Proximal Policy Optimization with Continuous Bounded Action Space via the Beta Distribution

Irving G. B. Petrazzini
Automation and Systems Engineering
Federal University of Santa Catarina
Florianópolis, Santa Catarina, 88040-900
Email: irving.petrazzini@posgrad.ufsc.br

Eric A. Antonelo
Automation and Systems Engineering
Federal University of Santa Catarina
Florianópolis, Santa Catarina, 88040-900
Email: eric.antonelo@ufsc.br

2021 IEEE Symposium Series on Computational Intelligence (SSCI) | 978-1-7281-9048-8/21/$31.00 ©2021 IEEE | DOI: 10.1109/SSCI50451.2021.9660123

***Abstract*—Reinforcement learning methods for continuous control tasks have evolved in recent years generating a family of policy gradient methods that rely primarily on a Gaussian distribution for modeling a stochastic policy. However, the Gaussian distribution has an infinite support, whereas real world applications usually have a bounded action space. This dissonance causes an estimation bias that can be eliminated if the Beta distribution is used for the policy instead, as it presents a finite support. In this work, we investigate how this Beta policy performs when it is trained by the Proximal Policy Optimization (PPO) algorithm on two continuous control tasks from OpenAI gym. For both tasks, the Beta policy is superior to the Gaussian policy in terms of agent's final expected reward, also showing more stability and faster convergence of the training process. For the CarRacing environment with high-dimensional image input, the agent's success rate was improved by 63% over the Gaussian policy.**

I. Introduction

Deep Reinforcement Learning (RL) has achieved unprecedented results on challenging high-dimensional continuous state-space problems, surpassing human performance in 29 out of 49 Atari 2600 games in [1], for instance. Later, AlphaGO, an agent that combines reinforcement learning and Monte Carlo balanced search tree algorithms with self play was able beat Lee Sedol, a 9th-dan, world champion [2]. In this context, Convolutional Neural Networks (CNNs) [3] serve as function approximators for the Q-value function, since they can efficiently process image inputs and learn useful feature representations from these high-dimensional continuous state-space domains.

Handling discrete action spaces in a deep reinforcement task usually resumes to defining an output layer of a neural network that has the same dimension of the action space. If this space is small, an action can be easily drawn from the distribution yielded by the layer's activation. Otherwise, finding the best action for high-dimensional or continuous action spaces constitutes an expensive optimization process per se, which needs to be run inside another loop, the agent-environment cycle.

Many interesting real-world problems such as control of robotic arms and autonomous cars require a continuous action space. Instead of modeling the state-action Q-value function, model-free continuous control via reinforcement learning is made possible by directly optimizing a policy function which maps states to probability distributions over continuous action spaces. This family of policy gradient methods have undergone important advancements allowing for high-dimensional continuous state spaces (such as images) and continuous action spaces [4]–[7].

To model a stochastic policy, these methods choose the Gaussian distribution $\mathcal{N}(\mu, \sigma^2)$, whose parameters μ and σ^2 are to be estimated as outputs of a deep neural network. However, many real-world applications have bounded action spaces, usually owing to physical constraints, e.g., by the joints of a humanoid robot or manipulator, and by the accelerator and steering direction of a vehicle. Thus, in these cases, this Gaussian policy, which has infinite support, introduces a estimation bias since it can give a nonzero probability for *actions* outside the valid action space.

Recently, [8] proposes to model the stochastic policy with the Beta distribution, with parameters α and β, such that the resulting policy has a suitably bounded action space, that presents no bias as the previously considered Gaussian distribution. Instead of the mean and variance, now the outputs of the neural network represent the policy parameters α and β. The Beta distribution can be used with any on- or off-policy methods such as Trust Region Policy Optimization (TRPO) [4] and Actor-Critic Experience Replay (ACER) [7].

So far, the Beta distribution has been evaluated only for TRPO and ACER on a variety of problems. Proximal Policy Optimization (PPO), which evolved from TRPO but has a much simpler implementation and a similar performance to ACER, still lacks experimentation with the Beta distribution. This is the first work to report experiments on PPO with the Beta distribution on RL applications with high-dimensional observation spaces. Besides, our investigation focus on two continuous control applications from OpenAI Gym, the Lunar Lander and the Car Racing, both of which were not considered in [8].

The benefits of the approach are better stability and faster convergence of the training process. Furthermore, because the estimation bias is absent, the final learned Beta policy is superior to the final Gaussian policy. We also report results better than state-of-the-art on the Car Racing problem.

