Lunar Lander: A Continous-Action Case Study for Policy-Gradient Actor-Critic Algorithms

Roshan Shariff
Department of Computing Science
University of Alberta
Edmonton, AB, Canada, T6G 2E8
`roshan.shariff@ualberta.ca`

Travis Dick
Department of Computing Science
University of Alberta
Edmonton, AB, Canada, T6G 2E8
`tdick@ualberta.ca`

Abstract

We empirically investigate modifications and implementation techniques required to apply a policy-gradient actor-critic algorithm to reinforcement learning problems with continuous state and action spaces. As a test-bed, we introduce a new simulated task, which involves landing a lunar module in a simplified two-dimensional world. The empirical results demonstrate the importance of efficiently implementing eligibility traces and appropriately weighting the features produced by a tile coder.

Keywords: policy gradient, actor-critic, continuous actions, simulated environment

1 Introduction

Reinforcement learning has traditionally been applied to situations in which an agent must choose from a small number of discrete actions. The real world, however, usually offers a continuous spectrum of actions to animals or robots. The policy gradient approach can be applied to domains with continuous actions. We investigate the practical challenges involved in applying a recent policy-gradient actor-critic algorithm to a problem with continuous states and actions.

We introduce the task of landing a lunar module in a simplified two-dimensional simulation. The lander has a six-dimensional state space and two dimensions of control. We designed the task to be challenging and yet easy to simulate, and thus serve as a useful test-bed for experimenting with continuous states and actions.

This article is structured as follows: §2 is a detailed description of the lunar lander task. §3 describes our modifications and implementation of the policy gradient actor-critic algorithm introduced by Degris, Pilarski, and Sutton [1]. §4 presents preliminary empirical evaluations of several variations of the algorithm.

2 The Problem

We created a two-dimensional rigid-body simulator to model the descent of the Apollo lunar lander. The state of the simulator is completely described by six scalars: x and y indicate the position of the centre of mass of the lander, v_x and v_y its velocity, θ the attitude of the lander (its rotation from level flight), and ω its rotational velocity (i.e. the rate of change of θ). The controls available are the main descent engine throttle and the attitude control rockets. The descent engine thrusts the lander upwards, and can be smoothly throttled up to its maximum thrust. The attitude control system applies a torque to the lander (i.e. controls the rate of change of ω) but does not apply any translational force. It can be smoothly varied between torquing the lander clockwise at one extreme and counterclockwise at the other.

The simulation runs at a frequency of 20 Hz, whereas the agent runs at 2 Hz. Since the problem is laterally symmetric, it can be assumed without loss of generality that the x position is positive. If it is not, the world is mirrored, negating v_x, θ, and ω. The attitude control system input is then similarly negated before being applied to the simulator. This optimisation halves the size of the state space that the agent must learn to act in.

The impact of the landing gear with the lunar surface is modelled using the rigid-body collision algorithm of Guendelman, Bridson, and Fedkiw [2]. The impact forces generated on landing are applied to a simplified model of the strength of the lander legs to determine if the lander had crashed. The model was designed to roughly agree with the designed maximum landing velocity of the Apollo lunar module. It generates a scalar quantity called the impact stress that indicates how close the landing gear came to failing; when it exceeds 1.0 the module is considered to have crashed.

The parameters for the simulation were taken from a report on the mission simulator for the Apollo lander [3] (§6.4.3.1, "Ascent-Descent Configuration"). The values used were those for 50% propellant loading of the descent propulsion system and 100% propellant loading of the ascent propulsion system. The lunar gravitational acceleration was set to the standard value of $1.622\,\mathrm{m/s^2}$.

2.1 Reward

The reward function is intended to train the agent to land the lunar module as close as possible to a target location, and as softly as possible, while minimising use of the descent engine. When the module's legs are stationary on the ground, it is considered to have landed and is given a reward of $+1.0$. An episode ends when the module lands, crashes, or flies further than 100.0 m from the target. When the episode ends, the agent is additionally given a penalty based on its distance from the target location, which is fixed on the ground at $x = 0$. The penalty is $|x/w_{\mathrm{lander}}|$ where $w_{\mathrm{lander}} = 9.07\,\mathrm{m}$ is the width of the lander.

