Energy-efficient Dynamic Partitioning and Tensors Compression of AI Applications in Smart Eyewears

Abednego Wamuhindo Kambale*, Samin Shokrivahed*, Giacomo Verticale, Francesca Palermo, Diana Trojaniello, Danilo Ardagna

March 2, 2026

Abstract

Resource-constrained smart eyewear (SEW) devices face significant challenges when deploying deep neural networks due to limited computational capacity and battery life. Computational offloading to companion devices like smartphones and cloud servers addresses processing limitations, but data transmission becomes a critical bottleneck, consuming over 50% of total energy in some scenarios. Although lossless compression methods provide limited data reduction for intermediate tensors, lossy techniques such as Vector Quantization (VQ) offer higher compression ratios (requiring only 3.3 bits per float) at the expense of inference accuracy degradation. This paper presents an adaptive multi-stage compression framework that dynamically balances these trade-offs across the SEW-phone-cloud continuum. We employ VQ at the SEW-phone interface where aggressive compression is essential (achieving 89.6% tensor size reduction with 90% retained accuracy), followed by adaptive selection between quantization and run-length encoding for phone-to-cloud transmission based on network conditions. A Deep Q-Network (DQN) agent jointly optimizes network partitioning points and compression strategies to minimize energy consumption while preserving accuracy and meeting latency constraints. A large simulation campaign considering object detection and human pose estimation tasks demonstrate that our method achieves 55–70% energy savings and 86–91% violation reduction compared to Neurosurgeon (a dynamic partitioning baseline without compression), 45.8% energy savings versus local execution, and 61.1% savings over uncompressed offloading, with latency violation rates below 9% and acceptable accuracy loss (8.0–8.1%). These results enable practical deployment of AI applications on battery-limited SEW devices.

Deep Reinforcement Learning, Offloading, Tensor Compression, Cloud computing.

1 Introduction

Smart eyewear (SEW) has evolved beyond its traditional medical applications to enhance various aspects of daily life. The integration of sensors and advanced functionalities has

*These authors contributed equally to this work.

enabled a wide range of innovative applications [1]. Artificial intelligence (AI) plays a crucial role in augmenting these capabilities, allowing SEW to provide real-time insights and assistive features [2], [3]. However, many state-of-the-art AI models, particularly neural networks (NN), demand substantial memory and computational resources to deliver high performance. This poses a significant challenge for SEW, as these devices are inherently constrained in terms of memory, processing power, and battery life, making the deployment of AI models on such platforms particularly difficult [4]. This introduces a trade-off: on one hand, there is a demand for sophisticated AI-driven functionalities with good accuracy, while on the other, resource efficiency must be prioritized.

To address this challenge, several NN optimization techniques have been explored in the literature. Model compression [5] and model quantization [6] are widely used methods for reducing the size of deep neural networks (DNNs), although at the cost of potential accuracy loss. Another approach, task offloading [7], leverages network connectivity to distribute computational tasks between the SEW device, edge nodes, and cloud servers. This involves partitioning a DNN into multiple segments and executing only a portion locally while offloading the remaining computation to more powerful external devices.

Several studies [7], [8], [9], [10] have investigated task offloading strategies. Offloading requires SEW to communicate with edge and/or cloud devices via networks such as WiFi and 5G. However, these networks are susceptible to variability, potentially impacting user experience [11]. Moreover, studies have shown that data transfer accounts for over 50% of total energy consumption [12]. Reducing data transfer volume could help mitigate energy inefficiency.

This paper introduces tensor compression as a strategy to minimize data transfer size, thereby reducing both energy consumption and execution time violations. However, compression inevitably trades off accuracy as reducing data size and changing information representation lead to precision loss. To address this challenge, we employ a Deep Q-Network (DQN)-based agent that learns the system dynamics, which are influenced by variations in WiFi and 5G throughput, as well as cloud performance fluctuations. The Reinforcement Learning (RL) agent is trained to optimize energy efficiency and minimize the cost of 5G communication while mitigating compression-related accuracy loss.

Our main contributions are:

1. We propose a novel two-stage adaptive compression framework that employs Vector Quantization (VQ) at the SEW-phone interface to achieve aggressive size reduction (3.3 bits per float with approximately 90% accuracy). Subsequently, dynamic selection between quantization and Run-Length Encoding (RLE) is performed at the phone-cloud interface based on network conditions. To the best of our knowledge, this is the first application of VQ to intermediate tensor compression in the SEW computing continuum.
2. Unlike prior work that optimizes partitioning and compression independently [13], [14], or focuses solely on partitioning strategies [11], [15], we jointly optimize DNN partitioning configuration and compression scheme selection across two split points simultaneously. This holistic approach captures the interdependencies between partition boundaries and optimal compression strategies.
3. We validate our approach through simulation considering realistic network traces (WiFi

and SEW system profiling and real-world 5G datasets [16]) across two AI applications (object detection and pose estimation). Results demonstrate 55–70% energy savings and 86–91% violation reduction versus Neurosurgeon [7] (demonstrating the necessity of adaptive compression), 45.8% energy savings compared to local execution and 61.1% savings over uncompressed offloading, with violation rates below 9% and acceptable accuracy loss (8.0–8.1%).

The rest of this paper is structured as follows: Section 2 reviews related work. Section 3 introduces the reference system under consideration and the problem faced, while Section 4 presents our proposed approach for its solution. Section 5 reports the experimental results, and Section 6 concludes the paper, highlighting future research perspectives.

2 Related work

This work combines two keys concepts that have been explored independently in the literature. Section 2.1 reviews the literature related to DNN partitioning and RL-based task offloading in edge computing continuum, while Section 2.2 presents the work related to data compression.

2.1 Reinforcement Learning-based task offloading

Several works have addressed the challenge of DNN partitioning and task offloading across distributed computing infrastructures. In [17], authors developed an algorithm to identify optimal partitioning points in multi-device and multi-edge server configurations. They implemented a partitioning point retention scheme that mitigates search space complexity and generated a cost matrix processed using the Hungarian algorithm to ascertain device-edge server pairings that minimize the overall system cost. Tian et al. [9] introduced EEDPO, a DNN partitioning and offloading approach for multi-user environments that uses min-cut/max-flow theory and dynamic programming. Their formulation addresses task completion maximization while minimizing energy consumption through joint exploration of DNN model partitioning.

RL approaches have gained attention for their ability to handle dynamic offloading decisions. RL offers key advantages over dynamic programming and traditional solvers as it manages parameter uncertainties without explicit system models, enables online adaptation to system dynamics, and provides fast inference for real-time decision-making [18]. The work in [19] introduced a DRL-based algorithm that considers logical and data relationships among subtasks modeled as Directed Acyclic Graphs (DAGs). They introduced an embedding method to encode DAG vertices into sequences that encapsulate task profiles and dependency information. They used an off-policy Deep RL algorithm with clipped surrogate objectives, though the binary offloading mechanism limits tasks to either local or remote execution exclusively. Kambale et al. [11] presented a three-layer computing continuum architecture for SEW AI applications, formulating the problem as an MDP for image-processing inference tasks. Their RL-based approach dynamically selects configurations to minimize energy consumption and 5G costs while meeting latency requirements. Evaluating multiple tabular RL methods, they identified Q-learning as most effective, outperforming baselines

and achieving significant energy savings. However, like all tabular methods, their approach faces scalability and generalization limitations due to state discretization, restricting it to low-dimensional state-action spaces.

Sohn et al. [10] presented a multi-objective optimization framework combining RL with Lyapunov optimization to balance inference accuracy and computational efficiency, optimizing both frame dropping and offloading decisions for DNN-based object detection in resource-constrained Multi-access Edge Computing environments. They employed a layering approach with the lower layer managing joint flow control and DNN partitioning while the upper application layer uses a Lyapunov-guided RL algorithm to make frame-dropping decisions. More recently, Sedghani et al. [15] proposed a Federated RL framework for runtime DNN offloading in SEW that employs layer-level partitioning and adapts to dynamic network conditions. Leveraging multiple agent for training a shared policy, their approach produced a more stable policy with lower variability than single-agent training, particularly as agent count increases. Moreover, their framework maintained effectiveness under asynchronous learning. However, their approach does not consider tensor compression, leaving potential energy savings from reduced data transmission unexplored.

2.2 Compression

DNN partitioning strategies distribute computation across devices but do not address the communication overhead of intermediate tensor transfer, which can account for over 50% of energy consumption in offloading scenarios [12], making compression techniques essential for practical deployment.

Similarly, the same problem exists in distributed training, where workers must exchange high-dimensional gradient tensors, this communication quickly becomes a major bottleneck under limited bandwidth. Works in [20] and [21] address this problem. Abdelmoniem and Canini [20] propose a delay-aware, training-time gradient compression method that reduces communication overhead in distributed training, accelerating training while incurring a loss in model accuracy due to lossy compression. Xu et al. [21] perform a comparative analysis of various training-phase communication compression methods, including quantization, sparsification (selective transmission of major gradient components), and low-rank approximations, within their GRACE framework for distributed machine learning. Another line of work modifies the DNN architecture to inherently support compression. In [22], the authors insert a bottleneck embedding at selected points in the network and partition the model at these locations, allowing the device to send compact latent representations and thereby reduce communication overhead while preserving accuracy. However, this design requires partial retraining, introducing recurring overhead whenever the model is updated.

Computational, energy, and network bandwidth constraints likewise motivate partitioning and compression strategies in Mobile-Cloud inference settings. Choi et al. [23] propose a dynamic compression method for split computing between mobile devices and the cloud. Their approach compresses intermediate feature maps at the partition point using channel-wise slicing combined with Top-k pruning to reduce transmission latency. To handle varying network conditions, they perform compression-aware training with a SuperNetwork that supports multiple compression ratios via joint training. At runtime, the compression level is selected based on available bandwidth, enabling adaptive latency reduction with limited

accuracy degradation.

Alternatively, Carra and Neglia [14] examine the direct effect of quantizing only the coefficients of pre-trained models on tasks like image classification and object detection. They find that using just 4 bits per coefficient has a minimal impact on performance. Their work also explores lightweight, lossless encoding techniques such as RLE.

