LSTM: A Memorable Model

Atharva Pandhare
Department of Electrical and Computer Engineering
Rutgers University, Piscataway, NJ, 08854, USA
atharva.pandhare@rutgers.edu

***Abstract*—This paper presents the design, implementation, and performance analysis of a Long Short-Term Memory (LSTM) network optimized for vehicle trajectory prediction, a critical component in advanced driver assistance systems and autonomous driving applications. The custom LSTM architecture is built from first principles to provide deeper insights into recurrent neural network mechanics and their application to sequential time-series data. Our model is trained and evaluated on a comprehensive dataset comprising 9,400 sequential samples, each containing 12 parameters tracked across 67 time steps, with the objective of predicting future vehicle coordinates based on historical movement patterns. The implementation leverages PyTorch for efficient computation while employing Root Mean Square Error (RMSE) as the primary loss function. Through systematic hyperparameter optimization, particularly focusing on hidden layer dimensionality and learning rate scheduling, we achieve a final test RMSE of 12.55, demonstrating the effectiveness of our approach. This work not only offers a practical solution for trajectory forecasting but also provides valuable insights into the internal mechanisms of LSTM networks, including detailed gradient derivations for backpropagation through time.**

I. Introduction

In machine learning, sequential data represents information where the order of events or observations is crucial. This kind of data shows up everywhere. Think about the words in your sentence – the order defines the meaning. Alternatively, consider a car's movement over time; its current position depends heavily on where it was and how it moved just moments before. This "story-like" data is sequential, where past events influence present or future ones.

Machine learning gives us powerful ways to work with these sequences. Computers can learn to find patterns, make predictions, or understand the content of this ordered information. A typical daily example is text completion or next-word prediction, like when a phone suggests the next word as the user is typing. Machine learning models are trained on vast amounts of text (which is sequential data) to predict what you are likely to type next. Another good example is forecasting renewable energy generation in power systems; machine learning models analyze past sequences of energy production (like from solar or wind) to predict how much energy will be generated shortly. This ability to predict the "next step" or future values based on past sequential inputs is a core strength of machine learning. Similar principles allow these models to predict future vehicle coordinates based on past trajectory data, which is the focus of this project.

However, dealing with sequential data has its challenges. A big one is figuring out how events from way back in the sequence can affect things happening much later – these are called "long-range dependencies." It is like trying to understand the end of a long movie based on something that happened right at the beginning. Deep learning frameworks applied to tasks like entity summarization in knowledge graphs also leverage recurrent networks for processing such sequential text data.

Special kinds of Neural Networks were developed to handle these challenges. Recurrent Neural Networks (RNNs) were an early type designed with a sort of "memory" to keep track of past information. However, standard RNNs often struggle with those tricky long-range dependencies. This is where Long Short-Term Memory networks, or LSTMs, come in. LSTMs are an advanced type of RNN with a more sophisticated memory system, using special "gates" that allow them to decide what information is essential to remember and what can be forgotten over long periods. This makes them particularly good at tasks like predicting text, forecasting energy, or, as in our project, forecasting a car's future path based on its history of movements (like its coordinates, speed, and acceleration).

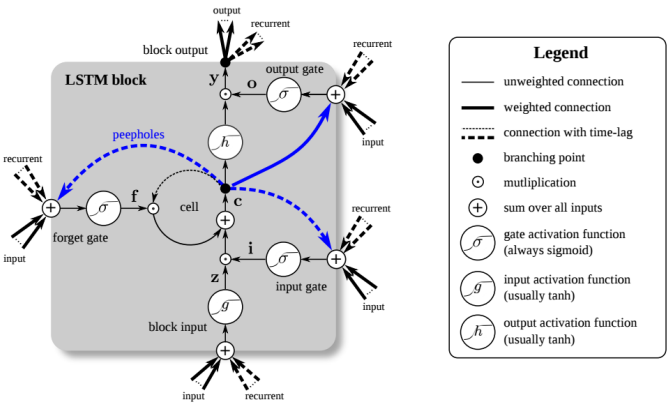


Fig. 1. LSTM Cell Structure

This paper focuses on predicting a vehicle's future coordinates by building an LSTM network from scratch. "From scratch" means we will not use the ready-made LSTM tools often found in programming libraries; instead, we will construct the core parts of the LSTM ourselves using basic matrix operations and activation functions. The main goal is to make accurate predictions and gain a deep, practical understanding of how LSTMs work on the inside. The data for this project comes from thousands of files with time series

data for a vehicle. Each File comprises 12 distinct parameters tracked for the car: spatial coordinates (x, y) providing precise vehicle positioning, velocity and acceleration vectors capturing motion dynamics, distance from the centerline measuring lane positioning, binary indicators signaling the presence of other vehicles to the left, right, and front of the subject vehicle, and distance measurements to these surrounding vehicles. This comprehensive feature set enables the LSTM model to learn complex interactions between a vehicle's movement patterns and its relationship to both road infrastructure and neighboring traffic participants, facilitating accurate trajectory prediction in diverse driving scenarios.

II. Gate Structure

The following equations define the Long Short-Term Memory (LSTM) network:

$$
\begin{aligned}
i_t &= \sigma(W_{x_i} x_t + b_{x_i} + W_{h_i} h_{t-1} + b_{h_i}) \\
f_t &= \sigma(W_{x_f} x_t + b_{x_f} + W_{h_f} h_{t-1} + b_{h_f}) \\
g_t &= \tanh(W_{x_g} x_t + b_{x_g} + W_{h_g} h_{t-1} + b_{h_g}) \\
o_t &= \sigma(W_{x_o} x_t + b_{x_o} + W_{h_o} h_{t-1} + b_{h_o}) \\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\
h_t &= o_t \odot \tanh(c_t)
\end{aligned}
$$

A. Training LSTMs with Backpropagation Through Time

Training a neural network, including LSTMs, involves adjusting its weights and biases to minimize a loss function that quantifies the difference between the network's predictions and target values. Backpropagation is a fundamental algorithm used for this purpose, efficiently calculating the gradient of the loss function with respect to each network parameter. An optimization algorithm, like gradient descent, then uses these gradients to update the parameters.

