Discrete and Continuous Action Representation for Practical RL in Video Games

Olivier Delalleau[*1], **Maxim Peter**[*1], **Eloi Alonso**[1], **Adrien Logut**[1]
[1]Ubisoft La Forge

Abstract

While most current research in Reinforcement Learning (RL) focuses on improving the performance of the algorithms in controlled environments, the use of RL under constraints like those met in the video game industry is rarely studied. Operating under such constraints, we propose Hybrid SAC, an extension of the Soft Actor-Critic algorithm able to handle discrete, continuous and parameterized actions in a principled way. We show that Hybrid SAC can successfully solve a high-speed driving task in one of our games, and is competitive with the state-of-the-art on parameterized actions benchmark tasks. We also explore the impact of using normalizing flows to enrich the expressiveness of the policy at minimal computational cost, and identify a potential undesired effect of SAC when used with normalizing flows, that may be addressed by optimizing a different objective.

Introduction

Reinforcement Learning (RL) applications in video games have recently seen massive advances coming from the research community, with agents trained to play Atari games from pixels (Mnih et al. 2015) or to be competitive with the best players in the world in complicated imperfect information games like DOTA 2 (OpenAI 2018) or StarCraft II (Vinyals et al. 2019a; 2019b). These systems have comparatively seen little use within the video game industry, and we believe lack of accessibility to be a major reason behind this. Indeed, really impressive results like those cited above are produced by large research groups with computational resources well beyond what is typically available within video game studios.

Our contributions are geared towards industry practitioners, by sharing experiments and practical advice for using RL with a different set of constraints than those met in the research community. For example, in the industry, experience collection is usually a lot slower, and there are time budget constraints over the runtime performance of RL agents. We thus favor off-policy algorithms to improve data efficiency by re-using past experience, and constrain our architectures to relatively small feedforward networks. The approach we propose in this paper is based on Soft Actor-Critic (Haarnoja et al. 2018b), which was originally designed for continuous actions problems. We explore ways to extend it to a hybrid setting with both continuous and discrete actions, a situation commonly encountered in video games. We also attempt to use normalizing flows (Rezende and Mohamed 2015) to improve the quality of the resulting policy with roughly the same number of parameters, and analyze why this approach may not be working as well as we initially expected.

[*]Equal contribution, alphabetical order

Background and related work

We consider the classical Markov Decision Process (MDP) setting where at each discrete time step t the agent observes a state s_t and must take an action $a_t \sim \pi(a_t|s_t)$, where π is the agent's policy. On the next time step, the environment transitions to the new state $s_{t+1} \sim P(s_{t+1}|s_t, a_t)$ and gives the agent a reward $r_t \sim P(r_t|s_t, a_t, s_{t+1})$. The agent's objective is to find an optimal policy π^* that maximizes the expected discounted return $\mathbb{E}_\pi\big[\sum_t \gamma^t r_t\big]$, where $\gamma \in [0, 1]$ is the discount factor.

In the following, we assume that a state is represented by a real-valued vector, in a format suitable to be provided as input to a neural network (e.g. with one-hot encoding of discrete state variables, and normalization of continuous features). Actions may be either discrete, continuous, or a mix of both: a key contribution of this paper is to present a simple generic approach to action representation, suitable for most situations one may encounter when training game-playing agents.

Soft Actor-Critic

Soft Actor-Critic (SAC) (Haarnoja et al. 2018b; 2018c) is a state-of-the-art model-free algorithm that was originally proposed for continuous control tasks. It is based on the idea of adding an entropy bonus to the objective optimized by the agent, i.e. maximizing $\mathbb{E}_\pi\Big[\sum_t \gamma^t\big(r_t + \alpha\mathcal{H}(\pi(\cdot|s_t))\big)\Big]$. A higher α encourages the agent to take actions that are more random, which in particular can help with exploration. This α parameter can be learned during training by setting a target entropy for the policy (Haarnoja et al. 2018c).

Normalizing flows

Normalizing flows (Rezende and Mohamed 2015) are invertible transformations applied on top of an initial distribution to transform it into another distribution, usually with the goal of making it more expressive. The original SAC (Haarnoja et al. 2018b) parameterizes the actor using a spherical Gaussian and uses the reparameterization trick to backpropagate through the parameters of the distribution. It is possible to apply normalizing flows on top of this Gaussian policy to make it more expressive (Mazoure et al. 2019), while still being able to sample from the policy as well as compute the log-density at any point. This makes it possible to use normalizing flows to reparameterize the actor in SAC to get more complex policies while keeping the training algorithm unchanged.

(Tang and Agrawal 2018) show that using an Inverse Autoregressive Flow (IAF) for on-policy trust region policy optimization can significantly improve exploration in high-dimensional tasks. (Ward, Smofsky, and Bose 2019) use Real-valued Non Volume Preserving (Real NVP) flows to improve exploration in sparse reward settings, while (Haarnoja et al. 2018a) use Real NVP flows to train maximum entropy policies in a hierarchical setting where each layer is trained on its own reward function. Our experiments with normalizing flows are similar to and inspired by (Mazoure et al. 2019), who suggest that normalizing flows can be used to improve the expressiveness of policies in SAC to get a policy with the same level of quality using less parameters.

As suggested in (Mazoure et al. 2019), we use radial flows in our experiments. We sample $\varepsilon \sim \mathcal{N}(0,1)$ (since we use the reparameterization trick), and denote by h_θ the function returning a sample from a Gaussian distribution with mean and standard deviation given by the policy parameterized by θ. With $\{f_{\phi_i}\}_{i=1}^N$ the set of normalizing flows, we can sample from the policy as follows:

$$\begin{aligned} w_0 &= h_\theta(\varepsilon, s_t) \\ w_i &= f_{\phi_i} \circ f_{\phi_{i-1}} \circ ... \circ f_{\phi_1}(w_0) \\ a_t &= \tanh(w_N) \end{aligned}$$

We denote by q_0 the density of the state-dependent Gaussian distribution w_0 is sampled from. The density of the policy is then tractable according to:

$$\log \pi(a_t, s_t) = \log q_0(w_0) - \sum_{i=1}^{N} \log \left| \det \frac{\partial f_{\phi_i}(w_{i-1})}{\partial w_{i-1}} \right| \quad (1)$$

The equations corresponding to the radial flows are taken from (Rezende and Mohamed 2015) and can be found in the Appendix (Table 3).