After every time step, the agent is given a penalty of 0.01 multiplied by the main engine throttle setting (which ranges from 0 to 1). If the legs contacted the ground during the time step, the impact stress is greater than zero, and $\log_{10}(1 + \mathit{impact})$ is added as a penalty. Finally, if an episode lasts more than 10 minutes, it is terminated with a reward of -1.

Each episode is started with the lander's position chosen uniformly at random, with the y coordinate at most 20 m and the x coordinate ranging between $-30\,\mathrm{m}$ and $30\,\mathrm{m}$. The attitude of the lander is chosen from a normal distribution with a mean of 0 and a standard deviation of $\pi/8\,\mathrm{rad}$. The translational and rotational velocities are initialized to zero.

3 Policy-Gradient Actor-Critic Algorithm

Since our problem has a continuous action space, we use (with modifications) the policy gradient algorithm introduced by Degris, Pilarsky, and Sutton [1]. We use an actor-critic architecture, with the critic using TD(λ) to compute the TD error δ, and two independent policy gradient actors, one for each of the two dimensions of the action space. The policy gradient actors use a scaled value of α, which is denoted algorithm AC-S in the above paper.

3.1 Policy Gradient Actor

Since our action space is bounded, we use the truncated normal distribution instead of the usual normal distribution to propose actions. Further, instead of applying the exp transfer function to the σ parameter, we apply a scaled and shifted logistic transfer function to avoid numerical issues.

We adapt the policy gradient algorithm to action spaces bounded by $a_{\min} \le a \le a_{\max}$. Since the usual normal distribution has unbounded support, the straightforward approach is to clamp the proposed action outside the agent; the environment treats actions outside the range $[a_{\min}, a_{\max}]$ as lying at the closer of the two endpoints. However, when a large part of the action distribution lies outside the interval, the endpoints will be selected with unduly high probability. Even worse, exploration suffers since only the tail of the distribution lies in the allowed range.

Instead of clamping the action outside the agent, we modify the policy gradient algorithm to use the truncated normal distribution. This distribution is obtained by truncating the usual normal distribution to a bounded range and then renormalizing it. We define

$$\tilde{a} = \frac{a-\mu}{\sigma}, \quad \tilde{a}_{\min} = \frac{a_{\min}-\mu}{\sigma}, \quad \tilde{a}_{\max} = \frac{a_{\max}-\mu}{\sigma},$$

i.e. the standardised forms of the action a and the minimum and maximum actions $a_{\min}$ and $a_{\max}$. Then the density function of the truncated normal distribution can be expressed as

$$f(a;\mu,\sigma) = \frac{1}{\sigma}\frac{\phi(\tilde{a})}{\Phi(\tilde{a}_{\max}) - \Phi(\tilde{a}_{\min})}$$

where $\phi(a)$ and $\Phi(a)$ are respectively the p.d.f. and the c.d.f. of the standard normal distribution.

It is clear that for $\sigma \gg a_{\max} - a_{\min}$, the truncated normal distribution closely approximates the uniform distribution. Thus, it is natural to set an upper bound on the value of σ. We would also like to force the algorithm to maintain a certain minimum level of exploration. Thus, instead of using the exp transfer function (as in [1]) which approaches zero on one side and $+\infty$ on the other, we use the logistic transfer function scaled and shifted to range from $\sigma_{\min}$ to $\sigma_{\max}$, which we call $\sigma(s)$. The original formulation uses $\exp(\mathbf{u}_\sigma \cdot \mathbf{x}_\sigma(s))$, which easily overflows a floating point representation.