For Recurrent Neural Networks (RNNs), including LSTMs, a specialized version of backpropagation called Backpropagation Through Time (BPTT) is employed. RNNs process sequential data, meaning their computations at a given time step depend on previous time steps. BPTT addresses this by "unrolling" the recurrent network through time, effectively creating a deep feedforward network where each layer corresponds to a time step. The error is then propagated backward through this unrolled structure, from the final time step to the first, accumulating gradients for the shared weights across all time steps.

The primary objective of BPTT in LSTMs is to compute the derivatives of the overall loss function with respect to all the learnable parameters (weights W and biases b for each gate and cell computation). While standard BPTT can suffer from issues like vanishing or exploding gradients, especially over long sequences, LSTMs are specifically designed with gating mechanisms to mitigate these problems, allowing them to learn long-range dependencies more effectively. In practice, to manage computational and memory resources for very long sequences, a variation called Truncated BPTT is often used. This approach limits the backpropagation to a fixed number of previous time steps, rather than the entire sequence.

B. Deriving Gradients

Training an LSTM network using BPTT involves adjusting its weights based on the gradients of a loss function with respect to these weights. This requires calculating the partial derivatives for each component of the LSTM cell using the chain rule. The core components whose gradients we need to derive are the gates, the cell state, and the hidden state updates.

An LSTM cell utilizes several gates to control the flow of information, enabling it to learn long-term dependencies. These gates are:

- **Input Gate (i_t):** This gate decides which new information from the current input (x_t) and previous hidden state (h_{t-1}) should be stored in the cell state (c_t). It employs a sigmoid function (σ) to output values between 0 (block) and 1 (allow).
- **Forget Gate (f_t):** This gate determines what information from the previous cell state (c_{t-1}) should be discarded or forgotten. Like the input gate, it uses a sigmoid function.
- **Candidate Gate (g_t):** Often referred to as the cell input activation, this component generates a vector of new candidate values that could be added to the cell state. It typically uses a hyperbolic tangent function ($\tanh$) to scale values between -1 and 1.
- **Output Gate (o_t):** This gate controls which part of the current cell state (c_t) is outputted as the new hidden state (h_t). It also uses a sigmoid function, and the cell state information is typically passed through a $\tanh$ function before being modulated by this gate.

Understanding the role of each gate is crucial for interpreting their respective gradient calculations. We will now proceed to derive the partial gradients for each of these components with respect to their inputs and parameters.

1) Input Gate (i_t):

$$i_t = \sigma(W_{x_i} x_t + b_{x_i} + W_{h_i} h_{t-1} + b_{h_i})$$

Partials:

- $i_t = \sigma(p)$

$$\nabla_p i_t = (1 - \sigma(p))\sigma(p)$$

- $p = q + v$

$$
\begin{aligned}
\nabla_q p &= 1 \\
\nabla_v p &= 1
\end{aligned}
$$

- $q = W_{x_i} x_t + b_{x_i}$

$$
\begin{aligned}
\nabla_{W_{x_i}} q &= x_t \\
\nabla_{x_t} q &= W_{x_i} \\
\nabla_{b_{x_i}} q &= 1
\end{aligned}
$$

- $v = W_{h_i} h_{t-1} + b_{h_i}$

$$
\begin{aligned}
\nabla_{W_{h_i}} v &= h_{t-1} \\
\nabla_{h_{t-1}} v &= W_{h_i} \\
\nabla_{b_{h_i}} v &= 1
\end{aligned}
$$

Final Gradients:

$$\begin{aligned}
\nabla_{W_{x_i}} i_t &= \nabla_p i_t \nabla_q p \nabla_{W_{x_i}} q = \nabla_p i_t \cdot x_t \\
\nabla_{x_t} i_t &= \nabla_p i_t \nabla_q p \nabla_{x_t} q = \nabla_p i_t \cdot W_{x_i} \\
\nabla_{b_{x_i}} i_t &= \nabla_p i_t \nabla_q p \nabla_{b_{x_i}} q = \nabla_p i_t \\
\nabla_{W_{h_i}} i_t &= \nabla_p i_t \nabla_v p \nabla_{W_{h_i}} v = \nabla_p i_t \cdot h_{t-1} \\
\nabla_{h_{t-1}} i_t &= \nabla_p i_t \nabla_v p \nabla_{h_{t-1}} v = \nabla_p i_t \cdot W_{h_i} \\
\nabla_{b_{h_i}} i_t &= \nabla_p i_t \nabla_v p \nabla_{b_{h_i}} v = \nabla_p i_t
\end{aligned}$$

2) Forget Gate (f_t):

$$f_t = \sigma(W_{x_f} x_t + b_{x_f} + W_{h_f} h_{t-1} + b_{h_f})$$

Partials::

- $f_t = \sigma(p)$

$$\nabla_p f_t = (1 - \sigma(p))\sigma(p)$$

- $p = q + v$

$$\begin{aligned}
\nabla_q p &= 1 \\
\nabla_v p &= 1
\end{aligned}$$

- $q = W_{x_f} x_t + b_{x_f}$

$$\begin{aligned}
\nabla_{W_{x_f}} q &= x_t \\
\nabla_{x_t} q &= W_{x_f} \\
\nabla_{b_{x_f}} q &= 1
\end{aligned}$$

