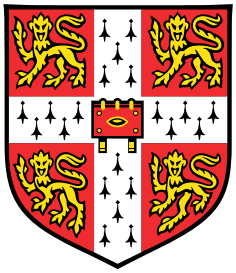
Deep Reinforcement Learning for 3D Molecular Design



Adrian Salovaara Black

Department of Engineering
University of Cambridge

This dissertation is submitted for the degree of
Master of Philosophy in Machine Learning and Machine Intelligence

Homerton College

August 2022

Declaration

I, Adrian Salovaara Black of Homerton College, being a candidate for the MPhil in Machine Learning and Machine Intelligence, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

In this work, I used and extensively modified the Python software package MolGym (Simm, Pinsler, and Hernández-Lobato, 2020; Simm, Pinsler, Csányi, et al., 2021) to incorporate new molecular modeling capabilities. Principally, I implemented a novel infinite atom bag paradigm along with facilities for automatic entropy adjustment, stochastic atom penalties, and imitation learning. Naturally, I also used MolGym's software dependencies. Of note: PyTorch (Paszke et al., 2019), Sparrow (Husch, Vaucher, and Reiher, 2018; Bosia et al., 2021), e3nn (Geiger et al., 2022), and ASE (Larsen et al., 2017).

The length of this dissertation is 14,962 words.

Adrian Salovaara Black
August 18, 2022

Acknowledgements

I must express gratitude to my advisors José Miguel Hernández Lobato, Gabor Csanyi, and Gregor Simm for providing me thoughtful guidance and insights throughout this project. I'd also like to thank Raf Czlonka for providing computing support. Lastly, thank you to my friends and family, near and far!

Abstract

By applying machine learning to molecular design, researchers aim to tame vast molecular search spaces and accelerate the discovery of useful structures. A notable new approach, MolGym, leverages deep reinforcement learning to sequentially assemble molecules on a 3D canvas. MolGym demonstrates compelling results, but is restricted to building molecules that correspond with pre-specfied chemical formulae. To address this limitation, we introduce a novel paradigm where agents may learn the optimal size and composition of molecular structures. In particular, we adopt dynamic per-atom penalties as a means to guide, but not unduly constrain, the structures agents assemble. We also implement stochastic environments with randomized penalties in order to further empower agents and broaden molecular diversity. Although we successfully construct small molecules, we encounter significant optimization challenges in scaling to larger, complex structures. We thus propose and implement an imitation learning technique to ameliorate optimization.

Table of contents

Chapter 1

Introduction

Precisely designing molecules with desirable characteristics holds tremendous potential in applications like material synthesis and drug discovery. However, the search space of useful molecules is exceedingly vast. For instance, it is estimated that there are between 10^{30} and 10^{60} synthesizable and realistic, drug-like structures (Polishchuk, Madzhidov, and Varnek, 2013). Researches have thus sought to leverage machine learning in the hunt for beneficial molecules.

In this work, we adopt a specific reinforcement learning approach to molecular design: MolGym (Simm, Pinsler, and Hernández-Lobato, 2020; Simm, Pinsler, Csányi, et al., 2021). In MolGym, an agent learns to construct molecules by sequentially placing atoms onto a 3D canvas. The objective of the agent, encoded into a sparse reward function, is to form molecules that are low-energy and thereby stable. This is a complex modeling task and thus MolGym employs deep neural networks.

To build a molecule, a MolGym agent sequentially selects atoms from a finite, pre-specified bag. As atoms are placed on the canvas, the bag is gradually emptied. When the bag is completely empty, the molecule is considered complete. But this framework has important limitations. For instance, one may reasonably intend the agent to learn to cease placing atoms before the bag is empty. The currently formed canvas molecule may be stable and forcing additional atoms energetically unfavorable. Alternatively, the agent may run out of atoms to place when an additional atom(s) would result in a more optimal molecule. In either case, the pre-specified atom bag unduly constrains the agent.

The principal contribution of this work is enhancing MolGym to support an 'infinite' bag of atoms. Under the infinite bag paradigm, we enable MolGym agents to learn the size and composition of optimal molecular structures. Practitioners no longer specify

hard constraints on the size and makeup of molecules but soft atom penalties to guide, but not force, certain molecular forms.

In Chapter 2 (Background), we provide a review of the key concepts necessary to understand the technical details of this dissertation. Then in chapter 3 (Infinite Atom Bags), we elaborate on the infinite bag paradigm and detail its implementation into MolGym. We also discuss how practitioners might formulate atom penalties and present a modification to the MolGym training algorithm that improves robustness. Next in chapter 4 (Randomized Atom Penalties), we motivate randomized atom penalties as a means to train more flexible and powerful MolGym agents. We then detail how randomized atom penalties are carefully integrated into MolGym by revising our notion of full state. We proceed in chapter 5 (Scaling to Complex Structures) to detail optimization challenges in training agents to construct more complex molecules. We propose strategies to improve optimization, including an imitation learning method that exploits expert demonstrations. Finally, we close in chapter 6 (Conclusion) with a discussion on future research directions.

Chapter 2

Background

In this chapter we detail the necessary technical background. First, in section 2.1, we provide a review of basic chemistry concepts central to our molecular design objectives. Next, in section 2.2, we outline various machine learning approaches to molecular design including the framework adopted in this work: MolGym (Simm, Pinsler, and Hernández-Lobato, 2020; Simm, Pinsler, Csányi, et al., 2021). In section, 2.3 we describe Reinforcement Learning, Markov Decision Processes (MDPs), and the specific MDP formulated by MolGym3. We proceed to detail how the MolGym3 MDP is modeled using a neural network in section 2.4. Then in section 2.5 we discuss the neural network's training procedure, the soft-actor Critic (SAC) (Haarnoja, Tang, et al., 2017), and a particular, unusual implementation of SAC permitted under the assumptions of MolGym3.

2.1 Chemistry Fundamentals

In this work, we tackle the problem of generating useful molecules. Although a comprehensive chemistry overview is well beyond our scope, it is worthwhile to briefly review the basics of what molecules are and how we might define *useful* molecules.

The world is composed of tiny bit of matter called *atoms*. Atoms are themselves composed of several subatomic particles: *protons* (positively charged) and *neutrons* (neutrally charged) comprise an atom's nucleus while *electrons* (negatively charged) orbit the nucleus. We commonly group atoms according to their number of protons to define *elements* with different properties e.g., oxygen (8 protons) or carbon (4 protons).

Molecules are groups of atoms held together by chemical bonds. There are many sorts of chemical bonds (e.g., covalent bonds, hydogen bonds,...) and they vary in their mechanisms of interaction and strength. Although a full treatment of bonding is

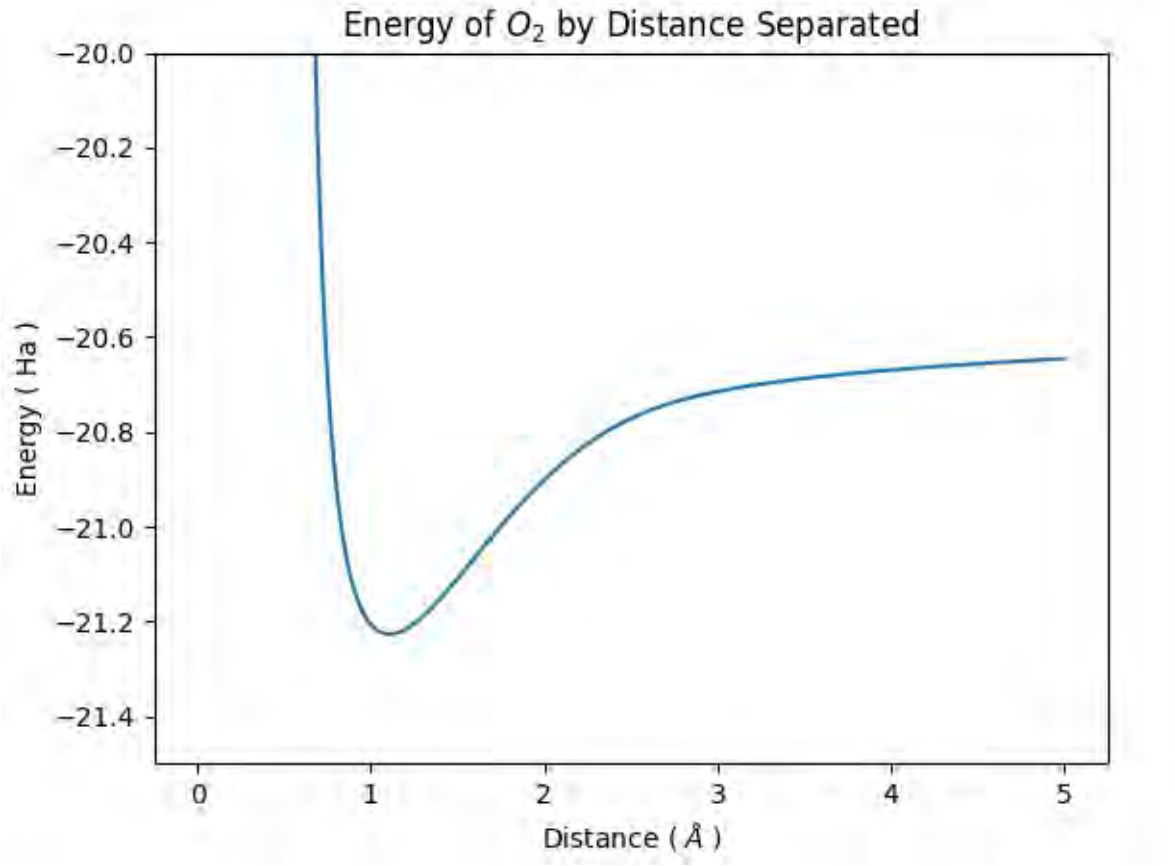


Fig. 2.1 The energy curve for molecular oxygen shows a minimum energy is obtained when the atoms are at a (inter-nuclear) distance slightly over 1 Angstrom.

out-of-scope, we note that atoms can obtain a lower energy state by forming bonds. For example, in figure 2.1 we plot molecular oxygen O_2 which consists of two covalently bonded oxygen atoms.[1] When the oxygen atoms are very close (< 1Å) the positively charged nuclei strongly repel one another and the energy of system spikes. On the other hand, when the oxygen atoms are far apart the energy of the system plateaus to that of two independent oxygen atoms. But at middle distances (≈ 1Å), the oxygen atoms form two covalent bonds in order to fill their outer (valence) electron shells and the energy of the system obtains a minimum.

In designing *useful* molecules, there are myriad desirable properties one might seek depending on the precise application. In this work, we focus on designing stable

[1]This and subsequent energy plots were obtained through the semi-empirical Parametrized Method 6 (PM6) (Stewart, 2007) with the software package Sparrow (Husch, Vaucher, and Reiher, 2018; Bosia et al., 2021).

molecules. In particular, as lower energy molecules are more stable we aim to generate molecules with low energy. Later, we will detail precisely how we formulate the objective of generating low energy molecules as a reinforcement learning problem. But first, we will provide a broad overview of how researchers have approached molecule design with machine learning.

2.2 ML Approaches to Molecular Design

The vast search space of potential molecules has motivated a variety of ML approaches to molecular design. In this section, we briefly review key approaches and their limitations before introducing the MolGym deep reinforcement learning framework.

Researchers have developed a variety of supervised generative models for synthesizing molecules (Gómez-Bombarelli et al., 2018; De Cao and Thomas, 2018). However, such approaches rely on large datasets to adequately explore uncharted chemical space. Alternatively, reinforcement learning (RL) methods only require an appropriate reward function to generate molecules (Olivecrona et al., 2017; Simm, Pinsler, and Hernández-Lobato, 2020). A considerable recent body of work has emerged using RL operating on graph-based molecular representations (Z. Zhou et al., 2019; You et al., 2018). Yet, graph-based representations are fundamentally limited as they fail to represent 3D information. In particular, graph-based representations constrain generated molecules to a small chemical space, prevent the use of quantum-mechanical reward functions, and make it difficult to encode geometric constraints into the design (Simm, Pinsler, Csányi, et al., 2021). Instead we will build on an RL approach to generating molecules where atoms are represented in 3D Cartesian coordinate: MolGym (Simm, Pinsler, and Hernández-Lobato, 2020).

2.2.1 MolGym

MolGym uses deep reinforcement learning to train agents that build 3D molecular structures (Simm, Pinsler, and Hernández-Lobato, 2020). Specifically, a MolGym agent learns to assemble stable, low energy molecules through an auto-regressive policy for sequentially placing atoms on a 3D canvas. An updated MolGym (Simm, Pinsler, Csányi, et al., 2021) henceforth referred to as MolGym2 improved capabilities around constructing highly symmetric molecules by employing the rotationally covariant neural network architecture Cormorant (Anderson, Hy, and Kondor, 2019). In this work we will enhance the latest and presently unpublished MolGym3. MolGym3 departs from

earlier versions by using an off-policy training algorithm, namely the soft actor-critic (Haarnoja, Tang, et al., 2017). MolGym3 also adopts the E3nn library to model molecular rotational covariances/invariances (Geiger et al., 2022) and opts for a sparse reward function. For clarity going forwards, we will refer to MolGym when discussing a property consistent across all versions and otherwise explicitly name either MolGym1, MolGym2, or MolGym3.

2.3 RL and Markov Decision Processes

We now detail reinforcement learning (RL), introduce Markov decision processes (MDP), and elaborate on the specifics of our MDP setup for molecular design with MolGym.

2.3.1 Reinforcement Learning Preliminaries

Reinforcement learning concerns training an agent that interacts with an environment to maximize some reward (Sutton and Barto, 2018). We formalize the reinforcement learning environment as a Markov Decision Process $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{T}, \mu_0, \gamma, r)$ with state space $\mathcal{S}$, action space $\mathcal{A}$, transition dynamics $\mathcal{T} : \mathcal{S} \times \mathcal{A} \mapsto S$, discount factor γ, an initial state distribution μ_0, and a reward function $r : \mathcal{S} \times \mathcal{A} \mapsto S$.[2] We call such a process *Markov* because the probabilities of future states are conditionally independent of past states given the present state. We next define a policy, essentially an agent's strategy, as a potentially stochastic mapping from states to actions $\pi(a_t|s_t)$ where $s_t \in \mathcal{S}, a_t \in \mathcal{A}$. We can then define the value of a particular state s_t under policy π as the expected discounted future reward beginning in state s_t and henceforth following policy π:

$$V^{\pi}(s_t) = \mathbb{E}_{\tau\sim\pi}[\sum_{t'=t} \gamma^{t'-t} r(s_{t'}, a_{t'})|s_t] \tag{2.1}$$

Note we use τ to denote a trajectory sampled from policy π consisting of successive state-action pairs $(s_{t'}, a_{t'})$.

Ultimately, the RL objective is to determine a policy π that maximizes the expected discounted future reward from the initial state distribution. That is, the value V^{π} of the initial state distribution:

$$J(\pi) = \mathbb{E}_{s_0\sim\mu_0}[V^{\pi}(s_0)] = \mathbb{E}_{\tau\sim\pi, s_0\sim\mu_0}[\sum_{t'=0} \gamma^{t'} r(s_{t'}, a_{t'})] \tag{2.2}$$

[2]Note that the reward function $r(s_t, a_t)$ defines the reward for transitioning *from* state s_t by taking an action a_t.

2.3.2 MolGym3 MDP Specification

Next, we present the MDP specification for MolGym3.

A state $s_t = (\mathcal{C}_t, \beta_t)$ comprises a 3D canvas $\mathcal{C}_t$ and bag of atoms β_t. The canvas $\mathcal{C}_t = \mathcal{C}_0 \cup \{(e_i, x_i)\}_{i=0}^{t-1}$ consists of the initial set of atoms $\mathcal{C}_0$ and all atoms placed up until episode time instance $t-1$. Each placed atom is represented by its element identity $e_i \in \{H, O, ...\}$ and its position in 3D Cartesian coordinates $x_i \in \mathbb{R}^3$. The initial set of atoms C_0 may be defined or empty. The bag $\beta_t = \{(e, m(e))\}$ is a multiset of to-be-placed atoms with $m(e)$ denoting the multiplicity of element e.

An action $a_t = (e_t, x_t)$ consists of placing an element $e_t \in \beta_t$ from the bag at position $x_t \in \mathbb{R}^3$ on the canvas. We thus define a deterministic transition function $\mathcal{T}(s_t, a_t) = s_{t+1} = (\mathcal{C}_{t+1}, \beta_{t+1})$ that returns an updated canvas with atom (e_t, x_t) added and an updated bag with one e_t atom removed $\beta_{t+1} = \beta_t / e_t$. We emphasize that deterministic transition dynamics do *not* imply our policy is deterministic. At a particular state s_t, the action a_t is chosen stochastically according to $a_t \sim \pi(\cdot|s_t)$. However, having chosen an action $a_t \sim \pi(\cdot|s_t)$ then the next state s_{t+1} is defined deterministically according to $s_{t+1} = \mathcal{T}(s_t, a_t)$.

In Molgym1/2, the reward function $r(s_t, a_t)$ is defined as the negative energy difference between the updated canvas C_{t+1} and the previous canvas C_t plus the element e_t (the atom just placed) set at the origin of an empty canvas. This reward motivates the agent to place atoms that form stable, low energy structures. It also avoids biasing the agent to just select atoms with lower intrinsic energies. MolGym3 defines a similarly motivated reward, but employs it in a sparse setting. That is, an agent receives 0 reward at all but the terminal state, where then the reward is given by the negative energy difference between the terminal canvas C_T and the sum of energies of all constituent atoms of C_T (when individually placed at the origin of an empty canvas):

$$r(s_t, a_t) = \begin{cases} -\left[E(C_T) - \sum_{i=0}^{T} E(e_i, [0,0,0]^T)\right], & \mathcal{T}(s_t, a_t) \in \mathcal{S}_{terminal} \\ 0, \text{otherwise} \end{cases} \tag{2.3}$$

The electronic energy E is calculated via the semi-empirical Parametrized Method 6 (PM6) (Stewart, 2007) in the software package Sparrow (Husch, Vaucher, and Reiher, 2018; Bosia et al., 2021).

Finally, we also assume the MolGym3 reward is path-independent. Taken together with deterministic transition dynamics, we may implement the reward function $r(s_t, a_t)$ as a function of the next state $\rho(s_{t+1})$ with $\rho(s_{t+1}) = \rho(\mathcal{T}(s_t, a_t)) = r(s_t, a_t)$.

With the MolGym3 MDP defined, we will proceed to detail how an agent decides both 1) what atom to place next and 2) where to place it.

2.4 The MolGym3 Model

In this section, we motivate and describe the policy π of a MolGym agent. The policy $\pi(a_t|s_t)$ can be thought of as probability function that returns the probability of an action a_t given the agent is in state s_t. We choose to parameterize $\pi_\theta(a_t|s_t)$ by a neural network (with parameters θ) as the policy operates on an infinite state space $\mathcal{S}$ and action space $\mathcal{A}$ and should learn a complex, nonlinear mapping. The network π_θ should also respect molecular symmetries to enable efficient policy learning.

2.4.1 Covariant Neural Networks

We desire a policy $\pi_\theta(a_t|s_t)$ that is covariant to rotation and translation. That is, if we rotate or translate the 3D canvas C_t, then the position of the next placed atom x_t (recall $a_t = (e_t, x_t)$) should be rotated or translated accordingly (Simm, Pinsler, and Hernández-Lobato, 2020). This is because the energy of a molecule and a rotated/translated copy are equivalent. To learn data-efficiently it is important to ensure these symmetries in the neural network architecture. Specifically, MolGym3 devises a covariant network architecture using the PyTorch framework e3nn (Geiger et al., 2022). In e3nn, the scalar operations in a conventional neural network are replaced with generalized tensor equivalents. Input features are tagged by how they transform under rotation/parity and outputs must possess equal or higher symmetry (Curie, 1894).

2.4.2 Action Selection

We now outline how an action, i.e., the next atom's identity and position, is chosen under a policy π_θ in MolGym3.

First, the policy model constructs a representation of the current canvas that resembles an enhanced graph. Specifically, we represent each canvas atom (e_i, x_i) as a node and draw edges between all atoms within some predefined distance. Crucially, we also encode the 3D position of each atom into the representation (in this way the

approach of MolGym differs from purely graph based molecular representations e.g., Z. Zhou et al., 2019; You et al., 2018). The canvas graph representation is then passed through an embedding network that outputs a covariant embedding s_{emb_i} for each current canvas atom i that captures interactions with neighboring atoms. We then enhance these embeddings with knowledge of the bag, through an additional geometric tensor product, to compute covariant embeddings s_{cov_i}. We additionally generate an invariant representation s_{inv_i} of each current canvas atom by norming the geometric tensors in the covariant representation. We shall later see that some operations in the policy model should be invariant.

Using these representations, the policy model determines the next atom by sequentially selecting an existing canvas focal atom f_t, the identity of the next placed atom e_t, the distance the next atom will lie from the focal atom d_t, and finally the orientation $\tilde{x}_t$. The policy can thus be decomposed as:

$$\pi(a_t|s_t) = \pi(\tilde{x}_t, d_t, e_t, f_t|s_t) = p(\tilde{x}_t|d_t, e_t, f_t, s_t)p(d_t|e_t, f_t, s_t)p(e_t|f_t, s_t)p(f_t|s_t) \quad (2.4)$$

$\mathbf{p(f_t|s_t)}$: We initially select a focal atom f_t from those currently on the canvas. The focal atom serves as a local reference near which we will place the next atom. We choose the focal atom through a multi-layer perceptron MLP_f that takes each invariant representation s_{inv_i} as input (as the focal atom choice should be *invariant* to rotation/translation). The output layer of MLP_f yields a scalar representing the unnormalized probability of selecting a current canvas atom as the focal atom. By batching s_{inv} we can then compute a softmax distribution over all canvas atoms and accordingly sample f_t.[3]

$\mathbf{p(e_t|f_t, s_t)}$: Next, we select the element type for the next atom e_t through another multi-layer perceptron MLP_e which takes as input the invariant representation of the focal atom s_{inv_f}. As with the focal atom, the selection of e_t should be *invariant* to rotation/translation. The output layer of MLP_e then consists of a softmax over all elements with nonzero multiplicity in the bag. Finally, the output softmax probabilities $MLP_e(s_{inv_f})$ are used to parameterize a categorical distribution from which we sample e_t.

$\mathbf{p(d_t|e_t, f_t, s_t)}$: We now compute the distance d_t that the next atom e_t will lie from the focal atom f_t using a mixture density network (Bishop, 1994). Specifically, we first concatenate the invariant representation of the focal atom s_{inv_f} to a one hot representation of the element e_t to form a representation that incorporates both f_t

[3]We select the origin as the focal atom f_t location if the canvas is empty

and e_t. This representation is then input into a network that predicts the means and mixing coefficients of a M component Gaussian mixture model (GMM).[4] Finally, the distance d_t is sampled from the GMM.

$\mathbf{p}(\tilde{\mathbf{x}}_\mathbf{t}|\mathbf{d_t}, \mathbf{e_t}, \mathbf{f_t}, \mathbf{s_t})$: Lastly, we compute the orientation $\tilde{x}_t$ from a spherical distribution. First the distance d_t is converted into a representation that uses spherical Bessel functions (Klicpera, Groß, and Günnemann, 2020). Using weights that are a linear function of this Bessel representation of the distance d_t, we then compute a tensor product between the covariant representation of the focal element s_{cov_f} and a one-hot encoding of the element e_t. The output tensors are interpreted as spherical harmonics and used to construct a spherical distribution from which we ultimately sample $\tilde{x}_t$ using rejection sampling. Note, we use the covariant representation s_{cov_f} during this step as the orientation $\tilde{x}_t$ should be *covariant* to rotation/translation.

