DeepHEC: Deep Reinforcement Learning for robotic Hand-Eye Coordination

Hayden Sampson <sampsoh@edgehill.ac.uk>

2021

Abstract

Teaching robots seemingly simple skills is a difficult challenge. Tasks that come second nature to humans, such as picking, packing, stowing, inserting and grasping, are exceptionally arduous for robotic systems to learn and perform effectively. Recent advancements in Deep Reinforcement Learning (DRL) allow robots to learn these skills from input-output data through iterative training cycles without having to derive the underlying mathematical models. Here, we explore the effectiveness of DRL for robotic control that involves visual pixel image inputs and arm control action output for a task concerning grasping a diverse range of objects from a table. This research extends current work by including methods to improve data efficiency of algorithms, such as Hindsight Experience and introducing the use of Attention Mechanisms to evaluate performance on such a task and environment with a sparse reward system. Our results show... indicating...

1 Introduction

The use of reinforcement learning methods to perform and succeed in certain tasks and increased significantly over recent years. Initial success using reinforcement learning was presented in its' ability to learn how to solve certain simple tasks such as balancing an inverted pendulum [1] or learning to land a lunar lander between two flags [1]. This opened up opportunities to explore the effectiveness of reinforcement learning for tasks requiring pixel image input by playing a range of Atari games [2]. Such study introduced more complex environments for reinforcement learning, such as those including much larger state and action spaces. Because of this, standard reinforcement learning was not viable for effective and efficient performance. Deep Neural Networks (DNN) solved this issue, with combination of reinforcement learning that introduced Deep Reinforcement Learning (DRL) methods to allow agents to outperform human level on a number of Atari games [2]. Since, there has been an exponential increase in work and study to further improve these methods and to succeed in a variety of other tasks that an agent could solve. One such problem includes

those tasks related to robotic arm control and movement, including grasping, inserting, pick-and-place and sliding. In today's world, it is important for robots with some Degree of Freedom (DoF) to be able to generalise to tasks concernc-ing arm movement and manipulation, without underlying mathematical models. Although there has been a lot of good work and study into the development of DRL models for these tasks [3] [4] [5], there has been yet to find an appropriate solution to generating high performing data efficient models. Many DRL meth-ods require a large amount of data, and therefore time, in order to be successful at the given task, especially for robotic arm control. Because of this, there is ample opportunity for exploration into developing DRL models that are much more sample efficient and can train much faster whilst providing a high level performance. In this case, we have explored two such methods: Episodic Self Imitation Learning (ESIL) [6] and Attention Mechanisms [7].
The following discussions will be presented as such: Section 2 as a Literature review to explore and discuss the relevant work appropriate to this study, Section 3 contains background describing some of the theory behind the algorithms and methods implmented in this study, Section 4 for a description our model architecture and environment, followed by Section 5 discussing the setup of our experiments and subsequent results. Finally, Section 6 will draw some conclu-sions on our work with a discussion for potential future work and improvements.

2 Literature Review

There has been extensive research in studies regarding the areas in Deep Re-inforcement Learning, including specific focus on the problems of robotic arm movement and grasping. DeepMind's Atari [2] first brought into the light the ef-fectiveness of using deep reinforcement learning and 'Deep Q Networks' (DQN) for outperforming human performance over a range of Atari games. The nov-elty of this study showed how these algorithms can make use of graphical pixel images as state inputs to output appropriate actions in the environment. What was also impressive was the breakaway from model-based methods to model-free, where a network could show similar high-scoring performances when trained on a variety of different games.
In terms specific to robotic arm control, Google Brain [3] has explored the subject through a comparative implementation of a number of well-known off-policy learning methods. This presented a good starting place for analysing algorithmic performances for such tasks, and has proven to be highly influential for the motivation behind this proposed research. Where the study lacks is in the comparison and analysis of on-policy methods such as TRPO [8] and PPO [9] learning, which have also been proven through other studies to be effective for robotic control [4]. As well as this, the study has omitted methods that have been proven to improve algorithm performance and sample efficiency, such as Prioritised Experience Replay [Schaul2016] or Hindsight Experience Replay [Andry2018], that also prove to be highly effective for robotic arm environ-ments.

