E2HRL: An Energy-Efficient Hardware Accelerator for Hierarchical Deep Reinforcement Learning

AIDIN SHIRI, University of Maryland Baltimore County, USA
UTTEJ KALLAKURI, University of Maryland Baltimore County, USA
HASIB-AL RASHID, University of Maryland Baltimore County, USA
BHARAT PRAKASH, University of Maryland Baltimore County, USA
NICHOLAS R. WAYTOWICH, Army Research Laboratory, USA
TIM OATES, University of Maryland Baltimore County, USA
TINOOSH MOHSENIN, University of Maryland Baltimore County, USA

Recently, Reinforcement Learning (RL) has shown great performance in solving sequential decision-making and control in dynamic environment problems. Despite its achievements, deploying Deep Neural Network (DNN) based RL is expensive in terms of time and power due to the large number of episodes required to train agents with high dimensional image representations. Additionally, at the interference the large energy footprint of deep neural networks can be a major drawback. Embedded edge devices as the main platform for deploying RL applications, are intrinsically resource-constrained and deploying deep neural network based RL on them is a challenging task. As a result, reducing the number of actions taken by the RL agent to learn desired policy, along with the energy-efficient deployment of RL is crucial. In this paper, we propose Energy Efficient Hierarchical Reinforcement Learning (E2HRL), which is a scalable hardware architecture for RL applications. E2HRL utilizes a cross-layer design methodology for achieving better energy efficiency, smaller model size, higher accuracy, and system integration at the software and hardware layers. Our proposed model for RL agent is designed based on the learning hierarchical policies, which makes the network architecture more efficient for implementation on mobile devices. We evaluated our model in three different RL environment with different level of complexity. Simulation results with our analysis illustrate that hierarchical policy learning with several levels of control improves RL agents training efficiency and the agent learns the desired policy faster compared to a none hierarchical model. This improvement is specifically more observable as the environment or the task becomes more complex with multiple objective subgoals. We tested our model with different hyperparameters to achieve the maximum reward by the RL agent while minimizing the model size, parameters, and required number of operations. E2HRL model enables efficient deployment of RL agent on a resource constraint embedded devices with the proposed custom hardware architecture which is scalable and fully parameterized with respect to the number of input channels, filter size, and depth. The number of processing engines (PE) in the proposed hardware can vary between 1 to 8, which provides the flexibility of trade-off different factors such as latency, throughput, power and energy efficiency. By performing a systematic hardware parameter analysis and design space exploration, we implemented the most energy-efficient hardware architectures of E2HRL on Xilinx Artix-7 FPGA and NVIDIA Jetson TX2. Comparing the implementation results shows Jetson TX2 boards achieve 0.1 ~ 1.3 GOP/S/W energy efficiency while Artix-7 FPGA achieves 1.1 ~ 11.4 GOP/S/W, which denotes 8.8X ~ 11X better energy efficiency of E2HRL when model is implemented on FPGA. Additionally, compared to similar works our design shows better performance and energy efficiency.

Additional Key Words and Phrases: Reinforcement Learning, CNN, Energy Efficient Hardware Accelerator, FPGA, CPU

Authors' addresses: Aidin Shiri, University of Maryland Baltimore County, USA; Uttej Kallakuri, University of Maryland Baltimore County, USA; Hasib-Al Rashid, University of Maryland Baltimore County, USA; Bharat Prakash, University of Maryland Baltimore County, USA; Nicholas R. Waytowich, Army Research Laboratory, USA; Tim Oates, University of Maryland Baltimore County, USA; Tinoosh Mohsenin, University of Maryland Baltimore County, USA.

1084-4309/2022/2-ART $15.00
https://doi.org/10.1145/3498327

1 INTRODUCTION

Reinforcement Learning (RL) is a goal-oriented paradigm of machine learning in which the agent tries to learn a policy to achieve complex tasks by trial and error. Reinforcement Learning is used for problems that involve sequential-decision making where the agent needs to take actions in an environment to maximize cumulative future rewards. In Reinforcement Learning, goals are specified using a reward function [36]. Human feedback can be used to specify goals as well as shown in [9]. Applications such as playing video games by receiving only pixelated image inputs, Robotics and robotic arm manipulation, and self-driving vehicles like autonomous cars and drone navigation system are areas that Reinforcement Learning has shown great achievements. Despite demonstrating great performance in tasks that previously were hard to accomplish by conventional machine learning methods in recent years, the previous papers in Reinforcement Learning focused on the algorithm and theoretical aspects, and very few works have considered efficient hardware design for RL [21].

Learning the policy for the RL agent can be a complex and time-consuming task. It is important to train the agent in the fastest and most efficient way, especially when the agents act in a real-world environment where performing actions could be expensive. One scalable way to do this is by breaking down the policy that RL agent needs to learn into multiple levels of hierarchy. In this method desired policy is divided into individual sub-policies. Two neural networks are trained for performing the desired task, one for selecting the proper subgoal, and the other for producing the proper action to perform in the environment. Besides making it easy to specify goals and rewards, hierarchical subgoals can additionally be reused in other similar environments that might have the same subgoals.

Standard RL policies struggle with solving long horizon tasks with sparse rewards. Complex tasks can be broken down into smaller sub-tasks and we can make use of this information to build agents with multiple levels of control. We use this notion to build a reinforcement learning agent architecture and experiment with different parts of the network architecture for efficient deployment of RL agents on the embedded devices. The novelty of this paper is proposing an efficient hardware architecture for reinforcement learning based on hierarchical policy learning. So far, most of the previous works have focused on only hardware or software aspect of reinforcement learning.

In this paper, we propose E2HRL, a hardware-friendly architecture for Reinforcement Learning, which uses hierarchical policy learning. By employing cross stack design methodology, we have designed a scalable and parameterized hardware for efficient deployment of RL on the embedded devices. We performed a Neural Network hyperparameter analysis to find the best trade-off between accuracy and model size and a scalable low power hardware is designed for the optimized model and is implemented both on FPGA and ARM CPU.This paper makes the following contributions:

- Propose a hardware-friendly architecture for RL which works based on hierarchical policy learning.
- Perform a Neural Network hyperparameter analysis to find the best trade-off between accuracy and model size.
- Design a scalable and parameterized hardware for efficient deployment of RL on the embedded devices.
- Implementation of the E2HRL model with different levels of parallelism on Artix7 FPGA and compare them in terms of power consumption, energy efficiency, latency, and utilized resources.

The rest of this paper is organized as follows: Related prior work is provided in Section II. Section III explains the background of deep Reinforcement Learning and Hierarchical policy learning, along with the environment setup for our experiments. Section IV describes the proposed E2HRL system architecture. Section V talks about neural network optimization and software experimental results and analysis for energy efficiency. Section VI describes the E2HRL hardware architecture design and implementation results and comparison with similar works. Finally, we conclude in Section VII.

2 RELATED WORKS

Despite past efforts and early success on Reinforcement Learning, most of the initial works in the RL domain are limited to simple tasks such as training a specific robot [17], automatic inverted flight control [23], and optimizing dialog policy [34]. These works do not take optimizing the power and performance trade-offs into account. The advent of deep learning has caused significant progress in RL like other machine learning areas, which resulted in the feasibility of creating powerful autonomous agents that can interact with the environment and learn to perform complex tasks over time with trial and error. In recent years, several works that have used RL algorithms have been proposed. Recently, Google Deep Mind announced that the AlphaGo Zero beat the previous champion-defeating AlphaGo 100-0 using an algorithm based only on RL, with no human data, guidance, or any domain knowledge beyond the game rules [33]. Authors in [22] have achieved human-level control in many Atari games with Deep RL. Several works are also proposed which use robotic simulators to train the network with an unlimited amount of data and afterward, transferring the knowledge from the simulator to real-world [26] [38]. Training autonomous robots to navigate to designated locations is another application as well [45]. Recent growth in neural networks and deep learning-based algorithms have been coupled with a significant increase in the model size and networks with a high number of parameters. To address the high computational and storage complexity of neural networks, different computing platforms have been engaged for a diverse range of applications. While most of the current research use software solutions based on general-purpose CPU and GPU platforms to address computational issues, specialized hardware accelerators have demonstrated superior performance in terms of energy efficiency and meeting real-time requirements. Domain-specific accelerators can achieve orders of magnitude improvements in performance per watt compared to general-purpose computers for machine learning applications [7]. Despite great interest and innumerable valuable works in design and implementation of hardware accelerators for different Artificial Intelligence and Machine Learning applications such as image classification[14], Recurrent Neural Network Based Language Models [19], automated tools for deep neural networks implementation with resource and performance estimation [28, 29] , stochastic computing neural network accelerators [20] , very few works have considered hardware requirements for RL as well [8, 31, 32]. Authors in [15, 16] proposed low power Deep Reinforcement Learning SoC which supports CNN,RNN, and FC layers. FPGA accelerators for neural network based RL has drawn more attention in recent year due to their flexibility and better performance and energy efficiency compared to other heterogeneous platforms[5, 6, 30, 35, 37, 40]. Despite ll of the valuable past efforts, still a comprehensive approach for efficient deployment of RL application on hardware design is yet to be proposed. In summary, most of the previous works in the RL domain are limited to the software implementations [24], or on general-purpose GPU at best-case scenarios. Embedded GPUs are among the best devices to handle precise floating-point operations that are able to perform typically at Tera floating-point operations per second (TFLOPS) given a power budget of a few watts (e.g. 1.3 TFLOPS at 7.5 watts for the NVIDIA Jetson TX2 [1]). However, their energy consumption per multiplication or addition operations is still high (0.17 TOP/J) as a result of handling expensive floating-point operations. Adopting high precision operations is mandatory for training machine learning algorithms, but during their test and inference, cheaper operations such as binary/ternary operations that consume significantly less energy can deliver the same functionality.

