


This work was written as part of one of the author's official duties as an Employee of the United States Government and is therefore a work of the United States Government. In accordance with 17 U.S.C. 105, no copyright protection is available for such works under U.S. Law. Access to this work was provided by the University of Maryland, Baltimore County (UMBC) ScholarWorks@UMBC digital repository on the Maryland Shared Open Access (MD-SOAR) platform.

Please provide feedback

Please support the ScholarWorks@UMBC repository by emailing scholarworks-group@umbc.edu and telling us what having access to this work means to you and why it's important to you. Thank you.

An Energy Efficient EdgeAI Autoencoder Accelerator for Reinforcement Learning

NITHEESH KUMAR MANJUNATH¹ (Member, IEEE), AIDIN SHIRI¹,
MORTEZA HOSSEINI¹ (Graduate Student Member, IEEE), BHARAT PRAKASH¹,
NICHOLAS R. WAYTOWICH², AND TINOOSH MOHSENIN¹ 

¹Department of Computer Science and Electrical Engineering, University of Maryland, Baltimore, MD 21250, USA

²U.S. Army Research Laboratory, Aberdeen, MD, USA

This article was recommended by Associate Editor Y. Li.

CORRESPONDING AUTHOR: N. K. MANJUNATH (e-mail: n67@umbc.edu)

This work was supported by the U.S. Army Research Laboratory through Cooperative Agreement under Grant W911NF-10-2-0022.

ABSTRACT In EdgeAI embedded devices that exploit reinforcement learning (RL), it is essential to reduce the number of actions taken by the agent in the real world and minimize the compute-intensive policies learning process. Convolutional autoencoders (AEs) has demonstrated great improvement for speeding up the policy learning time when attached to the RL agent, by compressing the high dimensional input data into a small latent representation for feeding the RL agent. Despite reducing the policy learning time, AE adds a significant computational and memory complexity to the model which contributes to the increase in the total computation and the model size. In this article, we propose a model for speeding up the policy learning process of RL agent with the use of AE neural networks, which engages binary and ternary precision to address the high complexity overhead without deteriorating the policy that an RL agent learns. Binary Neural Networks (BNNs) and Ternary Neural Networks (TNNs) compress weights into 1 and 2 bits representations, which result in significant compression of the model size and memory as well as simplifying multiply-accumulate (MAC) operations. We evaluate the performance of our model in three RL environments including DonkeyCar, Miniworld sidewalk, and Miniworld Object Pickup, which emulate various real-world applications with different levels of complexity. With proper hyperparameter optimization and architecture exploration, TNN models achieve near the same average reward, Peak Signal to Noise Ratio (PSNR) and Mean Squared Error (MSE) performance as the full-precision model while reducing the model size by $10\times$ compared to full-precision and $3\times$ compared to BNNs. However, in BNN models the average reward drops up to 12% - 25% compared to the full-precision even after increasing its model size by $4\times$. We designed and implemented a scalable hardware accelerator which is configurable in terms of the number of processing elements (PEs) and memory data width to achieve the best power, performance, and energy efficiency trade-off for EdgeAI embedded devices. The proposed hardware implemented on Artix-7 FPGA dissipates 250 μJ energy while meeting 30 frames per second (FPS) throughput requirements. The hardware is configurable to reach an efficiency of over 1 TOP/J on FPGA implementation. The proposed hardware accelerator is synthesized and placed-and-routed in 14 nm FinFET ASIC technology which brings down the power dissipation to 3.9 μJ and maximum throughput of 1,250 FPS. Compared to the state of the art TNN implementations on the same target platform, our hardware is $5\times$ and $4.4\times$ ($2.2\times$ if technology scaled) more energy efficient on FPGA and ASIC, respectively.

INDEX TERMS Reinforcement learning, autonomous systems, autoencoder, binary neural networks (BNNs), ternary neural networks (TNNs), EdgeAI, energy efficiency, FPGA, ASIC.

I. INTRODUCTION

REINFORCEMENT Learning is a goal-oriented paradigm of machine learning in which an agent tries

to learn a policy to complete complex tasks by trial and error. RL has shown great accomplishments in problems that require sequential decision making where an agent

needs to take actions in an environment to maximize cumulative future rewards. Despite achieving great success in unsupervised problems such as robotics and autonomous navigation [1], training the agent is a compute-intensive and time-consuming process because of the large amount of trial and error actions required to learn new policies. The newly decided actions are taken based on recently experienced events as well as the new experience that the agent gains through the course of learning. The new events can typically be new images visioned by the agent that are initially high dimensional data, whose complexity impact on the performance of the learning process. The number of actions in an RL agent is the second important factor that complicates the learning process. Thus, reducing the complexity of both high dimensional event data (images) and the number of actions can facilitate the learning process of the agent and decrease the hardware complexity, leading to improved power dissipation, latency, and efficiency during the model deployment.

Autoencoder deep neural networks are a class of neural networks, categorized as an unsupervised learning algorithm, that are adopted for tasks such as image denoising [2] and data compression [3]. Autoencoders can be used to compress an image data into an abstract representation, and have shown promising results in reducing the time-intensive RL training process [4]. Autoencoders compress high dimensional input data such as large images into a small vector, named code-word or latent representation, which include the essential information of the input data. Employing an autoencoder network before the RL agent speeds up the policy learning process at the expense of additional computation for further data compression. During the inference, the autoencoder network is considerably larger in terms of model size and computation compared to the trained RL agents network. The slow policy learning process during training and the large autoencoder network architecture during testing are two main drawbacks that limit deploying reinforcement learning on embedded devices.

In this article, we propose an energy-efficient hardware architecture for reinforcement learning coupled with autoencoder neural networks that can be implemented on EdgeAI embedded devices. The autoencoder models with compressed neural networks are assessed for their performance with the RL agent in three environments with varying complexities. This article makes the following contributions:

- Train an autoencoder in full-precision as a baseline for image compression to reduce the overall complexity of the RL agent.
- Reduce the complexity and model size of the baseline autoencoder to Binary and Ternary precision models to evaluate the autoencoders performance with three real-world reinforcement learning environment simulators involving varying environment complexity.
- Perform extensive hyperparameter search on TNNs and BNNs to reduce memory footprint while maintaining the maximum achievable episode reward from RL agent,

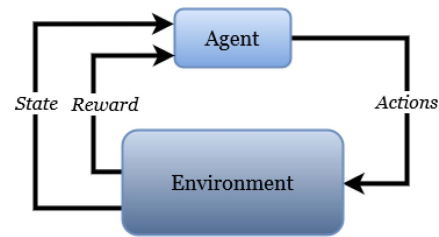


FIGURE 1. Illustration of the RL agent interacting with the environment to achieve the maximum possible reward.

as well as meeting the required PSNR and MSE in the three environments.

- Propose an energy-efficient and scalable TNN autoencoder hardware that allows experimenting with the number of PEs and memory data widths to achieve the best power, performance, and energy efficiency trade-off for EdgeAI embedded devices.
- Implement the design on FPGA and post-layout ASIC in the 14nm FinFET fabrication process and experiment with the number of PEs and memory width to get optimum energy efficiency and power consumption.
- Compare the proposed implementation with the state of the art TNN implementations on FPGA and ASIC.

II. BACKGROUND AND RELATED WORKS

Despite past efforts and early success on reinforcement learning, most of the initial works in the RL domain are limited to tasks such as training a specific robot [5], automatic inverted flight control [6], and optimizing dialog policy [7]. These works do not take into account for optimizing the power and performance trade offs. The advent of deep learning has caused significant progress in RL like other machine learning areas, which resulted in 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 [8]. Authors in [9] have achieved human-level control in many of Atari games with Deep RL. Several works are also proposed which use the robotic simulators to train the network with an unlimited amount of data and afterward, transferring the knowledge from simulator to real world [10], [11]. Training autonomous robots to navigate to designated locations is also another application [1]. One of the problems with RL is low sample efficiency where the agent needs to interact with the environment for a very large number of steps to learn good policies. As a pre-training step, autoencoders can be used to learn the state representations and speed up RL [4]. Autoencoders significantly reduce the required amount of

actions to train the agent and consequently, the energy consumption to learn the desired policy. This method resulted in $4\text{-}5\times$ performance improvement for implementing the design on Nvidia Jetson TX2. The feasibility of speeding up the RL with an autoencoder network for autonomous driving application in real-world tasks is demonstrated in [12].

Recent growth in neural networks and deep learning based algorithms have been coupled with a significant increase of the model size and networks with a high number of parameters. To address the high computational and storage complexity of the neural networks, different computing platforms have been engaged for the 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 [13]. While designing hardware accelerators have been widely engaged in various fields of machine learning such as image [14], medical [15], [16], and voice [17], only a few works have considered hardware requirements for RL applications [18].

