Winter Research Summary - PPO-Clip for continuous and bounded action spaces using the Beta distribution

Jack Cashman, under the supervision of Dr Nan Ye

July 20, 2024

1 Introduction

The following is a brief summary of *some* of the work that I have completed in the UQ Winter Research Program for the School of Mathematics and Physics. As part of this, my work has been supervised by Dr Nan Ye, whose feedback I am grateful for.

I have investigated the effectiveness of using a beta distribution as the prior for the policy in the PPO-Clip algorithm. Specifically, I have considered control problems with a continuous state space, and a continuous and bounded action space.

All code implementation can be found on my fork of OpenAI's clean `clean-rl` repository, available here. Specifically, I have implemented two files, `ppo_cts_action_beta_prior_single_net.py` and `ppo_cts_action_beta_prior_double_net.py`. These are single-file implementations of the PPO-clip algorithm for use in continuous and bounded action environments which assume a beta distribution for the prior of the stochastic policy. In addition to this, I have also made available the resources used in my weekly presentations as part of the program via the GitHub repo.

2 Background Knowledge and Motivation

In the continuous action setting, Proximal Policy Optimisation algorithms learn a stochastic policy, from which actions can be sampled. Prior to the learning, one assumes a prior (parameteric) distribution over the policy, and then learns a mapping from states (or observations) to a vector that parameterises the prior distribution. Typically, this mapping is in the form of a neural network.

It is common practice for the prior to be taken as a diagonal Gaussian distribution. That is, given some action space $\mathcal{A} \subseteq \mathbb{R}^d$, we learn a mapping from a state to a mean vector $\boldsymbol{\mu} \in \mathbb{R}^d$, and a vector of variances $\boldsymbol{\Sigma} \in \mathbb{R}^d$. Given this, actions can be gathered from sampling from a $\mathcal{N}(\boldsymbol{\mu}, \text{diag}(\boldsymbol{\Sigma}))$ distribution.

The support for such a distribution is $\mathbb{R}^d$, but in many settings we have that $\mathcal{A} \neq \mathbb{R}^d$, so this induces an estimation bias for actions on the boundary of $\mathcal{A}$, which are more likely to be chosen. One such example of when $\mathcal{A} \neq \mathbb{R}^d$ is when $\mathcal{A}$ is a hyper-rectangle. That is, we have that:

$$\mathcal{A} = \left\{ \begin{pmatrix} a^{(1)} & \cdots & a^{(d)} \end{pmatrix}^\intercal : a_L^{(i)} \leq a^{(i)} \leq a_H^{(i)}, \quad 1 \leq i \leq d \right\}$$

More intuitively, each of the d components of any specific action vector have a constant lower and upper bounds. This setting arises extremely often in continuous control environments. For example, if an agent is learning to rotate a turret, according to a spherical co-ordinate system, then we would have $\mathcal{A} = \{(\theta, \phi) : 0 \leq \theta \leq 2\pi, 0 \leq \phi \leq \pi\}$. In my research I propose that assuming a beta distribution as the prior for the policy serves as a promising alternative to the traditional approach and eliminates this estimation bias.

The probability density function of d independently distributed beta random variables $X_1, \dots X_d$, where $X_i \sim \text{Beta}(\alpha_i, \beta_i)$ is given by:

$$f_{\mathbf{X}}(x_1, \dots, x_d) \propto \prod_{i=1}^{d} x^{\alpha_i - 1}(1-x)^{\beta_i - 1}$$

when, for each $1 \leq i \leq d$, we have that $x_i \in (0,1)$, and f is 0 elsewhere. This bounded support is the motivation behind assuming a beta distribution as the prior for the policy. Namely, given the affine mapping $\mathbf{T} : (0,1)^d \to \mathcal{A}$ defined by:

$$\mathbf{T}(\mathbf{p}) = \left(\text{diag}\begin{pmatrix} a_H^{(1)} - a_L^{(1)} & \cdots & a_H^{(d)} - a_L^{(d)} \end{pmatrix}\right)\mathbf{p} + \begin{pmatrix} a_L^{(1)} & \cdots & a_L^{(d)} \end{pmatrix}^{\mathsf{T}}$$

We are able to map between the support of the beta distribution (which is required for efficient sampling), and the action space, $\mathcal{A}$. Note that provided that $a_H^{(i)} \neq a_L^{(i)}$ for $1 \leq i \leq d$, $\mathbf{T}$ is invertible. Given this, let $f_{\mathbf{Y}}(y_1, \dots, y_d)$ be the distribution over $\mathcal{A}$. Then, the analytical expression for $f_{\mathbf{Y}}$ is:

$$f_{\mathbf{Y}}(y_1, \dots, y_d) = \left(\prod_{i=1}^{d} a_H^{(i)} - a_L^{(i)}\right)^{-1} f_{\mathbf{X}}\left(\frac{y_1 - a_L^{(1)}}{a_H^{(1)} - a_L^{(1)}}, \dots, \frac{y_d - a_L^{(d)}}{a_H^{(d)} - a_L^{(d)}}\right)$$

However, this is not necessary in implementation as we can simply sample from $\begin{pmatrix} X_1 & \cdots & X_d \end{pmatrix}^{\mathsf{T}}$ using well designed `PyTorch` libraries, and then map this to the corresponding action.

