Baltic J. Modern Computing, Vol. 6 (2018), No. 4, pp. 335–348
https://doi.org/10.22364/bjmc.2018.6.4.02

Evaluating the Effectiveness of Deep Reinforcement Learning Algorithms in a Walking Environment

Arjun NEERVANNAN

University High School, 4771 Campus Drive, Irvine, CA 92612

`arjun.neervannan@gmail.com`

Abstract. Deep Reinforcement Learning algorithms have shown to perform well on complex tasks, such as video games and chess. However, when it comes to locomotive tasks, picking the right algorithm and hyperparameters continues to be a challenge for many researchers. This project addressed that issue by determining which one of three reinforcement learning algorithms worked most effectively to help a computer learn to walk, without any external supervision or guidance, in a simulated environment. In addition, the project also determined the best learning rate for the algorithms by testing out 6 learning rates. A walking environment was used as it is considered to be a good representative for a large class of reinforcement learning problems. Proximal policy optimization was found to be the most effective, followed by the trust-region policy optimization and the vanilla policy gradient. The algorithms worked best with learning rate 1e-3.

Keywords: Proximal Policy Optimization; Deep Neural Networks; Optimization

1 Introduction

Reinforcement algorithms, especially the ones that combine neural networks, called Deep Reinforcement Algorithms, have shown powerful learning abilities with the demonstration of AlphaGo and similar complex tasks Mnih et al.,2015, Sutton and Barto,2015, Silver et al.,2017. However, choosing the right algorithm and set of hyperparameters for locomotive tasks is a difficult task Peng et al.,2016. A walking environment was used as it is considered to be a good representative for a large class of reinforcement learning problems.

1.1 Basic Terminology

Reinforcement learning is a class of machine learning algorithms used to help computers learn to make decisions in an environment with or without any external guidance during the learning process Sutton and Barto,2015. The agent learns to make decisions solely off the states of the environment, the rewards that the environment returns to the agent, and the actions that the agent takes. The goal of reinforcement learning is to maximize a numerical reward by learning what to do and mapping situations to actions Lu,2017. In the absence of existing training data, the agent learns from experience, updating its weights to maximize the reward.

An artificial neural network (ANN) is a computational method inspired by neurological systems to represent complex functions Schmidhuber,2014. ANNs can be represented by neurons and axons, which form a large net. Neurons are organized in layers, and the organization of neurons in a layer can change the purpose of that layer. In this project, only fully-connected (FC) layers, in which all neurons in subsequent layers are connected to each other, were used.

ANNs update their weights to improve the accuracy of the function using a method known as gradient descent or ascent (depending on the type of problem), which computes the delta between the expected and actual value, and updates each weight in the network by propagating backwards through the network Ruder,2017. The ”value” differs depending on the case, but in this case it refers to a reward value, given by the environment. Gradient descent also uses a hyperparameter called the learning rate (LR) that controls the speed of the convergence Goodfellow et al., 2017. As neural networks represent extremely complex functions, gradient descent has to take small steps to reach the optimal set of weights to prevent itself from overshooting the local optima Goodfellow et al., 2017, Baird Moore, 1999. Off-the-shelf implementations of gradient descent, such as Adam (**ADA**ptive **M**oment estimation), which adaptively adjusts the learning rate as the gradient descent algorithm nears convergence, are easier to use in applications Kingma D. and Ba, J., 2015. As the goal of the algorithm was to maximize the reward, gradient ascent was used rather than gradient descent . However, the foundational methodology between gradient descent and ascent remains constant.

1.2 Vanilla Policy Gradient

The general goal of policy gradient methods is to create a policy, or strategy on which the agent can rely to make decisions in a virtual environment, that maximizes the possible reward Sutton and Barto,2015, Williams. Policy gradient methods differ from other reinforcement learning methods such as value-iteration update functions and actor-critic methods as they directly optimize the policy rather than optimizing the reward for each action and state (as in the case of Q-learning) Li et al.,2017. As a result, they tend to have better convergence rates and can work on environments with infinite action spaces (infinite available actions at each step), such as the one used in this experiment; however, they also are computationally intensive and often have high variances. Nevertheless, policy gradients have become the state-of-the-art algorithms to use in locomotion tasks Silver et al.,2017.

