Enhancing the Performance of Optimization Algorithms for Offloading Tasks in Mobile-Edge Computing Networks

Yohanes Armenian Putra [1] and Hilal Hudan Nuha [2,*]

[1] Telkom Indonesia, Indonesia
[2] School of Computing, Telkom University, Bandung, Indonesia
Email: yohanesar@student.telkomuniversity.ac.id (Y.A.P.); hilalnuha@ieee.org (H.H.N.)
*Corresponding author

***Abstract*—In the rapidly evolving field of Mobile-Edge Computing (MEC), the demand for efficient Deep Reinforcement Learning (DRL) algorithms is critical due to the constraints of computational resources and the need for real-time processing. This paper introduces Optimized Nadam, an enhanced variant of the Nadam optimizer, specifically designed to address these challenges. By eliminating the computationally intensive product term, Optimized Nadam significantly reduces computational overhead while ensuring robust performance across varying load conditions. Experimental results demonstrate that Optimized Nadam achieves substantial reductions in Total Time Consumed, outperforming standard Nadam by up to 37.6% under typical load scenarios. Furthermore, the algorithm consistently exhibits a lower Average Time Per Channel, indicating superior convergence speed. Optimized Nadam maintains a high Normalized Computation Rate in dynamic environments with fluctuating load conditions, closely aligning with Nadam, thus showcasing its resilience and adaptability. The observed reduction in training loss across all test scenarios underscores Optimized Nadam's efficiency in achieving rapid and stable convergence. These findings position Optimized Nadam as a viable alternative to traditional optimization algorithms such as Adam and Nadam, particularly in resource-constrained MEC deployments where computational efficiency and real-time processing are paramount.**

***Keywords*—deep reinforcement learning, mobile-edge computing, Nadam, optimization algorithms, optimized Nadam**

I. Introduction

Mobile-Edge Computing (MEC) has emerged as a crucial paradigm in modern network architectures, enabling low-latency and high-bandwidth services by offloading computational tasks from mobile devices to edge servers [1, 2]. MEC's ability to bring computation closer to the data source helps reduce delay, which is critical for real-time applications such as autonomous driving, augmented reality, and smart cities [3, 4]. However, the dynamic nature of these applications presents significant challenges in optimizing offloading decisions, particularly in resource-constrained environments.

Recent advancements in Deep Reinforcement Learning (DRL) have shown promise in addressing these challenges, as DRL can effectively handle complex environments with varying network conditions [5, 6]. The Nadam optimizer, which integrates Nesterov momentum into the Adam algorithm [7], has been widely used in DRL due to its robustness and convergence properties. The Adam algorithm, introduced by Kingma and Ba, is known for its adaptive learning rate and has become a standard in training deep learning models [8]. However, the computational complexity associated with Nadam, particularly in resource-constrained environments like MEC, poses challenges. The overhead introduced by the u product term in Nadam increases both time and computational power required for training models [9]. Here, u represents the momentum term, which is defined mathematically as follows:

$$u_t = \beta_1 m_t + (1 - \beta_1) g_t \quad (1)$$

TABLE I. List of Notations

Symbol	Description
u_t	Momentum term at time step t
m_t	First moment estimate at time step t
g_t	Gradient at time step t
β_1	Exponential decay rate for the first moment estimates
β_2	Exponential decay rate for the second moment estimates
v_t	Second moment estimate at time step t

m_t is the first moment estimate, g_t is the gradient at time step and β_1 is the exponential decay rate for the first moment estimates. Table I provides a list of necessary symbols to clarify the notations used throughout this paper. To address these issues, this paper introduces Optimized Nadam, an optimized variant of Nadam that eliminates the

Manuscript received October 9, 2024; revised December 31, 2024; accepted March 4, 2025; published June 13, 2025.

u product component. This modification aims to reduce computational overhead while maintaining performance, thus making it more suitable for MEC environments. Optimized Nadam seeks to enhance the efficiency of DRL applications in MEC by streamlining the optimization process, ultimately contributing to improved real-time decision-making capabilities in dynamic network scenarios.