In summary, most of the previous works in the RL domain are limited to the software implementations [19], 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 [20]). 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. In this article, we propose a real-time low-power hardware accelerator for RL applications, by taking advantage of binary/ternary precision neural networks whose efficiency can reach up to 1.1 and 42.3 TOP/J on FPGA and ASIC respectively.

III. PROPOSED SYSTEM ARCHITECTURE

An overview of our proposed design, training methods, and the architecture of the system will be discussed in this Section. The proposed design has two separate components: the RL agent and a convolutional autoencoder network. The autoencoder network is trained and placed before the RL agent to speed up the policy learning process. The RL agent and the autoencoder network are trained offline and the trained model will be deployed to hardware for testing. Fig. 2 shows the high-level block diagram of the proposed system that consists of an autoencoder and an RL agent. The RL

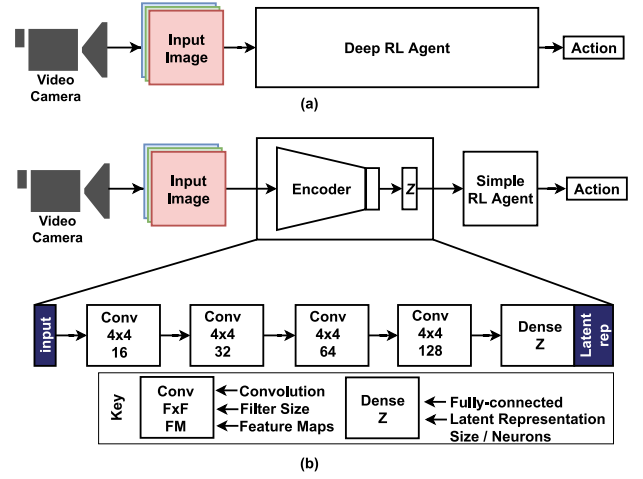


FIGURE 2. (a) Block-level representation of conventional deep RL agent. (b) Block-level representation of the proposed system. The image from the video camera or simulator is input to the pre-trained autoencoder to obtain latent representation. RL agent perceives this as input, and appropriate actions are taken. The base configuration of the autoencoder is shown with four convolutional and a fully-connected layer. Stride is 2 for all convolutional layers.

agent has the goal of learning a specific policy by striving to maximize the long-term cumulative reward returned from the environment through a set of trials and errors. This poses a challenge in terms of the significant training time required to train the model, without the pre-trained autoencoder that provides the latent representation to achieve the same reward as RL with a pre-trained autoencoder. Once we have the pre-trained autoencoder, training time for the RL agent decreases significantly as shown in [4]. The comparison of the autoencoder and RL agent in terms of computation and parameters for the baseline model is shown in Table 1. It shows the overhead of the RL agent comparing to the AE network in full precision model. The complexity of RL agent is insignificant when compared to AE network in a way that it comprises less than 0.05 percent of number of computations and roughly 1 percent of the number of parameters. Therefore, it is fair to say the main bottleneck of the design is the AE network.

A. TRAINING THE RL AGENT

Most of modern reinforcement learning algorithms use deep neural networks as the RL agent to map sequences of states to desired actions and adjust the network through back-propagation with respect to returned reward from the environment. States are usually a high dimensional image fed to the RL agent from the environment. The agent learns a policy by receiving rewards upon actions it is allowed to take without specifying how to take the actions or how to accomplish the task. Generally, RL is modeled through Markov Decision Processes (MDP). The MDP is a list of elements (S, A, P, R, γ) , which denotes state space, action space, transition function, reward function, and discount factor respectively. As shown in Fig. 1, the agent tries to interact with the environment by performing actions and the

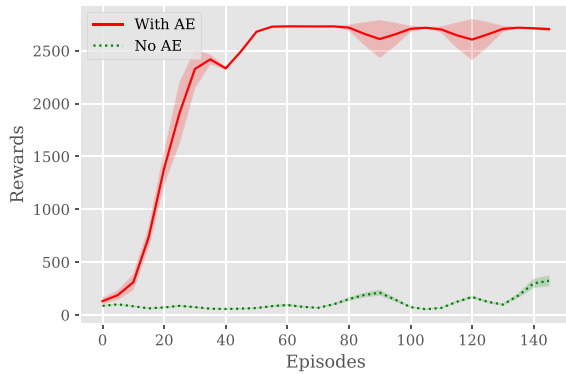


FIGURE 3. This figure shows the episode rewards accumulated while training the RL agent with and without AutoEncoder (AE) network. We observe that the RL agent converge considerably faster when trained with pre-trained AE comparing to the traditional model when deployed on real-time application of autonomous driving car.

environment takes the agent's action along with the current state and returns a reward and next state. The reward is a feedback from the environment by evaluating the agent's performance for completing a task. The policy, $p(a_t|s_{<t}, a_{<t})$ is the function which maps state to the action that guarantees the optimum future reward. This is usually an expectation, $E_p[\sum_t r_t]$. The Trajectory is a sequence of state actions [21].

Convolutional neural networks are generally the best choice for extracting features and meaningful representations out of image context. However, combining feature extraction and decision-making network makes the RL policy learning process quite slow. Therefore, separating the feature extraction part from the decision-making section can significantly enhance the training time of the policy learning process. As a result, we placed the convolutional autoencoder network before the RL agent to address this problem. The RL agent in our proposed system consists of two fully-connected layers and an output layer with the number of neurons equal to the number of actions the agent can take in the environment. This is explained for all the three environments used for experimentation in Section V.

B. PRE-TRAINING THE AUTOENCODER NETWORK

The autoencoder networks are trained with a large number of input images gathered from the environment to accelerate the RL agent's policy learning process. The autoencoder has the task of feature extraction and dimensionality reduction of the environment images (states). The number of required images to train the autoencoder network is related to the complexity of the environment and discussed in further detail in Section V. The autoencoder is forced to compress the input image into a small vector of latent representation, and then reconstruct it from compressed representation to measure the quality of latent representation, using measures such as MSE and PSNR. After training, the autoencoder part of the network is taken and placed before the agent to feed it with the most important features of the environment. This

TABLE 1. Comparison of autoencoder and RL in terms of computation and model size for the baseline network. The autoencoder has a significant portion of model size and is the best choice for compression and complexity reduction.

Factor	Autoencoder	RL	Ratio
Parameters	377K	4.3K	44×
Computation	20.1M	4.3K	2,392×
Model Size	1,476 KB	16.8 KB	88×
Feature map Size	56 KB	0.24 KB	234×

methods viability for embedded devices has been proved in previous works [22].

We initially illustrate the credibility of our proposed model for the task of real-time autonomous driving car in Donkeycar environment [22]. Figure 3 shows the episode rewards accumulated by the RL agent for learning the policy of driving the car while staying on track with and without AE network. We observe that in the environment with a large number of states the RL agent converge considerably faster when trained with pre-trained AE comparing to the traditional model when deployed on real-time application of autonomous driving car. Starting from these findings, we use the same method to accelerate the RL agents policy leaning process and implement it with an efficient low bit-width hardware accelerator for pre-trained autoencoder.

IV. LOW BIT WIDTH AUTOENCODER NETWORK

The disadvantage of the encoder is that it contains a significant portion of the design in comparison with the RL agent as inferred from the Table 1, which denotes the fact that quantization and compressing the encoder, helps in deployment in the hardware in terms of memory requirement and complexity of MAC operations. In this work, we explore various methods of quantizing the encoder that generates a good latent representation which is fed into the RL agent of three fully-connected layers. In the following sections, we have mentioned several quantization techniques.

Neural networks in general are computationally intensive and can have large model sizes. Hardware that implements a neural network for inference commonly has two major components: 1) circuitry that perform multiply and accumulate operations, and 2) memory units and buffers that store the neural network model and weights and the temporary intermediate feature map. The two components contribute to the majority of the power dissipation. The complexity of the operations (e.g., fixed-point or floating-point operations) impacts the power of the MAC units, and the model size determines the memory footprint that, per se, impacts the power dissipation for the storage units in the hardware. Seeking efficient implementation of neural networks, many methods have been proposed that propose complexity reduction and model size compression in neural networks. One of these methods is quantization. In a quantized neural network, the parameters and the intermediate data are represented with a certain number of bits that are less than traditional precision levels as in 32-bit (single precision) or 64-bit (double precision) systems. The

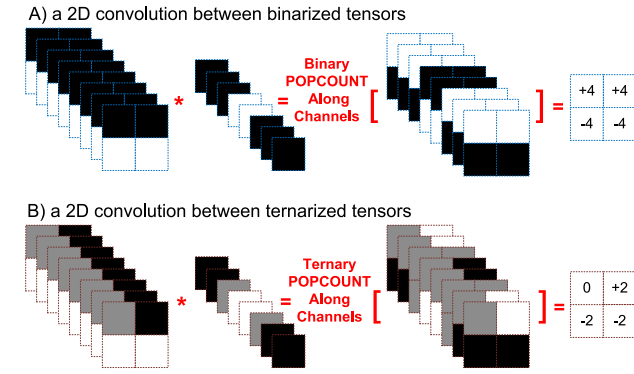


FIGURE 4. A 2D convolution between $2 \times 2 \times 8$ and $1 \times 1 \times 8$ tensors. The whole computation is broken down into two steps: element-wise multiplication, and accumulation over channels. A) the tensors are binarized, where colors white and black represent values -1 and $+1$ respectively. The multiplication is equivalent to element wise xnor-ing. B) the tensors are ternarized, where colors black, grey, and white represent values -1 , 0 , and $+1$ respectively.

extreme case of quantization is when the value precision levels are narrowed down to 1 bit or two bits as in binary or ternary precision neural networks. The two extreme cases are the subject of this Section and will be further explained below.