3 BACKGROUND

In this section, we present a brief background of reinforcement learning and hierarchical Reinforcement Learning. Then we describe the environmental setup and how the tasks can be solved using hierarchical Reinforcement Learning.

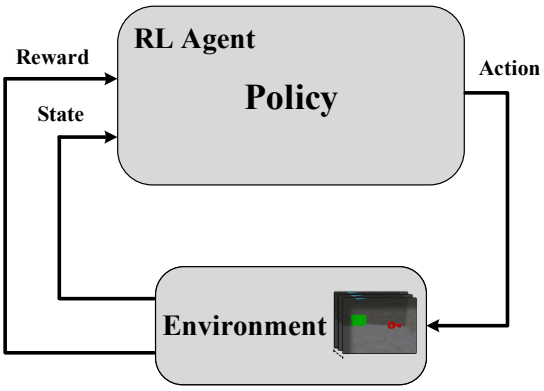


Fig. 1. Traditional Reinforcement Learning agent block diagram. The agent interacts with the environment by executing an action based on its policy. The environment then provides a new state and a scalar reward to the agent

3.1 Deep Reinforcement Learning

Reinforcement Learning agents can learn to solve complex sequential decision making tasks by interacting with the environment as shown in Figure. 1. Typically, we need to define a reward function, state and action spaces and the actual learning algorithm. Given these ingredients, the agent learns to solve the tasks autonomously without intervention. The agent only gets a reward based on the performance of the tasks to learn the desired policy without specifying how to accomplish the task. Usually,Reinforcement Learning is formalized using Markov Decision Processes (MDP). An MDP is defined as a tuple (S, A, P, R, γ), where S is the state space, A is the action space or the set of actions available to an agent, P is the unknown transition function, R is the reward function and $\gamma \in (0, 1)$ is the discount factor. The RL agent interacts with the environment by acting according to a policy π which is a mapping from states to actions, or a probability distribution over actions. The goal at each step is to maximize the discounted sum of future rewards, $\sum_{t'=t}^{\infty} \gamma^{t'-t} R_{t'}$, and the quality of the policy at time t is measured by the value function $E_{\pi}(\sum_{t}^{\infty} \gamma^{t} R_{t+1} | s_0 = s)$, with initial state s.

3.2 Hierarchical Reinforcement Learning

Reinforcement learning techniques have shown great promise in the past few years, getting us a few steps closer to deploying them in the real world. However, there are still many challenges before we can safely and efficiently deploy them in the real world, especially when these technologies work closely with other humans. One of the issues is low sample efficiency, where we require a large number of resources in terms of computing time and data to train these autonomous agents. One way to tackle this problem is to make use of the hierarchical structure present in complex sparse feedback tasks to improve learning. It is possible to use the hierarchical structure and learn multiple levels of control to solve tasks as shown by [3] and [18]. These methods have been shown to improve the performance in the sparse reward and long-horizon tasks. In this work, we use the hierarchical structure with 2 levels of control and in addition with the assumption that we know the subgoals to solve the task. In the following sections, we explain the environmental setup and the E2HRL architecture.

3.3 Environmental Setup

Our experiments are performed on the MiniWorld environment [4]. It is a minimalistic 3D environment for Reinforcement Learning and robotics research. The agent can navigate rooms and manipulate objects using its

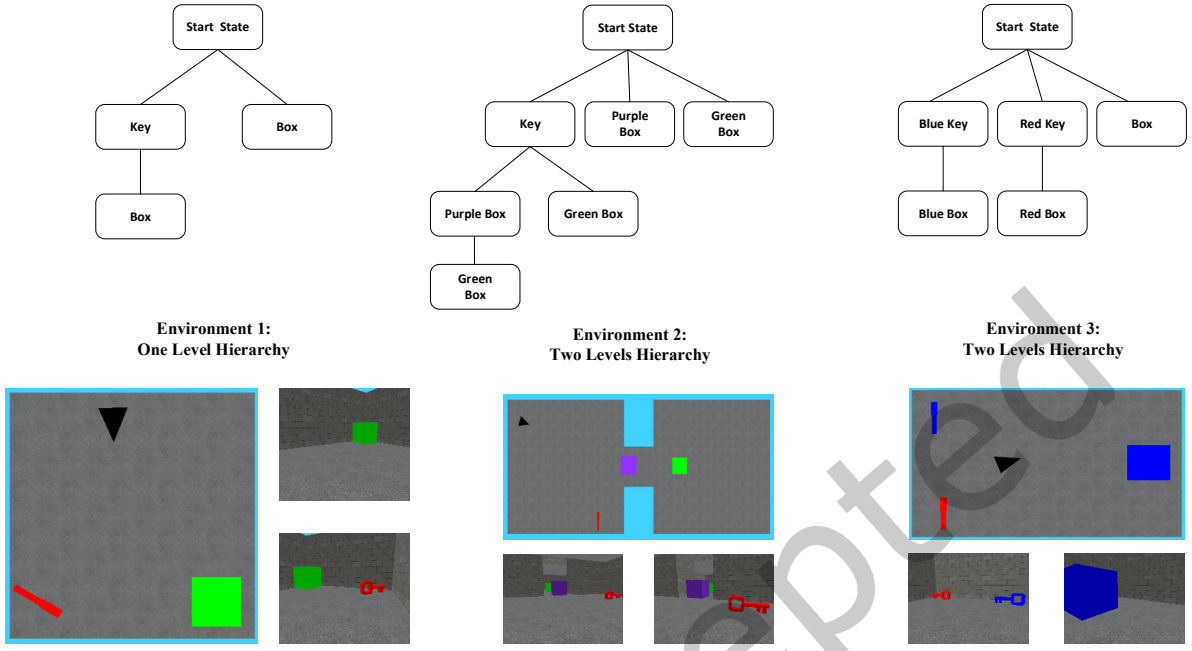


Fig. 2. This figure shows the task graphs for each of these environments in addition to Three Miniworld environments. In Environment 1, the task is to first collect the red key and then reach the green box. In Environment 2, The task is to first collect the red key and then reach the door which can be opened only after collecting the key. And then reach the green box (which represents the final goal location). In Environment 3, The agent has to check the color of the box and then decides to collect the right key and go to the box. The agent receives a reward when it reaches the box, provided it has collected the right key first.

first-person view as shown in figure 2. In our experiments we set up three scenarios with different difficulty levels. We assume that humans defining the task have knowledge of how the tasks can be divided into smaller sub-tasks. Sub-tasks are defined prior to training based on domain knowledge. This is similar to some prior work in the field as seen in [2]. Figure 2 shows more details about how each task is divided into sub-tasks in our experiments.

In environment 1, shown in figure 2 (a), the agent is in a room that has a box and a key. The agent is spawned at random locations in each episode. The task is to first collect the red key and then reach the green box. The agent receives a reward when it reaches the box, provided it has collected the key first. If the agent reaches the box without the key, it receives no reward. As described earlier this task can be decomposed into 2 subgoals: 1. reach the key, 2. reach the box.

In environment 2, shown in figure 2 (b), the agent is in a room that has a key. The box is in the other room but is blocked by a purple box (which represents a door). The task is to first collect the red key and then reach the door which can be opened only after collecting the key. And then reach the green box (which represents the final goal location). The agent receives a reward when it reaches the box, but the only way to reach it is to first collect the key, open the door and then go to the green exit. If the agent cannot complete the task in a fixed number of steps the episode ends and it receives no reward. This task can be decomposed into 3 subgoals as well: 1. reach the red key, 2. open the door, 3. Reach the green exit.