More specifically, the underlying principle behind the vanilla policy gradient (VPG) method is to maximize the expected future discounted reward in the environment by performing gradient descent on the policy directly to reach the optimal weights Sutton and Barto,2015. Policy gradients differ from other methods in this way in that they do not receive the reward at each timestep; rather, they use the total reward at the end of the episode to optimize the policy. Although this method is seemingly inefficient as certain actions taken in an episode may have contributed to the reward more than others, in the end, this disparity has little effect as the policy, through exploration of actions, eventually learns which actions give a better reward.

The policies that are iterated through are formally defined by $\Pi = \{\pi_\theta, \theta \in \mathbb{R}^m\}$, which represents a set of policies Π that contain policies π_θ parametrized by weights θ. Since ANNs are typically used to represent policies, each policy π_θ can be thought of as a neural network parametrized by weights θ that outputs an action at each state. The equation below shows the value for each policy, and the goal of the VPG method is to create a policy that maximizes this value. t represents the timestep, γ represents the discount factor, and $r(t)$ represents the reward given at each step Li et al.,2017. A discount factor is used to lower the weight that the algorithm gives to future rewards (in future states).

$$J(\theta) = \mathbb{E}\left[\sum_{t=0} \gamma^t r(t) | \pi_\theta\right]$$

The goal of the algorithm is to reach a set of parameters $\theta*$ such that $\theta* = argmax J(\theta)$ by performing gradient ascent on the policy directly to reach the optimal weights Sutton and Barto,2015.

The basic process that VPGs follow can be represented as pseudocode, shown below:

```
Initialize Parameter $\theta$
For iteration 1, 2, 3  do:
    – Using policy $\pi_\theta$, interact with the environment by taking the actions (output from the policy) until the episode ends
    – At the end of the episode, obtain the total reward for that episode from the environment
    – Update the policy $\pi_\theta$ by using gradient ascent on parameters $\theta$
End for
```

The gradient estimator used to update the weights is

$$\nabla_\theta J(\theta) \approx \sum_{t>1} r(\tau) \nabla_\theta \log \pi_\theta(a_t | s_t)$$

where θ represents the weights that parametrize policy π_θ, t represents the timestep, a_t and s_t represent the action and state taken at timestep t, respectively, and $r(\tau)$ represents the total reward at the end of the episode (with τ as a trajectory of states and actions from one episode). Again, the policy gradient is only given the cumulative reward at the end of the episode, rather than individual rewards per timestep. The purpose of this algorithm is to update the weights θ by increasing or decreasing the probability of actions (represented by $\pi_\theta(a_t|s_t)$). Given a high episodic reward, the algorithm assumes that **all** the actions taken in that episode were good actions and pushes up the probabilities of all the actions, and vice versa given a low episodic reward Li et al.,2017. Again, while this may seem simplistic, this method does work as the policy, by trying out different actions, eventually learns which actions are good and which are not.

However, determining which rewards are better than others is also a challenge, as the range of the rewards between a bad episode and a good episode can be very small, and consequently, the updates will also be small, even though the actions taken in the good episode should receive a higher weight. To solve this problem, a baseline function is used to compare the rewards to determine which ones are better and which ones are worse Li et al.,2017.

1.3 Trust-Region Policy Optimization

The trust-region policy optimization algorithm (TRPO) builds off of the VPG algorithm by using Kullback-Leibler (KL) Divergence to constrain each optimization step to a ”trusted region” around the original policy Schulman et al.,2015. This constrained optimization step ”guarantees a monotonic improvement” to the policy, essentially making the ascent to convergence more controlled Schulman et al.,2015.

Below is the derivation for the TRPO algorithm Lu,2017.