Additional policy model details are available in the MolGym2 paper (Simm, Pinsler, Csányi, et al., 2021) though the reader should be aware that some differences exist with the updated MolGym3 implementation adopted in this work.

To actually learn π_θ we perform a 2-step training iteration: First, we collect episode rollouts according to the current policy and add the rollouts, split into state-action-reward-state transition tuples (s_t, a_t, r_t, s_{t+1}), to a training buffer.[5] Then, we update the policy following the soft-actor critic algorithm - detailed next.

2.5 Training

In this section, we describe how MolGym3 trains a model to learn a desirable molecule generation policy. We will provide background on the state-action value function Q and then motivate and explain the soft-actor critic (Haarnoja, Tang, et al., 2017).

2.5.1 State-Action Value Function Q

We first introduce the state-action value function $Q^\pi(s_t, a_t)$ which defines the expected discounted future reward of being in a particular state s_t, taking action a_t, and thereafter following policy π:

[4] The standard deviations are treated as global network parameters and thus not output by the mixture density network.

[5] To increase sample efficiency, we also generate additional 'artificial' rollouts from each true rollout. Specifically, we randomize the order of atom placement to form supplementary rollouts that assemble an identical molecule.

$$Q^{\pi}(s_t, a_t) = r(s_t, a_t) + \mathbb{E}_{\tau\sim\pi}[\sum_{t'=t+1} \gamma^{t'-t} r(s_{t'}, a_{t'})|s_t, a_t] \tag{2.5}$$

It is similar to the value function (Equation 2.1) but the initial action a_t need not be chosen according to policy π. As such, Q-functions are prominent fixtures in many off-policy reinforcement learning approaches (e.g., Mnih et al., 2013).

In our undiscounted and finite-horizon environment the Q-function becomes:

$$Q^{\pi}(s_t, a_t) = r(s_t, a_t) + \mathbb{E}_{\tau\sim\pi}[\sum_{t'=t+1}^{T} r(s_{t'}, a_{t'})|s_t, a_t] \tag{2.6}$$

The Q-function can also be rewritten in terms of the value function to emphasize that policy π has no bearing on the immediate reward $r(s_t, a_t)$ nor next state $s_{t+1} \sim \mathcal{T}(s_t, a_t)$.

$$Q^{\pi}(s_t, a_t) = r(s_t, a_t) + V^{\pi}(s_{t+1}) \tag{2.7}$$

For notational simplicity going forwards we will omit the policy superscript π but it should be implied that both the state value function V and state-action value function Q are defined in terms of a policy.

Since the transition dynamics $\mathcal{T}$ of our environment are deterministic we can substitute $\mathcal{T}(s_t, a_t) = s_{t+1}$ into the Q-function as:

$$Q(s_t, a_t) = r(s_t, a_t) + V(\mathcal{T}(s_t, a_t)) \tag{2.8}$$

Then recalling our path independent reward assumption $r(s, a) = \rho(\mathcal{T}(s, a))$ we can reformulate our Q-function as entirely determined by the next state as opposed to the current state and action. That is, the Q-function comes to closely resemble a value function that operates on the next state but subsumes the current reward.[6]

$$Q(s_t, a_t) = \rho(\mathcal{T}(s_t, a_t)) + V(\mathcal{T}(s_t, a_t)) = Q(\mathcal{T}(s_t, a_t)) = Q(s_{t+1}) \tag{2.9}$$

Going forwards we will occasionally still refer to Q-functions in MolGym3 as $Q(s_t, a_t)$ for clarity with the wider literature. However, we emphasize that Q-functions are specially implemented in MolGym3 to (by our environment assumptions) take only the next state as input. When this latter formulation is vital to understanding, we will present Q-functions as functions of the next state $Q(s_{t+1})$.

[6]The critical difference between V and Q in our sparse reward setting is that at a terminal state s_T, $Q(s_T)$ may have non-zero reward while $V(s_T) = 0$.

Just as we parameterize a policy π_θ by a neural net to represent complex high-dimensional (or even infinite) state and action spaces, it is common to parameterize action-value functions Q_ϕ with neural nets (Schulman et al., 2015; Mnih et al., 2013). In MolGym3, each network[7] Q_ϕ takes the next state s_{t+1} as input and first computes a rotationally covariant embedding s_{emb_i} for each current canvas atom i that captures interactions with nearby atoms. Although this step resembles the policy network π_θ, the computations now diverge. The Q_ϕ network, unlike the policy network, proceeds to (for now) ignore the bag as it is designed to first compute a representation of the present canvas configuration and then consider future additions. In detail, the covariant canvas atom embeddings s_{emb_i} are first converted into invariant representations through a norming operation. Each invariant canvas atom representation is then separately passed through an MLP and the outputs summed to produce a vector representation for the canvas. Lastly, this vector canvas representation is concatenated with the bag and the result is passed through a final MLP that returns the scalar state-action value output.

With this understanding of Q networks, we may now describe the soft actor-critic (SAC) algorithm that MolGym3 uses to train agents.

2.5.2 Soft-Actor Critic

The soft-actor critic (SAC) is an algorithm for training a stochastic, deep neural network parameterized RL agent that combines an off-policy approach, entropy maximization, and an actor-critic architecture (Haarnoja, A. Zhou, Abbeel, et al., 2018). Let's examine each design aspect in detail.

Off Policy RL

Off-policy RL algorithms permit the agent to learn a target policy that differs from the policy used to generate the data. This allows for far greater sample efficiency than on-policy approaches as off-policy algorithms may reuse previously collected data. In MolGym3, for instance, the model may train and continue to learn from older molecule rollouts. However, off-policy algorithms present stability and convergence challenges when combined with high-dimension nonlinear function approximators e.g., neural networks (Maei et al., 2009; Tsitsiklis and Van Roy, 1996).

[7] We will discuss the need for multiple Q networks forthcoming

Maximum Entropy RL

The soft actor-critic algorithm achieves improved stability compared to previous popular off-policy approaches (Lillicrap et al., 2015) by learning a stochastic policy with entropy regularization. That is, an agent seeks to balance maximizing expected future reward against acting more randomly i.e., exploring. This means reformulating the RL objective (Equation 2.2) with an entropy regularization term:

$$J(\pi) = \mathbb{E}_{\tau\sim\pi, s_0\sim\mu_0}[\sum_{t'=0}^{T} r(s_{t'}, a_{t'}) + \alpha\mathcal{H}(\pi(\cdot|s_{t'}))] \tag{2.10}$$

where $\mathcal{H}$ gives the entropy over the distribution $\mathcal{H}(\pi(\cdot|s_{t'})$ and α is a temperature hyperparameter that balances the relative importance of maximizing entropy and maximizing the episode reward. In particular, increasing α will favor a more stochastic policy. Note in Equation 2.10 that we've also dropped the discount factor from Equation 2.2 as we're assuming an undiscounted setting and that the the episode terminates at time T.[8]

As we parameterize the policy by neural net parameters θ, the RL objective becomes:

$$J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta, s_0\sim\mu_0}[\sum_{t'=0}^{T} r(s_{t'}, a_{t'}) + \alpha\mathcal{H}(\pi_\theta(\cdot|s_{t'}))] \tag{2.11}$$

To then learn an optimal parameterized policy π_θ, SAC employs an actor-critic architecture.

Actor Critic Architecture

In an actor-critic architecture, a *critic* is used to assist an *actor* in learning the optimal policy (Konda and Tsitsiklis, 1999). Actor-critic approaches to reinforcement learning may help reduce variance and deliver faster convergence (Konda and Tsitsiklis, 1999). In the soft-actor critic a parameterized stochastic policy $\pi_\theta(a_t|s_t)$ (the actor) is learned with the aid of a parameterized soft Q-function $Q_\phi(s_t, a_t)$ (the critic) that approximates state-action values under policy π_θ. The Q-function is termed *soft* because it approximates not only the future rewards following policy π_θ but also the entropy maximization terms:

[8] Although a maximum entropy objective can be formulated in a discounted setting, it is considerably more involved and seperate from our aims. Nonetheless, details may be found in Appendix A of Haarnoja, A. Zhou, Abbeel, et al., 2018.

$$Q_\phi(s_t, a_t) \approx Q^{\pi_\theta}(s_t, a_t) = r(s_t, a_t) + \mathbb{E}_{\tau\sim\pi_\theta}[\sum_{t'=t+1}^{T} r(s_{t'}, a_{t'}) + \alpha\mathcal{H}(\pi_\theta(\cdot|s_{t'})\,|s_t, a_t] \quad (2.12)$$

With these pieces in place, we proceed to detail how we train the soft-actor critic.

Optimization

To find an optimal policy, the soft actor-critic algorithm alternates between a policy evaluation and a policy improvement step.

In the *policy evaluation* step, the Q-function parameters are updated via stochastic gradient descent to minimize the Bellman residual. To explain this, first note that we can recursively estimate Q_ϕ using a target network $Q_{\phi'}$ that is computed as an exponential moving average (to aid training stability) of recent weights Q_ϕ:

$$\hat{Q}_\phi(s_t, a_t) = r(s_t, a_t) + \mathbb{E}_{a_{t+1}\sim\pi_\theta(\cdot|s_{t+1})}[Q_{\phi'}(s_{t+1}, a_{t+1})] + \alpha\mathcal{H}(\pi_\theta(\cdot|s_{t+1})) \quad (2.13)$$

where s_{t+1} is given by the deterministic transition dynamics $\mathcal{T}(s_t, a_t) = s_{t+1}$.

Then by reframing the entire expression as an expectation over the next action a_{t+1} according to the policy π_θ we can substitute in the definition of entropy (equation 2.14) and obtain equation 2.15.

$$\mathcal{H}(\pi_\theta(\cdot|s_{t+1}))) = \mathbb{E}_{a_{t+1}\sim\pi_\theta(\cdot|s_{t+1})}\left[-\log(\pi_\theta(a_{t+1}|s_{t+1}))\right] \quad (2.14)$$

$$\hat{Q}_\phi(s_t, a_t) = \mathbb{E}_{a_{t+1}\sim\pi_\theta(\cdot|s_{t+1})}\left[r(s_t, a_t) + Q_{\phi'}(s_{t+1}, a_{t+1}) - \alpha\log(\pi_\theta(a_{t+1}|s_{t+1}))\right] \quad (2.15)$$

As discussed in section 2.5.1, MolGym3 implements Q-functions operating solely on the state and that may be derived as $Q(s_t, a_t) = Q(\mathcal{T}(s_t, a_t)) = Q(s_{t+1})$. Thus we may re-express equation 2.15 as a backup estimator B that instead takes the next state s_{t+1} and transition reward r_t as input:

$$B(r_t, s_{t+1}) = \begin{cases} r_t, & \text{if } s_{t+1} \text{ terminal} \\ \mathbb{E}_{a_{t+1}\sim\pi_\theta(\cdot|s_{t+1})}\left[r_t + Q_{\phi'}(s_{t+2}) - \alpha\log(\pi_\theta(a_{t+1}|s_{t+1}))\right], & \text{otherwise} \end{cases} \quad (2.16)$$

We then train our neural network Q_ϕ to minimize the squared residual error with the backup B estimated through the target network Q'_ϕ. We label this the policy evaluation loss $J_{eval}(\phi)$ and compute it as an expectation over previously sampled states and actions from a rollout data buffer D:

$$J_{eval}(\phi) = \mathbb{E}_{(s_t,a_t,r_t,s_{t+1})\sim D}\left[\left(Q_\phi(s_{t+1}) - B(r_t, s_{t+1})\right)^2\right] \tag{2.17}$$

Note that we sample the current state s_t and action a_t as well as the next state s_{t+1} from previously collected rollout data. However, the next action a_{t+1} in the backup term B is sampled from the current policy and may differ from that collected during the rollout.

In practice, we maintain two separate Q-networks Q_{ϕ_1}, Q_{ϕ_2} and two separate target networks $Q_{\phi'_1}$, $Q_{\phi'_2}$, using the minimum target network estimate on each sample, $\min(Q_{\phi'_1}(s_{t+1}), Q_{\phi'_2}(s_{t+1}))$, to compute the backup B. This mitigates overoptimism and empirically boosts SAC performance (Haarnoja, A. Zhou, Abbeel, et al., 2018). Thus the policy evaluation loss becomes:

$$\begin{aligned} J_{eval}(\phi) = \mathbb{E}_{(s_t,a_t,r_t,s_{t+1})\sim D}&\left[\left(Q_{\phi_1}(s_{t+1}) - \min(B_1(s_{t+1}, r_t), B_2(s_{t+1}, r_t))\right)^2\right] \\ &+ \left[\left(Q_{\phi_2}(s_{t+1}) - \min(B_1(s_{t+1}, r_t), B_2(s_{t+1}, r_t))\right)^2\right] \end{aligned} \tag{2.18}$$

where the B index refers to the target Q-network used to compute the backup e.g.,

$$B_2(s_{t+1}, r_t) = \begin{cases} r_t, & \text{if } s_{t+1} \text{ terminal} \\ \mathbb{E}_{a_{t+1}\sim\pi_\theta(\cdot|s_{t+1})}\left[r_t + Q_{\phi'_2}(s_{t+2}) - \alpha\log(\pi_\theta(a_{t+1}|s_{t+1}))\right], & \text{otherwise} \end{cases} \tag{2.19}$$

In the *policy improvement* step, we update the policy by minimizing the expected Kullbach-Leibler Divergence D_{KL} (Kullback and Leibler, 1951) between the policy π_θ and the exponential of the updated soft Q-function Q_ϕ.[9]

$$J_{imprv}(\theta) = \mathbb{E}_{s_t\sim\mathcal{D}}\left[D_{KL}\left(\pi_\theta(\cdot|s_t)\middle\|\frac{\exp(\frac{1}{\alpha}Q_\phi(s_t,\cdot))}{Z_\phi(s_t)}\right)\right] \tag{2.20}$$

where $Z_\phi(s_t)$ is a normalization factor that we will hereafter drop as the minimize D_{KL} objective results in the same solution for an unnormalized distribution. Addition-

[9]The SAC authors prove that this particular formulation ensures the new policy is superior to the old policy (Haarnoja, A. Zhou, Abbeel, et al., 2018), albeit only for finite action spaces which is not the case for MolGym.

ally, bear in mind that the expectation is computed over states s_t that are collected from episode rollouts and stored in a replay buffer $\mathcal{D}$.

The temperature hyperparameter α may be tuned to adjust the stochasticity of the learned policy. In the SAC paper's derivations, the temperature hyperparameter α is subsumed into the reward by scaling by its inverse α^{-1}. We emphasize that this is not because the temperature hyperparameter is unimportant but because the authors choose to tune the reward scale (effectively the inverse temperature) instead. In fact, the SAC authors note that the algorithm is very sensitive to the reward scale / temperature and it was the only hyperparameter that required tuning. Although we have some flexibility to adjust rewards in MolGym (more discussion forthcoming), the reward scale is largely set by the energy of the molecular system. Thus, unlike in the SAC paper we will need to carefully tune the temperature parameter α, as opposed to the reward scale, and so we explicitly include α through the equations that follow.

By the definition of KL Divergence we can express the loss function as:

$$J_{imprv}(\theta) = \mathbb{E}_{s_t \sim \mathcal{D},\, a_t \sim \pi_\theta(\cdot|s_t)} \left[\log \pi_\theta(a_t|s_t) - \frac{1}{\alpha} Q_\phi(s_t, a_t) \right] \tag{2.21}$$

Our objective of minimizing $J_{imprv}(\theta)$ is unchanged if we multiply through by α and thus we may rewrite $J_{imprv}(\theta)$ as:

$$J_{imprv}(\theta) = \mathbb{E}_{s_t \sim \mathcal{D},\, a_t \sim \pi_\theta(\cdot|s_t)} \left[\alpha \log \pi_\theta(a_t|s_t) - Q_\phi(s_t, a_t) \right] \tag{2.22}$$

This expectation involves sampling actions from the policy π_θ and thus problematically requires taking gradients through the samples when optimizing the parameters θ through gradient descent.

$$\nabla_\theta J_{imprv}(\theta) = \nabla_\theta \mathbb{E}_{s_t \sim \mathcal{D},\, a_t \sim \pi_\theta(\cdot|s_t)} \left[\alpha \log \pi_\theta(a_t|s_t) - Q_\phi(s_t, a_t) \right] \tag{2.23}$$

To resolve this, we derive a gradient update based on the REINFORCE gradient estimator (Williams, 1992). First, we break the expectation in equation in 2.22 into separate expectations over states $\mathbb{E}_{s_t \sim \mathcal{D}}$ and actions $\mathbb{E}_{a_t \sim \pi_\theta(\cdot|s_t)}$:

$$J_{imprv}(\theta) = \mathbb{E}_{s_t \sim \mathcal{D}} \left[\alpha \mathbb{E}_{a_t \sim \pi_\theta(\cdot|s_t)} \left[\log \pi_\theta(a_t|s_t) \right] - \mathbb{E}_{a_t \sim \pi_\theta(\cdot|s_t)} \left[Q_\phi(s_t, a_t) \right] \right] \tag{2.24}$$

Taking the gradient:

$$\nabla_\theta J_{imprv}(\theta) = \mathbb{E}_{s_t \sim \mathcal{D}} \left[\alpha \nabla_\theta \mathbb{E}_{a_t \sim \pi_\theta(\cdot|s_t)} \left[\log \pi_\theta(a_t|s_t) \right] - \nabla_\theta \mathbb{E}_{a_t \sim \pi_\theta(\cdot|s_t)} \left[Q_\phi(s_t, a_t) \right] \right] \quad (2.25)$$

We may rectify the first term in equation 2.25 by re-expressing the term as an integral, applying the product derivative rule, exploiting the log-derivative trick, and finally rewriting it back as an expectation:

$$\begin{aligned}
&\alpha \nabla_\theta \mathbb{E}_{a_t \sim \pi_\theta(\cdot|s_t)} [\log \pi_\theta(a_t|s_t)] && (2.26)\\
&= \alpha \int \nabla_\theta [\pi_\theta(a_t|s_t) \log \pi_\theta(a_t|s_t)] \; da_t && (2.27)\\
&= \alpha \int \log \pi_\theta(a_t|s_t) \nabla_\theta \pi_\theta(a_t|s_t) + \pi_\theta(a_t|s_t) \nabla_\theta \log \pi_\theta(a_t|s_t) \; da_t && (2.28)\\
&= \alpha \int \log \pi_\theta(a_t|s_t) \pi_\theta(a_t|s_t) \nabla_\theta \log \pi_\theta(a_t|s_t) + \pi_\theta(a_t|s_t) \nabla_\theta \log \pi_\theta(a_t|s_t) \; da_t && (2.29)\\
&= \alpha \int \pi_\theta(a_t|s_t) \left[(\log \pi_\theta(a_t|s_t) + 1) \nabla_\theta \log \pi_\theta(a_t|s_t) \right] \; da_t && (2.30)\\
&= \alpha \mathbb{E}_{a_t \sim \pi_\theta(\cdot|s_t)} \left[(\log \pi_\theta(a_t|s_t) + 1) \nabla_\theta \log \pi_\theta(a_t|s_t) \right] && (2.31)
\end{aligned}$$

Similarly for the second term in equation 2.25:

$$\begin{aligned}
&- \nabla_\theta \mathbb{E}_{a_t \sim \pi_\theta(\cdot|s_t)} \left[Q_\phi(s_t, a_t) \right] && (2.32)\\
&= - \int \nabla_\theta [\pi_\theta(a_t|s_t) Q_\phi(s_t, a_t)] \; da_t && (2.33)\\
&= - \int \pi_\theta(a_t|s_t) \nabla_\theta \log \pi_\theta(a_t|s_t) Q_\phi(s_t, a_t) \; da_t && (2.34)\\
&= - \mathbb{E}_{a_t \sim \pi_\theta(\cdot|s_t)} \left[\nabla_\theta \log \pi_\theta(a_t|s_t) Q_\phi(s_t, a_t) \right] && (2.35)\\
& && (2.36)
\end{aligned}$$

Substituting these two pieces into equation 2.25 we obtain equation 2.37. We observe that we no longer take the gradient of an expectation over samples from π_θ and thus may optimize the policy parameters through gradient descent.

$$\begin{aligned}
\nabla_\theta J_{imprv}(\theta) = \mathbb{E}_{s_t \sim \mathcal{D}, a_t \sim \pi_\theta(\cdot|s_t)} [&\alpha (\log \pi_\theta(a_t|s_t) + 1) \nabla_\theta \log \pi_\theta(a_t|s_t) \\
&- Q_\phi(s_t, a_t) \nabla_\theta \log \pi_\theta(a_t|s_t)]
\end{aligned} \quad (2.37)$$

To reduce the variance of gradient estimates in practice we apply an advantage function $A(s_t, a_t) = Q_\phi(s_t, a_t) - V_\psi(s_t)$ in place of $Q_\phi(s_t, a_t)$. Due to the special implementation of Q-functions in MolGym3 as operating on next states (see section 2.5.1) the advantage function may employ a single network Q_ϕ:

$$A(s, a) = Q_\phi(\mathcal{T}(s_t, a_t)) - Q_\phi(s_t) = Q_\phi(s_{t+1}) - Q_\phi(s_t) \tag{2.38}$$

The gradient update is thus:

$$\begin{aligned}\nabla_\theta J_{imprv}(\theta) = \mathbb{E}_{s_t \sim \mathcal{D}, a_t \sim \pi_\theta(\cdot|s_t)}[\alpha(\log \pi_\theta(a_t|s_t) + 1)\nabla_\theta \log \pi_\theta(a_t|s_t) \\ -(Q_\phi(s_{t+1}) - Q_\phi(s_t))\nabla_\theta \log \pi_\theta(a_t|s_t)]\end{aligned} \tag{2.39}$$

Note that due to the deterministic transition dynamics we may implicitly determine s_{t+1} in equation 2.39 from s_t and a_t.

To mitigate overoptimism, each Q_ϕ in the advantage function is separately selected as the minimum evaluation of the two Q-functions Q_{ϕ_1}, Q_{ϕ_2} that we learn during policy evaluation. Our final expression for the policy improvement gradient update is thus:

$$\begin{aligned}\nabla_\theta J_{imprv}(\theta) = \mathbb{E}_{s_t \sim \mathcal{D}, a_t \sim \pi_\theta(\cdot|s_t)}[\alpha(\log \pi_\theta(a_t|s_t) + 1)\nabla_\theta \log \pi_\theta(a_t|s_t) \\ -(\min(Q_{\phi_1}(s_{t+1}), Q_{\phi_2}(s_{t+1})) - \min(Q_{\phi_1}(s_t), Q_{\phi_2}(s_t)))\nabla_\theta \log \pi_\theta(a_t|s_t)]\end{aligned} \tag{2.40}$$

We use an automatic differentiation library (PyTorch's Autograd) to optimize $J_{imprv}(\theta)$ and thus must re-express equation 2.40 as a loss function for implementation. In this loss function (equation 2.41), we carefully distinguish policy network terms through which we will compute gradients updates (π_θ) from those that we detach from the gradient computation graph (π_θ^{detach}). Specifically, we detach the policy network terms in equation 2.40 that are outside a ∇_θ gradient operator.

$$\begin{aligned}J_{imprv}(\theta) = \mathbb{E}_{s_t \sim \mathcal{D}, a_t \sim \pi_\theta^{detach}(\cdot|s_t)}[\alpha(\log \pi_\theta^{detach}(a_t|s_t) + 1) \log \pi_\theta(a_t|s_t) \\ -(\min(Q_{\phi_1}(s_{t+1}), Q_{\phi_2}(s_{t+1})) - \min(Q_{\phi_1}(s_t), Q_{\phi_2}(s_t))) \log \pi_\theta(a_t|s_t)]\end{aligned} \tag{2.41}$$

Finally, we add the policy evaluation loss $J_{eval}(\phi)$ (equation 2.18) and policy improvement loss $J_{imprv}(\theta)$ (equation 2.41) to formulate the overall loss as a function of Q network parameters ϕ and policy network parameters θ.

$$J(\theta, \phi) = J_{imprv}(\theta) + J_{eval}(\phi) \tag{2.42}$$

Importantly, we only compute gradients for parameters θ and ϕ in the expressions for $J_{imprv}(\theta)$ and $J_{eval}(\phi)$ respectively. Other modules that appear in these expressions (e.g., the Q_ϕ networks in equation 2.41) are temporarily detached from the computation graph.