Further research [Blake2020] explores increasing efficiency through attention models, with the aim of reducing the visual field of an agent to only focus on the important aspects of an input image, in order to reduce feature state space and dimensionality.

3 Background

3.1 Deep Reinforcement Learning

Generally, standard reinforcement learning is concerned with producing an *agent* that learns how to take appropriate *actions* within a given *environment* to collect *rewards* and succeed at a certain task. An indicator that a task has been achieved successfully is the amount of reward an agent will receive, therefore an agent will learn to act in whichever way will allow it to gain the most reward. Agents will make decisions on how to act based on their *observations* of the current *state* of the environment, producing state-action transition pairs. How an agent makes a decision on what action to take is called the agent's *policy*. Taking an action causes the environment to change and update state as well as giving the agent some reward. The agent can then make a new observation on the updated state and take a new action.

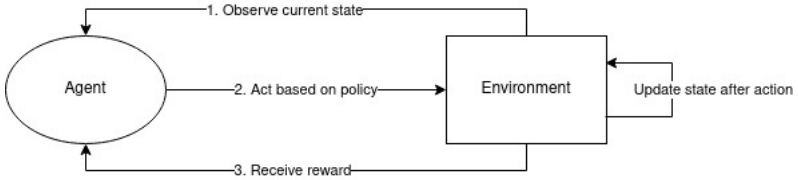


Figure 1: Reinforcement Learning Cycle

This cycle is repeated for T *timesteps* until the environment reaches its termination state, with the states between and including the initial state and final state defining an *episode*. The learning process therefore involves an agent figuring out which actions to take that will eventually lead to the greatest reward at the end of an episode, also known as its' *cumulative reward*.
The standard reinforcement learning approaches assume a discrete state and action space and hence, suffers from the *curse-of-dimensionality* problem, which simply states that computational complexity increases exponentially with increasing dimensions of state and action space. The problem of discrete state space can be solved by using deep networks that can take continuous values as inputs, including images. In other words, Deep Reinforcement Learning (DRL) methods use deep networks to overcome some of the limitations of standard reinforcement learning. These methods can be broadly divided into two categories: model-based and model-free methods. We are primarily interested in the model-free methods, that do not act based on their knowledge or estimation of the environment, but solely on the observation of the current state of the

environment arrived at by using a policy. These model-free methods can be further divided into two sub-categories depending on how the agents learn, namely policy-based and value-based learning. Value-based methods estimate a *Q-value* (cumulative future reward) to every state-action pair by using a Q-function *Q(s, a)*, then selects an action which gives a maximum Q-value at a given state. So, in essence, value-based learning is concerned with estimating or approximating the Q-function. On the other hand, policy-based learning aims at estimating a policy directly without estimating a Q function. Policy parameters are updated to maximize or minimize a given objective function. For instance, policy gradient methods update the policy parameters by applying gradient ascent to maximize a given objective function. The advantage of using policy-based learning for this research problem is that it can be much more easily applied to agents with a continuous action space, as an action distribution is being directly output from the network, rather than a Q-value for each possible action available. It is possible to combine these two approaches in the form of an Actor-Critic algorithm [KonTsi2000] that employ a policy-based method to choose an action, with a value-based method to assign values to specific state-action pairs that critique how good that action was. Actor-Critic methods will form the underlying architecture used in this research. Another categorization can be made based on how the agents are trained. The on-policy methods train the agent using data generated using its current policy. This is an online approach where the agent is learning while doing its job. In contrast, off-policy methods use a set of experience for training which are generated by a different policy than the one currently being used by the agent. Here, an on-policy approach is implemented through the state-of-the-art Proximal Policy Optimisation (PPO) algorithm.

