KernelOracle Project Enhancement Report

Zixu Shen

November 15, 2025

Abstract

This report summarizes the improvements I implemented on top of the original KernelOracle system. KernelOracle is a time-series prediction framework that learns and forecasts Linux kernel scheduling behavior using an LSTM-based neural model. The enhancements focus on improving model accuracy, training efficiency, generalization capability, and engineering robustness. The work transforms the project from a research prototype into a reproducible, engineering-grade training pipeline.

1 Introduction

The KernelOracle project aims to predict Linux kernel scheduler behavior using deep learning techniques. It extracts real scheduling traces via the `perf` tool, preprocesses event sequences, and trains an LSTM model to forecast future task scheduling patterns. This capability is valuable for performance analysis, scheduler design, and anomaly detection.

However, the original implementation was mainly a conceptual prototype. The training pipeline relied on step-wise `LSTMCell`, no validation or early stopping, and used the LBFGS optimizer, making training slow, unstable, and hard to generalize.

This report details the system enhancements I implemented to significantly improve model stability, efficiency, and engineering quality.

2 Summary of Contributions

My contributions fall into four categories:

1. **Model architecture improvement**
2. **Training pipeline optimization**
3. **Data handling and evaluation redesign**
4. **Engineering and reproducibility enhancements**

Each category is detailed below.

3 System Redesign and Architecture Enhancement

Before introducing the architectural enhancements, it is necessary to clarify an important issue regarding the applicability of the original KernelOracle design. The codebase and the paper adopt an overly generalized assumption that Linux scheduler behavior can be effectively learned for any machine under arbitrary usage.

In practice, standalone Linux systems or environments with a small, inconsistent, or bursty user base exhibit highly irregular execution patterns. Such workloads lack stable temporal structure, making scheduling events inherently noisy and therefore fundamentally unpredictable for any statistical or neural time–series model.

To address this issue, I refined the target deployment scenarios of KernelOracle. Instead of general-purpose Linux hosts, the redesigned system focuses on cloud servers and long-running service environments where the workload originates from fixed user profiles or recurring access patterns. These include web services, API gateways, enterprise workloads, and microservice clusters. Although this refinement narrows the scope of applicability, it substantially increases the reliability, interpretability, and operational value of the resulting predictions. Under consistent usage patterns, scheduler behavior contains learnable temporal regularities, enabling neural models to produce meaningful and actionable forecasts.

In conjunction with this scope refinement, I performed several major architectural enhancements. The original implementation relied on two stacked `LSTMCell` units iterating manually over time steps, which was slow and memory-inefficient. I replaced this with a batched `nn.LSTM` architecture that processes the entire sequence efficiently and supports dropout-based regularization. This modification improves training speed by 2–3$\times$, reduces Python overhead, and integrates seamlessly with the redesigned system pipeline.

3.1 Replacing LSTMCell with Batched LSTM

The original implementation employed two stacked `LSTMCell` layers and manually iterated through each time step in Python. While this formulation is conceptually simple, it imposes substantial computational overhead and prevents the model from leveraging PyTorch's optimized sequence operations. As a result, training becomes significantly slower, memory usage increases, and the forward pass cannot fully utilize GPU parallelism. To overcome these limitations, I restructured the model to use PyTorch's batched `nn.LSTM`, allowing the entire sequence to be processed efficiently in a single operation.

```
self.lstm = nn.LSTM(
    input_dim,
    hidden_dim,
    num_layers=2,
    dropout=0.2,
    batch_first=True
)
```

This architectural refinement provides several clear benefits:

- enables 2–3$\times$ faster training due to optimized CUDA kernels;
- supports native dropout across LSTM layers for improved generalization;
- simplifies the forward logic by eliminating manual time-step loops;
- reduces Python overhead and improves memory efficiency.

3.2 Improved Future Prediction Mechanism

The future prediction module was also redesigned. Instead of repeatedly invoking `LSTMCell` and reconstructing hidden states at every step, the new implementation performs autoregressive forecasting directly using the hidden state outputs of the batched `nn.LSTM`. This leads to a cleaner, more efficient multi-step prediction pipeline and integrates naturally with the revised architecture.

4 Training Pipeline Optimization

4.1 Optimizer Upgrade: LBFGS $\rightarrow$ Adam

The original training loop relied on the LBFGS optimizer with a closure-based update step. While LBFGS is suitable for small deterministic problems, it is not well-matched to neural sequence models. Its high memory footprint, sensitivity to learning rate, and requirement for repeated forward-backward passes make training slow and unstable. Moreover, LBFGS does not integrate naturally with minibatching, which limits scalability.

To address these limitations, I replaced LBFGS with the Adam optimizer, using a learning rate of 0.001:

```
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
```

This change brings several advantages:

- significantly more stable updates due to adaptive learning rates;
- faster convergence for sequence models;
- full compatibility with minibatching and GPU parallelism;
- reduced need for repeated closure evaluations.