II. System Model and Methods

The core of this research lies in modifying the Nadam optimizer by removing the u product term, which is traditionally used for bias correction in the momentum update equation. The original Nadam algorithm updates parameters. θ_{t+1} using the following equations:

$$m_t = \beta_1 . m_{t-1} + (1 - \beta_1) . g_t \,, \tag{2}$$

$$v_t = \beta_2 . v_{t-1} + (1 - \beta_2) . g_t^2 \,, \tag{3}$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \,, \tag{4}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \,, \tag{5}$$

$$\theta_{t+1} = \theta_t - \alpha . \frac{\hat{m}_t}{\sqrt{\hat{v}_t + \epsilon}} \,, \tag{6}$$

where m_t and v_t are the first and second moment estimates, and β_1, β_2 are decay rates [9, 10]. In Optimized Nadam, the u product term, which affects the bias correction, is removed, simplifying the update rule and reducing computational complexity. This modification is particularly beneficial in MEC environments, where computational resources are limited and efficiency is paramount [8, 11].

The u product term is removed from the Nadam optimizer by directly modifying the momentum update equation. In the original Nadam, the momentum term u_t is defined as in Eq. (1). By eliminating this term, the update rule becomes more straightforward, reducing the number of computations required during each iteration. This simplification leads to lower overhead, particularly in environments where computational resources are constrained.

A. System Workflow and Flowchart

The workflow of the system implemented is shown in Fig. 1. It outlines the key stages in optimizing offloading decisions in MEC using DRL.

The workflow begins with Data Loading, where input data, including channel gains and other network parameters, is processed to train the DRL model. Data Processing involves normalization and batching for efficient learning.

The Model Initialization (MemoryDNN) phase sets up the MemoryDNN architecture. Optimizer Configuration configures the optimization algorithms, including Optimized Nadam, for training. The Model Training process involves iterative training using the preprocessed data and configured algorithms.

The Offloading Decision Process (DROO Algorithm) uses DRL to determine the optimal offloading strategy based on real-time network conditions. This decision feeds into the Resource Allocation Optimization, where network resources are dynamically allocated.

Memory Management ensures effective memory use, while Model Evaluation assesses performance. The Model Saving step preserves the trained model, and Validation Testing ensures it generalizes well to unseen data.

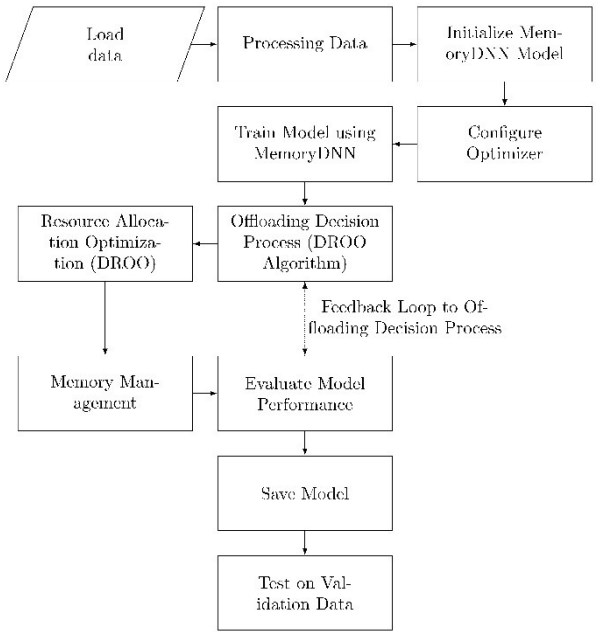

Fig. 1. System workflow for MEC offloading optimization [12].

B. System Design

The system design is illustrated in Fig. 2, showing the components and their interactions within the MEC environment.

Key components include:

1) **Channel Gain Input** (h_t): Wireless channel gain for each time frame.
2) **Deep Neural Network** (DNN): Processes input data, generating relaxed offloading actions.
3) **Quantization**: Converts DNN-generated actions into discrete binary actions.
4) **Convex Optimization Solver**: Computes Q-values for possible actions.
5) **Replay Memory**: Stores past experiences for model improvement.
6) **Training Samples and Batch Sampling**: Randomly selects samples for training.
7) **Offloading Policy Update**: Updates policy based on training results.

The data used in this research is sourced from real-time network monitoring systems that provide online streams of channel gain and other relevant parameters. The characteristics of the data include:

- Channel Gain (h_t): This represents the wireless channel conditions at each time frame, which is critical for making informed offloading decisions. The channel gain data is typically collected in real-time from network sensors or simulation environments that model wireless communication.
- Network Parameters: Additional features such as latency, bandwidth, and user demand are also collected to provide a comprehensive view of the network state. These parameters are essential for training the DRL model to optimize offloading strategies effectively.