3.2 Proximal Policy Optimisation

Proximal Policy Optimisation (PPO) is an on-policy algorithm first introduced by [9], that sought to improve performance of Trust Region Policy Optimisation (TRPO) [8], whilst also simplifying implementation and increasing sample efficiency. The idea behind the algorithm is simple, limit the magnitude in which the policy is updated after each training step to erase the possibility of the gradient update stepping too far, in which doing so controls the stability of the training process. The step control can be implemented through two methods: value clipping or through the use of a KL penalty. For clipping, there is simply a value ϵ that acts as the upper and lower boundaries for the step magnitude. Alternatively, a KL Divergence method can be used to describe the similarity between two probability distributions and constrains the objective step function based on this. For this research, PPO-Clipping has been implemented.
PPO training consists of collecting experience data over K timesteps in the form of $< s, a, r, ns, d >$, where **s** represents the current state of the environment, **a** the action taken in this state, **r** the reward collected for taking this action in this state, **ns** the next state achieved through taking the action in this state and **d**, the inidicator whether the episode has terminated at this state. Thus

K samples in this form are collected, know as the set of *trajectories*. The total *returns* (V) and *advantages* (A) and then computed on this set using a *Generalised Advantage Estimation* (GAE) method [10], which provides values used in the Actor-Critic step functions.

Algorithm 1 PPO-Clip

1: Input: initial policy parameters θ_0, initial value function parameters ϕ_0
2: **for** $k = 0, 1, 2, ...$ **do**
3: Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
4: Compute rewards-to-go $\hat{R}_t$.
5: Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.
6: Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min \left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), \;\; g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$$

typically via stochastic gradient ascent with Adam.
7: Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

typically via some gradient descent algorithm.
8: **end for**

Figure 2: PPO
OpenAI SpinningUp implementation [11]

3.3 Episodic Self Imitation Learning

In regards to improving DRL methods to be more sample efficient, a lot of promise has been presented through the use of Hindsight Experience Replay (HER) [12]. This works more specifically for environments with sparse rewards, such as the one explored in this study. Originally, HER as a method had only been applicable to use with off-policy methods, such as Deep Deterministic Policy Gradient (DDPG) [5]. This is because the hindsight experience used in training is collected from a memory buffer that contained samples from all past policies. In order to make hindsight experience compatible with on-policy methods, Episodic Self Imitation Learnig (ESIL) [6] was introduced which combined aspects of Self Imitation Learning (SIL) [13] and HER to allow hindsight experience collected from the current policy only to be used in training. ESIL extends standard Actor-Critic routines by introducing *goals* as states of which the agent is trying to achieve. As well as an initial state, a desired goal is initialised for the agent. The agent will then only receive a reward if the desired goal is reached, which is often not the case in a sparse reward system. For the agent to effectively learn from this, the achieved state of the agent is then considered to be the original desired goal as hindsight experience. This hindsight experience is then stored in memory along with the original trajectory experience. Therefore

during training, the agent learns from sampling all of this experience. In the case where the agent fails to reach the original desired goal, it will still be able to learn useful information from the hindsight experience. This is the general concept of goal-based hindsight experience methods.

3.4 Attention Mechanisms

Attention Mechanisms were first introduced in [reference] and were inspired by the way humans perceive and focus on certain aspects of our vision to take in the most important information for whatever task we are undergoing. Two main types of attention were realised, bottom-up saliency attention and top-down task relevant attention (or vice versa, check this). Saliency attention comes from the idea of what grabs our attention that is in our field of vision, in other words what is the most visually interesting thing that demands attention. Task related attention comes from the idea of what we need to pay attention to in order to fulfill a task, for example throwing a ball into a hoop, we would likely focus most of our attention on the hoop. Attention can be implemented through a number of ways [references], though here we opt for a *Self Attention* [reference] approach after the image input has passed through the Convolutional Neural Network process. This allows the Attention mechanism to attend to a much simpler, filtered input sequence, rather than a more complex multi-dimensional image input. Attention mechanisms make use of input sequences to produce *key* and *query* vectors that undergo some matrix manipulation and softmax activation to weight indices of the sequence that should be attended to. The output of a weighted sequence therefore allows a network to indentify which parts of an input have a higher importance in regards to affecting performance. This saves the agent from needing to process and attend the entire input equally, as it learns to be more efficient by only processing the important areas of the input.