Conversely, numerous approaches consider lossy quantization methods during inference time. Among these, Vector Quantization (VQ) stands out as one of the most promising. For example, Liu et al. [24] introduce a method called Aligned Vector Quantization for running large vision language models across an edge device and the cloud. The key idea is to place a small quantization module at the point where the model is split so that the edge device sends a highly compressed version of its intermediate features instead of large raw tensors or images. Aiming for the same high compression and low accuracy loss approach we look for less complex VQ approach which does not require changing the model architecture. In [25], the authors incorporate vector quantization into a variational autoencoder framework and demonstrate VQ-VAE on images, audio, and video data. VQ-VAE employs a lightweight vector quantization module, using simple nearest-neighbor assignments in a small codebook, which makes the quantization step computationally inexpensive. Inspired by this work we have experimented the light weight VQ in inference time in SEW.

3 System Description and Problem Statement

Section 3.1 provides an overview of the reference system, while Section 3.2 outlines the formulation of the optimization problem that we investigate in this paper.

3.1 System description

Figure 1 shows a typical scenario in which a SEW user executes image-based AI applications, such as object detection. These tasks have strict execution time constraints that directly impact user experience, e.g. slow object tracking typically requires 1–5 fps, i.e., 200 ms–1 s, per frame [26]. The goal is to optimize execution while minimizing data transfer and extending SEW battery life. Given SEW resource limitations, a mobile phone and cloud infrastructure assist in computation offloading. The SEW maintains a WiFi connection to a mobile phone, which offers higher computational power and communicates with the cloud via 5G. The AI application consists of a DNN that can be partitioned at specific points, allowing different partitions to execute across multiple devices, via candidate DNN configurations [8], [11]. An RL agent dynamically selects the optimal DNN execution strategy. Three execution scenarios are possible: 1) SEW-Only mode, the entire DNN runs locally on the SEW, eliminating data transfer, which is preferred when battery capacity is sufficient or network conditions are poor; 2) SEW-Phone mode, if local execution is infeasible or inefficient, the SEW offloads an intermediate tensor to the mobile phone, reducing its computational load; 3) SEW-Phone-Cloud mode, for high-computation tasks, the phone further offloads tensors to the cloud, which processes the data and returns results via the phone to the SEW. To optimize execution, intermediate tensors undergo compression (Figure 1) before offloading, reducing transfer size while balancing execution time, energy efficiency and accuracy loss.

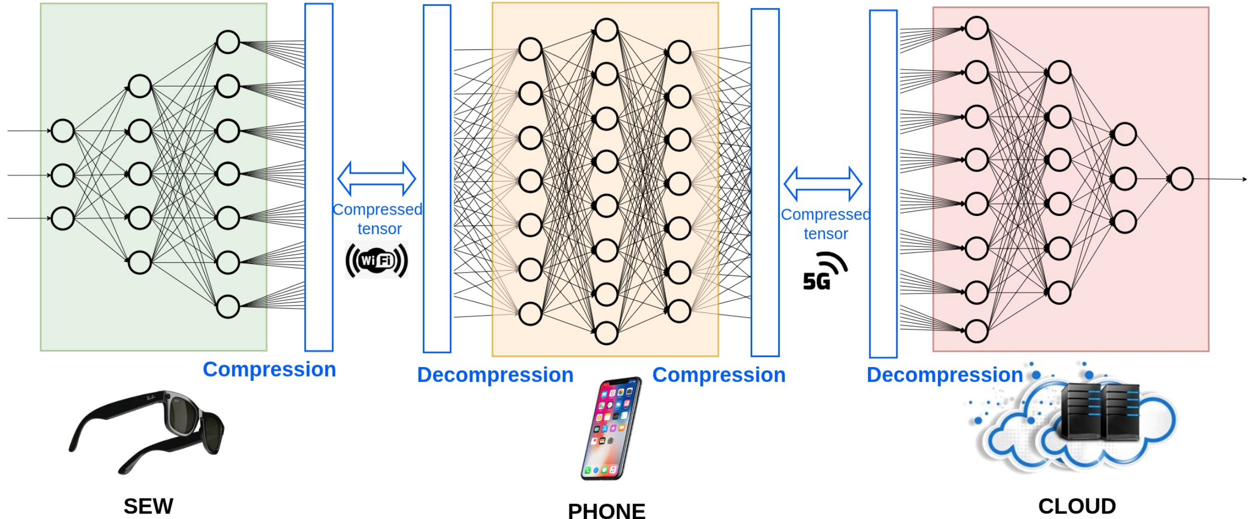


Figure 1: Reference scenario for performing AI tasks with tensor compression on SEW, mobile phone and cloud.

The compression pipeline supports multiple techniques with varying compression-accuracy trade-offs: quantization reduces bit precision of tensor elements, RLE exploits value repetitions for lossless compression, and VQ applies codebook-based lossy compression for more aggressive size reduction. These techniques are described in detail in Section 4.1. The RL agent dynamically selects the partition configuration and compression levels to mitigate performance degradation while balancing data size reduction and model accuracy loss depending on system current conditions.

The agent also accounts for network and cloud variability. WiFi performance fluctuates in congested environments, affecting application performance [11], [27]. Similarly, 5G mmWave, while offering high throughput, experiences significant variability [16]. Cloud workloads introduce additional queuing delays, impacting latency [28]. By adapting to these factors, the proposed process ensures efficient AI execution across the SEW-phone-cloud system.

3.2 Problem Statement

We consider image-based AI applications that process requests supplied in the form of image frames with a frequency Λ (expressed in frames per second). The system control period has a length τ. Therefore, all the defined parameters are measured in each time window τ.

Let $\mathcal{D} = \{1, 2, 3\}$ denote the set of indices for computing components: the SEW ($d = 1$), the mobile phone ($d = 2$), and the cloud server ($d = 3$). Following the approach in [11], let $\mathcal{K} = \{\kappa^1, \kappa^2, \ldots, \kappa^I\}$ represent the set of all feasible partition configurations, where $|\mathcal{K}|$ is the total number of configurations. Each configuration $\kappa^i \in \mathcal{K}$ specifies a fixed partitioning of the DNN across the three components, expressed as an ordered tuple $\kappa^i = (p^i_1, p^i_2, p^i_3)$, where p^i_d denotes the partition (i.e., subset of DNN layers) assigned to device d under configuration κ^i. The partitions are executed sequentially which means p^i_1 executes on the SEW, p^i_2 runs on the phone and p^i_3 on the cloud. To maintain a consistent one-to-one mapping between devices and partitions across all configurations, we introduce a placeholder partition $p_\emptyset$ that represents no computation. When a device d executes no layers in configuration κ^i, we set $p^i_d = p_\emptyset$. For example, if only the SEW and phone process layers, then $p^i_3 = p_\emptyset$.

Let $\mathcal{Q} = \{q^1, q^2, \ldots, q^j\}$ denote the set of data reduction schemes, where each scheme $q^j \in \mathcal{Q}$ represents a compression technique which may be quantization only, quantization and run-length encoding (RLE) or vector quantization (VQ). Throughout this formulation, we use the terms "quantization", "RLE", and "VQ" if we want to explicitly differentiate them, and "data reduction scheme" or "compression scheme" for all as they all describe methods for data reduction applied to intermediate tensors. The available schemes, as in other literature approaches [14], [25], include quantization alone (utilizing 6, 7, or 8 bits), quantization combined with RLE applied at 6, 7, or 8-bit levels and VQ (with 2 or 3 bits per element). More details are provided in Section 4.1.

Unlike the partitioning strategy, the data reduction schemes can be independently selected for each communication link. Specifically, split point 1 refers to the SEW-to-phone communication link, and split point 2 refers to the phone-to-cloud communication link. Both data reduction schemes are selected from the same set $\mathcal{Q}$. This formulation allows the same data reduction scheme to potentially be applied at both splitting points, or different schemes to be selected independently for each communication link.

3.2.1 Decision Variables

We introduce one master binary decision variable and three auxiliary binary decision variables:

- u^{ijk} which is 1 if configuration κ^i is chosen and compression schemes q^j and q^k are selected for split point 1 and 2, respectively, and 0 otherwise. This is the master decision variable.
- x^i which is 1 if configuration κ^i is chosen and 0 otherwise. This auxiliary variable extracts the partition selection from u^{ijk}.
- y^{ij} which is 1 if configuration κ^i is selected and compression scheme q^j is chosen for split point 1 and 0 otherwise. This auxiliary variable is used with parameters that depend on partition and first compression only.
- z^{ik} which equals 1 if configuration κ^i is selected and compression scheme q^k is chosen for split point 2 and 0 otherwise. This auxiliary variable is used with parameters that depend on partition and second compression only.

The cardinalities $|\mathcal{K}|$ and $|\mathcal{Q}|$ denote the total number of partition configurations and data reduction schemes, respectively. Moreover both $q^j, q^k \in \mathcal{Q}$.

3.2.2 Constraints

At each time slot τ, exactly one combination of partition configuration and data reduction schemes must be selected:

$$\sum_{i=1}^{|\mathcal{K}|} \sum_{j=1}^{|\mathcal{Q}|} \sum_{k=1}^{|\mathcal{Q}|} u^{ijk} = 1 \tag{1}$$

The auxiliary variables are defined through linking constraints that extract specific aspects of the master decision:

$$x^i = \sum_{j=1}^{|\mathscr{Q}|} \sum_{k=1}^{|\mathscr{Q}|} u^{ijk} \quad \forall i \in \mathcal{K} \tag{2}$$

$$y^{ij} = \sum_{k=1}^{|\mathscr{Q}|} u^{ijk} \quad \forall i \in \mathcal{K}, \forall j \in \mathscr{Q} \tag{3}$$

$$z^{ik} = \sum_{j=1}^{|\mathscr{Q}|} u^{ijk} \quad \forall i \in \mathcal{K}, \forall k \in \mathscr{Q} \tag{4}$$

All decision variables are binary:

$$x^i, y^{ij}, z^{ik}, u^{ijk} \in \{0, 1\} \tag{5}$$

3.2.3 Parameters

For any given combination of partition configuration κ^i and data reduction schemes q^j (SEW→phone) and q^k (phone→cloud), we define the following parameters:

- μ^{ijk}: The mean Average Precision (mAP) obtained by running configuration κ^i with compression scheme q^j for SEW-to-phone transfer and scheme q^k for phone-to-cloud transfer. This value is obtained by comparing the output of the model after applying q^j and q^k with the ground truth (output) from the non-partitioned model using a large validation dataset.
- t_d^i: the local execution time of the partition p_d^i of configuration κ^i on device $d \in \mathcal{D}$. This time depends only on the partition configuration, not on the compression scheme.
- $l_{q,SEW}^{ij}$ and $l_{q,phone}^{ik}$: the compression overhead on the SEW and the phone respectively, when configuration κ^i is selected and compression schemes q^j and q^k are applied at split point 1 and 2, respectively. These latencies depend on i because the partition configuration determines the size of the intermediate tensor, which affects the compression overhead.
- $l_{-q,phone}^{ij}$ and $l_{-q,cloud}^{ik}$: the decompression overhead on the phone and the cloud, with configuration κ^i and compression schemes q^j and q^k at split point 1 and 2, respectively.
- e_1^{ij}: per-request energy consumption (in joules) for local computation on the SEW. This energy comprises the DNN partition execution and the compression energy consumption and it is determined through profiling on the SEW.
- e_2^{ijk}: per-request energy consumption (in joules) for local computation on the phone. It includes the DNN partition execution, decompression, and compression energy consumption and it is determined through profiling on the phone.
- δ_{12}^{ij}: The size of intermediate tensor data transferred from the SEW to the phone when using configuration κ^i and applying data reduction scheme q^j to the intermediate tensors.
- δ_{23}^{ik}: The size of intermediate tensor data transferred from the phone to the cloud when using configuration κ^i and applying data reduction scheme q^k to the intermediate tensors.