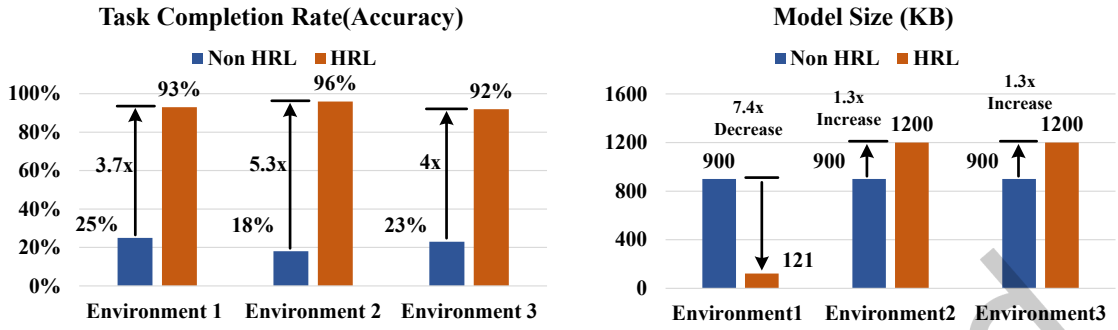


Fig. 3. Comparison between task completion rate and model size of none hierarchical and hierarchical reinforcement learning models. All models are trained for 1 million steps and their performance are measured afterwards.

In environment 3, shown in figure 2 (c), the agent is in a room that has a box, a blue key and a red key. The agent is spawned at random locations in each episode and the box is either red or blue. The agent has to check the color of the box and then decide to collect the right key and go to the box. The agent receives a reward when it reaches the box, provided it has collected the right key first. If the agent reaches the box without the key, it receives no reward. Or, if the agent reaches the box with the wrong colored key, it receives no reward. As described earlier this task can also be decomposed into 2 subgoals: 1. reach the red/blue key, 2. reach the box. But here the agent has 3 subgoals to choose from, 1. reach the red key, 2. reach the blue key, 3. reach the box.

4 PROPOSED SYSTEM ARCHITECTURE

In this section, we present an overview of the proposed E2HRL architecture and explain the agent training in detail. E2HRL is designed to perform efficiently as an agent in Reinforcement Learning environments. figure 5 shows the proposed network architecture of the E2HRL, which consists of convolutional, Dense, LSTM and softmax layers. We compared standard conventional reinforcement learning baselines with the hierarchical reinforcement learning architecture in terms of accuracy and memory size as shown in Figure 3. We train both with the same budget in terms of environment steps. As seen in Figure 3, the proposed HRL achieves 7.4X less memory for Environment 1. For the other two environments, HRL utilizes slightly higher memory, 1.3X, when compared to the baseline. However the accuracy in terms of final task completion percentage is consistently better for all test cases, 3.7X ~ 5.3X. Significant improvement in the task completion rate justifies the slightly higher memory requirement for HRL.

4.1 E2HRL Architecture

The agent consists of 2 modules – the Subgoal module and the Action module. Both these modules get the image embedding from the convolutional layers which are used to process the image observation. The Subgoal module is responsible for producing the subgoal given the current observation. The action module takes the current observation and the subgoal produced by the subgoal module to output the action which is executed in the environment. The architecture is shown in figure 4 where the agent is trained in 2 phases. Both policies are trained using proximal policy optimization [27]. The high-level policy does not need to output a goal at every environment time step. It outputs a subgoal and sleeps for several time steps (K), which is then used by π_C to

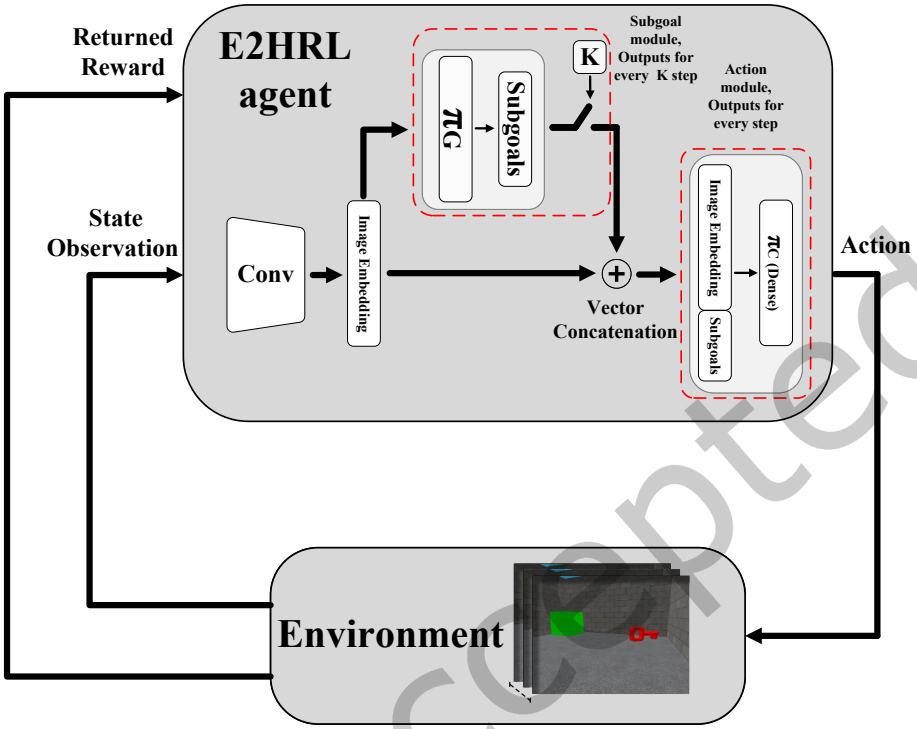


Fig. 4. Block diagram of the proposed architecture of the hierarchical RL agent. Here the policy π of the agent consists of 2 main modules – the subgoal module π_G and the action module π_C. The image observation is first processed using a convolution network to get an image embedding. The subgoal module is responsible for providing a subgoal or a high-level action given the current observation. The action module gets the image embedding and the subgoal and calculates the action which is executed in the environment. The subgoal module is activated once every K steps. The detailed architecture of E2HRL agent is provided in the Figure 4 and Figure 9.

output low-level actions. Additionally, this helps to reduce the hardware and time complexity. We experiment with different intervals, K and reported the results in the next section.

4.2 Training the Subgoal and Action Modules

First, the low-level policy (π_C) is trained to perform the subgoals. This is done by concatenating a one-hot encoded task id to the actor network. The agent is trained to perform the subgoals given by these task ids. Once the agent is able to perform these individual subgoals, the action module is frozen and the subgoal modules are trained in the next phase. In the second phase, the high-level policy (π_G) is trained to select subgoals. This output is then concatenated to the low-level policy, which executes a policy to reach the subgoals. As seen in figure 4, the subgoal module consists of an MLP with multiple fully connected layers. The input to these layers is the image embedding from the convolutional image encoder. This architecture is sufficient for the first simple task. But the second and third tasks are more complicated and need some form of memory.

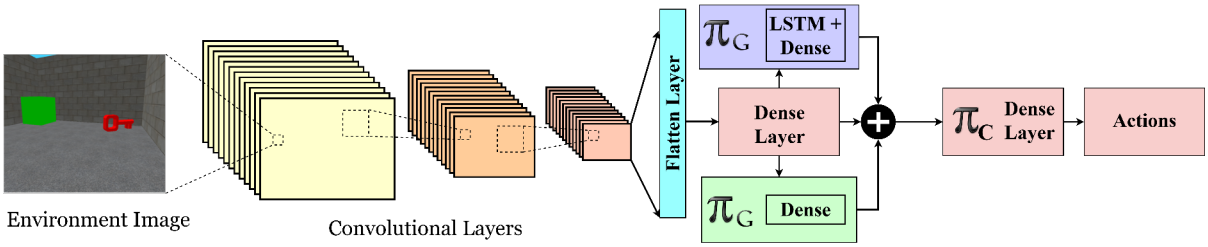


Fig. 5. The proposed network architecture of the E2HRL, as described in figure 4. The subgoal module has 2 configurations. One with fully connected (FC) layers shown in green and one with LSTM shown in purple

Table 1. FC-HRL and LSTM-HRL neural network architecture in details

FC-HRL Layers	LSTM-HRL Layers	Input	Window	# Kernels	Activation
Conv 1	Conv 1	40*30*3	(3,3)	32	ReLU
Conv 2	Conv 2	20*15*32	(3,3)	32	ReLU
Conv 3	Conv 3	10*8*32	(3,3)	32	ReLU
Flatten	Flatten	5*4*32	-	-	-
Dense [πG]	LSTM [πG]	32	-	-	ReLU
Dense [πC]	Dense [πC]	32	-	-	ReLU
Action	Action	3	-	-	Softmax

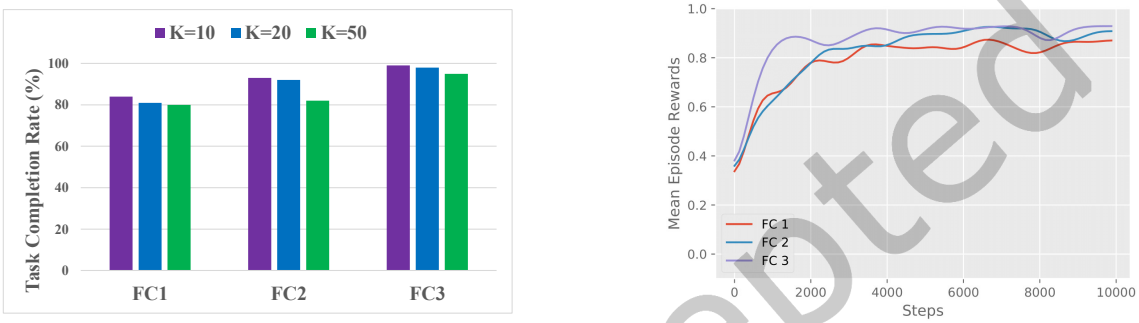
Therefore we add an LSTM layer after we get the image encoder and before passing it to the MLP. Long Short-Term Memory (LSTM), are a particular type of recurrent neural network which have internal contextual state cells that act as long-term or short-term memory cells. This is an important characteristic when the prediction of the neural network depends on the historical context of inputs, rather than only the last input. This modification seems to help solve the tasks faster and is required for the second and the third task.