- $v = W_{h_f} h_{t-1} + b_{h_f}$

$$\begin{aligned}
\nabla_{W_{h_f}} v &= h_{t-1} \\
\nabla_{h_{t-1}} v &= W_{h_f} \\
\nabla_{b_{h_f}} v &= 1
\end{aligned}$$

Final Gradients:

$$\begin{aligned}
\nabla_{W_{x_f}} f_t &= \nabla_p f_t \nabla_q p \nabla_{W_{x_f}} q = \nabla_p f_t \cdot x_t \\
\nabla_{x_t} f_t &= \nabla_p f_t \nabla_q p \nabla_{x_t} q = \nabla_p f_t \cdot W_{x_f} \\
\nabla_{b_{x_f}} f_t &= \nabla_p f_t \nabla_q p \nabla_{b_{x_f}} q = \nabla_p f_t \\
\nabla_{W_{h_f}} f_t &= \nabla_p f_t \nabla_v p \nabla_{W_{h_f}} v = \nabla_p f_t \cdot h_{t-1} \\
\nabla_{h_{t-1}} f_t &= \nabla_p f_t \nabla_v p \nabla_{h_{t-1}} v = \nabla_p f_t \cdot W_{h_f} \\
\nabla_{b_{h_f}} f_t &= \nabla_p f_t \nabla_v p \nabla_{b_{h_f}} v = \nabla_p f_t
\end{aligned}$$

3) Output Gate (o_t):

$$o_t = \sigma(W_{x_o} x_t + b_{x_o} + W_{h_o} h_{t-1} + b_{h_o})$$

Partials:

- $o_t = \sigma(p)$

$$\nabla_p o_t = (1 - \sigma(p))\sigma(p)$$

- $p = q + v$

$$\begin{aligned}
\nabla_q p &= 1 \\
\nabla_v p &= 1
\end{aligned}$$

- $q = W_{x_o} x_t + b_{x_o}$

$$\begin{aligned}
\nabla_{W_{x_o}} q &= x_t \\
\nabla_{x_t} q &= W_{x_o} \\
\nabla_{b_{x_o}} q &= 1
\end{aligned}$$

- $v = W_{h_o} h_{t-1} + b_{h_o}$

$$\begin{aligned}
\nabla_{W_{h_o}} v &= h_{t-1} \\
\nabla_{h_{t-1}} v &= W_{h_o} \\
\nabla_{b_{h_o}} v &= 1
\end{aligned}$$

Final Gradients:

$$\begin{aligned}
\nabla_{W_{x_o}} o_t &= \nabla_p o_t \nabla_q p \nabla_{W_{x_o}} q = \nabla_p o_t \cdot x_t \\
\nabla_{x_t} o_t &= \nabla_p o_t \nabla_q p \nabla_{x_t} q = \nabla_p o_t \cdot W_{x_o} \\
\nabla_{b_{x_o}} o_t &= \nabla_p o_t \nabla_q p \nabla_{b_{x_o}} q = \nabla_p o_t \\
\nabla_{W_{h_o}} o_t &= \nabla_p o_t \nabla_v p \nabla_{W_{h_o}} v = \nabla_p o_t \cdot h_{t-1} \\
\nabla_{h_{t-1}} o_t &= \nabla_p o_t \nabla_v p \nabla_{h_{t-1}} v = \nabla_p o_t \cdot W_{h_o} \\
\nabla_{b_{h_o}} o_t &= \nabla_p o_t \nabla_v p \nabla_{b_{h_o}} v = \nabla_p o_t
\end{aligned}$$

4) Candidate Gate (g_t):

$$g_t = \tanh(W_{x_g} x_t + b_{x_g} + W_{h_g} h_{t-1} + b_{h_g})$$

Partials:

- $g_t = \tanh(p)$

$$\nabla_p g_t = 1 - \tanh^2(p)$$

- $p = q + v$

$$\begin{aligned}
\nabla_q p &= 1 \\
\nabla_v p &= 1
\end{aligned}$$

- $q = W_{x_g} x_t + b_{x_g}$

$$\begin{aligned}
\nabla_{W_{x_g}} q &= x_t \\
\nabla_{x_t} q &= W_{x_g} \\
\nabla_{b_{x_g}} q &= 1
\end{aligned}$$

- $v = W_{h_g} h_{t-1} + b_{h_g}$

$$\begin{aligned}
\nabla_{W_{h_g}} v &= h_{t-1} \\
\nabla_{h_{t-1}} v &= W_{h_g} \\
\nabla_{b_{h_g}} v &= 1
\end{aligned}$$

Final Gradients:

$$\begin{aligned}
\nabla_{W_{x_g}} g_t &= \nabla_p g_t \nabla_q p \nabla_{W_{x_g}} q = \nabla_p g_t \cdot x_t \\
\nabla_{x_t} g_t &= \nabla_p g_t \nabla_q p \nabla_{x_t} q = \nabla_p g_t \cdot W_{x_g} \\
\nabla_{b_{x_g}} g_t &= \nabla_p g_t \nabla_q p \nabla_{b_{x_g}} q = \nabla_p g_t \\
\nabla_{W_{h_g}} g_t &= \nabla_p g_t \nabla_v p \nabla_{W_{h_g}} v = \nabla_p g_t \cdot h_{t-1} \\
\nabla_{h_{t-1}} g_t &= \nabla_p g_t \nabla_v p \nabla_{h_{t-1}} v = \nabla_p g_t \cdot W_{h_g} \\
\nabla_{b_{h_g}} g_t &= \nabla_p g_t \nabla_v p \nabla_{b_{h_g}} v = \nabla_p g_t
\end{aligned}$$