A. BINARY NEURAL NETWORKS

Binary neural networks encompass a varied number of neural networks [23], [24] where either weights or activations or both are binary values with the most extreme quantization level being the 1 bit representation for -1 or $+1$ values. The single bit value represents a $+1$ if equal to 1, and -1 if equal to 0. BinaryConnect [25] is the first work to propose neural networks with binarized weights. BNN [23] is an extension to the BinaryConnect where both weight and activation values are binarized. Compared to a double-precision counterpart, binarization can compress both the model size and the feature map size by up to $64\times$. It also reduces the complexity of multiplication and accumulation operations down to bit-wise operations, where a multiplier can be implemented with an XNOR logic, and an accumulator logic that adds up N single-bit values. This can be implemented with a logic that takes in an N bit data (resulted by an N -bitwise XNOR operation) and adds up its bits considering its 0-valued bit as -1 values. This logic is referred to as population-count, dubbed as popcount.

As an example, Fig. 4-A shows a 2D convolution between a binarized $2 \times 2 \times 8$ tensor with a binarized $1 \times 1 \times 8$ filter in 2 steps. In the first step, channels of the tensor are bitwise-xnored with the filter to result in an intermediate binarized tensor that hold elementwise multiplications, and in the second step, the elementwise multiplications are summed along the channels of the intermediate tensor which, as described above, is carried out with a popcount logic. The activation function that binarizes the feature map data is a sign function that is denoted by the following equation if implemented deterministically:

$$x^b = f_b(x) = \text{sign}(x) = \begin{cases} +1, & \text{if } x \geq 0 \\ -1, & \text{otherwise.} \end{cases} \quad (1)$$

If a sign activation function follows a binarized 2D convolution, and if the weight values in input and filter tensors are packed along the channels of the two tensors, then every output neuron of such combination is calculated as follows:

$$Y = \text{sign}(\text{popcount}(\text{xnor}(w, x))), \quad (2)$$

where XNOR performs bitwise XNOR operation between overlapping patches, that contribute to the generation of equivalent output, from the two tensors, and popcount accumulates over all the resulted elements considering 0 as -1 , and finally, the sign function extracts the sign bit of the total sum.

The main drawback of the BNNs is that their accuracy may drop considerably after binarization. One of the main reasons of the accuracy loss in BNNs is that the model is forced to pick a non-zero value (either -1 or $+1$) for every pair of input and output neurons in a BNN layer even if they are totally independent. In order to fix such inflexibility, ternary neural networks were proposed that let a third value, 0, in the field of their representable values.

B. TERNARY NEURAL NETWORKS

Ternary Neural Networks have stronger expressiveness than their binary counterparts, and similarly, encompass a varied number of neural networks that target either weights [26], [27] or both weights and activations [28], [29]. Their main difference to binary networks is that they include the value 0, thus allowing them to prune unnecessary interconnections between independent neurons. They assign 2 bits to represent 3 values. To make ternary weights networks, a ternarizing activation is used that is threshold-based and is defined as:

$$x^t = f_t(x|\Delta) = \begin{cases} +1, & \text{if } x > \Delta \\ 0, & \text{if } |x| \leq \Delta \\ -1, & \text{if } x < -\Delta, \end{cases} \quad (3)$$

where Δ is a thresholding quantity that defines the vicinity region to map non-zero values to 0. Δ is set to 0.5 in all experiments of this article, however, it can trigger the sparsity (percentage of zero weights) in the ternarized values of a ternary neural network. In [27] it is shown that triggering the Δ to yield a ternary weight ResNet-20 [30] with sparsity between 30% to 50% results in a validation accuracy approximately equal to that of the full-precision ResNet-20. The accuracy gets deteriorated if the sparsity level deviates from either side of 30% to 50%.

As an example of ternarized neural networks, Fig. 4-B shows a 2D convolution between a ternarized $2 \times 2 \times 8$ tensor with a binarized $1 \times 1 \times 8$ filter in 2 steps. In the first step, channels of the tensor are elementwise-multiplied with the filter using simple multiplier logic to result in an intermediate ternarized tensor that includes all elementwise multiplications, and in the second step, the elementwise multiplications are summed along the channels of the intermediate tensor with a logic similar to that of the



FIGURE 5. Screenshot of environment simulation window, (Left) Donkeycar: The RL agent (toy car) should navigate the track without going off the course, (Center) Miniworld sidewalk: The RL agent navigates the red cube while staying in the sidewalk, i.e., right side of safety cones, and (Right) Miniworld Object Pickup: The RL agent navigates to pickup five objects including cube, key and sphere in a room.

popcount logic as described for BNNs. Similar to BNNs, the 2-bit ternary values can be packed in wide memory entries and the dot product between two packed vectors followed by a ternary activation can be denoted as:

$$Y = f_t(Tpopcount(Tmul(w, x))), \quad (4)$$

where $Tmul$ performs element-wise multiplication between aligned packed 2-bit entries in w and x to generate a temporary vector, and $Tpopcount$ performs a summation over individual 2-bit results considering 11, 00, and 01 representing -1 , 0 , and $+1$ respectively. The ternary activation function f_t will then decide whether the final result is negative, zero, or positive to return a -1 , 0 , or $+1$ ternary value. Both $Tmul$ and $Tpopcount$ functions can be broken into extremely small logic similar to XNOR and popcount logic in BNNs.

V. ENVIRONMENT SETUP

In this Section, we explain the experimental setup for testing the RL agent with full-precision and low bit-width autoencoder neural networks. Three test environments with different complexities are selected to evaluate the RL agent performance with the proposed configuration. DonkeyCar simulator [22], the Miniworld sidewalk, and Miniworld Object Pickup environments [31] are chosen as environments, which have small, medium, and large complexity in terms of details in the environment and the actions taken by the RL agent respectively. For each case, we collect a specific number of images using a human expert to train the autoencoder network. Autoencoder has the task of compressing the environment image into a small code-word or latent representation of the image to pass this meaningful information to the RL agent as input. The quality of the reconstructed image from the decoder is measured by MSE and PSNR. Fig. 5 illustrates snapshots from each environment. These environments can easily interact via Python libraries with the Keras Machine Learning backend.

A. CASE STUDY 1: DONKEYCAR SIMULATOR

The first environment we use for testing the RL agent is the Donkeycar [22] simulator as illustrated in Fig. 5-left. In this environment, the RL agent is a toy car that learns the policy of driving on the track while increasing its speed. Steering and throttling are two continuous actions that can be taken

by the agent within the environment. The reward function that the RL agent receives from the environment is a function of cross-track error that measures the distance between the center of the track and car [4]. So the performance measured in terms of episode length is proportional to the length of track covered by the agent. The car in the image always begins at the start of a random track selected by the software.

The dataset used for training and testing the autoencoder network in the environment consists of 15000 3-channel images, where the size of each image is $120 \times 160 \times 3$, from which, 13000 images were utilized in the training set and 2000 in the testing set. This dataset is collected such that a sufficient number of images from each direction of the environment are included.

B. CASE STUDY 2: MINIWORLD SIDEWALK AND OBJECT PICKUP ENVIRONMENT

Donkeycar simulator is a simple environment with limited action space. To evaluate our model in a more complex environment, we choose the Miniworld environment, which is a minimalistic 3D environment simulator for reinforcement learning and robotic research [31]. Although graphics of the Miniworld environment is basic and physics are simpler than the real world environments, Miniworld has more complexity than the Donkeycar simulator in terms of the environment and RL tasks. Therefore, the RL agent performance can be evaluated in a more complex setting. In Miniworld, the RL agent always starts at the random position in the environment and is tasked to navigate towards the goal within a specified number of steps. Miniworld sidewalk and object pickup are two environments selected for our experiments.

