2026 IEEE International Conference on Robotics and Automation (ICRA 2026)
June 1-5, 2026. Vienna, Austria

Quantization of DRL Models for Embedded Microcontrollers

Peter Böhm[1,2], Archie C. Chapman[1], Peyman Moghadam[2], Pauline Pounds[1], and Jen Jen Chung[1]

Abstract— **For Deep Reinforcement Learning (DRL) models to deliver actual utility, they must function within production environments, which often lack the extensive computational resources of training environments. This requirement for dedicated GPU resources is not economically feasible and can be especially prohibitive in low-cost robotic contexts. Neural network quantization serves as a viable solution to these constraints. This technique aims to lessen computational and memory requirements, while maintaining performance. By reducing the precision of the DRL network weights and the network input (sensory observations), the deployment size can be compacted to fit within MCU class devices, while ensuring that inference operates at adequate frequencies. This paper investigates the impact of quantization on DRL policies and presents a quantization-friendly network architecture for the Soft Actor-Critic (SAC) and TD3 algorithms. We propose a streamlined actor network optimized for inference-only deployments and quantization, and integrate a GRU-based encoder into the DRL framework using a custom, quantization-compatible implementation. The changes enable both to be quantized to integer precision. We then deploy the quantized policies on a microcontroller-scale device (ESP32-S3) to control a low-cost quadrupedal robot using only proprioception and on-board inference.**

I. Introduction

An essential step in narrowing the simulation-to-reality gap lies in the effective deployment of trained models in tangible, real-world contexts. However, DRL, along with deep learning in general, is well-known for its substantial computational requirements; AI models need large memory and processing power to function optimally. This capacity is typically provided by high-performance processing units, such as CPUs and GPUs.

In real-world contexts, such as robotics or embedded systems, high-performance hardware is not always accessible or desirable. Energy efficiency, physical size, and cost constraints often prompt a more compact solution. This, in turn, requires deep learning models to be optimized to work efficiently on low-power, resource-constrained devices, such as microcontrollers. The process of model optimization and efficient deployment must be an integral part of the application of DRL models to real-world problems, bridging the gap between simulation and reality.

Major deep learning frameworks like TensorFlow and PyTorch provide quantization tools. However, those tools are limited to their respective native formats. This restriction also imposes a specific runtime that needs to be available during inference. An alternative is *Open Neural Network Exchange* (ONNX) that enables framework interoperability, enabling models trained in one framework to be transferred to another for inference. It supports optimized inference across diverse hardware platforms including CPUs, GPUs, and accelerators, ensuring consistent model representation through standardization. The portability of ONNX models makes them deployable on the cloud, desktop applications, and some more capable edge devices. Nevertheless, at present, there is no available ONNX runtime suitable for small microcontrollers such as the ESP32 or Cortex-M series.

[1]Peter Böhm, Pauline Pounds, Archie Chapman, and Jen Jen Chung are with the School of Electrical Engineering and Computer Science, The University of Queensland, QLD, 4072, Australia {p.bohm,archie.chapman,pauline.pounds, jenjen.chung}@uq.edu.au
[2]Peter Böhm and Peyman Moghadam are with CSIRO Robotics, DATA61, CSIRO, Australia. {peter.bohm,peyman.moghadam} @csiro.au

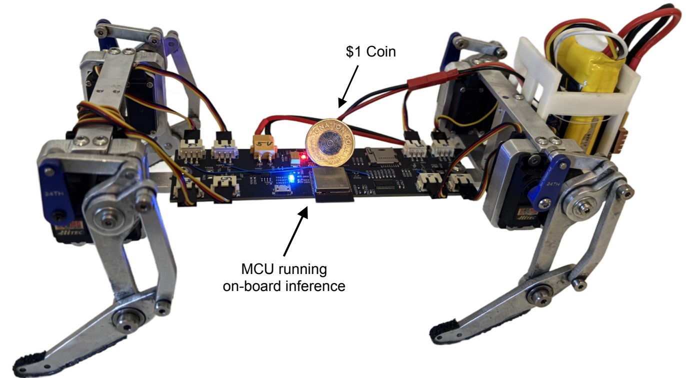


Fig. 1: Low-cost quadrupedal robot used in experiments, with an on-board quantized DRL model relying solely on proprioceptive observations.

Further, most of these tools are specifically designed for optimization of computer vision models, which limits their effectiveness or even feasibility for DRL applications. The most common obstacle is missing support for certain operations, such as ScatterND, that either breaks the quantization or prevents the model from being exported to a format suitable for deployment on an embedded microcontroller.

This same obstacle also affects quantization of recurrent neural networks, with major frameworks lacking built-in support for quantization of Gated Recurrent Units (GRU) [1].

Efficiently constructing deep neural networks requires a combination of techniques tailored to specific needs, as detailed in [2]. Crucial strategies include the application of quantization and reduced parameter precision, which can significantly decrease model size and computational complexity, achieved by lowering the bit-width of the numbers representing weights and activations in the neural network.

Though the topic of quantization in DRL is still relatively under-studied, promising indications have emerged from studies such as [3] and [4]. Both works demonstrate that

979-8-3315-8160-2/26/$31.00 ©2026 IEEE

policy models can be quantized down to 6-8 bit precision with minimal impact on performance across a number of simulation tasks and algorithms.

In this paper, we investigate the effects of quantization on DRL-trained policies. We propose a streamlined network architecture for the actor component of the Soft Actor-Critic (SAC) algorithm [5], optimized explicitly for inference-only deployments and quantization. To enhance both training efficiency and performance of the quantized models, we integrate a GRU-based observation encoder as described in [6]. Such GRU encoders have demonstrated improved DRL performance in noisy real-world conditions and multiple simulated environments. Additionally, recent work by Fujimoto et al. [7] showed that employing an MLP-based encoder to learn embeddings significantly enhances training efficiency in simulation settings.