5 E2HRL NEURAL NETWORK OPTIMIZATION FOR ENERGY EFFICIENCY

Neural network optimization is the process of systematic model architecture engineering to achieve the best possible performance with the smallest model size and least computation. We performed extensive analysis on the model search space and evaluate our model performance. Search space defines the networks that can be examined to produce the final architecture. The initial E2HRL model consists of three convolution layers, a flatten layer, and a dense (fully-connected) layer with an output of 32 for generating an image embedding vector. Next, the embedding vector is fed to the subgoal module which is a dense or LSTM layer to produce the subgoal vector. Finally, subgoal vector and image embedding are concatenated together to fed another dense layer with Softmax activation and generate the desired output for the agent to act in the environment. The initial HRL agent architecture is shown in detail in Table 1.

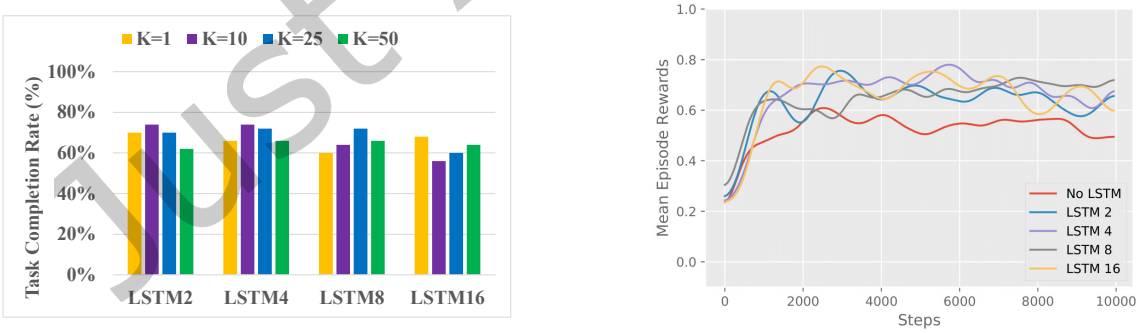
We performed a macro-architecture search for layer type, hyper-parameters, and connections with other layers to achieve the best performance. Furthermore, for each block, we optimize the convolutional layer kernel size and filter number, as well as the dropout rate, the existence of a pooling layer after each convolution layer. A number of optimum architectures that produced the best results for the three environments discussed in the previous section are summarized in Table 2. By comparing the number of parameters, and model size of different configurations, we observed that Config 2 for FC-HRL and LSTM4 for LSTM-HRL achieves almost the same

reward as the best configurations, while having less memory and computation complexity. Therefore, we select Config 2 and LSTM4 as the optimal design for implementing on hardware. Figures 6,7, and 8 illustrate the model size and performance of different configurations with respect to the time step, K, amount of time steps that π_G sleeps. We observe that for a simple environment like environment 1, the model performs better with smaller K. As the complexity of the environment increases, the models perform better when K is larger.



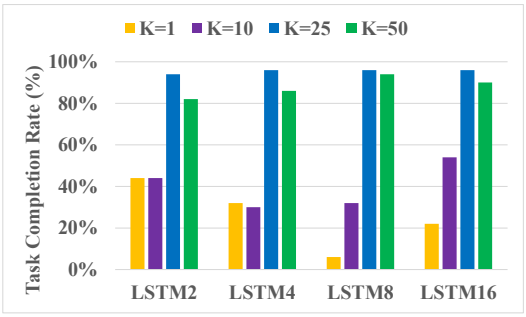
(a) (b)

Fig. 6. Environment 1 (a) Performance and Memory size of hierarchical RL agent for different configurations with respect to K.
(b) Reward achieved by the HRL agent in the Miniworld environment with respect to the number of episodes for different configurations

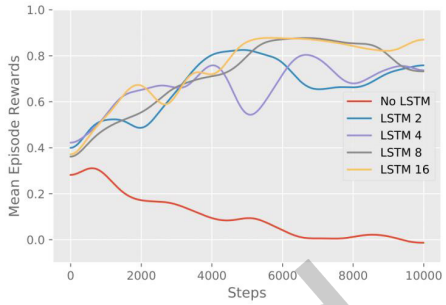


(a) (b)

Fig. 7. Environment 2 (a) Performance and Memory size of hierarchical RL agent for different configurations with respect to K.
(b) Reward achieved by the HRL agent in the Miniworld environment with respect to the number of episodes for different configurations



(a)



(b)

Fig. 8. Environment 3 (a) Performance and Memory size of hierarchical RL agent for different configurations with respect to K.
(b) Reward achieved by the HRL agent in the Miniworld environment with respect to the number of episodes for different configurations

6 E2HRL HARDWARE ARCHITECTURE DESIGN, IMPLEMENTATION AND ANALYSIS

This section describes the hardware architecture design and implementation of E2HRL with commercial off-the-shelf devices. We have implemented our method on the Xilinx Artix-7 FPGA board and Nvidia Jetson TX2 (shown in figure 9) to measure the energy-efficient implementations for embedded devices.

6.1 Hardware Architecture Design

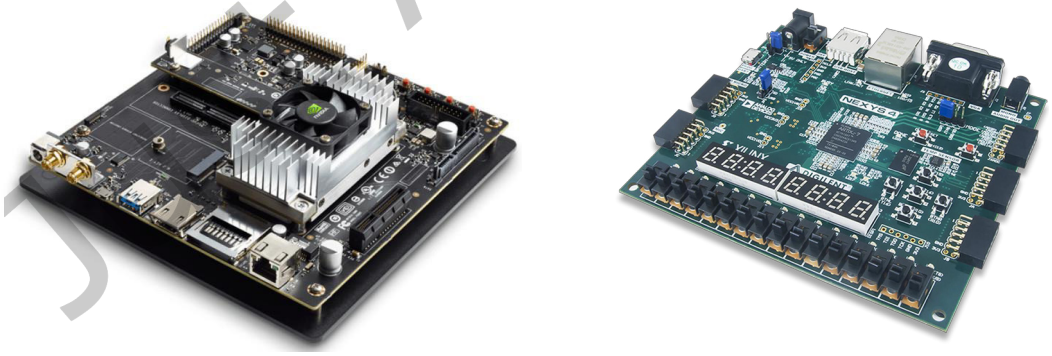
Figure 10 shows the detailed block diagram of the E2HRL hardware architecture. The architecture mainly focuses on important hardware facets such as on-chip memory re-utilization and parallel computations. Another key feature is to fit the hardware on smaller FPGAs with smaller Block RAM sizes. This additionally implies the hardware should occupy minimum area while meeting the latency requirements. The designed hardware is such that it can be easily re-configured and meanwhile portable. The reconfiguration is mainly targeted at implementing newer models, improving the overall performance and improving the parallelizations. This is achieved by increasing the memory and the MAC widths, changing the memory sizes and increasing the number of PEs in the design. In figure 10 the main blocks are the PE array to perform MAC operations in parallel, memory blocks to hold the weight, feature and the intermediate outputs; individual address generation blocks for the convolution and fully connected layers and an independent LSTM block (in the case of LSTM-HRL). The top

Table 2. E2HRL different network configurations with different computation and memory complexity. Each model is trained for one million Steps

