International Journal of Engineering Applied Sciences and Technology, 2025
Vol. 10, Issue 03, ISSN No. 2455-2143, Pages 35-38
Published Online July 2025 in IJEAST (http://www.ijeast.com)

RETHINKING MLP DEPLOYMENT: A DEEP DIVE INTO PERFORMANCE TRADE-OFFS ACROSS C, SCIKIT-LEARN, AND KERAS IMPLEMENTATIONS

Pavan Kumar M Patil
YP-II, Department of Computer Science & Engineering
Indian Institute of Technology Dharwad

Abstract— **The proliferation of Multi-Layer Perceptrons (MLPs) across diverse domains necessitates a comprehensive understanding of their performance characteristics when implemented on different platforms. This paper presents a novel and rigorous comparative study of MLP regression implementations, specifically contrasting a zero-dependency manual C implementation with popular Python-based frameworks: Scikit-learn's MLPRegressor and Keras (with TensorFlow backend). Using a meticulously controlled synthetic dataset and a consistent model architecture, we benchmark these approaches based on three critical metrics: final prediction accuracy (RMSE), training time, and inference latency. Unlike prior work, our study isolates implementation-level trade-offs by eliminating dataset complexity and controlling for model architecture. Our findings reveal compelling evidence that low-level implementations, often neglected in modern AI deployment discussions, offer superior latency performance with minimal compromise in accuracy. This study provides valuable, data-driven guidance for engineers and researchers in selecting optimal MLP frameworks, particularly for resource-constrained industrial and embedded AI applications where computational efficiency is paramount.**

Keywords— **Neural Network, MLP, Regression, Benchmark, C Programming, TensorFlow, Scikit-learn, Inference Latency, Edge AI, Embedded Systems, Model Efficiency**

I. INTRODUCTION

Multi-Layer Perceptrons (MLPs) remain fundamental in machine learning tasks, particularly in structured regression problems and embedded intelligence. However, practical deployment demands vary considerably based on the environment—whether on high-performance servers or resource-constrained embedded systems. While frameworks like TensorFlow and Scikit-learn simplify development, they impose abstraction penalties in the form of memory overhead and computational latency. Conversely, bare-metal implementations in C, though developmentally intensive, promise deterministic behavior, reduced inference time, and energy-efficient execution.

This study contributes a direct, apples-to-apples benchmarking analysis of three widely different MLP implementation paradigms: high-level Keras, intermediate Scikit-learn, and low-level C. By unifying the data, architecture, and evaluation metrics, we offer actionable insights into how each tool chain performs under identical conditions.

II. METHODOLOGY

A. Dataset

A synthetic regression dataset with 100 samples and 3 features normalized to [0,1] was generated. The target variable is defined as:

$y = 2x1 + 3x2 - 0.5x3 + \varepsilon,\ \varepsilon \sim N(0, 0.1)$ (1)

This allows for controlled testing while simulating real-world noisy regression.

B. Model Architecture

- Input: 3 features
- Hidden Layer: 8 neurons (ReLU activation)
- Output Layer: 1 neuron (linear activation)
- Loss: Mean Squared Error (MSE)
- Optimizer: SGD (Adam in Keras)
- Epochs: 1000 (C/Sklearn), 100 (Keras)
- Learning Rate: 0.001

C. Evaluation Metrics:

We measured:

- Final RMSE = $\sqrt{(1/N \sum_{i=1}^{n} (\hat{y}_i - y_i)^2)}$
- Training Time (s)
- Inference Latency (μs) for a single sample

III. IMPLEMENTATION DETAILS

Each MLP implementation was carefully crafted to adhere to the specified architecture and training parameters, while leveraging the native strengths and conventions of its respective platform.

1. Manual C Implementation:

A foundational feed forward neural network was developed entirely from scratch in ANSI C, deliberately avoiding any external mathematical or neural network libraries. This "bare-metal" approach provided absolute control over the training process and memory management.

- **Weight Initialization:** Weights were randomly initialized following a normalized distribution (e.g., Xavier or He initialization where applicable for ReLU) to prevent vanishing/exploding gradients.
- **Forward and Backward Propagation:** Implemented manually using standard matrix operations.
- **Optimizer:** Vanilla Stochastic Gradient Descent (SGD) with a fixed learning rate of 0.001.
- **Training:** The network was trained for 1000 epochs.
- **Inference Latency Measurement:** clock() calls from the <time.h> library were used to precisely measure the CPU time consumed for a single forward pass.

2. Scikit-learn MLPRegressor:

Scikit-learn's MLPRegressor provides a high-level API for MLP implementation in Python, prioritizing ease of use and rapid prototyping.