Building upon these insights, we develop a quantization-friendly architecture that incorporates the GRU-based encoder into the DRL actor network. Effective quantization is achieved via our proposed implementation of the multi-layer GRU network, which is specifically adapted for compatibility with standard quantization tools, unlike implementations provided by mainstream deep learning frameworks.

We examine a range of quantization types across diverse platforms, including a full-scale computer, a resource-constrained Raspberry Pi 4, and a low-power processing unit (ESP32-S3) with limited resources. After training and quantizing our control policies, we deploy them to an embedded microcontroller (ESP32-S3) mounted on a low-cost quadrupedal robot. The microcontroller executes all inference locally, using only the robot's proprioceptive sensors for feedback, thereby enabling fully autonomous real-time control without external computation or exteroceptive inputs.

The structure of the paper is as follows: Section II discusses related work, followed by the technical preliminaries in Section III. Section IV introduces the streamlined and quantizable network architecture. The experimental setup is described in Section V and we report results in Section VI. Section VII summarises the conclusions of the work.

II. Related Work

Neural Network (NN) quantization reduces the computational and memory requirements of NNs while maintaining their performance. Various methods are employed to improve NN model efficiency while maintaining optimal accuracy and generalization trade-offs. These include improving NN design, pruning low-saliency neurons, using knowledge distillation, and quantization by reducing the precision of NN weights. Per Gholami [8], for the trained model parameters θ stored in floating point precision, the objective of quantization is to decrease the precision of both the parameters and intermediate activations while maintaining the model's generalization power/accuracy.

The effects of quantization in DRL settings have been explored in [3]. Post-training quantization (PTQ) experiments in Atari Arcade Learning [9] and several OpenAI Gym environments [10] reveal that the performance difference between 8-bit and 16-bit post-training quantization is negligible. This is attributed to the relatively narrow weight distribution of the policy, which allows 8 bits to capture the weight distribution with minimal error.

In their Quantization-Aware Training (QAT) experiments, the performance relative to the full-precision baseline remains consistent until 5/6-bit quantization, beyond which a decline in reward is observed. However, this study is limited to simulation experiments and deployment on a Raspberry Pi running a full operating system, which is capable of executing even full-scale policies.

For Long Short-Term Memory (LSTM) and GRU architectures, multiple quantization approaches have been proposed, such as binary, ternary, and quaternary-connect methods [11]. Furthermore, post-training quantization and quantization-aware training techniques have been developed specifically for recurrent neural network (RNN) architectures [12]. FastGRNN, introduced in [13], provides a compact, kilobyte-scale GRU implementation that integrates residual connections and utilizes low-rank, sparse, and quantized matrices. A limitation of these approaches is their reliance on highly customized implementations, making integration into larger neural networks impractical and precluding compatibility with standard quantization tools.

Despite the availability of numerous quantization tools, applying them to DRL models intended for embedded microcontroller deployment presents several challenges. Most existing tools are tailored primarily for computer vision and image processing, thereby lacking support for operations commonly employed in DRL neural networks. Even in scenarios where quantization is achievable, exporting the resulting quantized models for deployment on small embedded processors frequently proves difficult. Finally, a specific challenge arises from gated feature extraction, particularly in the quantization of the GRU network. To overcome these challenges, a streamlined implementation is required for the quantization of both DRL networks and GRU encoders.

III. Technical Preliminaries

A. Quantization

A quantization operator is used to map floating-point values to their quantized representations. The *uniform quantization* operator is commonly used in neural network quantization [14], [15]. Two options for the transformation function are: (i) *affine* $f(x) = S \cdot x + Z$, and (ii) *scale* $f(x) = S \cdot x$, where S is a scaling factor, and Z is a zero point. The quantization error is minimized by optimizing S and Z. For affine quantization they are given by:

$$S = \frac{2^b - 1}{\beta - \alpha},$$

$$Z = -\operatorname{round}(\beta \cdot S) - 2^{b-1},$$

where $[\alpha, \beta]$ is a clipping range used to bound real values, and b is the quantization bit width. For the real value of zero

to be representable, Z must be rounded to an integer value. The quantize operation is then defined as:

$$\text{clip}(r, l, u) \begin{cases} l, & r < l \\ r, & l \leq r \leq u \\ u, & r > u \end{cases}$$

$$Q(r) = \text{clip}(\text{round}(S \cdot r + Z), -2^{b-1}, 2^{b-1} - 1).$$

The quantized values $Q(r)$ can be converted back to the real values through *dequantization*:

$$\tilde{r} = \frac{1}{S}(Q(r) - Z).$$

Due to the rounding errors, $\tilde{r}$ will not exactly match r.

There are also non-uniform quantization methods [16]–[18]. Non-uniform quantization is a more effective way of capturing signal information by assigning bits and discretizing the range of parameters non-uniformly. However, it is challenging to implement on general computation hardware, so uniform quantization is currently the standard due to its simplicity and efficient mapping to hardware.

Depending on when the quantization happens, we can split the quantization methods into *Post-Training Quantization* (PTQ) and *Quantization Aware Training* (QAT). In this work, we apply PTQ, which involves quantizing an already trained network with little or no data, and no end-to-end training. On the other hand, QAT, requires retraining the neural networks with simulated quantization during the training process.

B. GRUs and Gated Feature Extraction

As derived in [1], the GRU uses *reset gate* r and *update gate* z that depend on the previous hidden state $h_{(t-1)}$. The gates are used to generate a *candidate hidden state* $\tilde{h}$; an activation function then computes the new hidden state h_t.

