Customisable Control Policy Learning for Robotics

Ce Guo*, Wayne Luk*, Stanley Loh Qing Shui*, Alexander Warren† and Joshua Levine†
*Imperial College London, United Kingdom
Email: {c.guo, w.luk, qing.loh17}@imperial.ac.uk
†Intel Corporation, United Kingdom
Email: {alexander.warren, joshua.levine}@intel.com

***Abstract*—Deep reinforcement learning algorithms integrate deep neural networks with traditional reinforcement learning methodologies. These techniques have been developed and used for various applications to produce exciting results in many fields, including robotics. However, physical robots require a large amount of training episodes which can damage the robot if directed by immature policies. Training using simulations can serve as a viable alternative before a robot is deployed in the field. This study addresses a computational challenge of deep reinforcement learning by developing a hardware architecture for the Deep Deterministic Policy Gradient (DDPG) algorithm. Additionally, we identify the customisation opportunities for a full-stack development framework with reinforcement learning to discover control policies for robotic arms. Finally, we transfer policies encoded in fixed-point numbers from our FPGA DDPG implementation to a robotic arm to evaluate the feasibility of our learning platform.**

I. Introduction

The conventional development procedure of robots involves programming a set of action policies. The explicit specification of policies is adopted when the working environment is uncomplicated and the robot has a limited number of simple sensors. However, working environments of modern robots are increasingly complicated and unpredictable. Robots are equipped with various modern sensors, such as lidars and cameras, to cope with complicated environments while ensuring safety. Programming these robots becomes tedious and error-prone. One way to improve the programming quality and productivity is to train robots by rewarding expected actions rather than hard-coding policies. A critical technology for training robots is reinforcement learning.

A direct way to train robots with reinforcement learning is to run learning algorithms on the physical robotic system. However, training on physical robots may not be practical because a robot can cause physical damage to humans, the environment and itself during training. The risk of physical damage is especially high at the early stages of training when the policy set is immature. Apart from safety issues, the training procedure is often too slow. In particular, reinforcement learning algorithms require a large amount of robot-environment interaction before finding a reasonable policy set. Moreover, the environment may not be able to give an instant reward for the robot after each action, because the value of the reward depends on sensor data or human attendance.

Training robots in a simulated environment is an alternative that reduces cost and improves efficiency. Simulated robots cause no physical damages. Also, the interactions can be fast as they occur digitally. However, the policy learning procedure still consumes a large amount of time even with simulation because the training process of function approximators involves time-consuming function optimisation. In this study, we employ Field Programmable Gate Arrays (FPGAs) to accelerate reinforcement learning for robotic arms. FPGAs provide substantial speedups to various deep learning algorithms which benefit reinforcement learning.

To the best of our knowledge, policies learned on FPGAs in existing work [1] can only process discrete action spaces. Moreover, due to the limited complexity of policies, the system in [1] can only drive the robotic arm in a two-dimensional space. This paper is the first to transfer policies learned on an FPGA to a physical robotic arm to move objects with continuous actions in a three-dimensional space. Key contributions of this paper include the following:

1) A customisable hardware architecture for the deep deterministic policy gradient (DDPG) method [2]. (Section III)
2) A policy learning platform for the DDPG architecture including a 3D-printed robotic arm and its simulator in the virtual space. (Section IV)
3) An empirical study on the efficiency and quality of policy learning on CPUs and FPGAs. (Section V)

II. Background

This section reviews the deep deterministic policy gradient (DDPG) method and existing hardware accelerators for reinforcement learning.

A. Deep Deterministic Policy Gradient

The Deep Deterministic Policy Gradient (DDPG) method [2] is rooted in Deep-Q-Networks (DQNs). Algorithms derived from DQNs [3], [4] show increasing success in reinforcement learning problems with high-dimensional state and action spaces. The original deep-Q-networks solve problems with discrete action spaces. However, practical applications often have continuous action spaces. A natural solution is to discretise the action space. Unfortunately, the discretised action space scales exponentially with the number of degrees of freedom in the continuous problem.

Actor-critic methods circumvent the problems of continuous action spaces by introducing a function approximator to choose the action for any set of states. Combined with another function approximator to determine the value of a state-action pair, the actor-critic method is able to work efficiently in both continuous state and action spaces. Back-propagation is carried out first on the critic network. Once the update is complete, back-propagation is carried out again on the critic network, but this time the back-propagation goes back through the action inputs to the actor network, triggering a gradient update. Policy gradient methods are frequently used with stochastic policies where policies are represented using a probability distribution. Actions are sampled and used to estimate the return of the transition. Gradient ascent is then applied to maximise the total reward. The use of deterministic policies and policy gradients is a recent development in reinforcement learning thought to not have a model-free form. However, it was recently found that a such a form did exist, which enables continuous state/action space problems to be learnt without the need for complex environmental information. The DDPG method is one of the pioneering model-free methods for such a class of problems.