	FC-HRL (Environment 1)			LSTM-HRL (Environment 2 & 3)			
	FC1	**FC2**	FC3	LSTM2	**LSTM4**	LSTM8	LSTM16
# Parameters	11.7 K	**30.2 K**	94 K	39 K	**39 K**	39 K	39 K
Memory	45.5 KB	**121 KB**	376 KB	1.2 MB	**1.2 MB**	1.2 MB	1.2 MB
# Computations	2.8 M	**9.3 M**	32.3 M	2.4 M	**2.4 M**	2.4 M	2.4 M
Task Completion	84%	**93%**	98%	94% & 90%	**96% & 92%**	96% & 92%	96% & 80%

control logic generates the necessary control signals based on the input configuration. It generates the controls for the convolution, fully connected and the LSTM address generation blocks as well. The generated addresses are passed on to the available on-chip memories to retrieve the corresponding weight and feature values to be passed into the PE array. The design of the PE has been done in such a way that it can perform one MAC operation per clock cycle. The MAC values outputted from the PE arrays is stored in the feature memory as input to the layer that follows after the current computation. The top control logic also accepts a hyper-parameter K. By maintaining an internal k-counter and comparing the two values it keeps track of when the FC/LSTM branch has to be run. The output of this branch is written in the last section of the feature map which allows saving the previous FC/LSTM output until it is re-run. The architecture for the LSTM block is presented in the figure 10. The main blocks in the designed hardware include:

- Convolution: The convolution block allows to perform the 2D convolution operations through repeated MAC operations. The control unit generates the necessary control signals to enable the blocks. The convolution address generator generates the addresses according to the tiling scheme explained in Fig.12(c), where a single input channel is selected and convoluted with multiple filters parallelly. Once the MACs have the N parallel data values from the Weight and the data memories, it performs the MACs in parallel and stores the results sequentially in the Image embedding memory.
- Fully Connected: The operations performed are similar to that of Convolution. The top control logic generates the necessary control signals, the address generator passes the addresses to the memories, which are fetched into the PE-array. MAC results are stored back into the appropriate memory depending on the hyper-parameter values.
- LSTM: This is the module on the branch that is executed depending on the hyper-parameter *k*. Simplifying equations 1 - 4, we can observe that it aims at implementing matrix vector multiplication. Therefore, by the concatenation of the kernel and the recurrent kernel weights into two memories the memory footprint can be minimized. From the Fig.10, we can notice the lstm module is composed of two MAC modules and memories. The memories hold the kernel, recurrent kernel weights, biases and cell-state data respectively. Additionally, the module also preforms *Hard Tanh* and *Hard Sigmoid* activations.



Nvidia Jetson TX2 Xilinx Artix-7

Fig. 9. Common Off-the-Shelf edge platforms

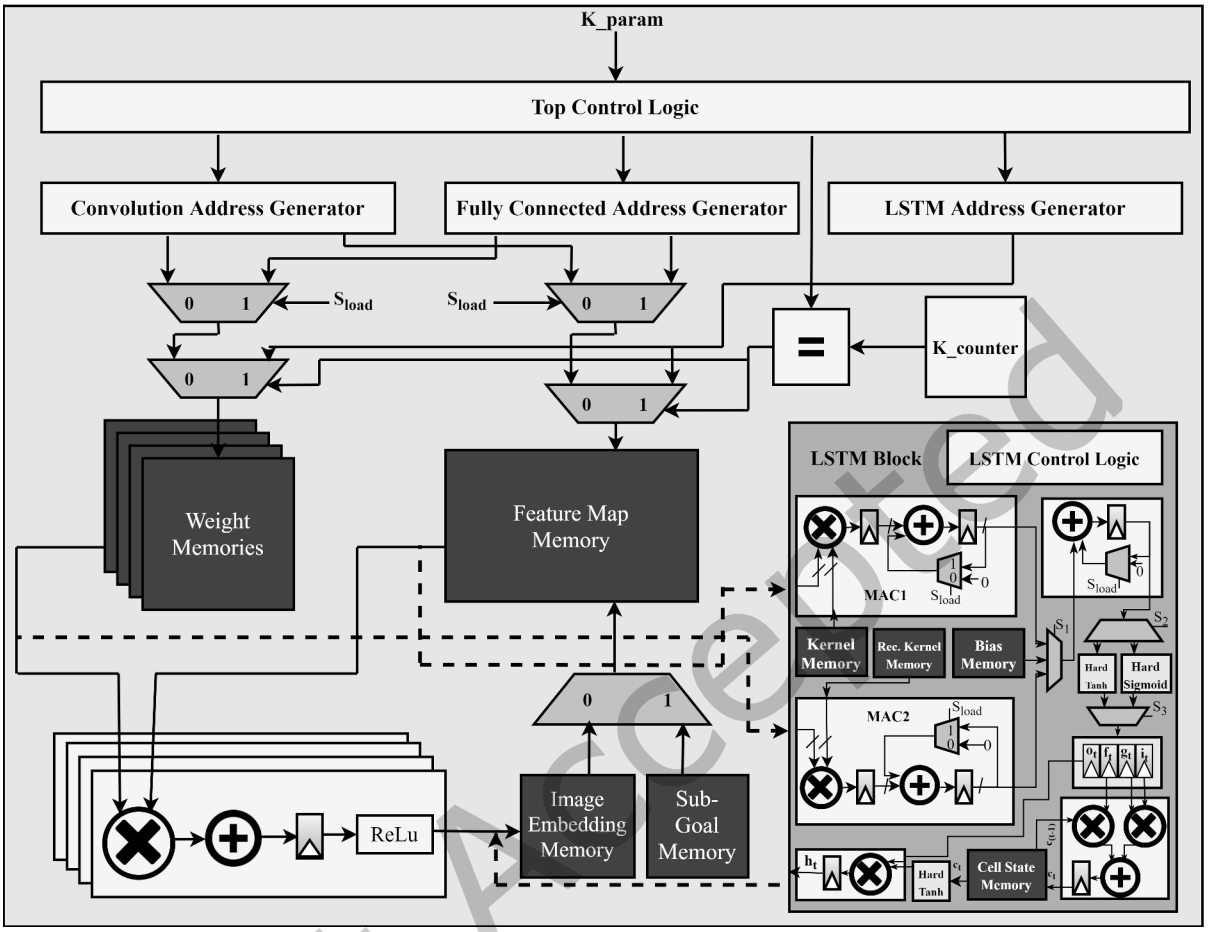


Fig. 10. Block diagram of the proposed E2HRL hardware architecture which consists of Convolution, LSTM and Fully Connected layers. As shown in 5, for K-1 number of operations the design runs through a sequence of Convolutions and Fully Connected layers, for the K'th round the design utilizes an additional Fully Connected layer (in case of FC-HRL) or the LSTM layer (in case of LSTM-HRL)

- Memory: The memory modules on the hardware is mainly divided among the weight, feature, image embedding and the subgoal memories. For both the designs, i.e., FC-HRL and the LSTM-HRL modules, the input sizes are 40x30x3. The Weight Memory is used to hold all the weights needed for the convolution, fully connected and the LSTM layers. The Feature Map memory is designed to keep a maximum intermediate features (i.e., the largest feature map in the convolution layer). The LSTM layer additionally, also has dedicated memories to hold the kernel values, recurrent kernel vales, the bias vales and the cell state. Contents from these memories are read into the MAC modules in accordance with the control signals received from the LSTM control logic. Additionally, the top Weight Memory is halved and replicated each time the number of PEs doubles, this allows for convenient data sharing for the tiling scheme.

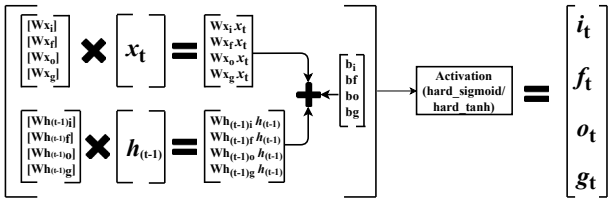


Fig. 11. Concatenation of equation 1-4 for efficient hardware implementation

According to [11], at a particular time step t, the computation inside the LSTM unit is done as per the following equations:

$$i_t = sigmoid(W_{x_i} x_t + W_{h_i} h_{t-1} + b_i) \tag{1}$$

$$f_t = sigmoid(W_{x_f} x_t + W_{h_f} h_{t-1} + b_f) \tag{2}$$

$$o_t = sigmoid(W_{x_o} x_t + W_{h_o} h_{t-1} + b_o) \tag{3}$$

$$g_t = tanh(W_{x_g} x_t + W_{h_g} h_{t-1} + b_g) \tag{4}$$

$$c_t = f_t \bigodot c_{t-1} + i_t \bigodot g_t \tag{5}$$

$$h_t = tanh(c_t) \bigodot o_t \tag{6}$$

where $\bigodot$ is the element-wise multiplication. W_{x_i}, W_{x_f}, W_{x_o}, W_{x_g} are the kernel weights and W_{h_i}, W_{h_f}, W_{h_o}, W_{h_g} are the recurrent kernel weights, i_t is input gate, f_t is forget gate, o_t is output gate, g_t is block input gate, c_t is cell state and h_t is LSTM output.