The activation of the j-th hidden unit is computed as follows [1]:

$$\begin{aligned} r_j &= \sigma\left([W_r\mathbf{x}]_j + \left[U_r\mathbf{h}_{\langle t-1\rangle}\right]_j\right) \\ z_j &= \sigma\left([W_z\mathbf{x}]_j + \left[U_z\mathbf{h}_{\langle t-1\rangle}\right]_j\right) \\ \tilde{h}_j^{\langle t\rangle} &= \phi\left([W_n\mathbf{x}]_j + \left[U_n\left(\mathbf{r} \odot \mathbf{h}_{\langle t-1\rangle}\right)\right]_j\right) \\ h_j^{\langle t\rangle} &= z_j \odot h_j^{\langle t-1\rangle} + (1 - z_j) \odot \tilde{h}_j^{\langle t\rangle} \end{aligned} \tag{1}$$

where σ is the logistic sigmoid function, ϕ is the tanh function, $\mathbf{x}$ and $\mathbf{h}_{t-1}$ are the input and the previous hidden state, W_r and U_r are weight matrices.

Gated feature extraction [6] involves encoding state observations using a GRU network that is separate from the DRL network. Specifically, *R*-type networks encode only the observations, whereas *RA*-type networks encode a combination of state observations and actions. The resulting encoding is then fed as an input to the DRL network. In this work, we evaluated policies trained via RA-SAC and RA-TD3 alongside those trained by the basic SAC and TD3 algorithms to demonstrate the need for quantizable GRU encoders in real-world policy deployments.

IV. Quantization

A. Quantizable GRU

Direct quantization of GRU networks is not currently supported by major machine learning frameworks such as PyTorch, TensorFlow, and ONNX. Importantly, GRU networks converted to ONNX do not explicitly unroll the hidden state, and since ONNX is a stateless framework, this means that out-of-the-box conversions will strictly result in an erroneous instantiation. Furthermore, existing quantization tools do not offer full functionality (i.e. not all operations have quantization support), thus, the resulting networks often remain unchanged in size and continue utilizing FP32 weights. Frameworks also differ in their computations of the candidate hidden state, arising primarily from whether the reset gate r is applied before or after matrix multiplication, leading to discrepancies in hidden state outputs given identical inputs. As an additional complication, Pytorch only provides access to its top level GRU module and does not expose the lower level operations. Thus, to ensure consistent quantization, we propose a novel implementation of the GRUCell using basic standardized operators, with which we can reconstruct the GRU module by cascading multiple GRUCells.

Our quantization method applies several techniques to improve computational performance and guarantee consistency in the network output regardless of the learning framework that's used. The first three equations in (1) involve six weights representing six hidden layers. To improve efficiency, the computation of the candidate hidden state $h_j^{\langle t\rangle}$ in the third equation can be rearranged such that the Hadamard product between r and the previous hidden state $h_{(t-1)}$ is performed after multiplication with the weight matrix[1]:

$$\tilde{h}_j^{\langle t\rangle} = \phi\left([W_n\mathbf{x}]_j + \left[\mathbf{r} \odot \left(U_n\mathbf{h}_{\langle t-1\rangle}\right)\right]_j\right). \tag{2}$$

After this change, there are two sets of weights and two inputs x_t and h_{t-1}. Specifically, W_r, W_z, W_n multiply the input x_t, while U_r, U_z, U_n multiply the previous hidden state h_{t-1}. For efficiency, some frameworks consolidate them into two fully connected layers.

Following the forward pass, the output of each layer is decomposed into three terms: $W_r\mathbf{x}, W_z\mathbf{x}, W_n\mathbf{x}$ and $U_r\mathbf{h}_{\langle t-1\rangle}, U_z\mathbf{h}_{\langle t-1\rangle}, U_n\mathbf{h}_{\langle t-1\rangle}$, which are subsequently used in equations (1) and (2). Different frameworks implement distinct mappings of these components, leading to disparate outputs for identical inputs. Our explicit definition ensures the portability of trained weights across different frameworks. This two layered network comprises the GRUCell and produces its hidden state as output.

The GRU module is constructed by cascading multiple GRUCells. In this configuration, the first cell receives the state observations as input (x_t), while each subsequent cell receives as input the hidden state produced by its predecessor. The module's output consists of a stack of hidden states from all constituent cells. This arrangement results in an

[1]This optimization is used by PyTorch, but is optional by Keras or TensorFlow, resulting in differing outputs for identical inputs.

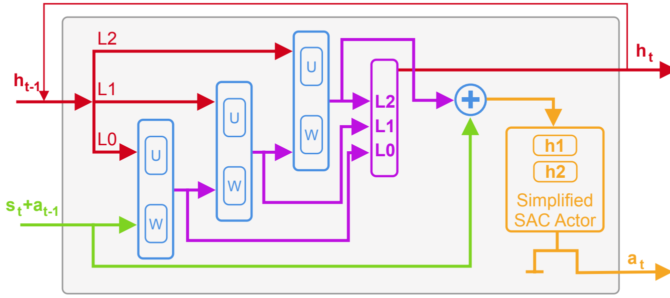


Fig. 2: Augmented RA-SAC network structure optimized for quantization. The hidden state is comprised of three layers ($L0$, $L1$, $L2$), each corresponding to one of the GRUCells. Within each GRUCell, there are two hidden layers (U and W). The state s combined with action a_{t-1} serves as an input to the initial GRUCell and is also added to the output of the final GRUCell. The resulting concatenation is then fed into the simplified SAC/TD3 actor.

implementation composed of only fully connected layers, elementwise operations, and activations, making it suitable for quantization with existing tools.