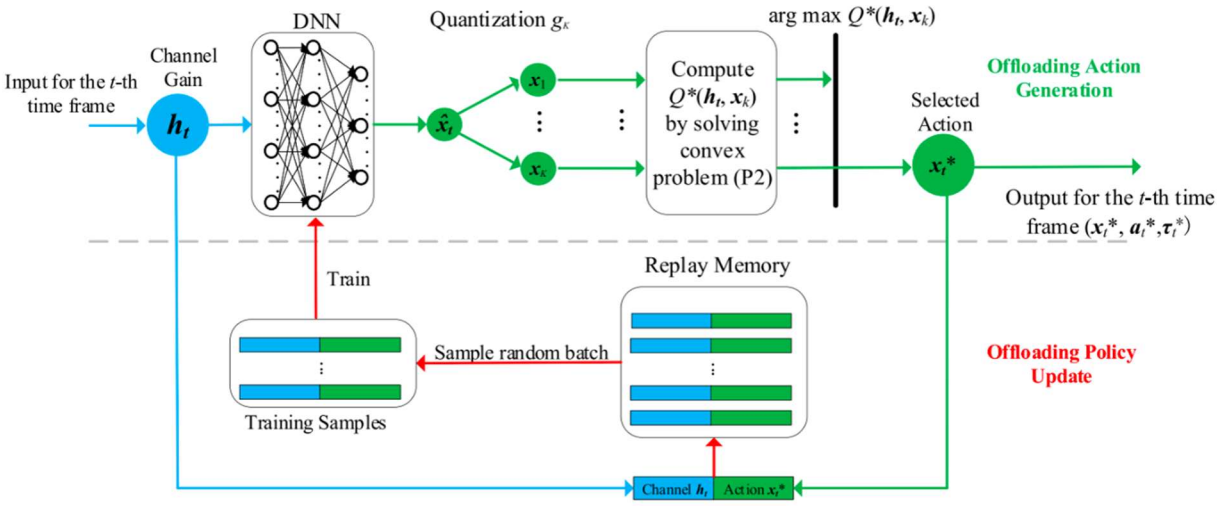

Fig. 2. System design for DRL-Based offloading in MEC [12].

The data is continuously streamed and compiled into a structured format suitable for analysis. Each data point includes timestamps, channel gain values, and associated network parameters, allowing for dynamic updates and real-time decision-making.

The flow of inputs through these components is crucial for the system's functionality. The channel gain input feeds directly into the DNN, which processes this information to generate potential offloading actions. These actions are then quantized and evaluated by the convex optimization solver, which determines the best course of action based on the computed Q-values. The results are stored in replay memory, allowing continuous learning and adaptation of the offloading policy.

C. Implementation of Deep Reinforcement Learning

The implementation of DRL in this research follows a systematic approach, which includes the following steps:

1) Data Acquisition: The dataset used in this research is sourced from the DROO GitHub repository. This dataset has already undergone data cleaning and preprocessing, ensuring that it is free from inconsistencies and noise. The data includes relevant features such as channel gain and other network parameters, which are essential for training the DRL model.
2) Model Initialization: The DRL model is initialized with a Deep Neural Network (DNN) architecture. The parameters of the DNN are set randomly at the beginning to allow for effective learning during training.
3) Training Process: The model is trained iteratively using the preprocessed dataset. During each iteration, the model learns to optimize offloading decisions based on the input features, adjusting its parameters to minimize the loss function.
4) Offloading Decision Making: The trained model utilizes the Online DROO Algorithm to make real-time offloading decisions based on the current network conditions. This involves generating relaxed offloading actions, quantizing them into discrete actions, and selecting the optimal action based on computed Q-values.
5) Model Evaluation: After training, the model's performance is evaluated using metrics such as Total Time Consumed, Average Time Per Channel, and Training Loss. This evaluation helps in assessing the effectiveness of the DRL approach in optimizing offloading tasks in Mobile-Edge Computing environments.
6) Continuous Learning: The model incorporates a mechanism for continuous learning, where it updates its knowledge based on new data and experiences gathered during operation. This adaptability is crucial for maintaining performance in dynamic network conditions.