Analyzing equations equations 1 - 4 we can conclude that it aims to implement matrix vector multiplication. Hence, by concatenating the kernel and the recurrent kernel weights into two memories as shown in figure 11 the memory allocation can be minimized. This allows efficient hardware implementation as well. The LSTM block consists of two MAC modules accompanied by more arithmetic logic and additional memories. The memories hold kernel and recurrent kernel weights, biases and cell-state data respectively. It also performs *Hard Tanh* and *Hard Sigmoid* activations dynamically.

Originally, LSTM uses sigmoid and tanh as activation functions as we can see from equations 1 - 6. As both of these functions are non-linear, hardware implementation is complex than other activation functions such as ReLU. In order to reduce the hardware complexity, we have used hard sigmoid and hard tanh defined by the following equations:

$$hardtanh(x) = \begin{cases} -1, & x < -1 \\ x, & -1 \le x \le 1 \\ 1, & x > 1 \end{cases} \tag{7}$$

$$hardsigmoid(x) = \begin{cases} 0, & x < -2.5 \\ 0.2 \times x + 0.5, & -2.5 \le x \le 2.5 \\ 1, & x > 2.5 \end{cases} \tag{8}$$

Figure 12 shows the basic methods for tiling in convolution. In figure 12 (a) (input channel tiling), a single feature map is convoluted collaterally by multiple input feature map channels. In figure 12 (b) (image patch tiling), the input feature map is divided into smaller sections and is convoluted parallelly. Finally, in figure 12 (c) (output channel tiling), a single input channel is selected and is convoluted with multiple filters parallelly. From [43] we can conclude the best throughput and best parallel memory access and computation is achieved in the output tiling scheme. The Convolution implemented here follows this output tiling scheme. This is achieved by reading N values from the feature map memory while reading the number of PE times N number of Weight values. N values from the PEs are concatenated and stored as input for the upcoming operation.

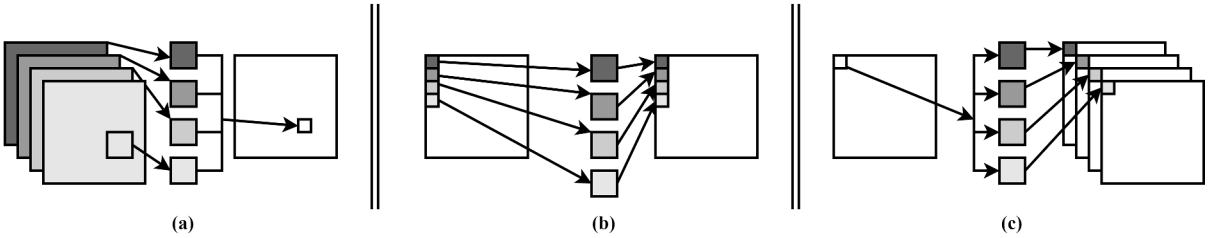


Fig. 12. (a) Input channel tiling (b) Image patch tiling (c) Output channel tiling Examination of various parallel tiling procedures for convolutional layers. Output channel tiling has the least memory communication conflict and hence chosen as tiling scheme for proposed hardware

We implemented E2HRL hardware on low-power Xilinx Artix-7 FPGA to measure its performance. Hardware is designed to be configurable with different hyper-parameters to meet custom application requirements such as low power consumption or high throughput. Parameters include the number of processing elements (PE), filter shapes, number of filters in the convolutional layers, sizes of the dense layers that are configurable to engage maximum parallel processing ability or utilize the least possible hardware resources. The hardware consists of three main units: (1) Convolution block for performing Convolution operation. (2) Fully Connected block that implements the dense layer functionality or an LSTM block to implement the LSTM operations (3) on-chip memory which stores image embedding and subgoal vector. The convolution block operates with a multiplier embedded within it. Along with that, there are separate memory blocks for saving weights and feature maps. The input data is fed to the convolution block to calculate the *valid* convolution of the input using a multiplication unit (MU) and an adder with ReLu activation logic. Strides of two in each direction replaced the time-consuming max-pooling operation. Following the convolutional layers, the first full-connected layer reads the data from the image embedding memory and operates using one multiplier-adder, and a few registers. The address flow and control unit generates memory addresses depending on the layer functionality of either convolution, dense or LSTM. Later, the subgoal vector generated by the Fully Connected/LSTM layer is concatenated with the output of the convolution layer, and form the input of the final Fully Connected layer with Softmax activation to produce proper action. The output of this layer is the action generated by the agent to navigate toward the relevant task within the environment. The control unit manages the data flow through different parts of the hardware. 32-bit fixed-point number format is used for representing the data inside the hardware.

6.2 Xilinx Artix-7 implementation results

As already discussed, the hardware architecture is written in Verilog HDL and is configurable for the desired number of PEs, data and memory widths. Table 3 shows the FPGA implementation results for the E2HRL hardware on the Artix-7 board obtained from the Xilinx Vivado Design Suite. The results illustrates FPGA utilization, latency, power, energy and performance for the design while varying the number of PEs from 1 through 8 at 100MHz operating frequency. The peak performance of the design using its best configuration (i.e. at 100MHz with 8PEs) is 6.4GOPS. The design with this configuration achieves an average performance of 4GOPS for the FC-HRL model while achieving 1 GOPS for the LSTM-HRL model. The energy efficiency peaks at 11.4GOPS/W for the 8 PE design at 100MHz for the FC-HRL and 3GOPS/W for the LSTM-HRL.

Figure13 (right) shows the power consumed for increasing performance when the design is running at 100Mhz. For the FC-HRL model (blue) the performance reaches 4 GOPS at 8 PE, 100 MHz while consuming 348mW power. Similarly, for the LSTM-HRL (green) the performance reaches 1GOPS at 8PE, 100MHz while consuming 389 mW power. The aim of this hardware is to implement it on embedded devices. Hence, we can say that the required deadline to meet is 30 fps, the average number of frames an embedded video camera can generate. All configurations in Table 3 meet this deadline of 33.34ms comfortably. In figure 13 (left) the impact of increasing the

Table 3. The implementation results of proposed E2HRL hardware on Xilinx Artix 7 FPGA. The results are obtained, at a clock frequency of 100 MHz, and the required latency to reach target 30 FPS is 33.34 ms.

Model Config.	#PE	LUT	BRAM	DSPs	Power (mW)	Latency (ms)	Energy (mJ)	Performance (GOPS)	Energy Efficiency (GOPS/W)
FC HRL	1	1249	52	11	295	5.15	1.5	0.5	1.7
	2	1797	52	15	308	2.67	0.9	1	3
	4	2236	52	23	322	1.47	0.5	2	6.2
	8	3186	52	39	348	0.9	0.3	4	11.4
LSTM HRL	1	2485	54	26	336	6.4	2.1	0.4	1.1
	2	3033	54	30	349	4.1	1.4	0.6	1.7
	4	3472	54	38	363	2.85	1	0.8	2.2
	8	4422	54	54	389	2.3	0.8	1	3

number of PEs on energy when the design is running to meet the deadline of 30fps is represented. As the number of PEs is increased the energy drops rapidly initially and gradually saturates. At 8PE the energy consumed is 4.5 mJ for the FC-HRL model and 4.9 mJ for the LSTM-HRL model. Additionally, figure 14 shows the power usage breakdown for the FC-HRL and the LSTM-HRL models implemented on Artix-7 FPGA. The figure shows the breakdown w.r.t the static and dynamic power consumption. The breakdown for the dynamic power is represented as well. From the figure, we can observe that the power consumed by the block-RAMs consume 52% and 46% for the FC-HRL and the LSTM-HRL models respectively. The signal power for the FC and the LSTM HRL models is 15% and 18%.

6.3 Nvidia Jetson TX2 Implementation Results

The architecture is designed to be flexibly deployable for general-purpose devices, allowing the developed machine learning models to be implemented on computing machines ranging from front-end edge devices to back-end computer servers. At least two hardware-level characteristics are assigned to all deep neural network models: model size and amount of computing operations per inference, all of which are upper-bounded by the platform resources they deploy to, or by the inference deadline. Both the hardware resource constraints and the diagnostic delay can satisfy the implementation targets while adding all the components of the framework together.