Additionally, we define the following system parameters that are independent of the configuration and compression scheme selection:

- r_{WiFi}: The throughput of the WiFi connection between SEW and phone (in bytes per second).
- r_{5G}: The throughput of the 5G connection between phone and cloud (in bytes per second).
- g: The unit cost per byte transferred over 5G (in currency units per byte).
- θ_{SEW}: The power consumption of the SEW wireless interface during data transmission (in watts).
- θ_{phone}: The power consumption of the phone 5G interface during data transmission (in watts).
- α: Energy cost coefficient (in $/joule) for converting energy consumption to monetary cost in the objective function.
- L_{max}: Maximum allowable end-to-end latency for satisfactory user experience (in seconds).

3.2.4 Data Transmission Latency and Cost

The transmission latencies between the SEW and phone (l_{sp}), and between the phone and cloud (l_{pc}), are expressed as:

$$l_{sp} = \frac{\sum_{i=1}^{|\mathcal{K}|} \sum_{j=1}^{|\mathcal{Q}|} \delta_{12}^{ij} \cdot y^{ij}}{r_{WiFi}} \tag{6}$$

$$l_{pc} = \frac{\sum_{i=1}^{|\mathcal{K}|} \sum_{k=1}^{|\mathcal{Q}|} \delta_{23}^{ik} \cdot z^{ik}}{r_{5G}} \tag{7}$$

The 5G usage cost is defined as:

$$c_{5G} = \sum_{i=1}^{|\mathcal{K}|} \sum_{k=1}^{|\mathcal{Q}|} \Lambda \cdot g \cdot \delta_{23}^{ik} \cdot z^{ik} \tag{8}$$

3.2.5 Application Latency and Compression Overhead

The execution time on the SEW comprises the DNN partition execution time and the compression overhead and is expressed as:

$$l_{SEW} = \sum_{i=1}^{|\mathcal{K}|} t_1^i \cdot x^i + \sum_{i=1}^{|\mathcal{K}|} \sum_{j=1}^{|\mathcal{Q}|} l_{q,SEW}^{ij} \cdot y^{ij} \tag{9}$$

In the same way, the execution time on the phone comprises the DNN partition execution time, the decompression overhead for data coming from SEW and the compression overhead for data to send to the cloud:

$$l_{phone} = \sum_{i=1}^{|\mathcal{K}|} t_2^i \cdot x^i + \sum_{i=1}^{|\mathcal{K}|} \sum_{j=1}^{|\mathcal{Q}|} l_{-q,phone}^{ij} \cdot y^{ij} + \\ + \sum_{i=1}^{|\mathcal{K}|} \sum_{k=1}^{|\mathcal{Q}|} l_{q,phone}^{ik} \cdot z^{ik} \tag{10}$$

We model the cloud server as an M/M/1 queueing system, where task arrivals from all connected users follow a Poisson distribution as commonly assumed in cloud and edge computing literature [11], [28][1]. Under this assumption, the cloud processing latency follows an exponential distribution.

The total end-to-end latency is:

$$l_{total} = l_{SEW} + l_{phone} + l_{sp} + l_{pc} + l_{cloud} \tag{11}$$

3.2.6 Energy Consumption

The power consumption for local execution on the SEW and phone is:

$$P_{exeSEW} = \sum_{i=1}^{|\mathcal{K}|}\sum_{j=1}^{|\mathcal{Q}|} \Lambda \cdot e_1^{ij} \cdot y^{ij} \tag{12}$$

$$P_{exePhone} = \sum_{i=1}^{|\mathcal{K}|}\sum_{j=1}^{|\mathcal{Q}|}\sum_{k=1}^{|\mathcal{Q}|} \Lambda \cdot e_2^{ijk} \cdot u^{ijk} \tag{13}$$

Recall that energy consumption e_1^{ij} and e_2^{ijk} are determined through energy profiling on the SEW and the phone respectively. P_{exeSEW} comprises the power consumed by DNN partition execution and by the tensor compression before transmission to the phone, while $P_{exePhone}$ encapsulates the power usage for tensor decompression, DNN partition execution and tensor compression before transmission to the cloud.

The data transfer power consumption is:

$$P_{wu} = \frac{\sum_{i=1}^{|\mathcal{K}|}\sum_{j=1}^{|\mathcal{Q}|} \Lambda \cdot \theta_{SEW} \cdot \delta_{12}^{ij} \cdot y^{ij}}{r_{WiFi}}, \tag{14}$$

$$P_{5gu} = \frac{\sum_{i=1}^{|\mathcal{K}|}\sum_{k=1}^{|\mathcal{Q}|} \Lambda \cdot \theta_{phone} \cdot \delta_{23}^{ik} \cdot z^{ik}}{r_{5G}}, \tag{15}$$

where θ_{SEW} and θ_{phone} denote the power consumption of the SEW wireless interface and phone 5G interface during data transmission, respectively.

The total energy consumption is:

$$E_{SEW}^{\tau} = \tau(P_{exeSEW} + P_{wu}) \tag{16}$$

$$E_{phone}^{\tau} = \tau(P_{exePhone} + P_{5gu}) \tag{17}$$

3.2.7 Optimization Problem

The agent objective is to optimize the execution of DNN layers across the SEW, mobile phone, and cloud, while considering the available computational resources and the remaining battery capacity on the SEW. The decision-making process can be formulated as an

[1] Since AI systems show more deterministic response times, modeling them with exponential response times (like in M/M/1) overestimates variability and delays. In reality, AI response times are less variable and more predictable—closer to deterministic processing (M/D/1). Thus, assuming exponential response times is a worst-case scenario for our RL agent, as actual AI performance benefits from lower variability and more concentrated response time distributions.

optimization problem aimed at minimizing a cost function that balances energy consumption, communication costs, and inference accuracy:

$$\min_{x,y,z,u} \left(\frac{\alpha(E_{SEW} + E_{phone}) + c_{5G}}{mAP} \right) \tau \tag{P1a}$$

subject to constraints (1)-(5), and:

$$l_{total} < L_{max} \tag{P1b}$$

where:

$$mAP = \sum_{i=1}^{|\mathcal{K}|} \sum_{j=1}^{|\mathcal{Q}|} \sum_{k=1}^{|\mathcal{Q}|} \mu^{ijk} \cdot u^{ijk} \tag{18}$$

The latency constraint (P1b) guarantees real-time performance, ensuring the total execution time remains below L_{max} to meet application-specific requirements for satisfactory user experience. The optimization problem (P1) can be solved optimally by exhaustive enumeration , with polynomial complexity of $O(|\mathcal{K}| \times |\mathcal{Q}|^2)$. However, solving it optimally in real-time is challenging due to: (1) large action spaces ($|\mathcal{K}| \times |\mathcal{Q}|^2$ combinations), (2) unknown system dynamics (cloud workload and network conditions which vary unpredictably), and (3) strict real time constraints requiring decisions within milliseconds. These characteristics motivate an RL approach, which learns optimal policies through environment interaction without explicit models. Section 4.2 details our Deep Q-Network (DQN) solution that adaptively balances energy, latency, and accuracy under dynamic conditions.

4 Methodology

In this section, we provide a concise overview of the proposed approach to tackle the problem outlined in the previous section. Section 4.1 describes the quantization strategy applied to DNN intermediate tensors to reduce the transmission energy while Section 4.2 details the RL approach.

4.1 Tensor compression

In our work, we aim to apply a lightweight post-training compression method during inference. This approach is designed to be compatible with state-of-the-art, off-the-shelf deep neural networks (DNNs) without requiring retraining or on-device optimization–both of which would demand additional development effort and could lead to accuracy degradation. The three compression techniques evaluated in this study are as follows:

4.1.1 Quantization

We employ linear quantization [14], which scales tensor elements (generally in floating-point representations) within min-max ranges to fixed-precision representations while preserving relative differences. The number of quantization bits determines the size-accuracy trade-off at each split point.

4.1.2 Quantization + Run-length Encoding

To improve the results that can be achieved by quantization, we also considered RLE, a lossless compression technique that exploits repetition patterns in data by storing consecutive occurrences as (count, value) pairs. In our implementation, we use (n-1)-bit quantization and allocate the most significant bit for RLE encoding, trading one bit of accuracy for substantial size reduction. RLE takes as input tensors quantized using (n-1) bits and further compresses them, with maximum repetition count $Max_{count} = 2^{n-1} - 1$. Below is an illustration of the RLE compression method: This sequence of data has to be transmitted after compression by RLE in 8 bits (using 7 bits and saving 1 bit for RLE flag):

`5,0,0,0,0,0,0,0,0,0,0,0,0,0,0`

RLE Compressed is as follows:

`5,142,0`

the binary representation is:

`00000101,10001110,00000000`

The most significant bit is used for RLE (shown by red in the representation of the bits), which indicates if an element is a value (0) or a repetition representation (1). The first value is sent as is because there is no repetition, the second value is repeated 14 times, which is why we send 14 plus the 1 in the most significant bit value (adding 2^7) resulting in value 142 as the element to be sent to show the number of repetitions which is followed by the corresponding value which is zero.

4.1.3 Vector Quantization

VQ [29] is a compression technique that maps high-dimensional vectors onto a finite codebook of representative entries. Each input vector is approximated by its nearest codebook entry, with only the corresponding index transmitted. Unlike linear quantization, which uniformly partitions space into equal intervals, VQ divides the vector space into non-uniform regions whose shape and size depend on input distribution, with each region centroid serving as the representative codeword. VQ operates on sub-vectors of specified size rather than individual elements, enabling compression ratios corresponding to non-integer bits per element (i.e., float to be transmitted). Our VQ implementation is depicted in Figure 2 and Figure 3.