B. Quantized DRL

The quantizable GRU module can be incorporated as an encoder head to a DRL network. During the training of *RA*-based algorithms, the GRU encoder is situated outside the DRL network. This modular design enables the GRU to be replaced with alternative preprocessors or encoders, such as frame stacking, LSTM-based, or MLP-based encoders, and enables separate training of the encoder from the primary DRL network. However, to reduce data transfer and manipulation overhead between the encoder and the DRL network during inference, we embed the GRU within the DRL network. This integration requires modifications to the SAC/TD3 actor architecture. Since ONNX and quantized model inference on a microcontroller is stateless, the GRU's hidden state must be managed outside the network. To achieve this, we incorporate the hidden state as an additional network input, and the updated hidden state is appended to the output. The resulting network therefore has two inputs (state observations x_t and previous hidden state h_{t-1}) and two outputs (actions a_t and hidden state h_t).

The actor component of the SAC algorithm employs a squashed normal distribution with a $\tanh$ transform to maximize exploration entropy. The final layer outputs both the mean and standard deviation for each action's squashed normal distribution, resulting in an output size that is twice the dimensionality of the action space. During training, this distribution is sampled to determine the next action. In contrast, during inference only the means, which represent the actions, are required. To optimize the structure for quantization, we omit the standard deviations and the network only returns the means by discarding the second half of the final layer's output.

The final step involves loading the saved weights checkpoint. Due to the modifications made to the network structure, the trained weights from the original network are not directly compatible with the new network. Instead, we load the trained weights of the GRU and RA-TD3/RA-SAC networks into their respective original structures, and transfer them layer by layer to the unified network. The resulting network, shown in Fig. 2, is then exported to ONNX and used for quantization.

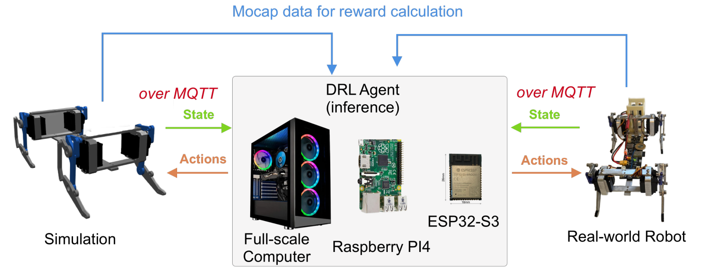


Fig. 3: We evaluate off-board inference across three platforms with decreasing computational resources: a full-scale computer, Raspberry PI4, and ESP32-S3.

We perform quantization on the ONNX model using the ESP-DL library, as our target MCU is the ESP32-S3. However, alternative frameworks, typically vendor-specific, can be employed for other target platforms, as most support the ONNX format. For comparison, we also apply quantization to the models using the native ONNX quantization tool.

V. Experimental Setup

To investigate the impact of quantization on DRL models, we compared the performance of the quantized models against their full-precision counterparts across three distinct setups: i) a full-scale computer, ii) a resource-constrained Raspberry Pi 4, and iii) an ESP32-S3 MCU. The setup is shown in Fig. 3. We present a performance comparison on both a real-world quadrupedal robot and a simulated environment to ensure the reproducibility of our results.

A. Quadrupedal Locomotion Task

To validate the scalability of our architecture to complex problems and continuous action spaces, we evaluated its performance on a challenging quadrupedal locomotion task. The robot, depicted in Fig. 1 and originally presented as a DRL testbed in [6], is actuated by 8 servos, with each leg controlled by two servos. The servos are controlled by an onboard ESP32-S3. The action space consists of 8 continuous inputs, corresponding to 2 servos per leg.

The observation space comprises a combination of sensor readings, including orientation (yaw, pitch, and roll) and acceleration in the z-axis of the quadruped's base frame, the position of each servo, the current drawn by each leg, and total current drawn. The reward function is given by:

$$
\begin{aligned}
R = \Delta d &+ 0.5\exp(-10|yaw_{error}|) + r_{velocity} \\
&+ 0.1\exp(-1.5|current - current_{cut\text{-}off}|) \\
&- 0.025(current_{FL}) - 0.025(current_{FR}) \\
&- 0.025(current_{BL}) - 0.025(current_{BR}) \\
&- 0.1|pitch| - 0.1|roll| - 0.025|acc_z - 9.8| \\
&- 0.025|acc_x| - 0.025|acc_y| - 0.025(action_smoothness),
\end{aligned}
$$

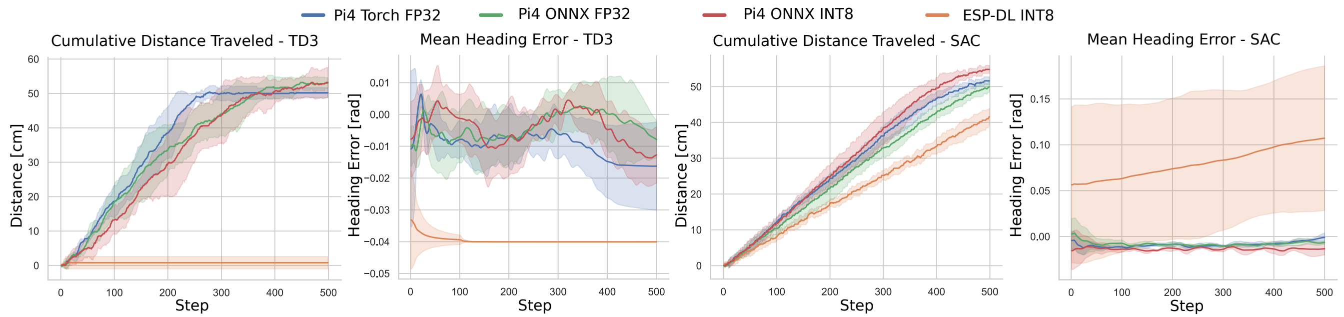


Fig. 4: Simulation comparison of cumulative distance traveled and mean heading error for TD3 and SAC without the GRU encoder. The ESP-DL quantized TD3 fails to move, whereas the ESP-DL quantized SAC advances slowly without correcting its heading. Distance traveled saturates at $\approx 50\,\mathrm{cm}$, corresponding to the start of the obstacle.