4 Model Architecture and Environment

4.1 Approach

In this research, we primarily make use of actor-critic models as shown in Figure 3 below. In this model, we have two separate networks - one for estimating the policy required for generating the necessary action and the other for approximating the value function that can be used to evaluate the goodness of the action taken. These two networks are updated iteratively to find the optimal policy for a given problem by maximizing the total cumulative reward. This architecture provides flexibility in selecting different architectures and algorithms for each of the actor and critic modules independently. Each of the actor and critic models will use a deep neural network (DNN) to estimate action and value from the input state information. The environment provides the reward and the next state for a given state-action pair. The goal of learning will be to update

parameters and so that certain objective cost functions are either maximized and minimized.

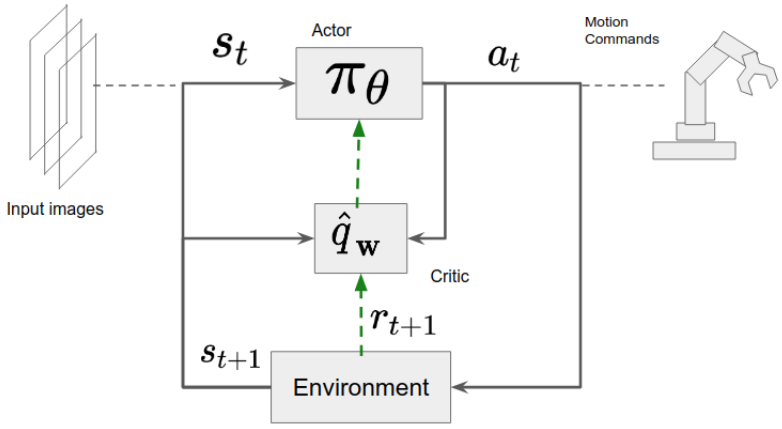


Figure 3: Actor-Critic Architecture

4.1.1 Simulation Environment

Our objective is to develop, test and validate new deep reinforcement learning algorithms for solving robotic problems. In this regard, we have selected PyBullet's KukaCam [14] as our simulation environment. Some of the screenshots for this environment are shown in Figure 4. In this environment, a 6DoF Kuka robotic arm is expected to pick up different kinds of objects from the tray kept on the table based on the input images seen by an overhead camera. The view of the camera is shown in the top left corner of each image. This image along with the current joint values of the robotic arm (robot pose) forms the current 'state' of the environment. The agent is rewarded a value of 1 whenever the robot successfully picks up an item from the tray, otherwise the rewarded value is 0, implying a sparse reward system. Each episode consists of a series of robot movements attempting to pick an item. The objective is to pick the item successfully based on an input image obtained from the overhead camera.

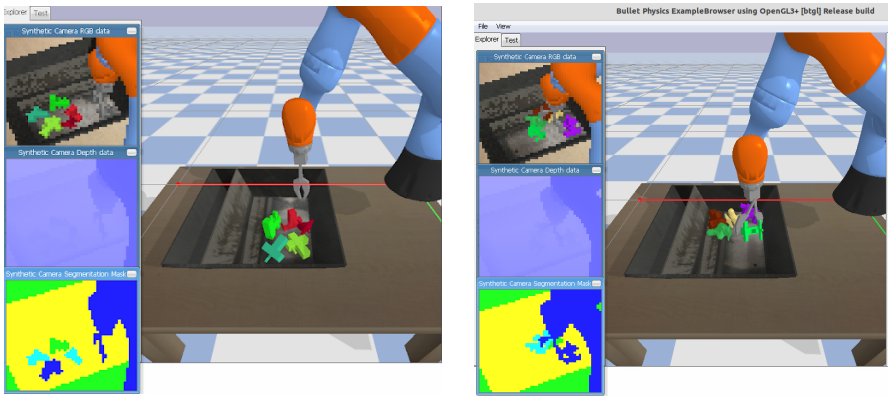


Figure 4: Pybullet KukaCam Diverse Object Environment

4.1.2 Convolutional Neural Network

Our Convolutional Neural Network (CNN) takes as input a pixel image of size 48x48x3 (H, W, C). This is then passed through multiple layers of Convolution, Batch Normalisation and Max Pooling until it is eventually Flattened. The flattened layer output is then passed through two fully connected layers two produce the final output. To the best of our knowledge there is no standard CNN architecture or pre-trained model set out for such a task, therefore the design we have implemented is intended to be simple yet deep enough so that it produces an appropriate convolution output. The CNN implemented is not pre-trained and is trained along with the Actor and Critic networks. A visualisation of our CNN can be seen in Figure 5.