VQ components in the pipeline are as follows:

- **Chunk Quantizer:** Input tensor $T = \{t_1, t_2, \ldots, t_S\}$ of size S is divided into M smaller vectors of size C, $V = \{v_1, v_2, \ldots, v_M\}$, with padding applied when $S \mod C \neq 0$.
- **Vector Quantizer:** Each vector v_m is quantized by finding its nearest codeword h_ψ using Euclidean distance. Let V be the matrix of input vectors and H be the codebook matrix. The distance matrix is computed as:

$$D = \|V\|^2 \mathbf{1}_\Psi^\top + \mathbf{1}_M (\|H\|^2)^\top - 2VH^\top \tag{19}$$

where $D_{m\psi}$ represents the distance between vector v_m and codeword h_ψ, and Ψ is the codebook size. Computing D via matrix multiplications is GPU-efficient. Each vector is replaced by its nearest codeword index:

$$q_m = \arg\min_\psi \|v_m - h_\psi\|_2^2 \tag{20}$$

The output contains M indices. If the codebook has Ψ codewords, each index requires $\log_2 \Psi$ bits. Both Chunk Quantizer and Vector Quantizer are part of the encoder pipeline (Figure 2).

- **Codebook:** The codebook directly affects accuracy and compression ratio. It is generated by training on representative data via K-means clustering [29] with predetermined size: codewords are randomly initialized, vectors iteratively assigned to nearest codewords via Euclidean distance, and codewords updated as cluster centroids until reaching the predefined iteration. At the end of the chosen epochs, we retrieve the codebook filled with Ψ codewords of equal size C.
- **Vector Decoder:** The decoder receives the compressed vector and maintains a codebook copy. For each index, it performs a lookup to find the corresponding codeword, replaces the index with this codeword, and reconstructs an estimate of the original tensor (Figure 3).

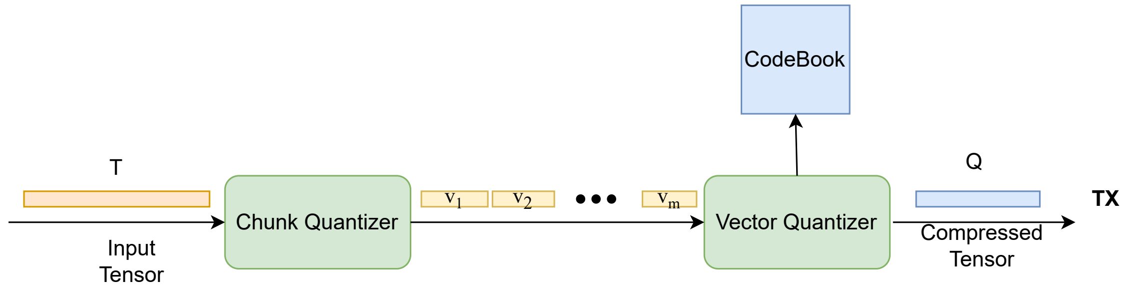


Figure 2: Encoder pipeline: input tensor T is divided into smaller vectors, quantized using a codebook, and replaced with codeword indices to produce compressed output Q.

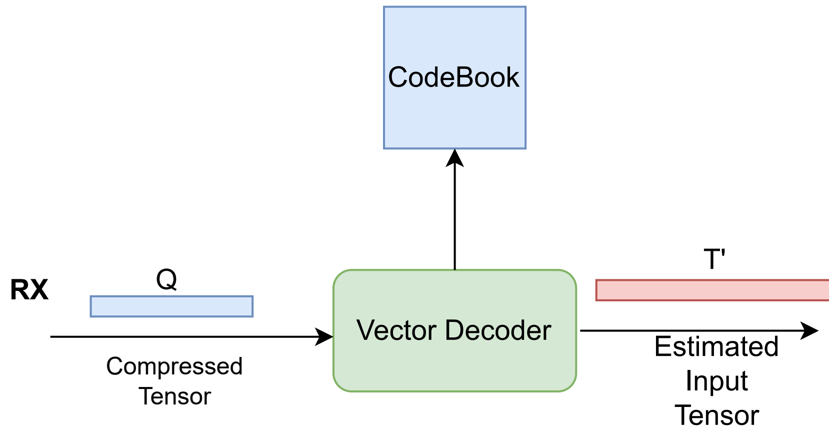


Figure 3: Decoder pipeline: compressed Q is received, indices are replaced with codewords to reconstruct tensor T', which is fed to the next model partition.

4.2 Reinforcement Learning Approach

Considering the challenges presented at the end of Section 3.2.7, we propose an RL approach, which learns optimal policies through interaction with the environment without requiring

explicit system models [18]. In RL, an agent learns to make sequential decisions by observing the environment state s, selecting an action a, receiving a reward R, and transitioning to a new state s'. Through repeated interactions, the agent learns to maximize cumulative long-term rewards by discovering which actions lead to favorable outcomes.

We formulate our problem as a Markov Decision Process (MDP) $\langle \mathbf{S}, \mathbf{A}, \mathbf{P}, R, \gamma \rangle$, where $\mathbf{S}$ is the state space, $\mathbf{A}(s)$ is the action set in state s, $P(s'|s,a)$ is the transition probability, $R(s,a,s')$ is the immediate reward, and $\gamma \in [0,1]$ is the discount factor.

The state captures system conditions and current configuration:

$$s = (l_{\text{SEW}}, l_{\text{phone}}, l_{\text{cloud}}, r_{\text{WiFi}}, r_{\text{5G}}, q_{12}, q_{23}). \tag{21}$$

where l_{SEW}, l_{phone}, l_{cloud} represent latency on the SEW, the phone and the cloud respectively, r_{WiFi} and r_{5G} denote WiFi and 5G network throughputs, and q_{12}, q_{23} are the active quantization levels on SEW→phone and phone→cloud links, respectively.

Each action specifies a partition configuration and data reduction schemes:

$$\mathbf{A}(s) = \{a^{ijk} : i \in |\mathcal{K}|, j \in |\mathcal{Q}|, k \in |\mathcal{Q}|\} \cup \{\eta\} \tag{22}$$

where a^{ijk} selects partition κ^i, quantization q^j for SEW→phone, and q^k for phone→cloud transfers. The action η maintains the current configuration. The total action space size is $|\mathcal{K}| \times |\mathcal{Q}|^2 + 1$. This action space can be reduced by filtering out actions yielding μ^{ijk} lower than a specified threshold or fixing a quantization scheme for a specific communication link.

Following [11], we define the immediate cost as a weighted sum of normalized components:

$$\begin{aligned} c(s,a,s') = \omega_{e_{\text{SEW}}} \frac{E_{\text{SEW}}}{E_{\text{SEW,max}}} + \omega_{e_{\text{phone}}} \frac{E_{\text{phone}}}{E_{\text{phone,max}}} + \omega_{\text{mAP}} c_{\text{mAP}} \\ + \omega_{\text{5G}} \frac{c_{5G}}{c_{5G,\text{max}}} + \omega_{\text{viol}} c_{\text{viol}} + \omega_{\text{rcfg}} c_{\text{rcfg}} \end{aligned} \tag{23}$$

where $c_{\text{mAP}} = 1 - \frac{\text{mAP} - \text{mAP}_{\text{min}}}{\text{mAP}_{\text{max}} - \text{mAP}_{\text{min}}}$ penalizes accuracy loss. The costs $c_{rcfg} = \mathbb{1}_{\{a \neq \eta\}}$ and $c_{lat}(s,a,s') = \mathbb{1}_{\{l_{total} > L_{max}\}}$ equal 1 when their conditions are met and 0 otherwise, discouraging frequent reconfiguration and penalizing latency violations, respectively. Each cost component is weighted by the importance weight ω associated to it. The reward is $R(s,a,s') = -c(s,a,s')$.

Since system dynamics are unknown, we employ Deep Q-Network (DQN) [30], a model-free algorithm that approximates the optimal action-value function using a neural network. The agent follows an ε-greedy strategy [18], exploiting the action yielding the highest estimated reward with probability $1-\varepsilon$ and exploring all actions with probability ε to balance learning and optimization.

5 Experimental Results

This section presents the experimental results obtained after profiling AI applications in a prototype environment and performing an extensive simulation campaign. Section 5.1 provides a brief overview of the experimental setup, followed by a discussion of the results in Section 5.

5.1 Experimental setup

Our reference system comprises: (1) an NVIDIA Jetson Orin Nano [31] representing the SEW, (2) a Samsung Galaxy S23 mobile device, and (3) a Dell Precision 5480 PC as the cloud server.

5.1.1 Applications and profiling

As AI applications, we utilized YOLOv5n [32] and YOLO-pose [33] deep learning models for object detection and human pose estimation, respectively. To determine the parameters defined in Section 3.2, we performed DNN partitioning and profiling for each model across the three devices, using the mobile phone as a WiFi hotspot. We considered both models in ONNX format [34] to support cross-platform inference and eliminate framework dependencies. The partitioning process yielded 104 and 170 candidate configurations for YOLOv5n and YOLO-pose, respectively (see Appendix A). Profiling determined the execution times on all devices (incorporated into state representation) and energy consumption (used for reward computation). Energy consumption was measured on the Jetson through the NVIDIA Power measurement API and on the Samsung S23 through ad hoc profiling, while data transmission power was estimated as described in [11] and in Appendix A.

We collected the WiFi throughput trace during profiling, observing a maximum uplink speed of 300 Mbps. For 5G connectivity, we utilized a real-world measurement dataset from [16], consisting of 11,024 samples with uplink throughput values ranging from 0 to 230.75 Mbps. We replayed these base traces with modifications to mitigate periodicity, ensuring sufficient training steps for each experiment. These modifications include the addition of random noise (±10% of the value at time step t), random shifts, and the inversion of the trace at its midpoint (see [11] for additional details).

We modeled the cloud server as an M/M/1 queue, where task arrivals follow a Poisson distribution, as described in Section 3.2.5. Cloud processing latency samples were drawn from an exponential distribution, with the mean equal to 2.5 the time determined by the profiled latency which comprises the decompression overhead and the execution time of the selected DNN configuration. This corresponds a maximum utilization on the server side equal to 60% as it is common in cloud systems [35], [36].