In the Miniworld sidewalk environment, the agent must learn the policy of walking to a point where a cube is located while staying on the sidewalk. The environment rewards the agent when reaching the object and terminates the episode when the agent walks outside the sidewalk. Miniworld sidewalk is illustrated in the center image in Fig. 5. The agent learns to recognize to stay towards the right of safety cones and navigate straight to the cube. The RL agent must reach the cube within a specified maximum number of 250 steps.

Miniworld Object Pickup is the most complex environment of the three simulators. We select this environment to measure the performance of our models in complex reinforcement learning environments with complex tasks to

recognize different objects and navigate to collect them. In object pickup, the RL agent must learn the policy of picking five objects that may include cube, key, and sphere as seen from the right image in Fig. 5 in a single room by navigating towards them. The agent gets a +1 reward for collecting each object. The episode terminates when either all objects are collected or when the limit of 400 steps is reached.

The dataset used for training and testing the autoencoder network in Miniworld sidewalk environment consists of 20,000, 3-channel images with a size of $120 \times 120 \times 3$. However, 18,000 images are used for the training set and 2000 for the testing set. For the Miniworld Object Pickup environment, a dataset of 28000, 3-channel images with a size of $120 \times 120 \times 3$ is collected to train the autoencoder network and 2000 images are used in testing the autoencoder. Part of the dataset was gathered by random simulation of RL agent and rest by the human expert carefully navigating in place of RL agent in the environment/simulator.

VI. EXPERIMENTAL RESULTS

This section explains the experiments that we performed to reach the optimum configuration in terms of accuracy and model size. The autoencoder network is trained multiple times with different hyperparameters such as the number of layers, code-word size, number of epochs, and the quality of the reconstructed image is evaluated in terms of PSNR, MSE, and model size. Also, different filter sets are chosen to determine the best number of parameters for learning. Once the optimum configuration established for each case through design optimization, the base configuration is quantized to the binary and ternary neural network to calculate the performance of the RL agent as the basis of comparison of all cases.

A. DESIGN OPTIMIZATION

The Architecture presented in [4] is selected as a baseline for the experiments. Afterward, the network is modified to enhance the MSE and PSNR, two factors of merit that illustrates the accuracy of the reconstructed image. Hyperparameters such as the number of layers, code-word size, and number of epochs are manipulated to reach the best possible result.

It is observed that increasing the number of layers increase the accuracy of the autoencoder till a certain point. We Augmented the number of layers until no further accuracy could be gained through layer augmentation. The best accuracy is achieved when we have four convolution layers followed by a fully connected layer. The stride of 2 down samples the feature map, similar to the max-pooling operation. Because of the image size, no more convolution is possible after the fourth one.

Optimizing the number of filters is another way of increasing the accuracy of the model. Therefore, when implementing the compressed neural network such as Binary neural networks (BNNs) and Ternary neural networks (TNNs) the number of the filters is doubled (2.0 BNN and 2.0 TNN) and

quadrupled (4.0 BNN and 4.0 TNN) to increase the number of parameters and observe the potential increase in the aggregated reward from the RL agent. The parameter that affects the accuracy most of all is the latent representation size. Simulations shows that increasing the latent representation enhances the PSNR and MSE, significantly, while keeping the number of parameters and computations in a reasonable range.

Finally, accuracy is optimized in terms of the number of epochs. Each epoch is a single forward and backward trial to learn the model. Like all other types of neural networks, after a certain epoch, the accuracy of the model drops because of overfitting. For the base configuration, 50 epochs gave the best PSNR and MSE results. As the number of filters and latent representation size increase the optimum number of epochs also increases accordingly. In full-precision models, ReLu is used as activation function whereas, in BNN and TNN configurations binary tanh and ternary tanh [32] are used as activation for each layer.

B. BINARY AND TERNARY NEURAL NETWORKS PERFORMANCE

For evaluating the performance of the binary and ternary models, the best model parameters of each autoencoder model are chosen. These models are tested for image reconstruction and then paired with the RL agent to measure the reward achieved in each environment.

1) CASE STUDY 1: DONKEYCAR ENVIRONMENT

Each configuration in this environment is trained for 10,000 episode length with each track limiting to 600 episode length and tested with 5 random tracks. The average of these 5 tests is considered as episode length. Fig. 6 shows the reconstructed images extracted from the DonkeyCar simulator for full-precision, BNN, and TNN configurations. The baseline full-precision model with 64 codewords achieves the highest MSE and PSNR which results in the RL agent to perform better and achieve the highest episode length. The baseline binary configuration shows mediocre performance in reconstructing the image, so increasing the latent representation size increases the PSNR and MSE. But even with a codewords of 1,024, the RL agent with binarized autoencoder has the least episode length. The ternary model, on the other hand, performs almost as good as the full-precision model with 256 codewords and the results are more acceptable. Table 2 shows the model size for each cases. It can be understood that by increasing the model parameters we can improve the performance of the BNN. But, the same accuracy can be achieved by the TNN models with lesser parameters and computation cost since TNN has a much better representation of the data than BNN.

2) CASE STUDY 2: MINIWORLD ENVIRONMENT

As mentioned in the previous section, the Miniworld environment is a more complex setting compared to the DonkeyCar.

TABLE 2. Episode length (reward), PSNR and MSE comparison between different configurations in DonkeyCar environment.

Config.	Latent Space	Operations (Million)	Model Size (KB)	MSE	PSNR	Episode length
Full Precision	64	19.9	1,476	0.0003	34.1	560
1.0 BNN	1,024	23.0	421	0.0031	25.4	194
1.0 TNN	256	20.6	242	0.0010	29.9	490

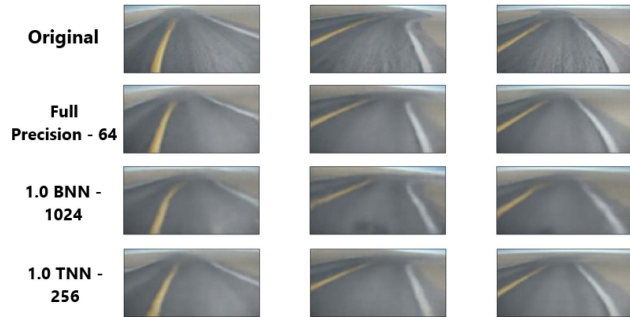


FIGURE 6. Comparison of autoencoder reconstructed images with the original image for DonkeyCar environment. TNN configuration has better quality of reconstruction compared to BNN.

TABLE 3. Reward, PSNR and MSE comparison between different configurations in Miniworld sidewalk environment.

Config.	Latent Space	Operations (Million)	Model Size (KB)	MSE	PSNR	Average Reward
Full Precision	64	19.9	1,476	0.0011	30.9	94.3
2.0 BNN	1,024	46.9	884	0.0038	25.5	8.7
4.0 BNN	1,024	92.2	1,937	0.0036	25.6	69.3
2.0 TNN	256	41.2	569	0.0014	29.4	94.0

In this environment, one forward movement is considered a single step, similar to the episode in the DonkeyCar environment. As we move towards more complex environments, the binarized configurations fail to keep up with the TNNs in terms of image reconstruction and the RL reward. Miniworld sidewalk environments RL agent is trained for 10,000 steps with a maximum of 250 steps for each episode. After 250 steps, if the RL agent has not completed the task, it earns no reward, and the next simulation starts. Fig. 7 shows the reconstructed images in autoencoder. Table 3 shows the summary of performance in the Miniworld sidewalk environment. The trend of BNN configurations achieving more profound reward even with double and quadruple filters and 1,024 latent representation size is evident: Due to further expressiveness (having zero values) in TNNs, the 2.0 TNN is two times larger than the 2.0 BNN, but having 85.3% higher accuracy/reward for the Sidewalk environment.