II. Background

A. Markov decision process

We model our continuous control reinforcement learning task as a finite Markov decision process (MDP). An MDP consists of a state space S, an action space A, an initial state s_0, and a reward function $r(s,a) : S \times A$ that emits a scalar value for any transition from state s taking action a. At each time step t, the agent selects an action a_{t+1} according to a policy π, i.e., $a_{t+1} = \pi(s_t)$, such that agent's future expected reward is maximized. A stochastic policy can be described as a probability distribution of taking an action a_{t+1} given a state s_t denoted as $\pi(a|s) : S \rightarrow A$. A deterministic policy can be obtained by taking the expected value of the policy $\pi(a|s)$.

B. Policy Gradient Methods

Value-based reinforcement learning methods first learn to approximate a value function $Q(s,a)$. The policy is obtained by finding the action that maximizes the latter, e.g., $\pi(s) = \arg\max_a Q(s,a)$. On the other hand, policy gradient methods optimize directly an parametrized policy $\pi_\theta(a|s)$ that can model Categorical or Continuous actions for discrete and continuous spaces, respectively.

For a given scalar performance measure $L(\theta) = v_{\pi_\theta}(s_0)$, where v_{π_θ} is the true value function for π_θ, the policy determined by θ, performance is maximized through gradient ascent on L

$$L(\theta) = \int_S \rho^\pi(s) \int_A \pi_\theta(s,a) r(s,a) da ds \tag{1}$$

$$= \mathbb{E}_{s\sim\rho^\pi, a\sim\pi_\theta}[r(s,a)] \tag{2}$$

$$\theta_{t+1} = \theta_t + \alpha \widehat{\nabla_\theta L(\theta_t)} \tag{3}$$

where $\rho^\pi(s) = \sum_{t=0}^{\infty} \gamma^t p(s_t = s)$ is the unnormalized discounted state visitation frequency in the limit [9] and α is the learning rate.

C. Proximal Policy Optimization

Proximal Policy Optimization [5] is one of the most commonly used policy gradient methods. Among the several variants for the performance measures available, we consider the clipped surrogate objective as in [5], as follows:

$$L_t^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min(r_t(\theta)\hat{A}, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t) \right] \tag{4}$$

where θ_{old} is the vector of policy parameters before the update; $r_t(\theta)$ denotes the probability ratio $\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$; ϵ is a hyperparameter used to clip the probability ratio by $\text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)$, avoiding large policy updates [5]; and $\hat{A}_t$ is an estimator of the advantage function at timestep t, which weights the ratio $r_t(\theta)$. Here, $\hat{\mathbb{E}}_t$ denotes an empirical average over a finite set of samples.

The implementation of policy gradient considers a truncated version of the Generalized Advantage Estimator (GAE), as in [10]:

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + ... + ... + (\gamma\lambda)^{T-t+1}\delta_{T-1} \tag{5}$$

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t), \tag{6}$$

where the policy is run for T timesteps (with T less than the episode size). As commonly used in the literature, γ and λ are discount factor and GAE parameter, respectively. To perform a policy update, each of N (parallel) actors collect T timesteps of data. Then we construct the surrogate loss on these NT timesteps of data, and optimize it with ADAM algorithm [11] with a learning rate α, in mini-batches of size $m \leq NT$ for K epochs. Notice that $V_\theta(s)$ in GAE is learned simultaneously in order to reduce the variance of the advantage-function estimator.

Once we use a neural network architecture that shares parameters between the policy and value function, we must use a loss function that combines the policy surrogate and a value function error term. This objective is further augmented by adding an entropy term to ensure sufficient exploration. Combining these terms, we obtain the following objective, which is (approximately) maximized at each iteration [5]:

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t \left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t) \right], \tag{7}$$

where S denotes an entropy bonus; L_t^{VF} is the value function (VF) squared-error loss $(V_\theta(s_t) - V_t^{\text{targ}})^2$, with $V_t^{\text{targ}} = r_t + \gamma V_\theta(s_{t+1})$; and $c1$, $c2$ are coefficients for the VF loss and entropy term, respectively.

D. Gaussian Distribution

The Gaussian distribution is defined by the following probability density function:

$$f(x,\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{1}{2} \left(\frac{x-\mu}{\sigma} \right)^2 \right) \tag{8}$$

whose parameters μ and σ are to be estimated by a deep neural network that models a so-called Gaussian policy, i.e., a parametrized policy $\pi_\theta(a|s) \sim \mathcal{N}(\mu, \sigma^2)$.

Therefore, when acting in stochastic mode, the agent samples the policy whereas in deterministic mode, $\pi(a|s) = \mu$. Since the Gaussian distribution has an infinite support, these sampled actions are clipped to the agent's bounded action space.

