \theta^* = \arg\min_\theta\ \mathcal{L}(\theta)$$

To note: the authors could have easily had the model predict an entire episode when training it (like it would be in generating synthetic model rollouts), but they always just predict the next time step only (horizon $h = 2$) with the solver. The whole point of having a continuous model is that you can solve forward for any t you choose! Then we would suppose this loss as an integral over t.

The chosen model parameterization over s is:

$$\mathrm{d}s \ = \ f_\theta(s,a)\,\mathrm{d}t \ + \ \Sigma_\phi(s,a)\,\mathrm{d}W_t\,.$$

Where subscript terms include trainable parameters. In particular, this is a neural SDE, where Σ_ϕ is what scales the variance of a given state transition distribution, and f_θ tracks the mean of this distribution. Thus, Σ_ϕ can be conveniently trained to effectively quantify how far a current s_t, a_t pair is from the dataset using some carefully chosen decomposition of Σ into sub models. f_θ is also decomposed to leverage some prior knowledge of physics in the Mujoco datasets (basically, the fact that velocity is a function of a position).

The notion that the model can also be trained to predict rewards is glossed over, although it is necessary if we are to generate full sythentic data tuples to supplement the original dataset and train a policy. One can 'append' a model, $dr_s = f_{\theta_r}(s_t, a_t)$ to the SDE solve, by adding a row of zeros in the diffusion matrix as we're not predicting a distribution over it. The additional loss term to train this, assumed to be a simple mean squared error, is not mentioned. We have this version of the model implemented.

To assert that Σ predicts the 'farness' of s_t, a_t from the dataset, a few extra loss terms are needed other than $\mathcal{L}_{data}$: $\mathcal{L}_{sc}$, $\mathcal{L}_{mu}$, $\mathcal{L}_{grad}$ (page 7 equation 7).

4.3 Implementation

Code exists at `https://github.com/matthew-lowery/NSDE_ORL`.

4.3.1 Data collection

Our implementation is agnostic to the dataset, so long as it is a rigid body type and has a velocity for every position in a particular piece of a system, i.e. a joint. We initially tried the latest release of half-cheetah, but reverted to the inverted pendulum problem because for difficult datasets, it was hard to decipher whether the model was not not converging because it was wrong is wrong or it is not expressive enough. Ideally we would have had access to the authors' fixed datasets collected from a of an agent of a given skill level interacting with an environment. Alas, we encountered some weird bugs as the datasets needed to be collected from external software. Thus we collected samples from the latest inverted pendulum-v5 gym environment in Gymnasium. And later once things were working, also collected samples form the latest Swimmer-v5 environment.

4.3.2 Training the Model

Model sizes for all problems are standardized and reported in the paper, so we set those in a config file, while each loss term has a coefficient to weight its effective importance for this method: $\lambda_{grad}, \lambda_{data}$ and so on. Yet neither λ_{data} nor λ_{sc} are reported. Do we assume they are to be 1? In our initial experiments, the relative scales of the losses, given these assumed coefficients were: $\sim 1e4$, $1e-6$, $1e1$, and $1e0$ for $\mathcal{L}_\mu$, $\mathcal{L}_{grad}$, $\mathcal{L}_{data}$, and $\mathcal{L}_{sc}$ respectively. Orders of magnitude different! Thus, the model was overwhelmingly biased to minimizing $\mathcal{L}_\mu$, despite the fact that $\mathcal{L}_{data}$ of course is paramount.

Thus just to see if the model would converge on the dataset, we ran an experiment removing the other loss terms. We collected 400 examples for training and 100 for testing, shuffling it first and using the Adam optimizer. With a little fiddling of the solver tolerance, immediate signs of learning (figure 1)! What a relief. The model achieved a mean absolute error of 0.144 on the test dataset for predicting cumulative reward at a current time step, and a $0.095\,\ell_2$ relative error in predicting the next state without training it to the ground: 400 epochs in about 5 minutes on a Nvidia 4080 GPU.

Next question, can it fit a slightly harder dataset out of the box? Given the same dataset train test split size, and in a similar number of epochs, we achieve a mae of 0.136 and $0.191\,\ell_2$ on Gym's Swimmer-v5 for the same two test metrics respectively (figure 2).

If we transition to probabilistic training, things seem to fair well with a λ_μ set to 0.005. This makes its scaling on par with the other losses. Now we see signs of learning for all the metrics.