Key steps of the DDPG algorithm include the following: (1) Action selection. Given the current state, the action selection step performs a forward propagation using the actor network to find a proper action to execute. (2) Action execution. Given the action from the previous step and the current state, the environment changes to a new state. (3) Experience sampling. This step consists of storing state, action, reward and transition information from the current step as well as generating random numbers in order to select the set of experiences to be used as training samples. (4) Back-propagation. The gradients for the actor and critic networks are computed. (5) Network update. A function optimiser updates the evaluation network using the gradients. The update rule contains many expensive multiplication and division operations along with exponential operators.

B. Reinforcement learning on FPGAs

Robotic systems based on reinforcement learning usually have latency requirements or power constraints. For instance, the control signals of motors need to be generated in real time responding to sensor data, while autonomous wheeled robots are usually battery-powered for high mobility. ASICs and FPGAs are well known for their low latency and low power consumption. However, hardware-based reinforcement learning implementations have not been a popular topic in research. As such, there is little support from third-party software, libraries and simulators. Whilst hardware-based inference engines for neural networks are relatively common, their application to reinforcement learning problems are few. Papers on hardware-based reinforcement learning include an accelerator for deep Q-Learning in 2017 [5] and two papers on trust region policy optimisation in 2017 [6] and 2018 [1].

The dearth of research provides a significant opportunity to investigate the effectiveness of combining reinforcement learning and hardware acceleration. This allows for specialist robotic systems to be built without commercial robots which can constrain the learning problem. Large cost savings are also achieved as traditional off-the-shelf robotic systems are often highly expensive due to the associated production and design time value.

III. Hardware architecture for DDPG

We describe a customisable hardware architecture to speed up the DDPG method in this section. The architecture on the FPGA platform collaborates with a CPU host to learn policies efficiently. Specifically, the CPU streams network parameters and the state transitions to the FPGA. The FPGA computes the gradients of the network parameters and sends the gradients back to the CPU. An optimisation algorithm on the CPU platform then updates the network parameters using these gradients.

The neural networks in DDPG are constructed using a cascade of alternating layers. The structures for the forward pass for the action network and the target network in the back-propagation step are identical, except that the evaluation network parameters are used for the actor gradient computation while the target network parameters are used for the critic gradient computation. This allows the same set of hardware resources to support both the forward pass and the backward pass.

Through the use of parameter and input controllers, parameters and environment experiences are selectively streamed out of the on-chip memory to the processing elements (PEs) of each layer. Section III-A and III-B respectively describe the PEs for the odd layer and the even layer. Section III-C discusses the CPU-FPGA interaction including streaming and padding.

A. Odd layer

In order to form a continuous pipeline for the DDPG, we extended the multiplication method described in [6] to support matrix-matrix multiplications. At each clock cycle, an element from each block of the weight matrix is multiplied by a subset of elements from the input matrix.

Fig. 1 illustrates the process for the first valid clock cycle of the matrix multiplication. Each element in a blue row of the weight matrix is multiplied by the element at the corresponding position in a blue column of the input. The products of each set are summed, forming a multiply-and-add (MADD) tree. The example matrix multiplication facility in Fig. 1 has $4 \times 4 = 16$ MADD trees.

For the second clock cycle, the process is repeated with the second element of the blue rows and columns. Through the use of a multiplexer, the new output is added to the accumulated value to receive the cumulative sum. Once the output of the last element of each row is added to the accumulator, the dot product between each selected row with all selected columns in the input is computed. The bias is added and the activation function of the layer is evaluated. The activation gradients are also computed in the activation unit and stored in on-chip memory for the backward pass later. A signal to the next layer indicates whether the outputs of the odd layer is valid to avoid calculations with incomplete outputs.