The third and most complex environment considered is Miniworld Object Pickup in which we performed exhaustive simulations. This environment is considered for autoencoder hardware implementation and results. It comprises of more complex environment setting and elaborate RL agent tasks. In this simulation, the accuracy of the information in the latent representation must be in a way that, the RL agent learns to pick up 5 different objects in one episode. Each episode is constrained to 400 steps within which the RL

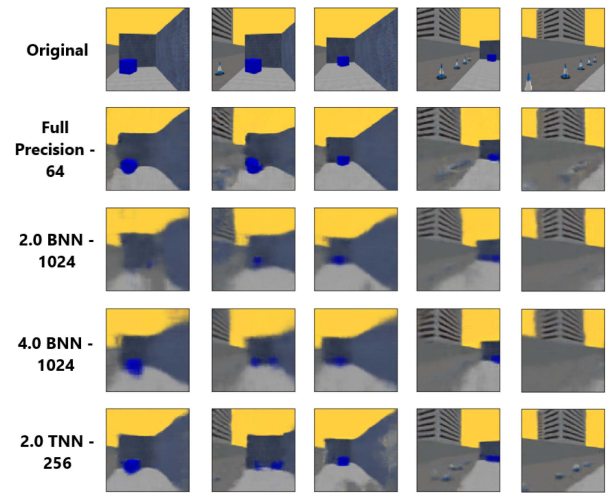


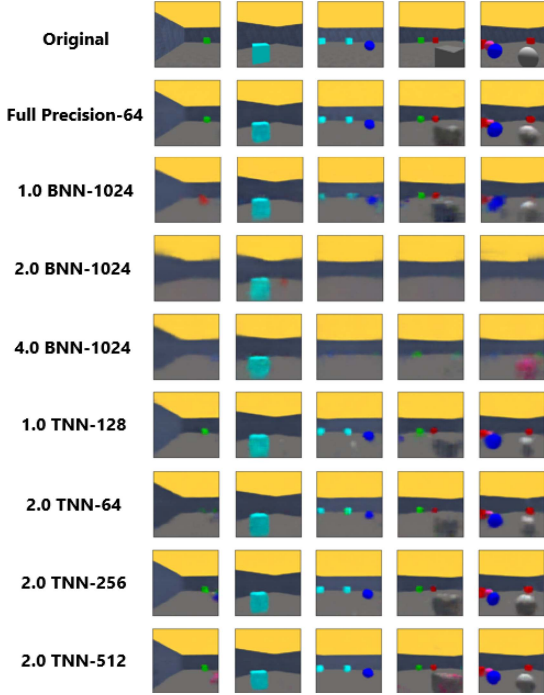
FIGURE 7. Comparison of autoencoder reconstructed images with original image for Miniworld sidewalk environment. Reconstructed images from 2.0 BNN-1024 configuration is highly distorted compared to TNN configurations.

agent must pick up all objects. If the RL agent fails to pick up all objects, then the number of objects agent picked up in that episode is the reward earned. Each neural network configuration is trained for 2 million steps and a model is saved at every 50,000 steps. For each saved model 50 episodes of the evaluation were performed and averaged to get the reward. Since, the RL trains based on the Monte Carlo method, we ran 5 such training and testing simulations involving 2 million steps to collect the data exhaustively. Fig. 9 shows the average reward from 5 evaluation runs for some configurations over 2 million steps. It can be seen that all TNN configurations follow the reward trend achieved by full-precision-64 configuration. 1.0 BNN-1024 takes almost 2 million episodes to reach the reward accuracy that TNNs achieve in just 250,000 steps. From Table 4, the full-precision base configuration in the object pickup environment reaches an average reward of 84.88% compared to 94.3 in the sidewalk environment. The reason for this is that the latter environment is more complex and we average over more train and test runs. As explained previously, due to expressiveness (having zero values) in TNNs, even with smaller latent space 1.0 TNN 128 compared with 1.0 BNN 1024 is smaller in model size by 2.9 \times and achieves 10% higher average reward. As seen from the results of this environment, TNNs are better in autoencoder data representation compared to BNNs. Fig. 8 shows the comparison of reconstructed images in autoencoder with the original image.

Based on the experimental analysis done from the Tables 2, 3 and 4 it is apparent that TNN configurations perform better than BNNs. Even though increasing the latent representation size and the number of filters increases the average reward for BNNs, it won't be deployable in Embedded FPGA such as Artix7-100T as shown in Fig. 10. The Artix7-100T FPGA has 4,860Kb of BRAM so, BNNs that perform marginally better require more memory. For Hardware implementation, 1.0 TNN-128 is considered since

TABLE 4. Reward, PSNR and MSE comparison between different configuration in Miniworld object pickup.

Config.	Latent Space	Operations (Million)	Model Size (KB)	MSE	PSNR	Average Reward
Full Precision	64	19.9	1,476	0.0006	32.0	84.8
1.0 BNN	1,024	23.0	421	0.0034	25.5	72.8
2.0 BNN	1,024	46.9	484	0.0091	21.5	52.2
4.0 BNN	1,024	92.2	1,937	0.0059	23.4	54.1
1.0 TNN	128	20.2	142	0.0021	27.6	82.0
2.0 TNN	64	39.9	269	0.0022	27.8	75.1
2.0 TNN	256	41.2	569	0.0013	29.9	82.3
2.0 TNN	512	42.8	969	0.0010	31.2	82.5

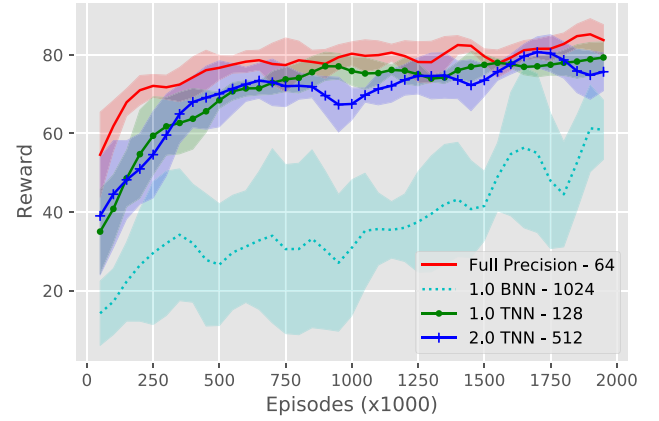
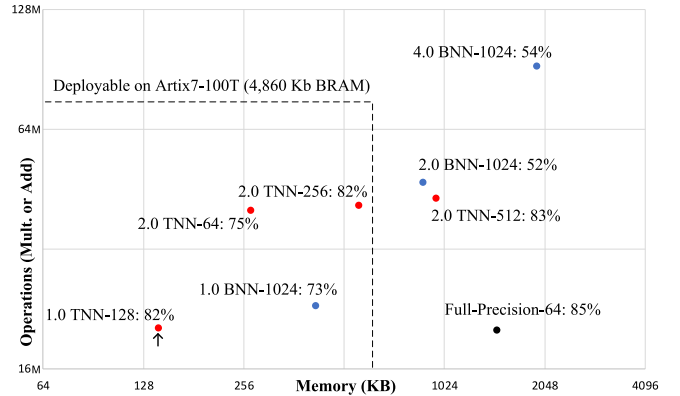
**FIGURE 8.** Comparison of autoencoder reconstructed images with original image for Miniworld Object Pickup environment. Due to over fitting BNN configurations with large number of parameters has missing objects whereas, almost all reconstructions from TNN configurations has retain objects.

it has a reward of 82% compared to 82.56% of 2.0 TNN-512 as it is outside the deployable region.

VII. ACCELERATOR ARCHITECTURE

The hardware accelerator for autoencoder is explored with fundamental aspects in consideration such as parallel computation and efficient memory sharing. The main objectives of the hardware architecture design are meeting the latency requirement, ability to fit the model in a small area, and feasibility of being fully reconfigurable as the hardware needs to be implemented for different applications having different requirements.

Fig. 12 depicts the autoencoder hardware architecture. The hardware is designed with configuration 1.0 TNN-128 in consideration which is explained in Section VI. The main modules of autoencoder consist of the following.

**FIGURE 9.** RL agent rewards against the validation of training episodes for selected configurations in Table IV. TNN configuration had only less than a 2% reward drop from the full-precision model compared to a 12% drop in BNN configuration. So, TNN configurations are considered for hardware implementation.**FIGURE 10.** Exploring the best model that yields the highest reward, while meeting the deploying constraints: FPGA BRAM size, and time/operation count to output. 1.0 TNN-128 is considered for hardware implementation since 2.0 TNN-512 is outside the deployable region.

- Convolution** consists of control unit and address generator which performs 2D convolution to fetch data into the PE array.
- Fully-connected** which is similar to convolution and also consists of control unit and address generator that performs matrix vector multiplication operations by fetching data into the PE array.
- PE array** preforms MAC operations using Ternary Multiplication Unit (TMU). This module also consists of ternary tanh activation function.
- Auto encoder top** controls data path logic to PE array and control logic to Convolution and Fully connected modules.