As a concluding note, gradients updates in MolGym are computed using Adam with decoupled weight decay (AdamW) (Loshchilov and Hutter, 2017). The Adam optimizer (Kingma and Ba, 2014) uses per-parameter adaptive learning rates based on gradient histories and AdamW incorporates weight decay.

2.6 Background Summary

We've now introduced the assorted background (Chemistry and Molecular Modeling, Reinforcement Learning and MDPs, the soft-actor critic, and MolGym) essential to understand the ideas in this dissertation. In the next chapter, we propose a key contribution to MolGym: infinite atom bags.

Chapter 3

Infinite Atom Bags

3.1 Motivation

An important limitation of MolGym1 (Simm, Pinsler, and Hernández-Lobato, 2020) is that the agent must construct molecules from a prespecified initial atom bag. For example, to build thionyl tetrafluoride (SOF_4) the intial atom bag must consist exactly of 1 sulfur, 1 oxygen, and 4 flouride atoms. But in practice the target element multiplicities may be unknown in advance. After all, a key aim of MolGym is to help discover novel molecules.

MolGym2 (Simm, Pinsler, Csányi, et al., 2021) provides agents more flexibility by introducing a *stochastic-bag* task. This involves sampling the initial bag from a distribution over bags prior to each episode. Specifically, the initial bag $\mathcal{B}_0$ is constructed by sampling the initial bag element counts from a multinomial distribution $(m(e_1), ..., m(e_{max})) \sim Mult(\zeta, p_e)$ where $m(e_i)$ specifies the multiplicity of an element in the initial bag, p_e defines the multinomial event probabilities for selecting each element type, and ζ is the size of the bag which is itself sampled uniformly from a predefined interval $[\zeta_{min}, \zeta_{max}]$. But although MolGym2 provides some facility for constructing molecules without prespecified bags, it still effectively requires a practitioner to set a range of bags. Moreover, the range of bags is narrowly constrained by the multinomial distribution parameters. In turn, the size and composition of generated structures becomes a combination of the practitioners constrained multinomial range specification and the random initial bag draws. Critically, the agent is not able to actually learn the optimal size and composition of structures.

Instead of exactly specifying element multiplicities (or encoding element multiplicity likelihoods into a multinomial distribution) practitioners may instead prefer to specify variable per-element cost penalties. Why might this be useful? Consider training

an agent in the standard prespecified bag setting with $\{\mathcal{B}_0 = C_5H_5\}$. It may be very energetically favorable to place a sixth carbon atom to build benzene C_6H_6 and yield a higher reward. But alas there is a hard constraint on the agent that prevents placing more than five carbon atoms. In contrast, if the agent could potentially place unlimited carbon atoms (subject to some, possibly variable per-atom penalty) then the agent could *itself decide* to place a sixth carbon if the additional reward exceeded the additional penalty. This is a powerful paradigm that allows the agent to learn to build structures of different sizes and compositions. In the next section, we detail precisely how we formulate this *infinite bag* setting in MolGym.

3.2 Infinite Bag Formulation

Recall from section 2.3.2 that the bag $\beta_t = \{(e, m(e))\}$ is a multiset of to-be-placed atoms. Figure 3.1 (top) illustrates the evolution of the bag tensor over the course of constructing an H_2O molecule in the standard finite bag setting. As the molecule is built, the counts in the bag tensor decrease until there are no atoms left to place and the episode terminates. The bottom of figure 3.1 illustrates how we modify the bag tensor in the infinite bag setting. We fix the atoms remaining in the bag to 1 at every time-step (top row) so there are no longer hard restrictions on the agent selecting particular atoms. We also introduce a STOP atom Z which the agent may select to terminate the episode. Lastly, we append a row of per-atom penalties. At each timestep, the agent incurs the current penalty of the chosen atom. In this example, the oxygen penalty increases once the agent places a single oxygen atom while the hydrogen penalty increases once the agent has placed two hydrogen atoms. At this point, both hydrogen and oxygen penalties are relatively high and the agent next chooses the Z atom to terminate the episode. All accrued penalties are then added to the original energy reward (equation 2.3) at episode termination to form a new reward function:

$$r(s_t, a_t) = \begin{cases} -\left[E(C_T) - \sum_{i=0}^{T} E(e_i, [0,0,0]^T)\right] + P_{accrued}, & \mathcal{T}(s_t, a_t) \in \mathcal{S}_{terminal} \\ 0, \text{otherwise} \end{cases} \tag{3.1}$$

where C_T is the terminal molecular structure on the canvas and $P_{accrued}$ tallies all penalties accrued over the course of constructing C_T.

It is worth highlighting that the agent selects an action a_t based solely on the current bag β_t (and current canvas $\mathcal{C}_t$). This implies that the agent's policy model takes

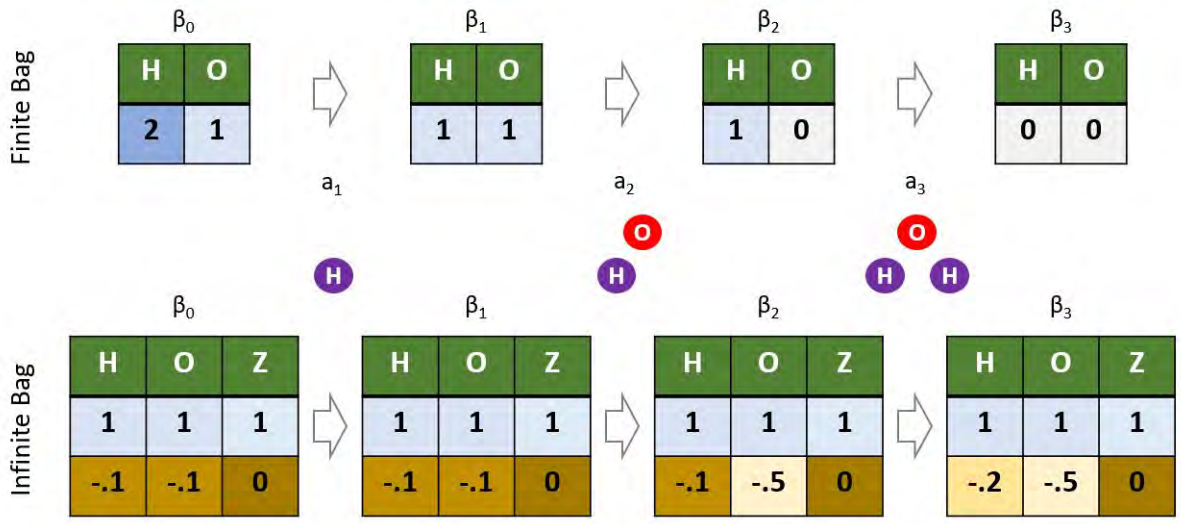


Fig. 3.1 The evolution of the bag tensor in constructing H_2O in the finite (top) and infinite (bottom) bag setting. In the finite bag setting, the bag tensor maintains counts of to-be-placed atoms. In the infinite bag setting, we fix these counts to 1 for each atom so that the agent can always choose to place any atom. Additionally, we introduce a STOP atom Z that the agent selects to terminate an episode. Lastly, we append a row of per-atom penalties that evolve over the course of an episode. In this example, the hydrogen penalty rises once the agent places 2 hydrogen atoms and the oxygen penalty rises once the agent places 1 oxygen atom. This incentivizes the agent to construct H_2O.

only the current atom penalties as input (i.e., those encoded into β_t). The agent may nonetheless learn how these penalties are expected to change through future timesteps in order to guide the current decision.

We've now described the infinite bag structure, but glossed over how the per-atom penalties are initialized and evolve. In the next section, we discuss how a practitioner might specify appropriate penalty values.

3.3 Atom Penalty Determination

Practitioners guide the infinite bag MolGym agent to construct certain structures (or families of structures) by specifying a schedule of per-atom penalties. In the preceding section, the penalty for placing hydrogen and oxygen atoms were raised after placing 2 hydrogen and 1 oxygen atom, respectively, so to motivate the agent to build water. Generally, the penalty schedule will vary considerably depending on the practitioner's objectives. Nevertheless, penalties ought to be selected to roughly match

the magnitudes of the original energy reward function for likely generated structures. This ensures neither the original energy objective nor per-atom penalties completely dominate the agent's reward function. Mathematically this means the terms $P_{accrued}$ and $-\left[E(C_T) - \sum_{i=0}^{T} E(e_i, [0,0,0]^T)\right]$ in the reward function (equation 3.1) ought to be scaled similarly.

As an example, atom penalties may be practically motivated by examining dissociation curves. In figure, 3.2 we show the dissociation curve for O_2. A system of two oxygen atoms at an optimal distance may attain a minimum energy of -21.23 Hartrees (dashed green line). If the oxygen atoms are spread far apart than the system energy amounts to the dashed purple line, equivalently the sum of energies for two isolated oxygen atoms.[1] Recall that the original reward of an episode (equation 2.3) is the negative energy difference between the terminal molecular system and the sum of energies of all constituent atoms. Thus the reward for building O_2, assuming no atom penalties, is the dip from Energy Separated (dashed purple line) to Min Energy (dashed green line). It follows that the agent should learn to build O_2 so long as the penalty for placing 2 oxygen atoms is no greater than that dip. Assuming the penalty of placing the first and second oxygen are equivalent, the agent should generally build O_2 if the per-oxygen atom penalty is no greater than half the energy dip. We caveat with *generally* because we optimize not just reward but also policy entropy and these objectives need to be properly balanced via the temperature hyperparameter. It's also usually helpful to add margin. We thus label 1/4 the dip as the 'Build Cost' as it is favorable for the agent to construct new O_2 molecules (by placing the 1st O). On the other hand, we label 3/4 the dip as the 'Cease Cost' as although it is favorable to still complete any outstanding O_2 molecule constructions, the agent should thereafter cease.

Of course it is not always so clear how to set appropriate penalties for more complex molecules. Nevertheless, examining dissociation curves can help guide the approximate scale of penalties. For instance, consider an agent that should learn to construct H_2O. By examining the dissociation curves for HO, O_2, and H_2 (figure 3.3) we can motivate a high penalty for placing a second oxygen atom since the O_2 energy dip is relatively large. Specifically, we might consider a penalty ≈ -0.5Å to discourage O_2 formation.

An idea worth emphasizing is that the infinite bag formulation of MolGym affords the practitioner some flexibility in setting atom penalties. Nonetheless, the scale of the new reward function (i.e., now incorporating atom penalties) remains largely

[1] Technically the Energy (blue line) should smoothly converge to the Energy Separated (dashed purple line). However, approximating dissociation curves precisely can be difficult for semi-empirical quantum chemical methods. Ultimately, we favor the speed of such methods and as incorrect dissociation energies typically only result in a shift in return/reward.

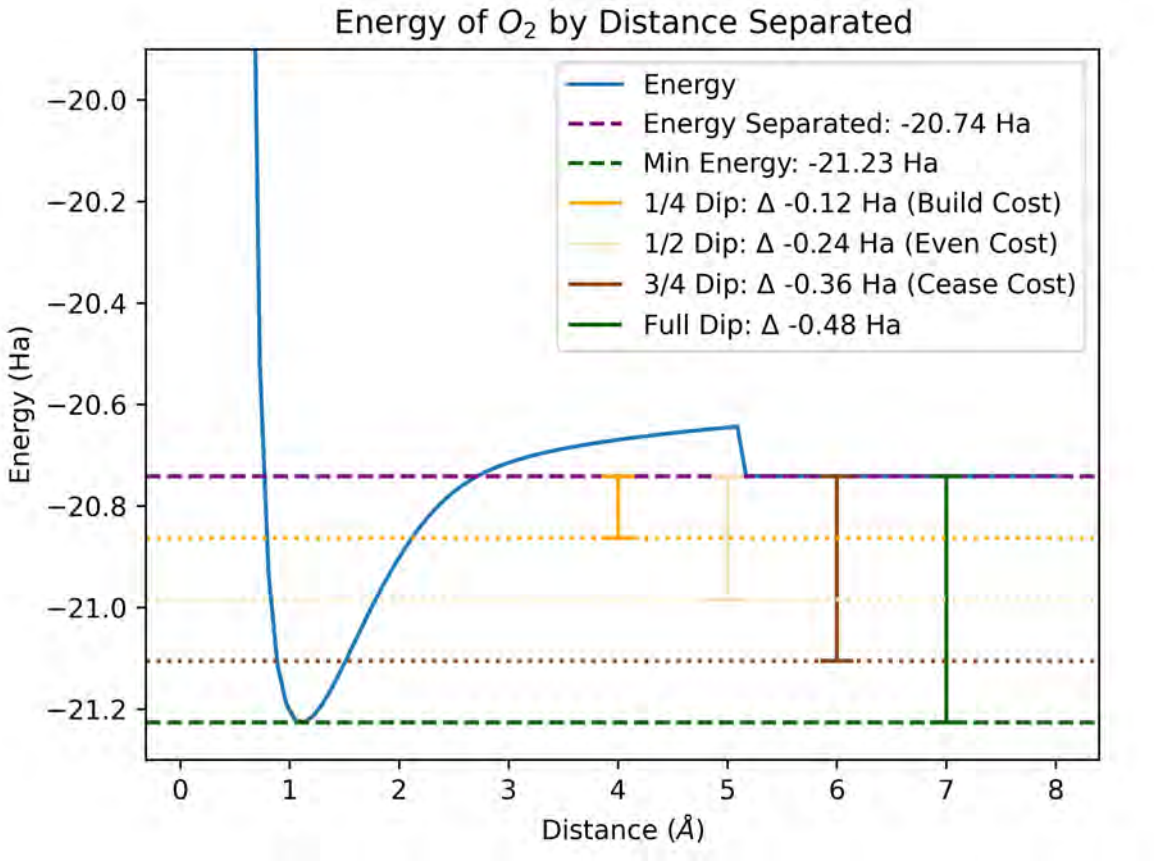


Fig. 3.2 Dissociation curve for O_2 superimposed with example construction penalties.

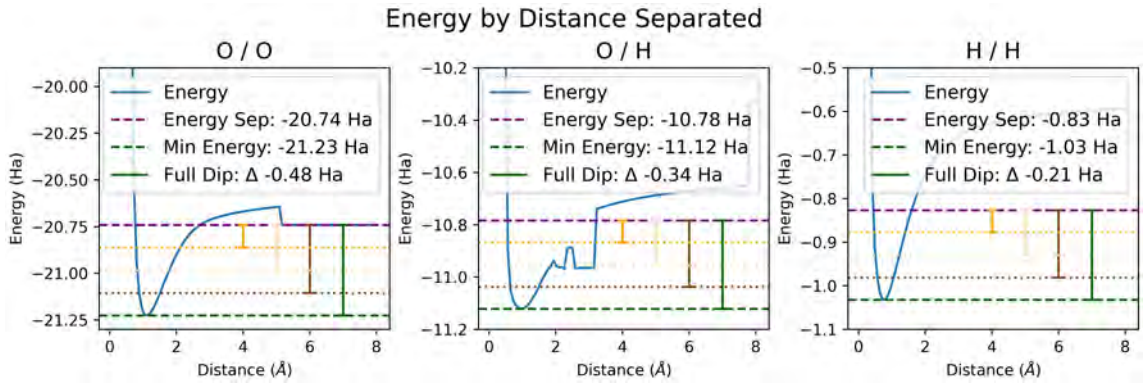


Fig. 3.3 Dissociation curves for O_2, OH, and H_2, superimposed with example penalties.

predetermined by the scale of the original energy reward function. This is particularly important because unlike in conventional RL, the optimal policy in maximum entropy RL *depends* on the reward function scale. Specifically, the reward scale changes the stochasticity of the optimal policy. This is evident in the SAC policy improvement step objective (equation 2.20 repeated below as equation 3.2) as the reward scale inevitably alters Q-function magnitudes and thus the policy stochasticity. In particular, larger reward magnitudes correspond to a more deterministic optimal policy and smaller reward magnitudes to a less deterministic optimal policy. It is then crucial to appropriately tune the temperature hyperparameter α to the reward scale as a sub-optimal temperature may induce an inappropriately stochastic policy (Haarnoja, A. Zhou, Abbeel, et al., 2018; Haarnoja, A. Zhou, Hartikainen, et al., 2018).

$$J_{imprv}(\theta) = \mathbb{E}_{s_t \sim \mathcal{D}} \left[D_{KL} \left(\pi_\theta(\cdot|s_t) \middle\| \frac{\exp(\frac{1}{\alpha} Q_\phi(s_t, \cdot))}{Z_\phi(s_t)} \right) \right] \tag{3.2}$$

3.4 Improving Robustness

3.4.1 Temperature Hyperparameter

Although the soft actor-critic algorithm is robust to most hyperparameter settings, it is fairly brittle to the temperature α (Haarnoja, A. Zhou, Abbeel, et al., 2018; Haarnoja, A. Zhou, Hartikainen, et al., 2018). We may observe this in MolGym when constructing even simple molecules. In figure 3.4, we train an agent to build H_2O under the infinite bag setting for a range of temperatures α. Evidently, the temperature α must be carefully calibrated to avert seriously degraded performance. Moreover, even a fine-tuned temperature may be problematic.

In figure 3.5 we continue training the successful run (with $\alpha = .02$) from figure 3.4. We observe that although the agent momentarily learned to build H_2O it eventually breaks catastrophically. The root of this is partly a flaw in our implementation. But it also illuminates a more fundamental drawback in the SAC algorithm.

Consider that the reward scale changes over the course of training. Specifically, reward magnitudes shrink as the policy becomes better and the agent learns to avoid highly negative states. As previously discussed, smaller reward magnitudes correspond to a less deterministic optimal policy during the policy improvement step (equation 3.2). This means the the agent will update the correct policy to become increasingly random! Accordingly, in figure 3.5 (bottom row) we observe policy entropy gradually increases once the correct policy is found.

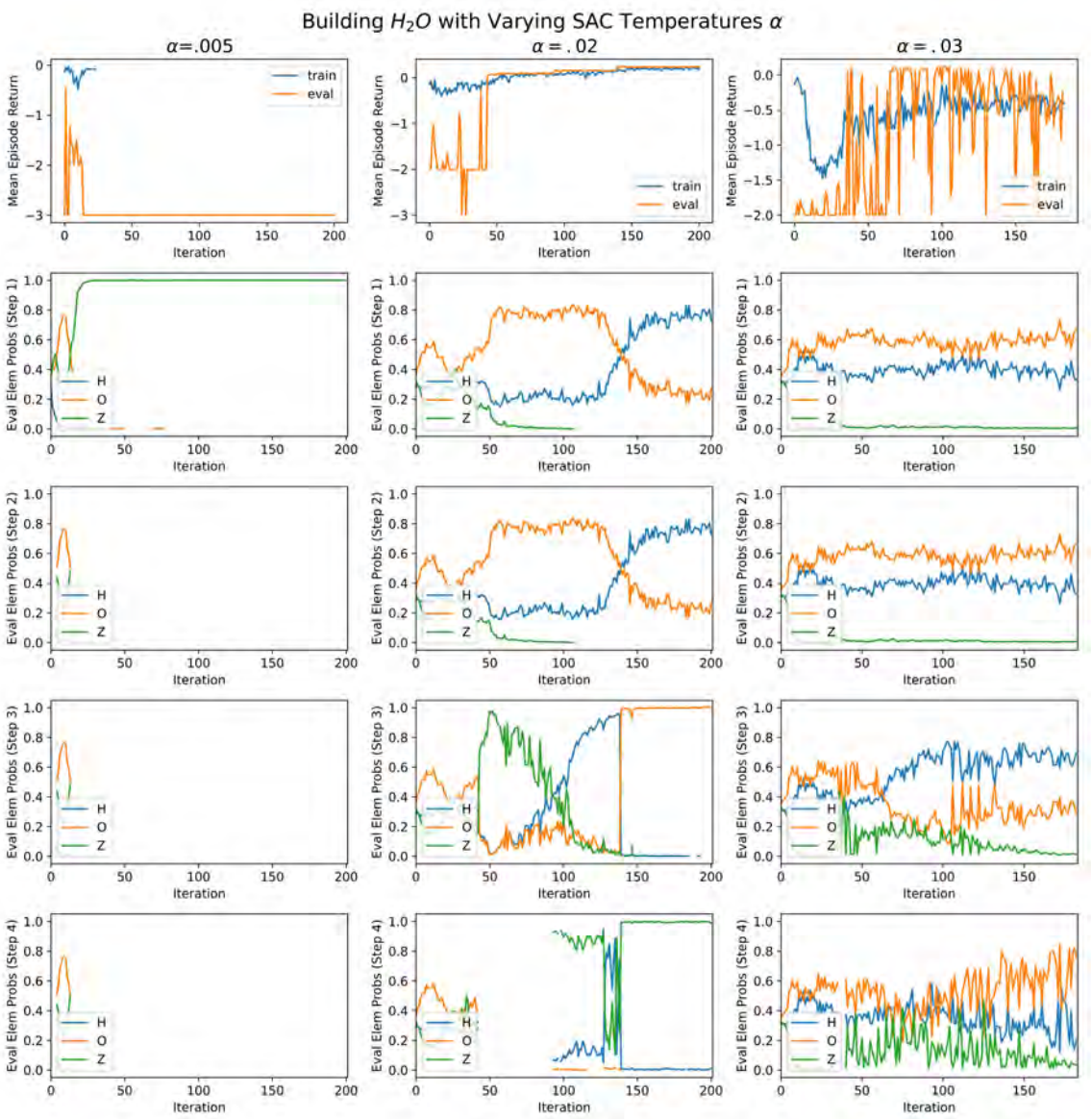


Fig. 3.4 In each column, we train MolGym in the new infinite bag setting with a different temperature parameter α. The top row shows the episode return over the course of training. Subsequent rows provide estimated probabilities for placing a particular element during the first 4 episode steps. These estimates are computed as the mean probabilities over evaluation runs. When α is well-tuned (middle column) the agent learns to obtain maximum reward by sequentially placing H,H,O, and finally Z to terminate the episode. But if α is incorrectly set the agent fails to learn to build even a simple molecule like H_2O.