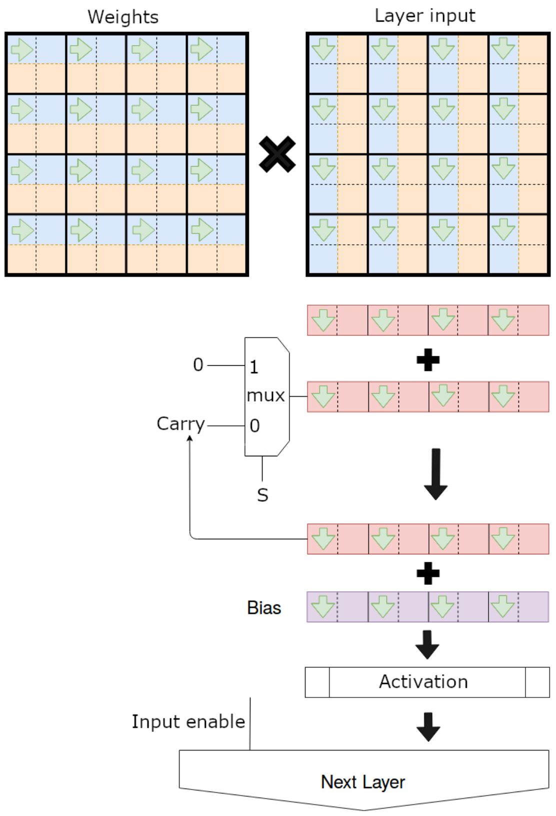


Fig. 1. Matrix multiplication in odd layers

As the orange columns of the input matrix arriving from the previous layer may not be ready, the dot products for the odd layer are computed in column-major order of the weight matrix. The odd layer's matrix multiplication algorithm operates on the premise that the entire input vector, i.e. a blue column, is available. The second set of dot products for our example are computed using the orange rows of the weight matrix and the blue columns of the input matrix. Once all rows of the weight matrix have been used for the dot products, the input controller begins to select elements from the orange columns to compute the next set of outputs. A running count of valid inputs identifies the availability of the selected input. If the selected input is not available, the matrix multiplication stalls to guarantee the correctness of the product.

B. *Even layer*

Inputs to an even layer have a different pattern from an odd layer. A full dot product is computed in the previous layer at regular intervals assuming no stalling. Therefore, the even layer receives a set of inputs at regular intervals. Reusing the matrix-multiplication processing element for the odd layer is inefficient as the PE would be stalled due to the periodic trickle of inputs.

To have the even layers perform useful computations in such a scenario, the weight selector traverses through row-major order first before a different column is selected. This allows multiple computations to use the same input before the next input is required to continue with the dot product.

Partial results of each row are stored in an array of accumulators. Once the next element of the same row is used, the partial results are added to the output of the MADD tree. Similarly to the odd layer, the output from the even layer is considered valid if the last element of each row is selected. The key difference between an even layer and an odd layer is that the output from the even layer releases a burst of valid outputs instead of a consistent stream.

C. *CPU-FPGA interaction*

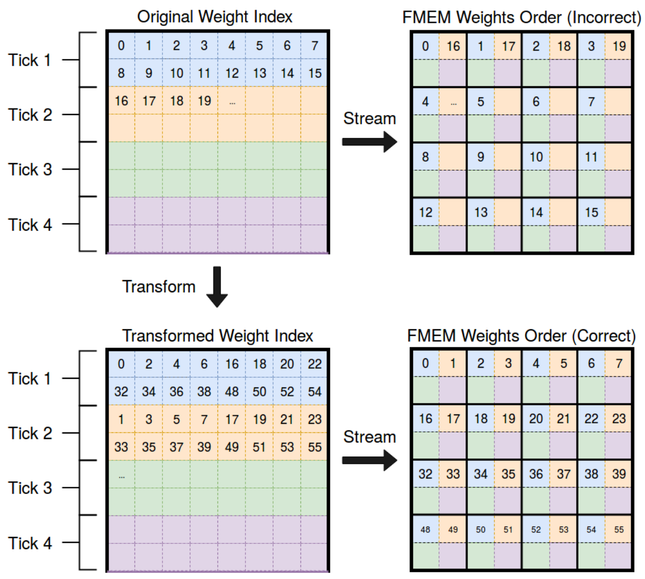
One of the caveats of the DDPG algorithm is that the input to the critic network requires the output of the actor network. It is not trivial to coordinate the flow of intrinsic state inputs from the CPU and derived action outputs from the actor network to arrive at the same processing element of the first critic layer. We differentiate between the two sets of weights to be applied to the two types of inputs and split them into two separate layers. Computations run independently in these two distinct layers and their outputs are given to a fusion layer which synchronises the data from the two input streams. No bias term is added and a linear activation function is used in order to produce correct results. The inputs are first stored in the on-chip memory and read when there are sufficient inputs in both sets of memory. The bias for the critic layer is added to the fusion layer and the activation function is applied thereafter. The activation gradient is also stored in the fusion layer.

To parallelise the computations across the PEs of each layer, the original matrices from the CPU must first be transformed so that the order of inputs into the MADD trees is correct as illustrated in Fig. 2. We reorder the inputs on the CPU platform as it is resource-intensive and difficult to implement on the FPGA platform. There is a slight computational overhead for the transformation, but the overhead is minuscule.

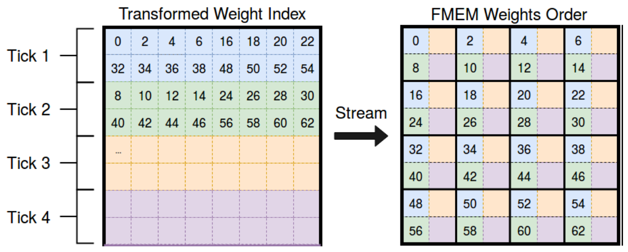
To minimise the initial rollout period of the DDPG pipeline, the order of the weights for the even layers is based on the transpose of the weight matrix. Instead of waiting for all weights to stream into their respective layers, this optimisation allows all layers to immediately begin forward propagation as long as any weight/input pairs are ready.