5.1.2 Compression evaluation setup

Quantization

The number of bits employed for compression is the key factor determining both compression ratio and tensor fidelity. Reducing the number of bits increases the compression ratio but decreases accuracy. Furthermore, the maximum number of repetitions that can be encoded with RLE is also constrained by the bit count, reducing compression efficiency for highly repetitive patterns. To find the optimal number of bits that balances accuracy and compression ratio, we experimented with a range of 6 to 8 bits. In these results, compression ratio is expressed as the number of bits per float, computed using (24).

$$R_c = \frac{\text{Number of elements in compressed tensor * quantization bits}}{\text{number of float elements in the input tensor } T} \tag{24}$$

Vector Quantization

We learn a codebook of 1024 vectors (codewords) to represent intermediate tensor data, stored locally on each device. Following common practice [25], [37], we use 1024 codewords to balance compression, latency, and memory. During encoding, tensors are divided into non-overlapping chunks of length C, with each chunk replaced by its nearest codeword index. With 1024 codewords requiring 10 bits per index, compression ratio is $10/C$ bits per float. After experimenting with $C \in \{40, 20, 10, 5, 3\}$, we selected $C \in \{5, 3\}$ based on accuracy requirements and tensor dimensions, yielding 2 and 3.33 bits/float respectively. Larger C enables more aggressive compression but may reduce quality. Compression ratio is computed using (25).

$$R_c = \frac{\text{Codeword index size * Number of chunk vectors } M}{\text{number of float elements in the input tensor } T} \tag{25}$$

Dataset for Evaluation

For object detection, we utilize 2788 images extracted from ImageNet videos [38]. The VQ codebook is trained on a custom dataset from our laboratory and evaluated on the ImageNet images. For human pose estimation, we reuse the VQ codebook trained for object detection and evaluate on 5000 images from the MS COCO validation set [39]. Mean average precision (mAP) is computed for every combination of split points (layers selected at split point 1 and 2) and compression schemes across all validation images for both tasks.

5.1.3 Training parameters and agent validation

We built the DQN agent using a fully connected neural network with three hidden layers (512, 1024, and 512 neurons) and ReLU activation functions. The agent was trained for up to 10^5 steps with a discount factor $\gamma = 0.99$. We defined strict latency thresholds of 200 ms and 150 ms for object detection and human pose estimation, respectively, to ensure good user experience. Cost importance weight selection reflects system priorities: SEW battery life is most critical ($\omega_{e_{SEW}} = 0.75$), followed by accuracy preservation ($\omega_{mAP} = 0.15$) and latency compliance ($\omega_{viol} = 0.06$). Phone energy receives lower weight ($\omega_{e_{phone}} = 0.03$) as phones have larger batteries than SEW devices. Reconfiguration and 5G costs are penalized lightly ($\omega_{rcfg} = \omega_{5G} = 0.005$). After training, the RL agent was evaluated for 10^5 steps across 10 different WiFi and 5G network traces with varying cloud latency sequences. We generated these evaluation traces from base network traces using the same augmentation process as for training but with 10 different random seeds. The results reported in subsequent sections are averaged across these 10 validation runs.

We considered different scenarios that helped assess the benefit of the proposed approach and the specificity of each compression schemes. In *Quantization* scenario the RL agent is trained on quantization-only at both split points, selecting among 6, 7, and 8-bit quantization. *Comp. RLE* represents a scenario where quantization-only or RLE can be chosen at both split points, including mixed strategies (quantization at split point 1 with run-length encoding at split point 2, or vice versa). *Comp. VQ* represents the proposed approach where VQ is applied at split point 1, with options for quantization-only or run-length encoding at split point 2, using 6, 7, or 8 bits. Even though VQ yields a high compression ratio, we do

not consider it for the second split point since the compression time exceeded 1.5s (about 10× the application threshold) because the phone lacks a hardware accelerator. *No Compression* denotes a scenario where intermediate tensors are transmitted without undergoing any compression. Note that *No Compression* is exactly the scenario for one RL agent in [15].

Also, we consider two baselines: *All Local*, which executes the entire DNN locally on the SEW and *Neurosurgeon* [7], a SOTA algorithm that selects optimal partition points to minimize either latency or energy based on previous samples of the network throughput using layer-wise prediction models. We extended it from single to dual split points and configured it for energy minimization to enable fair comparison. For each of these scenarios, we run 10 validation runs and report the averaged results in the next subsection.

To keep the inference accuracy high and acceptable, we filtered out actions (i.e., combination of a configuration and compression schemes at split point 1 and 2) that yield accuracy (i.e., mAP) below 90%. Table 1 report the number of actions for object detection and human pose estimation after the filtering process.

Table 1: Action space sizes after filtering (mAP $\geq$ 90%)

Scenario	**Object detection**	**Pose estimation**
Quantization	458	308
Comp. RLE	622	618
Comp. VQ	201	131
No compression	104	170

5.2 Results

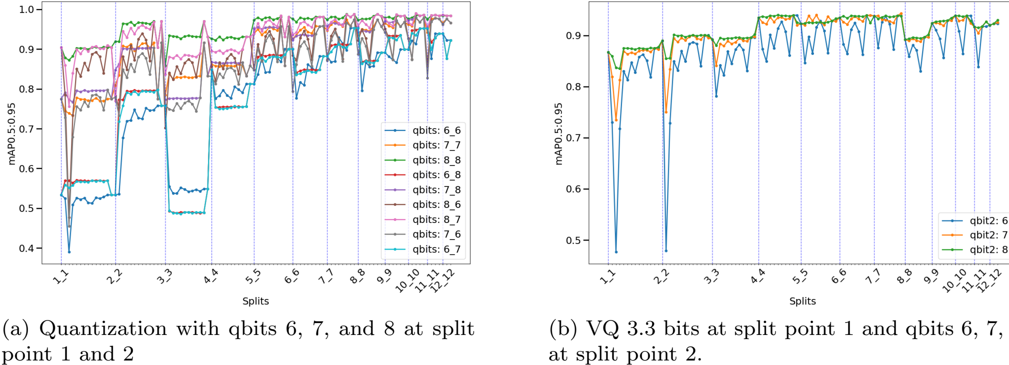
Accuracy Results

We extend our evaluation by comparing two compression methods across different accuracy-compression trade-offs. Results are reported in Table 2 and Table 3 for Compression (Q+RLE) and VQ, respectively, at a single split point. Each table shows the compression scenario (number of bits for quantization or number of elements in each chunk vector for VQ), the compression ratio (compressed size relative to original tensor size in bits), and bits per float element computed using (24) and (25), respectively. These metrics, together with the corresponding average accuracy, indicate which method is most efficient at each split point. The results show that VQ achieves approximately 30% (compared to 7-bit) to 45% (compared to 8-bit) more size reduction compared to Compression (Q+RLE) while maintaining similar average accuracy.

Table 2: Achieved Compression criteria with Q+RL

Compression bits	**Bit per float (avg)**	**Compressed (avg)/Original size**	**Mean Accuracy**
8	6.4	19.7%	93%
7	4.8	15.1%	85%
6	3.4	10.8%	73%

Exploiting the results reported in Table 2 and Table 3, we chose different compression scenarios in each split point. As illustrated in Figure 1, we have two split points in which we quantize before sending to the next device, resulting in double accuracy reduction. For



(a) Quantization with qbits 6, 7, and 8 at split point 1 and 2

(b) VQ 3.3 bits at split point 1 and qbits 6, 7, 8 at split point 2.

Figure 4: Accuracy comparison for all possible combinations of split point 1, split point 2, and different quantization schemes.

Table 3: Achieved Compression criteria with VQ

Chunk size	Bit per float	Compressed/Original size	Mean Accuracy
5	2	6.7%	71.32%
3	3.3	10.4%	89.92%

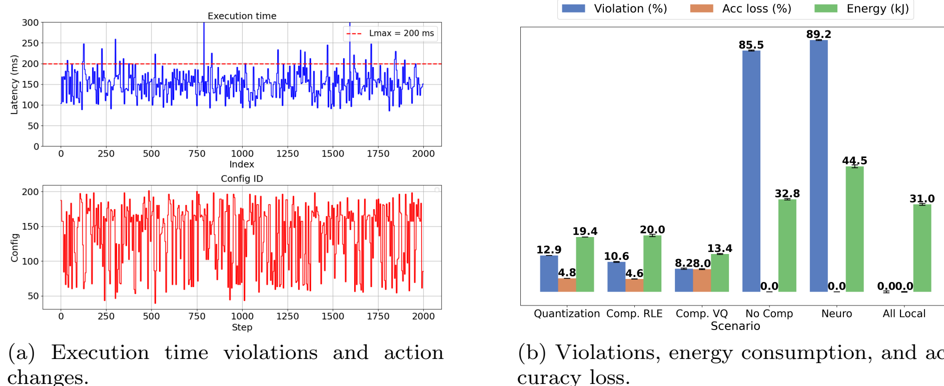
analyzing the final accuracy model, we have experimented with different methods and got the final accuracy results for each partitioning scenario. Figure 4a reports the mean average precision (mAP) for the 6, 7, and 8-bit quantization, illustrating all possible split points of a YOLOv5n DNN. For each quantization level, the reported results are with respect to the non-quantized YOLOv5n version, treating its output as the ground-truth. The accuracy results are the same for the compression (Q + RLE), since RLE is lossless and does not introduce any further accuracy loss. Each vertical line shows the configuration in which, the phone does not perform any computations and just transfers the received quantized tensor to the cloud to be completed. Starting from one vertical line, until the next one, we fix the initial split point and adjust the second point for all possible partitions, the runtime is partitioned twice, and intermediate tensors are quantized at each iteration.

Figure 4b reports double compression accuracy results applying VQ with 3.3 bits per element at the first split point, while the second split point uses the compression (Q+RLE), since VQ is too slow on the phone. Comparing Figure 4a and Figure 4b, we conclude that with VQ, choosing the 3 element chunk sizes, we can achieve comparable accuracy to Quantization with 7 and 8 bits.

RL Agent with Object Detection Task

Figure 5a demonstrates (for a representative validation run leveraging the *Comp. VQ* scenario) that the RL agent successfully learns an effective policy, maintaining execution times below the threshold most of the time. This is further confirmed in Figure 5b, where the DQN agent achieves an average violation rate of only 8.2% (±0.18%, representing the standard deviation across 10 validation runs). Moreover, Figure 5a illustrates the agent ability to dynamically adapt its actions based on system conditions.

The comparative analysis in Figure 5b evaluates six distinct scenarios: the proposed *Comp. VQ*, five alternative approaches described earlier in Section 5.1.3. The plot reports execution time violations, accuracy loss from compression, and SEW energy consumption



(a) Execution time violations and action changes.

(b) Violations, energy consumption, and accuracy loss.