5) Cell State Update (c_t):

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t$$

Partials:

- $c_t = q + p$

$$\nabla_q c_t = 1$$
$$\nabla_p c_t = 1$$

- $q = f_t \odot c_{t-1}$

$$\nabla_{f_t} q = c_{t-1}$$
$$\nabla_{c_{t-1}} q = f_t$$

- $p = i_t \odot g_t$

$$\nabla_{i_t} p = g_t$$
$$\nabla_{g_t} p = i_t$$

Final Gradients:

$$\nabla_{f_t} c_t = \nabla_q c_t \nabla_{f_t} q = c_{t-1}$$
$$\nabla_{c_{t-1}} c_t = \nabla_q c_t \nabla_{c_{t-1}} q = f_t$$
$$\nabla_{i_t} c_t = \nabla_p c_t \nabla_{i_t} p = g_t$$
$$\nabla_{g_t} c_t = \nabla_p c_t \nabla_{g_t} p = i_t$$

6) Hidden State Update (h_t):

$$h_t = o_t \odot \tanh(c_t)$$

Partials:

- $h_t = o_t \odot q$

$$\nabla_{o_t} h_t = q$$
$$\nabla_q h_t = o_t$$

- $q = \tanh(c_t)$

$$\nabla_{c_t} q = 1 - \tanh^2(c_t)$$

Final Gradients:

$$\nabla_{o_t} h_t = \tanh(c_t)$$
$$\nabla_{c_t} h_t = \nabla_q h_t \nabla_{c_t} q = o_t \cdot (1 - \tanh^2(c_t))$$

III. Implementation

Implementing the LSTM model incorporates several modern deep learning techniques and tools to optimize performance and facilitate efficient training. This section explores the key implementation aspects contributing to the model's effectiveness in trajectory prediction.

A. Dataset Distribution

Dataset splitting is critical to machine learning model development, directly impacting model evaluation reliability and performance. The implementation employs a 70/15/15 distribution ratio for training, validation, and testing sets. This balanced approach provides sufficient data for the LSTM to learn complex trajectory patterns while reserving adequate samples for validation during training and unbiased final evaluation. The splitting process uses stratified sampling to maintain consistent data characteristics across all partitions, preserving each subset's original trajectory pattern distribution. For time-series data like vehicle trajectories, this careful partitioning ensures the model encounters representative driving scenarios during both training and evaluation phases, increasing confidence that performance metrics accurately reflect real-world capability rather than dataset artifacts.

B. Xavier Initialization

Xavier initialization is a weight initialization strategy designed to maintain consistent variance of activations and gradients across neural network layers. This is particularly important for LSTMs, where the complex gating mechanisms are susceptible to vanishing or exploding gradients. By initializing weights from a distribution with variance scaled based on the number of input and output connections, Xavier initialization enables stable signal propagation through the LSTM network from the start of training. This helps the model converge faster and perform better, especially when dealing with the temporal dependencies in sequential data like car trajectories.

C. Dropout Regularization

Dropout regularization prevents overfitting by randomly deactivating a portion of neurons during each training iteration. In LSTMs, dropout must be applied carefully due to the recurrent nature of the network. The implementation typically applies dropout to non-recurrent connections while preserving the recurrent state information. This selective application allows the model to maintain its temporal memory capabilities while benefiting from regularization. For trajectory prediction tasks, dropout helps the model generalize from training data patterns rather than merely memorizing specific sequences, improving performance on unseen vehicle movement patterns.

D. Adam Optimization

Adam (Adaptive Moment Estimation) optimization combines the benefits of AdaGrad and RMSProp by adapting learning rates individually for each parameter based on both first-order moments (mean) and second-order moments (uncentered variance) of gradients. This adaptivity makes Adam particularly suited for training LSTMs, which contain parameters serving diverse roles in the network architecture. For car trajectory prediction, Adam helps navigate the complex loss landscape efficiently, adapting to frequently and infrequently updated parameters in the model, such as those in the various gates controlling information flow through time steps.

E. Learning Rate Scheduling with Cosine Annealing

Cosine annealing schedules the learning rate to follow a cosine function, gradually decreasing from an initial value to a minimum before potentially restarting in a cyclical pattern. This technique helps optimization escape local minima and explore the parameter space more effectively. For LSTM models predicting car trajectories, cosine annealing provides larger parameter updates early in training and smaller, more refined updates as training progresses. The implementation shows that this approach significantly improved model convergence, allowing the network to find optimal parameters for capturing short-term and long-term patterns in vehicle movement data.

F. Automatic Differentiation with PyTorch

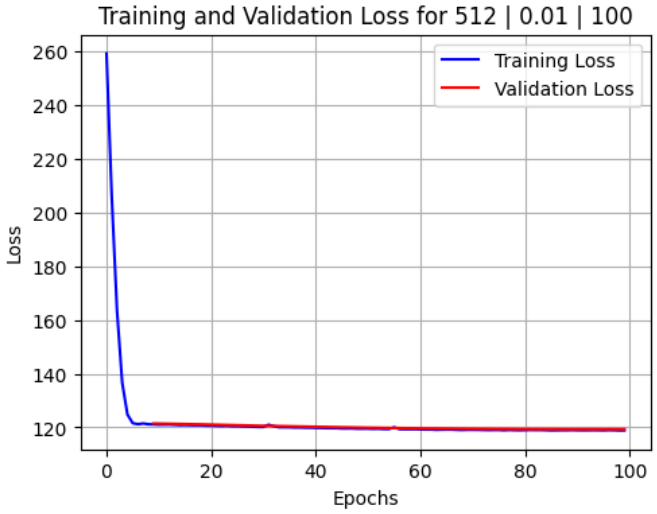
Automatic differentiation in PyTorch enables efficient computation of gradients without manually defining derivative formulas. This capability is crucial for LSTMs, which involve multiple interconnected gates and states propagated through time. PyTorch's dynamic computational graph handles back-propagation through time (BPTT) automatically, calculating gradients flowing backward through multiple time steps. This dramatically simplifies the implementation of complex recurrent operations and allows researchers to focus on model architecture rather than the intricacies of gradient computation. Automatic differentiation enables efficient training on thousands of sequential data samples with varying vehicle movement patterns for trajectory prediction.