Taking into account these two modifications, the vector of compatible features $\frac{\nabla_{\mathbf{u}}\pi(a|s)}{\pi(a|s)}$ for the μ and σ parameters are

$$\frac{\nabla_{\mathbf{u}_\mu}\pi(a \mid s)}{\pi(a \mid s)} = \frac{\mathbf{x}_\mu(s)}{\sigma}\left(\tilde{a} + \frac{\phi(\tilde{a}_{\max}) - \phi(\tilde{a}_{\min})}{\Phi(\tilde{a}_{\max}) - \Phi(\tilde{a}_{\min})}\right)$$

$$\frac{\nabla_{\mathbf{u}_\sigma}\pi(a \mid s)}{\pi(a \mid s)} = \frac{\mathbf{x}_\sigma(s)}{\sigma}\,(\sigma(s) - \sigma_{\min})\left(1 - \frac{\sigma(s)-\sigma_{\min}}{\sigma_{\max}-\sigma_{\min}}\right)\left(\tilde{a}^2 - 1 + \frac{\tilde{a}_{\max}\cdot\phi(\tilde{a}_{\max}) - \tilde{a}_{\min}\cdot\phi(\tilde{a}_{\min})}{\Phi(\tilde{a}_{\max}) - \Phi(\tilde{a}_{\min})}\right)$$

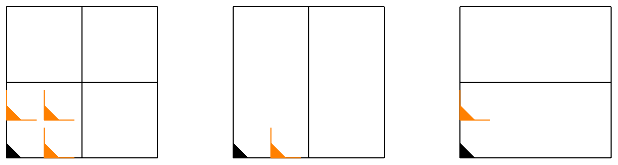

Figure 1: Tilings of a two-dimensional space.

We note that as $a_{\min} \to -\infty$ and $a_{\max} \to \infty$, the truncated normal distribution and its gradients match the untruncated normal distribution.

Finally, we use the AC-S variant of the policy gradient algorithm [1], which uses a step size α that is scaled by the variance of the truncated normal distribution. It is important to note that μ and σ are no longer the mean and the standard deviation of this distribution; they are simply parameters that influence the shape of the distribution. In fact, the variance of the distribution is given by

$$\sigma^2 \left[1 - \frac{\tilde{a}_{\max} \cdot \phi(\tilde{a}_{\max}) - \tilde{a}_{\min} \cdot \phi(\tilde{a}_{\min})}{\Phi(\tilde{a}_{\max}) - \Phi(\tilde{a}_{\min})} - \left(\frac{\phi(\tilde{a}_{\max}) - \phi(\tilde{a}_{\min})}{\Phi(\tilde{a}_{\max}) - \Phi(\tilde{a}_{\min})} \right)^2 \right].$$

3.2 Tile Coding

The continuous state space is turned into a feature representation using a tile coder with rectangular grids. We choose subsets of the state variables and produce separate families of tilings for each subset. Each state variable is associated with a tile size, the number of tiles, and the number of offsets. For each subset of variables there is a family of tilings, each having a number of tiles that is the product of the number of tiles for each included variable. Further, each family contains a number of tilings which is the product of the number of offsets for each included variable. The tilings in each family are offset by different amounts. The offsets are evenly spaced and densely cover the set of state variables being tiled.

For example, figure 1 shows tilings of a simple two-dimensional state space. Each of the state variables, x and y, is associated with two tiles and two offsets. The leftmost diagram shows the family of tilings that covers both state variables. Each tiling is a 2×2 grid. The black triangle marks the primary tiling, which is not offset in either dimension. The three orange triangles show the offsets of the other three tilings in the family. The second and third diagrams show tilings covering just the x and just the y dimensions, respectively. The tilings in these families are 2×1 and 1×2 grids, and each family has the primary tiling as well as one other offset version. These three families together comprise 8 tilings, and between them generate a total of 24 features (i.e. tiles).

State variables that naturally wrap around are represented without arbitrary boundaries that prevent generalisation. Other state variables are clamped to the rectangular region covered by the tiling, so that the tiles on the edges activate when the state variable is outside the represented range. Finally, the feature indices generated by the tile coder are hashed into a smaller set to reduce memory requirements.

One problem with this tile coding scheme is that the subsets of state variables have a highly varying number of tilings. For example, in figure 1, the two-dimensional subset has four tilings, whereas each of the one-dimensional subsets has two tilings. The difference is greatly magnified when more state variables are present. When the generated features are used in an agent, the higher-dimensional state subspaces are vastly overrepresented, and the lower dimensional ones are all but ignored. Thus the advantages to generalisation from the lower-dimensional tilings are almost entirely negated.