Stream padding is necessary for matrix dimensions that do not match the parallelism factor. This is easily and quickly achieved on the CPU using memory copy instructions. Additionally, stream padding is used as a buffer against loop latency issues found in the even layer's accumulator banks. In order to compute the correct results from the even layer, the output block dimension has to be padded to be at least the same value as the loop latency.

The output of the loss layer is fed back into the last layer of the critic network. As a result, the type of matrix multiplication PE has to be changed. Odd layers will require an even layer PE and vice versa. Once any feedback input enters a layer, the Hadamard product between the input and the activation gradient (stored during the forward propagation step) results in the gradient of the bias term which is passed back to the



(a) Odd layers



(b) Even layers

Fig. 2. Weight transformation

CPU to perform bias updates. It is simultaneously stored in the on-chip memory to be used to compute the gradients.

In order to reduce the amount of DSPs used for computing the weight gradients and feedback gradients, a pair of 2-input multiplexers are used to select the inputs based on the type of gradient being computed in that clock cycle. The node delta is computed first in order to complete the rollout of inputs to all layers in order to maximise throughput. Once the last set of node delta is computed, the selector switches and the matrix-multiplication PE begins computing the gradient which is subsequently transferred back to the CPU host.

IV. Policy learning platform

This section presents an efficient policy learning platform for robotic arms using the proposed hardware architecture. The platform is different from the one in [1] regarding kinematics and the action space. This section provides a detailed description of the platform as a reference of the problem complexity and an example of mechanic settings. The platform includes (i) a 3D-printed robotic arm which executes the policies learned on the DDPG architecture, (ii) a simulator of the arm which produces data for the DDPG architecture.

A. 3D-printed physical robot

Amongst many academic circles, off-the-shelf robotic arms are generally used for research in various fields of engineering and science. They are designed to be general purpose and contain reasonably powerful motors and feedback sensors to minimise the efforts on mechanics. Naturally, such flexibility comes at a significant cost in terms of price and customisability. In this study we combine the robot and task specifications to design a feasible robotic robotic arm configuration.

We use a 4-DoF (Degrees of Freedom) robotic arm. The robot consists of four rigid links connected with three joints. The first link counting from the ground stands upright on a rotating base while the other three links are free to move in the space. A robotic arm with a large number of degrees of freedom is inherently more flexible but becomes more difficult to control for three reasons:

1) Regarding mechanical engineering, a large number of links necessitates a larger number of motors and actuators to control the links. Motors are generally bulky and heavy. Motors close to the base would have to be much stronger in order to lift each subsequent link-motor extension. With too many link-motor extensions, the required motor can be costly.
2) Regarding computer simulation, a high DoF requires a long simulation time as the state-space of the problem scales exponentially with the DoFs.
3) Regarding reinforcement learning, a higher DoF results in more complicated system dynamics. The reinforcement learning method has to employ a more sophisticated model, such as a deeper neural network, to capture the interaction patterns.

We customise a parameterisable, open-source, and 3D-printable robot [7] by Zortrax, a 3D printing company. The design itself is inspired by the KUKA robotic arm which is commonly used for industrial applications. The customised robotic arm utilises stepper motors for actuation, which is different from the servo-driven KUKA robot. The step angle of the stepper motors is 1.8 degrees, which enables the arm to move in precise steps and to make minute rotations. Such minor movements are not possible using servos at the same price level, as sufficient current has to be channelled to the servo to overcome the deadband. The robotic arm components are printed using a 3D-printer.

Key customisations on mechanics include the following. The link lengths are adjusted to fit our customised robot specification. The pitches of the gears are adjusted to minimise backlash. Large parts are dissected into smaller sub-components in order to fit into the build area of our 3D printer. We use an electromagnet to act as the effector of the robot. The electronics used are sized according to the new weight distribution and energy requirements.

Fig. 3 shows the boundary positions of the robotic arm in the experiments. Fig. 4 presents two photos of the 3D-printed robotic arm.