IV. Hyperparameter Adjustments

Hyperparameter tuning is a crucial step in developing effective machine learning models. It involves setting the optimal configuration of parameters defined before the learning process begins. I experimented with a few configurations to identify the best settings for the LSTM model's performance on trajectory prediction tasks.

A. First Run

The initial run utilized a hidden layer size of 512, a dropout rate of 0.2, and a learning rate of 0.01. The resulting training and validation loss curves are presented in Figure 2.

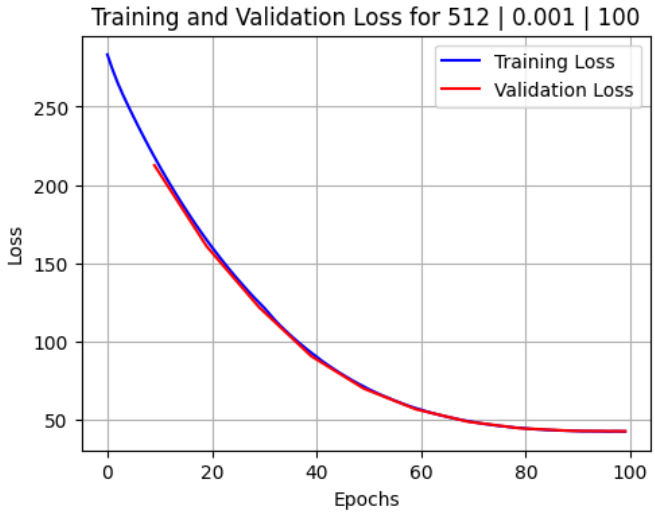


Training and Validation Loss
Fig. 2. hidden size: 512, dropout: 0.2, learning rate: 0.01, epochs: 100
run time: 5.5s Test Loss: 116.0590

As observed in Figure 2, the loss curve exhibits a steep initial descent followed by a plateau. This behavior often indicates that the learning rate is too high, causing the optimization process to overshoot the optimal point or oscillate around it, hindering further convergence to a lower loss value. The test loss for this configuration was 116.0590.

B. Second Run

Based on the observations from the first run, the learning rate was reduced by an order of magnitude to 0.001, keeping the hidden size at 512 and the dropout rate at 0.2. The objective was to achieve a more stable and consistent reduction in loss.



Training and Validation Loss
Fig. 3. hidden size: 512, dropout: 0.2, learning rate: 0.001, epochs: 100
run time: 5.5m
Test Loss: 44.7887

The loss curves for the second run, shown in Figure 3, demonstrate a smoother and more gradual descent. This suggests that the lower learning rate allowed for more stable convergence. While the convergence is smoother, the test loss of 44.7887, though a significant improvement, suggests there is still room for enhancing model capacity or further refining parameters. The curve seems cut short, meaning there was likely no convergence.

C. Third Run

Based on Run 2's observations, the number of epochs was increased to 300 while maintaining the same hyperparameters (hidden size 512, dropout 0.2, learning rate 0.001). The objective was to achieve a more complete convergence pattern and determine whether extended training would yield further improvements.

The results from the third run, as shown in Figure 4, reveal a dramatic improvement in model performance. The extended training period allowed the model to achieve a much lower test loss of 12.5533, which represents a significant reduction from the previous run's 44.7887. The loss curves exhibit a more complete convergence pattern, with both training and validation losses steadily decreasing before stabilizing. This confirms that the model from Run 2 had not fully converged and benefited substantially from additional training iterations. The relatively small gap between training and validation loss throughout training indicates good generalization capabilities despite the extended training period.

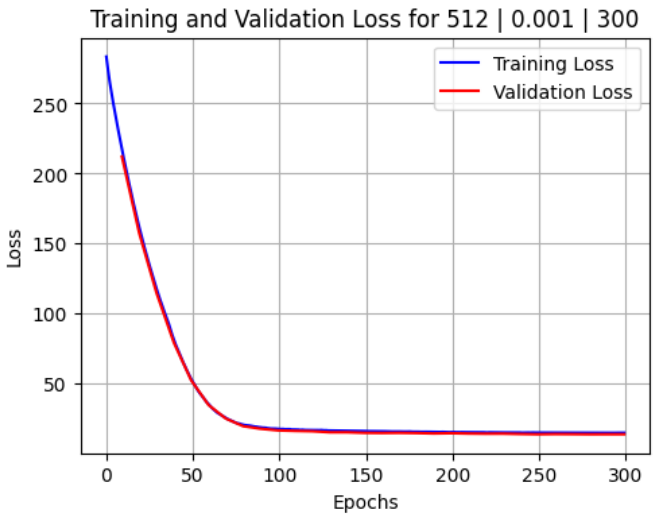


Fig. 4. Training and Validation Loss
hidden size: 512, dropout: 0.2, learning rate: 0.001, epochs: 300
run time: 16.5m
Test Loss: 12.5533