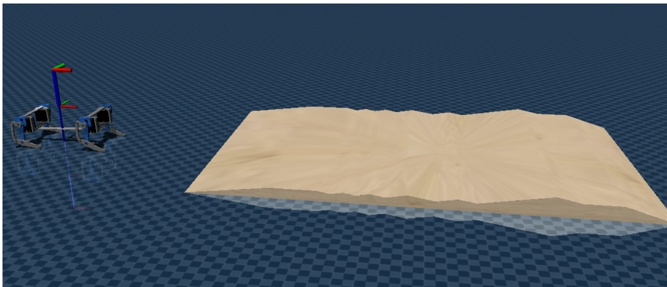


Fig. 5: Simulated environment used for training and evaluation. The robot must traverse a compound obstacle comprising an ascent, an uneven terrain section, and a descent; flat approach and exit planes provide easier traversal and baseline reward accumulation.

where Δd is the distance traveled in the direction specified by the control signal, $r_{velocity} = 2(\exp(-10|\Delta d_{target} - \Delta d|)$, and $action_smoothness = (a_t - a_{t-1})^2 + (a_t - 2a_{t-1} + a_{t-2})^2$ [19]. Per-leg current penalty is only applied if the current exceeds $current_{cut\text{-}off}/4$.

This reward encourages forward motion at a specific velocity, while marginally penalizing deviation from a level orientation, as well as penalizing excessive acceleration to reduce bouncing and improve motion smoothness.

We used a high performance computer equipped with an AMD Ryzen 5 CPU and 64 GB of RAM and RTX2080Ti GPU to train four sets of locomotion policies, one for each of the following DRL algorithms: (i) TD3, (ii) SAC, (iii) RA-TD3, and (iv) RA-SAC. Training was conducted in a simulated environment[2], with each set comprising five independent runs with different random seeds. For each set, we selected the policy that performed best on the real robot for the subsequent quantization experiments. We compared our quantization method (INT8) against the original trained model (FP32 Torch) and its FP32 counterpart exported to the ONNX format. We use the built-in ONNX quantization tool for generating policies deployed on the full-scale computer and the Raspberry Pi, while the ESP-DL library was used to quantize the policies deployed on our target ESP32-S3 MCU. Further, for the ONNX-exported policies, we compared our custom GRU formulation, referred to in the following as *Aug*, against the native (*Nat*) GRU implementation provided by ONNX to show that our formulation retains the same performance at inference but with a 20% reduction in model size due to complete quantization. Naturally, this only applies to the RA-based algorithms as pure TD3 and SAC do not include a GRU encoder.

[2]`https://github.com/real-world-drl/quant-drl`

B. Full-scale Computer

As a baseline evaluation, we used the same high-performance computer to run inference on the trained policies and their corresponding quantized variants. We evaluated the policies both in simulation and on the real robot. To connect to the robot hardware and receive the necessary state input information required for policy inference, proprioceptive observations were streamed over MQTT from the real robot, while positional data was streamed from a motion capture system. Similarly, in the simulation evaluations, two separate MQTT topics were used to stream proprioceptive and positional data. In both cases, the actions were inferred and then streamed back to the robot for actuation.

C. Resource Constrained Setup using Raspberry Pi 4

To assess the impact of quantization in a resource-constrained system, a Raspberry Pi 4 (4 GB RAM Model B) was employed. This compact computer provides a full 64-bit Linux environment, allowing the same setup to be compiled for the Cortex-A72 (Arm v8) processor. With its quad-core architecture and clock speed of 1.8 GHz, this processor is capable of storing and running inference on all of the tested policies and can clearly illustrate the effects of quantization on inference speed. The DRL agent was implemented identically to that on the full-scale computer.

D. MCU Deployment Setup

We used the ESP32-S3, with the dual-core Xtensa LX7 architecture, operating at a clock frequency of 240 MHz. Due to the constraints of the system, only the INT8 ESP-DL quantized models could be run on the MCU. Further, the quantized models were loaded and executed from high-speed octal SPI PSRAM, operating at 80 MHz, rather than from internal SRAM to satisfy memory constraints. Inference

was executed on core 1, while core 0 handled all other tasks, including communication, sensor readings, and actuation.

In the simulated experiments, observations were streamed from the simulation running on the full-scale computer via MQTT using the on-board Wi-Fi module. Actions inferred from the policy were also transmitted back to the simulation over MQTT. In contrast, the real-world experiments relied only on proprioceptive observations already available to the MCU, comprising IMU readings, current measurements, and servo positions. The inferred actions were rescaled and actuated on-board. To calculate the policy's locomotion performance, we used motion capture to measure the distance traveled and heading at each step.

VI. RESULTS

Our experimental evaluation aims to answer the following questions: (i) How does quantization impact the performance of DRL in terms of reward, inference time, and model size? (ii) What are the effects of reducing computational resources on inference performance? (iii) Does the custom GRU implementation incur any performance penalties compared to the native GRU implementation? and (iv) How does the quantized model perform when deployed on an embedded microcontroller for onboard inference in a low-cost quadrupedal walking task?

A. Effects of Quantization on Policy Performance

As shown in Fig. 4, the plain variants of SAC and TD3 failed to learn viable policies: in both cases the robot struggled at the start of the obstacle and could not progress. Consequently, these variants were excluded from the real-world experiments and subsequent analysis.