The autoencoder finite-state machine (FSM) logic controls the address generation modules such as convolution and fully-connected. The convolution block is active for the first 4 of convolutional layers and the fully-connected layer is activated in the end for generating the address of the last layer. The generated address is discreetly sent to the on-chip input memory and the appropriate data is retrieved. In each

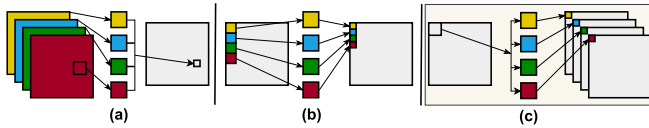


FIGURE 11. (a) Input channel tiling. (b) Image patch tiling. (c) Output channel tiling. Examination of various parallel tiling procedures for convolutional layers. Output channel tiling has the least memory communication conflict and hence chosen as tiling scheme for proposed hardware.

memory location, N ternary values are packed to accelerate parallel computation. Every ternary value is 2-bit in width as explained in Section IV. The packed data is then sent to the PE array to perform MAC operations. In every PE, the feature map and filter data is split into N ternary values and multiplied in ternary multipliers concurrently. There is no filter cache or buffer as seen in the previous compressed neural network hardware architectures [33] which contributes to memory size since it is critical in EdgeAI embedded hardware devices with on-chip memory. The output from the add block in PE is M -bit which is $\lceil \log_2(2N) + 1 \rceil$ -bit and is accumulated in the accumulator. The accumulated output is activated by ternary tanh after which the activate output from all PEs is packed and stored in the output memory. Resource sharing is carefully taken into consideration such that the major logic of every PE is only a pipeline of multiplier, adder, and an accumulator to satisfy the equation (3). Once the convolution is completed the packed data is loaded back into Feature Map memory for the next layer of computation.

In [34] various methods of parallelism that can be used to accelerate convolutional layers are explored. The basic process for the three tiling methods is shown in Fig. 11. The first method is referred to as input channel tiling, where for a given feature map multiple input feature map channels are convoluted collaterally. In the second method, which is referred to as image patch tiling, convolution is performed by the break input feature map into small image patches and perform convolution on all patches simultaneously. In the third method, output channel tiling, for a given input channel multiple filters or input channels are convoluted in parallel.

Using the rooftop model these three methods were tested in [35] to determine which provides the best throughput in FPGA. They find that output channel tiling has the best parallel I/O memory access and computation. Therefore, we primarily utilize output channel tiling. It has minimum dependency among parallel PEs. It can be seen from Fig. 12 that N packed values are read from the feature map memory but $n \times N$ values are read from the weight memory equal to the number of PEs in PE array. The output from each PE is packed and concatenated until N packed values are received from the PE array. As a result, there is no data dependency between any PEs as they operate only on the specific filter.

In our hardware, the performance is proportional to N packed values. Since these N packed values are split and

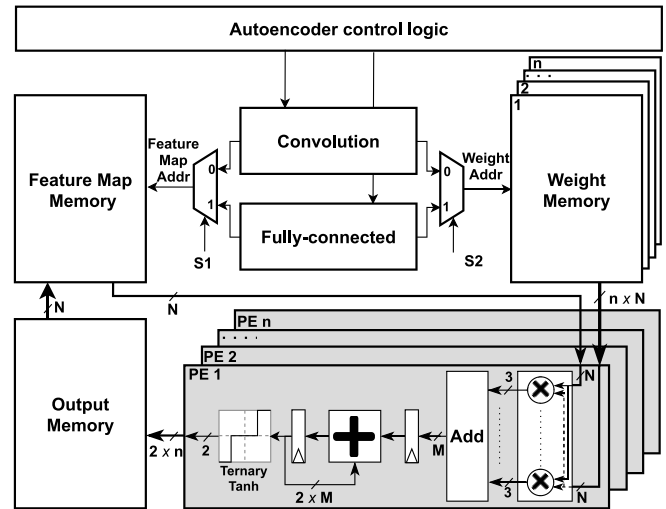


FIGURE 12. Block diagram of hardware architecture which consists of feature map memory and weight memory that are addressed by the convolution and fully-connected address control unit to fetch data to PE array. The processed data from the PE is stored temporarily in the output memory. Convolution and fully-connected blocks are controlled by autoencoder control logic.

tilled across n PEs this expedites the computation by n times. Clock frequency f is another parameter contributing to the performance of the Hardware. Therefore, in our hardware the performance is proportional to the three metrics:

$$\text{Performance} \propto f \times N \times n. \quad (5)$$

VIII. HARDWARE IMPLEMENTATION RESULTS AND ANALYSIS

To emulate the autoencoder performance on the proposed hardware architecture, we implemented and generated results on both FPGA and ASIC. The configuration 1.0 TNN-128 from Table 4 for Miniworld pickup object environment is implemented on hardware because it achieves the desired reward when tested with the RL agent and can fit in low-power embedded FPGA. All the results generated in this section are implemented for this configuration but, the autoencoder hardware architecture is reconfigurable to other TNN configurations as well.

A. FPGA IMPLEMENTATION

The autoencoder hardware architecture is described in Verilog HDL and can be reconfigured to the desired number of PEs and memory width depending on the application. We implemented autoencoder hardware on low-power Xilinx Artix-7 family with 135 Block RAMs (BRAMs), the smallest in the Artix-7 family. To ensure parity between the power and performance the hardware is pipelined to run at 100 MHz on Artix-7. Table 5 shows the various results such as utilization, power consumption, and latency obtained using Vivado Design Suite by Xilinx. For readability, hardware configurations with memory width greater than 16-bit are shown in Table 5. However, for plotting energy and power-performance graphs, all the memory width implementations

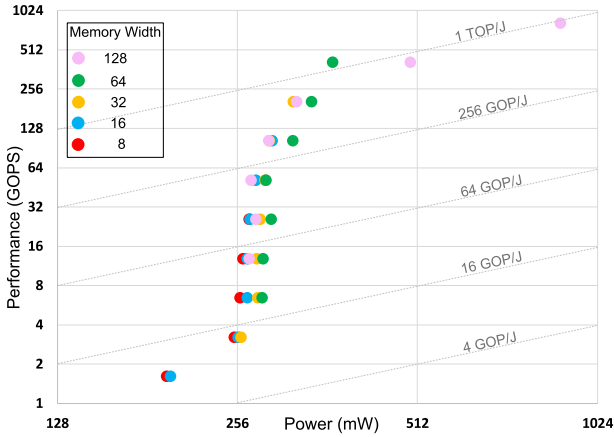


FIGURE 13. A scatter plot illustrating performance-power zones where different AE hardware configurations lie in. The AE hardware is scalable up to 1 TOP/J in FPGA implementation at 100MHz clock frequency.

are considered. Our hardware with the most parallel configuration can process 1350 FPS with as little as 0.66 mJ energy consumption.

Fig. 13 shows the scatter plot of hardware energy efficiency trade-off between performance and power. Our hardware is capable of reaching 1 TOP/J with 64-bit width memory and 64 PEs. Fig. 14 demonstrates the impact of increasing the number of PEs and memory width on energy consumption. The graph shows that wider memories are energy-efficient. Although, when memories reach 128-bit width the energy consumption starts increasing. As the number of PEs increases, the latency decreases for the same operating frequency. Since energy is the total power consumed per unit time it tends to decrease as latency decreases. The hardware in this article is targeted for deployment in embedded devices, the autoencoder hardware has to meet the latency of 30 FPS which is the average number of frames an embedded video camera can generate per second [36]. All the hardware configurations except the one with a data width of 8 and single PE meets 33.34 ms of latency. The hardware configuration with 32-bit memory width and 64 PEs is the most energy-efficient FPGA implementation with 250 μ J of energy consumption and therefore is the best hardware configuration for FPGA deployment.

B. ASIC IMPLEMENTATION

To further reduce the overall power consumption for inference of autoencoder TNN hardware, an Application Specific Integrated Circuit (ASIC) is implemented at the post layout level in 14nm FinFET Educational Design Kit (EDK) [37] provided by Synopsys with 0.8V power supply. A standard cell register transfer level (RTL) to post-layout implementation flow using Synopsys Design Compiler (DC), and Integrated Circuit Compiler (ICC) II is utilized to design the ASIC. We choose the most energy-efficient hardware configuration with memory width 32 and 64 PEs for ASIC

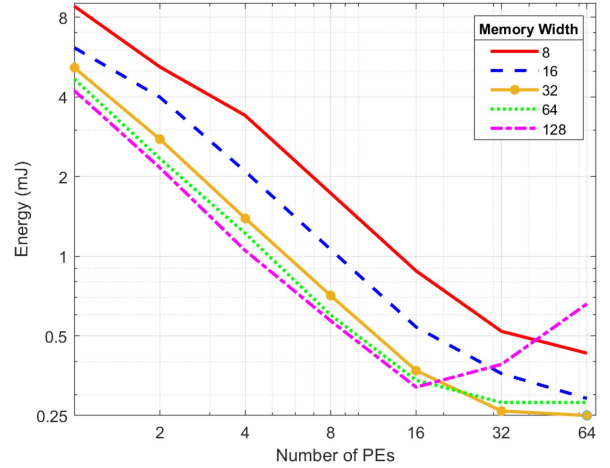


FIGURE 14. The logarithmic plot, showing the impact of the increasing number of PEs and memory width on energy consumption. The circle symbol marked configuration with 32-bit memory width and 64 PEs has the least energy consumption.