An MDP is defined as a tuple $(S, A, P_{sa}, \gamma, R, p_0)$, where:

- S is a finite set of N states
- A is a set of k actions, $a_1, a_2, ...a_k$
- $P_{sa}(s')$ represents the probability of landing at state s' upon taking action a at state s
- $\gamma \epsilon [0, 1)$ is the discount factor
- $r : S \rightarrow \mathbb{R}$ is the reward function (defined by the environment)
- $p_0 : S \rightarrow \mathbb{R}$ is the initial state distribution
- $p_\pi : S \rightarrow \mathbb{R}$ is the discounted visitation frequencies (the discounted probability of landing at each state)

$$p_\pi(s) = P[s_0 = s] + \gamma P[s_1 = s] + \gamma^2 P[s_2 = s] + ...$$

$$n(\pi) = \mathbb{E}_{s_0, a_0, ...}\left[\sum_{t=0} \gamma^t r(s_t)\right]$$

The above equation represents the expected discounted cumulative reward of policy π, where $s_0 \sim p_0(s_0)$, $a_t \sim \pi(a_t|s_t)$, $s_{t+1} \sim P(s_{t+1}|a_{t+1})$.

$$Q_\pi(s_t, a_t) = \mathbb{E}_{s_{t+1},a_{t+1},\ldots}\left[\sum_{l=0}^{\infty} \gamma^l r(s_{t+1})\right]$$

The above equation is the action-value function, which expresses the expected value of taking action a_t at state s_t and then following the policy π afterwards.

$$V_\pi(s_t) = \mathbb{E}_{a_t,s_{t+1},,\ldots}\left[\sum_{l=0}^{\infty} \gamma^l r(s_{t+1})\right]$$

The above equation is the value function, which expresses the expected value of following the policy π from state s_t onwards.

$$A_\pi(s,a) = Q_\pi(s,a) - V_\pi(s)$$

The above equation is the advantage function, which expresses the "advantage" of taking action a_t over following the policy π at state s_t. As with the VPG, the TRPO does not have the advantage value for each timestep, rather having a cumulative advantage from the whole episode.

The below identity was proved by Kakade and Langford 2002 Kakade Langford,2002.

$$n(\pi) = n(\pi_0) + \mathbb{E}_{p_\pi}\mathbb{E}_{a\sim\pi(s)}[A_{\pi_0}(s,a)]$$

where π is the new policy and π_0 is the old policy. However, the gradient estimator does not have π yet; therefore, p_π does not exist. The TRPO algorithm instead uses p_{π_0} as an approximation of p_π. Hence, the objective function becomes

$$L_{\pi_0} = n(\pi_0) + \mathbb{E}_{p_{\pi_0}}\mathbb{E}_{a\sim\pi(s)}[A_{\pi_0}(s,a)]$$

Schulman et al 2015 then used KL divergence to create a surrogate objective by penalizing the Loss function Schulman et al.,2015. KL divergence, which calculates the distance between two probability distributions, calculates the distance between the old policy and the new policy and penalizes the loss function by that value.

$$n(\pi) \geq L_{\pi_0}(\pi) - Cmax_s D_{KL}(\pi_0(s))||\pi(s)$$

where $\epsilon = max_s \left|\mathbb{E}_{a\sim\pi'(s)[A_{\pi_0}](s,a)}\right|$ and $C = \frac{4\epsilon\gamma}{(1-\gamma)^2}$.

However, calculating the KL divergence term over the whole state space is intractable Kakade Langford,2002. Instead, Schulman et al. used mean KL divergence over the whole state space as an approximation Schulman et al.,2015.

$$\overline{D}_{KL}(\pi_0||\pi) = \mathbb{E}_{s\sim p_{\pi_0}}\left[D_{KL}(\pi_0(s))||\pi(s)\right]$$

Therefore, the base TRPO optimization problem is

$$maximize_\theta\left[L_{\theta_0}(\theta) - C\overline{D}_{KL}(\pi_0||\pi)\right]$$

However, in practice, TRPO does not use a penalty term or the penalty coefficient C as the step sizes would be very small. Instead, a hard limiter is used Schulman et al.,2015.

$$\begin{aligned} &maximize_\theta \qquad L_{\theta_0} \\ &subject\,to \qquad \overline{D}_{KL}(\pi_0||\pi) \leq \delta \end{aligned}$$

The algorithm can then be optimized using the conjugate gradient method Schulman et al.,2015. Because of this constrained optimization step, the TRPO algorithm provides a steady, consistent update to the policy. It is fairly computationally expensive, but further implementations of the base TRPO algorithm have been created to simplify the optimization steps Lu,2017. TRPO uses Natural Gradient Ascent to update the ANN as it is built directly into the TRPO algorithm; consequently, TRPO is not compatible with other optimizers, such as Adam. Natural Gradient Ascent uses KL divergence to constrain the optimization step of an ANN Grosse.