Mixing discrete and continuous actions

Most reinforcement learning research papers focus on environments where the agent's actions are either discrete or continuous. However, when training an agent to play a video game, it is common to encounter situations where actions have both discrete and continuous components. Typical examples include:

- Playing with the same inputs as a player, whose controller may be equipped with both an analog stick (providing a range of continuous values) and buttons that can be pressed (yielding potentially many discrete actions through the various button combinations).
- Letting the agent choose among a set of high-level discrete actions (ex: move, jump, fire), each of them being associated with continuous parameters (ex: target coordinates for the move action, direction for the jump action, aiming angle for the fire action).
- Wanting the agent to control systems that have both discrete and continuous components, like driving a car by combining steering and acceleration (both continuous) with usage of the hand brake (a discrete binary action).

Such situations require algorithms that are able to handle a combination of discrete and continuous actions. In what follows, we propose a parameterization of the policy that can be easily implemented in SAC, yielding a powerful generic off-policy RL algorithm for training game-playing agents.

In order to deal with a mix of discrete and continuous action components, a first approach would be to use a fully continuous actor and somehow find a way to convert part of its continuous output into discrete actions (van Hasselt and Wiering 2009; Hausknecht and Stone 2016; Cianflone et al. 2019). Alternatively, one may use instead a fully discrete actor by discretizing the continuous actions, taking special care to prevent their number from exploding (Metz et al. 2017; Andriotis and Papakonstantinou 2018; Tang and Agrawal 2019).

What we would like instead is a method that would naturally incorporate both discrete and continuous actions within the same algorithm (SAC) in a principled way. In order to accommodate for the wide range of potential ways for an agent to interact with a video game environment, we generalize several existing ideas regarding action representation. We first describe below our proposed generic setting, then relate it to specific examples from the literature.

We denote an agent action a by a combination of discrete components $a^d = (a_1^d, \dots, a_D^d)$ and continuous components $a^c = (a_1^c, \dots, a_C^c)$. Each a_i^d is an integer between 1 and K_i, and represents the i-th discrete action that can be taken by the agent. Each a_j^c is an m_j-dimensional continuous vector in $\mathcal{X}_j \subset \mathbb{R}^{m_j}$, and represents its j-th continuous action. Discrete components are assumed to be independent given the observed state s, while continuous components are independent given both s and the discrete actions, yielding the following decomposition:

$$\begin{aligned} \pi(a|s) &= \pi(a^d|s)\pi(a^c|s,a^d) \\ &= \pi(a_1^d|s)\dots\pi(a_D^d|s)\pi(a_1^c|s,a^d)\dots\pi(a_C^c|s,a^d) \\ &= \Pi_i \pi(a_i^d|s) \Pi_j \pi(a_j^c|s,a^d) \end{aligned}$$

Here we slightly abuse notations by using the same letter π to denote both discrete probability mass functions and probability density functions applied to different components of

the action. A rigorous treatment would rely on measure theory but is beyond the scope of this paper. We observe that many classical action representations fit the above decomposition:

1. A single discrete action taken in the set $1, \dots, K$, as in Atari games (Bellemare et al. 2013). Here $D = 1$, $K_1 = K$ and $C = 0$. This yields
$$\pi(k|s) = \pi(a_1^d = k|s)$$
2. C independently sampled 1D continuous actions, as is typically done in continuous control tasks when computing $a_j^c = \tanh(\mu_j(s) + \varepsilon\sigma_j(s))$ with ε sampled from a standard normal distribution (Haarnoja et al. 2018b). Here $D = 0$, C is the total dimension of the continuous action space, and $m_j = 1$ for all j. This yields
$$\pi(x|s) = \Pi_{j=1}^{C} \pi(a_j^c = x_j|s)$$
3. A single m-dimensional continuous action vector, getting rid of the independence constraint from the previous case #2. This can be achieved for instance with normalizing flows, where the continuous distribution being learned does not need to be be axis aligned anymore (Mazoure et al. 2019). Here $D = 0$, $C = 1$ and $m_1 = m$. This yields
$$\pi(x|s) = \pi(a_1^c = x|s)$$
4. An m-dimensional continuous action whose value should depend on a discrete action taken in $1, \dots, K$, as proposed for parameterized action spaces by (Wei, Wicke, and Luke 2018). Here (in the general case with no independence assumption on the individual continuous components), this means that $D = 1$, $K_1 = K$, $C = 1$ and $m_1 = m$. This yields
$$\pi(k, x|s) = \pi(a_1^d = k|s)\pi(a_1^c = x|s, a_1^d = k)$$
5. An alternative action representation for parameterized action spaces, where the agent takes a discrete action in $1, \dots, K$, and each discrete action k is parameterized by a different continuous m_k-dimensional vector. This is similar to what has been used e.g. by (Bester, James, and Konidaris 2019). Here $D = 1$, $K_1 = K$, $C = K$ and m_k is the dimension of the parameter being used when the discrete action is k. This yields
$$\pi(k, x_k|s) = \pi(a_1^d = k|s)\pi(a_k^c = x_k|s)$$
The difference compared to the previous formulation #4 is that instead of using a single continuous parameter whose value depends on the discrete action being taken, we create multiple independent continuous parameters (one for each discrete action). Since each continuous parameter a_k^c is only used when the agent takes its associated discrete action k, its value does not need to depend on the discrete action chosen by the agent, which is why $\pi(a_k^c = x_k|s)$ does not need to be conditioned on a_1^d.
6. A set of D discrete components, with each component a_i^d ($1 \leq i \leq D$) being a discrete action taken in $(1, \dots, K_i)$. Such a representation has been used in particular by (Tang and Agrawal 2019) to tackle continuous control tasks by discretizing each continuous dimension i into K_i discrete bins. In this example D is the number of original continuous dimensions, K_i is the number of bins in the discretization of the i-th dimension, while there are no continuous actions anymore ($C = 0$). This yields
$$\pi(k_1, \dots, k_D|s) = \Pi_{i=1}^{D} \pi(a_i^d = k_i|s)$$
As motivated by (Tang and Agrawal 2019), such a representation avoids the exponential explosion of discrete actions that would occur if one chose to use instead a single discrete component as in #1. Note that a similar idea is used in the action branching architecture of (Tavakoli, Pardo, and Kormushev 2018).