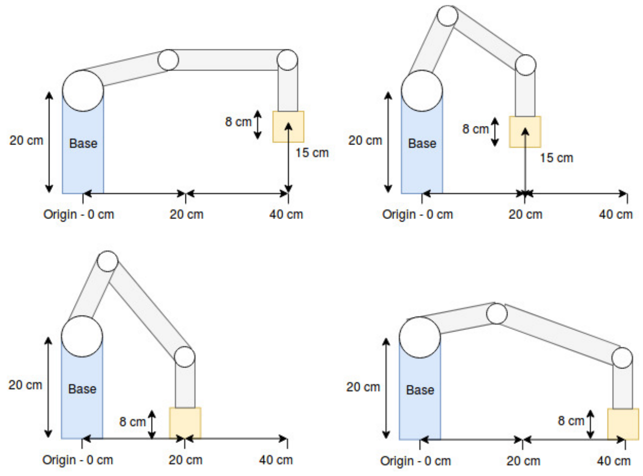
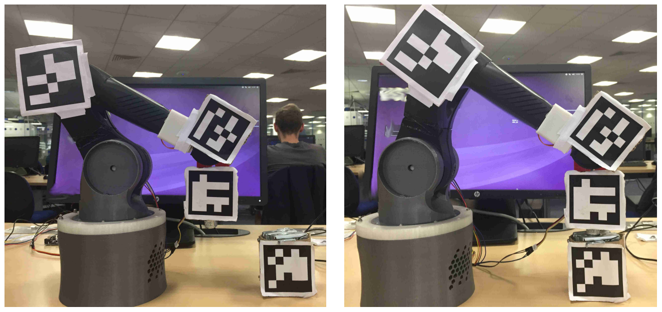


Fig. 3. Boundary positions of experimental robotic arm



(a) Box not attached (b) Box attached

Fig. 4. 3D-printed robotic arm

B. Simulation

Simulation improves the learning efficiency by eliminating physical interactions. In our learning platform, two pieces of code are developed respectively to simulate the robot and its environment.

The state space and action space are defined as follows. The state space of the robotic arm environment contains the distances of the links to the box. Each component of a distance vector is represented as a real number. The goal states represent the intended position the robotic arm is to bring the box to. In addition to the distance vectors, the state space also includes two Boolean inputs that indicate if the box is picked up by the arm and if the box is sufficiently close to the goal state. The action space consists of 5 outputs, 4 of which represent the rotation rate of each motor. The sign of the output represents the direction of rotation. The last output represents the activation criterion of the electromagnet. Since the electromagnet operates using digital logic, a threshold is set on the robot controller to activate if the value is sufficiently high. The action output and current state are processed by an environment step function which rotates the linkages by a step proportional to the action value. The reward observed and the transition state are returned to the reinforcement learning algorithm in order to store them in the experience buffer.

The simulator continuously monitors the state of the robotic arm. In order to simulate picking up the box, the distance between the position of the third link $\vec{p}_3 = (x_3, y_3, z_3)$ and the top of the box $\vec{p}_\diamond = (x_\diamond, y_\diamond, z_\diamond)$ is checked every step. If the distance is sufficiently small and the output value of the action corresponding to the magnet is greater than 0, the top of the box will be stuck to the robot's effector in the simulation. The box will only remain stuck if the magnet is activated for any subsequent steps. If the condition is not met, the box will begin free-falling until it reaches the ground.

V. Evaluation

This section presents an empirical study to demonstrate the potential of the proposed policy learning platform based on the DDPG architecture. Section V-A presents the setup of experiments. Section V-B and V-C respectively discuss the efficiency of learning and the quality of policies.

A. Experiment Setup

We impose constraints on our platform as a collection of inequalities. The more loosely bound these constraints are, the more adaptable the robot has to be which will cause steps further in the framework to be even more difficult. The objective of the experiment is to move a box from a start position to a goal position. A constraint in our experiment is the scale of the problem. We impose the following constraints: The mass of the box is no more than 120 grams. The side length of the box is 80mm. The start position $\vec{p}_\circ = (x_\circ, y_\circ, z_\circ)$ satisfies:

$$200\text{mm} \leq x_\circ < 400\text{mm} \tag{1}$$
$$y_\circ = 0\text{mm} \tag{2}$$
$$200\text{mm} \leq z_\circ < 400\text{mm} \tag{3}$$

The goal position $\vec{p}_\bullet = (x_\bullet, y_\bullet, z_\bullet)$ satisfies:

$$200\text{mm} \leq x_\bullet < 400\text{mm} \tag{4}$$
$$0\text{mm} \leq y_\bullet < 150\text{mm} \tag{5}$$
$$200\text{mm} \leq z_\bullet < 400\text{mm} \tag{6}$$

Reward functions have to be shaped specifically to guide the policy optimisation process [8]. We design a composite shaped-reward function [9] which returns an increasing reward to the robotic arm. Each component is weighted differently based on multiple trials of the simulation. The full task is subdivided into several components. The reward function of state S is:

$$R(S) = -\sum_{i=1}^{4} w_i r_i(S) \tag{7}$$

where

$$r_1(S) = \mathbf{d}(\vec{p}_3, \vec{p}_\diamond) \tag{8}$$

$$r_2(S) = \frac{\pi}{2} - \arccos \frac{\sqrt{(x_2 - x_3)^2 + (y_2 - y_3)^2}}{\mathbf{d}(\vec{p}_2, \vec{p}_3)} \tag{9}$$

$$r_3(S) = \begin{cases} 0 & \text{if } \mathbf{d}(\vec{p}_3, \vec{p}_\diamond) < \epsilon \\ \max(a_{\text{mag}}, 0) & \text{otherwise} \end{cases} \tag{10}$$

$$r_4(S) = \mathbf{d}(\vec{p}_\diamond, \vec{p}_\bullet) \tag{11}$$

where $w_1 \dots w_4$ are weight coefficients for the four reward components; $\mathbf{d}(\cdot, \cdot)$ is the Euclidean distance between a pair of vectors; $a_{\text{mag}} \in [-1, 1]$ is the control signal for the electromagnet; $\vec{p}_\bullet$ is the goal position.