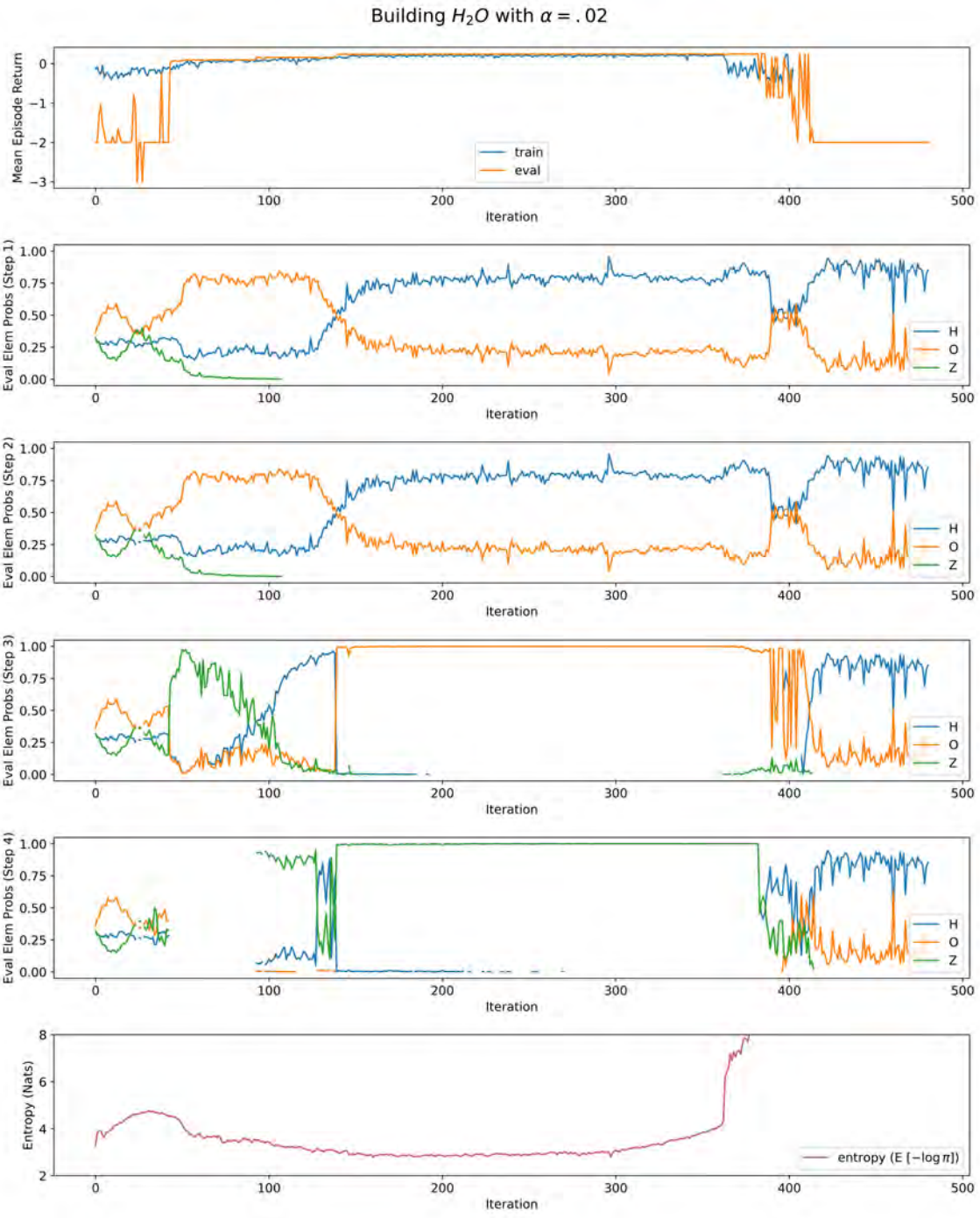


Fig. 3.5 The seemingly successful H_2O run with $\alpha = .02$ in figure 3.4 is continued, only to eventually break calamitously. The top row shows episode the return over the course of training. The following 4 rows indicate the estimated probabilities for placing a particular element during the first 4 episode steps in evaluation. The bottom row provides policy entropy (computed as $\mathbb{E}_{s_t \sim D, a_t \sim \pi(\cdot|s_t)}[-\log \pi(a_t|s_t)]$ over training batches). Once the agent finds the correct policy, policy entropy begin to increases gradually, then suddenly, and model performance degenerates.

.

This phenomena is especially striking because our implementation relies on advantage function estimates $A_\phi(s_t, a_t)$. That is, the update objective (equation 3.2) really resembles:

$$J_{imprv}(\theta) = \mathbb{E}_{s_t \sim \mathcal{D}} \left[D_{KL} \left(\pi_\theta(\cdot|s_t) \middle|\middle| \frac{\exp(\frac{1}{\alpha} A_\phi(s_t, \cdot))}{Z_\phi(s_t)} \right) \right] \tag{3.3}$$

After learning the correct policy, the advantage function estimates along correct policy trajectories are both accurate, as they are well-sampled, and very small in magnitude. The small magnitudes occur because the value of the current state should be similar to the optimal next state (especially in MolGym's undiscounted setting). Thus for a learned correct policy, the advantage function estimates in equation 3.3 tend to be smaller than the q-function estimates in equation 3.2. This implies an update to a policy π_θ with relatively higher entropy in our advantage function implementation.

The dependence of the policy entropy on the reward scale is particularly problematic in the infinite bag setting. Agents act naively early in training and thus typically incur significant per-atom penalties. As a result, reward scales vary more dramatically over the course of training compared to the finite bag setting. An appropriate temperature α for the initial rollouts may then be much too high in later iterations.

Nonetheless, an agent following a nearly correct stochastic policy should be able to learn that its occasional, random sub-optional actions are undesirable. But instead we observe the agent continue to take sub-optimal actions and eventually completely destabilizes. The root of this behavior is apparent in figure 3.6 which breaks down the entropy of the policy into the entropy of the focus, element, distance, and spherical orientation distributions. Evidently, the distance $p(d_t|e_t, f_t, s_t)$ distribution entropy drives the overall entropy blowup. Why? Recall, that the SAC objective (equation 2.11) includes an entropy maximization term. This term is usually balanced against the reward maximization objective. However, for the STOP atom Z, the agent can arbitrarily increase the entropy of the distance distribution because the STOP atom's distance d_T has no effect on the reward. The agent is thus able to initially increase the entropy of the distance distribution for the STOP atom innocuously. But this eventually destabilizes the parameters governing the distance distributions for the other atoms and in turn the model breaks.[2] To correct for this, we sever the backward gradient computation for the STOP atom's distance distribution entropy. We also do

[2]In additional detail: Recall that the distance distribution is a Gaussian mixture model parameterized by a neural network. Continuing to arbitrarily increase the GMM entropy ultimately corrupts the parameters of the underlying neural network from outputting appropriate GMM parameters for all atoms

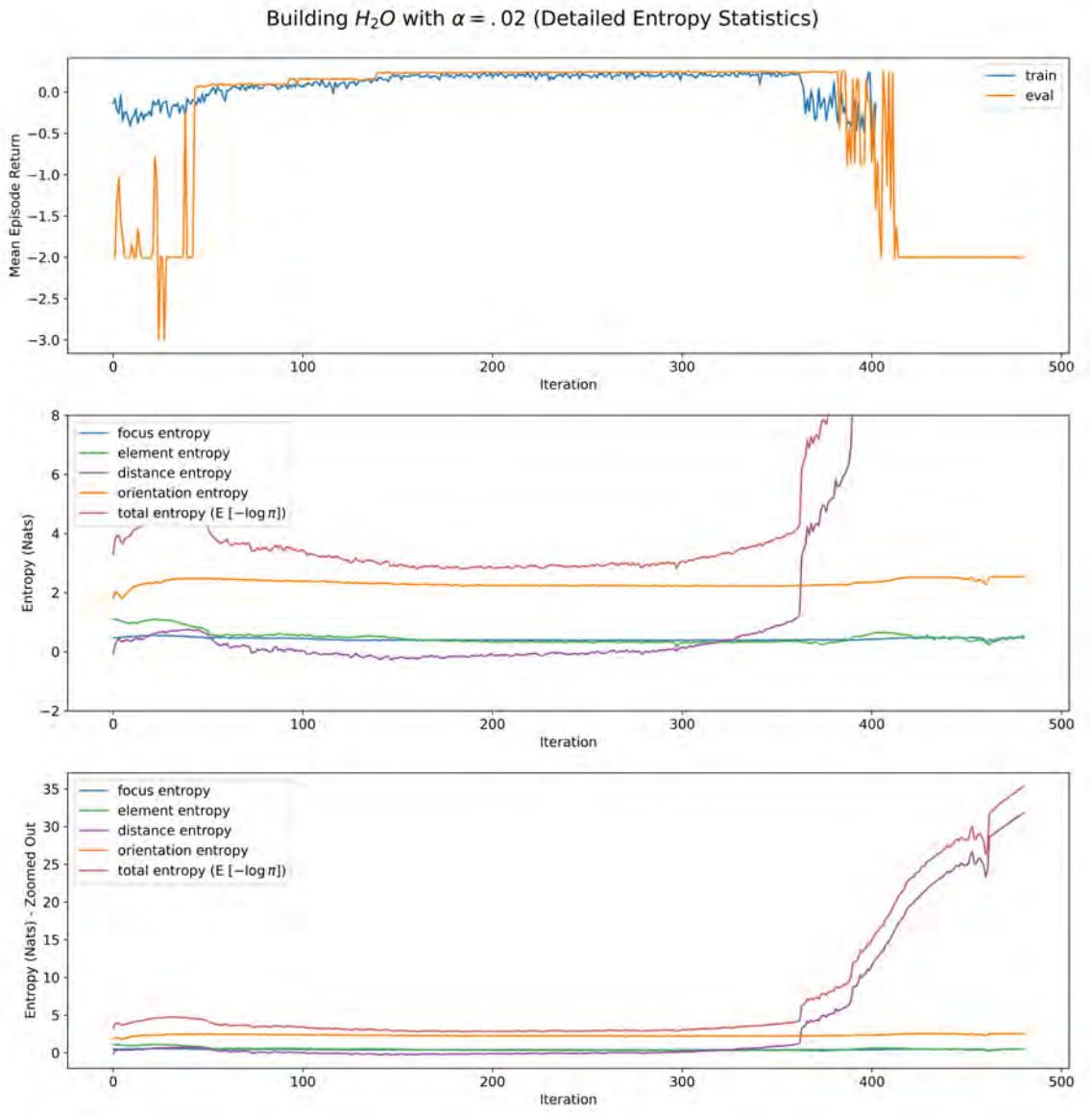


Fig. 3.6 Detailed entropy statistics for the initially successful but then degenerate H_2O run with $\alpha = .02$. Plots show reward and policy entropy over the course of training. The policy entropy (computed as $\mathbb{E}_{s_t \sim D, a_t \sim \pi(\cdot|s_t)}[-\log \pi(a_t|s_t)]$ over training batches) is broken into the entropy of its constituent distributions. Evidently the blowup in overall policy entropy is driven by the entropy of the distance distribution.

.

this for the STOP atom's orientation distribution.[3] In figure 3.7, we show that this prevents blowing up the entropy and breaking the model.

We still observe in figure 3.7 that policy entropy slightly increases after the agent learns the correct policy. This is because the optimal policy entropy still depends on the reward scale and the relatively small advantage function estimates of the correct policy imply a higher entropy policy update. In this case, nothing calamitous occurs and the agent just injects a bit more randomness into the order of atom placement. Nonetheless, if reward magnitudes vary more significantly over the course of training, an appropriate temperature at the beginning of training (when reward magnitudes are high) may be far too high once the agent determines the correct policy (and reward magnitudes are low). To then compensate for a potentially more varied reward scale, we implement a newer variant of the soft actor-critic with automatic entropy adjustment (Haarnoja, A. Zhou, Hartikainen, et al., 2018). As we shall see, SAC with automatic entropy adjustment not only prevents policy entropy from escalating at the conclusion of training. It also provides more hyperparameter robustness than by using a fixed temperature (as in figures 3.4 and 3.8) and averts a catastrophic entropy blowup (as in figure 3.6) even without severing the STOP atom's distance/orientation distribution entropy gradients.

3.4.2 SAC with Automatic Entropy Adjustment

The key idea in SAC with automatic entropy adjustments (Haarnoja, A. Zhou, Hartikainen, et al., 2018) is to recast the original SAC objective (equation 2.10) as now determining a stochastic policy that maximizes return subject to a minimum expected entropy constraint:

$$\max_{\pi_{0:T}} \mathbb{E}_{(s_t,a_t)\sim\rho_{\pi_t}}\left[\sum_{t=0}^{T} r(s_t,a_t)\right] \text{ s.t. } \mathbb{E}_{(s_t,a_t)\sim\rho_{\pi_t}}[-\log(\pi_t(a_t|s_t))] \geq \overline{\mathcal{H}} \ \forall t \tag{3.4}$$

where $\overline{\mathcal{H}}$ is the target minimum entropy, π_t specifies the policy at timestep t, and ρ_{π_t} denotes the state action marginal distribution induced by the policy π_t. Critically, the entropy of the optimal policy no longer depends on the scaling of rewards. Instead, the policy entropy must exceed a fixed minimum entropy $\overline{\mathcal{H}}$ at every time t. Ensuring this requires incorporating and appropriately updating a dynamic temperature α. More

[3] Although the distance entropy clearly engenders the entropy blowup in our example, we also risk arbitrarily increasing the entropy of the STOP atom's orientation distribution as its orientation $\tilde{x}_T$ likewise has no effect on reward.

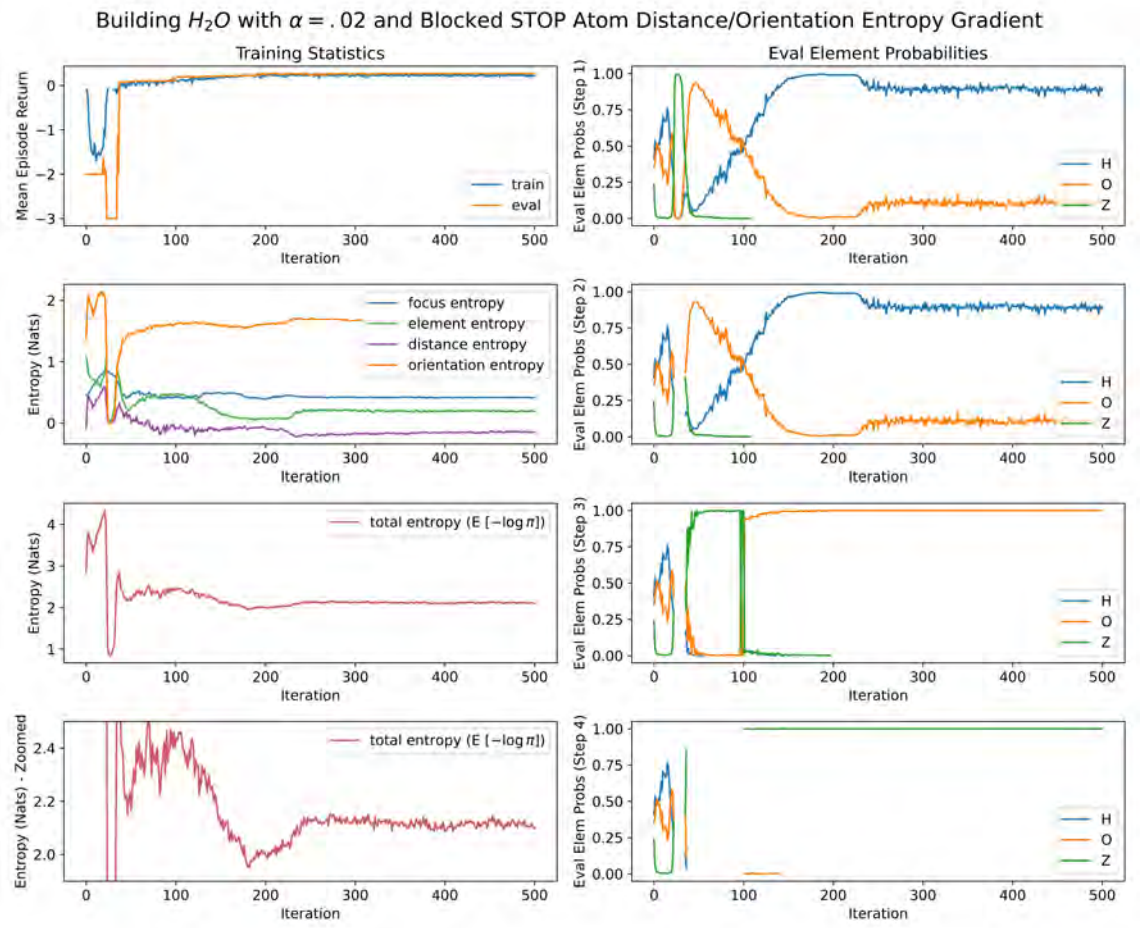


Fig. 3.7 Successfully building H_2O in the infinite bag setting by severing backward gradient computation for the STOP atom's distance and orientation distribution entropies. Left column plots show reward and policy entropy over the course of training. Right column plots provide estimated probabilities for placing a particular element during the first 4 episode steps. After the agent fixates on the correct policy (at roughly 180 iterations) entropy rises slightly due to the shrunken reward scale. However, we prevent entropy from further blowing up (in contrast to figure 3.6) by blocking gradient flows from the STOP atom's distance and orientation distribution entropies.

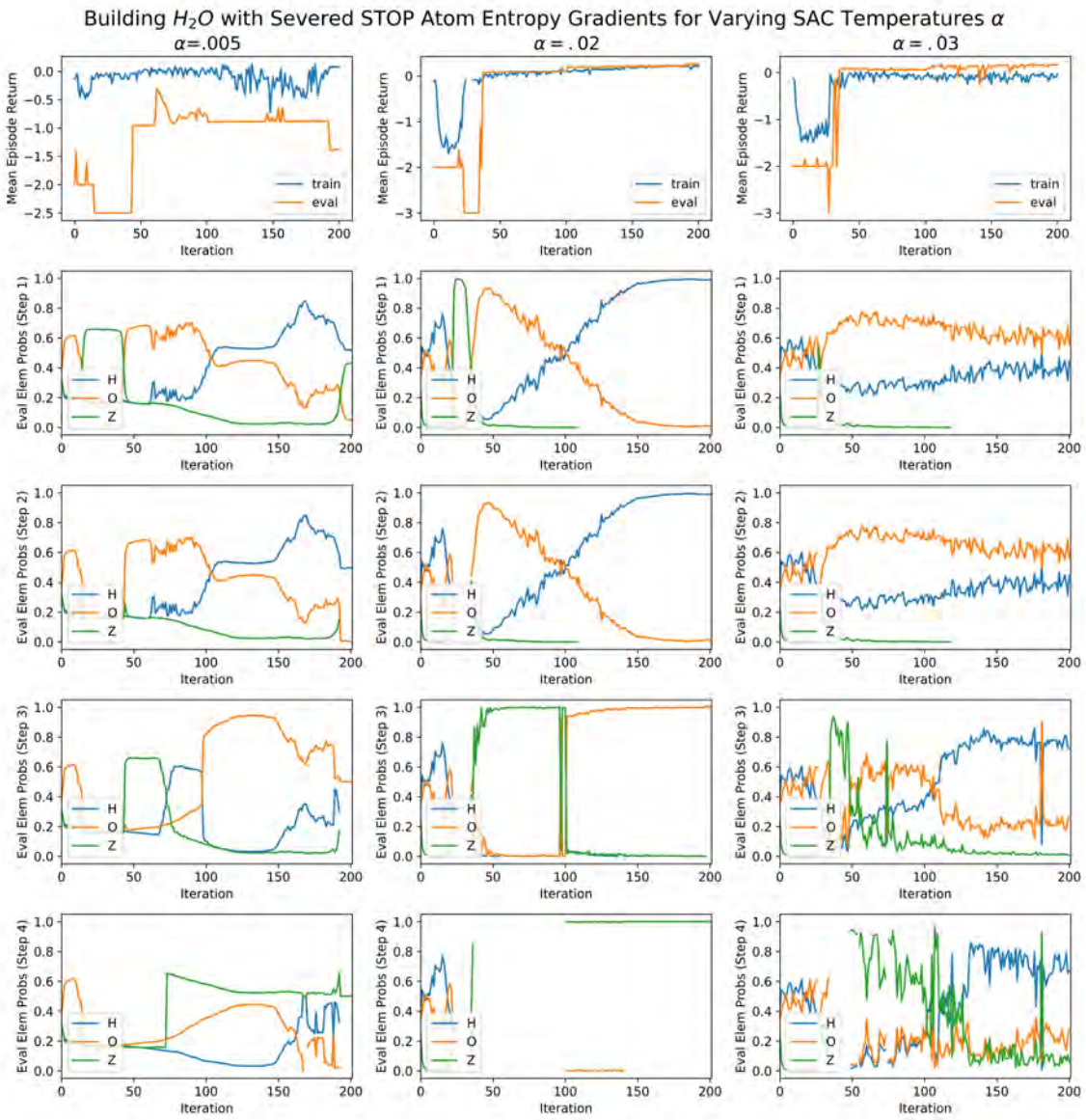


Fig. 3.8 In each column we train MolGym in the new infinite bag setting with a different temperature hyperparameter α, similar to figure 3.4. However, in these runs we sever backward gradient computation for the STOP atom's distance and orientation distribution entropies. Although this prevents the catastrophic entropy blowup in the run with $\alpha = .02$, it does not resolve the more fundamental brittleness of the SAC algorithm to the temperature hyperparameter. The agent with $\alpha = .03$ ultimately obtains a suboptimal reward by building hydrogen peroxide H_2O_2 (instead of H_2O) while the agent with $\alpha = .005$ performs even worse.

precisely, a dual objective may be derived from equation 3.4 but the derivation[4] relies on assumptions that do not hold for neural networks. Nonetheless, the authors observed that updating gradients on the temperature α with the following objective[5] still works empirically:

$$J_{temp}(\alpha) = \mathbb{E}_{s_t \sim \mathcal{D},\, a_t \sim \pi(\cdot|s_t)} \left[-\alpha(\log \pi(a_t|s_t) + \overline{\mathcal{H}}) \right] \tag{3.5}$$

By implementing automatic entropy adjustment we decouple the optimal entropy from the reward scale and thus prevent the policy from growing more stochastic as reward magnitudes shrink. Moreover, although the objective imposes only a lower bound on entropy, we also mitigate the policy from blowing up with excessively high entropy (as in figures 3.5 and 3.6). Specifically, the loss function (equation 3.5) causes the temperature of the policy to decrease if the current policy entropy $\mathcal{H}$ is greater than the minimum target entropy $\overline{\mathcal{H}}$. We derive this conclusion as follows:

$$\begin{aligned} J_{temp}(\alpha) &= \mathbb{E}_{s_t \sim \mathcal{D},\, a_t \sim \pi(\cdot|s_t)} \left[-\alpha(\log \pi(a_t|s_t) + \overline{\mathcal{H}}) \right] && (3.6) \\ &= \alpha \left(\mathbb{E}_{s_t \sim \mathcal{D},\, a_t \sim \pi(\cdot|s_t)} \left[-\log \pi(a_t|s_t) \right] - \overline{\mathcal{H}} \right) && (3.7) \\ &= \alpha \left(\mathcal{H} - \overline{\mathcal{H}} \right) && (3.8) \end{aligned}$$

Taking the gradient with respect to α:

$$\nabla_\alpha J_{temp}(\alpha) = \mathcal{H} - \overline{\mathcal{H}} \implies \nabla_\alpha J_{temp}(\alpha) > 0 \quad (\text{if } \mathcal{H} > \overline{\mathcal{H}}\ ) \tag{3.9}$$

Since we minimize $J_{temp}(\alpha)$ the corresponding update to α is negative:

$$\alpha \leftarrow \alpha - \lambda_\alpha \nabla_\alpha J_{temp}(\alpha) \tag{3.10}$$

where λ_α is a learning rate. Thus if the policy has excessively high entropy, the temperature is updated to dynamically shrink.