Hence, given a state, we propose that to retrieve an action according to a learnt policy, one must:

1. Use a function approximator to map the state to vectors $\begin{pmatrix} \alpha_1 & \cdots & \alpha_d \end{pmatrix}^{\mathsf{T}}$ and $\begin{pmatrix} \beta_1 & \cdots & \beta_d \end{pmatrix}^{\mathsf{T}}$
2. Draw a sample from the random vector $\begin{pmatrix} X_1 & \cdots & X_d \end{pmatrix}$, where for $1 \leq i \leq d$ we have $X_i \sim \text{Beta}(\alpha_i, \beta_i)$
3. Map the sample to an associated action via the affine transformation $\mathbf{T}$

The ensuing sections of the report focus on implementation details with the ultimate goal of maximising a (smoothed) episodic reward over a variety of continuous control environments that satisfy our definition of $\mathcal{A}$ as presented on the bottom of page 1.

3 Implementation details and Experiments

3.1 Modification for Numerical Stability

Note that by definition we have that $\alpha_i, \beta_i > 0$ for $1 \leq i \leq d$. However, if at least one of the α_i or β_i is strictly less than 1 the joint probability density function, f, is unbounded. This is problematic from a numerical stability stand-point. Specifically, in my implementations I need to call a function that returns $\log(f(\mathbf{x}))$ for some $\mathbf{x} \in (0,1)^d$. If f is unbounded then it is entirely possible for $f(\mathbf{x})$ to be arbitrarily large. If it is large enough then the function incorrectly returns $\log(f(\mathbf{x})) = \infty$, which breaks the algorithm.

In order to rectify this, I have required that all $\alpha_i, \beta_i \geq 1$. Under this restriction, f is a bounded function so the algorithm is numerically stable. I will expand on how I have enforced this restriction in the next section.

3.2 Network architectures for predicting α and β

As I have just mentioned, for numerical stability I require that $\alpha_i, \beta_i \geq 1$. In order to enforce this, the (un-restricted) outputs of the function approximator for the parameters are pushed through the ReLU function (to map $\mathbb{R}^d \to [0,\infty)^d$). Then, each component is incremented by 1 (to map $[0,\infty)^d \to [1,\infty)^d$). I did explore alternative activation functions, and the results are available here. So, the current architecture is:

$$\begin{tikzcd}
\text{state} \arrow[r] & \boxed{\text{Neural Net}} \arrow[r] & \text{ReLU} + 1 \arrow[r] & \text{Action}
\end{tikzcd}$$

Still, we must determine a suitable Neural Net to use in the mapping from state to the action. Furthermore, we could either use a single network to output the vector $\begin{pmatrix}\alpha_1 & \cdots & \alpha_d & \beta_1 & \cdots & \beta_d\end{pmatrix}^\intercal$. Or, we could use a network to output the $\begin{pmatrix}\alpha_1 & \cdots & \alpha_d\end{pmatrix}^\intercal$ and a separate network to output $\begin{pmatrix}\beta_1 & \cdots & \beta_d\end{pmatrix}^\intercal$. I have implemented the former in `ppo_cts_action_beta_prior_single_net.py` and the latter in `ppo_cts_action_beta_prior_double_net.py`. In the former, I use 2 NNs, each with 2 hidden layers of size 64. In the latter I've used one network with 2 hidden layers; one of size 96 and the other of size 128.

Now, in order to benchmark the algorithms against one another, I've tested each algorithm on 3 `Mujoco` environments. Specifically, I've used the Half Cheetah, Walker2D, and Hopper environments. I chose these 3 environments as they're all complex continuous control tasks which meet the requirement of the action space being some hyper-rectangle. Each algorithm was tested on 5 random seeds for each environment, and the maximum (smoothed) episodic reward in the first 100,000 steps was logged via `tensorboard`:

	`Cheetah`	`Hopper`	`Walker`
Double Net	284 ± 46.6	358.2 ± 43.2	347 ± 20.3
Single Net	631 ± 48.4	383.4 ± 15.6	346.4 ± 10.8