An interpretation of the four components in Eq. 8–11 is as follows: $r_1(S)$ is the Euclidean distance between the end effector and box; $r_2(S)$ uses the tip positions of the second link and the end effector to find the angle with respect to the ground plane; $r_3(S)$ discourages the robotic arm from expending energy by switching on the electromagnet unless it is attempting to pick the target object; $r_4(S)$ represents the Euclidean distance between the box and the goal which also serves as the termination condition.

In the experiments on the learning efficiency, all timings are taken based on a NumPy implementation of the DDPG using Python 3. The 8-thread software runs on a workstation with an Intel Core i7-6700 CPU (14nm, 4 cores, 3.4 GHz). The proposed hardware architecture is implemented on the Maxeler MAX4 platform with an Intel Stratix-V FPGA (28nm, 200 MHz). The host computer for the FPGA uses an Intel Xeon E5-2640 CPU (32nm, 6 cores, 2.5 GHz). We customise the hardware proposed in Section III with respect to three DDPG models as follows:

1) FPGA-based DDPG model. Each network has a single hidden layer with 108 ReLu nodes. The output layer of the actor networks consists of 5 TanH nodes while the output layer of the critic network consists of a linear node. The batch size of the inputs is 32. This set of hyper-parameters are tuned and found to work well for the environment while maintaining a relatively fast learning speed.
2) FPGA-based DDPG model with expanded action space. While designing the hardware implementation of the DDPG, it is found that the output layers require a minimum of 24 nodes in order for the correct results to be received. This implies that, for environments with relatively simple action spaces (< 24), the calculations involving padded zeros do not affect the policy learnt in any way. Thus, the timings of this hypothetical model with an expanded action space is also taken as a benchmark against our FPGA-DDPG implementation.
3) Deeper model. While training models using the standard DDPG, it is found that increasing the number of layers can lead to a significant increase in training time while also increasing the susceptibility to overfitting in our robotic arm environment. Moreover, we find that a single layer of hidden nodes works best. However, more complicated reinforcement learning problems tend to require deeper neural networks in order to learn higher order features. In order to accommodate this, we expand the second model with an additional hidden layer with 108 nodes for each network.

On the FPGA platform, all multiplications are done using 32-bit fixed-point numbers with 8 integer bits and 24 fractional bits. Fixed-point computations are much faster than floating-point computations and therefore allow a higher pipelining factor. It is possible to tune individual layers to have the optimal number of integer bits given the environment. However, we use 8 integer bits as we assume that we do not know the expected magnitudes of the parameters throughout the training process.

A straightforward method to maximise speed is to compute the critic gradients to replicate the same forward propagation structure from the actor learning process and use a second stream for the target parameters. However, this is practically impossible given the limited resources of our FPGA platform. As a compromise between speed and resources, we reuse the resources for the forward propagation in the critic network. Once the forward pass of the back-propagation algorithm for the actor network is completed, the fast memory (FMEM) implemented with BRAM begins streaming the parameters of the target network to the layers. The matrix multiplication PEs begin computing the target network outputs. For the backward pass of the critic network, the loss layer streams the feedback values to a separate network which also computes the actual values for the temporal difference. This modification allows two sets of actor-critic neural networks to be compressed into a single design while only using approximately 1.5 times the resources of a single set.

The resource usage of the models is shown in Table I and II. Note for model 3, we only build the actor learning network in order to have a higher chance of success in the fitting stage of compilation. Due to resource limitations, we employ two FPGAs, one each for the actor and critic learning process. Since the learning processes are independent of each other, we run both designs concurrently.

TABLE I
RESOURCE USAGE FOR MODEL 1 AND 2

Resource	Utilisation	Available	Percentage
Logic utilisation	143569	262400	54.71%
- Primary FFs	304025	524800	57.93%
- Secondary FFs	14221	524800	2.71%
Multipliers (18x18)	3769	3926	96.00%
DSP blocks	1913	1963	97.45%
Block memory (M20k)	1913	2567	74.52%

B. Learning efficiency

Table III shows the efficiency of gradient computation and transmission using the FPGA-based DDPG designs. Normalised timings are based on the 200% lithography differences between the Core i7-6700 CPU and the Stratix-V FPGA.