In figure 3.9 we show that an agent with an auto-tuned temperature α successfully finds and maintains the correct H_2O build policy even without severing gradient flows from the STOP atom's distance/orientation entropy. Figure 3.9 also demonstrates how the temperature α is adjusted over the course of training so that the optimal policy attains a particular target entropy $\overline{\mathcal{H}} = 2$.

[4]Details of the derivation are available in Haarnoja, A. Zhou, Hartikainen, et al., 2018.

[5]Note that we've made minor adjustments to the notation in the SAC with automatic entropy adjustment paper (Haarnoja, A. Zhou, Hartikainen, et al., 2018) to align with our own.

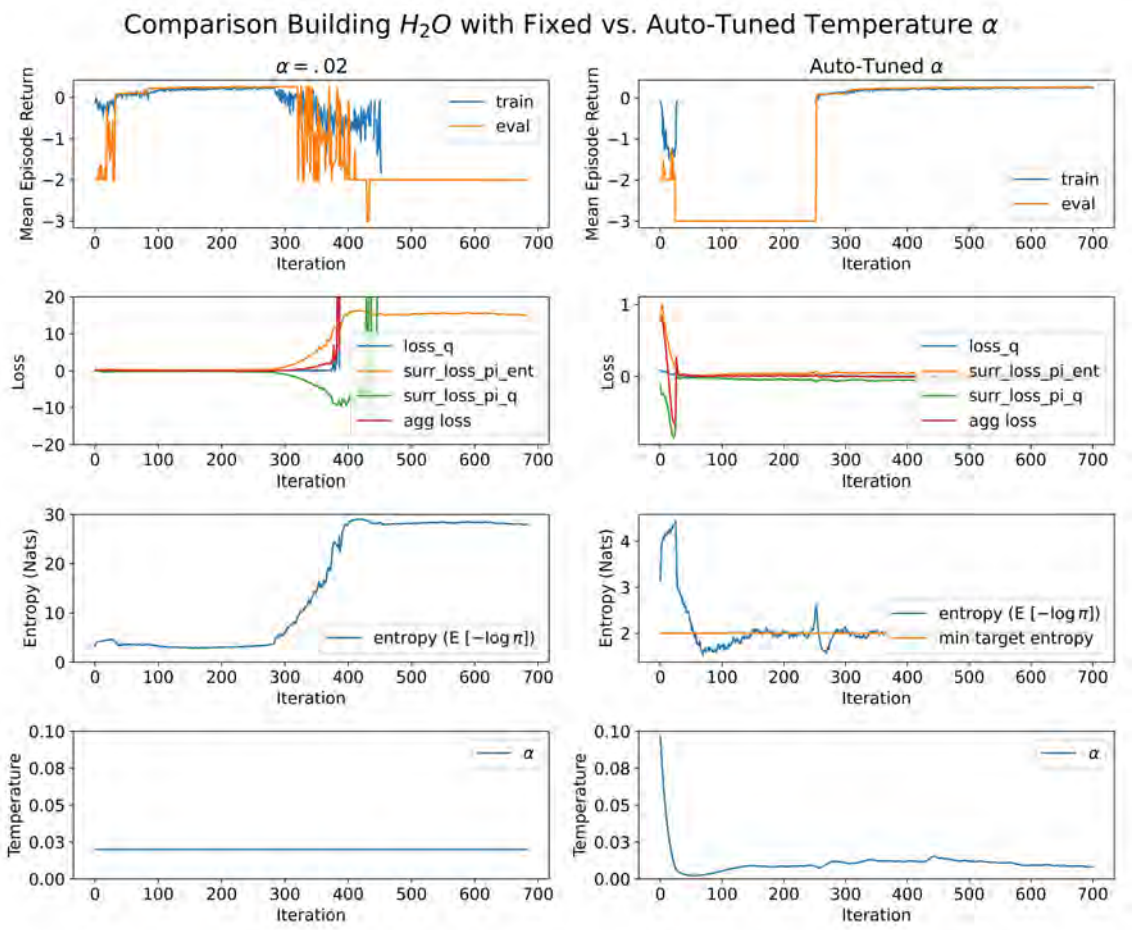


Fig. 3.9 Comparison building H_2O between an agent with a fixed temperature α (column 1) and an agent with a dynamic, auto-tuned temperature α (column 2). Note that the STOP atom gradient's on the distance/orientation distribution entropy are not severed for the purposes of this comparison. Both agents initially determine the correct policy, but the agent with the fixed temperature α eventually degenerates when entropy escalates late in training. In contrast, the auto-tuned temperature agent dynamically adjusts α so that the policy entropy (and temperature) remain low once the agent determines the correct policy.

Using SAC with automatic entropy adjustment requires us to set two hyperparameters (an initial α_0 and a minimum target entropy $\overline{\mathcal{H}}$) as opposed to a single fixed α. Nonetheless, basic ablation experiments (figures A1, A2) show these are decently robust, especially when compared with the brittleness of setting a fixed temperature α as noted in the literature (Haarnoja, A. Zhou, Abbeel, et al., 2018; Haarnoja, A. Zhou, Hartikainen, et al., 2018) and also evident in our experiments (figures 3.4, 3.8).

Nevertheless, SAC with automatic entropy adjustment is no panacea. We note for instance that even in the successful H_2O training run (figure 3.9, column 2) that the agent spends over 200 training iterations following a degenerate near-deterministic policy wherein it places the STOP atom at the start of each episode. Fortunately, the agent is able to eventually recover as it's programmed to ignore such bad rollouts and train exclusively on the varied, interesting episodes it accrued earlier. Nonetheless, it is certainly not ideal that the agent quickly fixates on a poor near-deterministic policy as this behavior slows convergence. Often we may prefer greater exploration at the outset i.e., the temperature α and optimal entropy to more gradually fall than in figure 3.9. We're able to adjust initial exploration somewhat by tuning the target entropy and initial temperature (figures A1 and A2). But although we may realize moderate convergence speedups through tuning, an issue remains: the target entropy is, we recall, the *minimum* target entropy over the entire course of training. Thus it needs to be set sufficiently low so the agent can learn the usually near-deterministic correct policy. In turn, the temperature often falls too steeply at the beginning of training in order for the current policy entropy to roughly match the minimum target entropy (e.g., see figures 3.9, A.2, A.1).

To encourage sufficient early exploration we may lower the learning rate λ_α for the temperature parameter.[6] In figure 3.10 we see that by lowering λ_α appropriately we may avoid adopting an incorrect, overly-deterministic policy early in training. However if λ_α is set too small, the agent takes excessively long to lower α to where the correct near-deterministic policy may be realized. In summary, by specifying λ_α we gain greater control over initial exploration, but we also introduce an additional hyperparameter.[7]

A straightforward alternative is to dynamically adjust the temperature α according to a decay schedule. In figure 3.11 we show that by lowering α over the course of training we can explore amply at the outset and avoid blowing up the entropy of

[6] In practice we implement gradient updates on α (equation 3.5) using an AdamW optimizer (Loshchilov and Hutter, 2017) and so lower the learning rate parameter of AdamW.

[7] Of course the learning rate λ_α must be specified regardless. But if we set λ_α to the same AdamW learning rate as the model optimizer (instead of distinctly setting λ_α) then we effectively avoid needing to tune it.

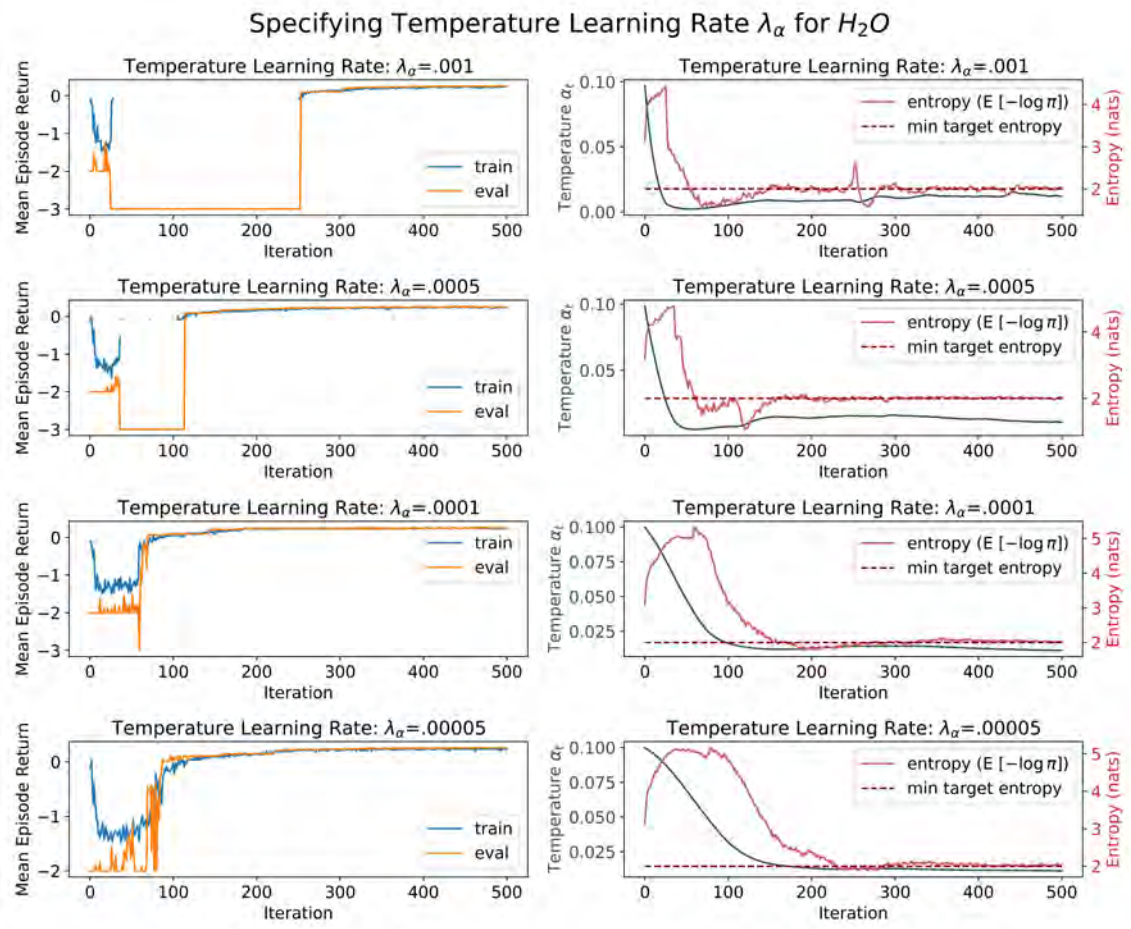


Fig. 3.10 Building H_2O with an auto-tuned temperature α for different temperature learning rates λ_α. By tuning λ_α appropriately, the agent may explore sufficiently at the outset of training and converge to the correct policy sooner.

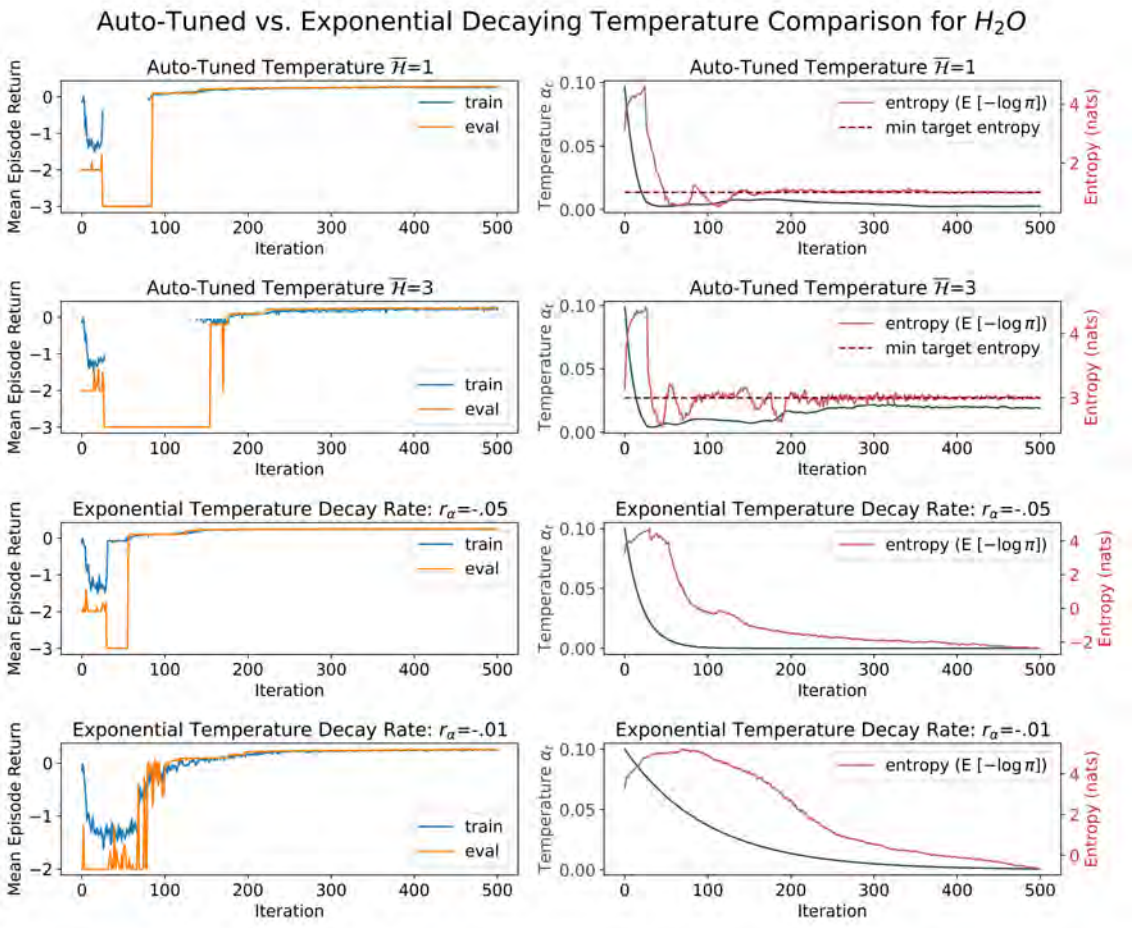


Fig. 3.11 Comparison building H_2O with an auto-tuned temperature (top 2 rows) and an exponentially decaying temperature (bottom 2 rows). In all, the initial temperature is set at $\alpha_0 = .1$. We observe that if the exponential decay rate is tuned appropriately, we can explore sufficiently at the beginning of training and find the correct policy promptly.

the optimal policy as the reward scale shrinks later in training. This approach also involves setting only two hyperparameters (the initial temperature α_0 and decay rate r) versus three in the auto-tuned SAC variant (the initial temperature α_0, minimum target entropy $\overline{\mathcal{H}}$, and learning rate λ_α).

There is no absolute answer to whether a fixed, exponentially decaying, or auto-tuned temperature is optimal. However, if the reward scale varies significantly over the course of training, as may occur in the infinite bag setting, either a exponentially decaying or auto-tuned temperature is likely preferred. The auto-tuned temperature is well motivated and cleverly adjusts to a dynamic reward scale but may induce insufficient initial exploration without an appropriate temperature learning rate λ_α.

Lastly, a decaying temperature provides the flexibility to explore amply at the outset, but needs to be carefully tuned to the timescale of a run. All options are now implemented in MolGym and simple to switch between.

3.5 Infinite Bag Summary

In this chapter, we motivated infinite atom bags as a compelling new paradigm that enables MolGym agents to now learn the optimal size and composition of molecular structures. We proceeded to detail our implementation of the infinite bag setting into MolGym using per-atom penalties. Finally, we addressed the brittleness of the SAC algorithm's temperature hyperparameter by implementing a more robust variant: SAC with automatic entropy adjustment (Haarnoja, A. Zhou, Hartikainen, et al., 2018). Altogether, we successfully built small molecules (e.g., H_2O) under the infinite bag setting while improving hyperparameter robustness. Next, we will explore how to further empower MolGym infinite bag agents with randomized atom penalties.

Chapter 4

Randomized Atom Penalties

4.1 Motivation

In the previous chapter, we enabled MolGym agents to learn to build molecules based on dynamic per-atom penalties. The penalties were dynamic in the sense that they varied over the course of a molecule's construction. However, the underlying penalty schedule remain fixed. For example, to encourage O_2 construction we set the environment to always raise the penalty on oxygen after 2 oxygen were placed. In this chapter, we explore randomizing the penalty schedule. This allows practitioners to create more powerful agents that flexibly react to changing penalty conditions and are capable of constructing a broader diversity of molecules.

4.1.1 Motivating Example: Molecular Oxygen and Ozone

As a guiding example we will consider training a MolGym agent that should build both molecular oxygen (O_2) and ozone (O_3). This implies that our environment should behave stochastically; it should sometimes raise the penalty on oxygen after 2 are placed and sometimes raise the penalty on oxygen after 3 are placed.

4.2 Implementation

We implement environment stochasticity by drawing a randomized penalty schedule at the start of each rollout. In our example, the penalty schedule is constructed such that the penalty increases on oxygen after $n \sim \text{randint}(2, 3)$ oxygens are placed. More generally we enable the practitioner to specify a random range for each atom and additionally allow multiple penalty increases. For instance, a practitoner may specify

Fig. 4.1 An agent is trained to construct both O_2 and O_3 through randomized penalties. We classify the molecules built over 4 evaluation rollouts for each of the final 100 training iterations. Evidently, the agent learns to always construct either O_2 or O_3 but often chooses the incorrect molecule to build. O_3 Incorrect means that the agent built O_3 when the oxygen penalty increased after 2 oxygens were placed and thus could've achieved a higher reward by building O_2. Similarly, O_2 Incorrect means that the agent built O_3 when the oxygen penalty did *not* increase after 2 oxygens were placed and thus could've achieved a higher reward by building O_3.

that the penalty on sulfur increases after $n \sim \text{randint}(2, 3)$ sulfur are placed and then increases again after another $n \sim \text{randint}(0, 2)$ sulfur are placed. This provides the flexibility to construct highly adaptable MolGym agents.

If we now train our agent to build O_2/O_3, we find that although the agent learns to build both molecular oxygen and ozone it fails to distinguish when constructing each is appropriate (see figure 4.1). That is, the agent fails to learn to build O_2 when the oxygen penalty increases after 2 oxygen are placed (and O_3 when the oxygen penalty increases after 3 oxygen are placed). To resolve this, we must reconsider what constitutes the agent state under the infinite bag setting.

4.3 Full State Estimation

In this section, we will establish how randomized penalties subvert our current notion of full state, describe why this is problematic given the SAC policy update, and finally derive a fix that restores a true full state.

We'll explain by an example. First, consider two environments:

- In env A, the oxygen penalty increases after 2 oxygen are placed. Thus the agent receives a high reward for building O_2 and a lower reward for building O_3
- In env B, the oxygen penalty increases after 3 oxygen are placed. Thus the agent receives a low reward for building O_2 and a higher reward for building O_3

Let state s_{3O} consists of a canvas containing O_3 and a bag where the oxygen penalty is high.[1]. Next let's consider the Q-function estimates for state s_{3O}. Recall that in MolGym, the Q-function effectively acts only on the next state as opposed to the current state and action (for details see section 2.5.1). Additionally recall that rewards are sparse. That is, all rewards are zero except at episode termination where the reward consists of the standard energy reward plus all accrued penalties (equation 3.1).

- In env A, $Q_\phi(s_{3O})$ should be low because although the agent has successfully built O_3 the agent placed an oxygen atom after the penalty on oxygen was raised.
- In env B, $Q_\phi(s_{3O})$ should be high because the agent successfully built O_3 and placed all three oxygens before the oxygen penalty was raised.

This is problematic as the confusion over the correct value of $Q_\phi(s_{3O})$ results in suboptimal policy updates. To illustrate this, recall the policy improvement step (equation 2.20 reproduced here with a Q-function that operates on the next state):

$$J_{imprv}(\theta) = \mathbb{E}_{s_t \sim \mathcal{D}} \left[D_{KL} \left(\pi_\theta(\cdot|s_t) \middle|\middle| \frac{\exp(\frac{1}{\alpha} Q_\phi(s_{t+1}))}{Z_\phi(s_t)} \right) \right] \tag{4.1}$$

Now let state s_{2O} come from a rollout of environment A with 2 oxygen atoms on the canvas. Our policy improvement step becomes:

$$J_{imprv}(\theta) = D_{KL} \left(\pi_\theta(\cdot|s_{2O}) \middle|\middle| \frac{\exp(\frac{1}{\alpha} Q_\phi(s_{t+1}))}{Z_\phi(s_{2O})} \right) \tag{4.2}$$

Note that if $Q_\phi(s_{3O})$ is high then we update the policy π_θ to more frequently select the suboptimal action (placing a third O) that results in $s_{t+1} = s_{3O}$. Alternatively,

[1] Note in either environment that the oxygen penalty will be high after 3 oxygens are placed

if $Q_\phi(s_{3O})$ is low we correctly select that action less frequently. However, in the case $Q_\phi(s_{3O})$ is low but s_{2O} comes from a rollout of environment B then we update the policy π_θ to less frequently select the optimal action that results in $s_{t+1} = s_{3O}$. In summary, for either setting of $Q_\phi(s_{3O})$ faulty policy updates are certain to arise.

We emphasize that the poor calibration of $Q_\phi(s_{3O})$ is *not* simply because $Q_\phi(s_{3O})$ reflects environment uncertainty. It is true that, for instance, the value estimate of the initial state $Q_\phi(s_{init})$ should reflect environment uncertainly as the agent does not yet know when penalties will increase. But the environment uncertainty should be resolved at the moment the agent computes $Q(s_{3O})$ as penalties have already increased.

Interestingly, although $Q_\phi(s_{3O})$ should differ between environments A and B, the optimal action from state s_{3O} will not. In fact, the optimal action from any state s is the same in environment A and B. This is because the agent's current goal is to maximize reward from state s and the action to achieve so is entirely dependent on s, and in particular independent of all previously incurred penalties. In this sense the MolGym Markov decision process is fully observable with the state s just consisting of the canvas, bag, and current penalties. However, a complication with observability arises due to our choice of a decision algorithm (SAC) that depends on value estimates of s. Specifically, the next action decision is mediated by Q-function estimates that depend on previously incurred penalties. Since s does not contain such information, we effectively end up with a partially obersvable Markov decision process.[2]

A solution to restore full observability to the MDP is to incorporate the total penalty accrued into the state s. Thus we redefine the state as $s' = s \cup p$ where p is the total penalty accrued from state s_0 to state s. This resolves the conundrom with environments A and B as the Q-functions now take slightly different inputs:

- In env A, $Q_\phi(s_{3O} \cup p_A)$ where p_A is relatively high
- In env B, $Q_\phi(s_{3O} \cup p_B)$ where p_B is relatively low

After incorporating the total accrued penalty into the state, we need only adapt the Q-network. This is because, as noted previously, the optimal next action is to maximize future reward and independent of past accrued penalties. Thus we may dismiss the total accrued penalty as an input to the policy network. For the Q-network, recall that the final computation involves concatenating an embedding of the canvas with the bag and passing the results through a MLP. We modify this slightly so that

[2]Note this is only true for randomized penalty schedules as otherwise s suffices to correctly estimate Q_ϕ once the agent learns the penalty schedule

Table 4.1 An agent trained to build both oxygen (O_2) and ozone (O_3) is evaluated over 100 rollouts once training concludes. For each episode a random penalty schedule is selected such that the penalty on oxygen increases after either 2 or 3 oxygen are placed on the canvas. We observe that the agent has correctly learned to build oxygen when the penalty increases after 2 oxygen are placed and ozone when the penalty increases after 3 oxygen are placed.