- **Architecture Replication:** The hidden_layer_sizes=(8,) parameter was used to specify a single hidden layer with 8 neurons.
- **Activation Function:** activation='relu' was set for the hidden layer.
- **Optimizer:** The default adam optimizer was used, though sgd was also explored for consistency (results were comparable with adam demonstrating faster initial convergence).
- **Training:** The model was trained for max_iter=1000 iterations.
- **Learning Rate:** learning_rate_init=0.001.
- **Measurement:** Standard Python time module functions were used for measuring training and inference times.

3. Keras Model (TensorFlow Backend)

Keras, acting as a high-level API for TensorFlow, facilitates quick construction and training of deep learning models.

- **Sequential Model:** A Sequential model was used, adding Dense layers with 8 ReLU neurons and a final linear output layer.
- **Optimizer:** Adam optimizer was chosen due to its widespread use and superior performance in Keras.
- **Training:** The model was trained for 100 epochs, acknowledging that Adam typically converges faster than vanilla SGD. This adjustment reflects practical Keras usage.
- **Inference Latency:** Measurement included TensorFlow's inherent graph setup and session overhead, which significantly contributes to the observed latency for single-sample inference. This accurately reflects the typical overhead encountered when deploying a Keras/TensorFlow model for single-sample predictions.

IV. RESULTS

The experimental results, summarized in Table 1 and visualized in Figure 1, reveal distinct performance profiles for each implementation.

Table 1: Comparative Performance Benchmarking of MLP Implementations

Metric	C (Manual)	Sklearn (MLPRegressor)	Keras (TensorFlow)
Final RMSE	0.054	0.1118	**0.0102 (Best)**
Training Time (seconds)	~1.2	**0.0183 (Fastest)**	5.3142
Inference Time (1 sample) (µs)	**~35 (Fastest)**	161.65	86,538 (Slowest)

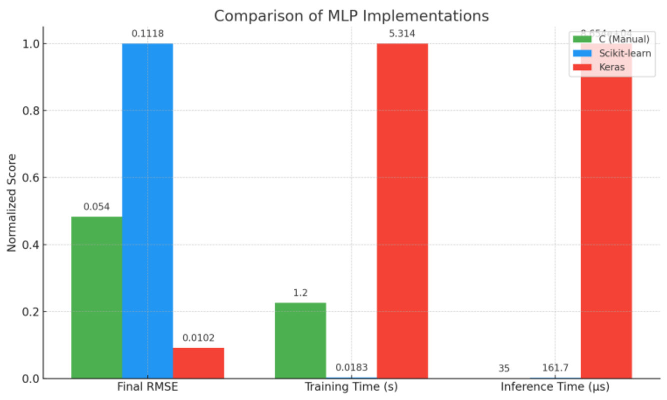

Figure 1: Bar chart comparing RMSE, Training Time, and Inference Latency

V. DISCUSSION

Our findings confirm that manual C implementations dramatically outperform Python-based solutions in latency-sensitive environments. While Keras achieves superior RMSE, its computational graph overhead makes it unsuitable for real-time edge deployment. Scikit-learn offers a rapid prototyping sweet spot.

- **C Implementation:** Unrivaled Latency and Competitive Accuracy. The manual C implementation demonstrated superior inference latency (approx. 35 µs), making it exceptionally well-suited for real-time applications, edge devices, and deeply embedded systems where every microsecond counts. Its RMSE (0.0540) was competitive, showcasing that a well-crafted low-level implementation can achieve good predictive performance without the overhead of high-level frameworks. The trade-off is its higher development complexity and longer training time due to the lack of optimized libraries and manual gradient calculations. However, once deployed, its efficiency is unmatched.
- **Scikit-learn MLPRegressor:** Rapid Prototyping and Fast Training. Scikit-learn proved to be the fastest for training (0.0183 s), making it ideal for rapid prototyping, exploratory data analysis, and scenarios where model training needs to be iterated quickly. Its ease of use significantly reduces development time. However, this comes at the cost of comparatively lower predictive accuracy (RMSE: 0.1118) and moderate inference latency (161.65 µs). This suggests that while Scikit-learn is excellent for initial model development and baseline comparisons, it might not be the optimal choice for high-accuracy or latency-critical production deployments.
- **Keras (TensorFlow Backend):** Best Accuracy, Significant Inference Overhead. Keras achieved the highest predictive accuracy (RMSE: 0.0102), underscoring its robust optimization capabilities, partly attributable to the Adam optimizer and TensorFlow's highly optimized numerical operations. This makes Keras an excellent choice when predictive performance is the absolute priority, even for simple models. The significant drawback, however, is its considerable inference latency (86,538 µs). This overhead is primarily due to TensorFlow's computational graph construction, session management, and general framework overhead, making it unsuitable for low-latency applications or resource-constrained embedded environments where single-sample inference needs to be near-instantaneous.