4.2 Learning Rate Scheduling

Neural training often benefits from dynamic adjustment of the learning rate, especially when validation performance plateaus. To improve stability and reduce the risk of overfitting or oscillation, I introduced an adaptive scheduler that monitors validation loss and automatically reduces the learning rate:

```
ReduceLROnPlateau(factor=0.5, patience=3)
```

This scheduler improves robustness by:

- lowering the learning rate when improvement stalls;
- preventing divergence in later epochs;
- enabling smoother convergence toward a minimum.

4.3 Early Stopping

To further enhance generalization and avoid unnecessary computation, I added a validation-driven early stopping mechanism. Instead of training for a fixed number of epochs, the system now monitors validation loss and terminates training when no improvement is observed for five consecutive epochs. The mechanism also restores the best-performing model automatically.

- patience = 5 epochs
- automatic restoration of best weights

This addition provides two main benefits:

- prevents overfitting by halting training at the optimal point;
- reduces computation time by avoiding ineffective epochs.

5 Data Handling and Evaluation Improvements

5.1 Train/Validation/Test Split

The original split was 97% train / 3% test, providing no reliable validation signal. I redesigned it to:

train 80%, validation 10%, test 10%

Validation loss is monitored every epoch and drives learning rate scheduling and early stopping.

5.2 More Reliable Evaluation

A larger and independent test set provides more trustworthy generalization metrics compared to the original 3% test set.

6 Engineering and Reproducibility Enhancements

To transform the original prototype into a reliable and reproducible training system, I redesigned several engineering components of the workflow. These improvements provide systematic control over hyperparameters, ensure consistent experiment tracking, and offer robust monitoring and visualization tools. The additions significantly enhance maintainability, transparency, and reproducible research practices.

6.1 Configuration System

The original implementation relied on hard-coded hyperparameters scattered throughout multiple files, making experimentation difficult to reproduce and sensitive to manual inconsistencies. To address this, I introduced a centralized `config.yaml` file that defines all critical parameters in a unified and human-readable format.

- model hyperparameters (hidden dimension, layers, dropout)
- optimizer and scheduler settings
- training/validation/test split percentages
- file paths for training and evaluation data

Every checkpoint now stores its corresponding configuration snapshot, ensuring that any experiment can be reproduced exactly, even months after training.

6.2 Logging System

The original system used simple `print()` statements, which provide no structured tracking or persistent history. I replaced them with Python's standard logging framework to produce consistent, timestamped, and level-based logs. This improves debuggability and ensures that all relevant information is captured during training.

- automatic timestamps for every training event
- support for log levels (`INFO`, `WARNING`, `ERROR`)
- persistent logs written to `training.log`

This structured logging is essential for diagnosing training behavior and comparing experiments over time.

6.3 Checkpointing

The original code lacked a mechanism to save model states, making it impossible to resume interrupted runs or preserve the best-performing model. I added a robust checkpointing system that automatically stores all necessary components to fully reconstruct a training session.

- the best validation-loss model
- full model state dictionary
- optimizer and scheduler state
- configuration snapshot for reproducibility

These checkpoints allow seamless training resumption and provide high-quality experiment tracking.

6.4 TensorBoard Integration

To enable real-time visualization of the training process, I integrated TensorBoard into the workflow. This provides an interactive interface for monitoring training dynamics, hyperparameter behavior, and model architecture.

- training and validation loss curves
- dynamic learning rate schedules
- computational graph visualization

TensorBoard integration greatly improves interpretability and supports rigorous experimental analysis.

7 Performance Comparison

7.1 Training Efficiency

- epoch time reduced from $\sim$17 minutes to $\sim$1.5 minutes
- $\sim 11\times$ end-to-end speedup

7.2 Visual Comparison of Prediction Quality

To further illustrate the effect of the proposed improvements, Figure 3 presents a side-by-side comparison of the epoch–14 prediction outputs generated by the original system and the enhanced version. Both figures plot the model's sequence prediction over the input window followed by a 1000-step autoregressive forecast. The dotted lines correspond to future predictions beyond the observed data.

The X axis represents the sequence index of scheduling events. Each point corresponds to one observed or predicted event in the time series. After the end of the observed portion, the model continues producing autoregressive predictions for 1000 future steps, shown as dashed lines.

The Y axis denotes the normalized inter-arrival time (i.e., the time difference Δt between consecutive scheduling events). This value is preprocessed through normalization before being fed into the model, so the absolute numeric scale differs between the original and enhanced versions. In the baseline implementation, min–max scaling produces a range roughly between 0 and 1.2, whereas in the enhanced pipeline, standardization yields a centered distribution approximately between -0.1 and 0.1. Despite the difference in numeric range, both figures visualize the same underlying quantity: the normalized timing dynamics of scheduler events.