To overcome this problem, the tile coder weights each tiling differently. The tilings in each family are weighted by $1/\sqrt{n}$, where n is the number of tilings in the family. In the diagram above, the four tilings of the two-dimensional space would be weighted $1/\sqrt{4}$, whereas the two tilings in each one-dimensional space would each be weighted $1/\sqrt{2}$. The feature vector $\mathbf{x}(s)$ is thus not a binary vector, but has differing weights for each activated feature.

With binary feature vectors, the step size α used for parameter updates is divided by the number of active features. This preserves the meaning of $\alpha = 1$ as eliminating the observed TD error. The appropriate generalisation to weighted feature vectors is to divide α by the squared Euclidean norm of the feature vector. For binary feature vectors, the squared Euclidean norm is indeed the number of active features. To avoid having to scale α, the tile coder normalises the feature vectors, so that their squared norm is 1.

3.3 Eligibility Trace

The run-time performance of the algorithm depends on the feature vector being highly sparse. The traditional accumulating eligibility trace would grow less sparse with time, destroying the incremental nature of the algorithm. To avoid this, we maintain the eligibility trace as a bounded queue of feature vectors. During the update step, each of the feature vectors in the queue is used in turn, scaled by the usual λ^n falloff. Once the weight associated with the feature vector falls below a predetermined threshold, it is removed from the queue. Thus the maximum length of the queue is $\lceil \log_\lambda(\text{threshold}) \rceil$. This gives an incremental algorithm whose updates are numerically close to the standard λ-trace.

3.4 Parallel Training

To take advantage of multi-core processors, we train the algorithm in parallel, using shared memory for the parameter vectors. This allows us to conduct more training episodes in a given time, but it means that updates to the parameter vector from different, independent episodes are interleaved. We observed that the running time decreased linearly with the number of cores simultaneously executing episodes. A modern dual core system took approximately 10 minutes for each run of 20,000 episodes.

4 Experiments

All the experiments were conducted with the same tile coder settings, given in table 1. The tile coder used all subsets of the state variables containing 1, 2, and 6 states. This resulted in 2300 active features, with 162,580,216 $\approx 2^{27.3}$ features in total. These were hashed down to 2^{20} features.

State	No. of tiles	Tile size	No. of offsets
x	6	5 m	2
y	4	5 m	2
v_x	8	2 m/s	4
v_y	8	2 m/s	4
θ	4	$\pi/2$ rad	8
ω	6	$\pi/6$ rad/s	4

Table 1: Tile coder settings

The parameter vectors for the critic and the actors were initialised to default values. This was done by setting each entry in the vector to $v_0 / \|\mathbf{x}(s)\|_1$, where v_0 is the desired initial value and $\|\mathbf{x}(s)\|_1$ is the sum of the feature weights (which does not depend on s). This gives $\mathbf{x}(s) \cdot \theta = v_0$ for any s.

The value function used by the TD(λ) was initialised optimistically to 1.0, which is the maximum attainable reward. For the main engine thrust actor, the initial μ parameter was set to $\mu_0 = 0.5\,\text{m/s}^2$ and the initial σ parameter was set to $\sigma_0 = \text{thrust}_{\text{max}}/10$. For the attitude control, the initial μ parameter was set to $\mu_0 = 0$ and the initial σ value was set to $\sigma_0 = \text{rcs}_{\text{max}}/6$.

The sigmoidal transfer function applied to the σ parameter in the policy gradient actor is parameterised by maximum and minimum values, σ_{min} and σ_{max}. For the thrust actor, we set $\sigma_{\text{min}} = \text{thrust}_{\text{max}}/64$ and $\sigma_{\text{max}} = \text{thrust}_{\text{max}}/2$. For the attitude control actor, we set $\sigma_{\text{min}} = \text{rcs}_{\text{max}}/32$ and $\sigma_{\text{max}} = \text{rcs}_{\text{max}}$. Thus, since the thrust control ranges from 0 to $\text{thrust}_{\text{max}}$ and the attitude control ranges from $-\text{rcs}_{\text{max}}$ to $+\text{rcs}_{\text{max}}$, the maximum value of σ is equal to half of the valid action range.