4.3.3 Training the Policy

We attempted to write a couple versions of the A2C Actor Critic algorithm in Jax, based on the blog post mentioned in class `https://medium.com/@dixitaniket76/advantage-actor-critic-a2c-algorithm-explained-and-implemented-in-pytorch-dc3354b60b50`,

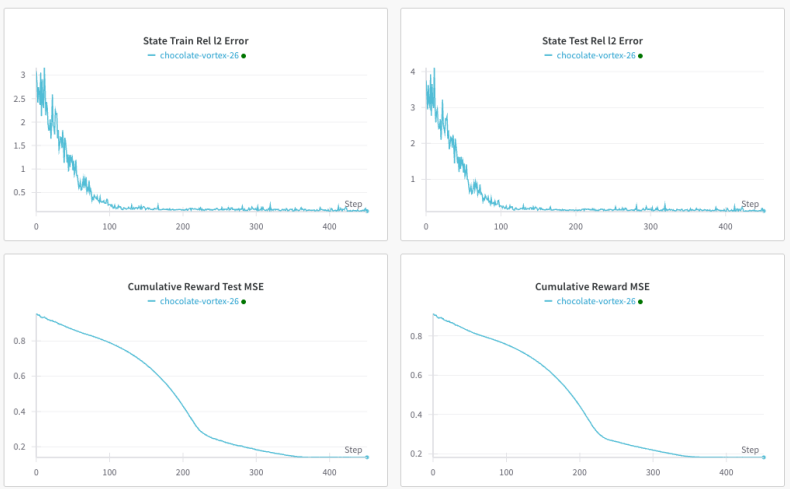

Figure 1: Initial Model experiment on Gymnamsium's Inverted Pendulum v5 `https://gymnasium.farama.org/environments/mujoco/inverted_pendulum/`, wherein all loss terms save for the data fitting loss are commented out.

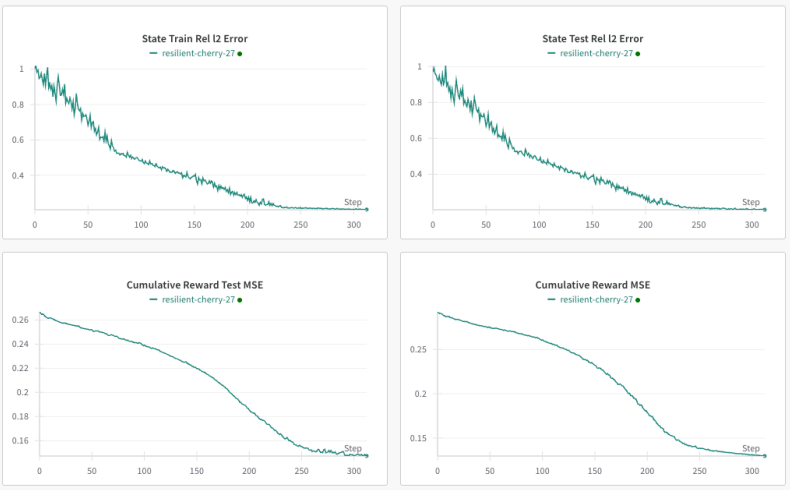

Figure 2: Initial Model experiment on Swimmer-v5 `https://gymnasium.farama.org/environments/mujoco/swimmer/`, wherein all loss terms save for the data fitting loss are commented out.

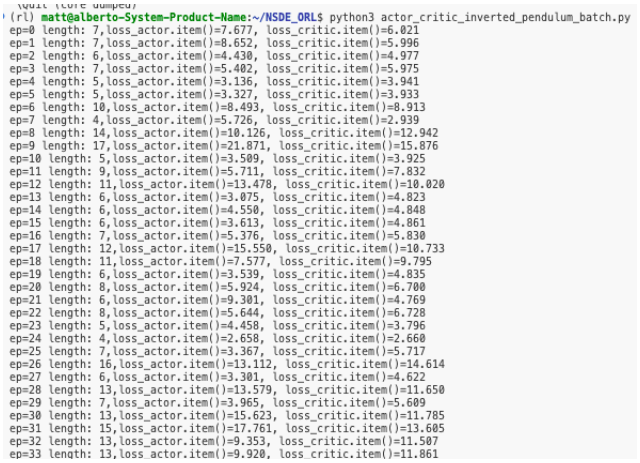

Figure 3: Sample output of our Jax A2C implementation in `https://github.com/matthew-lowery/NSDE_ORL/blob/main/actor_critic_inverted_pendulum.py` with episode lengths and each model's losses reported.

as ideally we would have used the model and its uncertainty aware rollouts to both generate data and train a policy. We just did not have enough time to get it working properly. For the A2C algorithm, we tried both updating the actor critic models based on an episode rollout batch, and during the episode rollouts, but we believe there is a minor bug lurking. We tried normalizing the advantage; we tried ensuring the actor model predicted action space outputs in the correct domain $[-3, 3]$ for the inverted pendulum by tanhing its output and multiplying it by three. We tried outputing the standard deviation of the actor and training it as well, instead of it being fixed. The agents score will sometimes gain to certain highs like balancing for 20 time steps and then collapse to 6, it's initial skill level. We did not suspect it was for lack of tuning the model because the dataset was not a difficult one. A sample output of the training is show in figure 3.