D. Online DROO Algorithm

To evaluate the effectiveness of Optimized Nadam in a DRL context, we implemented an Online DROO Algorithm designed for offloading decisions in MEC. The algorithm dynamically adjusts offloading based on real-time conditions. The steps are outlined below:

Algorithm 1 Online DROO Algorithm for Offloading Decision

Input: Wireless channel gain h_t at each time frame t, number of quantized actions K
Output: Offloading allocation x_t^* for each time frame t
Initialize the DNN with random parameters θ_1 and empty memory;
Set iteration number M and training interval δ;
for $t = 1, 2, \ldots, M$ **do**
  Generate a relaxed offloading action $\tilde{x}_t = f_\theta(h_t)$;
  Quantize $\tilde{x}_t$ into K binary actions $\{x_k\} = g_K(\tilde{x}_t)$;
  Compute $Q \times (h_t, x_k)$for all $[x_k]$ by solving (P2);
  Select best action $x_t^* = arg\ max\ Q \times (h_t, x_k)$;
  Update memory by adding (h_t, x_t^*);
  if t mod $\delta = 0$ **then**
    Uniformly sample a batch of dataset $\{(h_\tau, x_\tau^*)\} \epsilon\ T(\tau)$ from memory;
    Train DNN with $\{(h_\tau, x_\tau^*)\} \epsilon\ T(\tau)$ and update θ_t using other DRL algorithm;
  end if
end for

1) Explanation of the online DROO algorithm

The Online DROO Algorithm optimizes offloading decisions in Mobile-Edge Computing (MEC) using Deep Reinforcement Learning (DRL). The algorithm iteratively learns from real-time network conditions to make optimal offloading decisions. Here's a brief overview of the key steps:

1) **Input:** The algorithm **takes** the wireless channel gain ht at each time frame and the number of quantized actions K, which define the offloading decision space.
2) **Output**: The output is the optimal offloading allocation x_t^* for **each time** frame, determining whether tasks should be processed locally or offloaded to the edge server.
3) **Initialization**: The Deep Neural Network (DNN) is initialized with random parameters θ_1 , and the memory is set to **empty**. The DNN approximates Q-values to guide the decision-making process.
4) **Iteration Loop**: The algorithm runs for M iterations, during which it:
   - Generates a relaxed offloading action $\tilde{x}_t$ based on h_t.
   - Quantizes $\tilde{x}_t$ into K binary actions.
   - Computes $Q * (h_t, x_k)$ for each action.
   - Selects the best action x_t^* that maximizes the Q-value.
   - Updates memory with the state-action pair (h_t, x_t^*).
   - Trains the DNN every δ iteration using samples from memory to adapt to network changes.
5) **Completion:** After M iterations, the DNN has learned an optimal **offloading** policy that dynamically adapts to real-time conditions.

This algorithm enables real-time, dynamic offloading decisions in MEC, improving performance and reducing computational overhead by using an optimized version of the Nadam algorithm (Optimized Nadam). The iterative training process and use of quantized actions make the algorithm both flexible and robust, suitable for deployment in diverse MEC environments.

III. Performance Evaluation

The performance of Optimized Nadam was evaluated through simulations under various load conditions. The experiments focused on Total Time Consumed, Average Time Per Channel, and Training Loss, comparing Optimized Nadam with Adam, Adadelta, Adagrad, Adamax, Nadam, and Ftrl.

A. Normal Load Conditions

Under normal load conditions, Optimized Nadam showed a significant reduction in Total Time Consumed compared to Nadam, especially in scenarios wth 10,000 and 5,000 iterations. The reduction in total time is attributed to the removal of the uproduct term, which streamlines the computation process. The results are presented in Table II and visually depicted in Fig. 3.