E. Beta Distribution

The Beta distribution has finite support and can be intuitively understood as the probability of success, where $\alpha - 1$ and $\beta - 1$ can be thought of as the counts of successes and failures from the prior knowledge, respectively. For a random variable $x \in [0, 1]$, the Beta probability density function is given by:

$$h(x : \alpha, \beta) = \frac{\Gamma(\alpha\beta)}{\Gamma(\alpha)\Gamma(\beta)} x^{\alpha-1}(1-x)^{\beta-1}, \tag{9}$$

where $\Gamma(.)$ is the Gamma function, which extends the factorial to real numbers. For $\alpha, \beta > 1$, the distribution is uni-modal, as illustrated in Fig. 1. When acting deterministically, the Beta policy outputs $\pi_\theta(a|s) = \alpha/(\alpha+\beta)$. The α, β parameters that define the shape of the function are obtained as outputs of a deep neural network representing the parametrized policy $\pi_\theta(a|s)$.

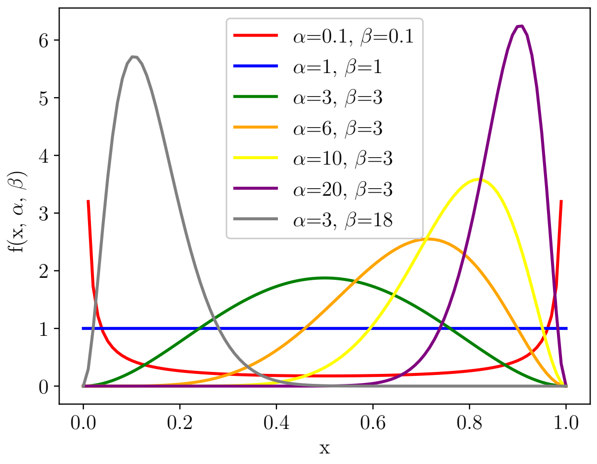

Fig. 1. Beta probability density function for different α, β pairs

F. Bias due to boundary effect

Modeling a bounded action space by a probability distribution with infinite support possibly introduces bias. As a result, the biased gradient imposes additional difficult in finding the optimal policy using reinforcement learning. The policy gradient estimator to optimize the parameters θ in (3), using Q as the target, can be obtained by differentiating (1), as follows:

$$\nabla_\theta L(\theta_t) = \int_{\mathbb{S}} \rho^\pi(s) \int_{\mathbb{A}} \pi_\theta(a|s) \nabla_\theta \log \pi_\theta(a|s) Q^\pi(s|a) da ds \tag{10}$$

where $Q^{\pi_\theta}(s, a)$ is a state-action value function for a policy π_θ. Thus, the policy gradient estimator using Q as the target is given by:

$$g_q = \nabla_\theta \log \pi_\theta(a|s) Q^{\pi_\theta}(s, a) \tag{11}$$

This gradient is estimated by averaging n samples with a fixed policy π_θ, so that

$$\nabla_\theta L(\theta_t) = \frac{1}{n} \sum_{i=1}^{n} g_q \rightarrow \mathbb{E}[g_q] = \nabla_\theta L(\theta_t) \text{ as } n \rightarrow \infty \tag{12}$$

Let $A = [-h, h]$ be an uni-dimensional action space, with $a \in A$. In the case of an infinite support policy, an action $a' \notin A$ is eventually sampled, to which the environments responds as if the action is either h or $-h$. The biased policy gradient estimator would be given by $g'_q = \nabla_\theta \log \pi_\theta(a|s) Q^\pi(s|a')$ in this case. Besides, focusing on the inner integral of (10), the bias is computed as follows (also shown in [8]):

$$\begin{aligned}
&\mathbb{E}[g'_q] - \nabla_\theta L(\theta) \\
&= \mathbb{E}_s \left[\int_{-\infty}^{\infty} \pi_\theta(a|s) \nabla_\theta \log \pi(a|s) Q^\pi(s, a') da \right] - \nabla_\theta J(\theta) \\
&= \mathbb{E}_s \Bigg[\int_{-\infty}^{-h} \pi_\theta(a|s) \nabla_\theta \log \pi_\theta(a|s) [Q^\pi(s, -h) - Q^\pi(s, a)] da \\
&\quad + \int_{h}^{\infty} \pi_\theta(s|a) \nabla_\theta \log \pi_\theta(a|s) [Q^\pi(s, h) - Q^\pi(s, a)] da \Bigg]
\end{aligned} \tag{13}$$

These last two integrals evaluate to zero if the policy's distribution support is within the action space A.