1.4 Proximal Policy Optimization

The Proximal Policy Optimization (PPO) algorithm builds on the base TRPO algorithm, rather using first-order optimization methods to simplify the computation Schulman et al.,2017. PPO does not use KL divergence, rather clipping the the ratio of the old and new policy to a certain range. It then takes the minimum across that clipped ratio and the original ratio, and finally multiplies that minimum by the Advantage estimate. Eliminating KL divergence from the surrogate objective function makes the PPO algorithm much simpler to implement.

The surrogate objective function for the PPO algorithm is

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t\big[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)\big]$$

given parameters θ, advantage function estimator $\hat{A}_t$, policy ratio $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ (probability ratio between new and old policy), and some hyperparameter ϵ used to clip the probability ratios Schulman et al.,2017. Clipping r_t between $[1-\epsilon, 1+\epsilon]$ prevents the ratio between the old policy and new policy from going too high, which ensures that improvement is fairly controlled and constant. Furthermore, Schulman et al.'s tests determined that the best value for ϵ was 0.2 Schulman et al.,2017.

Intuitively, the PPO uses a simpler calculation to create a lower bound on which the policy can optimize on, similar to an Majorize-Minimization algorithm Lange,2007. This can be thought of as a ”soft limit” as opposed to the hard limit of TRPO.

2 Experimental Design

The project tested the effectiveness of three different algorithms on a simulated walking environment, in which the computer had control over 6 joints and received 17 observations (17x1 input vector and 6x1 output vector) from each of those joints (see Figure 1). The goal was to move forward as far as possible, and rewards were based on the change in position as well as other metrics (see below for reward calculation). High rewards indicate a good performance. 6 learning rates, 0.1 (1e-1), 0.01 (1e-2), 0.001 (1e-3), 0.0003 (3e-4), 0.0001 (1e-4), and 0.00001 (1e-5), were tested on all of the agents to determine the best one. The computer ran each algorithm for 40,000 episodes, or for 16,000,000 time steps, since each episode was 400 timesteps long.

Other variables, including the network size and type, activation function, training episodes, and hardware used were kept constant between the algorithms. However, the Adam optimizer was used for the VPG and PPO algorithms but not for the TRPO algorithm. This was because TRPO already had a built-in optimizer technique that limited the search to a certain region, thus producing an identical effect as Adam (see 1.3). A two-layer ANN with tanh activations and 2 fully-connected layers with 32 nodes each was used.

The effectiveness of the algorithms was based on the following criteria:

- the 100-episode average reward after training was used to judge the performance of the algorithm for a learning rate. A higher reward meant that the agent performed better, and vice versa.
- The algorithm had to show consistent improvement across the episodes for the results to be considered. This ensured that the agent was not just randomly attaining a certain result.
- if an algorithm did not get the highest reward for that episode, it could still get the highest rank for that learning rate if it showed that it had a higher slope, or more ”momentum”. This was determined based on the ”learning curve”, or progression of rewards per episode over episodes, for each algorithm (all learning curves are in the appendix).

The program itself used TensorForce, a reinforcement learning library built on top of Tensorflow (a common machine learning library). OpenAI Gym environment (an open source platform for creating, evaluating and benchmarking agents in a game environment) was used to render after the completion of the learning phase Brockman, 2016. Matplotlib and Numpy were used to process the data and graph the results.

In addition, the final code modified the existing TensorForce examples code base, found on Github. However, the code used in this project bears very little semblance to the original due to the extensive modifications that were made to suit the goals of the experiment. Functions were added to help render and record the models every 1000 episodes as well as save the model every 100 episodes. This helped to evaluate and compare the performance of the models.

In the game environment itself, 17 state observations were given to the agent, and these included information about the position of the agent from the center, balance, and other metrics. There were 6 available actions at each state of the model, with one for each joint. Rewards were calculated based on the agents distance from the starting point (see below for reward calculation equation).