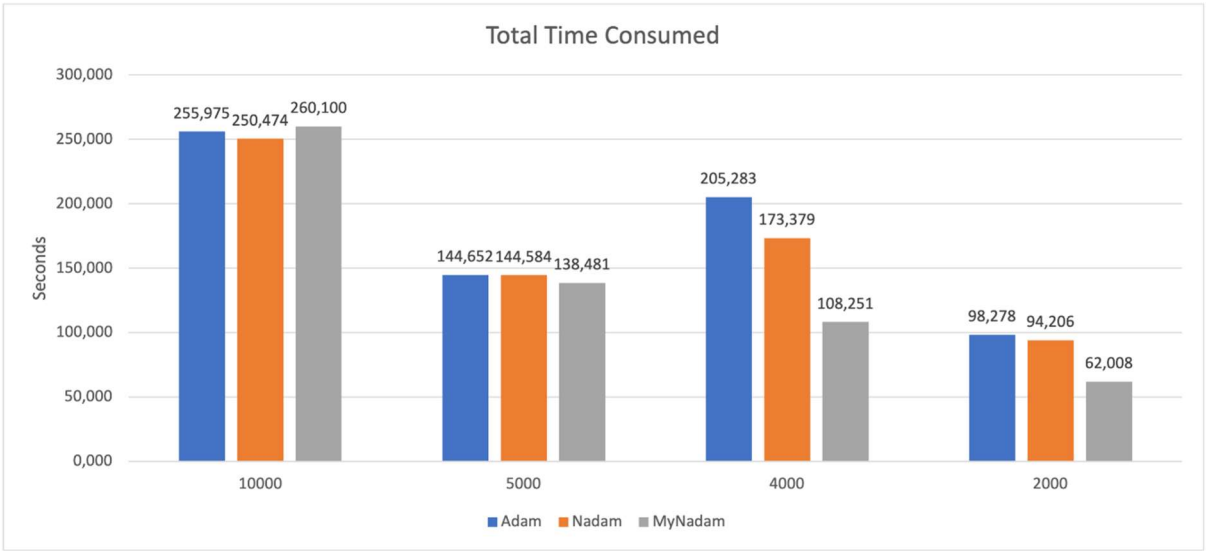

Fig. 3. Total time consumed by algorithms under normal load.

TABLE II. TOTAL TIME CONSUMED BY ALGORITHMS UNDER NORMAL LOAD (IN SECONDS)

Algorithms/ Iteration	10,000	5,000	4,000	2,000
Adam	255.975	144.652	205.283	98.278
Adadelta	311.818	174.661	208.503	110.094
Adagrad	353.660	181.412	207.054	110.296
Adamax	261.482	134.820	181.488	91.778
Ftrl	331.055	189.276	227.085	117.891
Nadam	250.474	144.584	173.379	94.206
Optimized Nadam	260.100	138.481	108.251	62.008

Table II clearly shows that Optimized Nadam consistently reduced the total computation time as compared to the standard Nadam and Adam algorithms, particularly under lower iteration counts. For instance, with 4,000 iterations, Optimized Nadam completed the task in 108.251 seconds, a marked improvement over Nadam's 173.379 seconds and Adam's 205.283 seconds. This demonstrates the efficiency gained by modifying the optimizer, which is especially critical in resource-constrained MEC environments.

In terms of Average Time Per Channel, Optimized Nadam consistently outperformed the other algorithms, particularly in high iteration scenarios. The absence of the uproduct term in Optimized Nadam allows it to achieve faster convergence, resulting in lower average time per channel. The detailed data is shown in Table II, and the corresponding bar chart is displayed in Fig. 4.

Table III highlights that Optimized Nadam consistently achieves a lower average time per channel across various iterations. For example, at 4,000 iterations, Optimized Nadam had an average time per channel of 0.0271 seconds, significantly lower than Nadam's 0.0433 seconds and Adam's 0.0513 seconds. This reduction in time per channel not only demonstrates the efficacy of the Optimized Nadam algorithm but also underscores its suitability for real-time applications where every millisecond counts.

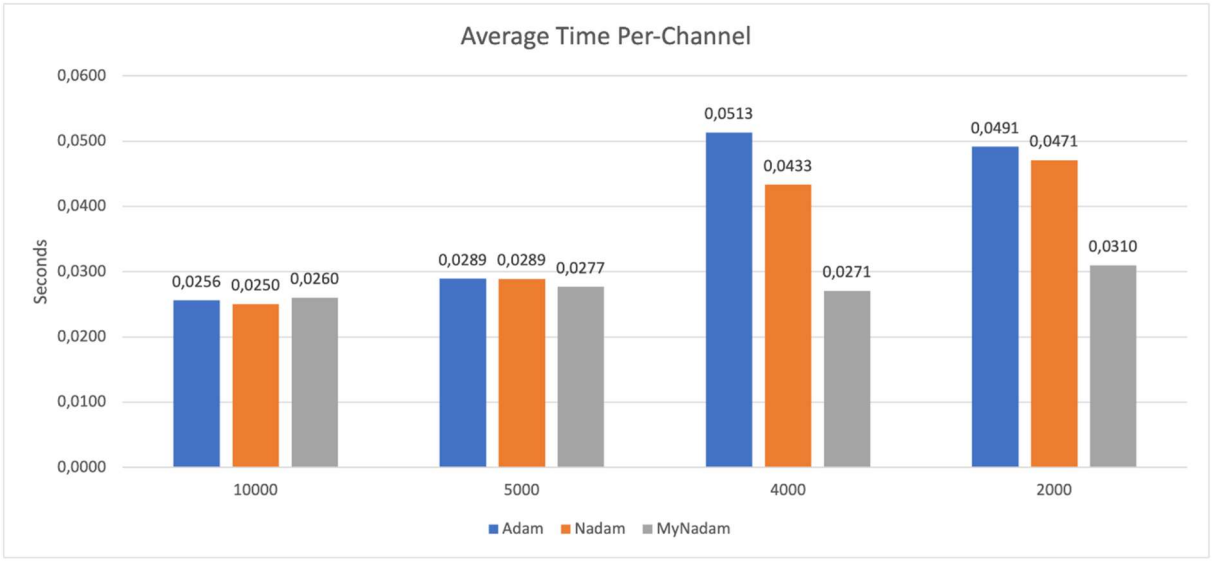

Fig. 4. Average time per channel for algorithms under normal load.