Fig. 6 presents the rewards of the RA-based algorithms achieved by the original trained model and the different quantization variants in simulation. Because the Raspberry Pi 4 runs the same models under the same runtime (albeit on a slower CPU with a different architecture), the modest increase in inference latency ($\approx 1\,\text{ms}$) remains well below the control step of $50\,\text{ms}$. The resulting rewards are consistent with those from the full-scale computer experiments; we therefore omit the results on the full-scale computer.

Excluding the ESP-DL quantized models, the results indicate that the rewards obtained by each algorithm exhibit minimal variance, suggesting that quantization of the DRL models does not significantly impact their performance in terms of reward. Furthermore, the identical outputs generated by *AugGRU* and *NatGRU* for a given input and hidden state result in similar rewards, as expected.

ESP-DL–quantized models show a reduction in distance traveled (16% for RA-SAC and 5% for RA-TD3). The RA-SAC shortfall is largely attributable to the robot reaching the obstacle and encountering difficulty on the initial ascent. Otherwise, the robot maintains forward motion at a reduced velocity throughout the episode, indicating that the deficit does not accumulate over time.

Quantization significantly reduces the model size, as illustrated in Table I. The different quantization methods yield comparable reductions in model size. For the unmodified TD3/SAC algorithms, quantization results in models that are approximately three to four times smaller than their full-precision counterparts. Notably, the custom GRU implementation (*Aug*) yields a quantized model that is about 20% smaller than the quantized native GRU implementation (*Nat*), due to the native ONNX quantization not quantizing the GRU weights. Furthermore, the RA-based ESP-DL quantized models exhibit a more substantial size reduction, being approximately 40% smaller than the ONNX quantized models.

TABLE I: Model sizes (kB) for each policy.

	FP32	FP32 ONNX		INT8 ONNX		INT8
	Torch	Nat	Aug	Nat	Aug	ESP-DL
RA-SAC	371	750	1,153	477	382	225
RA-TD3	364	732	1,136	465	377	221
SAC	300	322		103		81
TD3	293	301		87		77

TABLE II: Policy inference times (μs) across systems.

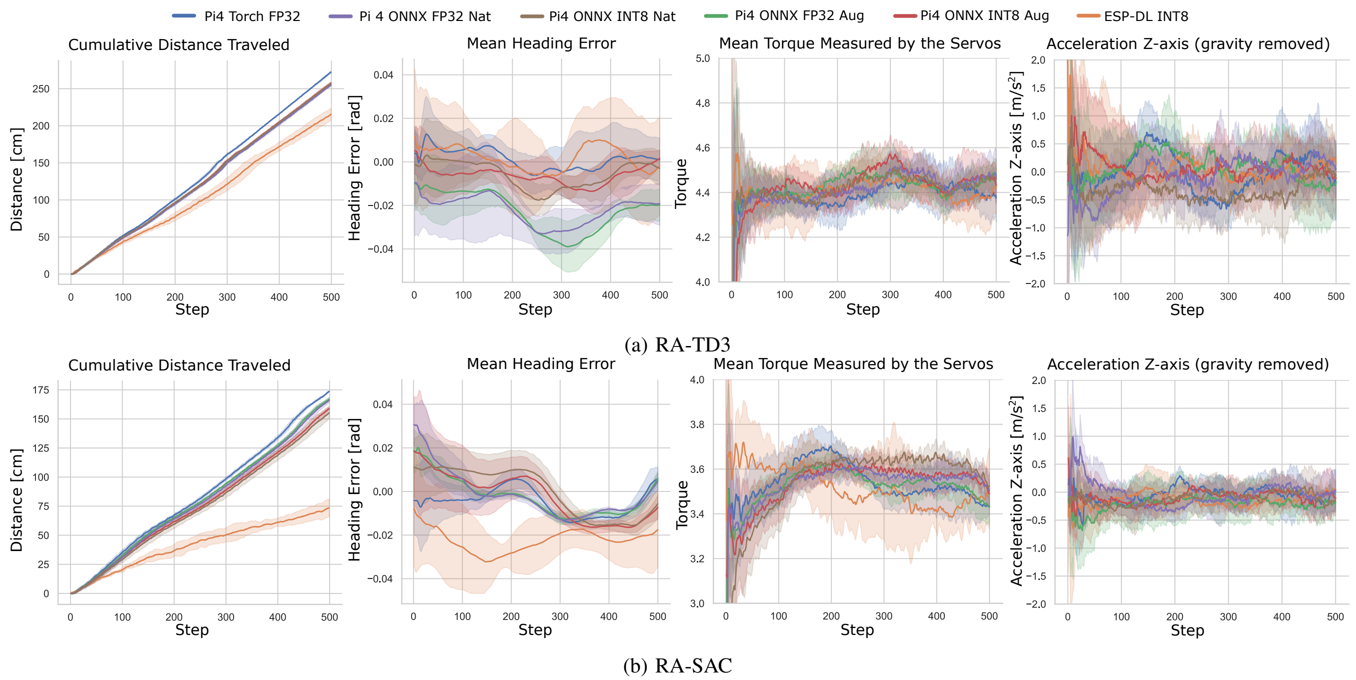
		FP32	FP32 ONNX		INT8 ONNX	
		Torch	Nat	Aug	Nat	Aug
Full	RA-SAC	868 ± 319	430 ± 99	509 ± 99	469 ± 79	419 ± 103
Full	RA-TD3	834 ± 136	395 ± 79	478 ± 87	395 ± 96	442 ± 93
Full	SAC	447 ± 108	217 ± 41		219 ± 52	
Full	TD3	393 ± 75	201 ± 39		198 ± 36	
Pi 4	RA-SAC	3658 ± 327	1177 ± 225	1192 ± 232	1114 ± 235	1113 ± 262
Pi 4	RA-TD3	3458 ± 332	1137 ± 253	1156 ± 240	1066 ± 221	1103 ± 435
Pi 4	SAC	1557 ± 144	467 ± 98		448 ± 95	
Pi 4	TD3	1318 ± 129	425 ± 79		404 ± 86	
		INT8 ESP-DL				
MCU	RA-SAC	5372 ± 133				
MCU	RA-TD3	5348 ± 69				
MCU	SAC	1330 ± 85				
MCU	TD3	1080 ± 24				

B. Effects of Reducing Computational Resources

As illustrated in Table II, utilizing the ONNX runtime results in a significant reduction in inference time, exceeding 50%, across all algorithms. The native GRU implementation yields marginally faster inference times, likely due to optimizations in the native implementation. However, given that these differences are on the order of 30-50μs, they can be considered negligible. In contrast, quantization provides only marginal improvements, on the order of 70μs.