		Oxygen penalty rises after..	
		2 oxygen placed	**3** oxygen placed
Over 100 eval rollouts...	# O_2 built	45	0
	# O_3 built	0	55

the MLP takes as input a concatenated tensor consisting of the canvas embedding, bag, *and* total accrued penalty.

4.4 Results: Molecular Oxygen and Ozone

With the modifications to the state described in the previous section, we may now empower an agent to build both oxygen and ozone. Specifically, we train an agent on rollouts where the oxygen penalty randomly increases after between 2 or 3 oxygen are placed on the canvas. In figure 4.2 we demonstrate how the evaluation element probabilities vary over the course of training. The agent initially learns to build O_2 exclusively but eventually learns to tailor its molecular construction to the particular penalties it encounters over a rollout. We may confirm the agent has learned to correctly alter its actions according to current penalties through both the final training iterations (figure 4.3) and supplementary evaluation results (table 4.1).

4.5 Alternative: Dense Penalties

Rather than incorporating the total accrued penalty into the state s_t, we may resolve the trouble with randomized penalties by reformulating them as dense (negative) rewards. Consider in MolGym's sparse reward paradigm that penalties are not realized until an episode terminates. In other words, the environment tracks the penalties incurred at each step but does not communicate accrued penalties to the agent until the episode concludes.

Instead, we may direct the environment to immediately return the penalty incurred at each episode step. This creates a framework where $Q_\phi(s_t)$ should estimate the sum of all *future* penalties from state s_t plus the final molecule energy reward. Thus, using

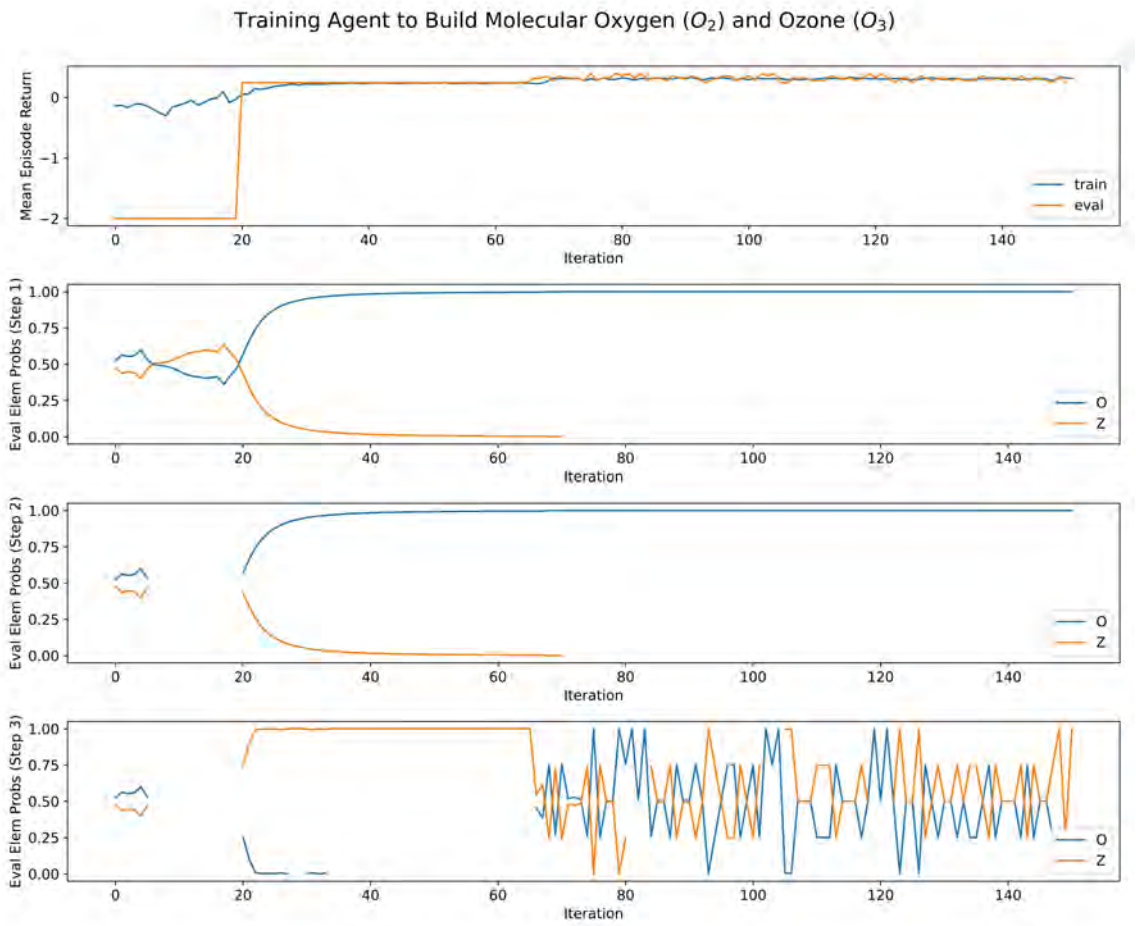


Fig. 4.2 Reward and evaluation element probabilities over the course of training an agent to build both oxygen O_2 and ozone O_3. Early in training, the probability of placing a third oxygen is always zero (the agent builds O_2 exclusively) but eventually the probability of placing a third O jumps between 0%, 25%, 50%, 75%, & 100% at every iteration. This is because each evaluation iteration consists of 4 rollouts and each rollout is equally probable to have raised the penalty on oxygen at the second or third step. Thus the average probability that the agent places the third O is correctly dependent on how many episodes (out of 4) involve the later penalty increase.

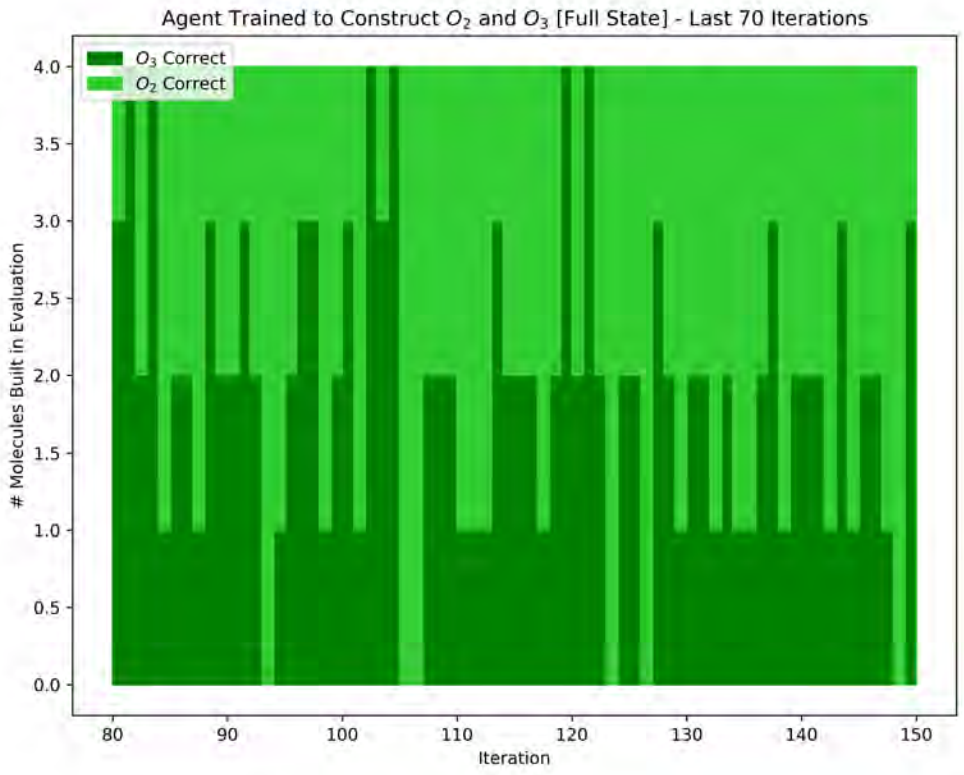


Fig. 4.3 An agent is trained to construct both O_2 and O_3 through randomized penalties, as in figure 4.1, but now we incorporate the total accrued penalty into the state. We classify the molecules built over 4 evaluation rollouts for each of the final 70 training iterations. Evidently, the agent learns to always correctly construct either O_2 or O_3 depending on the penalty schedule.

our earlier example, $Q_\phi(s_{3O})$ should estimate the same value for both environments A and B. This is because $Q_\phi(s_t)$ no longer depends on any past penalties as the agent has already incurred them as (negative) rewards. Thus, we need not introduce the total accrued penalty into the state s_T.

We've glossed over an important detail relating to MolGym's implementation of Q-functions as acting solely on the next state $Q(s_{t+1}) = Q_\phi(\mathcal{T}(s_t, a_t))$ (see section 2.5.1 for additional detail). Specifically, that implementation relies on sparse rewards to ensure path independence. In the case of dense rewards, the transition dynamics are still deterministic but $Q_\phi(s_{t+1})$ depends on the now nonzero reward incurred transitioning from s_t to s_{t+1}. Thus we may no longer recast $Q_\phi(s_{t+1}) = Q_\phi(\mathcal{T}(s_t, a_t))$ as the penalties incurred transitioning into state s_{t+1} may vary.

The straightforward resolution is to simply undo the $Q_\phi(s_{t+1}) = Q_\phi(\mathcal{T}(s_t, a_t))$ simplication in MolGym by implementing Q-functions that operate on both state and action (ie., the standard way: $Q(s_t, a_t)$). However, this presents a far tougher modeling challenge. In particular, the Q-function must learn to not just model states, but how those states are augmented by actions. This is needlessly complicated given the deterministic transition dynamics. By exploiting those dynamics, we can rewrite the Q-function as operating on the next state as before plus a transition reward penalty rp_t:[3]

$$Q_\phi(s_t, a_t) = Q_\phi(\mathcal{T}(s_t, a_t), r_{pt}) = Q_\phi(s_{t+1}, r_{pt}) \tag{4.3}$$

The function $Q_\phi(s_t, a_t)$ estimates the sum of all future rewards from s_t if the agent takes action a_t. In MolGym, this is the final molecule energy plus the value of all penalties incurred from state s_t onwards. Thus we can futher simplify:

$$Q_\phi(s_{t+1}, rp_t) = Q_\phi(s_{t+1}) + rp_t \tag{4.4}$$

where $Q_\phi(s_{t+1})$ estimates the final molecule energy plus the value of all future penalties incurred from state s_{t+1} and rp_t is the penalty incurred transitioning from state s_t to s_{t+1}. As a result, we may implement dense penalties into MolGym with a minimal modification to the current Q-function $Q_\phi(s_{t+1})$: we must simply add the reward penalty rp_t that was incurred transitioning into state s_{t+1}.

[3]We are careful to use the notation rp_t to distinguish the (negative) reward penalty incurred at timestep t from the total reward r_t incurred at that timestep. This is because although the reward penalty constitutes the entire reward at most timesteps, the terminal reward will also include the energy reward.

Dense penalties are now enabled as a hyperparamter option for the MolGym infinite bag setting. However, we found that incorporating the total accrued penalty into the state usually functioned better (quicker convergence, more robust) in resolving randomized penalties.

4.6 Randomized Penalty Summary

In this chapter, we discussed incorporating randomized atom penalties into MolGym's infinite bag setting. We first described how randomized penalties may bring about more versatile MolGym agents and then outlined a straightforward implementation. We next proved why MolGym states don't incorporate sufficient information to model randomized atom penalties through the soft-actor critic Q-function estimates. To resolve this, we integrate the accrued penalties over a rollout into the state. Alternatively, we showed how using dense penalties also corrects the Q-function estimates. Ultimately, we managed to train an agent that alternatively built molecular oxygen O_2 or ozone O_3 according to a random penalty schedule. We've now seen MolGym's infinite bag successfully applied in constructing small molecules under different settings. In the next chapter, we explore scaling to larger molecular structures.

Chapter 5

Scaling to Complex Structures

In the preceding chapters, we've introduced the powerful infinite bag setting into MolGym and demonstrated its success in building small molecules. Next, we detail efforts to scale MolGym to construct larger and more complex molecules under the infinite bag setting. As we will show, this presents thorny optimization challenges due to the enormous space of potential molecules. We thus propose both a simple technique to robustify optimization (specifying initial probability distributions) and a more involved strategy (applying a variant of Soft Q Imitation Learning).

5.1 Optimization Challenges

Optimization on infinite bag MolGym agents routinely failed in learning more complex structures. More precisely, first suppose we encode penalties such that an agent obtains maximum reward by building a particular, now more complex, target molecule. This agent policy ordinarily corresponds to the global minimum loss.[1] However, the optimizer often stumbled in locating parameters near the global minimum loss and thus the agent failed to learn the target molecule. The loss landscape under the infinite bag setting was likely difficult to optimize due to 1) the exponential growth in constructable chemical formulas and 2) constraints on penalty formulation.

[1] This policy only *ordinarily* corresponds to the global minimum loss because of the additional entropy maximization objective. However, if the penalties and temperature are tuned appropriately then the statement strictly holds.

5.1.1 Exponential Growth in Chemical Formulas

In the original MolGym finite bag setting we specified a bag (e.g., SOF_6 to construct pentafluorosulfur hypofluorite) and then in every rollout constrained the agent to place exactly those atoms. Moreover, the agent was trained on rollouts consisting exclusively of SOF_6.

Under the infinite bag setting, this becomes far more complex. Now the agent may place any number of S,F, and O and is trained on rollouts that may've built extremely varied configurations (e.g, S, $S_3F_4O_6$, S_8O_2, etc...). Specifically, an agent trained on M distinct elements with a maximum canvas size of N is capable of building molecules with an exponential number $O(M^N)$ of different chemical formulas. In contrast, under the finite bag setting the number of unique chemical formulas is just 1.

Unsurprisingly, we rarely sample rollouts consisting of exactly the desired SOF_6 atoms early in training. Assuming that we initially select from S,O,F and the STOP atom Z with equal probability, then we expect to sample SOF_6 every 4681 rollouts in expectation.[2] Moreover, that rare sample of SOF_6 merely consists of the atoms SOF_6 and almost certainly doesn't geometrically resemble pentafluorosulfur hypofluorite on the canvas. Of course, we intend to collect better rollouts later in training and regardless the agent is capable of learning from the transitions of suboptimal rollouts, SOF_6 or otherwise. But clearly the agent's optimization is far less constrained toward the objective (building SOF_6) than under an explicit finite bag specification. As a consequence, optimization often gets stuck far from the global minimum loss.

In figure 5.1 we observe that an agent trained with penalties that incentive SOF_6 construction ends up building structures like SOF_2, S_2OF_5, and $SOF_5 + O$.[3] We emphasize that the energy reward and our selection of penalties imply that the agent would receive a markedly higher reward if it instead constructed SOF_6. So why does the optimizer get stuck? In the SOF_2 run, the shrunken gradient norms suggest the optimizer has become trapped near some sort of critical point (Goodfellow, Bengio, and Courville, 2016). We note that although the optimizer may have lurched into a local minimum, saddle points tend to be more common in high dimensional spaces (Goodfellow, Bengio, and Courville, 2016). In the other runs, the less convergent gradient norms, particularly in the case of S_2OF_5, prevent us from conclusively labeling

[2]The probability of randomly sampling SOF_6 is $\frac{1}{4}^9 * 8$ [options for step that S is chosen] $*$ 7 [options for step that O is chosen] . Note that the step the STOP atom Z is chosen is fixed (the final step) and the steps F is chosen are thereby determined to the 6 remaining steps. Then, to calculate the expected number of rollouts before sampling SOF_6 we simply invert this probability.

[3]In the case of constructing $SOF_5 + O$, the plus signifies that the agent builds the molecule SOF_5 (Pentafluoro(oxido)-lambda6-sulfane) and places a separate, nearby oxygen atom.

a critical point as the culprit. The optimizer may not technically be stuck in that it continues to make meaningful updates to the network weights. However, we observe that the nearly certain odds of initially placing either 2 oxygen (when generating $SOF_5 + O$) or 2 sulfur (when generating S_2OF_5) create rollouts that sharply diverge from resembling SOF_6. When such rollouts come to dominate the training buffer, the network is optimized primarily with respect to them, and thus a vicious cycle emerges as the agent becomes entrenched generating more rollouts with 2 oxygen (or 2 sulfur) for training. This severely limits, if not outright prevents (depending on the buffer size), the agent from finding the optimal policy i.e., building SOF_6.

Figure 5.2 demonstrates how dominating suboptimal rollouts may induce a severely adverse training cycle. We observe that if the agent building $SOF_5 + O$ simply dropped the isolated oxygen atom, reward would increase significantly. It might seem easy for the agent to realize this simple improvement, at least relative to constructing SOF_6. However, recall the agent is optimized over rollout data collected in a training buffer. That buffer has come to consist largely of state-action-reward-state (SARS) tuples (s_t, a_t, r_t, s_{t+1}), where the state variables contain 2 canvas oxygens. One might presume this arises particularly because the 2 oxygens are placed first and thus most SARS samples contain states with 2 canvas oxygens. In fact, we randomize a rollout's order of atom placement before it is subsumed into the training buffer. But evidently this insufficiently mitigates the extent to which states with 2 canvas oxygens influence the buffer. Ultimately, the agent fails to learn to place just a single oxygen molecule and generate the higher rewarding SOF_5.

Notwithstanding the difficulties constructing SOF_6, we must remind ourselves that a key motivation for MolGym is to design novel molecular structures. That we set out to build SOF_6 (pentafluorosulfur hypofluorite) but ended up constructing other, similar molecules is perhaps even somewhat desirable. In the preceding experiments, we constructed two identifiable molecules: SOF_2 (Thionyl fluoride) and SOF_5 (pentafluoro(oxido)-lambda6-sulfane). The chemical formula for the final molecule (S_2OF_5) could not be found in the National Institute for Standard and Technology Chemistry WebBook (Linstrom and Mallard, 2001) or PubChem (Kim et al., 2021) databases. However, its training run was intentionally stopped early when the buffer became overwhelmed by suboptimal rollouts and the agent deemed unlikely to build molecules with a single S and thus learn SOF_6. Moreover, higher gradient norms suggest the networks are still making significant updates. In turn, though the agent is unlikely to recover to build SOF_6, it may still discover other structures. Regardless, it

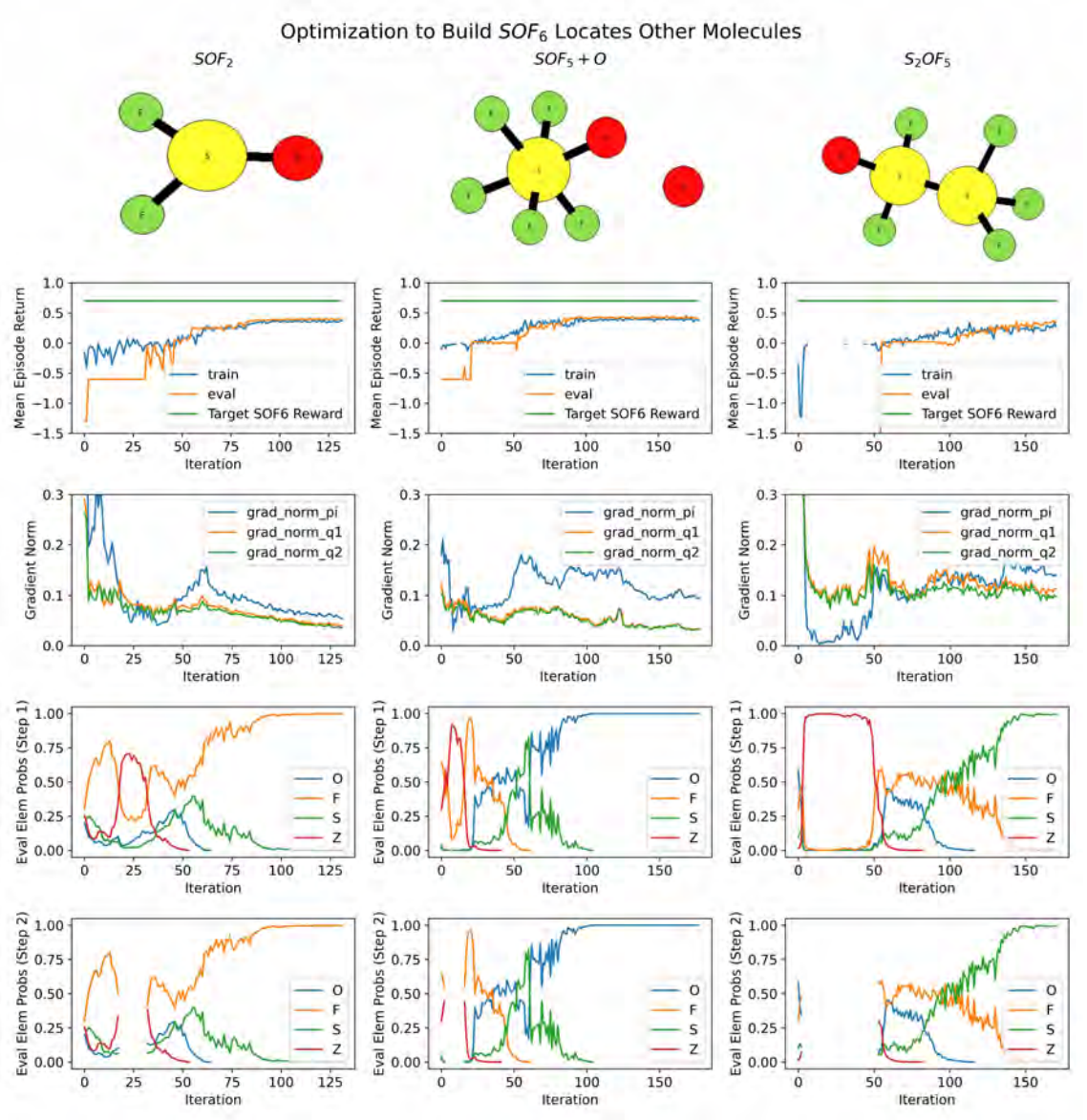


Fig. 5.1 Agents trained to build pentafluorosulfur hypofluorite (SOF_6) get stuck over the course of optimization constructing other moleculues: SOF_2, $SOF_5 + O$, and S_2OF_5. In all cases, the agent could hypothetically obtain a higher reward of ~ 0.7 if optimized correctly to build pentafluorosulfur hypofluorite. The top figures display a typical structure built in evaluation. The plots below show the reward obtained over the course of training along with the hypothetical target reward for building SOF_6. The next row contains the gradient norms for the q functions and policy networks over the course of training. Note that the shrunken norms for the SOF_2 run suggests the optimizer is stuck around a critical point. In the final 2 rows, we show the evaluation element probabilities for the first 2 steps of a rollout. Evidently, the $S_2OF_5 + O$ and S_2OF_5 runs learn to first place 2 oxygen or 2 sulfur, respectively, with near certainty. This is problematic as the optimizer may become stuck in a vicious cycle generating and training on suboptimal rollouts.