The graphs below show the cumulative return as it changes over the first 20,000 episodes, averaged over n independent runs (the value of n is given for each experiment). The slope of the curve at any point indicates the average return per episode, and an increase in the slope indicates that the agent is improving its performance. The shaded orange regions indicate the 95% confidence interval of the estimate of the mean (i.e. $\pm$ twice the standard error in the mean).

All experiments were conducted with $\alpha = 0.1$ for both the critic and the actors, and with $\lambda = 0.75$ unless otherwise indicated.

4.1 Untruncated vs. Truncated Normal Distributions

We compared the use of the truncated and untruncated normal distributions to propose actions. In the latter case, actions beyond the allowable range are clamped by the simulator, outside the agent. The graph (figure 2) shows the mean cumulative sum over $n = 30$ independent repetitions of the experiment.

The untruncated normal distribution has a higher cumulative return, and higher average return per episode towards the end. To our surprise, the untruncated normal distribution performs better on this experiment. However, we used the same initial parameter values σ_0 and μ_0 for both variations of the experiment. Unlike for the untruncated normal distribution, μ and σ are not the mean and standard deviation of the truncated normal distribution. It is possible that optimising the initial parameters could change the relative performance of the two algorithms.

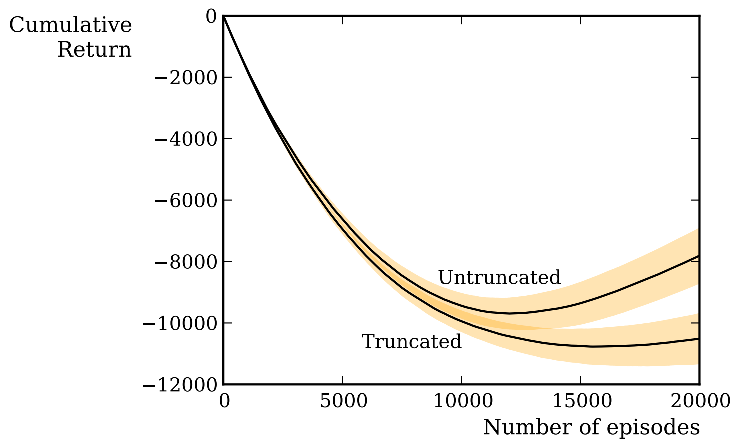

Figure 2: Untruncated vs. truncated normal distributions (section 4.1).

4.2 Feature Weighting and Tile Coding Subspaces

We compared the use of weighted features (as described in section 3.2) to unweighted features. We also evaluated the use of only the complete 6-dimensional state space (in which case weighting makes no difference, since there is only one family of tilings), instead of 1-, 2-, and 6-dimensional subsets of state variables. The results are shown in figure 3, where each cumulative return curve is averaged over $n = 30$ independent runs.

Adding tilings that span lower-dimensional subsets of features significantly improved performance and learning rate. Weighting the different families of tilings produced another significant improvement in performance.

We conclude that for this problem the increase in generalisation allowed by the addition of coarse tilings significantly benefits the algorithm. The vast discrepancies in the number of tilings produced for each subset of state variables negates some of the benefits of the coarse tilings, but is solved by appropriately weighting each tiling.

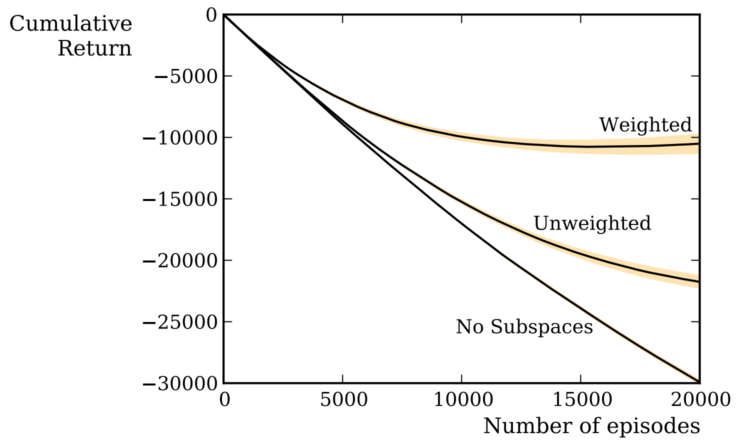

Figure 3: Feature weighting and tile coding subspaces (section 4.2).