The reward calculation (given by the environment) is

$$R(\alpha, \beta, T, A) = -0.1 \sum_{n=1}^{T} a_n^2 + \frac{\alpha - \beta}{T}; A = \{a_1, a_2, ...\}$$

Where α is the starting position of the simulated walker, β is the ending position, A is the set of actions, and T is the total number of timesteps in that episode.

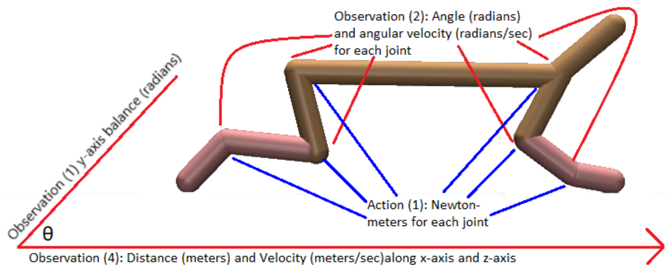
Observation (2): Angle (radians) and angular velocity (radians/sec) for each joint
Observation (1) y-axis balance (radians)
θ
Action (1): Newton-meters for each joint
Observation (4): Distance (meters) and Velocity (meters/sec)along x-axis and z-axis

Fig. 1: The diagram above shows the observations that the agent receives as well as the actions that it can take.

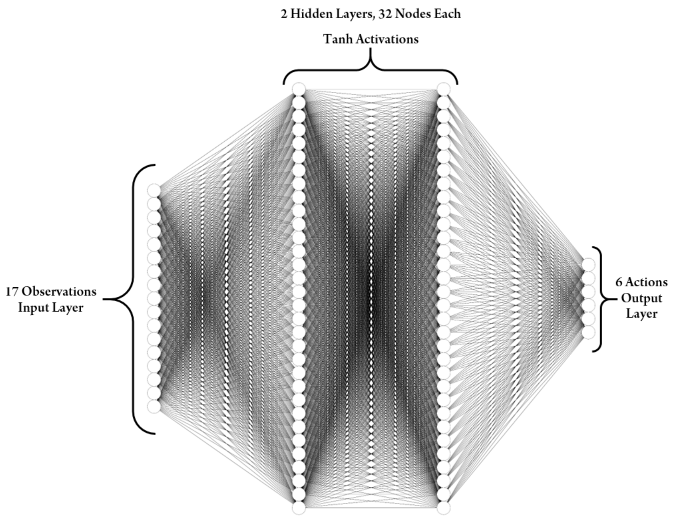

Fig. 2: The diagram above shows the neural network structure used in this project. It is a 2-layer, 32-node network with 17 inputs and 6 outputs.

3 Results

Table 1 shows all of the results from the experiment. For reasons detailed in the Analysis section (Section 4), the results from Learning Rates 1e-1, 1e-2, and 1e-5 were dropped.

Algorithm	1e-1	LR 1e-2	LR 1e-3	LR 3e-4	LR 1e-4	LR 1e-5
TRPO	470.515	595.417	773.893	669.667	709.202	410.042
PPO	34.448	369.436	1086.395	814.096	665.178	-1.633
VPG	246.349	1159.34	443.049	208.143	58.494	-82.808

Table 1: Average Reward for Algorithms and Learning Rates

PPO outperformed TRPO and VPG for LRs 1e-3 and 3e-4. TRPO outperformed VPG for all of the LRs except for LR 1e-2, where VPG got the highest results. Furthermore, TRPO tended to perform well across many learning rates, while PPO performed very well for a select range of learning rates and VPG performed poorly for most learning rates. **Table 2** shows the modified results in numerical form, while Figure 2 shows the results as a line graph.