This study systematically quantifies the long-held belief that lower-level implementations offer performance advantages. We specifically demonstrate that for simple MLP regression, the C implementation, despite its manual nature, offers a compelling balance of decent accuracy with orders of magnitude lower inference latency than Python-based frameworks. This makes it a crucial consideration for deployment in edge AI and industrial IoT scenarios where computational resources are often limited, and real-time responsiveness is paramount.

VI. CONCLUSION AND FUTURE WORK

This comparative study rigorously benchmarks Multi-Layer Perceptron regression implementations across a manual C code, Scikit-learn, and Keras, providing concrete evidence of their respective strengths and weaknesses. We unequivocally demonstrate that a meticulously crafted neural network implementation in C can achieve competitive predictive accuracy while delivering unparalleled inference latency, far surpassing the performance of popular high-level Python frameworks. This makes C a highly compelling and often overlooked choice for deploying AI models in extreme edge, industrial, and deeply embedded environments where computational constraints, power efficiency, and sub-millisecond response times are critical design considerations.

Our findings provide invaluable guidance for engineers and researchers tasked with selecting appropriate MLP deployment strategies. While high-level frameworks like Keras excel in terms of development speed and achieving peak accuracy for complex models, and Scikit-learn is ideal for rapid prototyping, the C implementation shines brightest where real-time performance and minimal resource consumption are non-negotiable.

Future Work:
To further enrich this comparative analysis and broaden its applicability, future research directions include:

- **Real-World Datasets:** Extending the study to diverse real-world datasets with varying complexities and sizes to validate the observed trends under more practical conditions.
- **Advanced Architectures:** Investigating the performance trade-offs for more complex neural network architectures, such as Convolutional Neural Networks (CNNs) for image processing or Recurrent Neural Networks (RNNs) for sequential data, where the overhead of high-level frameworks might become even more pronounced.
- **Microcontroller Benchmarking:** Directly deploying the C implementation onto representative microcontrollers (e.g., ARM Cortex-M series) to provide precise, hardware-specific performance metrics for embedded deployment scenarios.
- **Memory Footprint Analysis:** Quantifying the memory consumption of each implementation, a critical factor for resource-constrained embedded systems.

Quantization and Optimization Techniques: Exploring the impact of model quantization and other optimization techniques (e.g., pruning, knowledge distillation) on the performance of each implementation, particularly for the C version.

VII. REFERENCES

[1]. I. Goodfellow, Y. Bengio, and A. Courville, Deep Learning, MIT Press, 2016.

[2]. M. Abadi, P. Barham, and J. Chen, “TensorFlow: A System for Large-Scale Machine Learning,” in Proc. 12th USENIX Symp. on Operating Systems Design and Implementation (OSDI), pp. 265–283, 2016.

[3]. J. Samarakoon, C. Perera, and S. Jayarathna, “A Survey on Embedded Machine Learning,” ACM Computing Surveys, 2021.

[4]. M. Tambe, A. Chavan, and M. Joshi, “Towards Real-Time TinyML Deployment: A Benchmark Study,” in Proc. IEEE Edge Computing Conf., pp. 95–102, 2022.

[5]. F. Pedregosa, G. Varoquaux, and A. Gramfort, “Scikit-learn: Machine Learning in Python,” Journal of Machine Learning Research, vol. 12, pp. 2825–2830, 2011.

[6]. F. Chollet, “Keras,” https://github.com/fchollet/keras, 2015.

[7]. T. Addison and S. Vallabh, “Controlling Software Project Risks – An Empirical Study of Methods Used by Experienced Project Managers,” in Proc. SAICSIT, pp. 128–140, 2002.

[8]. M. A. Babar and H. Paik, “Using Scrum in Global Software Development: A Systematic Literature Review,” in Proc. 4th IEEE Int. Conf. on Global Software Engineering (ICGSE), pp. 175–184, 2009.

[9]. W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, Numerical Recipes: The Art of Scientific Computing, Cambridge University Press, 2007.

[10]. L. Bottou, “Stochastic Gradient Descent Tricks,” in Neural Networks: Tricks of the Trade, pp. 421–436, 2012.

[11]. K. He, X. Zhang, S. Ren, and J. Sun, “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification,” in Proc. IEEE Int. Conf. on Computer Vision (ICCV), pp. 1026–1034, 2015.

[12]. D. P. Kingma and J. Ba, “Adam: A Method for Stochastic Optimization,” in Proc. Int. Conf. on Learning Representations (ICLR), 2015.