4.3 λ Parameter Study

To judge the effect of the eligibility trace on learning performance, we compared different values of λ (figure 4). As in the other experiments, α was fixed at 0.1. The $\lambda = 0$ cumulative reward curve was obtained by averaging $n = 30$ runs, whereas the others average $n = 57$ independent runs.

Setting $\lambda = 0.75$ gives the best cumulative return, closely followed by $\lambda = 0.5$ and $\lambda = 0.9$. Disabling eligibility traces (by setting $\lambda = 0$) drastically decreases performance. The agent with $\lambda = 0.5$ seems to have learned slightly better then the one with $\lambda = 0.75$ near the end of 20,000 episodes, but it is unclear to us whether this effect is statistically significant.

We note that this parameter study does not vary α, and we do not make claims about the absolute performance of different values of λ. However, subject to the constraint that $\alpha = 0.1$, we conclude that that this problem benefits greatly from the use of an eligibility trace, and the value of $\lambda = 0.75$ lies in the optimal region between $\lambda = 0.5$ and $\lambda = 0.9$.

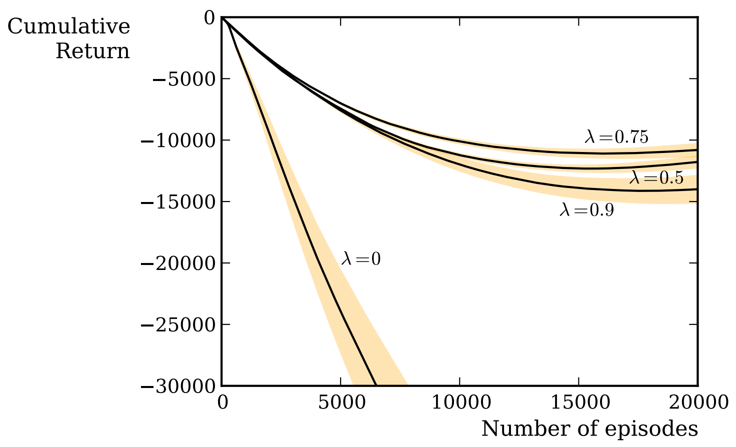

Figure 4: λ parameter study (section 4.3).

5 Conclusions

We introduced the lunar lander task and discussed the implementation of an actor-critic policy-gradient agent for it. Various modifications were necessary for the algorithm to learn how to land the lunar module. We therefore consider this task a useful test-bed for investigating continuous action reinforcement learning algorithms.

Finally, we presented empirical evidence evaluating the impact of the algorithm modifications. The large improvement in performance we achieved with the same underlying algorithm suggests that eligibility traces and the details of tile coding have a significant impact on real-world performance.

References

[1] Thomas Degris, Patrick M. Pilarski, and Richard S. Sutton. Model-Free Reinforcement Learning with Continuous Action in Practice. In *American Control Conference (ACC)*, pages 2177–2182, Montreal, QC, Canada, June 2012.

[2] Eran Guendelman, Robert Bridson, and Ronald Fedkiw. Nonconvex Rigid Bodies with Stacking. *ACM Transactions on Graphics*, 2003. URL: `http://dl.acm.org/citation.cfm?id=882358`.

[3] MIT Instrumentation Library. Apollo Guidance, Navigation and Control: Guidance System Operations Plan for Manned LM Earth Orbital and Lunar Missions using Program Luminary. Section 6: Control Data, November 1968. URL: `http://www.ibiblio.org/apollo/NARA-SW/R-567-sec6.pdf`.