Regardless, as far as incorporating the learned NSDE model in a offline policy optimization procedure, the uncertainty is easily quantified by solving the learned SDE at whichever time points required from an initial state action pair, then querying the sub model of Σ, $\eta(s_t, a_t)$ at each t and summing the values. The idea is that if the sum goes beyond a tuned threshold, truncate the rollout or nullify it entirely. Otherwise, query the trained NSDE model just as one would the gym environment to retrieve a next state and predicted reward (which is also discounted by the model's uncertainty estimate) and train a policy.

5 Results & Conclusions

This implementation obviously incomplete. But it is reassuring that not too many extrapolations from the math and details written in the paper have to be made in order to get something working. We were able to create had a continuous, effective model of state for arbitrary Mujoco datasets. We were able to do this in just a few code files versus the hundred in the original paper's codebase: `https://github.com/cevahir-koprulu/sde4mbrltatu/tree/iclr2025`. Given that we had no background in reinforcement learning prior to this course, implementing (for the most part) a freshly published ICLR paper on the topic was rewarding.

We would have liked to explore the probabilistic aspects of the model to a much greater degree. For example visualizing the generative states the model produced, and seeing if they were reasonable. Moreover, training the model with a GAN style procedure to see how expressive it could really be.

Of course the real goal was meant to be training a policy on some fixed dataset, then training a state model on it, and sampling synthetic rollouts to add to the fixed dataset and hopefully be able to learn a better policy. We are eager to get things working on our own time.

6 Results and Conclusion

References

[1] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[2] Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. Neural ordinary differential equations, 2019.

[3] Patrick Kidger. *On Neural Differential Equations*. PhD thesis, University of Oxford, 2021.

[4] Patrick Kidger, James Foster, Xuechen Li, Harald Oberhauser, and Terry Lyons. Neural sdes as infinite-dimensional gans, 2021.

[5] Cevahir Koprulu, Franck Djeumou, and ufuk topcu. Neural stochastic differential equations for uncertainty-aware offline RL. In *The Thirteenth International Conference on Learning Representations*, 2025.

[6] Xuechen Li, Ting-Kam Leonard Wong, Ricky T. Q. Chen, and David Duvenaud. Scalable gradients for stochastic differential equations, 2020.

[7] Xuanqing Liu, Tesi Xiao, Si Si, Qin Cao, Sanjiv Kumar, and Cho-Jui Hsieh. Neural sde: Stabilizing neural ode networks with stochastic noise, 2019.

[8] Yang Song, Conor Durkan, Iain Murray, and Stefano Ermon. Maximum likelihood training of score-based diffusion models, 2021.

[9] Yihao Sun, Jiaji Zhang, Chengxing Jia, Haoxin Lin, Junyin Ye, and Yang Yu. Model-Bellman inconsistency for model-based offline reinforcement learning. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 33177–33194. PMLR, 23–29 Jul 2023.

[10] Tianhe Yu, Aviral Kumar, Rafael Rafailov, Aravind Rajeswaran, Sergey Levine, and Chelsea Finn. Combo: Conservative offline model-based policy optimization, 2022.

[11] Tianhe Yu, Garrett Thomas, Lantao Yu, Stefano Ermon, James Zou, Sergey Levine, Chelsea Finn, and Tengyu Ma. Mopo: Model-based offline policy optimization, 2020.

[12] Hongchang Zhang, Jianzhun Shao, Shuncheng He, Yuhang Jiang, and Xiangyang Ji. DARL: distance-aware uncertainty estimation for offline reinforcement learning. In Brian Williams, Yiling Chen, and Jennifer Neville, editors, *Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023, Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence, IAAI 2023, Thirteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2023, Washington, DC, USA, February 7-14, 2023*, pages 11210–11218. AAAI Press, 2023.

[13] Junjie Zhang, Jiafei Lyu, Xiaoteng Ma, Jiangpeng Yan, Jun Yang, Le Wan, and Xiu Li. Uncertainty-driven trajectory truncation for data augmentation in offline reinforcement learning, 2023.