SAC with mixed discrete-continuous actions

Choosing an appropriate policy parameterization

The examples from the previous section are a subset of all possible ways one can represent the action distribution over a mix of discrete and continuous components, using our generic proposed decomposition. From a practitioner point of view, there is no single best representation that will fit all use cases. For instance, if the agent needs to press buttons on a controller, and there are four buttons which can be set on/off, one can either consider a single discrete component with $2^4 = 16$ actions, or four independent binary discrete components. The latter approach has the benefit of reducing the number of parameters that need to be learned, thanks to the factored representation, and thus generally scales better as the number of discrete components increases. On the other hand, the independence assumption can make it harder for the agent to learn coordinated button presses, so the factored approach may perform badly when interactions between the discrete components really matter. In general, we give the following advice to obtain an appropriate representation for a given task, based on our own experience:

- Identify which action components (both discrete and continuous) should be made dependent of each other. When in doubt, it is advised to start with a simpler parameterization based on independent components, and only investigate later the potential benefits of more complex parameterizations. Note that in an MDP there always exists an optimal deterministic policy, for which all action components are independent given the state. As a result, it could be tempting to assume that everything can always be made independent (in order to simplify the model), but in practice this may slow down learning, in particular because it can prevent coordinated exploration across components (think of the above example with button presses).
- When a continuous component depends on a discrete component, consider duplicating it (one for each discrete action) as long as the model size remains reasonable: this will allow you to consider them as independent, making it easier for the model to specialize the value of the component to each discrete action. For instance, consider an (x, y) continuous component which gives the 2D coordinates of a mouse click, where the agent has to select among several discrete actions before clicking (ex:

attack, heal, follow): this continuous component may be replaced with three independent ones (x_a, y_a), (x_h, y_h) and (x_f, y_f) associated to each discrete action, as in point #5 above. If this is too costly (due to a large number of discrete actions), you can instead (as in #4) build the policy network in such a way that the continuous component head takes as input the discrete one: for details refer to (Wei, Wicke, and Luke 2018).

- If possible, try to avoid dependencies among continuous dimensions, so as to keep a simple parameterization where each action dimension can be sampled independently. For instance, if your continuous action is a pair (a_x, a_y) giving the acceleration of your agent along the x and y axes, the agent may struggle to explore properly in situations where it needs to navigate narrow corridors that may not be axis aligned, since accelerations on both axes must be correlated to avoid bumping into the walls. In this specific case, one could for instance make the acceleration actions relative to the direction the agent is currently facing (by rotating the axes accordingly), making it easier for the agent to explore a wide range of forward accelerations without deviating from its trajectory.

Practical implementation

Network architecture Fig. 1a shows the typical architecture for the actor and critic networks used in standard continuous SAC implementations. Using a different policy parameterization (like one of those described previously) calls for a different network architecture. One common case is shown in Fig. 1b, in the situation where the agent must take a combination of one discrete action a^d with a set of independently sampled continuous parameters a^c.

Note that here, we chose to take an approach similar to (Xiong et al. 2018) where the critic's output layer contains the predicted Q-values of all discrete actions, instead of feeding the discrete action as input as done in the so-called "multi-pass" architecture of (Bester, James, and Konidaris 2019). This is because the former is the most commonly used architecture for discrete actions when using the popular Deep Q-Network algorithm and its variants (Mnih et al. 2015; Hessel et al. 2017), but we acknowledge that the multi-pass architecture of (Bester, James, and Konidaris 2019) is also a valid alternative. We actually implemented it in SAC, but our preliminary results did not show meaningful improvements, so we did not investigate it further at this time.

More complex policy parameterizations would lead to more elaborate architectures for the actor network than shown in Fig. 1b, e.g.:

- In the case of multiple independent discrete components, the actor would output several corresponding discrete distributions $\pi_1^d, \ldots, \pi_D^d$.
- If the continuous dimensions must be correlated, a different parameterization of a^c may be used, for instance using normalizing flows (Mazoure et al. 2019).
- If the continuous action a^c must depend on the discrete action chosen by the agent, then a^d can be used as input when computing μ^c and σ^c (Wei, Wicke, and Luke 2018).

Learning algorithm The SAC algorithm (Haarnoja et al. 2018b) is based on the idea of giving an entropy bonus proportional to the entropy of $\pi(a|s)$. When the action has a discrete component, the joint entropy definition yields

$$\mathcal{H}\big(\pi(a^d, a^c|s)\big) = H\big(\pi(a^d|s)\big) + \sum_{a^d} \pi(a^d|s)\mathcal{H}\big(\pi(a^c|s, a^d)\big)$$

Although we could simply give a bonus proportional to this entropy, we argue that in some situations it may be beneficial to give different weights to its discrete and continuous parts. This is because otherwise, depending in particular on the number of discrete and continuous actions, there would be a risk for one of these two entropies to "overshadow" the other, which could harm exploration. As a result, we use as entropy bonus the weighted sum

$$\alpha^d H\big(\pi(a^d|s)\big) + \alpha^c \sum_{a^d} \pi(a^d|s)\mathcal{H}\big(\pi(a^c|s, a^d)\big) \qquad (2)$$

where hyperparameters α^d and α^c encourage exploration for discrete and continuous actions respectively. Note that these two hyperparameters can be tuned automatically during learning, using the same optimization technique as described in (Haarnoja et al. 2018c), by setting target values for the discrete and continuous parts in eq. 2.

In terms of practical implementation, a list of the changes between our proposed Hybrid SAC and the original version can be found in the Appendix. Note that when there are only discrete actions, our approach is equivalent to the one proposed concurrently by (Christodoulou 2019).

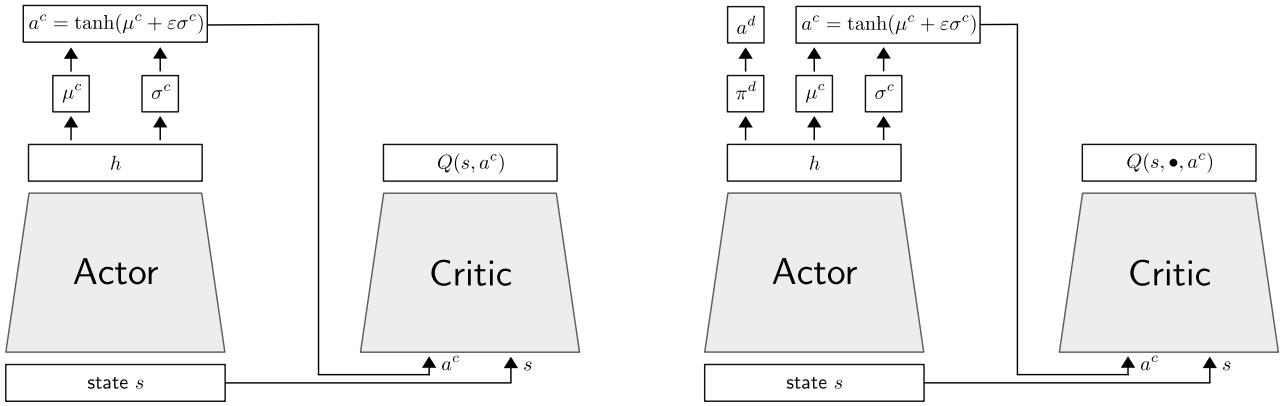
Experiments with parameterized actions

We evaluate our Hybrid SAC implementation on the same three parameterized actions environments used by (Bester, James, and Konidaris 2019):

- *Platform* is a simple platformer-like game where the agent has three discrete actions (run, hop and leap), each associated with a 1D continuous parameter controlling the horizontal displacement.
- *Goal* is a soccer-based game where the agent needs to score a goal past a keeper that tries to intercept the ball. There are again three discrete actions, with respectively 2D, 1D and 1D continuous parameters.
- *Half Field Offense* is another soccer-based game, also with three discrete actions, but this time with respectively 2D, 1D and 2D continuous parameters.