Figure 5: Execution time violations, quantization levels, and performance comparison across different data reduction scenarios for object detection.

for each scenario. Error bars in the figure represent the standard deviation, providing approximate 95% confidence intervals. The low coefficient of variation across all metrics (CV $< 8\%$) demonstrates measurement reliability and consistent performance.

Results show that quantization significantly improves both execution time compliance and energy efficiency. The *Quantization* scenario achieves a 12.9% violation rate (CV = 2.58%) compared to 85.5% for *No Compression*, with only 4.9% accuracy loss. For energy consumption, *Quantization* reduces consumption by 44.1% and 30.6% compared to *No Compression* and *All Local*, respectively. The *Comp. RLE* scenario demonstrates additional benefits beyond quantization alone, further reducing violations to 10.6% while providing 8.3% and 36.5% energy savings compared to *Quantization* and *All Local*, respectively. This approach incurs comparable accuracy loss to *Quantization* (4.8% vs. 4.6%).

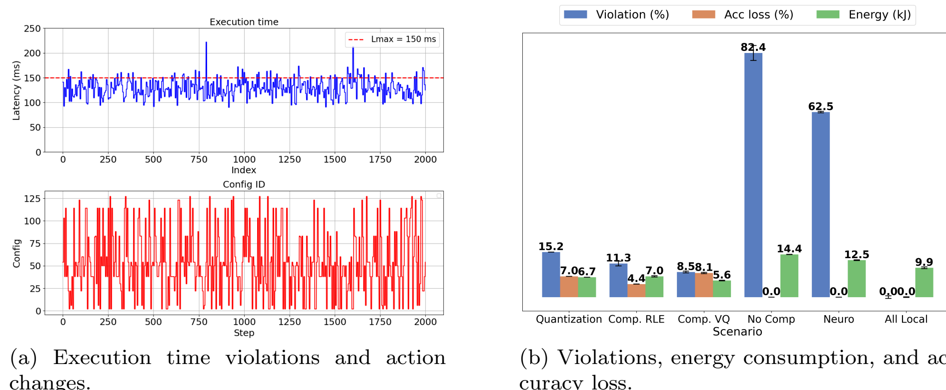
The proposed *Comp. VQ* achieves the best overall performance with the lowest violation rate of 8.2% ($\pm$0.22%) and energy savings of 14.7%, 21.9%, 45.8%, and 56.4% compared to *Comp. RLE*, *Quantization*, *All Local*, and *No Compression* respectively. While *Comp. VQ* incurs higher accuracy loss (8.0% $\pm$ 0.15%) compared to *Quantization* or *Comp. RLE* (4.8% and 4.6%), this is inherent to vector quantization lossy compression. Nevertheless, given the energy efficiency benefits, *Comp. VQ* provides the best trade-off between energy consumption and accuracy loss. These results validate the effectiveness of applying VQ at the first split point while selecting between quantization and RLE at the second split point, as VQ significantly reduces tensor size on the SEW, which translates into reduced data transfer energy consumption.

Human Pose Estimation

This section analyzes the RL agent performance for human pose estimation, comparing the proposed *Comp. VQ* against the same four operational scenarios used for object detection. It is important to note that, for *Comp. VQ* scenario, we reused the codebook trained for object detection without retraining to assess the robustness of the proposed approach. Recall that, results represent averages across 10 independent validation runs with a 150 ms latency threshold.

As for object detection, Figure 6a refers to a representative validation run leveraging the *Comp. VQ* scenario. The figure shows that the agent successfully learns an effective pol-

icy, maintaining execution times predominantly below the 150 ms threshold. The temporal analysis of agent decisions (figure below) demonstrates adaptive behavior through selection of different action which incorporate VQ, quantization and RLE. Figure 6b reveals several



(a) Execution time violations and action changes.

(b) Violations, energy consumption, and accuracy loss.

Figure 6: Execution time violations, quantization levels, and performance comparison across different data reduction scenarios for human pose estimation. Error bars represent the standard deviation across 10 validation runs.

key insights into the effectiveness of the proposed data reduction technique and RL policy. As for object detection, the *Comp. VQ* scenario achieves optimal balance across all metrics: 8.5% violation rate ($\pm$0.29%, CV = 3.42%), 5.6 kJ energy consumption, and 8.1% accuracy loss—representing the most efficient configuration for practical deployment. The low coefficient of variation for violations between pose estimation (3.42%) and object detection (2.69%) demonstrates the robustness and consistency of the proposed approach across different DNN architectures. In contrast, *Comp. RLE* and *Quantization* exhibit higher violation rates of 11.3% ($\pm$0.80%) and 15.2% ($\pm$0.43%), consuming 25% and 19.6% more energy than *Comp. VQ*, respectively. Additionally, *Comp. VQ* provides 43.4% and 61.1% energy savings compared to *All Local* and *No Compression* respectively, highlighting the importance of distributed execution for resource-constrained SEW devices. These results demonstrate the robustness of the proposed method, which delivers strong performance across different DNN architectures, as highlighted in Figure 6b.

The benefits of tensor compression are evident across both applications. The *No Compression* scenario suffers from high violation rates (exceeding 80%) and the highest energy consumption in both object detection and human pose estimation. The *All Local* execution achieves zero violations and zero accuracy loss, as expected, but at the cost of significantly higher energy consumption–84.5% and 76.8% more than the proposed *Comp. VQ* for object detection and human pose estimation, respectively. This highlights the benefits of distributed execution with tensor compression. Across all scenarios, measurement variability remains low (CV $<$ 8%), confirming the reliability of the experimental results and the statistical significance of the observed performance differences (detailed statistical analysis provided in Appendix C).

Comparison with Neurosurgeon

We compare *Comp. VQ* with *Neurosurgeon* [7], a SOTA DNN partitioning framework that optimizes split points but does not employ tensor compression.

For object detection, Figure 5 shows that *Neurosurgeon* incurs 89.2% ±0.37% violations, failing in nearly 9 out of 10 frames because uncompressed tensors saturate the SEW communication bandwidth. *Comp. VQ* achieves 90.8% fewer violations (8.2% ± 0.22%) and 69.9% energy savings (13.4 kJ vs. 44.5 kJ) through adaptive compression. The 8.0% ±0.15% accuracy loss is an acceptable trade-off for practical real-time operation. Similar trends are observed in Figure 6 for human pose estimation. *Neurosurgeon* incurs 62.4% ±5.04% violations, failing in 6 out of 10 frames. *Comp. VQ* achieves 86.4% fewer violations (8.5% ± 0.29%), 8.1% ±0.20% accuracy loss, and 55.1% energy savings (5.6 kJ vs. 12.5 kJ).

The consistent advantage across both tasks (90.8% and 86.4% violation reductions) demonstrates that compression-aware partitioning is essential for SEW devices, where communication constraints cannot be addressed through partition optimization alone.

6 Conclusion

This paper presented an RL-based framework for dynamic runtime management of AI applications on SEW devices, integrating adaptive multi-stage compression (VQ, quantization, and RLE) with intelligent task offloading across SEW-phone-cloud infrastructure. The proposed DQN agent jointly optimizes partition configurations and compression strategies, adapting to real-time system conditions to balance energy efficiency, latency compliance, and inference accuracy.

Experimental validation on object detection (YOLOv5n) and human pose estimation (YOLO-pose) considering real hardware (NVIDIA Jetson Orin Nano, Samsung Galaxy S23) demonstrated substantial improvements over baseline approaches. The *Comp. VQ* strategy–combining vector quantization at the SEW-phone link (3.3 bits/element) with adaptive quantization/RLE selection at the phone-cloud link–achieved violation rates below 9%, reduced energy consumption by 45.8% compared to local execution and 61.1% compared to uncompressed offloading, while maintaining acceptable accuracy loss (8.0-8.1%).

Future work could validate on production multi-user cloud environments with realistic queueing delays and contention, extend to real SEW prototypes like Microsoft HoloLens beyond Jetson Orin Nano, explore Federated RL approaches [15] combined with our compression strategies to leverage collaborative learning benefits, and evaluate on multi-stage AI pipelines.

References

[1] G. Koutromanos and G. Kazakou, "Augmented reality smart glasses use and acceptance: A literature review," *Computers & Education: X Reality*, vol. 2, p. 100 028, 2023.

[2] D. Rossos, A. Mihailidis, et al., "Ai-powered smart glasses for sensing and recognition of human-robot walking environments," in *IEEE RAS/EMBS BioRob*, 2024, pp. 62–67.

[3] C. S. Ranganathan, V. Kannagi, et al., "A smart eyewear using iot and cnns for visual assistance," in *ICoICI*, 2024, pp. 461–466.

[4] A. Biglari and W. Tang, "A review of embedded machine learning based on hardware, application, and sensing scheme," *Sensors*, vol. 23, no. 4, 2023.

[5] W. Gao, Y. Guo, et al., "Efficient neural network compression inspired by compressive sensing," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 35, no. 2, pp. 1965–1979, 2024.

[6] E. Soufleri and K. Roy, "Network compression via mixed precision quantization using a multi-layer perceptron for the bit-width allocation," *IEEE Access*, vol. 9, pp. 135 059–135 068, 2021.

[7] Y. Kang, J. Hauswald, et al., "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," *ACM SIGPLAN Notices*, vol. 52, pp. 615–629, 4 2017.

[8] A. W. Kambale, H. Sedghani, et al., "Runtime management of artificial intelligence applications for smart eyewears," *IEEE/ACM UCC*, 2023.

[9] X. Tian, P. Xu, et al., "Energy-efficient dnn partitioning and offloading for task completion rate maximization in multiuser edge intelligence," *Wirel. Commun. Mob. Comput.*, vol. 2023, Jan. 2023.

[10] V. Sohn, S. Kim, et al., "Joint frame drop and object detection task offloading for mobile devices via rl with lyapunov optimization," *IEEE Transactions on Mobile Computing*, 2025.

[11] A. W. Kambale, H. Sedghani, et al., "Tabular reinforcement learning methods for artificial intelligence tasks offloading in smart eye-wears," *ACM Trans. Auton. Adapt. Syst.*, Oct. 2025.

[12] M. Yan, C. A. Chan, et al., "Modeling the total energy consumption of mobile network services and applications," *Energies*, vol. 12, no. 1, 2019.

[13] G. Li, L. Liu, et al., "Auto-tuning neural network quantization framework for collaborative inference between the cloud and edge," in *ICANN*, Springer, 2018, pp. 402–411.

[14] D. Carra and G. Neglia, "Dnn split computing: Quantization and run-length coding are enough," in *GLOBECOM*, IEEE, 2023, pp. 7357–7362.