Algorithm	LR 1e-3	LR 3e-4	LR 1e-4
TRPO	773.893	669.667	709.202
PPO	1086.395	814.096	665.178
VPG	443.049	208.143	58.494

Table 2: Average Reward for Algorithms and Learning Rates

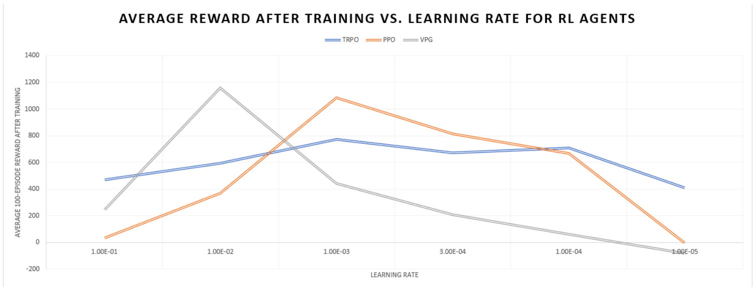

Fig. 3: The above graph shows the average 100-episode reward for the three different algorithms for all six learning rates.

The below six graphs show the learning curves for the algorithms for each learning rate.

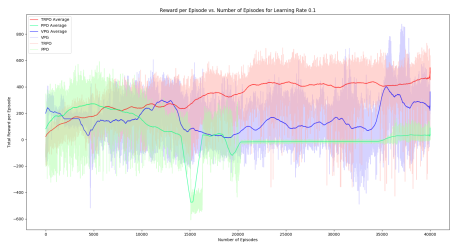

Fig. 4: The above graph shows the reward over episodes for learning rate 1e-1. Green = PPO, Red = TRPO, Blue = VPG.

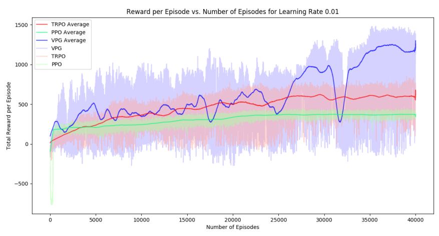

Fig. 5: The above graph shows the reward over episodes for learning rate 1e-2. Green = PPO, Red = TRPO, Blue = VPG.

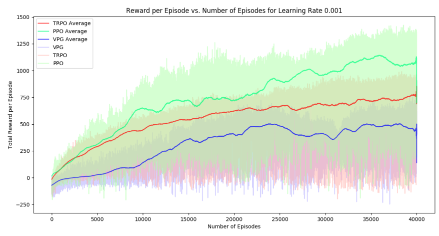

Fig. 6: The above graph shows the reward over episodes for learning rate 1e-3. Green = PPO, Red = TRPO, Blue = VPG.

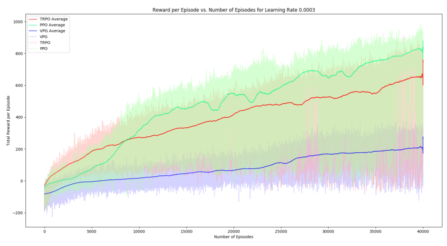

Fig. 7: The above graph shows the reward over episodes for learning rate 3e-4. Green = PPO, Red = TRPO, Blue = VPG.

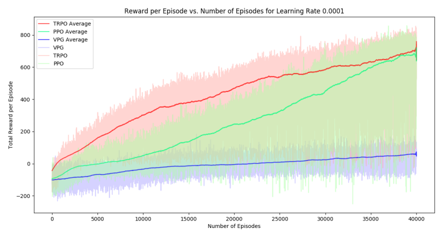

Fig. 8: The above graph shows the reward over episodes for learning rate 1e-4. Green = PPO, Red = TRPO, Blue = VPG.

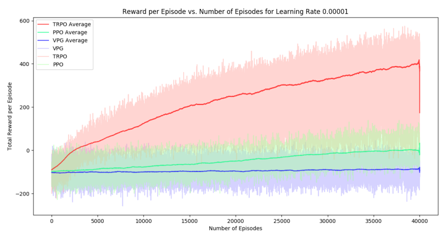

Fig. 9: The above graph shows the reward over episodes for learning rate 1e-5. Green = PPO, Red = TRPO, Blue = VPG.

4 Analysis and Discussion

The hypothesis was partially supported, as the Proximal Policy Optimization (PPO) algorithm outperformed the Trust-Region Policy Optimization (TRPO) for two out of the three learning rates examined, and consistently had a higher momentum in the learning curves. VPG performed significantly worse that TRPO and PPO for all three learning rates.