In order to allow for a fair comparison with the state-of-the-art Multi-Pass Q-Network (MP-DQN) algorithm of (Bester, James, and Konidaris 2019), we re-used their evaluation code and tried to match their hyperparameters whenever possible. We list the main remaining differences between our work and theirs in the appendix.

Results are summarized in Table 1. Both algorithms perform equally well on *Platform*, while MP-DQN exhibits slightly better performance on *Goal* and significantly better performance on *HFO*. Note however that the MP-DQN results on *HFO* are based on an implementation that mixes



(a) Standard SAC

(b) Hybrid SAC (one discrete component and one continuous)

Figure 1: (a) On the left, the standard SAC architecture for continuous actions. The actor outputs the mean and standard deviation vectors μ^c and σ^c that are used to sample an action a^c by injecting standard normal noise ε and applying a $\tanh$ non-linearity (to keep the action within a bounded range). The critic takes both the state s and the actor's action a^c to estimate their corresponding Q-value. (b) On the right, an example of our proposed Hybrid SAC architecture, with two independent components (one discrete and one continuous). The actor computes a shared hidden state representation h that is used to produce both a discrete distribution π^d (typically from a $\mathrm{softmax}$ layer) as well as the mean μ^c and standard deviation σ^c of the continuous component. The discrete action a^d is sampled from π^d while the continuous action a^c is computed as in the standard SAC. The critic network still takes both the state s and the continuous action a^c as input, but now predicts the Q-values of all discrete actions in its output layer.

	Platform Return	*Goal* P(Goal)	*HFO* P(Goal)
MP-DQN	0.987 ± 0.039	0.789 ± 0.070	0.913 ± 0.070
MP-DQN (no MC)	-	-	0.509 ± 0.110
Hybrid SAC	0.981 ± 0.013	0.728 ± 0.047	0.639 ± 0.141

Table 1: Comparison between the Multi-Pass Deep Q-Network (MP-DQN) algorithm from (Bester, James, and Konidaris 2019) and our Hybrid SAC implementation. Mean performance with 95% confidence interval is computed over 30 seeds. Since the MP-DQN results on *HFO* take advantage of Monte-Carlo returns, while our Hybrid SAC does not, we also report in the second row the (significantly degraded) performance of MP-DQN without Monte-Carlo returns.

Monte-Carlo returns with one-step returns to speed up convergence, an improvement that we did not implement in our Hybrid SAC. The second row reports the performance of MP-DQN without mixing Monte-Carlo returns on *HFO*, showing that it degrades considerably (at least with the same hyper-parameters as MP-DQN).

While investigating potential reasons for the slightly worse average performance of Hybrid SAC on *Goal*, we realized that the entropy bonus from eq. 2 may have an undesirable effect. Discrete actions with a small $\pi(a^d|s)$ lead to a reduced entropy bonus for their associated $\pi(a^c|s, a^d)$. This may cause the distribution of some continuous parameters to sometimes "collapse". Our preliminary experiments with a variant aimed at avoiding this collapse matched the results of MP-DQN, but a more in-depth analysis of this variant is still needed before we can confidently report on its performance.

Results in a commercial video game

We trained a vehicle in a Ubisoft game, using the proposed Hybrid SAC with two continuous actions (acceleration and steering) and one binary discrete action (hand brake). The objective of the car is to follow a given path as fast as possible. A video of the resulting behavior is available at https://youtu.be/bmrNMDEkPyQ. Note that the agent operates in a test environment that it did not see during training, and that the discrete hand brake action plays a key role in staying on the road at such a high speed.

Experiments with Normalizing Flows

Our main objective in an industry setting is to optimize the final performance of the policy under a budget constraint on its inference runtime. A potential avenue that could help is to augment the Gaussian policy obtained from a standard SAC algorithm with radial flows as advised in (Mazoure et al. 2019), who report significantly improved performance with a reduced number of parameters. They suggest that such improvements could be related to the ability of the policy to be more expressive, for example by allowing it to be multimodal. In theory, training multimodal policies could yield agents that behave more naturally, for example in a driving situation where they could avoid an obstacle by turning either left or right.

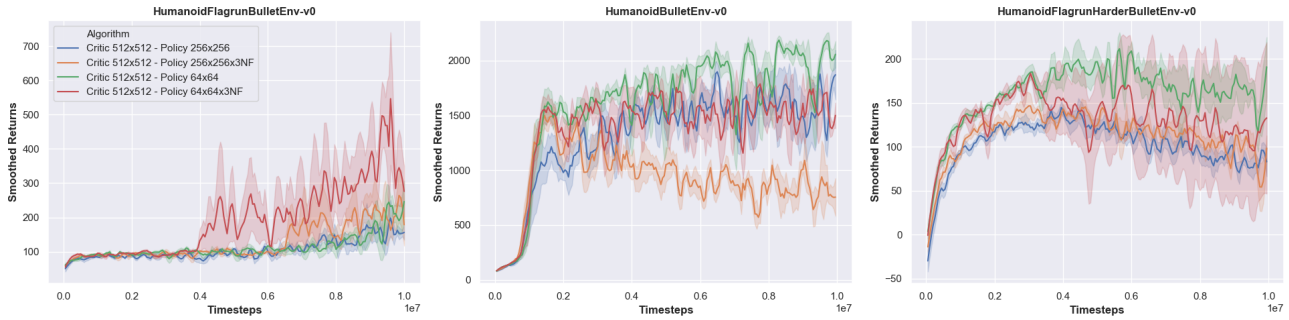


Figure 2: Comparison of the performance of SAC with and without radial normalizing flows on three Roboschool PyBullet environments. Curves are averaged on 5 random seeds, and smoothed using Savitzky-Golay filtering with window size 7.