III. Experiments

This section presents results for the proximal policy gradient method (PPO) on two continuous control problems from OpenAi gym [12]: the LunarLanderContinuous-v2 with low-dimensional state space; and the CarRacing-v0 with high-dimensional image input (Table I).

For all architectures, the last two layers output two 2-dimensional real vectors. For the Gaussian distribution, each dimension of the policy outputs its mean μ and its standard deviation σ, whereas for the Beta distribution the network outputs its parameters α and β >1. Here, 1 is added to a softplus layer $log(1 + exp(x))$ to ensure both α and β are larger than 1. Our implementation for PPO was based on a popular reinforcement learning library found in [13].

TABLE I
Environments

Environment	$\|\|\mathcal{S}\|\|$	$\|\|\mathcal{A}\|\|$
LunarLander	8	2
CarRacing	96x96x3	3

For each environment, we trained five models using different seeds for both the Gaussian and Beta distributions for a fixed number of total time steps. After completing the training, each model was evaluated in 100 consecutive episodes in both stochastic mode (sampling from the distribution) and deterministic mode (using the average of each distribution as the optimal action).

The hyperparameters for PPO can be seen in Table II for both control problems. Notice that the PPO configuration for the Lunar Lander was adapted from the one used for the MuJoCo environment in [5], whereas for the CarRacing, the parameters found in Atari [1] were used as a starting point.

TABLE II
HYPERPARAMETERS FOR TRAINING

	Lunar Lander	CarRacing
Horizon (T)	2048	500
Parallel environments (N)	1	8
Adam step size (lr)	$3 \times 10^{-4} \times \alpha$	$2.5 \times 10^{-4} \times \alpha$
Number of PPO epochs (K)	10	10
Mini-batch size (m)	32	64
Discount (γ)	0.99	0.99
GAE parameter (λ)	0.95	0.95
Clipping parameter (ϵ)	0.2	0.1
Value Function coefficient (c_1)	0.5	0.5
Entropy coefficient (c_2)	0	0.01
Total timesteps	10^6	5×10^6

α is linearly annealed from 1 to 0 over the course of learning

A. *LunarLanderContinuous-v2*

The LunarLanderContinuous-v2 environment simulates the landing of a space module on the moon. The overall objective corresponds to landing the module on the lunar surface delimited by two flags, approaching zero speed at the final step (Fig. 2). It has an unbounded, 8-dimensional observation space and a 2-dimensional action space. The actions are the main engine throttle and the secondary engine throttle, both bounded in the interval $[0, 1]$. The agent loses points for firing up the engines and for crashing (landing at high speed). The simulation is considered solved if the agent manages to score at least 200 points [14].

The agent follows an actor-critic framework, where the actor $\pi_\theta(a|s)$ consists of a neural network made of 3 fully-connected layers of 64 units each, with $\tanh$ activation functions. The output layer has 2 linear neurons to model either the Gaussian or the Beta distribution over the actions. The critic $V_{\theta_v}(s)$ does not share layers with the actor, but has an equivalent architecture of 3 hidden layers with only one output neuron which represents the value function.

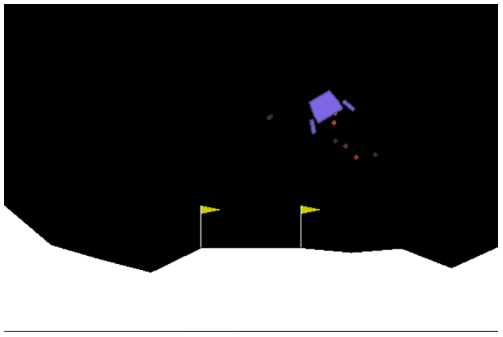

Fig. 2. LunarLanderContinuous-v2 Environment

B. *CarRacing-v0*

The CarRacing-v0 environment [15] simulates an autonomous driving environment in 2D. For each episode, a random track with 12 curves is generated. Each track is comprised of N tiles, with N ranging from 250 to 350. The agent receives 1000/N points for visiting each tile and loses 0.1 point for each frame. The episode ends in one of three situations:

1) Agent visited all tiles
2) Agent does not visit all tiles in 1000 frames
3) Agent gets too far way from the track and falls in the abiss (-100 points added)

Therefore, if the agent visits all tiles in 732 frames, the reward is 1000 - 0.1*732 = 926.8 points. Should the agent miss one or more tiles in its first lap attempt, the episode keeps on until the agent visit missing frame or the time limit is reached. The task is considered to be solved if the agent is able to get an average reward of at least 900 points in 100 consecutive trials (episodes).