TABLE II
RESOURCE USAGE FOR MODEL 3

Resource	Utilisation	Available	Percentage
Logic utilisation	129013	262400	49.17%
- Primary FFs	267999	524800	51.07%
- Secondary FFs	12124	524800	2.31%
Multipliers (18x18)	3749	3926	95.26%
DSP blocks	1896	1963	96.59%
Block memory (M20k)	1930	2567	75.19%

Although each episode takes less than one second to finish, a complete training task may need to go through thousands or millions of episodes depending on the complexity of the system dynamics. The execution time column presents the execution time for one episode of training. The speedup columns record the speedup of the FPGA-DDPG designs against the NumPy-based software on CPU. Theoretically, an Infiniband connection can transmit data faster than a PCIe 2.0 x4 connection. However, the Linux driver of the Infiniband connection in our system limits the speed.

TABLE III
COMPUTATION AND TRANSMISSION OF GRADIENTS

M	Connection	FPGA exe. time (ms)	Speedup	Norm. speedup
1	PCIe	0.250	2.57	5.14
2	PCIe	0.250	3.68	7.36
3	Infiniband	2.311	1.33	2.66

The results suggest that if we stream the gradients and parameters back and forth between the CPU and FPGA, the amount of acceleration is low due to the IO bottleneck. This shows that it is highly beneficial to keep the entire training process within the FPGA while minimising transmissions between the CPU host and the FPGA. For larger models, a group of FPGAs will have to be used which will result in communication overhead between FPGAs. However, the bandwidth between FPGAs is much larger compared to Infiniband.

We now compare the total amount of time used to complete 1300 training episodes each involving the computation of 300 gradients. This comparison covers a Python NumPy implementation and a C implementation of the simulator described in IV-B that calls the FPGA kernel function. Note that timings may vary as some episodes terminate early because they achieve the goal state. An exact comparison is difficult as the algorithms progress differently even with the same random seed because the original DDPG uses floating-point computations while the FPGA version uses fixed-point computations.

TABLE IV
POLICY LEARNING WITH GRADIENT TRANSMISSION

Model	NumPy (s)	C+FPGA (s)	Speedup	Norm. speedup
1	361.2	88.6	4.07	8.14
2	401.2	88.6	4.53	9.06

Table IV shows the results on the full policy learning procedure. Similar to the results on gradient computation, the timing values are heavily inflated due to the high reliance on the bandwidth between the CPU and FPGA. In order to more accurately assess the speed improvement of the hardware implementation, we implement a version of the algorithm that streams only once from the CPU to the DRAM of the acceleration card during initialisation. The gradients are stored in the DRAM that can be extracted to the CPU via Infiniband.

Isolating the data transmission and computation allows us to measure the speedup granted by the computation kernel. Model 2 is used for this experiment. The results for gradient computation are presented in Table V.

TABLE V
GRADIENT COMPUTATION WITHOUT TRANSMISSION

Model	FPGA exe. time (ms)	Speedup	Norm. Speedup
1	0.0492	13.1	26.2
2	0.0492	18.7	37.3

As the IO bottleneck caused by the PCIe interface results in significant stalling of the FPGA, we calculate the full potential of the proposed hardware architecture by estimating the theoretical maximum acceleration, as shown in Table VI. The table shows the acceleration if the entire DDPG including an optimiser is fully implemented on the FPGA at 200 MHz. The parameters of the neural network only needs to be streamed in once. However, an optimiser is not implemented on the FPGA in this study. Thus parameters had to be streamed back out in order to perform the updates on the host CPU and subsequently streamed back into the FPGA for the next iteration.

TABLE VI
THEORETICAL ACCELERATION FOR POLICY LEARNING

Model	Cycles	Time (ms)	Speedup	Norm. speedup
1	5450	0.028	23	47
2	5450	0.028	33	67
3	17000	0.085	35	71

The gradient computation takes up a significant portion of the training process of the DDPG. This is further exacerbated for larger networks. The second choice for acceleration would be the network update algorithm which takes approximately half of the time required for gradient computation. The other steps do not scale significantly with the network size. However, a pure FPGA implementation would significantly reduce any IO bottleneck between the CPU and FPGA.

If the simulation kernel is too complex to effectively implement on the FPGA, action and state vectors can be streamed across the PCIe/Infiniband during training time as a compromise. This paper focuses our efforts on the gradient kernel, the most complex part of the DDPG. Since the update kernel is not the focus of this project, the gradients are streamed back to the CPU host to perform the updates using software. This results in a significant IO bottleneck which masks the true benefit of our hardware implementation.

C. Policy quality

We transfer the simulation policies produced by Model 1 in Section V-A to our custom robotic arm to evaluate their quality. In order to assess the viability of the policy transfer from the simulated environment to the real world, a series of experiments are repeated on the real robot. Once training on the simulation is complete, the actor network parameters are saved locally. Parameters from different episodes of the training are saved, starting from 400 episodes and every 200 episodes thereafter up to 1200.