Bullet Roboschool benchmarks

Our SAC baseline consists of a two-layer feedforward network outputting the mean and the standard deviation parameterizing a spherical Gaussian. The SAC-NF agent has the same architecture, but adds several radial flows on the output of the Gaussian. The resulting action is then squashed using a $\tanh$ as in (Haarnoja et al. 2018b). All the networks are trained using the Adam optimizer (Kingma and Ba 2014), details of the models' architectures can be found in appendix in Table 2.

We evaluate the different architectures on the PyBullet Roboschool benchmark (Coumans and Bai 2016). We take one step of training every ten environment steps, and evaluate the policy every 50,000 steps. All the results are averaged over 5 random seeds. Since our intention is to see if the boost in policy expressiveness provided by normalizing flows can really help during training, we use bigger networks for the two critics so that the training is not limited by their relatively low capacity. Results of this comparison can be found in Fig. 2.

Fig. 2 shows that while a smaller policy with two hidden layers of 64 neurons with normalizing flows can get results that are competitive with bigger networks during the first million iterations as also reported by (Mazoure et al. 2019), this advantage does not always hold as training goes further. Our results suggest that using normalizing flows on top of SAC does not yield a significant advantage compared to simply using the Gaussian policy of the baseline. In the following section, we will conduct an experiment on a toy environment to try to understand why.

Normalizing Flows and SAC

In SAC, the actor tries to optimize the Kullback-Leibler (KL) divergence between the policy and a softmax on the soft Q-values with temperature α. (Haarnoja et al. 2018b) demonstrate that updating the policy in such a way improves it until convergence, and (Abdolmaleki et al. 2018) show that this update constrains the change of the policy. We thus try to minimize:

$$J_\pi(\phi) = \mathbb{E}_{s_t \sim \pi_\phi}\left[D_{KL}\left(\pi_\phi(\cdot|s_t)\middle\|\frac{\exp\left(\frac{Q_\theta(s_t,\cdot)}{\alpha}\right)}{Z_\theta(s_t)}\right)\right] \tag{3}$$

where Z_θ is the partition function. However, the KL is not symmetric, and there is no theoretical ground in why π_ϕ should be the first argument. The main advantage of minimizing eq. 3 with π_θ in first position in the KL is tractability, as using the reparameterization trick allows us to minimize it without knowing the partition function. To do this, we rewrite the objective as an expectation on standard normal noise ε and then sample this expectation:

$$J_\pi(\phi) = \mathop{\mathbb{E}}_{\substack{s_t \sim \pi_\phi \\ \varepsilon_t \sim \mathcal{N}}}\Big[\log \pi_\phi\big(f_\phi(\varepsilon_t; s_t)|s_t\big) - Q_\theta\big(s_t, f_\phi(\varepsilon_t; s_t)\big)\Big] \tag{4}$$

where f_ϕ reparameterizes the policy in terms of the noise ε. The particular choice of using the KL divergence from π_ϕ to the target softmax is motivated mainly by the convenience of its implementation. However, in classification tasks we generally try to minimize the negative log-likelihood, which is equivalent to minimizing the KL divergence from the empirical distribution to the parameterized one. In policy distillation, (Czarnecki et al. 2019; Parisotto, Ba, and Salakhutdinov 2015; Schmitt et al. 2018) good results are reported when trying to minimize $\mathbb{E}_{\pi_\phi}\left[\sum_{t=1}^{\tau} \nabla_\phi H^\times\big(\pi(s_t)||\pi_\phi(s_t)\big)\right]$ where $H^\times$ is the cross-entropy, and the trajectories are sampled according to the student policy π_ϕ instead of the teacher policy π. We also did some experiments with distillation (not included here) which confirm that this way of doing policy distillation yields good results. All these observations suggest that if we interpret eq. 3 as trying to distill the "teacher" softmax over Q-values into the "student" parameterized policy π_ϕ, a KL in the other direction would yield better results. This motivates the following comparison to measure the difference between these alternative objectives.

In this comparison, we fix a random state s_0 and try to get our policy to approximate a toy distribution π, in this case a Gaussian mixture, for this fixed state. Several objectives are evaluated, and we monitor the impact of using each objective on the shape of the final distribution after 10,000 steps of training. All hyperparameters are identical to our Roboschool experiment. We compare the Gaussian policy as used in SAC to the SAC-NF policy which adds three radial flows on top of the Gaussian. We compare the two direc-

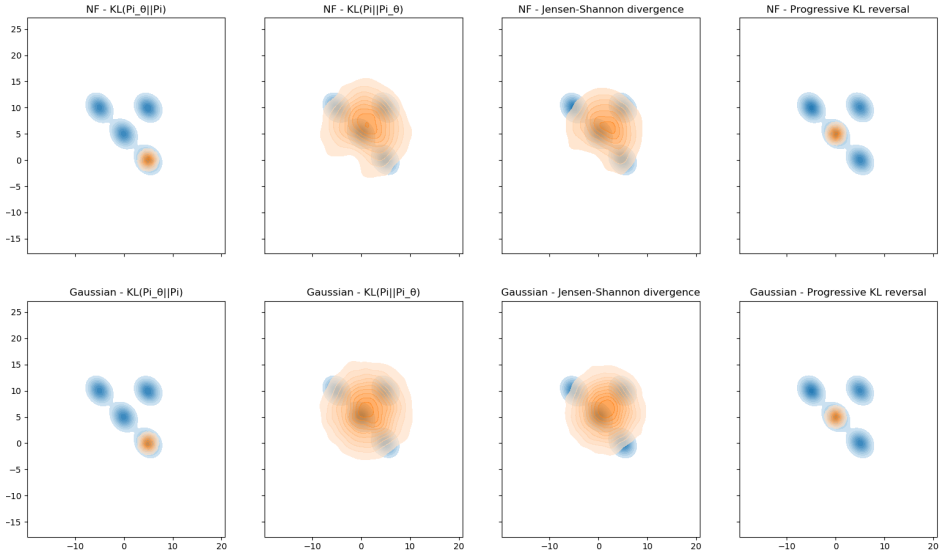


Figure 3: Comparison between the final shapes of the policy distribution with several objectives after trying to match a Gaussian mixture for 10,000 steps. The blue and orange densities correspond to the target Gaussian mixture π and the learned distribution π_ϕ respectively. Top row uses normalizing flows, while the bottom row is using a Gaussian policy. Various divergence metrics are evaluated from left to right.

tions of the KL divergence, as well as the Jensen-Shannon divergence (Lin 1991). We also tried to linearly switch from $D_{KL}(\pi_\phi||\pi)$ to $D_{KL}(\pi||\pi_\phi)$ during training. Results of this comparison can be found in Fig. 3. We use the kernel density estimate provided in the seaborn library (Waskom et al. 2018) to estimate the density of the distributions.