D. Fourth Run

For the fourth experiment, the model capacity was increased by doubling the hidden layer size to 1024, while maintaining the dropout rate at 0.2 and learning rate at 0.001. The number of epochs was set back to 100 to compare with Run 2 at the same training duration but with increased model complexity. The fourth run achieved a test loss of 13.4730, as shown in

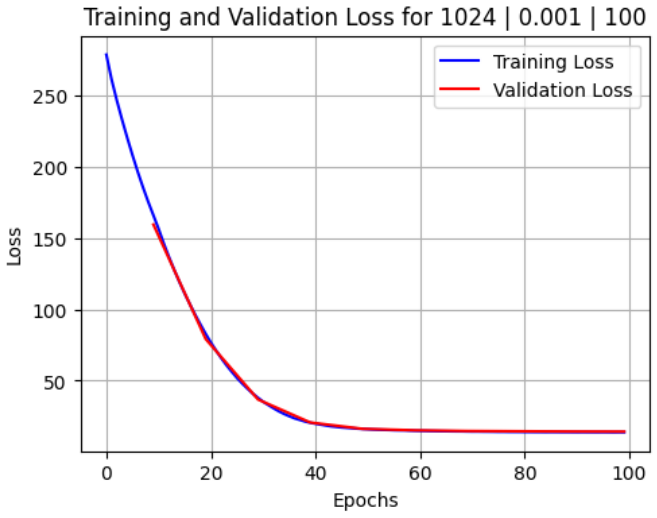


Fig. 5. Training and Validation Loss
hidden size: 1024, dropout: 0.2, learning rate: 0.001, epochs: 100
run time: 6.5m
Test Loss: 13.4730

Figure 5, significantly better than the second run with the same number of epochs but a smaller hidden size. However, it is slightly higher than the third run's loss of 12.5533. This suggests that while increasing model capacity improves performance, the number of training epochs substantially impacts the final model quality for this particular task. The loss curve indicates that the model was still actively learning when training concluded, suggesting that with additional epochs, this larger model might potentially outperform the smaller model with extended training.

E. Analysis of Test Results

The comparison between Run 3 and Run 4 reveals an interesting trade-off between model complexity, training duration, and computational efficiency. Despite utilizing different approaches, both runs achieved remarkably similar test losses (12.5533 for Run 3 versus 13.4730 for Run 4).

Run 3 employed a smaller model (hidden size 512) but trained for three times as many epochs (300), while Run 4 used a larger model (hidden size 1024) but trained for only 100 epochs. The minimal performance difference between these configurations (approximately 7.3% higher loss for Run 4) suggests multiple viable pathways to achieving strong results on this task.

From a computational efficiency perspective, the larger model in Run 4 requires significantly more resources per epoch. Doubling the hidden layer size from 512 to 1024 quadruples the matrix multiplication operations in the LSTM cells, which typically dominate the overall computation time (approximately 77% of compute time in LSTM networks). This parameter count increase affects training time and memory consumption, with larger models requiring substantially more GPU memory.

While Run 4 demonstrates that a more complex model can approach the performance of an extensively trained smaller model in fewer epochs, the efficiency comparison favors Run 3 for production environments where computational resources are constrained. The resource utilization differences become particularly pronounced for large batch sizes, where the computational demands of larger hidden sizes grow non-linearly.

Run 4's approach might be preferable for time-sensitive applications requiring rapid deployment despite its higher resource cost per epoch. However, for scenarios where computing resources are limited or energy efficiency is prioritized, the extended training of a smaller model (Run 3) provides a more resource-efficient path to optimal performance.

V. Beyond LSTMs

While LSTMs have proven highly effective for many sequence modeling tasks, a notable variant called the Gated Recurrent Unit (GRU) offers a more straightforward and often more computationally efficient alternative.

GRUs simplify the LSTM architecture primarily by reducing the number of gates. An LSTM employs three gates (input, forget, and output) and a separate cell state. In contrast, a GRU uses only two gates: an *update gate* and a *reset gate*. The update gate in a GRU essentially combines the roles of the LSTM's input and forget gates, deciding both what new information to store and what old information to discard. The reset gate determines how much of the previous hidden state to forget when computing the new candidate hidden state. GRUs

also do not maintain a separate cell state (c_t) distinct from the hidden state (h_t), further streamlining the structure.

This simpler design translates to several practical advantages. GRUs have fewer parameters than LSTMs (roughly 3/4 the parameter cost), which can lead to:

- **Faster training and execution:** With fewer computations per time step, GRUs generally train and run faster.
- **Reduced memory consumption:** Fewer parameters mean a smaller model size.
- **Potentially easier training:** The reduced complexity can sometimes make GRUs easier to train, especially with less training data.

In terms of performance, GRUs often achieve results comparable to LSTMs, particularly on tasks that do not require the extremely nuanced long-term memory management that LSTMs' additional gate might provide. LSTMs might still have an edge on datasets with very long and complex sequences where precise control over memory is critical, such as some language modeling tasks. However, for many applications, the efficiency gains of GRUs make them an attractive choice, especially when computational resources or training time are significant concerns. The choice between LSTM and GRU often depends on the specific problem, dataset size, and available resources, with empirical testing being a common way to determine the optimal model.

VI. Conclusion

This paper detailed the development and implementation of a Long Short-Term Memory (LSTM) network from scratch, aimed at the task of predicting car trajectories. The project successfully demonstrated the capability of LSTMs to model sequential data, leveraging their unique gate architecture to capture long-term dependencies inherent in time-series information. A thorough exploration of the LSTM's internal mechanisms, including the derivation of gradients for backpropagation through time, provided a foundational understanding crucial for both implementing and interpreting the model's behavior.