4.1 Learning Rate 1e-1

All three agents did not learn at all with learning rate 1e-1, and their rewards oscillated randomly (refer to Figure 3). Hence the results from this learning rate were disregarded.

4.2 Learning Rate 1e-2

With a learning rate of 1e-2, VPG appeared to attain a better result, but the results fluctuated randomly and did not show consistent improvement (refer to Figure 4). Both PPO and TRPO had flat learning curves showing no consistent improvement in the rewards. Hence the results from this learning rate were disregarded.

4.3 Learning Rate 1e-3

With the learning rate of 1e-3, all three agents performed 6well (refer to Figure 5). In fact this learning rate provided the highest reward for all 3 agents across all learning rates and hence was chosen as the best learning rate. **PPO outperformed TRPO and VPG with the highest reward.**

4.4 Learning Rate 3e-4

With this learning rate PPO outperformed TRPO and VPG (refer to Figure 6). Even though all 3 algorithms showed good performance, rewards were still lower than 1e-3. PPO had the best momentum for this learning rate.

4.5 Learning Rate 1e-4

With the learning rate of 1e-4, VPG did not perform well as can be seen with the fairly flat reward function (refer to Figure 7). TRPO outperformed PPO by a relatively small margin. However, in the learning curve graph, PPO had a much higher slope as the training neared completion (around 30000 episodes). This indicated that, although PPO had a smaller reward, it had a higher learning momentum, and therefore performed the best for learning rate 1e-4.

4.6 Learning Rate 1e-5

With the learning rate of 1e-5 both PPO and VPG did poorly as can be with the flat line for the reward growth with episodes (refer to Figure 8). Even though TRPO showed a fairly good learning curve, the reward at 40,000 episodes was almost half that of other learning rates, such as 1e-3 and 3e-4. For this reason the results from this learning rate were disregarded.

4.7 Final Rankings

Therefore, PPO was determined to be the most effective algorithm for this task as it not only outperformed TRPO in two out of three learning rates and VPG in all three learning rates, but also consistently showed to have a higher learning momentum through the learning process. TRPO was given the second-highest rank as it consistently obtained better rewards than VPG, and also had a much better learning curve than VPG. Learning rate 1e-3 was chosen as the best learning rate for the algorithms as the algorithms performed best with this learning rate.

5 Conclusion

The purpose of this project was to determine which reinforcement learning algorithm would perform the best to learn to walk, a simple locomotion task, in a simulated environment. The hypothesis, based on previous studies, stated that the Proximal Policy Optimization Algorithm would perform the best, followed by the Vanilla Policy Gradient and the Trust-Region Policy Optimization. Algorithms were graded across a set of criteria, which included the 100-episode average reward after training, the speed of the learning process, the consistency of improvement across episodes, and others.

The hypothesis was partially supported, as the PPO algorithm outperformed TRPO and VPG, in that order. The results from learning rates 1e-1, 1e-2, and 1e-5 were disregarded because they failed to meet the criteria. All algorithms performed best for learning rate 1e-3, and with that learning rate, PPO outperformed TRPO and VPG, in that order. With learning rates 1e-3 and 3e-4, PPO outperformed TRPO, and even though it performed marginally worse with Learning rate 1e-4, it consistently showed to have a higher momentum throughout the learning process. VPG performed significantly worse than PPO and TRPO on all three learning rates examined.

6 Acknowledgements

I would like to thank the following people for helping me with my project:

- Professor Alexander Ihler, Associate Professor of Information and Computer Science at University of California, Irvine and Director of UCIs Center for Machine Learning and Intelligent Systems
- Michael Schaarschmidt, PhD Student, Networks and Operating Systems Group Computer Laboratory, University of Cambridge, and co-author of Tensorforce
- Alex Kuhnle, PhD Student, NLIP group at the Computer Laboratory, University of Cambridge, and co-author of Tensorforce

References

Agarwal, S. (2018). Approximately Optimal Approximate RL, TRPO. Github, IEOR 8100, https://ieor8100.github.io/rl/docs/Lecture%207%20-Approximate%20RL.pdf.