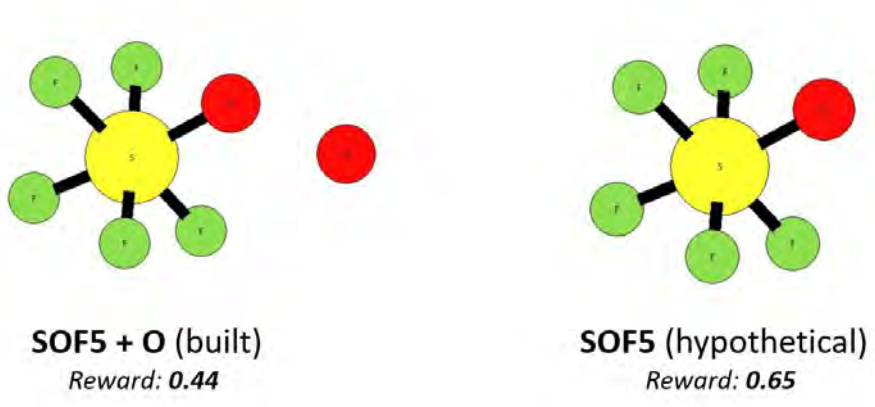


Fig. 5.2 Comparison of reward for building $SOF_5 + O$ versus simply SOF_5. Although the agent may achieve a substantially higher reward by just constructing SOF_5, the agent continues to build $SOF_5 + O$ as its training buffer is overwhelmed by rollouts containing 2 oxygen atoms.

is not necessarily problematic that the agent has learned to construct an unrecognized molecule. After all, MolGym is intended to discover new molecules.

MolGym thus retain some utility even when the optimizer fails to locate the global minima. Still, the optimization complexity is quite troubling for MolGym's infinite bag setting when applied to larger molecules. Moreover, optimization difficulties may subvert even simple molecules when penalties are not, or can not be, suitably discriminative.

5.1.2 Constraints on Penalty Formulation

We've previously discussed the importance of carefully constructing atom penalties to both accord to the scale of the energy reward objective and incentive the formation of desired structures (see section 3.3). Tuning penalties to more explicitly discriminate desirable and undesirable structures is often helpful for optimization. However, sometimes appropriate penalties are necessarily constrained to a relatively narrow range and may not as sharply guide what to construct. We will illustrate this with a practical example.

Earlier we successfully built H_2O by raising the penalty on hydrogen after 2 hydrogens were placed on the canvas and on oxygen after 1 oxygen was placed on the canvas. The actual penalties values were in particular selected so that we incentived the formation of H_2O over other molecules (H_2,O_3, etc..). Now consider training an agent to build multiple H_2O molecules. A sensible approach is to first set the hydrogen penalty to half the energy dip of H_2 so that the agent is not encouraged to just build

H_2. Then we need to set the oxygen penalty so that the reward for building H_2O exceeds that of other molecules. The challenge now is that we can't just raise the penalty on oxygen after a single O is placed because that would discourage building additional H_2O molecules. Fortunately, as evident in figure 5.3, appropriate initial penalties on oxygen exists that incentive the formation of H_2O over other molecules. However, figure 5.3 also shows that the range of appropriate penalties is narrow and all provide the agent only a limited reward boost for preferring to build H_2O. This renders the H_2O optimum difficult to locate as alternative molecules (e.g., O_3) provide a close, if inferior reward. In fact, if the agent has learned to optimize the geometric configuration of an alternative like O_3, but not H_2O, the reward for O_3 may actually be higher than a typical rollout of H_2O. All in all, this creates optimization difficulties that may prevent learning even simple structures when the penalty formulation is adversely constrained. To exemplify this challenge, we trained 35 agents with the H_2O penalty schedule described above and only 2 eventually achieved the optimal reward by building H_2O.

5.2 Optimization Support Strategies

In this section we describe two strategies that were implemented to aid optimization: 1) initial probability specification 2) leveraging expert rollouts.

5.2.1 Initial Probability Specification

Before we may begin training, we must collect random rollouts to initially fill the training buffer. Typically, we collect 10-100x more rollouts during the first iteration than in subsequent iterations. This is to ensure right from the start of training that the buffer is sufficiently large and contains suitably diverse experience.

To help guide the learning procedure towards an intended molecule (or class of molecules), we can overwrite the element probability distributions for the first iteration's rollouts. That is, instead of letting the agent select elements randomly we may specify, for example, that S is chosen with probability 70% and O, F, Z each with probability 10%. In turn, we are far more likely to sample rollouts that construct molecules like SOF_6 into the initial training buffer. This helps optimization avoid some local pitfalls by shifting network updates to weigh more on transitions from SOF_6 like rollouts. In particular, by defining the initial rollout S, O, F, Z probabilities as above we were able to train an agent that built SOF_4 (Thionyl tetrafluoride) and consequently achieved a

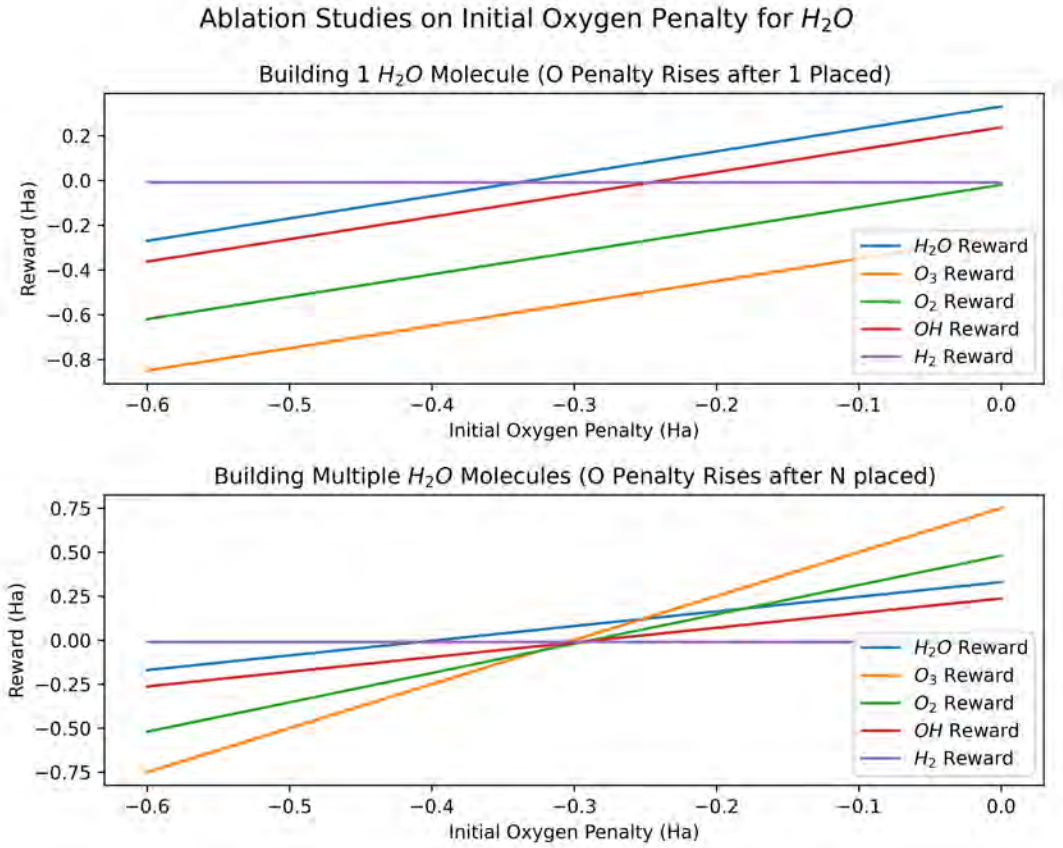


Fig. 5.3 Ablation studies on initial oxygen penalty for H_2O. Each plot shows reward that an agent would achieve in building different molecules subject to different initial oxygen penalties. In the top plot, the penalty schedule involves increasing the initial penalty on oxygen after a single oxygen is placed. In the bottom plot, we aim to create multiple H_2O molecules and thus the oxygen penalty is not increased until some arbitrary point in the future. Evidently, the range of appropriate initial oxygen penalties (i.e., those that favor the construction of H_2O) is considerably more narrow in the bottom plot and the reward differential therein more limited.

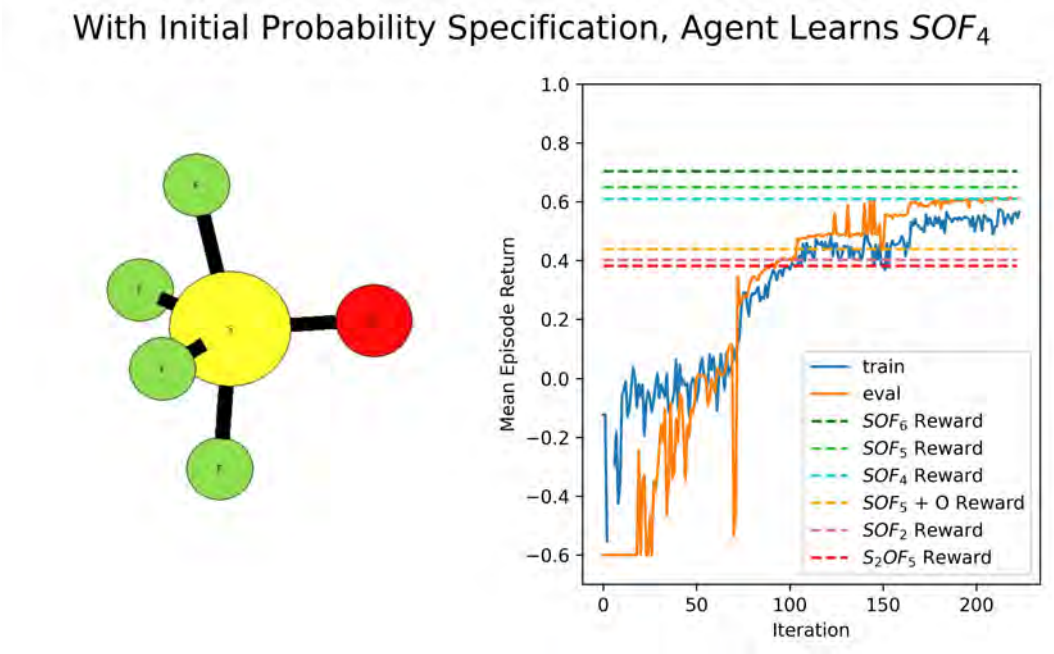


Fig. 5.4 A MolGym agent trained with penalties to incentive SOF_6 construction is additionally guided by initial rollout element probabilities (S=70%, O=10%, F=10%, Z=10%). The trained agent is able to outperform previous attempts by building SOF_4. In particular, we compare the superior SOF_4 reward to a sample of outcomes obtained without specifying the initial rollout element probabilities: SOF_2, S_2OF_5, and $SOF_5 + O$ (for additional detail see figure 5.1). Nevertheless, the agent still fails to learn to construct SOF_6 or SOF_5 when those molecules would yield an even higher reward.

significantly higher reward than was obtained by previous agents using random initial probabilities (see figure 5.4). Nonetheless, the agent still failed to build SOF_6, or even SOF_5, when both constructions could provide a notably higher reward. To further support the agent in learning to construct SOF_6, we expand on the idea of priming the training buffer and introduce expert rollouts.

5.2.2 Leveraging Expert Rollouts

Motivation

By specifying initial element rollout probabilities, practitioners may nudge the training buffer to include more rollouts assembling target molecules (e.g., SOF_6). But the practitioner's influence remains fairly limited. Consider that in specifying the probabilities S=70%, O=10%, F=10%, and Z=10%, we only expect to sample the atoms SOF_6

every 152 rollouts.[4] Furthermore, we emphasize that such rollouts merely contain the atoms SOF_6 and very improbably actually geometrically resemble the molecule SOF_6 (Pentafluorosulfur hypofluorite) on the canvas.

Practioners may exact considerably more sway over the composition of the training data by instead pre-filling the buffer with expert demonstrations. That is, instead of training just on the rollouts generated by the agent over the course of training, we also train on a set of expert rollouts that correctly assemble a target molecule e.g., SOF_6. This procedure may appear to assume that we've already managed to build the target molecule. However, our implementation relies on the practitioner supplying just a 3D representation of the molecule (i.e., the identities and locations of each atom in 3D space) and thus the data for expert demonstrations may be sourced from outside MolGym. Moreover, often a practitioner's intention is not so much to learn to build a known molecule like SOF_6, but discover alike but novel structures. In this case, we may pretrain with expert demonstrations to help direct the optimizer towards a desired region of parameter space before eventually unleashing the optimizer to explore unconstrained.

Imitation Learning and SQIL

Our approach to leveraging expert rollouts is related to the field of imitation learning and specifically Soft Q Imitation Learning (SQIL) (Reddy, Dragan, and Levine, 2019). Whereas agents are trained to maximize an extrinsic reward function in reinforcement learning, imitation learning concerns training agents to imitate expert behavior (*Imitation Learning Lecture Notes; CS234 Stanford University* 2021). A basic approach to imitation learning is Behavior Cloning (BC) which involves using supervised learning methods to minimize the discrepancy between an expert's demonstrated actions and those selected by the learned policy. Unfortunately, BC often performs poorly as the learned policy usually can not reason well about states outside the distribution demonstrated by the expert. This is problematic as the agent may easily drift away from the demonstrated states due to compounding errors. An approach to mitigate the drawbacks of BC that also integrates well with the soft actor-critic algorithm is Soft Q Imitation Learning (SQIL) (Reddy, Dragan, and Levine, 2019). SQIL, which may be theoretically motivated as a regularized variant of BC, incentives the agent to not

[4]The probability of randomly sampling $SOF6$ is $\frac{7}{10}^6 * \frac{1}{10}^3 * 8$ [options for step that S is chosen] $*$ 7 [options for step that O is chosen] . Note that the step the STOP atom Z is chosen is fixed (the final step) and the steps F is chosen are thereby determined to the 6 remaining steps. Then, to calculate the expected number of rollouts before sampling SOF_6 we simply invert this probability.

only imitate the expert in demonstrated states, but also select actions that return the agent to demonstrated states should it stray. Practically, SQIL involves modifying the SAC algorithm so that the buffer is initially filled with transitions tuples (s_t, a_t, r_t, s_{t+1}) from expert rollouts and each training step involves a balanced sample of expert and non-expert (meaning agent generated) transitions where all expert transitions provide a reward of $r = 1$ and all non-expert transitions provide a reward of $r = 0$.

At first, we incorporated SQIL unmodified into MolGym but obtained underwhelming results. However, we may both boost performance and broaden the usefulness of expert demonstrations by altering SQIL so that we still leverage an extrinsic reward function. Specifically, we embrace the ideas of prefilling the training buffer with expert rollouts and ensuring every training step balances expert and non-expert transitions.[5] However, we choose to keep our reward function (equation 3.1), consisting of the energy reward plus penalties, instead of adopting the binary 1/0 rewards for expert/non-expert transitions. Why? Put simply, we have a meaningful extrinsic reward function and should avail ourselves of it. Usually imitation learning problems are framed such that an agent must learn without an extrinsic reward function and just from expert demonstrations. But since we have a well-motivated reward function, it can make sense to incorporate it. In particular, by using an extrinsic reward function, the agent may learn beyond imitating the expert. Specifically, the agent may begin by pretraining on expert/non-expert transitions but, at some trigger point, switch to training only on non-expert transitions (i.e., the standard SAC training procedure). With unmodified SQIL, in contrast, the binary rewards compel the agent to only imitate the expert. If training is eventually switched to a regime with our extrinsic reward function than there is a mismatch in rewards with the pretraining stage that subverts further optimization.

As we don't exactly perform SQIL we lose its theoretical guarantees. However, we maintain the two key ideas that motivate SQIL's approach. First, our method still integrates information on transition dynamics into the policy by training with RL. This helps avoid suboptimally greedy action selection as under BC. Second, whereas SQIL regularizes implicit rewards with a sparsity prior, our method incorporates an explicit sparse reward function.

To summarize, we implemented an expert pretraining procedure that essentially involves ensuring each gradient update minibatch balances transitions from expert and non-expert rollouts. We can motivate our approach both as extending the idea of specifying initial probabilities to more deftly design the training buffer and as modifying Soft Q Imitation Learning so to still make use of an extrinsic reward signal.

[5] Note the exact balance need not be 1:1 and may be configured via a hyperparameter

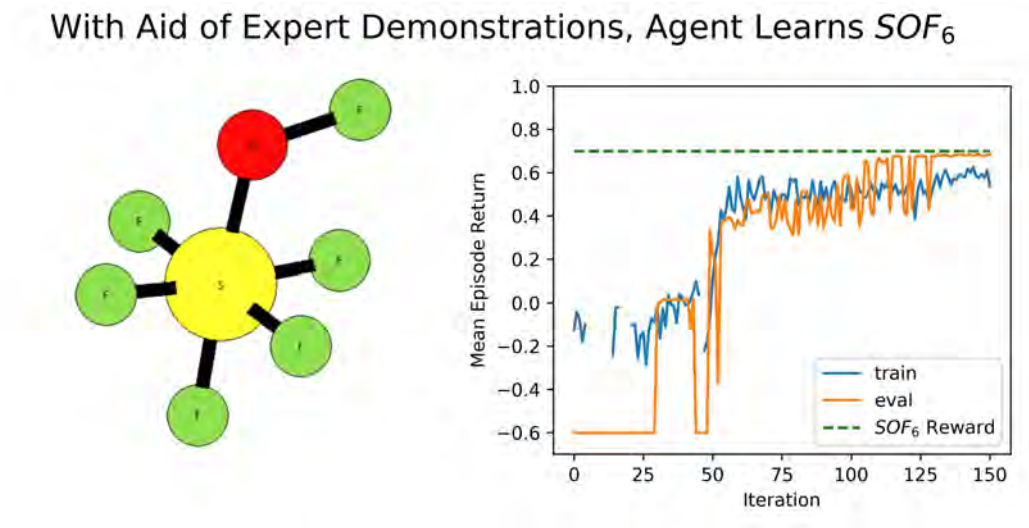


Fig. 5.5 We successfully train an agent to construct SOF_6 by modifying the training algorithm to exploit expert rollouts (for details on our technique see section 5.2.2). On the left, we show an SOF_6 molecule produced during evaluation. At right, we display mean training and evaluation rewards over the course of training along with the optimal reward for building SOF_6.

Results

At last, by leveraging expert demonstrations, we successfully trained an infinite bag MolGym agent that constructed SOF_6 (see figure 5.5). Specifically, we first collected several hundred approximately optimal 3D configurations of SOF_6 by training a finite bag MolGym agent. Next, from those SOF_6 molecules we devised expert rollout trajectories and built a supplementary training set consisting of expert rollout transitions. Finally, for each gradient update we ensured mini-batches balanced expert and non-expert transitions in a 1:2 ratio (although other ratios are possible).

Notwithstanding success in building SOF_6, some optimization challenges persist. In figure 5.6 we display several training runs that by using our expert training procedure manage to construct molecules resembling SOF_6. However, these runs all appear to converge slightly below the optimal SOF_6 reward. In examining the actual 3D molecules built we observe that they closely, but perhaps not perfectly, resemble SOF_6. This may simply be the result of the temperature configuration (recall that our training objective balances both maximizing episode reward and maximizing entropy) but it is nonetheless a troubling optimization detail that agents may nearly, but not quite perfectly, learn a molecule's ideal geometry. As a practical matter, this is less concerning than another optimization obstacle: model brittleness.

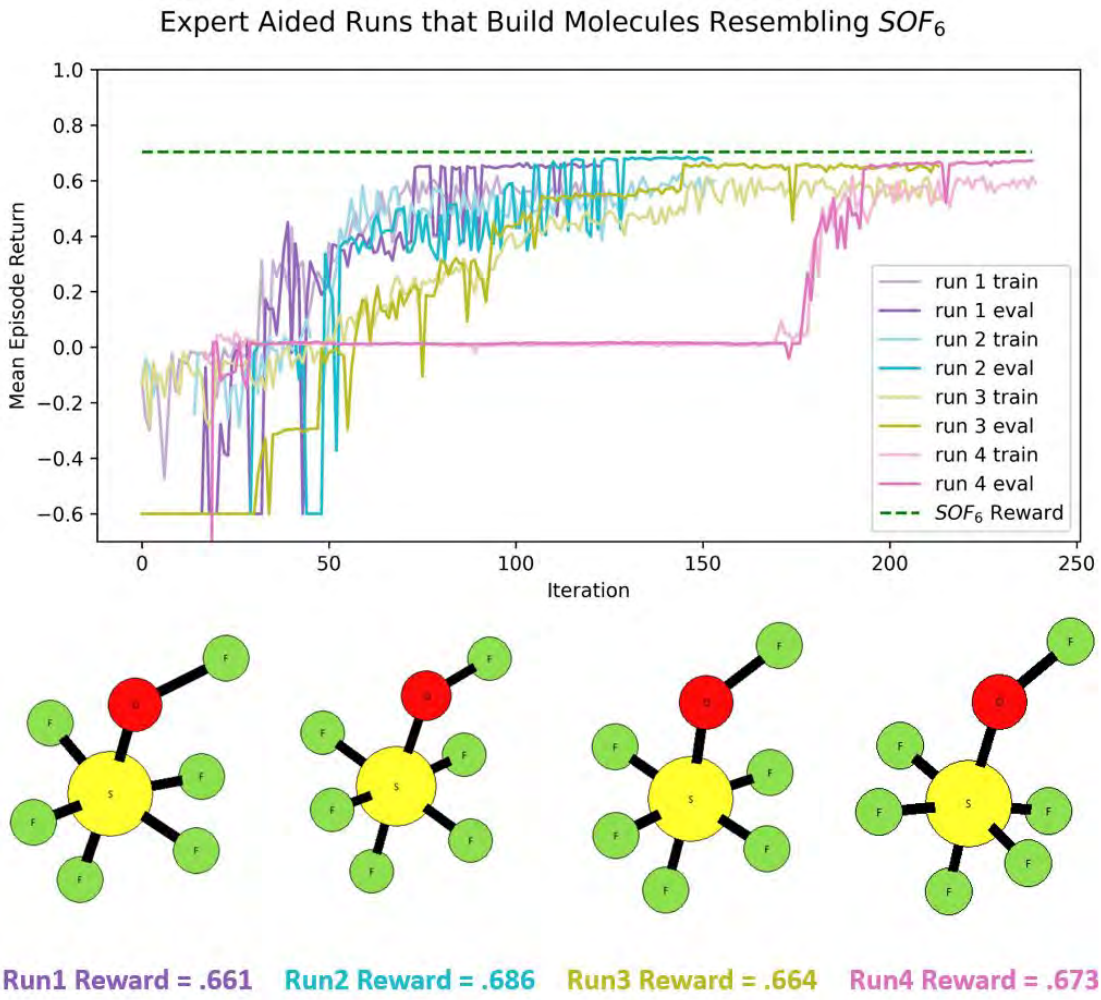


Fig. 5.6 Applying our expert training procedure, we generate several runs that successfully train agents to construct molecules resembling SOF_6. However, the runs' evaluation rewards converge to levels that are slightly suboptimal and the actual 3D molecules (as generated during evaluation) closely, but not always perfectly, resemble SOF_6.

