Week 3 Summary - Making Robots Learn Through Trial and Error

Jack Cashman

July 9, 2024

Summary and Motivation

In week 2 I found that using the Beta distribution as a prior for the policy in PPO yielded mixed results. In some instances, such as the `Cheetah` environment, it greatly boosted performance. However, in other environments such as the `Hopper` and `Walker` environments, using the Beta distribution results in extremely poor performance compared to the `Clean-rl` implementation of PPO, which uses a Gaussian distribution.

This week, I have attempted to improve the performance of PPO-Clip with a beta prior through addressing the numerical instability I faced last week, and then considering 3 different NN architectures in order to predict the α and β parameters for the Beta(α, β) policy.

Addressing numerical instability from last week

As I mentioned last week, I was encountering some issues with regard to the numerical stability of using the beta distribution as a prior. I've since managed to fix this. The pdf of the beta distribution, f_B, is:

$$f_B(x; \alpha, \beta) \propto x^{\alpha-1}(1-x)^{\beta-1}$$

Now, for $\alpha < 1$, as $x \to 0$ we have that $f_B \to \infty$. Similarly, if $\beta < 1$ then as $x \to 1$ we have that $f_B \to \infty$. This was the root cause of the instability. Essentially, if the actions taken by the agent were close to the end of the distributions, f_B takes on very large values. PyTorch has a method `log_prob` that is used in the `get_action_and_value` function, which returns the logarithm of f_B evaluated at some point. If this point is very close to either 0 or 1, and $\alpha < 1$ or $\beta < 1$ then f_B was so large that the return was ∞ due to the asymptote, which was breaking the code. In order to fix this, I've imposed that both $\alpha, \beta > 1$. Under the restriction, f_B is bounded for finite α and β, so all computation is stable provided that the learning rate is sufficiently small.

Now that the algorithm is stable, I am able to experiment with different methods of using the NNs in order to compute the α and β parameters.

Bench-Marking Performance

In all of the ensuing testing, we will benchmark our results against the `Clean-rl` implementation of PPO for continuous action spaces. Specifically, we will test on the `Mujoco` environments `Cheetah`, `Hopper`, and `Walker` for $100,000$ time-steps. Moreover, I will complete 5 runs on each environment using the random seeds $13, 21, 45, 66$ and 100. The results of the `Clean-rl` implementation on the seeds are:

	Seed 13	Seed 21	Seed 45	Seed 66	Seed 100	mean $\pm$ sd
`Cheetah`	266	-103	110.2	-18.5	-92.36	32.5 ± 139.4
`Hopper`	460.3	977.4	632.2	1103	655	765.6 ± 237.5
`Walker`	375.8	351.8	380	346.9	374.1	365.7 ± 13.6

Now that I have a benchmark, I will clarify the plan for the briefing. In each of the following experiments I will use one NN with 1 hidden layer of size 64 to predict a vector of α parameters, and an identical network to predict a vector of β parameters.

In each of the experiments, the function that is applied to the output of the second layer of these networks will change. I will denote this function $\mathbf{\Phi}$. I will test 3 different options for $\mathbf{\Phi}$, and examine the performance of each against the above table. The choices of $\mathbf{\Phi}$ that I will examine are:

$$1: \ \mathbf{\Phi}(\mathbf{x}) = \exp(\mathbf{x}) + \mathbf{1} \qquad 2: \ \mathbf{\Phi}(\mathbf{x}) = \text{softplus}(\mathbf{x}) + \mathbf{1} \qquad 3: \ \mathbf{\Phi}(\mathbf{x}) = c \times \text{Sigmoid}(\mathbf{x}) + \mathbf{1}$$

Where $\mathbf{1}$ is a vector of ones, which is used to ensure that all α and β are greater than or equal to 1, which is required for numerical stability. I will reference these as architectures 1 through to 3. For each, I will perform some hyper-parameter tuning, and benchmark the outcomes against the results presented above.