[15] H. Sedghani, A. W. Kambale, et al., "Federated reinforcement learning for runtime optimization of ai applications in smart eyewears," in *2025 33rd International Conference on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2025, to appear.

[16] A. Narayanan, X. Zhang, et al., "A variegated look at 5g in the wild: Performance, power, and qoe implications," in *ACM SIGCOMM*, New York, NY, USA, 2021, pp. 610–625.

[17] Z. Liao, W. Hu, et al., "Joint multi-user DNN partitioning and task offloading in mobile edge computing," *Ad Hoc Networks*, vol. 144, p. 103 156, 2023.

[18] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction.* MIT press, 2018.

[19] B. Gong and X. Jiang, "Dependent Task-Offloading Strategy Based on Deep Reinforcement Learning in Mobile Edge Computing," *Wireless Communications and Mobile Computing*, vol. 2023, pp. 1–12, 2023.

[20] A. M. Abdelmoniem and M. Canini, "Dc2: Delay-aware compression control for distributed machine learning," in *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*, IEEE, 2021, pp. 1–10.

[21] H. Xu, C.-Y. Ho, et al., "Grace: A compressed communication framework for distributed machine learning," in *2021 IEEE 41st international conference on distributed computing systems (ICDCS)*, IEEE, 2021, pp. 561–572.

[22] G. Castellano, F. Pianese, et al., "Regularized bottleneck with early labeling," in *ITC 2022-34th International Teletraffic Congress*, 2022.

[23] S. Choi, H. Lee, et al., "A dynamic compression technique for efficient offloading of computation between mobile devices and cloud," in *2023 International Technical Conference on Circuits/Systems, Computers, and Communications (ITC-CSCC)*, 2023, pp. 1–6.

[24] X. Liu, L. Zhang, et al., "Aligned vector quantization for edge-cloud collabrative vision-language models," *arXiv preprint arXiv:2411.05961*, 2024.

[25] A. Van Den Oord, O. Vinyals, et al., "Neural discrete representation learning," *Advances in neural information processing systems*, vol. 30, 2017.

[26] A. K. Brillantes, E. Sybingco, et al., "Vehicle tracking in low frame rate scenes using instance segmentation," in *HNICEM*, 2022, pp. 1–5.

[27] T. Pulkkinen, J. K. Nurminen, et al., "Understanding WiFi Cross-Technology Interference Detection in the Real World," in *ICDCS*, 2020, pp. 954–964.

[28] D. Ardagna, G. Casale, et al., "Quality-of-service in cloud computing: modeling techniques and their applications," *J. Internet Serv. Appl.*, vol. 5, no. 1, 11:1–11:17, 2014.

[29] R. Gray, "Vector quantization," *IEEE Assp Magazine*, vol. 1, no. 2, pp. 4–29, 1984.

[30] V. Mnih, K. Kavukcuoglu, et al., "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[31] Nvidia, *Jetson Orin Nano*, https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/nano-super-developer-kit/, 2024.

[32] Ultralytics, *Yolov5*, https://github.com/ultralytics/yolov5, Accessed: 2024-06-01.

[33] R. Varghese and S. M., "Yolov8: A novel object detection algorithm with enhanced performance and robustness," in *2024 International Conference on Advances in Data Engineering and Intelligent Computing Systems (ADICS)*, 2024, pp. 1–6.

[34] O. developers, *Onnx: Open neural network exchange*, `https://github.com/onnx/onnx`, 2019.

[35] Amazon Web Services, *Target tracking scaling policies for Amazon EC2 Auto Scaling*, `https://docs.aws.amazon.com/autoscaling/ec2/userguide/as-scaling-target-tracking.html`, Accessed: 2024-11-14, 2024.

[36] A. Alasaad, K. Shafiee, et al., "Auto-scaling techniques in cloud computing: Issues and research directions," *Sensors*, vol. 24, no. 17, p. 5551, 2024.

[37] lucidrains, *Vector-quantize-pytorch*, https://github.com/lucidrains/vector-quantize-pytorch, 2024.

[38] J. Deng, W. Dong, et al., "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*, IEEE, 2009, pp. 248–255.

[39] Microsoft, *Coco keypoint detection challenge.* https://cocodataset.org/keypoints-2020/.

[40] Shenzhen Qway Technology Co., Ltd, *TC66/TC66C USB Type-C Power Meter User Manual*, [Online], Available: `https://gzhls.at/blob/ldb/2/5/2/8/0e6f880a0a23ad0ef7a16e5c990f2c5b0216.pdf`, 2020.

[41] B. L. Welch, "The significance of the difference between two means when the population variances are unequal," *Biometrika*, vol. 29, no. 3/4, pp. 350–362, 1938.

A Profiling Methodology and System Characterization

This section provides detailed information on the profiling methodology used to characterize the system parameters defined in Section 3.2.

A.1 Hardware and Software Setup

We evaluated two DNN models: YOLOv5n for object detection and YOLO-pose for human pose estimation. Both models were converted to ONNX format to support cross-platform inference and eliminate framework dependencies. We used ONNX Runtime (v1.13.1) for inference execution across three computational layers:

- **SEW**: NVIDIA Jetson Orin Nano featuring a 512-core NVIDIA Ampere GPU with 16 Tensor Cores and a 6-core Arm Cortex-A78AE v8.2 64-bit CPU (6GB RAM, Ubuntu 20.04).

- **Mobile Device**: Samsung Galaxy S23 equipped with Snapdragon 8 Gen 2 SoC (8GB RAM, Android 13).
- **Cloud Server**: Dell Precision 5480 PC with Intel Core i7-13800HX CPU (14 cores, 5.2 GHz max) and NVIDIA RTX A1000 GPU (16GB RAM, Ubuntu 22.04).

The devices were connected via WiFi 5 using the Samsung Galaxy S23 as a mobile hotspot, with the Jetson and cloud server as WiFi clients.

A.2 DNN Partitioning and Configuration Space

The ONNX format imposes structural constraints on partitioning as splits can only occur at join nodes. The partitioning identified 12 feasible split points for YOLOv5n and 16 for YOLO-pose.

Since our three-tier architecture involves two potential split points, one between SEW and phone, and one between phone and cloud, the total configuration space includes 105 and 171 unique partitioning configurations for YOLOv5n and YOLO-pose, respectively.

A.3 Execution Time Profiling

For each of the 105 (YOLOv5n) and 171 (YOLO-pose) configurations, we conducted 10 profiling runs to measure:

1. **Partition execution times**(t_d^i): Time to execute partition p_d^i on device d
2. **Compression overhead** ($l_{q,SEW}^{ij}$, $l_{q,phone}^{ik}$): Time to compress intermediate tensors using schemes q^j and q^k
3. **Decompression overhead** ($l_{-q,phone}^{ij}$, $l_{-q,cloud}^{ik}$): Time to decompress received tensors

The profiling procedure for each configuration was as follows:

1. **SEW**: Execute partition p_1^i, compress the intermediate tensor using scheme q^j, and transmit to the phone. Measure execution time, compression time, and tensor size.
2. **Phone**: Receive and decompress the tensor, execute partition p_2^i, compress the output using scheme q^k, and transmit to the cloud. Measure decompression time, execution time, compression time, and tensor size.
3. **Cloud**: Receive and decompress the tensor, execute partition p_3^i, and return the final result. Measure decompression time and execution time.

Execution times were averaged over 10 runs per configuration, with standard deviations typically below 5%. Table 4 in Section A.6 summarizes the parameter ranges observed across all configurations.

A.4 Energy Consumption Profiling

A.4.1 Computation Energy (SEW)

On the Jetson Orin Nano, we leveraged the NVIDIA Jetson Power API, which provides real-time power consumption measurements through software queries. For each configuration κ^i

and compression scheme q^j, we measured:

$$e_1^{ij} = \int_0^{T_{comp}} P_{Jetson}(t)\, dt \tag{26}$$

where $T_{comp} = t_1^i + l_{q,SEW}^{ij}$ is the total computation and compression time, and $P_{Jetson}(t)$ is the instantaneous power consumption queried at 10 Hz. This yields the per-request energy consumption e_1^{ij} (in joules) for executing partition p_1^i and compressing with scheme q^j. It is important to not that during the energy consumption profiling no data is sent to the next device.

A.4.2 Computation Energy (Phone)

For the Samsung Galaxy S23, direct power measurement APIs are not available. We therefore employed a TC66 USB-C power meter [40] connected inline between the charger and the phone to measure energy consumption indirectly. The procedure was:

1. Fully charge the phone to 100% battery capacity.
2. Disconnect the charger and run the target workload (decompression $q^j \rightarrow$ execution $p_2^i \rightarrow$ compression q^k) repeatedly in a loop until battery capacity drops from 100% to 95%.
3. Reconnect the phone to the TC66 power meter and recharge from 95% to 100%, recording the total energy delivered ($E_{recharge}$).
4. Compute per-iteration energy: $e_2^{ijk} = E_{recharge}/N_{iterations}$

A.4.3 Data Transmission Energy

To isolate data transmission energy from computation, we conducted separate profiling experiments where devices transmitted tensors of varying sizes without executing DNN partitions. The procedure was:

1. Generate synthetic tensors of sizes matching observed intermediate outputs (ranging from 1.5 MB to 6.25 MB).
2. Transmit tensors over WiFi while measuring power consumption using the NVIDIA API (Jetson) and TC66 meter (phone).
3. Compute average transmission power: $\theta_{SEW} = \frac{1}{N}\sum_{i=1}^{N} E_i/T_i$ where E_i is energy consumed and T_i is transmission time for tensor i.

This yielded average transmission power values of $\theta_{SEW} = 2.66$ W and $\theta_{phone} = 4.50$ W. Data transfer energy consumption for a given configuration is then computed as:

$$E_{transfer,SEW} = \theta_{SEW} \cdot \frac{\delta_{12}^{ij}}{r_{WiFi}} \tag{27}$$

$$E_{transfer,phone} = \theta_{phone} \cdot \frac{\delta_{23}^{ik}}{r_{5G}} \tag{28}$$

where δ_{12}^{ij} and δ_{23}^{ik} are the compressed tensor sizes (measured during profiling), and r_{WiFi}, r_{5G} are the observed throughputs.