From this toy experiment, one reason why normalizing flows did not seem to improve performance on Roboschool could be that any advantage gained in expressiveness of the policy by enriching it with normalizing flows is lost by the optimization procedure used in SAC. Indeed, when using the same objective as SAC (leftmost column in Fig. 3), there seems to be very little difference between using a Gaussian policy and one with normalizing flows, since both collapse on a single mode of the target distribution. Note that in this comparison we did not take the impact of the temperature into account (another comparison on the impact of the temperature can be found in Fig. 4 and 5 in the Appendix). However, when we invert the KL (as we do in supervised learning and distillation) or use the Jensen-Shannon divergence, it appears that the normalizing flows help the policy better match the complete target distribution.

These results suggest that using normalizing flows could yield some benefits when used with other objectives than the one used in SAC. We ran some experiments reverting the KL using importance sampling, but training was too unstable. We identify the exploration of other metrics between distributions, such as the Jensen-Shannon divergence or the Wasserstein distance, as potential research avenues that could yield significant improvements when used in conjunction with normalizing flows in SAC.

Conclusion

We introduced Hybrid SAC, an extension to the SAC algorithm that can handle discrete, continuous and mixed discrete-continuous actions. It exhibits competitive performance with the state-of-the-art on parameterized actions benchmarks. We showed that Hybrid SAC can be successfully applied to train a car on a high-speed driving task in a commercial video game, demonstrating the practical usefulness of such an algorithm for the video game industry. Our study of the use of normalizing flows with the SAC algorithm also suggests that future approaches could further improve SAC by using other objectives than the KL, so as to better leverage normalizing flows.

Acknowledgments

We would like to thank the authors of (Mazoure et al. 2019) for insightful conversations and providing us with their implementation, as well as Paul Barde for his valuable feedback while writing this paper.

References

Abdolmaleki, A.; Springenberg, J. T.; Degrave, J.; Bohez, S.; Tassa, Y.; Belov, D.; Heess, N.; and Riedmiller, M. 2018. Relative entropy regularized policy iteration. *arXiv preprint arXiv:1812.02256*.

Andriotis, C. P., and Papakonstantinou, K. G. 2018. Managing engineering systems with large state and action spaces through deep reinforcement learning. *CoRR* abs/1811.02052.

Bellemare, M. G.; Naddaf, Y.; Veness, J.; and Bowling, M. 2013. The arcade learning environment: An evaluation platform for general agents. *J. Artif. Int. Res.* 47(1):253–279.

Bester, C. J.; James, S. D.; and Konidaris, G. D. 2019. Multi-pass q-networks for deep reinforcement learning with parameterised action spaces. *CoRR* abs/1905.04388.

Christodoulou, P. 2019. Soft actor-critic for discrete action settings. *CoRR* abs/1910.07207.

Cianflone, A.; Ahmed, Z.; Islam, R.; Bose, A. J.; and Hamilton, W. L. 2019. Discrete off-policy policy gradient using continuous relaxations. unpublished.

Coumans, E., and Bai, Y. 2016. Pybullet, a python module for physics simulation for games, robotics and machine learning. *GitHub repository*.

Czarnecki, W. M.; Pascanu, R.; Osindero, S.; Jayakumar, S. M.; Swirszcz, G.; and Jaderberg, M. 2019. Distilling policy distillation. *arXiv preprint arXiv:1902.02186*.

Haarnoja, T.; Hartikainen, K.; Abbeel, P.; and Levine, S. 2018a. Latent space policies for hierarchical reinforcement learning. *arXiv preprint arXiv:1804.02808*.

Haarnoja, T.; Zhou, A.; Abbeel, P.; and Levine, S. 2018b. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv preprint arXiv:1801.01290*.

Haarnoja, T.; Zhou, A.; Hartikainen, K.; Tucker, G.; Ha, S.; Tan, J.; Kumar, V.; Zhu, H.; Gupta, A.; Abbeel, P.; et al. 2018c. Soft actor-critic algorithms and applications. *arXiv preprint arXiv:1812.05905*.

Hausknecht, M., and Stone, P. 2016. Deep reinforcement learning in parameterized action space. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

Hessel, M.; Modayil, J.; van Hasselt, H.; Schaul, T.; Ostrovski, G.; Dabney, W.; Horgan, D.; Piot, B.; Azar, M. G.; and Silver, D. 2017. Rainbow: Combining improvements in deep reinforcement learning. In *AAAI*.

Kingma, D. P., and Ba, J. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Lillicrap, T. P.; Hunt, J. J.; Pritzel, A.; Heess, N.; Erez, T.; Tassa, Y.; Silver, D.; and Wierstra, D. 2016. Continuous control with deep reinforcement learning. In Bengio, Y., and LeCun, Y., eds., *ICLR*.

Lin, J. 1991. Divergence measures based on the shannon entropy. *IEEE Transactions on Information theory* 37(1):145–151.

Mazoure, B.; Doan, T.; Durand, A.; Hjelm, R. D.; and Pineau, J. 2019. Leveraging exploration in off-policy algorithms via normalizing flows. *arXiv preprint arXiv:1905.06893*.

Metz, L.; Ibarz, J.; Jaitly, N.; and Davidson, J. 2017. Discrete sequential prediction of continuous actions for deep RL. *CoRR* abs/1705.05035.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518(7540):529.

OpenAI. 2018. Openai five.

Parisotto, E.; Ba, J. L.; and Salakhutdinov, R. 2015. Actor-mimic: Deep multitask and transfer reinforcement learning. *arXiv preprint arXiv:1511.06342*.

Rezende, D. J., and Mohamed, S. 2015. Variational inference with normalizing flows. *arXiv preprint arXiv:1505.05770*.

Schmitt, S.; Hudson, J. J.; Zidek, A.; Osindero, S.; Doersch, C.; Czarnecki, W. M.; Leibo, J. Z.; Kuttler, H.; Zisserman, A.; Simonyan, K.; et al. 2018. Kickstarting deep reinforcement learning. *arXiv preprint arXiv:1803.03835*.

Tang, Y., and Agrawal, S. 2018. Boosting trust region policy optimization by normalizing flows policy. *arXiv preprint arXiv:1809.10326*.

Tang, Y., and Agrawal, S. 2019. Discretizing continuous action space for on-policy optimization. *CoRR* abs/1901.10500.