On the ESP32-S3, the inference times for the RA-based models are less than 6ms, while those for the plain SAC/TD3 models are less than 1.5ms.

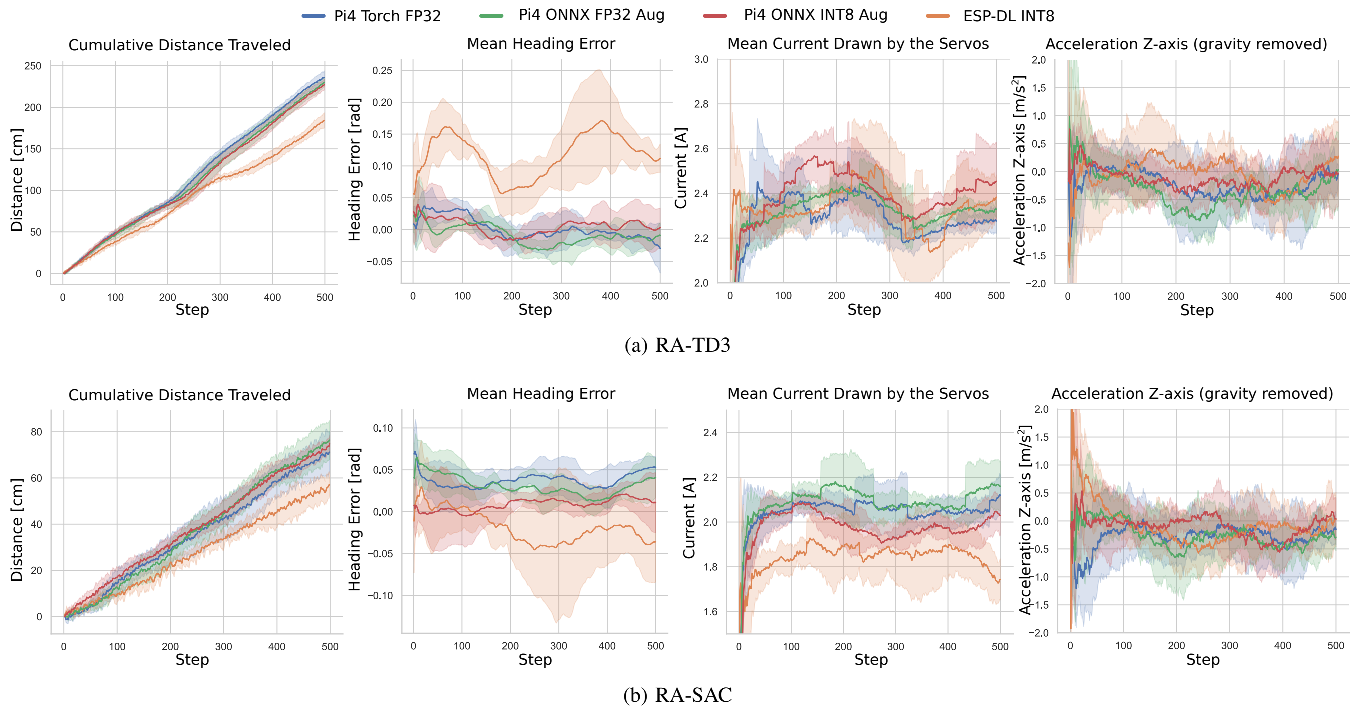
When considering resource utilization, in contrast to previous work [3], our experiments did not reveal a significant impact of quantization on memory usage. Regardless of the algorithm or quantization method employed, the total memory usage remained relatively constant at approximately 145MB. The consistent and low memory usage observed in our experiments can be partly attributed to the use of the C++ ONNX Runtime and LibTorch rather than Python-based inference. With regard to CPU utilization, the consumption remained stable, averaging around 75% of a single core's capacity.



(a) RA-TD3

(b) RA-SAC

Fig. 6: Comparison of selected reward components measured in the simulation, with mean heading error and mean motor torque computed over a 100-step sliding window. Results are averaged over five runs.



(a) RA-TD3

(b) RA-SAC

Fig. 7: Comparison of selected reward components measured on the physical robot, with mean heading error and mean motor current computed over a 100-step sliding window. Results are averaged over five runs.

C. Custom GRU Implementation Performance

The *AugGRU* implementation is designed to produce outputs identical to the native PyTorch GRU module, and as expected, achieves comparable results in both FP32 and INT8 quantized forms. The inference time comparisons further demonstrate that the optimizations in the native ONNX GRU result in differences on the order of tens of microseconds. Moreover, the inference times of less than 5ms achieved on the ESP32S3 suggest potential applications beyond DRL tasks, including modeling sequential data such as language, speech, and time-series signals.