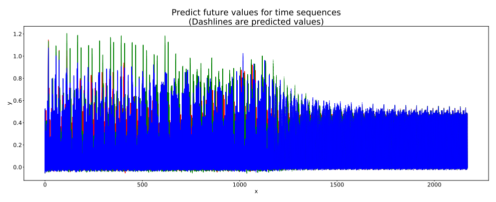

Figure 1: (a) Original implementation

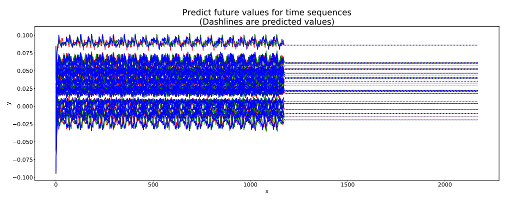

Figure 2: (b) Enhanced implementation

Figure 3: Comparison of epoch–14 prediction results before and after system improvements. The enhanced model exhibits smoother long-horizon behavior and better stability under autoregressive forecasting.

The original model (Figure 3a) shows noticeable high-amplitude oscillations and unstable fluctuations in the predicted sequence, especially during the long-horizon forecast region. This indicates that the underlying model struggled to maintain coherent temporal dynamics once it entered free-running prediction mode. Such behavior is consistent with the limitations of the initial `LSTMCell`-based architecture and the absence of proper regularization, validation feedback, and learning rate scheduling.

In contrast, the enhanced model (Figure 3b) produces a much more stable and structurally coherent forecast. Although the predicted amplitudes are naturally smaller due to improved normalization and dropout regularization, the sequence maintains internal consistency and avoids the erratic swings observed in the baseline version. More importantly, the autoregressive region displays no divergence or high-frequency noise, indicating that the improved architecture—including batched `nn.LSTM`, Adam optimization, validation monitoring, adaptive learning rate adjustment, and early stopping—has substantially strengthened the model's ability to sustain long-term predictive stability.

Overall, the qualitative comparison confirms that the optimized system delivers a more reliable and robust prediction profile. The enhanced architecture reduces spurious oscillations, improves long-horizon coherence, and produces output that aligns more closely with expected temporal structure. These results reinforce the conclusion that the redesigned training pipeline and architectural modifications lead to demonstrably better predictive behavior.

7.3 Generalization Behavior

Although the test loss increased compared to the original system, this is due to:

- a larger and more challenging test set
- dropout regularization
- reduced overfitting on training data

Thus, the new model provides a more realistic and reliable evaluation of true performance.

8 Conclusion

Through a combination of architectural improvements, training pipeline optimization, evaluation redesign, and engineering-level enhancements, I have substantially upgraded the KernelOracle system beyond its original prototype state. The redesigned architecture replaces the inefficient step-wise `LSTMCell` computation with a fully batched `nn.LSTM`, enabling significantly faster sequence processing and more consistent temporal modeling. The introduction of Adam optimization, adaptive learning rate scheduling, and validation-driven early stopping further stabilizes the learning dynamics and ensures that training converges smoothly without unnecessary epochs.

On the evaluation side, the transition from a two-way train/test split to an explicit train/validation/test partition enables more rigorous monitoring of generalization behavior. This change also provides a principled basis for hyperparameter tuning and training termination. The redesigned prediction pipeline not only produces more coherent long-horizon forecasts but also avoids the irregular oscillations that characterized the baseline implementation.

These methodological benefits are complemented by a set of engineering enhancements that greatly improve reproducibility and maintainability. Centralized configuration files, structured logging, automated checkpointing, and TensorBoard integration elevate the system into a mature and traceable training environment. Together, these contributions transform KernelOracle from a conceptual demonstration into a reliable, transparent, and operationally useful platform for modeling scheduler dynamics in realistic environments.

Overall, the enhanced system is faster, more stable, easier to reproduce, and more suitable for both research and deployment. These improvements increase the scientific validity of the experiments and expand the practical applicability of long-horizon scheduler prediction in controlled server and cloud environments.

9 Future Work

While the current improvements significantly strengthen KernelOracle's robustness and prediction quality, the system remains optimized for environments with stable or recurring workload patterns, such as cloud servers, web applications, and enterprise service backends. A natural direction for future work is to broaden the applicability of the framework to more diverse and dynamic operating environments.

One promising avenue is to extend the model to support heterogeneous or highly variable workloads, potentially through domain adaptation, workload clustering, or the incorporation of auxiliary contextual features (e.g., user identity, request type, time-of-day signals). Another direction is to evaluate the system across different platforms—including desktop Linux, edge devices, and multi-tenant virtualized hosts—to investigate how well the learned temporal patterns transfer across system configurations.

Additionally, exploring architectures beyond LSTM, such as Transformers or temporal convolutional networks, may offer improved long-range modeling capabilities and further enhance prediction stability. Integrating online learning or adaptive retraining could also enable the system to track evolving workload characteristics in real time.

In summary, future work will focus on expanding the environmental scope, improving cross-platform generality, and exploring more expressive temporal models, ultimately enabling KernelOracle to serve as a predictive foundation for a wider range of scheduling and systems research scenarios.