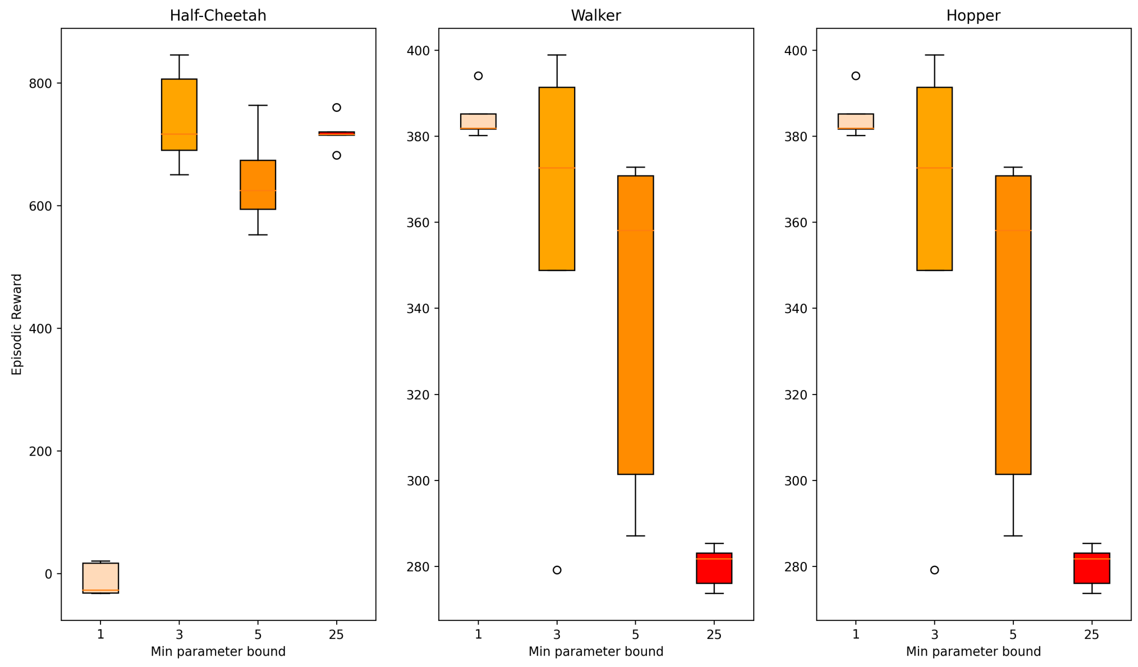
The single network performs better than the double network in the form of both increased averages, and a decreased standard deviation. Furthermore, the back propagation required to update the network is cheaper when only one NN is used. As a result, the single network implementation runs slightly faster than the double network in implementation, which is an added bonus!

3.3 Lower bounds on α and β

As I've previously mentioned, we require that all α_i and β_i are greater than or equal to 1 to ensure that the joint density function is bounded so that the algorithm is numerically stable. However, in implementation I found the that α_i and β_i produced by the network are always close to the lower bound (normally < 4).

In this setting, the learnt policy is quite stochastic. In fact, at the lower bound the function is identical to the uniform distribution. In order to restrict this, I've had the idea that instead of requiring $\alpha_i, \beta_i \geq 1$, we could instead restrict $\alpha_i, \beta_i \geq \text{Lb}$ for some suitable lower bound Lb. In both of the files I have produced, this lower bound is represented by the 'lb' argument in the `Args` data class.

Again, in order to compare results, I have performed similar testing as in Section 3.2. 4 lower bounds have been tested on 3 `Mujoco` environments for 5 seeds in each. The results are shown on the box plots below:



Overall, introducing a lower bound on the parameters of approximately 3 is optimal in the sense of maximising the episodic reward. This result is especially significant in the `Half-Cheetah` environment. As expected, when lower bounding the parameters with 25, the algorithms performance deteriorated due to the fact that the learn policy has very low variance. Moreover, this is reflected in the fact that the standard deviation when lb = 25 is relatively small.

4 Conclusion

Overall, we've shown that assuming a beta prior for the policy in specific continuous and bounded action spaces is a promising alternative to the traditional diagonal gaussian prior. Specifically, a sample of the performance across 3 `Mujoco` environments with 5 samples on each is:

	`Cheetah`	`Hopper`	`Walker`
Gaussian Prior	32.5 ± 139.4	765.6 ± 237.5	365.7 ± 13.6
Beta Prior	742 ± 72.8	415.5 ± 59.6	384.6 ± 5

There are other topics that I investigated during the program (such as regularising the stochasticity of the learnt policy via maximising the entropy of the policy, etc.) which I have chosen not to include in this report as the results were not significant enough to warrant further investigation given the limited time frame of the program. Nevertheless, slides pertaining to my presentation of these results are available through the `project_presentations` folder on my GitHub.

Overall the program has been a great opportunity, and I've learnt a great deal about (deep) reinforcement learning, and my own interest in research.

Cheers,
Jack Cashman