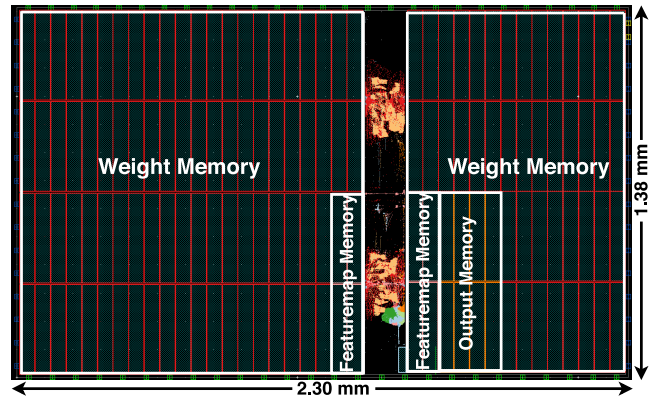


FIGURE 15. Post-layout view of autoencoder ASIC implementation in 14 nm EDK. Routing and power mesh visibility is turn off.

design. Since this configuration meets the latency requirement of 30 FPS on FPGA, the clock frequency has been reduced to 100 MHz to reduce the power consumption in ASIC design. Since the 7-series FPGA's are 28 nm, and the ASIC design that we used is 14nm, we see a significant overall power reduction of 65 \times and improves the energy efficiency by 64 \times . Of course, the reduced static power consumption of ASIC also contributes to a reduction in overall power consumption. The ASIC layout for the autoencoder hardware architecture is shown in Fig. 15. For visibility the VDD/GND power mesh is hidden from the screen capture view. The post layout implementation results are provided in Table 6.

C. COMPARISON WITH EXISTING WORKS

In this Section, we compare our ternary AE with the most related state-of-the-art TNN implementations: TNN models for image classification implemented on customized accelerators in ASIC and FPGA [28], [38] and an AE model with 8-bit precision for image compression [39].

TABLE 5. The implementation results of proposed TNN autoencoder 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.

Memory Width	# PE	LUT	Utilization FF DSP BRAM	Total Power(mW)	Latency(ms)	Perf(GOPS)	Energy(mJ)	Energy Eff(GOP/J)
32	1	906	492 10 73	260	19.8	3.2	5.15	12
	2	956	551 11 73	277	10	6.4	2.77	23
	4	1097	623 11 73	276	5	12.8	1.39	46
	8	1371	787 11 73	280	2.54	25.6	0.71	91
	16	1937	1061 11 73	285	1.3	51.2	0.37	180
	32	3428	1970 11 73	290	0.9	102.4	0.26	353
	64	6082	3466 11 73	318	0.8	204.8	0.25	644
64	1	1030	569 10 73	282	16.5	6.4	4.67	23
	2	1173	624 11 73	283	8.27	12.8	2.34	45
	4	1580	802 11 73	292	4.17	25.6	1.22	88
	8	2214	1049 11 69	286	2.1	51.2	0.6	179
	16	3792	1794 11 73	317	1	102.4	0.34	323
	32	6645	2974 11 73	341	0.8	204.8	0.28	601
	64	9968	3466 11 73	370	0.75	409.6	0.28	1,107
128	1	1217	703 10 73	268	15.76	12.8	4.22	48
	2	1499	775 11 73	275	7.87	25.6	2.16	93
	4	2097	999 11 67	270	3.9	51.2	1.06	190
	8	3468	1535 11 69	289	1.98	102.4	0.57	354
	16	5964	2483 11 73	322	1	204.8	0.32	636
	32	10953	4313 7 73	499	0.78	409.6	0.39	821
	64	23912	7951 7 73	889	0.74	819.2	0.66	922

TABLE 6. Post place and route results of the proposed TNN autoencoder hardware accelerator with 32-bit memory width and 64 PEs in ASIC 14nm technology.

Hardware Metrics	ASIC results
Core Area (mm ²)	3.15
#PEs	64
#SRAM Blocks each 512×32-bit	144
Clock Frequency (MHz)	100
Static Power (mW)	2.0
Dynamic Power (mW)	2.8
Total power (mW)	4.8
Performance (GOPS)	204.8
1.0 TNN-128 Deployed	Implementation results
# Operations (Million)	20.2
Model Size (KB)	142
Throughput(FPS)	1,250
Latency (ms)	0.8
Energy (μJ)	3.9
Energy Eff (TOP/J)	42.3

In [28] and [38], the TNNs are optimized for resource-efficiency and performance and use benchmarking datasets such as CIFAR100, SVHN, and GTSRB. Since both works target hardware implementation for the performance, they run at higher clock speeds compared to our hardware. The authors have generated several FPGA and ASIC implementation results. Therefore, we only consider those hardware results which are the most energy efficient, since the energy efficiency is generally a metric that reflects the quality of hardware design. The comparison results are summarized in Table 7. In summary, our FPGA implementation is 5× and 6.8× more energy-efficient compared to the FPGA implementation in [28] and [38] respectively. Since in [28] and [38] the efficiency metric is reported in FPS/W for two designs CNN-64 and CNN-128, we calculated the latency based on the reported throughput, the performance based on the model complexity, and the energy efficiency in TOP/J based on the calculated performance and the reported power

TABLE 7. Comparison of the proposed autoencoder hardware with state-of-the-art TNN and autoencoder related works.

Metrics	CNN-64 [28]	GTSRB [38]	[39]	This work	SVHN [28]	This work
Compression	TNN	TNN	8-bit	TNN	TNN	TNN
Input size	32×32	32×32	256×256	120×120	32×32	120×120
Platform	FPGA VC709	FPGA VC709	FPGA KCU105	FPGA Artix-7	ASIC 28nm	ASIC 14nm
Frequency (MHz)	200	250	150	100	500	100
Power (W)	4.8	6.64	2.76	0.37	0.22	0.005
Latency (ms)	0.29	0.04	15.73	0.75	0.13	0.8
Energy (mJ)	1.4	0.27	43.41	0.28	0.03	0.004
Performance (TOPS)	1.05	1.05	0.07	0.41	2.1	0.20
Energy Eff. (TOP/J)	0.22	0.16	0.26	1.1	9.5	42.3
Area (mm ²)	-	-	-	-	1.79	3.18

for comparison with our work. Having scaled our fabrication technology to that of [28] for the ASIC implementation, we observe 2.2× (4.4× with no scaling) improvement in efficiency compared to the ternarized CNN-64 implementation for SVHN in [28].

Compared to the convolutional AE with 8-bit precision model [39], our FPGA implementation with ternarized model is 4.2× more energy efficient.

IX. CONCLUSION

This article presents a scalable energy-efficient hardware architecture for compressed neural network autoencoders. The autoencoders in our work are employed to represent the high dimensional image data into small latent representations that feeds the Reinforcement Learning agents to interact with the environment to achieve a task. First, a baseline full-precision autoencoder is pre-trained using the data collected from the RL environment simulators. Next, the pre-trained autoencoder model is paired with a two-layer RL agent which thereafter is trained and tested in the simulator. This process

is replicated by ternarizing and binarizing the full-precision neural network of the autoencoder to measure the reward achieved by the RL agent. Our experiments with variant full-precision and binarized and ternarized AE models show that TNNs outperform BNNs in terms of RL's reward as well as number of parameters. We support the consistency of TNNs in our experimental setup for three environments including Donkeycar, Miniworld sidewalk, and Miniworld Object Pickup. In the Miniworld sidewalk, the 2.0 TNN-256 model's average reward is decreased by less than 1% whereas, 4.0 BNN-1024 is 25% less, compared to the full-precision baseline model. A similar exemplar is observed with experiments in Miniworld Object Pickup where 1.0 TNN-128 configuration achieves less than 3% average reward drop compared to a 12% drop in 1.0 BNN-1024 configuration. A custom low-power and energy-efficient hardware architecture is also designed for deployment of autoencoder in EdgeAI real-time systems to accelerate RL tasks and is implemented on Xilinx Artix-7 FPGA, with an average energy consumption of 250 μ J. The hardware is configurable to reach an efficiency of over 1 TOP/J on FPGA implementation. To further reduce the power consumption, autoencoder hardware is implemented in 14nm FinFET technology, which consumes $65\times$ less power compared to FPGA. Finally, we compared our work with state of the art TNN implementations where our hardware is $5\times$ and $4.4\times$ ($2.2\times$ if scaled) more energy-efficient on FPGA and ASIC implementations respectively.