Tavakoli, A.; Pardo, F.; and Kormushev, P. 2018. Action branching architectures for deep reinforcement learning. In *AAAI Conference on Artificial Intelligence*, 4131–4138.

van Hasselt, H., and Wiering, M. A. 2009. Using continuous action spaces to solve discrete problems. In *Proceedings of the 2009 International Joint Conference on Neural Networks*, IJCNN'09, 1144–1151. Piscataway, NJ, USA: IEEE Press.

Vinyals, O.; Babuschkin, I.; Chung, J.; Mathieu, M.; Jaderberg, M.; Czarnecki, W.; Dudzik, A.; Huang, A.; Georgiev, P.; Powell, R.; Ewalds, T.; Horgan, D.; Kroiss, M.; Danihelka, I.; Agapiou, J.; Oh, J.; Dalibard, V.; Choi, D.; Sifre, L.; Sulsky, Y.; Vezhnevets, S.; Molloy, J.; Cai, T.; Budden, D.; Paine, T.; Gulcehre, C.; Wang, Z.; Pfaff, T.; Pohlen, T.; Yogatama, D.; Cohen, J.; McKinney, K.; Smith, O.; Schaul, T.; Lillicrap, T.; Apps, C.; Kavukcuoglu, K.; Hassabis, D.; and Silver, D. 2019a. AlphaStar: Mastering the real-time strategy game StarCraft II. https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/.

Vinyals, O.; Babuschkin, I.; Czarnecki, W. M.; Mathieu, M.; Dudzik, A.; Chung, J.; Choi, D. H.; Powell, R.; Ewalds, T.; Georgiev, P.; et al. 2019b. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature* 1–5.

Ward, P. N.; Smofsky, A.; and Bose, A. J. 2019. Improving exploration in soft-actor-critic with normalizing flows policies. *arXiv preprint arXiv:1906.02771*.

Waskom, M.; Botvinnik, O.; O'Kane, D.; Hobson, P.; Ostblom, J.; Lukauskas, S.; Qalieh, A.; et al. 2018. mwaskom/seaborn: v0. 9.0 (july 2018). *DOI: https://doi.org/10.5281/zenodo* 1313201.

Wei, E.; Wicke, D.; and Luke, S. 2018. Hierarchical approaches for reinforcement learning in parameterized action space. In *2018 AAAI Spring Symposium Series*.

Xiong, J.; Wang, Q.; Yang, Z.; Sun, P.; Han, L.; Zheng, Y.; Fu, H.; Zhang, T.; Liu, J.; and Liu, H. 2018. Parametrized deep q-networks learning: Reinforcement learning with discrete-continuous hybrid action space. *CoRR* abs/1810.06394.

Appendix

Implementation differences between Hybrid SAC and SAC

- The entropy bonus used in the target value for the critic network Q is computed as in eq. 2, where the discrete part can be computed exactly (due to the finite number of discrete actions) while the continuous one needs to be approximated by sampling, as is usually done for continuous SAC.
- When optimizing the critic network Q with a transition sampled from the replay buffer, only the output associated with the discrete action taken in this transition is optimized, similar to the Deep Q-Network algorithm (Mnih et al. 2015).
- The discrete part $\pi(a^d|s)$ of the policy is optimized by minimizing the KL divergence between this distribution and the one induced by the softmax on the Q-values with temperature α_d. Since these Q-values depend on the continuous components a^c, we sample $a^c \sim \pi(a^c|s, a^d)$ in order to compute $q_d = Q(s, a^d, a^c)$ for each a^d, and take a gradient step to minimize the KL divergence between $\pi(a^d|s)$ and $P(a^d) \propto \exp(q_d/\alpha_d)$. As is usually done in continuous SAC, we multiply this gradient by α_d so as to prevent it from blowing up for small values of α_d.
- Finally, the same a^c sampled above are re-used to compute the update step for the continuous part of the policy. This update is essentially the same as in continuous SAC, as in eq. 7 of (Haarnoja et al. 2018c), except that it is performed as a weighted average over all discrete actions a^d, where the weight is given by $\pi(a^d|s)$ (i.e. mimicking the weighting scheme of eq. 2).

Differences between MP-DQN and Hybrid SAC

The main differences between our implementation of Hybrid SAC and the Multi-Pass DQN algorithm from (Bester, James, and Konidaris 2019) are the following:

- We do not use the Multi-Pass architecture in our critic Q, because it significantly slows down learning and did not seem to really help in our preliminary experiments with SAC. Additional experiments are needed to fully investigate the potential benefits of this architecture for Hybrid SAC.
- For the sake of simplicity, we use a squashing $\tanh$ to bound the actions instead of the inverting gradient technique (Hausknecht and Stone 2016), and do not use gradient clipping.
- Since our approach is based on SAC, while MP-DQN is based on a combination of DQN (Mnih et al. 2015) and DDPG (Lillicrap et al. 2016), we do not use ε-greedy exploration nor add noise to continuous actions, but instead rely on the actor's stochasticity for exploration.
- We tweaked our actor and critic learning rates (as well as SAC-specific hyperparameters like the target discrete and continuous entropies) by a cursory search over reasonable-looking values.
- In the *Platform* environment, we do not use the custom initialization of the continuous parameters from (Bester, James, and Konidaris 2019) because we found it easy enough to get good results without it.
- In the *Half Field Offense* environment, we do not mix Monte-Carlo returns with one-step returns. Incorporating Monte-Carlo returns is not entirely straightforward in SAC due to the need to account for the entropy bonus, so we leave it to future work.

Roboschool hyperparameters

Parameter	Value
Optimizer	Adam
Learning rate	3×10^{-4}
Discount (γ)	0.99
Replay buffer size	10^6
Alpha	0.05
Number of hidden layers	2
Neurons per hidden layer	256
Activation function	ReLU
Minibatch size	1024
Target smoothing coefficient	0.005
Training / environment steps	0.1
Number of environment steps	10^7
Number of radial flows	3

Table 2: SAC hyperparameters used in Roboschool.

Normalizing flows parameterization

We note d the dimension of the action space. We parameterize $\phi = (z_0, x, y) \in \mathbb{R}^3$ as follows:

Parameter	Value
α	$\exp(x)$
β	$-\alpha + \exp(y)$
$f_\phi(z)$	$z + \beta \cdot \frac{z - z_0}{\alpha + r(z)}$
$r(z)$	$\|\|z - z_0\|\|_2$
$\det \frac{\partial f_\phi(z)}{\partial z}$	$\left(1 + \frac{\beta}{\alpha + r(z)} - \frac{\beta r(z)}{(\alpha + r(z))^2}\right) \cdot \left(1 + \frac{\beta}{\alpha + r(z)}\right)^{d-1}$

Table 3: Parameterization of the normalizing flows.