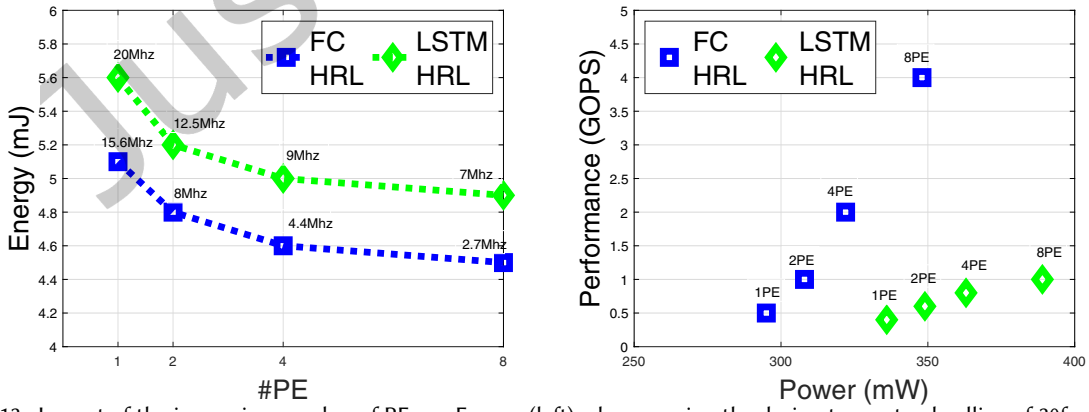


Fig. 13. Impact of the increasing number of PEs on Energy (left) when running the design to meet a deadline of 30fps. Power and performance trade off for the FC and LSTM HRL models (right)

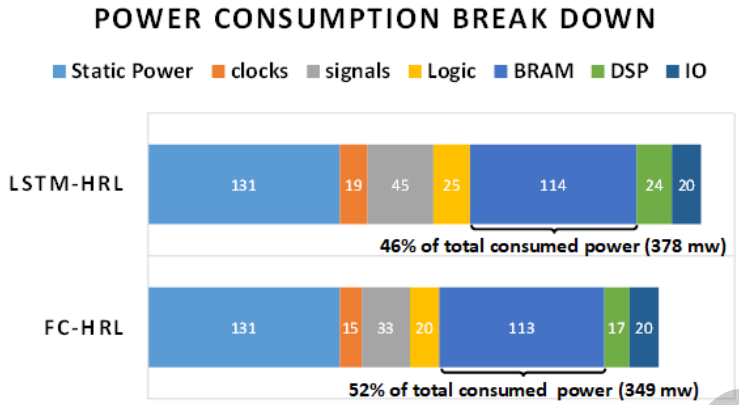


Fig. 14. Power consumption breakdown of the FC-HRL and LSTM HRL hardware architecture on Artix-7 FPGA. The figure shows the breakdown w.r.t the static and dynamic power consumption. It can be observed that the power consumed by the block-RAMs (BRAMs), 52% and 46% for the FC-HRL and the LSTM-HRL models respectively, are the major portion of the consumed power.

Table 4. Deploying the model to commercial off-the-shelf devices including a dual-core Denver CPU, a quad-core ARM A57 CPU, and a combination of ARM CPU + Pascal GPU from the NVIDIA TX2 board

Model Config.	TX2 Config.	CPU Freq. (MHz)	CPU Power (mW)	Latency (ms)	Performance (GFLOP/s)	Energy (mJ)	Energy Efficiency (GFLOP/s/W)
FC HRL	Denver CPU	345	1465	6.0	1.5	8.8	1
		2035	1598	7.7	1.6	15.9	1
	Arm A57 CPU	345	383	21.1	0.4	8.0	1
		2035	1303	5.6	1.7	7.2	1.3
LSTM HRL	Denver CPU	345	1340	9.8	0.2	13.1	0.1
		2035	2070	7.9	0.3	16.3	0.1
	Arm A57 CPU	345	383	27.5	0.08	10.5	0.2
		2035	1303	5.7	0.4	7.4	0.3

The trained models obtained from the previous Section are deployed on two mobile CPUs including Denver (dual-core) and ARM-Cortex A57 (quad-core) implementation with different frequency settings. The Nvidia Jetson TX2 development board, which has precise onboard power calculation, was used to perform all of the settings. We evaluated two different configurations, one with dual-core Denver CPU and another with quad-core ARM CPU. Table 4 shows the real-time implementation results in terms of latency, power, energy consumption and energy efficiency for different models. Latency was minimum 10ms for both the CPU-alone configurations when they were running at their maximum clock frequency of 2035 MHz. The most energy efficiency was achieved when quad-core Arm CPU was running alone at its maximum frequency which is 1.3 GFLOP/s/W when only fully connected layers were used in the subgoal module and 0.3 GFLOP/s/W when LSTM layers were used along with fully connected layers in the subgoal module. However, as the model is too small, the CPUs are underutilized. Moreover, small models do not benefited from heterogeneous platform such as GPU, therefor, GPU acceleration is not helping for the proposed models.

Table 5. Comparison of the E2HRL hardware implementation results with similar existing works

	[10]	[13]	[39]	[44]	[41]	**FC-HRL**	**LSTM-HRL**
Application	Image Classification (CIFAR)	Physical Activity Monitoring	Image Classification (CIFAR)	Image Classification (CIFAR)	RL	**RL**	**RL**
Platform	FPGA Zynq	FPGA Artix7	FPGA Zynq	FPGA Zynq	FPGA Zynq	**FPGA Artix7**	**FPGA Artix7**
Input Dimension	32x32x3	64x40x1	32x32x3	32x32x3	5x1	**40x30x3**	**40x30x3**
Precision	16-bit	16-bit	16-bit	32-bit	32-bit	**32-bit**	**32-bit**
Freq (MHz)	180	100	100	100	125	**100**	**100**
Throughput (fps)	35.7	491	46.7	63.5	18.8	**1110**	**435**
Perf (GOPS)	0.94	1.6	1.23	1.67	NR	**4**	**1**
Power (mW)	>17587	175	2944	1015	NR	**348**	**389**
Energy (mJ/frame)	>500	0.35	63	33.7	NR	**0.3**	**0.8**
Energy Efficiency (GOPS/W)	<0.05	9.14	0.42	1.65	NR	**11.4**	**3**

6.4 Discussion

In recent years, many FPGA-based accelerators for convolutional neural network and computer vision applications have been proposed. This paper proposed an scalable hardware for Reinforcement Learning applications which takes images as its input. Therefore, we compared our design with previous works that targeted computer vision and image classification applications. Implementation results show that compared to [10, 39, 44], our hardware with an optimum selection of configuration parameters, number of PEs, and frequency, consume less energy per classification and is more energy-efficient. Compared to [10], although our image dataset has a larger dimension, the results show significantly higher throughput and less power consumption, and better energy efficiency. Additionally, our design shows a performance almost equivalent to [13] peak performance, despite using significantly larger input data. Comparison of this work with the existing FPGA based deep neural network is summarized in Table 5.

As shown in the Table 3, the best configuration in terms of energy efficiency is 8 PEs that resulted in an average of 11.4 and 3 GOP/S/W energy efficiency and 0.3 and 0.8 mJ of energy for each inference for FC-HRL and LSTM HRL models respectively. Low power configurations consume 295 and 336 mW power to generates actions which is suitable for embedded applications that usually has limitted power envelope. Taking real time vision application constraint into the account[42], all of the configurations meet the typical 30 FPS constraint and consume less than 350 mW.

7 CONCLUSION

In this paper, we proposed E2HRL, a cross-layer energy-efficient hardware architecture design method using hierarchical Reinforcement Learning. We evaluate the performance of the model in several RL environments with different levels of complexity. Our Fully Connected and LSTM based models shows better performance compared to conventional RL, specially when the environment get more complex. After a systematic model architecture optimization , the best configuration is selected to be implemented on hardware. We designed an scalable hardware architecture which is configurable with different number of PEs and could be implemented to achieve high throughput or low power consumption. The best configuration in terms of energy efficiency is 8 PEs that resulted in an average of 11.4 and 3 GOP/S/W energy efficiency and 0.3 and 0.8 mJ of energy for each

inference for FC-HRL and LSTM HRL models respectively. E2HRL low power configurations consume 295 and 336 mW power to generates each action in the environment.

8 ACKNOWLEDGMENT

This project was sponsored by the U.S. Army Research Laboratory under Cooperative Agreement Number W911NF-10-2-0022. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

REFERENCES

[1] 2020 (accessed October, 2020). NVIDIA Jetson TX2. NVIDIA. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-tx2/

[2] Jacob Andreas, Dan Klein, and Sergey Levine. 2017. Modular multitask reinforcement learning with policy sketches. In *International Conference on Machine Learning*. PMLR, 166–175.

[3] Pierre-Luc Bacon, Jean Harb, and Doina Precup. 2017. The option-critic architecture. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 31.

[4] Maxime Chevalier-Boisvert. 2018. gym-miniworld environment for OpenAI Gym. https://github.com/maximecb/gym-miniworld.

[5] Hyungmin Cho, Pyeongseok Oh, Jiyoung Park, Wookeun Jung, and Jaejin Lee. 2019. Fa3c: Fpga-accelerated deep reinforcement learning. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 499–513.

[6] Lucileide MD Da Silva, Matheus F Torquato, and Marcelo AC Fernandes. 2018. Parallel implementation of reinforcement learning q-learning technique for fpga. *IEEE Access* 7 (2018), 2782–2798.

[7] William J. Dally, Yatish Turakhia, and Song Han. 2020. Domain-Specific Hardware Accelerators. *Commun. ACM* 63, 7 (June 2020), 48–57.