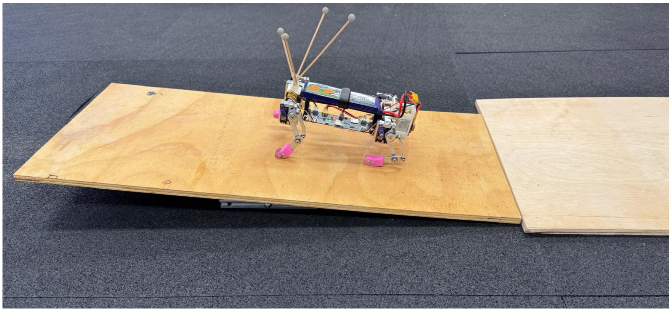


Fig. 8: Real-world evaluation environment: The robot traverses a compound obstacle consisting of a smooth wooden platform, a pivoting ramp transitioning from incline to decline, and a compliant gym mat.

D. Real-world Robot Deployment

Given the differing performance profiles of RA-SAC and RA-TD3, we ran two experimental configurations. RA-SAC was evaluated only on a flat gym-mat surface, whereas RA-TD3 was tested on the pivoting ramp shown in Fig. 8, involving surface transitions and both uphill and downhill motion.

Results from the real-world robot deployment are shown in Fig. 7. Consistent with the simulation results, all models executed on the Raspberry Pi 4 maintain comparable performance. ESP-DL quantized models running on-board exhibit a modest degradation, smaller than in simulation and primarily reflected in the mean heading error. Although this quantization introduces a measurable performance penalty, ESP-DL currently provides the only practical pathway for executing these models on ESP32-class hardware. Taken together, these results demonstrate the feasibility of our approach for quantizing and deploying trained policies on microcontrollers for real-world operation.

VII. Conclusion

This paper examined the impact of quantization on DRL policies for quadrupedal locomotion and introduced a quantization-friendly SAC architecture and GRU implementation. By integrating a GRU-based encoder into the actor and providing a custom multi-layer GRU implementation, we obtained a streamlined network that supports efficient inference and quantization.

A comprehensive evaluation on a desktop machine, a Raspberry Pi 4, and a low-power microcontroller showed that the performance degradation due to quantization is small, varies across toolchains and platforms, and crucially does not accumulate over time. Finally, we deployed quantized policies on a low-cost quadrupedal robot using an ESP32-S3 microcontroller, achieving fully autonomous, real-time control using only proprioceptive sensing and on-board inference.

Executing the entire neural-network inference on a low-cost microcontroller (less than $5 class) is both practically important, enabling fully embedded and affordable autonomy, and technically challenging due to severe compute, memory, and power constraints, as well as limited toolchain support.

References

[1] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[2] A. Goel, C. Tung, Y.-H. Lu, and G. K. Thiruvathukal, "A survey of methods for low-power deep learning and computer vision," in *2020 IEEE 6th World Forum on Internet of Things (WF-IoT)*. IEEE, 2020, pp. 1–6.

[3] S. Krishnan, M. Lam, S. Chitlangia, Z. Wan, G. Barth-Maron, A. Faust, and V. J. Reddi, "Quarl: Quantization for fast and environmentally sustainable reinforcement learning," *arXiv preprint arXiv:1910.01055*, 2019.

[4] F. Svoboda, D. Nunes, M. Alizadeh, R. Daries, R. Luo, A. Mathur, S. Bhattacharya, J. S. Silva, and N. D. Lane, "Resource efficient deep reinforcement learning for acutely constrained tinyml devices," in *Research Symposium on Tiny Machine Learning*, 2020.

[5] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," in *International conference on machine learning*. Pmlr, 2018, pp. 1861–1870.

[6] O. for double-blind review, "Omitted for double-blind review," in *2023 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2023, pp. 7126–7132.

[7] S. Fujimoto, W.-D. Chang, E. Smith, S. S. Gu, D. Precup, and D. Meger, "For sale: State-action representation learning for deep reinforcement learning," *Advances in neural information processing systems*, vol. 36, pp. 61 573–61 624, 2023.

[8] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, "A survey of quantization methods for efficient neural network inference," *arXiv preprint arXiv:2103.13630*, 2021.

[9] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: An evaluation platform for general agents," *Journal of Artificial Intelligence Research*, vol. 47, pp. 253–279, 2013.

[10] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "OpenAI gym," *arXiv preprint arXiv:1606.01540*, 2016.

[11] M. Z. Alom, A. T. Moody, N. Maruyama, B. C. Van Essen, and T. M. Taha, "Effective quantization approaches for recurrent neural networks," in *2018 international joint conference on neural networks (IJCNN)*. IEEE, 2018, pp. 1–8.

[12] A. Foucault, F. Mamalet, and F. Malgouyres, "Quantized approximately orthogonal recurrent neural networks," *arXiv preprint arXiv:2402.04012*, 2024.

[13] A. Kusupati, M. Singh, K. Bhatia, A. Kumar, P. Jain, and M. Varma, "Fastgrnn: A fast, accurate, stable and tiny kilobyte sized gated recurrent neural network," *Advances in neural information processing systems*, vol. 31, 2018.

[14] H. Wu, P. Judd, X. Zhang, M. Isaev, and P. Micikevicius, "Integer quantization for deep learning inference: Principles and empirical evaluation," *arXiv preprint arXiv:2004.09602*, 2020.

[15] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 2704–2713.

[16] E. Park, S. Yoo, and P. Vajda, "Value-aware quantization for training and inference of neural networks," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 580–595.

[17] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, "Haq: Hardware-aware automated quantization with mixed precision," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 8612–8620.

[18] D. Zhang, J. Yang, D. Ye, and G. Hua, "Lq-nets: Learned quantization for highly accurate and compact deep neural networks," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 365–382.

[19] Y. Kim, H. Oh, J. Lee, J. Choi, G. Ji, M. Jung, D. Youm, and J. Hwangbo, "Not only rewards but also constraints: Applications on legged robot locomotion," *IEEE Transactions on Robotics*, vol. 40, pp. 2984–3003, 2024.