Additional experiments with normalizing flows

Fig. 4 and 5 extend the results presented in Fig. 3 where we train a policy π_ϕ to match a target distribution π.

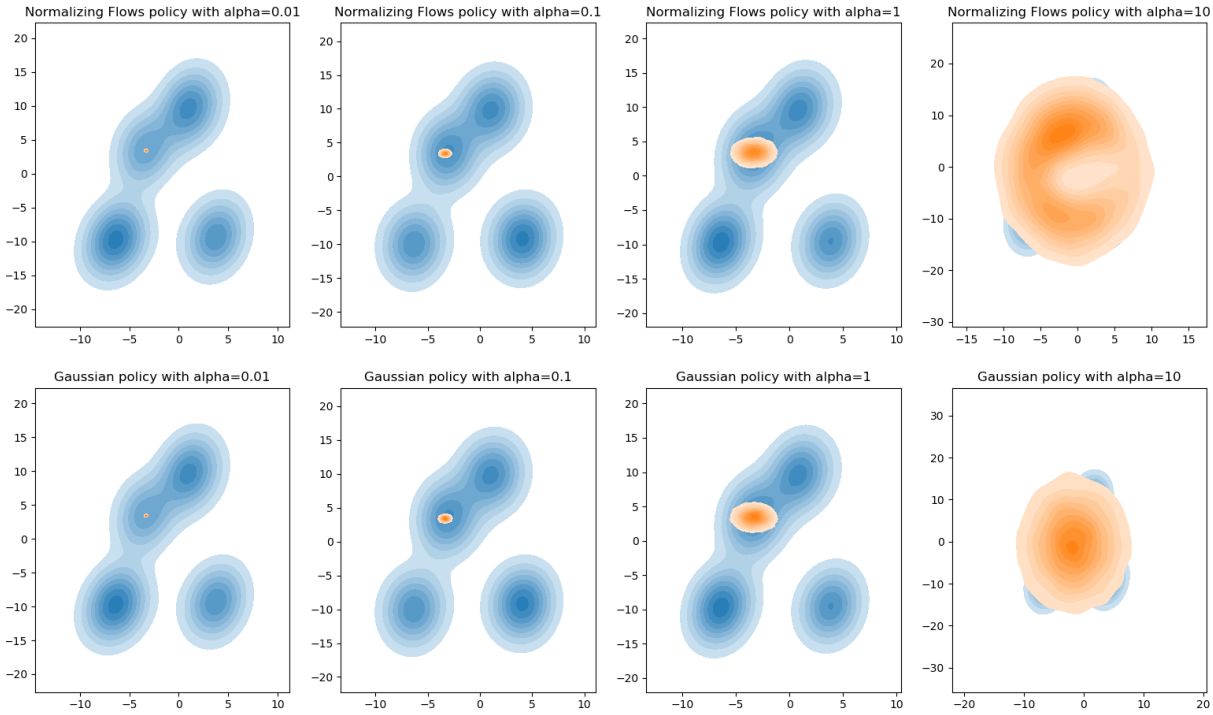


Figure 4: Comparison between the final shapes of the policy distribution π_ϕ (in orange) trained with the same objective as SAC after trying to match a Gaussian mixture π (in blue) with different temperatures α for 10,000 steps. Note the collapse of the policies on one mode of π unless α gets very high, for both normalizing flows (top row) and Gaussian policy (bottom row).

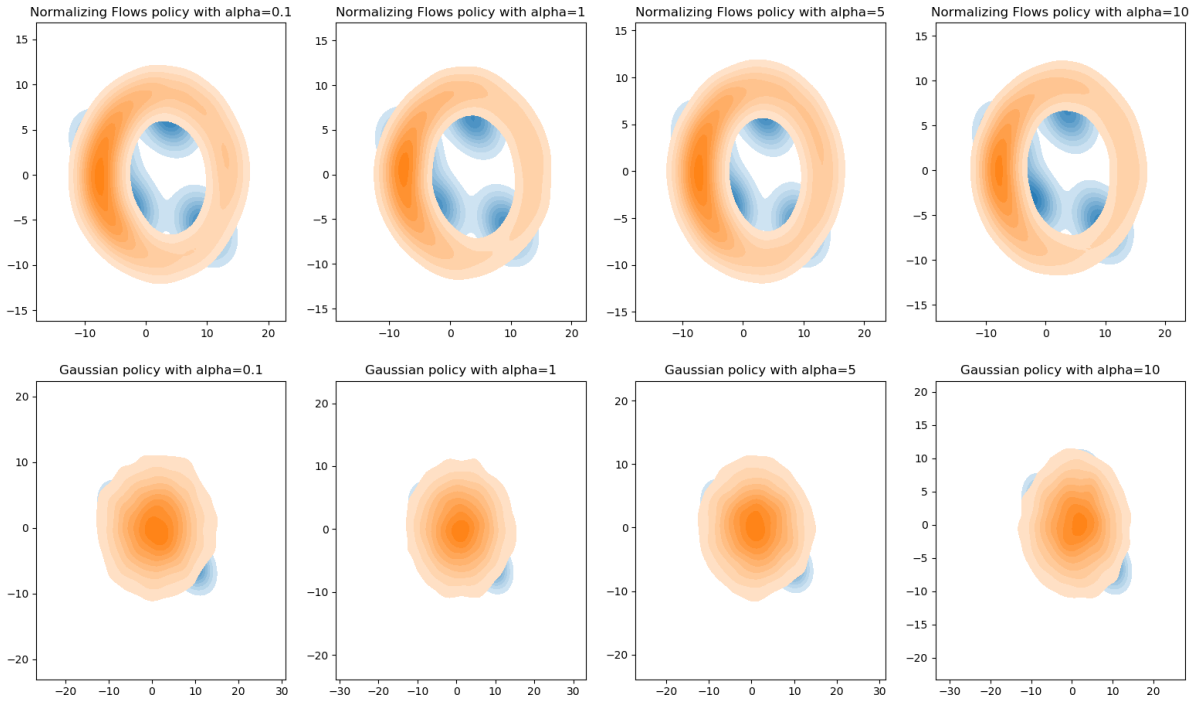


Figure 5: Same as Fig. 4 but swapping the arguments of the KL divergence objective. Note that the policies no longer collapse onto a single mode of the target π, and the normalizing flow policy is better able to approximate the shape of π.