A.5 Compression Scheme Profiling

For each of the three compression techniques (quantization, quantization + RLE, and vector quantization), we profiled:

- **Compression ratio**: $\delta^{ij}_{compressed}/\delta^{i}_{original}$
- **Accuracy impact**: mAP of compressed vs. uncompressed inference (μ^{ijk})
- **Compression/decompression overhead**: Time costs l^{ij}_{q} and l^{ij}_{-q}

To evaluate the impact of compression on the accuracy, we consider 2788 images from Imagenet dataset [38]. For the case of VQ, we trained the codebook on a custom dataset from our laboratory and evaluate on the Imagenet dataset. For human pose estimation, we consider the SOTA MS COCO dataset [39], especially the validation set which comprises 5000 images. This means the mAP for every combination split point 1, split point 2, compression schemes q^j and q^k is evaluated over 2788 and 5000 images for object detection and pose estimation, respectively.

A.6 Summary of Profiled Parameters

Table 4 presents the ranges of key parameters obtained from profiling across all configurations and compression schemes.

B DNN and environment parameters

Table 5 outlines the key parameters used for the DQN agent in our experiments.

C Statistical Analysis: Cross-Application Performance Validation

This section provides statistical validation of the proposed *Comp. VQ* approach robustness across Human Pose Estimation (PS) and Object Detection (OD) applications. Independent two-sample t-tests (Welch's t-test [41], $\alpha = 0.05$) were conducted on three key metrics–violations, accuracy loss, and energy consumption–using 10 independent validation runs per scenario.

C.1 Statistical Findings

C.1.1 Measurement Reliability

All metrics demonstrate good reliability with coefficients of variation below 8% across most scenarios (Table 6). The highest variability occurs in PS Comp. RLE violations (CV = 7.06%) and PS Quantization energy (CV = 4.32%), while most other measurements show consistency (CV $\leq$ 3.5%). This variability pattern confirms that observed differences reflect genuine performance characteristics rather than measurement noise.

Table 4: Summary of profiled parameter ranges

Parameter	YOLOv5n	YOLO-pose
Execution Time (ms)		
t_1^i (SEW partition)	0–38	0–47
t_2^i (Phone partition)	0–56	0–31
t_3^i (Cloud partition)	0–26	0–25
Compression Overhead (ms)		
$l_{q,SEW}^{ij}$ (6-8 bit Q)	2.1–7.0	2.0–5
$l_{q,SEW}^{ij}$ (6-8 bit RLE+Q)	4.2–15.5	2.5–9.2
$l_{q,SEW}^{ij}$ (VQ 3.3-bit)	7.0–15.2	4.0–11.8
$l_{q,phone}^{ik}$ (6-8 bit Q)	5.6–20.2	4.7–14.4
$l_{q,phone}^{ik}$ (6-8 bit Q+RLE)	7.8–25.2	5.4–16.8
$l_{-q,phone}^{ij}$ (decompression)	1.5–5.1	1.7–3.6
$l_{-q,cloud}^{ik}$ (decompression)	0.8–3.2	0.7–3.5
Tensor Sizes (MB)		
δ_{12}^i (uncompressed)	1.56–6.25	0.25–1
δ_{12}^{ij} (6-8 bit Q)	0.31–1.56	0.06–0.25
δ_{12}^{ij} (6-8 bit Q+RLE)	0.25–1.17	0.04–0.20
δ_{12}^{ij} (VQ 3.3-bit)	0.16–0.62	0.03–0.1
δ_{23}^{ik} (uncompressed)	1.56–6.25	0.25–1
δ_{23}^{ik} (6-8 bit Q)	0.31–1.56	0.06–0.25
δ_{23}^{ik} (6-8 bit Q+RLE)	0.25–1.17	0.04–0.20
Energy Consumption (mJ per request)		
e_1^{ij} (SEW compute+compress)	5–310	5–99
e_2^{ijk} (Phone compute+compress)	0–558	0–165

Table 5: Experiments parameters.

Objective parameters	
L_{max} (YOLOv5n, pose)	(200, 150) ms
$\omega_{e_{SEW}}$	0.75
$\omega_{e_{phone}}$	0.03
ω_{mAP}	0.15
ω_{5G}	0.005
ω_{viol}	0.06
ω_{rcfg}	0.005
Model hyper-parameters	
Discout factor γ	0.99
ϵ	0.05
lr	0.04
Replay buffer	10000
Batch size	128
Number of layers	3
Number of neurons per layer	(512, 1024, 512)
Activation function	ReLU
Dropouts	(0.4, 0.3, 0)
Discount factor γ	0.99
Rollout fragment length	5
Target update frequency	400

C.1.2 Comp. VQ Performance Validation

The proposed approach achieves optimal and consistent performance across both applications:

Violations: Near-identical rates of 8.47% (PS, CV = 3.42%) and 8.18% (OD, CV = 2.69%)—difference of only 0.29% ($p < 0.001$)–represent the **lowest violation rates** across all compression methods. The comparable variability between applications demonstrates robust performance despite using a codebook trained only for OD.

Accuracy Loss: Converges to approximately 8% for both applications (PS: 8.15% with CV = 2.45%, OD: 8.04% with CV = 1.86%)–a practically negligible difference of 0.11% ($p < 0.001$). This validates codebook generalization across different DNN architectures without retraining.

Energy Efficiency: Achieves **lowest energy consumption** (PS: 5.60 kJ with CV = 1.56%, OD: 13.40 kJ with CV = 1.34%) with the smallest OD/PS energy ratio (2.39×) among all compression methods, demonstrating particular effectiveness for computationally intensive tasks.

C.1.3 Neurosurgeon Performance Analysis

Neurosurgeon [7] represents a split computing baseline that dynamically partitions DNN execution between edge and cloud based on predicted latency, without employing data compression. The results reveal fundamental limitations of compression-free approaches:

Violations: Neurosurgeon exhibits violation rates of 62.40% (PS, CV = 8.07%) and 89.20% (OD, CV = 0.41%)—substantially higher than all compression-based methods. Notably, OD violations approach the *No Compression* baseline (85.51%), indicating that dynamic partitioning alone is insufficient for latency-constrained object detection.

Accuracy Loss: Both applications maintain 0% accuracy loss (matching *No Compression*), as Neurosurgeon transmits uncompressed intermediate tensors.

Energy Consumption: Neurosurgeon consumes 12.46 kJ (PS) and 44.49 kJ (OD)—representing 123.2% and 232.1% increases over *Comp. VQ*, respectively. The energy overhead of transmitting uncompressed tensors significantly reduces battery life compared to compression-based approaches.

The contrast between Neurosurgeon and *Comp. VQ* (62.40% vs. 8.47% PS violations, 89.20% vs. 8.18% OD violations) demonstrates that **adaptive compression is essential** for meeting real-time latency constraints in SEW split computing. The 8% accuracy loss under *Comp. VQ* is a worthwhile trade-off for 87% (PS) and 91% (OD) violation reduction, coupled with high energy savings.

C.1.4 Comparative Performance

Relative to the no compression baseline (violation rates >80%), *Comp. VQ* provides:

- **89.7%** (PS) and **90.4%** (OD) violation rate reduction and it is the best among all methods
- **45–56%** energy savings compared to *No Compression*
- Quantization: 81.6% (PS) and 84.9% (OD) reduction compared to *No Compression*
- Comp. RLE: 86.2% (PS) and 87.6% (OD) reduction compared to *No Compression*

While *Comp. VQ* incurs higher accuracy loss (∼8%) compared to Quantization (4.8–7.0%) or RLE (4.4–4.6%), it provides substantial energy savings:

- **vs. Quantization:** 16.4% (PS) and 31.0% (OD) energy reduction
- **vs. Comp. RLE:** 20.0% (PS) and 33.0% (OD) energy reduction

For resource-constrained SEW devices, this represents a favorable trade-off where 8% accuracy loss is acceptable given the extended operational lifetime.

C.2 Statistical Significance

All differences achieve statistical significance at $p < 0.001$ (except accuracy loss under no compression, where both = 0%). Cohen's d effect sizes often exceed 2.0, indicating substantial practical differences. However, the 0.11% accuracy difference between PS and OD under *Comp. VQ*, while statistically significant, is practically negligible.

C.3 Summary

This statistical analysis provides rigorous evidence that *Comp. VQ* achieves: (1) lowest violation rates with near-identical cross-application performance (difference < 0.3%), (2) best energy efficiency with 16–33% savings over alternatives compression methods, and (3) acceptable accuracy loss (∼8%) that generalizes across DNN architectures without codebook retraining. The consistent performance across applications validates the approach robustness for real-world SEW AI applications deployment.

Table 6: Statistical comparison of Human Pose Estimation (PS) vs. Object Detection (OD). Standard deviations represent 2× sample std (approximate 95% CI). P-values computed using Welch's t-test ($\alpha = 0.05$).

Scenario	Metric	PS (Pose Est.)			OD (Object Det.)			Sig.
		Mean	**Std**	**CV (%)**	**Mean**	**Std**	**CV (%)**	
Quantization	Violations (%)	15.20	0.43	2.82	12.89	0.33	2.58	***
	Accuracy Loss (%)	7.01	0.09	1.30	4.79	0.12	2.50	***
	Energy (kJ)	6.70	0.29	4.32	19.43	0.33	1.70	***
Comp. RLE	Violations (%)	11.34	0.80	7.06	10.62	0.16	1.49	***
	Accuracy Loss (%)	4.41	0.015	0.33	4.56	0.05	1.09	***
	Energy (kJ)	7.00	0.21	3.00	20.00	0.42	2.10	***
Comp. VQ	Violations (%)	8.47	0.29	3.42	8.18	0.22	2.69	***
	Accuracy Loss (%)	8.15	0.20	2.45	8.04	0.15	1.86	***
	Energy (kJ)	5.60	0.11	1.56	13.40	0.18	1.34	***
No Comp.	Violations (%)	82.41	0.26	0.31	85.51	0.18	0.20	***
	Accuracy Loss (%)	0.00	0.00	0.00	0.00	0.00	0.00	ns
	Energy (kJ)	14.41	0.12	0.83	32.80	0.52	1.58	***
Neurosurgeon.	Violations (%)	62.40	5.04	8.07	89.20	0.37	0.41	***
	Accuracy Loss (%)	0.00	0.00	0.00	0.00	0.00	0.00	ns
	Energy (kJ)	12.46	0.08	0.64	44.49	0.48	1.08	***

Note: CV = Coefficient of Variation (std / mean × 100%). Significance codes: *** $p < 0.001$, ** $p < 0.01$, * $p < 0.05$, ns = not significant. All differences are statistically significant at $p < 0.001$ except accuracy loss under no compression (both values = 0, test not applicable). $n = 10$ independent validation runs per scenario. Latency thresholds: 200ms (OD), 150ms (PS).