ACKNOWLEDGMENT

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] Y. Zhu *et al.*, "Target-driven visual navigation in indoor scenes using deep reinforcement learning," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, 2017, pp. 3357–3364.
- [2] L. Gondara, "Medical image denoising using convolutional denoising autoencoders," in *Proc. IEEE 16th Int. Conf. Data Min. Workshops (ICDMW)*, 2016, pp. 241–246.
- [3] L. Theis, W. Shi, A. Cunningham, and F. Huszár, "Lossy image compression with compressive autoencoders," 2017. [Online]. Available: arXiv:1703.00395.
- [4] B. Prakash, M. Horton, N. R. Waytowich, W. D. Hairston, T. Oates, and T. Mohsenin, "On the use of deep autoencoders for efficient embedded reinforcement learning," in *Proc. Great Lakes Symp. VLSI*, 2019, pp. 507–512.
- [5] N. Kohl and P. Stone, "Policy gradient reinforcement learning for fast quadrupedal locomotion," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, vol. 3, 2004, pp. 2619–2624.
- [6] A. Y. Ng *et al.*, "Autonomous inverted helicopter flight via reinforcement learning," in *Experimental Robotics IX*. Berlin, Germany: Springer, 2006, pp. 363–372.
- [7] S. Singh, D. Litman, M. Kearns, and M. Walker, "Optimizing dialogue management with reinforcement learning: Experiments with the NJFun system," *J. Artif. Intell. Res.*, vol. 16, no. 1, pp. 105–133, 2002.
- [8] D. Silver *et al.*, "Mastering the game of Go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, 2017.
- [9] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.
- [10] A. A. Rusu, M. Vecerik, T. Rothörl, N. Heess, R. Pascanu, and R. Hadsell, "Sim-to-real robot learning from pixels with progressive nets," in *Proc. Conf. Robot Learn.*, 2017, pp. 262–270.
- [11] E. Tzeng *et al.*, "Adapting deep visuomotor representations with weak pairwise constraints," in *Algorithmic Foundations of Robotics XII*. Cham, Switzerland: Springer, 2020, pp. 688–703.
- [12] A. Kendall *et al.*, "Learning to drive in a day," in *Proc. Int. Conf. Robot. Autom. (ICRA)*, 2019, pp. 8248–8254.
- [13] W. J. Dally, Y. Turakhia, and S. Han, "Domain-specific hardware accelerators," *Commun. ACM*, vol. 63, no. 7, pp. 48–57, Jun. 2020.
- [14] M. Khatwani, W. D. Hairston, N. Waytowich, and T. Mohsenin, "A low complexity automated multi-channel EEG artifact detection using EEGNet," in *Proc. IEEE EMBS Conf. Neural Eng.*, 2019.
- [15] H.-A. Rashid, N. K. Manjunath, H. Paneliya, M. Hosseini, W. D. Hairston, and T. Mohsenin, "A low-power LSTM processor for multi-channel brain EEG artifact detection," in *Proc. 21st Int. Symp. Qual. Electron. Design (ISQED)*, 2020, pp. 105–110.
- [16] M. Hosseini, H. Ren, H. Rashid, A. Mazumder, B. Prakash, and T. Mohsenin, "Neural networks for pulmonary disease diagnosis using auditory and demographic information," in *Proc. 3rd ACM SIGKDD Int. Workshop Epidemiol. Meets Data Min. Knowl. Disc.*, 2020, pp. 1–5.
- [17] S. Li, C. Wu, H. Li, B. Li, Y. Wang, and Q. Qiu, "FPGA acceleration of recurrent neural network based language model," in *Proc. IEEE 23rd Annu. Int. Symp. Field Program. Custom Comput. Mach.*, 2015, pp. 111–118.
- [18] A. Shiri *et al.*, "Energy-efficient hardware for language guided reinforcement learning," in *Proc. Great Lakes Symp. VLSI*, 2020, pp. 131–136.
- [19] B. Prakash, N. R. Waytowich, A. Ganesan, T. Oates, and T. Mohsenin, "Guiding safe reinforcement learning policies using structured language constraints," in *Proc. SafeAI Workshop 34th AAAI Conf. Artif. Intell.*, 2020, pp. 153–161.
- [20] NVIDIA. *Nvidia Jetson TX2*. Accessed: Oct. 2020. [Online]. Available: <https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-tx2/>
- [21] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: MIT press, 2018.
- [22] A. Raffin and R. Sokolov. (2019). *Learning to Drive Smoothly in Minutes*. [Online]. Available: <https://github.com/araffin/learning-to-drive-in-5-minutes/>
- [23] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2016, pp. 4107–4115.
- [24] M. Hosseini and T. Mohsenin, "Binary precision neural network manycore accelerator," *ACM J. Emerg. Technol. Comput. Syst.*, 2020.
- [25] M. Courbariaux, Y. Bengio, and J.-P. David, "BinaryConnect: Training deep neural networks with binary weights during propagations," in *Proc. Adv. Neural Inf. Process. Syst.*, 2015, pp. 3123–3131.
- [26] F. Li, B. Zhang, and B. Liu, "Ternary weight networks," 2016. [Online]. Available: arXiv:1605.04711.
- [27] C. Zhu, S. Han, H. Mao, and W. J. Dally, "Trained ternary quantization," 2016. [Online]. Available: arXiv:1612.01064.
- [28] H. Alemdar, V. Leroy, A. Prost-Boucle, and F. Pétrot, "Ternary neural networks for resource-efficient AI applications," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, 2017, pp. 2547–2554.
- [29] L. Deng, P. Jiao, J. Pei, Z. Wu, and G. Li, "GXNOR-Net: Training deep neural networks with ternary weights and activations without full-precision memory under a unified discretization framework," *Neural Netw.*, vol. 100, pp. 49–58, Apr. 2018.
- [30] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.
- [31] M. Chevalier-Boisvert. (2018). *Gym-Miniworld Environment for Openai Gym*. [Online]. Available: <https://github.com/maximecb/gym-miniworld>
- [32] G. D. Guglielmo *et al.*, "Compressing deep neural networks on FPGAs to binary and ternary precision with HLS4ML," 2020. [Online]. Available: arXiv:2003.06308.

- [33] M. Hosseini, H. Paneliya, U. Kallakuri, M. Khatwani, and T. Mohsenin, "Minimizing classification energy of binarized neural network inference for wearable devices," in *Proc. 20th Int. Symp. Qual. Electron. Design (ISQED)*, 2019, pp. 259–264.
- [34] A. Page, A. Jafari, C. Shea, and T. Mohsenin, "SPARCNet: A hardware accelerator for efficient deployment of sparse convolutional networks," *J. Emerg. Technol. Comput. Syst.*, vol. 13, no. 3, pp. 1–31, May 2017. [Online]. Available: <http://doi.acm.org/10.1145/3005448>
- [35] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays*, 2015, pp. 161–170.
- [36] A. Kurniawan, A. Ramadlan, and E. M. Yuniarno, "Speed monitoring for multiple vehicle using closed circuit television (CCTV) camera," in *Proc. Int. Conf. Comput. Eng. Netw. Intell. Multimedia (CENIM)*, 2018, pp. 88–93.
- [37] V. Melikyan, M. Martirosyan, A. Melikyan, and G. Piliposyan, "14nm educational design kit: Capabilities deployment and future," in *Proc. Small Syst. Simulat. Symp.*, 2018, pp. 37–41.
- [38] A. Prost-Boucle, A. Bourge, F. Pétrot, H. Alemdar, N. Caldwell, and V. Leroy, "Scalable high-performance architecture for convolutional ternary neural networks on FPGA," in *Proc. 27th Int. Conf. Field Program. Logic Appl. (FPL)*, 2017, pp. 1–7.
- [39] W. Zhao, Z. Jia, X. Wei, and H. Wang, "An FPGA implementation of a convolutional auto-encoder," *Appl. Sci.*, vol. 8, no. 4, p. 504, 2018.
- [40] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "Deep reinforcement learning: A brief survey," *IEEE Signal Process. Mag.*, vol. 34, no. 6, pp. 26–38, Nov. 2017.
- [41] T. Wang, C. Wang, X. Zhou, and H. Chen, "A survey of FPGA based deep learning accelerators: Challenges and opportunities," 2018. [Online]. Available: [arXiv:1901.04988](https://arxiv.org/abs/1901.04988).
- [42] S. M. P. Dinakarrao, A. Joseph, A. Haridass, M. Shafique, J. Henkel, and H. Homayoun, "Application and thermal-reliability-aware reinforcement learning based multi-core power management," *ACM J. Emerg. Technol. Comput. Syst. (JETC)*, vol. 15, no. 4, pp. 1–19, 2019.
- [43] H. Hantao, P. D. S. Manoj, D. Xu, H. Yu, and Z. Hao, "Reinforcement learning based self-adaptive voltage-swing adjustment of 2.5 DI/Os for many-core microprocessor and memory communication," in *Proc. IEEE/ACM Int. Conf. Comput. Aided Design (ICCAD)*, 2014, pp. 224–229.