Baird, L., Moore, A. (1999). Gradient Descent for General Reinforcement Learning. ArXiV, MIT Press, 1999, https://www.ri.cmu.edu/pub_files/pub1/baird_leemon_1999_1/baird_leemon_1999_1.pdf.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., Zaremba, W. (2016). OpenAI Gym. ArXiV, OpenAI, 5 June 2016, https://arxiv.org/pdf/1606.01540v1.pdf.

Goodfellow, I., Bengio, Y., Courville, A. (2107) Deep Learning. MIT Press.

Grosse, R. (2017). CSC2541 Lecture 5 Natural Gradient. Github, University of Toronto, http://csc2541-f17.github.io/slides/lec05a.pdf.

Kakade, S., Langford, J. (2002) Approximately Optimal Approximate Reinforcement Learning. EECS, UC Berkeley (2002), http://people.eecs.berkeley.edu/~pabbeel/cs287-fa09/readings/KakadeLangford-icml2002.pdf.

Kingma D., Ba, J. (2015). ADAM: A Method for Stochastic Optimization. ArXiV. Retrieved May 8 2018, from https://arxiv.org/pdf/1412.6980.pdf.

Kullback, S., (1959). Information Theory and Statistics. Dover Publ.

Lange, K., (2007). The MM Algorithm. UC Berkeley Statistics, UC Berkeley (Apr. 2007), http://www.stat.berkeley.edu/~aldous/Colloq/lange-talk.pdf.

Li, F., Johnson, J., Yeung, S. (2017). ”Lecture 14: Reinforcement Learning.” Stanford, Stanford May 2017, http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture14.pdf

Lu, Y. (2017). Notes on Trust Region Policy Optimization. Max’s Blog, University of Pennsylvania 25 June 2017, http://178.79.149.207/posts/trpo.html.

Li, Y. (2017). DEEP REINFORCEMENT LEARNING: AN OVERVIEW. ArXiV 15 Sept. 2017, https://arxiv.org/pdf/1701.07274.pdf.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A., Veness, J., Bellemare, M., Graves, A., Riedmiller, M., Fidjeland, A., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., Hassabis, D. (2015). Human-level control through deep reinforcement learning. Nature. Retrieved from https://web.stanford.edu/class/psych209/Readings/MnihEtAlHassibis15NatureControlDeepRL.pdf.

Peng, X., Panne, M. (2016). LEARNING LOCOMOTION SKILLS USING DEEPRL: DOES THE CHOICE OF ACTION SPACE MATTER? ArXiV, University of British Columbia 3 Nov. 2016, https://arxiv.org/pdf/1611.01055.pdf.

Ruder, S. (2017). An Overview of Gradient Descent Optimization Algorithms. ArXiV, Insight Centre for Data Analytics, NUI Galway Aylien Ltd. 15 June 2017, https://arxiv.org/pdf/1609.04747.pdf.

Schmidhuber, J. (2014). Deep Learning in Neural Networks: An Overview. ArXiV, University of Lugano and SUPSI 8 Oct. 2014, https://arxiv.org/pdf/1404.7828.pdf.

Schulman, J., Levine, S., Moritz, P., Jordan, M., Abbeel, P. (2015). Trust Region Policy Optimization. ArXiV. https://arxiv.org/pdf/1502.05477.pdf.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O. (2017). Proximal Policy Optimization Algorithms. ArXiV. Retrieved May 5, 2018, from https://arxiv.org/pdf/1707.06347.pdf.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., Driessche, G., Graepel, T., Hassabis, D. (2017). Mastering the Game of Go without Human Knowledge. Nature News, Nature Publishing Group, 18 Oct. 2017, https://www.nature.com/articles/nature24270.

Sutton, R., Barto A. (2015). Reinforcement Learning: An Introduction. Stanford, The MIT Press 2015, https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf.

Thomas, P., and Brunskill E. (2017). Policy Gradient Methods for Reinforcement Learning with Function Approximation and Action-Dependent Baselines. ArXiV, Carnagie Mellon University and Stanford University, 20 June 2017, https://arxiv.org/pdf/1706.06643.pdf.

Williams, R. (2017). Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. ArXiV, Northeastern University, http://www-anw.cs.umass.edu/~barto/courses/cs687/williams92simple.pdf.

Received August 2, 2018 , revised September 30, 2018, accepted October 1, 2018