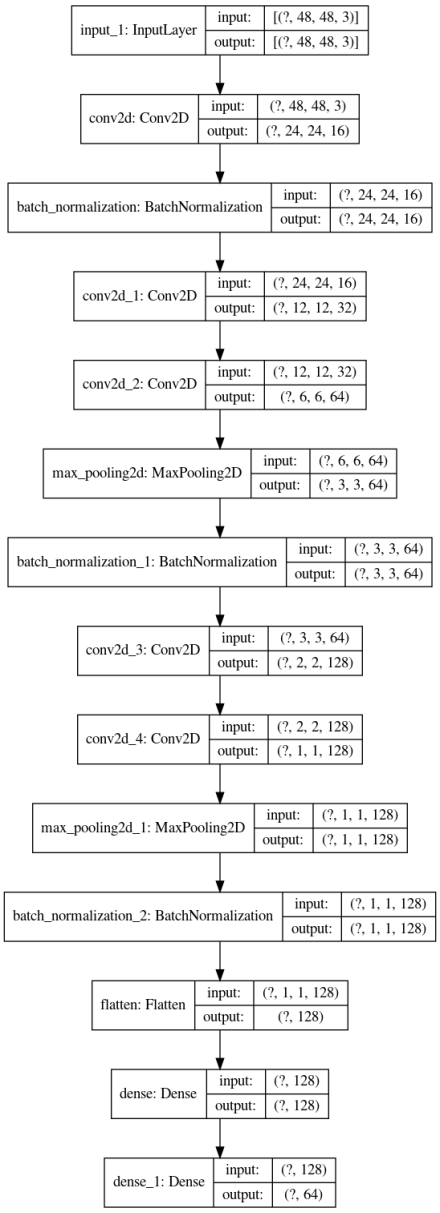
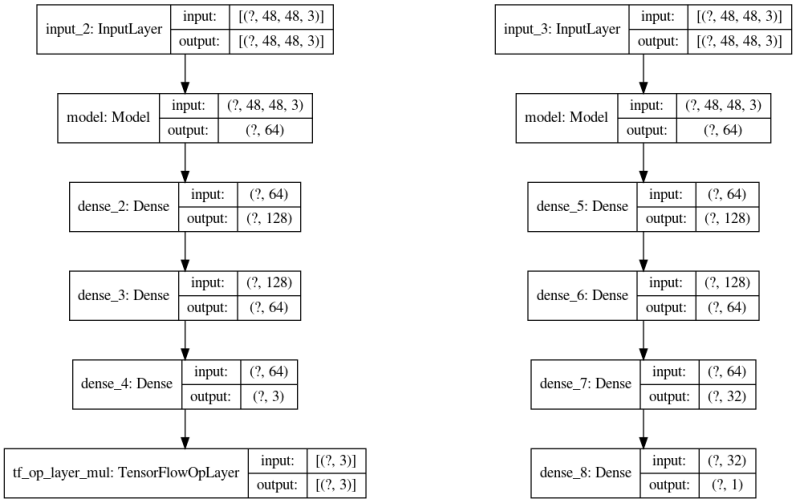


Figure 5: Convolutional Neural Network

4.1.3 Actor-Critic Network

Our Actor and Critic networks follow a simple design where the majority of complex layers are contained in the CNN. Both Actor and Critic take as input the pixel image of size 48x48x3 and pass the input to their CNN. The output

of the CNN then passes through three fully connected layers. For the Actor network, an output of appropriate action size is required in order to produce a valid action for the agent, whereas for the Critic network an output of a single value is required for the appropriate valuation of a state.
Visualisations of the Actor and Critic networks can be seen below in Figure 6 (a) and (b), respectively.