The observation space consists of top down images (Fig. 3) of 96x96 pixels and three (RGB) color channels. The latest four image frames were stacked and given as input to the agent's network after rescaling and preprocessing them to gray scale (totalling 84x84x4 input dimensions). The action space has three dimensions: one encodes the steering angle and is bounded in the interval $[-1, +1]$. The other two dimensions encode throttle and brake, both bounded to $[0, 1]$.

For our implementation, throttle and brake have been merged on a single dimension so that on a given step, the agent does not simultaneously accelerates and brakes. We believe this is a more representative structure of real world systems: separated control inputs (throttle/brake) but single activation mechanism (right foot). In practice, one output neuron is responsible for both actions, making the output of the agent to be a two-dimensional vector. With this approach, we were able to make the agent learn effectively, mainly because it does not enter a deadlock state resulting from accelerating and braking at the same time. If we did not follow this approach,

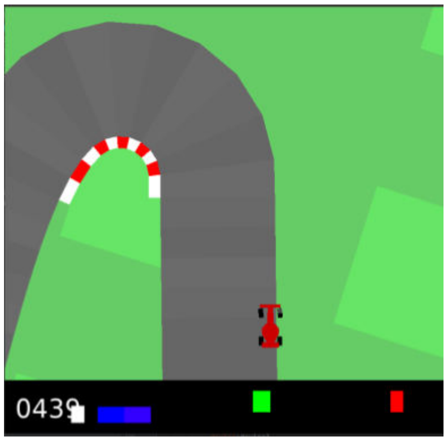

Fig. 3. CarRacing-v0 Environment

learning to control the vehicle would not take place. So far, we were not able to find other work in the literature that takes advantage on the aforementioned approach. Also, notice that we have not changed the original reward signal as some other works might have done.

The actor-critic network resembles that of [1] with respect to the shared encoder base comprised of the first 3 convolutional layers. Instead of connecting directly to the output layer as in [1], the shared base has an additional fully connected (FC) layer with 512 units. The critic $V_{\theta_v}(s)$ specializes further with its exclusive 1 FC layer of 512 units, that connects to a final output. The actor $\pi_\theta(a|s)$ has its own 2 FC layers with 512 units each on top of the shared base. The output layer is equivalent to the one from Section III-A, but its two neurons now refer to the steering angle or acceleration (brake/throttle).

IV. Results and Discussion

A. LunarLanderContinuous-v2

For the LunarLanderContinuous-v2 environment, we observe that using a Beta distribution allow for both a faster convergence and higher total reward during training. Five agents were trained with the same hyperparameters and different seeds.

After a million times steps, training is frozen and we evaluated each agent for 100 episodes in deterministic mode (using the mean of the policy's distribution as the action) and in stochastic mode (sampling the policy's distribution).

For the Gaussian distribution, we observe that the performance of the agents hovers around 225.7 and 219.0 points for the deterministic and stochastic policies, respectively. For the Beta distribution, we observe the agents perform at 267.0 (deterministic) and 273.6 points (stochastic). It is worth noting

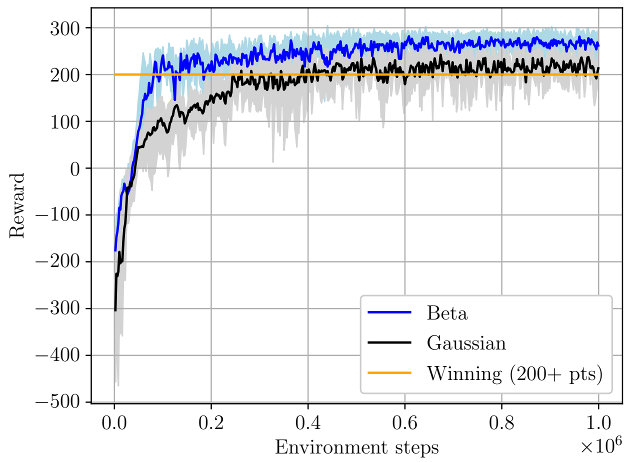

Fig. 4. Average rewards for five agents on the Lunar Lander task trained with Beta or Gaussian distribution for 1 million steps. The solid line represents the mean over a moving window of the previous 10 episodes for these five agents. The shaded area represents the interval between the minimum reward and maximum reward.