The iterative process of hyperparameter adjustment, particularly focusing on learning rate and hidden layer size, proved to be a critical phase in optimizing the model's predictive performance. The experiments showed that a systematic approach to tuning, reducing an initially high learning rate for smoother convergence, and subsequently increasing the hidden layer size to enhance model capacity, led to a significant reduction in test loss, achieving a final RMSE of 12.5533. This underscores the sensitivity of deep learning models to their hyperparameters and the necessity of careful tuning for practical applications.

The implementation incorporated several modern deep learning techniques, including Xavier initialization to maintain consistent signal propagation, dropout regularization applied selectively to non-recurrent connections, Adam optimization for adaptive parameter updates, and cosine annealing learning rate scheduling to improve convergence. Our dataset distribution strategy, using a 70/15/15 split for training, validation, and testing respectively, ensured robust model evaluation throughout the development process.

In summary, this project achieved its primary goal of developing a functional LSTM for trajectory prediction and provided deep insights into the intricate workings of recurrent neural networks. The hands-on approach of building the model from fundamental operations, coupled with systematic hyperparameter tuning, culminated in an effective predictive system and a comprehensive learning experience in deep learning for sequential data. The findings reinforce the power of LSTMs in handling complex, long-time lag tasks and the indispensable role of empirical experimentation in model development.

References

[1] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, pp. 1735–1780, 11 1997.

[2] J. Chung, Ç. Gülçehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *CoRR*, vol. abs/1412.3555, 2014. [Online]. Available: http://arxiv.org/abs/1412.3555

[3] K. Greff, R. K. Srivastava, J. Koutnik, B. R. Steunebrink, and J. Schmidhuber, "Lstm: A search space odyssey," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 28, no. 10, p. 2222–2232, Oct. 2017. [Online]. Available: http://dx.doi.org/10.1109/TNNLS.2016.2582924

[4] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533–536, 1986. [Online]. Available: https://api.semanticscholar.org/CorpusID:205001834

APPENDIX

IMPORTS

```
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
import torch
import torch.nn as nn
from torch.utils.data import DataLoader, TensorDataset, Dataset

# Device setup
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using␣device:␣{device}")
```

LOADING THE DATASET

```
data_directory = "data/Car_data/car_data"

X_data = []
y_data = []

for filename in os.listdir(data_directory):
    if filename.endswith('.csv'):
        file_path = os.path.join(data_directory, filename)

        df = pd.read_csv(file_path)

        file_data = df.values

        if len(file_data) >= 67:
            X_data.append(file_data[:62])
            y_data.append(file_data[62:67])
        else:
            print(f"Warning:␣File␣{filename}␣has␣fewer␣than␣67␣rows␣and␣will␣be␣skipped.")

X_tensor = torch.FloatTensor(np.array(X_data))
y_tensor = torch.FloatTensor(np.array(y_data))

dataset = TensorDataset(X_tensor, y_tensor)
dataloader = DataLoader(dataset, batch_size=128, shuffle=False)

print(f"Loaded␣{len(X_data)}␣files␣for␣training")
print(f"Input␣shape:␣{X_tensor.shape}")
print(f"Target␣shape:␣{y_tensor.shape}")

# First split data into temp (train+validation) and test
X_temp, X_test, y_temp, y_test = train_test_split(X_tensor.numpy(), y_tensor.numpy(), test_size
    =0.15, random_state=42)

# Then split the temp data into train and validation
# 0.176 of 85% is ~15% of the original dataset
X_train, X_val, y_train, y_val = train_test_split(X_temp, y_temp, test_size=0.176, random_state
    =42)

# Convert back to tensors
X_train_tensor = torch.FloatTensor(X_train)
y_train_tensor = torch.FloatTensor(y_train)
X_val_tensor = torch.FloatTensor(X_val)
y_val_tensor = torch.FloatTensor(y_val)
X_test_tensor = torch.FloatTensor(X_test)
y_test_tensor = torch.FloatTensor(y_test)

# Create datasets
```

```
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
val_dataset = TensorDataset(X_val_tensor, y_val_tensor)
test_dataset = TensorDataset(X_test_tensor, y_test_tensor)

# Create dataloaders
batch_size = 128
train_dataloader = DataLoader(train_dataset, batch_size=batch_size, shuffle=False)
val_dataloader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False)
test_dataloader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

print(f"Training␣set:␣{X_train_tensor.shape},␣{y_train_tensor.shape}")
print(f"Validation␣set:␣{X_val_tensor.shape},␣{y_val_tensor.shape}")
print(f"Test␣set:␣{X_test_tensor.shape},␣{y_test_tensor.shape}")
```