(a) Actor Network (b) Critic Network

Figure 6: Actor and Critic Networks

5 Experiments

5.1 Hyperparameters and Setup

All experiments have been performed on an HP Omen GPU laptop for a maximum of 1000 seasons. Here, a seasons consists of 1024 timesteps. Therefore, experiments are ran for a maximum total of 1000x1024 or 1,024,000 timesteps. If an agent manages to reach a set success value, then it has also been considered to solve the problem and therefore will also end the experiment. For these experiments, when an agent succeeds in picking up an object an average of 100 times over the previous 50 seasons, then it is considered to have solved the problem. Policy training data is collected over a season and then training is completed using batches for a number of epochs. The seasonal data is then cleared so that new data is collected on the updated policy. A validation procedure is ran after every 10 seasons to test the performance of the current policy. This validation runs 50 episodes of the environment, where the agent's success rate is calculated. This is to evaluate the performance of the current policy that is separate from the main process of seasonal episodes and timesteps. An

episode generally consists of 7-8 timesteps with a maximum of 20.

Hyper-parameter	**Round 1**	**Round 2**	**Round 3**
Seasons	1000	1000	1000
Success value	100	100	80
Actor Learning Rate	0.0002	0.001	0.0002
Critic Learning Rate	0.0002	0.001	0.0002
CNN Learning Rate	0.0002	0.001	0.0002
Epochs	10	10	10
Training Batch (steps per season)	1024	1024	512
Batch Size	128	128	128
Clipping Value ϵ	0.07	0.07	0.2
Discount factor γ	0.993	0.993	0.99
Lambda λ	0.7	0.7	0.95

Table 1: Hyperparameters

The second round of experiments explores how the learning rate of the networks affect performance. The third round of experiments explores how a larger clipping value and smaller season size affects performance. A larger clipping size will mean that the policy will be able to make a more sizeable change at each training step, whilst a smaller season size will mean there is less current policy data to train on In a final round of experiments, we also consider decaying the clipping value and entropy coefficient over training to see what impact[reference] this has on performance. After a season has been completed, the clipping value ϵ and entropy coefficient and multiplied by 0.999 and 0.998, respectively.

5.2 Results

5.2.1 Base

5.2.2 Increased Learning rate

5.2.3 Alternative parameters

5.2.4 Decaying clipping value

5.2.5 Overall Comparison

PPO
PPO+ESIL
PPO+Attention
PPO+ESIL+Attention

6 Discussions and Conclusion

6.1 Conclusion

6.2 Future Work

References

[1] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[3] Deirdre Quillen, Eric Jang, Ofir Nachum, Chelsea Finn, Julian Ibarz, and Sergey Levine. Deep reinforcement learning for vision-based robotic grasping: A simulated comparative evaluation of off-policy methods. *CoRR*, abs/1802.10264, 2018.

[4] A. Rupam Mahmood, Dmytro Korenkevych, Gautham Vasan, William Ma, and James Bergstra. Benchmarking reinforcement learning algorithms on real-world robots. *CoRR*, abs/1809.07731, 2018.

[5] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. In Yoshua Bengio and Yann LeCun, editors, *ICLR*, 2016.

[6] Tianhong Dai, Hengyan Liu, and Anil Anthony Bharath. Episodic self-imitation learning with hindsight, 2020.

[7] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[8] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization. *CoRR*, abs/1502.05477, 2015.

[9] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[10] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation, 2018.

[11] Joshua Achiam. Spinning up in deep reinforcement learning. ., 2018.

[12] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. *CoRR*, abs/1707.01495, 2017.

[13] Junhyuk Oh, Yijie Guo, Satinder Singh, and Honglak Lee. Self-imitation learning. *CoRR*, abs/1806.05635, 2018.

[14] Y. Bai E. Coumans. pybullet, a python module for physics simulation in robotics, games and machine learning, 2017.