We've discussed and plotted successful SOF_6 runs but must note that a majority either produced other (suboptimal) molecules or broke under exploding gradients. That is, significantly different outcomes result from varying hyperparamters or even just setting new random seeds. This is at least partly because the random sample of initial rollout varies substantially between runs. If we don't collect enough suitably diverse rollouts then training fixates on a small, concentrated dataset and can destabilize. In particular, because we discard rollouts that fall below a minimum reward[6] we may randomly end up with relatively few initial rollouts. Supplementing these rollouts with expert demonstrations only goes so far as we can't arbitarily increase the relative proportion of expert to non-expert transitions. This is because the agent will initially just derive simplistic rules from the demonstrations (e.g., placing F is good) that destabilize training before the agent can learn the more complex mapping. Collecting enough interesting initial rollouts is thus still key to ensuring that early training proceeds smoothly.

To recap, we've developed an expert training procedure that allows MolGym to build more complex molecules with infinite bags. However, we still require carefully configured hyperparameters and typically several runs from different random seeds.

5.3 Optimization Summary

In this chapter, we've discussed why MolGym's infinite bag paradigm raises difficult optimization challenges. We also proposed two mitigation strategies. Most notably, we introduced a method similar to Soft Q Imitation Learning that leverages expert demonstrations and allowed us to, at last, train an agent that built SOF_6. Still however, we needed to scrupulously tune hyperparameters and run with several random seeds in order to yield compelling results.

[6]Discarding especially poor rollouts helps focus training updates on more worthwhile transitions. The minimum reward threshold may be configured by a hyperparameter.

Chapter 6

Conclusion and Future Directions

In this dissertation we enhanced MolGym (Simm, Pinsler, and Hernández-Lobato, 2020; Simm, Pinsler, Csányi, et al., 2021), a reinforcement learning approach to 3D molecular design, by developing a powerful new learning paradigm: the infinite atom bag. Under the infinite bag setting, MolGym agents are no longer constrained to construct molecules according to prespecified chemical formulae. Instead, agents *learn* the optimal size and composition of molecular structures as guided by practitioner determined per-atom penalties. To further empower infinite bag MolGym agents, we introduced randomized atom penalties and thereby enabled agents that respond to dynamic penalty conditions and construct more diverse molecules. Moreover, to mitigate model brittleness we implemented a variant of the soft-actor critic with automatic entropy adjustment (Haarnoja, A. Zhou, Hartikainen, et al., 2018). Although now able to train agents that construct small molecules (e.g., H_2O, O_3) with relative ease, optimization challenges persisted in scaling to larger molecules and building structures with insufficiently discriminative per-atom penalties. Thus to robustify optimization we incorporated a technique to leverage expert rollouts in the vein of Soft Q Imitation Learning (Reddy, Dragan, and Levine, 2019). By doing so, we managed to train an agent to construct SOF_6, which had heretofore proved an elusive target. All in all, we implemented several powerful new capabilities into MolGym through the infinite bag paradigm. However, to encourage practical adoption we must first address the limitations of our work.

Optimization brittleness still hinders the real-world use of MolGym's infinite bag paradigm. Although we managed to construct a more complex molecule (SOF_6) by leveraging our expert training procedure, it was necessary to tune hyperparameters extensively. In fact, just the random seed could decisively sway a run's ultimate performance through the (random) initial rollouts. Now it must be said that these optimization challenges don't outright prevent practitioners from adopting MolGym's

infinite bag paradigm as-is. Many machine learning problems require extensive hyperparameter tuning and starting optimization from different points (e.g., with different random seeds) is a tried and true technique. Moreover, insofar as a key objective of MolGym is to design novel molecular structures, it is not necessarilly problematic that an agent incentivized to construct SOF_6 sometimes learns to construct SOF_2 or SOF_4. Still, a typical practitioner may not be deeply familiar with tuning RL hyperparamters (which can be notoriously tricky) and thus to facilitate adoption we ought to further investigate ameliorating optimization. Four potential directions are integrating other imitation learning approaches, considering alternative optimizers, increasing the capacity of the network, and exploring more advanced hyperparameter tuning procedures.

Let's start by discussing optimizers. MolGym's optimizer, Adam with decoupled weight decay (Loshchilov and Hutter, 2017), is widely regarded as effective and relatively robust. But there are competitive alternatives like RMSProp (Hinton, Srivastava, and Swersky, 2012) and intriguing variants of Adam such as Adam with Nesterov momentum (NAdam) (Dozat, 2016) and Adam with rectified early training variance (RAdam) (Liu et al., 2019). Additionally, although Adam adjusts per-parameter learning rates, we do not currently decay the global learning rate / upper bound. It may thus be worthwhile to consider a global learning rate scheduler based on the epoch count or some validation metric. Now we must remark that, given the effectiveness of AdamW, we're skeptical how much improvement may arise from reconfiguring the optimizer. Nonetheless, it may still be worthwhile considering the ease of interchanging optimizers and configuring schedulers in PyTorch.

Another direction is to investigate incorporating other imitation learning algorithms. We made significant progress by integrating a modified version of Soft Q Imitation Learning (Reddy, Dragan, and Levine, 2019). This suggests that there's value in capitalizing on expert demonstrations to remedy optimization. Two additional imitation learning approaches that look compelling and appear to integrate with MolGym's soft-actor crtic architecture are Self-Imitation Learning (Oh et al., 2018) and IQ-Learn, Inverse soft-Q Learning for Imitation (Garg et al., 2021).

Future research could also consider integrating more elaborate hyperparamter tuning. The finite bag MolGym already involves quite a few sensitive hyperparamters and the infinite bag setting stacks up several more. Optimizing all these hyperparameters through grid or random search presents a significant computation burden. It's thus sensible to explore Bayesian or bandit based approaches (e.g. L. Li et al., 2017) that may help direct the hyperparameter search and hopefully uncover more robust settings.

Lastly, optimization troubles may stem from the neural network no longer possessing sufficient capacity to succeed on the more intricate modelling challenge posed by infinite bags. In scaling to more complex molecules, we widened several fully-connected layers in both the policy and Q networks. Further research could explore deepening the networks, particularly the MLPs that integrate the bag tensor into the model. Doing so could help establish if network capacity is bottlenecking optimization on more complex modeling objectives.

Beyond robustifying optimization, future work on MolGym should follow the aims of practitioners. We've demonstrated MolGym's infinite bag paradigm capable of training agents on several molecule targets (e.g., H_2O, SOF_6, O_2 vs. O_3). Now research should move beyond proof-of-concept to more practical use cases. For instance, it may be interesting to build out MolGym's facilities for constructing repeated structures. Note this is now only possible because of the infinite bag setting. It would also be intriguing to explore using MolGym to assemble molecules that fit specific protein bonding sites. Recent research has taken a supervised approach (Luo et al., 2021) and we're curious how RL with MolGym might compare. Specifically, we might suspect an RL agent more capable at generalization. The infinite bag setting is again of great use as agents are no longer constrained to form molecules according to pre-specified chemical formulae. Instead, agents may learn to assemble molecules with a size and configuration that fits the bonding site.

To conclude, I hope you enjoyed reading this dissertation and learning about the exciting frontiers of molecular design with deep reinforcement learning.

References

Anderson, Brandon, Truong Son Hy, and Risi Kondor (2019). "Cormorant: Covariant molecular neural networks". In: *Advances in neural information processing systems* 32.

Bishop, Christopher M. (1994). *Mixture density networks*. Tech. rep.

Bosia, Francesco et al. (Dec. 2021). *qcscine/sparrow: Release 2.0.1*. Version 2.0.1. DOI: 10.5281/zenodo.5782828. URL: https://doi.org/10.5281/zenodo.5782828.

Curie, Pierre (1894). "Sur la symétrie dans les phénomènes physiques, symétrie d'un champ électrique et d'un champ magnétique". In: *Journal de physique théorique et appliquée* 3.1, pp. 393–415.

De Cao, N and Kipf Thomas (2018). "An implicit generative model for small molecular graphs. arXiv preprint 2018". In: *arXiv preprint arXiv:1805.11973* 3.

Dozat, Timothy (2016). "Incorporating nesterov momentum into adam". In.

Garg, Divyansh et al. (2021). "IQ-Learn: Inverse soft-Q Learning for Imitation". In: *Advances in Neural Information Processing Systems* 34, pp. 4028–4039.

Geiger, Mario et al. (Apr. 2022). *Euclidean neural networks: e3nn*. Version 0.5.0. DOI: 10.5281/zenodo.6459381. URL: https://doi.org/10.5281/zenodo.6459381.

Gómez-Bombarelli, Rafael et al. (2018). "Automatic chemical design using a data-driven continuous representation of molecules". In: *ACS central science* 4.2, pp. 268–276.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep Learning*. http://www.deeplearningbook.org. MIT Press.

Haarnoja, Tuomas, Haoran Tang, et al. (2017). "Reinforcement learning with deep energy-based policies". In: *International Conference on Machine Learning*. PMLR, pp. 1352–1361.

Haarnoja, Tuomas, Aurick Zhou, Pieter Abbeel, et al. (2018). "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor". In: *International conference on machine learning*. PMLR, pp. 1861–1870.

Haarnoja, Tuomas, Aurick Zhou, Kristian Hartikainen, et al. (2018). "Soft actor-critic algorithms and applications". In: *arXiv preprint arXiv:1812.05905*.

Hinton, Geoffrey, Nitish Srivastava, and Kevin Swersky (2012). "Neural networks for machine learning lecture 6a overview of mini-batch gradient descent". In: *Cited on* 14.8, p. 2.

Husch, Tamara, Alain C Vaucher, and Markus Reiher (2018). "Semiempirical molecular orbital models based on the neglect of diatomic differential overlap approximation". In: *International journal of quantum chemistry* 118.24, e25799.

Imitation Learning Lecture Notes; CS234 Stanford University (Feb. 2021).

Kim, Sunghwan et al. (2021). "PubChem in 2021: new data content and improved web interfaces". In: *Nucleic acids research* 49.D1, pp. D1388–D1395.

Kingma, Diederik P and Jimmy Ba (2014). "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980*.

Klicpera, Johannes, Janek Groß, and Stephan Günnemann (2020). "Directional message passing for molecular graphs". In: *arXiv preprint arXiv:2003.03123*.

Konda, Vijay and John Tsitsiklis (1999). "Actor-critic algorithms". In: *Advances in neural information processing systems* 12.

Kullback, Solomon and Richard A Leibler (1951). "On information and sufficiency". In: *The annals of mathematical statistics* 22.1, pp. 79–86.

Larsen, Ask Hjorth et al. (2017). "The atomic simulation environment—a Python library for working with atoms". In: *Journal of Physics: Condensed Matter* 29.27, p. 273002.

Li, Lisha et al. (2017). "Hyperband: A novel bandit-based approach to hyperparameter optimization". In: *The Journal of Machine Learning Research* 18.1, pp. 6765–6816.

Lillicrap, Timothy P et al. (2015). "Continuous control with deep reinforcement learning". In: *arXiv preprint arXiv:1509.02971*.

Linstrom, Peter J and William G Mallard (2001). "The NIST Chemistry WebBook: A chemical data resource on the internet". In: *Journal of Chemical & Engineering Data* 46.5, pp. 1059–1063.

Liu, Liyuan et al. (2019). "On the variance of the adaptive learning rate and beyond". In: *arXiv preprint arXiv:1908.03265*.

Loshchilov, Ilya and Frank Hutter (2017). "Decoupled weight decay regularization". In: *arXiv preprint arXiv:1711.05101*.

Luo, Shitong et al. (2021). "A 3D generative model for structure-based drug design". In: *Advances in Neural Information Processing Systems* 34, pp. 6229–6239.

Maei, Hamid et al. (2009). "Convergent temporal-difference learning with arbitrary smooth function approximation". In: *Advances in neural information processing systems* 22.

Mnih, Volodymyr et al. (2013). "Playing atari with deep reinforcement learning". In: *arXiv preprint arXiv:1312.5602*.

Oh, Junhyuk et al. (2018). "Self-imitation learning". In: *International Conference on Machine Learning*. PMLR, pp. 3878–3887.

Olivecrona, Marcus et al. (2017). "Molecular de-novo design through deep reinforcement learning". In: *Journal of cheminformatics* 9.1, pp. 1–14.

Paszke, Adam et al. (2019). "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

Polishchuk, Pavel G, Timur I Madzhidov, and Alexandre Varnek (2013). "Estimation of the size of drug-like chemical space based on GDB-17 data". In: *Journal of computer-aided molecular design* 27.8, pp. 675–679.

Reddy, Siddharth, Anca D Dragan, and Sergey Levine (2019). "Sqil: Imitation learning via reinforcement learning with sparse rewards". In: *arXiv preprint arXiv:1905.11108*.

Schulman, John et al. (2015). "Trust region policy optimization". In: *International conference on machine learning*. PMLR, pp. 1889–1897.

Simm, Gregor N. C., Robert Pinsler, Gábor Csányi, et al. (2021). "Symmetry-Aware Actor-Critic for 3D Molecular Design". In: *International Conference on Learning Representations*. URL: https://openreview.net/forum?id=jEYKjPE1xYN.

Simm, Gregor N. C., Robert Pinsler, and José Miguel Hernández-Lobato (2020). "Reinforcement learning for molecular design guided by quantum mechanics". In: *International Conference on Machine Learning*. PMLR, pp. 8959–8969.

Stewart, James JP (2007). "Optimization of parameters for semiempirical methods V: Modification of NDDO approximations and application to 70 elements". In: *Journal of Molecular modeling* 13.12, pp. 1173–1213.

Sutton, Richard S and Andrew G Barto (2018). *Reinforcement learning: An introduction*. MIT press.

Tsitsiklis, John and Benjamin Van Roy (1996). "Analysis of temporal-diffference learning with function approximation". In: *Advances in neural information processing systems* 9.

Williams, Ronald J (1992). "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Machine learning* 8.3, pp. 229–256.

You, Jiaxuan et al. (2018). "Graph convolutional policy network for goal-directed molecular graph generation". In: *Advances in neural information processing systems* 31.

Zhou, Zhenpeng et al. (2019). "Optimization of molecules via deep reinforcement learning". In: *Scientific reports* 9.1, pp. 1–10.

Appendix A

Supplementary Plots

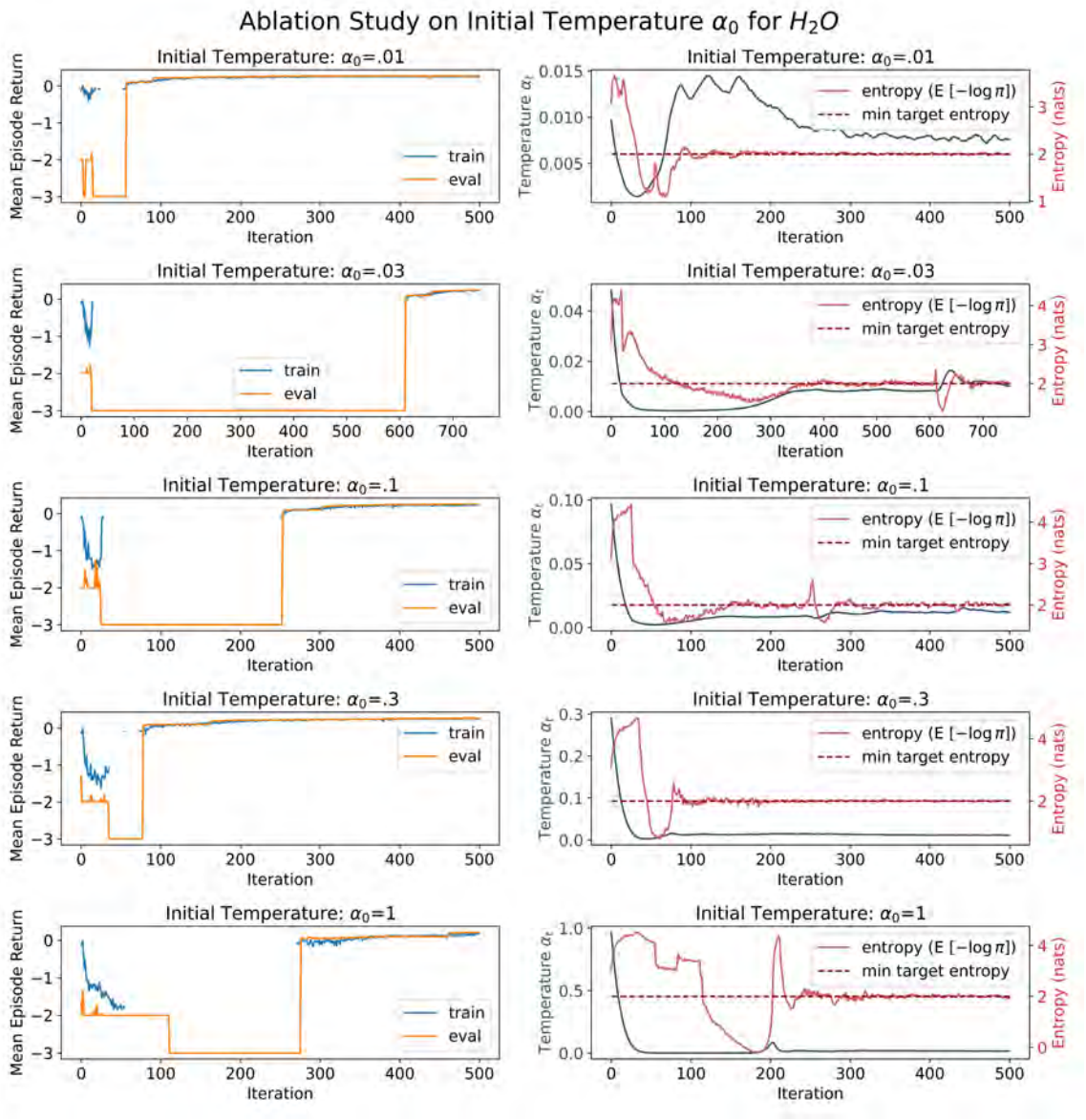


Fig. A.1 Ablation study on the initial temperature α_0 hyperparameter for SAC with automatic entropy adjustment (Haarnoja, A. Zhou, Hartikainen, et al., 2018) as applied to the construction of H_2O in MolGym's infinite bag setting. The auto-tuned temperature algorithm appears more robust than the original SAC fixed temperature implementation wherein the temperature demanded careful tuning to find the correct policy (see figures 3.4, 3.8). In contrast, all runs with an auto-tuned temperature learned the correct policy. Nonetheless performance, as measured by iterations to convergence, varied considerably and rather unpredictably across α_0 settings. For example, the run with $\alpha_0 = .03$ took an extremely long time to learn the correct policy, while runs with both higher *and* lower initial temperatures converged much earlier. Note all runs used a minimum target entropy $\overline{\mathcal{H}} = 2$.

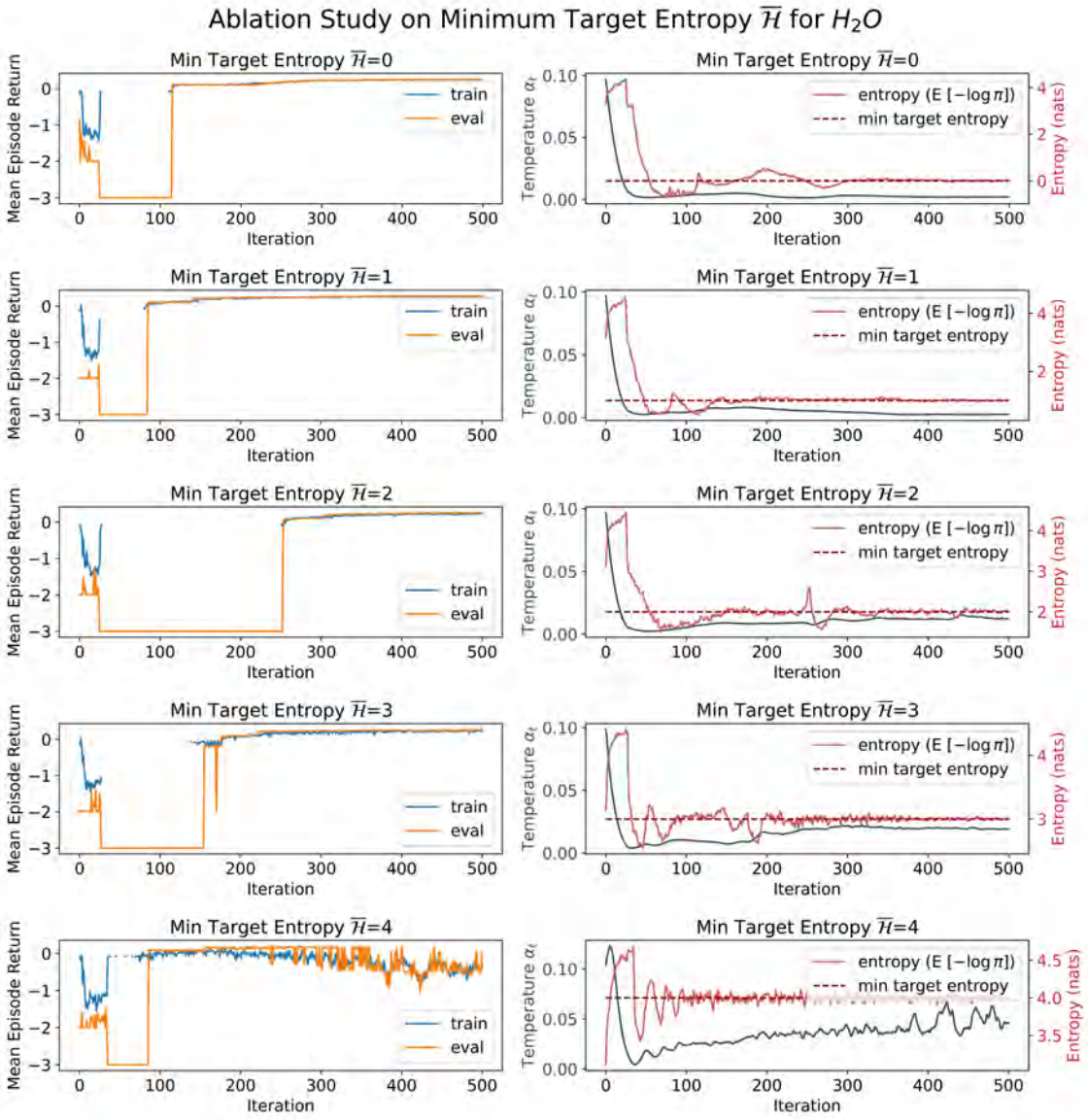


Fig. A.2 Ablation study on the minimum target entropy $\overline{\mathcal{H}}$ for SAC with automatic entropy adjustment (Haarnoja, A. Zhou, Hartikainen, et al., 2018) as applied to the construction of H_2O in MolGym's infinite bag setting. The auto-tuned temperature appears more robust than the original SAC fixed temperature implementation wherein the temperature demanded careful tuning to find the correct policy (see figures 3.4, 3.8). Only a single run with auto-tuned temperature failed to find the correct policy (due to excessively high minimum target entropy). Nonetheless performance, as measured by iterations to convergence, varied considerably and rather unpredictably across $\overline{\mathcal{H}}$ settings. Note all runs used a initial temperature of $\alpha_0 = .01$.