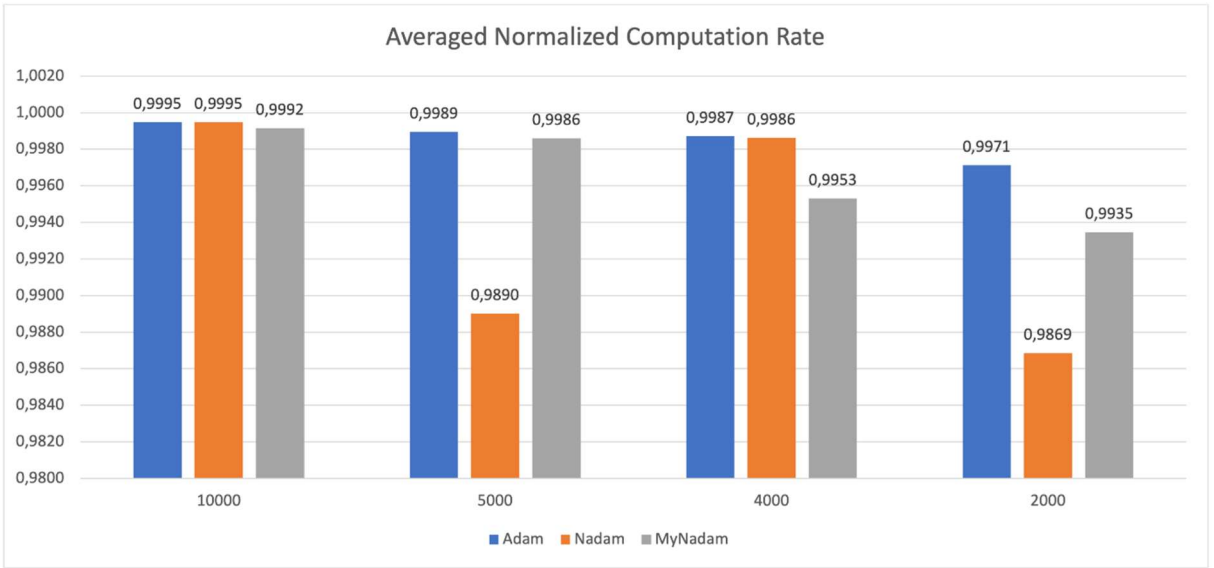

Fig. 5. Normalized computation rate under alternate load.

TABLE III. AVERAGE TIME PER CHANNEL FOR ALGORITHMS UNDER NORMAL LOAD (IN SECOND)

Algorithms/ Iteration	10,000	5,000	4,000	2,000
Adam	0.0256	0.0289	0.0513	0.0491
Adadelta	0.0312	0.0349	0.0521	0.0550
Adagrad	0.0354	0.0363	0.0518	0.0551
Adamax	0.0261	0.0270	0.0454	0.0459
Ftrl	0.0331	0.0379	0.0568	0.0589
Nadam	0.0250	0.0289	0.0433	0.0471
Optimized Nadam	0.0260	0.0277	0.0271	0.0310

In summary, both the Total Time Consumed and Average Time Per Channel metrics indicate that Optimized Nadam provides a substantial performance boost over the other algorithms, particularly in scenarios involving lower iteration counts, which are more common in practical MEC environments.

B. Alternate Load Conditions

Optimized Nadam demonstrated robust performance by maintaining a stable Normalized Computation Rate in alternate load conditions, where the network load fluctuates. As shown in Fig. 5, Optimized Nadam closely matched Nadam and outperformed Adam in most cases, highlighting its adaptability in dynamic environments.

The analysis of training loss further substantiates the effectiveness of Optimized Nadam. As illustrated in Fig. 6, the training loss decreases rapidly and remains stable, suggesting that removing the u product term does not hinder the optimizer's ability to minimize loss. This trend is evident across all iterations tested, where Optimized Nadam consistently demonstrates lower or comparable loss values compared to Nadam and Adam, reinforcing its efficiency in handling dynamic load conditions.