[8] Muhammad Mudassir Ejaz, Tong Boon Tang, and Cheng-Kai Lu. 2021. A fast learning approach for autonomous navigation using a deep reinforcement learning method. *Electronics Letters* 57, 2 (2021), 50–53.

[9] Sunil Gandhi, Tim Oates, Tinoosh Mohsenin, and Nicholas R Waytowich. 2019. Learning Behaviors from a Single Video Demonstration Using Human Feedback. (2019).

[10] Gopalakrishna Hegde, Nachiappan Ramasamy, Nachiket Kapre, et al. 2016. CaffePresso: an optimized library for deep learning on embedded accelerator-based platforms. In *2016 International Conference on Compliers, Architectures, and Sythesis of Embedded Systems (CASES)*. IEEE, 1–10.

[11] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[12] Morteza Hosseini, Mohammad Ebrahimabadi, Arnab Neelim Mazumder, Houman Homayoun, and Tinoosh Mohsenin. 2021. A fast method to fine-tune neural networks for the least energy consumption on fpgas. *UMBC Student Collection* (2021).

[13] Ali Jafari, Ashwinkumar Ganesan, Chetan Sai Kumar Thalisetty, Varun Sivasubramanian, Tim Oates, and Tinoosh Mohsenin. 2018. Sensornet: A scalable and low-power deep convolutional neural network for multimodal data classification. *IEEE Transactions on Circuits and Systems I: Regular Papers* 99 (2018), 1–14.

[14] Mohit Khatwani et al. 2019. A Low Complexity Automated Multi-channel EEG Artifact Detection using EEGNet. In *2019 IEEE EMBS Conference on Neural Engineering*. IEEE.

[15] Changhyeon Kim, Sanghoon Kang, Sungpill Choi, Dongjoo Shin, Youngwoo Kim, and Hoi-Jun Yoo. 2019. An Energy-Efficient Deep Reinforcement Learning Accelerator With Transposable PE Array and Experience Compression. *IEEE Solid-State Circuits Letters* 2, 11 (2019), 228–231.

[16] Changhyeon Kim, Sanghoon Kang, Dongjoo Shin, Sungpill Choi, Youngwoo Kim, and Hoi-Jun Yoo. 2019. A 2.1 TFLOPS/W mobile deep RL accelerator with transposable PE array and experience compression. In *2019 IEEE International Solid-State Circuits Conference-(ISSCC)*. IEEE, 136–138.

[17] Nate Kohl and Peter Stone. 2004. Policy gradient reinforcement learning for fast quadrupedal locomotion. In *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA'04. 2004*, Vol. 3. IEEE, 2619–2624.

[18] Amey Kulkarni. 2017. Heterogeneous and Scalable Sketch-based Framework for Big Data Acceleration on Low Power Embedded Cores. *Phd Dissertation* (February 2017).

[19] Sicheng Li et al. 2015. Fpga acceleration of recurrent neural network based language model. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 111–118.

[20] Shanshan Liu, Xiaochen Tang, Farzad Niknia, Pedro Reviriego, Weiqiang Liu, Ahmed Louri, and Fabrizio Lombardi. 2021. Stochastic Dividers for Low Latency Neural Networks. *IEEE Transactions on Circuits and Systems I: Regular Papers* 68, 10 (2021), 4102–4115.

[21] N. K. Manjunath, A. Shiri, M. Hosseini, B. Prakash, N. R. Waytowich, and T. Mohsenin. 2021. An Energy Efficient EdgeAI Autoencoder Accelerator for Reinforcement Learning. *IEEE Open Journal of Circuits and Systems* 2 (2021), 182–195. https://doi.org/10.1109/OJCAS.2020.3043737

[22] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (2015), 529.

[23] Andrew Y Ng et al. 2006. Autonomous inverted helicopter flight via reinforcement learning. In *Experimental robotics IX*. Springer, 363–372.

[24] Bharat Prakash et al. 2020. Guiding Safe Reinforcement Learning Policies Using Structured Language Constraints. In *SafeAI workshop Thirty-Fourth AAAI Conference on Artificial Intelligence*. AAAI.

[25] Pedro Reviriego, Shanshan Liu, Otmar Ertl, Farzad Niknia, and Fabrizio Lombardi. 2021. Computing the Similarity Estimate Using Approximate Memory. *IEEE Transactions on Emerging Topics in Computing* (2021).

[26] Andrei A Rusu et al. 2017. Sim-to-real robot learning from pixels with progressive nets. In *Conference on Robot Learning*. 262–270.

[27] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).

[28] Masoud Shahshahani and Dinesh Bhatia. 2021. Resource and Performance Estimation for CNN Models using Machine Learning. In *2021 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 43–48.

[29] Masoud Shahshahani, Mohammad Sabri, Bahareh Khabbazan, and Dinesh Bhatia. 2021. An Automated Tool for Implementing Deep Neural Networks on FPGA. In *2021 34th International Conference on VLSI Design and 2021 20th International Conference on Embedded Systems (VLSID)*. IEEE, 322–327.

[30] Shengjia Shao, Jason Tsai, Michal Mysior, Wayne Luk, Thomas Chau, Alexander Warren, and Ben Jeppesen. 2018. Towards hardware accelerated reinforcement learning for application-specific robotic control. In *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 1–8.

[31] Aidin Shiri, Arnab Neelim Mazumder, Bharat Prakash, Houman Homayoun, Nicholas R Waytowich, and Tinoosh Mohsenin. 2021. A Hardware Accelerator for Language Guided Reinforcement Learning. *IEEE Design & Test* (2021).

[32] Aidin Shiri, Arnab Neelim Mazumder, Bharat Prakash, Nitheesh Kumar Manjunath, Houman Homayoun, Avesta Sasan, Nicholas R Waytowich, and Tinoosh Mohsenin. 2020. Energy-Efficient Hardware for Language Guided Reinforcement Learning. In *Proceedings of the 2020 on Great Lakes Symposium on VLSI*. 131–136.

[33] David Silver et al. 2017. Mastering the game of go without human knowledge. *nature* 550, 7676 (2017), 354–359.

[34] Satinder Singh et al. 2002. Optimizing dialogue management with reinforcement learning: Experiments with the NJFun system. *Journal of Artificial Intelligence Research* 16 (2002), 105–133.

[35] Jiang Su, Jianxiong Liu, David B Thomas, and Peter YK Cheung. 2017. Neural network based reinforcement learning acceleration on fpga platforms. *ACM SIGARCH Computer Architecture News* 44, 4 (2017), 68–73.

[36] Richard S Sutton, Andrew G Barto, et al. 1998. *Introduction to reinforcement learning*. Vol. 2. MIT press Cambridge.

[37] Mineto Tsukada, Masaaki Kondo, and Hiroki Matsutani. 2020. A neural network-based on-device learning anomaly detector for edge devices. *IEEE Trans. Comput.* 69, 7 (2020), 1027–1044.

[38] Eric Tzeng et al. 2020. Adapting deep visuomotor representations with weak pairwise constraints. In *Algorithmic Foundations of Robotics XII*. Springer, 688–703.

[39] Ying Wang, Jie Xu, Yinhe Han, Huawei Li, and Xiaowei Li. 2016. DeepBurning: Automatic Generation of FPGA-Based Learning Accelerators for the Neural Network Family. In *Proceedings of the 53rd Annual Design Automation Conference* (Austin, Texas) *(DAC '16)*. Association for Computing Machinery, New York, NY, USA, Article 110, 6 pages. https://doi.org/10.1145/2897937.2898003

[40] Hirohisa Watanabe, Mineto Tsukada, and Hiroki Matsutani. 2020. An FPGA-Based On-Device Reinforcement Learning Approach using Online Sequential Learning. *arXiv preprint arXiv:2005.04646* (2020).

[41] Hirohisa Watanabe, Mineto Tsukada, and Hiroki Matsutani. 2021. An FPGA-Based on-device reinforcement learning approach using online sequential learning. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 96–103.

[42] Ming Yang, Shige Wang, Joshua Bakita, Thanh Vu, F Donelson Smith, James H Anderson, and Jan-Michael Frahm. 2019. Re-thinking CNN frameworks for time-sensitive autonomous-driving applications: Addressing an industrial challenge. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 305–317.

[43] Chen Zhang et al. 2015. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 161–170.

[44] Guanwen Zhong, Akshat Dubey, Cheng Tan, and Tulika Mitra. 2019. Synergy: An HW/SW Framework for High Throughput CNNs on Embedded Heterogeneous SoC. *ACM Trans. Embed. Comput. Syst.* 18, 2, Article 13 (March 2019), 23 pages. https://doi.org/10.1145/3301278

[45] Yuke Zhu et al. 2017. Target-driven visual navigation in indoor scenes using deep reinforcement learning. In *2017 IEEE international conference on robotics and automation (ICRA)*. IEEE, 3357–3364.