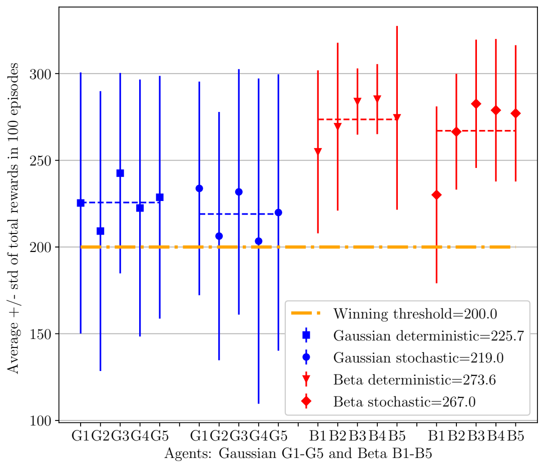

Fig. 5. Lunar Lander results: comparison of the Gaussian policy to the Beta policy in terms of the average rewards obtained by agents for 100 consecutive episodes after training. In blue (red), the average reward and standard deviation for each one of 5 agents using the Gaussian (Beta) policy. Both deterministic and stochastic policies were employed for evaluation. The winning threshold given by the horizontal black line represents the minimum threshold for successful completion of the task. Agents powered by the Beta distribution achieved superior performance and less variance.

that Agents B4 and B5, which were trained with the Beta distribution, were able to score at least 200 points for all 100 episodes (Fig. 5) whereas the best agent trained with the Gaussian distribution (G3, deterministic policy) was able to score above the 200 points threshold for 92% of the 100 evaluation episodes. We can also observe that the variance of the Gaussian policy is higher than that of the Beta policy, even at the latest training iterations (Fig. 4) or after training ends (Fig. 5).

B. CarRacing-v0

For the CarRacing-v0 environment, the number of agent-environment interactions was fixed to 5 million steps during training. Afterwards, an evaluation of the agent's performance takes place, measured as the average reward in 100 consecutive episodes. The task is solved if this value is at least 900 points. We have observed that agents trained with Beta and Gaussian distributions have a similar convergence rate during training time. In Figure 6, we show the average reward over a moving window of 10 episodes, along the training process. Each policy optimization takes in 500 environment steps across 8 parallel environments.

Using the performance measure for 100 consecutive episodes training, in Fig. 7, we show that the stochastic policy presented better average performance than the deterministic policy for both distributions. For the Gaussian distribution, we observe that the five agents with the deterministic policy fail to follow the track, presenting an average score of 370.4 points that is much lower than the required 900 score points to solve the task. In stochastic mode, the policy presents an improved

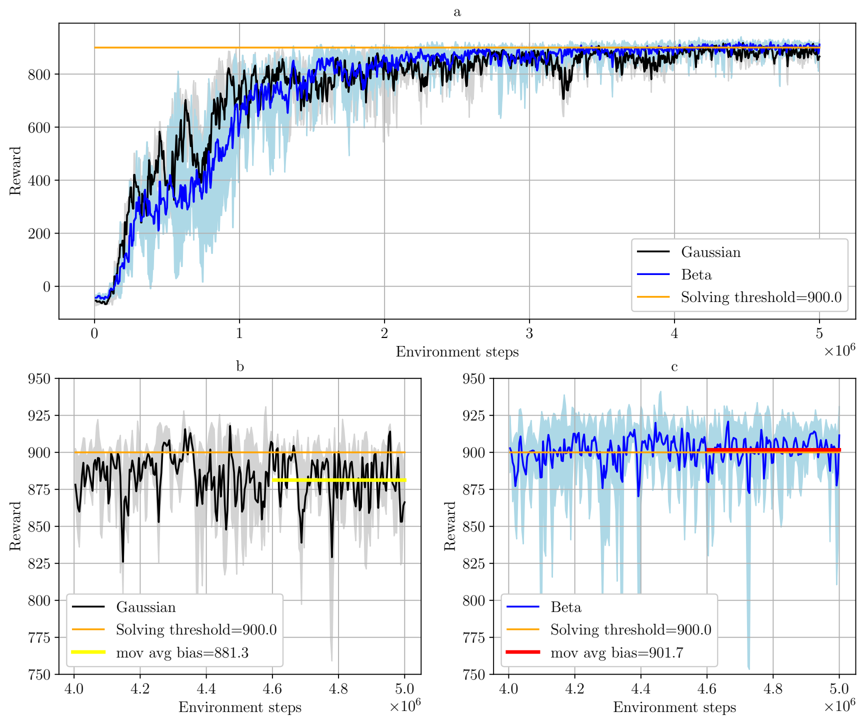

Fig. 6. Average rewards for 5 agents with different seeds for the CarRacing environment, plotted equivalently to Fig. 4. The final performance is shown on the bottom plots at a bigger scale, for agents with Gaussian policy (bottom left) and Beta policy (bottom right).