TABLE IV. NORMALIZED COMPUTATION RATE UNDER ALTERNATE LOAD

Algorithms/ Iteration	10.000	5.000	4.000	2.000
Adam	0.9992	0.9988	0.9985	0.9967
Adadelta	0.9351	0.8957	0.9267	0.9042
Adagrad	0.9800	0.9753	0.9707	0.9526
Adamax	0.9818	0.9953	0.9957	0.9885
Ftrl	0.9224	0.9418	0.9269	0.9408
Nadam	0.9992	0.9992	0.9984	0.9941
Optimized Nadam	0.9970	0.9979	0.9969	0.9955

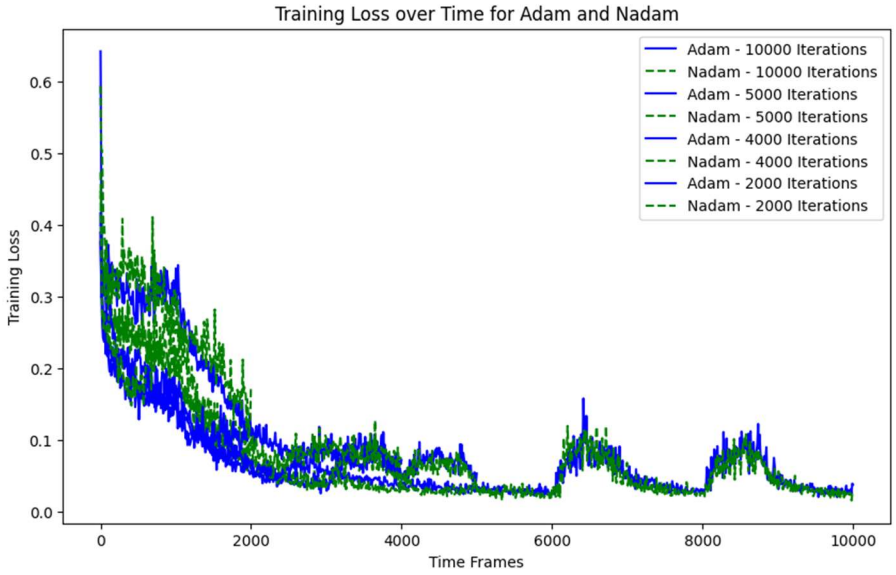

Fig. 6. Training loss under alternate load conditions.

IV. DISCUSSION

The experiments highlight the effectiveness of Optimized Nadam in Mobile-Edge Computing (MEC) environments, particularly under varying load conditions. This section summarizes key findings and discusses their implications for MEC deployments.

A. Reduction in Computational Complexity

Optimized Nadam successfully reduces the computational complexity associated with the original Nadam algorithm, primarily by eliminating the u product term. As shown in Tables II and II, this modification significantly reduces Total Time Consumed and Average Time Per Channel across all iteration scenarios. For example, at 4,000 iterations, Optimized Nadam completed the task 37.6% faster than Nadam, demonstrating its efficiency. This enhanced convergence speed is particularly crucial for real-time applications in MEC environments.

B. Robustness under Fluctuating Load Conditions

Optimized Nadam also demonstrated strong performance under fluctuating load conditions, maintaining stable Normalized Computation Rates that closely matched those of Nadam, as shown in Table III. For instance, at 10,000 iterations, Optimized Nadam achieved a computation rate of 0.9970, only slightly lower than Nadam's 0.9992, indicating robust performance even in dynamic environments. This stability ensures that MEC systems can consistently deliver low-latency services even when network conditions vary.

C. Training Loss and Convergence

The algorithm's ability to minimize training loss was another key area of improvement. As seen in Figure 6, Optimized Nadam exhibited a rapid and stable decrease in training loss, often outperforming both Nadam and Adam. This suggests that the simplifications introduced by Optimized Nadam do not compromise its optimization capabilities; rather, they enhance its ability to achieve faster and more stable convergence, which is particularly valuable in resource-constrained MEC environments.

D. Implications for MEC Deployments

The consistent performance improvements provided by Optimized Nadam make it a strong candidate for real-world MEC applications, where computational efficiency and rapid response times are critical. Its ability to reduce overhead while maintaining high performance makes it suitable for large-scale deployments, where scalability and efficiency are paramount. Additionally, its robustness under fluctuating load conditions ensures consistent service levels, supporting a wide range of applications from low-latency communication to high-throughput data processing.