Six tasks are evaluated in this experiment. The common start position of Task A and B is $(400, 0)$, and the goal positions are $(400, 150)$ and $(200, 0)$. The start position for Task C and D is $(300, 0)$, and the goal positions are $(400, 150)$ and $(200, 150)$. The start position for Task E and F is $(200, 0)$, and the goal positions are $(400, 150)$ and $(400, 0)$. Each task contains two sub-tasks:

1) An 'attach' sub-task where the electromagnet attaches the box. An 'attach' sub-task is considered completed if the head of the electromagnet touches the box and current flowing through the electromagnet is above zero.
2) A 'put' sub-task where the robotic arm moves the box to the goal position. A 'put' sub-task is considered completed if the robotic arm puts the box in a position within 30mm of the goal position.

TABLE VII
NUMBER OF EPISODES TO ACHIEVE GOAL

	TRPO [1] Attach	DDPG Attach	DDPG Put	F-DDPG Attach	F-DDPG Put
A	×	600	×	800	×
B	Yes	800	**800***	800	1000*
C	×	600	1200	600	**600**
D	×	800	1200	**600**	**1000**
E	×	600	1000	600	1000
F	Yes	800	**800***	**600**	1000*

* The robotic arm reaches the goal position without lifting the box

Table VII shows the results of policy learning with the number of training episodes. Smaller numbers correspond to faster convergence of policy search. The standard DDPG method (DDPG), the FPGA-based DDPG method (F-DDPG), and the TRPO in [1] are compared. A boldface number shows a result to be significantly superior than those produced by the other two methods for the same sub-task. As we are unable to replicate the exact configuration in [1], we provide the most optimistic estimation. In other words, we assume that the end effector of the TRPO-based robotic arm can approach any position on the straight line on which the centre of the robotic arm projects. The TRPO-based robotic arm does not put the object back on the ground. Therefore, we only show the results for the 'attach' tasks for TRPO. Also, the number of episodes to achieve the goal for TRPO is incomparable with DDPG. Therefore, we only show whether TRPO is able to finish each task.

The FPGA-based DDPG achieves comparable quality of learning with the original DDPG method. Also, the learning environments based on the FPGA-based DDPG and the original DDPG can produce policies to finish task C, D and E which the TRPO based robotic arm in [1] cannot finish. A curious observation is that the FPGA-based DDPG is able to reach the peak of its success rate more quickly compared to the standard DDPG. This is probably because the loss in numeric precision counteracts overfitting, which improves the quality of exploration, and enhances the robustness of optimisation.

VI. Conclusions and Future Work

This paper presents a reinforcement learning platform based on a customised hardware architecture for the Deep Deterministic Policy Gradient (DDPG) algorithm on FPGAs. The platform includes a physical robotic arm and its simulator in the virtual space. The FPGA based DDPG implementation learns policies using the simulated robotic arm. Even with an IO-bottleneck between the CPU host and FPGA, we are still able to achieve substantial acceleration over a CPU host without compromising policy quality. The policies encoded in fixed-point numbers successfully controls the physical arm to move objects in a three-dimensional space.

One direction of future work is to explore policy learning strategies with data sampled from low-precision simulation. Another direction is to investigate how the proposed learning framework performs as the degrees of freedom increase.

VII. Acknowledgements

The support of the United Kingdom EPSRC (grant numbers EP/L016796/1, EP/N031768/1, EP/P010040/1 and EP/L00058X/1), Corerain, Maxeler and Intel is gratefully acknowledged.

References

[1] S. Shao, J. Tsai, M. Mysior, W. Luk, T. Chau, A. Warren, and B. Jeppesen, "Towards hardware accelerated reinforcement learning for application-specific robotic control," in *International Conference on Application-specific Systems, Architectures and Processors*, pp. 1–8, IEEE, 2018.

[2] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[3] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[4] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, p. 529, 2015.

[5] J. Su, J. Liu, D. B. Thomas, and P. Y. Cheung, "Neural network based reinforcement learning acceleration on FPGA platforms," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 4, pp. 68–73, 2017.

[6] S. Shao and W. Luk, "Customised pearlmutter propagation: A hardware architecture for trust region policy optimisation," in *International Conference on Field Programmable Logic and Applications*, pp. 1–6, IEEE, 2017.

[7] Zortrax Inc, "Robotic arm assembly manual." https://zortrax.com/downloads/ROBOTIC_MANUAL.pdf, 2019. [Online; accessed 2-Apr-2019].

[8] A. Y. Ng, D. Harada, and S. Russell, "Policy invariance under reward transformations: Theory and application to reward shaping," in *ICML*, vol. 99, pp. 278–287, 1999.

[9] I. Popov, N. Heess, T. Lillicrap, R. Hafner, G. Barth-Maron, M. Vecerik, T. Lampe, Y. Tassa, T. Erez, and M. Riedmiller, "Data-efficient deep reinforcement learning for dexterous manipulation," *arXiv preprint arXiv:1704.03073*, 2017.