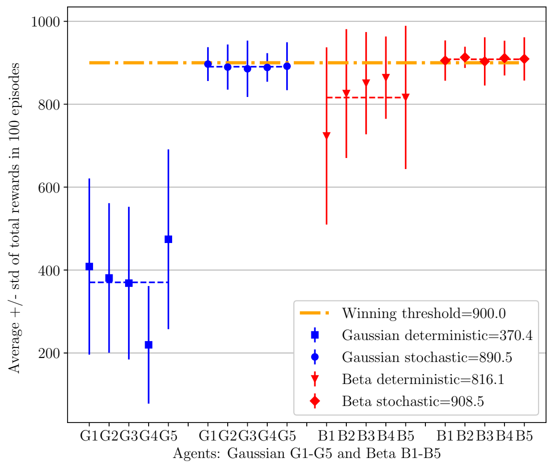

Fig. 7. Car Racing results after training: The evaluation followed the same procedure used for Lunar Lander (plots can be understood as in Fig. 5). For this task, the stochastic policy clearly yielded better performance than the deterministic one.

performance with an average score of 890.5 points, although in 38% of the 100 episodes the agents were not able to pass the winning threshold. For the Beta distribution, the agents' performance with the deterministic policy improves over the Gaussian policy by 320%, with average score of 816.1 points. These agents surpass the winning threshold in 26% of the evaluation episodes. In the stochastic mode, all agents were able to score above the winning threshold in at least 60% of the 100 of episodes played by each agent. All five agents with the Beta policy were able to successfully solve the game since each one of them reached a performance higher 900 points. This was not the case for the stochastic Gaussian policy, where each agent performed less than the threshold of 900 points. The best performing agent, B2, consistently reached scores above the other five agents, and it's the chosen agent to compare our approach with other works in the literature in the next section. Fig. 8 shows the resulting Gaussian and Beta policies at a specific timestep of the simulation, after training, when the car was about to turn left as it can be seen on the image fed to the policy network. The sampled distributions for both policies show that the Gaussian distribution, with its infinite support, falls outside the bounded action space, what is associated with the bias calculated in Section II-F. On the other hand, the Beta distribution fits well within the bounded action space, yielding an unbiased policy gradient estimator.

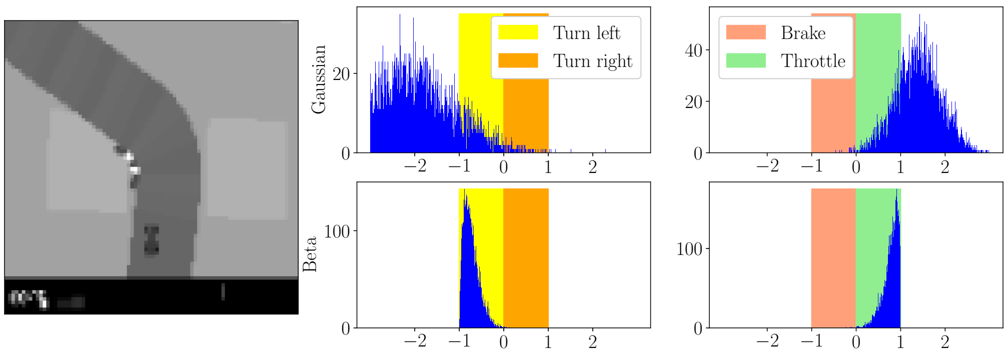

Fig. 8. Illustration of the Gaussian and Beta stochastic policy distributions in relation to the action space of the CarRacing environment. For a fixed observation s (preprocessed image in left plot), we sampled the Gaussian and Beta policies for 5000 actions. For the Gaussian distribution, a significant portion of the actions fall out of the valid direction and brake/throttle range (both $[-1, 1]$), whereas for the Beta distributions, all actions fall within boundaries.

C. Considerations on the CarRacing-v0 environment and other approaches

Simulation environments designed as test beds for reinforcement learning algorithms are primarily used in two ways:

1) To benchmark new algorithms or techniques without focusing particularly on a specific task;
2) To develop methods to solve a specific simulation task or benchmark, such as scoring more than 900 points on average in 100 consecutive runs for the CarRacing-v0 env, in an attempt to beat the other reported results.

Although our primary objective was the former, we emphasize that our work happens to fulfill to the latter as well. OpenAI CarRacing-v0 Leaderboard [16] hosts a series of self-reported scores. We compare our results only to those found in peer-reviewed articles (Table III), since they provide a basis for comparison and discussion.