Architecture 1: $\mathbf{\Phi(x) = \exp(x) + 1}$

In the most simply setting, to ensure that all outputs of the NNs used to approximate α and β are greater than 1, we should just push the outputs of the second layer (component-wise) through the exponential function (to map $\mathbb{R}$ to $\mathbb{R}^+$), and then shift upwards by one unit.

Empirically, the algorithm was found to perform well when the learning rate is relatively small (around 10^{-4}). In this setting, the results achieved are:

	Seed 13	Seed 21	Seed 45	Seed 66	Seed 100	mean $\pm$ sd
`Cheetah`	739.1	338.2	486.9	281.4	820	533.1 ± 213.65
`Hopper`	378.6	371.9	423.2	329.7	391.6	379 ± 30.3
`Walker`	382.7	360.6	382.8	356.2	414.4	379.3 ± 20.7

Architecture 2: $\mathbf{\Phi(x) = softplus(x) + 1}$

The softplus function can be viewed as a smooth approximation to the ReLU function. For a scalar, $x \in \mathbb{R}$, the softplus function is:

$$\text{softplus}(x) := \ln(1 + e^x)$$

This is an alternative architecture to $e^x + 1$, that is numerically more stable due to the logarithm involved. Again, we will need to add a vector of ones to the ouput in order to impose that $\alpha, \beta > 1$. Again, the algorithm performed well for small learning rates (around 10^{-4}). The results were:

	Seed 13	Seed 21	Seed 45	Seed 66	Seed 100	mean $\pm$ sd
`Cheetah`	500.1	370.5	254.7	236.8	597.4	391.9 ± 139.5
`Hopper`	489.6	352	680.7	330.9	673.7	505.4 ± 150.5
`Walker`	375.7	377.9	389.1	397.7	346.1	377.3 ± 17.5

Architecture 3: $\mathbf{\Phi(x) = c \times Sigmoid(x) + 1}$, for some $c \in \mathbb{R}^+$

The motivation behind this architecture is to impose to artificial upper bound on the values that α and β can take, which I've called c. First, the Sigmoid function 'squishes' the vector into $(0, 1)$. Then, multiplying through by c ensures that both $\alpha, \beta \in (0, c)$. Empirically the algorithm performs 'best' when $c \approx 5$, which is what I've implemented in the code. Furthermore, the learning rate that yielded optimal results was $\approx 10^{-4}$. The results of this are:

	Seed 13	Seed 21	Seed 45	Seed 66	Seed 100	mean $\pm$ sd
`Cheetah`	682.4	867.3	824.6	996.2	870.8	848.3 ± 100.8
`Hopper`	346.6	347.5	372.1	367.8	347.1	356.2 ± 11.3
`Walker`	379.7	344.9	397.9	279.4	419.6	364.4 ± 49

Analysis and Conclusion

A summary of the episodic rewards across all architectures and environment is:

	`Cheetah`	`Hopper`	`Walker`
`Cleanrl`	32.5 ± 139.4	765.6 ± 237.5	365.7 ± 13.6
Arch. 1	533.1 ± 213.65	379 ± 30.3	379.3 ± 20.7
Arch. 2	391.9 ± 139.5	505.5 ± 150.5	377.3 ± 17.5
Arch. 3	848.3 ± 100.8	356.2 ± 11.3	364.4 ± 49

Overall, utilising the beta distribution as a prior for the policy is promising. Specifically, architectures 2 and 3 show promising results in the way that they outperform the `Clean-rl` implementation in 2 of the 3 tested environments. Furthermore, the algorithms generalise well across different environments, and have no determinants performances (Like the `Clean-rl` implementation on the `Cheetah` environment).

As expected, architectures 1 and 2 perform similarly due to the fact that the activation functions used are similar to one another. There does not seem to be any clear benefit of capping the α and β values (as is done in architecture 3).

Could you please give me some pointers on where to proceed in the next week? I've seen that the squashed Gaussian function (as is used in the SAC algorithm) is a promising alternative for bounded continuous action spaces, but I wonder if this might be repetitive of what I've already done. Are there any other current modification to PPO-Clip that I could investigate?

Thanks Nan!