V. Conclusion

Optimized Nadam is an optimized variant of the Nadam algorithm, aimed at reducing computational complexity while maintaining performance, specifically for Mobile-Edge Computing (MEC) environments. Through extensive evaluations, Optimized Nadam demonstrated its ability to outperform traditional optimizers like Adam and Nadam regarding time efficiency and stability. Optimized Nadam achieved significant reductions in both Total Time Consumed and Average Time Per Channel under various load conditions, while maintaining a high Normalized Computation Rate even in dynamic network environments. The removal of the u product term streamlined the computation process without negatively affecting the optimizer's performance, making it highly suitable for real-time, resource-constrained MEC applications.

The analysis of training loss further underscored the effectiveness of Optimized Nadam, showcasing rapid convergence and stable loss minimization across different scenarios. The algorithm's adaptability under fluctuating load conditions highlights its robustness, ensuring consistent service levels and reduced latency, critical factors in MEC deployments.

A. Future Directions

Future research could further enhance Optimized Nadam by integrating adaptive mechanisms that dynamically adjust its parameters in response to real-time network conditions. This would improve its adaptability in highly dynamic MEC environments. Additionally, exploring the application of Optimized Nadam in other deep learning contexts, such as large-scale data centers or distributed AI systems, could extend its utility and provide further insights into its versatility. In conclusion, Optimized Nadam presents a compelling solution for optimizing Deep Reinforcement Learning (DRL) models in MEC environments, offering a well-balanced approach to improving both computational efficiency and performance. Its promising results lay a strong foundation for future work in both MEC-specific scenarios and broader deep-learning applications.

Conflict of Interest

The authors declare no conflict of interest.

Author Contributions

YAP develops the program and runs the experiments; HHN develops the theoretical methods; both authors had approved the final version.

Funding

The financial support of the Indonesia's DRTPM, DITJEN DIKTIRISTEK, KEMDIKBUDRISTEK through grant 106/E5/PG.02.00.PL/2024, 043/SP2H/RT-MONO/ LL4/2024, and 077/LIT07/PPMLIT/ 2024 is hereby acknowledged and appreciated.

References

[1] N. Abbas, Y. Zhang, A. Taherkordi, and T. Skeie, "Mobile edge computing: A survey," *IEEE Internet Things J.*, vol. 5, no. 1, pp. 450–465, 2018. doi: 10.1109/JIOT.2017.2750180

[2] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, "A survey on mobile edge computing: The communication perspective," *IEEE Commun. Surv. Tutorials*, vol. 19, no. 4, pp. 2322–2358, 2017.

[3] X. Wang, J. Li, Z. L. Ning, Q. Song, L. Guo, S. Guo, and M. S. Obaidat, "Wireless powered mobile edge computing networks: A survey," *ACM Comput. Surv.*. 2023. doi: 10.1145/3579992

[4] J. Z. K. H. Y. Mao, C. You, and K. B. Letaief, "Mobile edge computing: Survey and research outlook," *IEEE Commun. Surv. Tutorials*, vol. 19, no. 4, pp. 2322–2358, 2017.

[5] X. Zhang, D. Wu, and D. Niyato, "Droo: Deep reinforcement learning-based online offloading in mobile-edge computing with proportional resource allocation," *IEEE Trans. Wireless Commun.*, vol. 19, no. 10, pp. 6884–6899, 2020.

[6] Z. Akhavan *et al*., "Deep reinforcement learning for online latency aware workload offloading in mobile edge computing," in *Proc. GLOBECOM 2022-2022 IEEE Global Communications Conference*, 2022, pp. 2218–2223.

[7] T. Dozat, "Incorporating Nesterov momentum into Adam," in *Proc. Int. Conf. Learn. Represent.*, 2016, pp. 1–4.

[8] M. D. Zeiler, "Adadelta: An adaptive learning rate method," arXiv preprint arXiv:1212.5701, 2012.

[9] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," arXiv preprint arXiv:1412.6980, 2014.

[10] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *J. Mach. Learn. Res.*, vol. 12, pp. 2121–2159, 2011.

[11] Y. Fan and X. Cai, "A deep reinforcement approach for computation offloading in MEC dynamic networks," *EURASIP J. Adv. Signal Process.*, vol. 2024, no. 1, p. 48, 2024. doi: 10.1186/s13634-024-01142-2

[12] L. Huang, S. Bi, and Y. J. A. Zhang, "Deep reinforcement learning for online computation offloading in wireless powered mobile edge computing networks," *IEEE Trans. Mobile Comput.*, vol. 19, no. 11, pp. 2581–2593, 2020. doi: 10.1109/TMC.2019.2928811