Among the works that use Car Racing as a test bed, [17] claim to have been the first to solve the problem, using a recurrent world model. Other attempts included Deep Q-Networks with action-space discretization [18] and Genetic algorithms [19]. Other work that uses the Car Racing environment for benchmarking other algorithms are [20] and [21], and have been included for reference.

TABLE III
CarRacing-v0 Leaderboard

Method	Average Evaluation Score
PPO with Beta (Ours)	**913 +/- 26**
World models [17]	906 +/- 21
Adapted DQN [18]	905 +/- 24
Genetic Algorithms [19]	903 +/- 72
PPO with Gaussian (Ours)	897 +/- 41
Weight Agnostic NN [20]	893 +/- 74
PPO [21]	740 +/- 86
Random agent	-32 +/- 6

V. Conclusions

In this study, we observed that agents trained with PPO using a Beta distribution for the stochastic policy presented faster and more stable convergence of the training process (mainly for the Lunar Lander task), while their final performance was significantly superior to those trained with a Gaussian distribution. Thus, the Beta distribution is better able to satisfy the requirements of real-world applications with bounded action spaces, overcoming the estimation bias of the Gaussian policy.

Our results also show that continuous control with bounded action space for challenging car racing with random tracks and a high-dimensionality of the observation space (based on images) is much facilitated when the Beta distribution is employed. In fact, the agent's success in this task is considerably affected by this approach, achieving the best score so far on the CarRacing-v0 Leaderboard among the published work in literature. Finally, the results suggest that the Beta distribution should be a standard choice for those type of tasks.

Originally proposed in [8], the Beta distribution was tested in their work with TRPO/ACER on Atari games, which have high-dimensional observation space, but a discrete action space; and on robotic control tasks with a continuous action space and a low-dimensional observation space. In this work, we proposed to use the Beta distribution with PPO on high-dimensional image inputs and continuous action spaces.

We plan to extend these experiments to other types of reinforcement learning algorithms that are more sample efficient, in an attempt to verify if the Beta distribution transfers to other setups. Besides, experiments with more complex autonomous navigation in urban scenarios could benefit from the faster and more stable convergence as the training of end-to-end models is not a trivial task.

Acknowledgment

The authors would like to thank CAPES/Brazil for the financial sponsorship.

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[2] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton *et al.*, "Mastering the game of go without human knowledge," *nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[3] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[4] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *International conference on machine learning*. PMLR, 2015, pp. 1889–1897.

[5] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[6] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International conference on machine learning*. PMLR, 2016, pp. 1928–1937.

[7] Z. Wang, V. Bapst, N. Heess, V. Mnih, R. Munos, K. Kavukcuoglu, and N. de Freitas, "Sample efficient actor-critic with experience replay," *arXiv preprint arXiv:1611.01224*, 2016.

[8] P.-W. Chou, D. Maturana, and S. Scherer, "Improving stochastic policy gradients in continuous control with deep reinforcement learning using the beta distribution," in *International conference on machine learning*. PMLR, 2017, pp. 834–843.

[9] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Advances in neural information processing systems*, 2000, pp. 1057–1063.

[10] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation," *arXiv preprint arXiv:1506.02438*, 2015.

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

[12] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[13] I. Kostrikov, "Pytorch implementations of reinforcement learning algorithms," https://github.com/ikostrikov/pytorch-a2c-ppo-acktr-gail, 2018.

[14] O. Klimov, https://github.com/openai/gym/blob/master/gym/envs/box2d/lunar_lander.py.

[15] ——, https://github.com/openai/gym/blob/master/gym/envs/box2d/car_racing.py.

[16] OpenAI, https://github.com/openai/gym/wiki/Leaderboard#CarRacing-v0.

[17] D. Ha and J. Schmidhuber, "Recurrent world models facilitate policy evolution," 2018.

[18] P. Rodrigues and S. Vieira, "Optimizing agent training with deep q-learning on a self-driving reinforcement learning environment," in *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE, 2020, pp. 745–752.

[19] S. Risi and K. O. Stanley, "Deep neuroevolution of recurrent and discrete world models," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2019, pp. 456–462.

[20] A. Gaier and D. Ha, "Weight agnostic neural networks," *arXiv preprint arXiv:1906.04358*, 2019.

[21] R. Jena, C. Liu, and K. Sycara, "Augmenting gail with bc for sample efficient imitation learning," *arXiv preprint arXiv:2001.07798*, 2020.