LSTM MODULE

```
class LSTM_custom(nn.Module):
    def __init__(self, input_size=12, hidden_size=64, dropout=0.2):
        super(LSTM_custom, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.dropout_rate = dropout
        self.projection_layer = nn.Linear(self.hidden_size, input_size)
        self.dropout = nn.Dropout(dropout)  # Add dropout layer

        # Input gate
        self.W_xi = nn.Parameter(torch.nn.init.xavier_uniform_(torch.empty(input_size,
            hidden_size)))
        self.b_xi = nn.Parameter(torch.zeros(hidden_size))
        self.W_hi = nn.Parameter(torch.nn.init.xavier_uniform_(torch.empty(hidden_size,
            hidden_size)))
        self.b_hi = nn.Parameter(torch.zeros(hidden_size))

        # Forget gate
        self.W_xf = nn.Parameter(torch.nn.init.xavier_uniform_(torch.empty(input_size,
            hidden_size)))
        self.b_xf = nn.Parameter(torch.zeros(hidden_size))
        self.W_hf = nn.Parameter(torch.nn.init.xavier_uniform_(torch.empty(hidden_size,
            hidden_size)))
        self.b_hf = nn.Parameter(torch.zeros(hidden_size))

        # Cell gate
        self.W_xg = nn.Parameter(torch.nn.init.xavier_uniform_(torch.empty(input_size,
            hidden_size)))
        self.b_xg = nn.Parameter(torch.zeros(hidden_size))
        self.W_hg = nn.Parameter(torch.nn.init.xavier_uniform_(torch.empty(hidden_size,
            hidden_size)))
        self.b_hg = nn.Parameter(torch.zeros(hidden_size))

        # Output gate
        self.W_xo = nn.Parameter(torch.nn.init.xavier_uniform_(torch.empty(input_size,
            hidden_size)))
        self.b_xo = nn.Parameter(torch.zeros(hidden_size))
        self.W_ho = nn.Parameter(torch.nn.init.xavier_uniform_(torch.empty(hidden_size,
            hidden_size)))
        self.b_ho = nn.Parameter(torch.zeros(hidden_size))

    def input_gate(self, x_t, h_prev):
        return torch.sigmoid(
            torch.matmul(x_t, self.W_xi) + self.b_xi + torch.matmul(h_prev, self.W_hi) + self.
                b_hi
        )

    def forget_gate(self, x_t, h_prev):
        return torch.sigmoid(
```

```
            torch.matmul(x_t, self.W_xf) + self.b_xf + torch.matmul(h_prev, self.W_hf) + self.
                b_hf
        )

    def cell_gate(self, x_t, h_prev):
        return torch.tanh(
            torch.matmul(x_t, self.W_xg) + self.b_xg + torch.matmul(h_prev, self.W_hg) + self.
                b_hg
        )

    def output_gate(self, x_t, h_prev):
        return torch.sigmoid(
            torch.matmul(x_t, self.W_xo) + self.b_xo + torch.matmul(h_prev, self.W_ho) + self.
                b_ho
        )

    def lstm_cell(self, x_t, h_prev, c_prev):
        i_t = self.input_gate(x_t, h_prev)
        f_t = self.forget_gate(x_t, h_prev)
        g_t = self.cell_gate(x_t, h_prev)
        o_t = self.output_gate(x_t, h_prev)

        c_t = f_t * c_prev + i_t * g_t
        h_t = o_t * torch.tanh(c_t)
        h_t = self.dropout(h_t)
        return h_t, c_t

    def forward(self, x):
        batch_size, seq_len, feature_dim = x.size()
        h_t = torch.zeros(batch_size, self.hidden_size, device=x.device)
        c_t = torch.zeros(batch_size, self.hidden_size, device=x.device)

        for t in range(seq_len):
            h_t, c_t = self.lstm_cell(x[:, t, :], h_t, c_t)

        predictions = []
        current_x = x[:, -1, :]
        for i in range(5):
            h_t, c_t = self.lstm_cell(current_x, h_t, c_t)
            output = self.projection_layer(h_t)
            predictions.append(output)
            current_x = output
        return torch.stack(predictions, dim=1)
```

RMSE Loss

```
def RMSELoss(y_pred, y_true):
    return torch.sqrt(nn.MSELoss()(y_pred, y_true) + 1e-8)
```

Train Function

```
def  train_model(name, input_size=12, hidden_size=64, dropout=0.2, lr= 0.01, num_epochs=100):
    model = LSTM_custom(input_size=input_size, hidden_size=hidden_size, dropout=dropout).to(
        device)
    criterion = RMSELoss
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)
    lr_scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=num_epochs)

    train_losses = []
    val_losses = []
    learning_rates = []

    for epoch in range(num_epochs):
        model.train()
        train_loss = 0
```

```
        for inputs, targets in train_dataloader:
            inputs = inputs.to(device)
            targets = targets.to(device)
            outputs = model(inputs)
            loss = criterion(outputs, targets)

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            train_loss += loss.item()

        avg_train_loss = train_loss / len(train_dataloader)

        if (epoch + 1) % 10 == 0:
            model.eval()
            val_loss = 0
            with torch.no_grad():
                for inputs, targets in val_dataloader:
                    inputs = inputs.to(device)
                    targets = targets.to(device)

                    outputs = model(inputs)
                    loss = criterion(outputs, targets)

                    val_loss += loss.item()
            avg_val_loss = val_loss / len(val_dataloader)
        else:
            avg_val_loss = None

        lr_scheduler.step()

        train_losses.append(avg_train_loss)
        val_losses.append(avg_val_loss)
        learning_rates.append(optimizer.param_groups[0]['lr'])

        if (epoch + 1) % 10 == 0:
            print(f"Epoch [{epoch + 1}/{num_epochs}], "
                f"Train Loss: {avg_train_loss:.4f}, "
                f"Val Loss: {avg_val_loss:.4f}, "
                f"LR: {optimizer.param_groups[0]['lr']:.6f}")


    plt.figure(figsize=(12, 4))
    plt.subplot(1, 2, 1)
    plt.plot(train_losses, 'b-', label='Training Loss')
    val_epochs = [i for i, v in enumerate(val_losses) if v is not None]
    val_losses_filtered = [v for v in val_losses if v is not None]
    plt.plot(val_epochs, val_losses_filtered, 'r-', label='Validation Loss')
    plt.title(f'Training and Validation Loss for {name}')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
    plt.grid(True)

    return model
```

TEST FUNCTION

```
test_results = {}
def test_model(name, model, test_dataloader):
    criterion = RMSELoss

    model.eval()
    with torch.no_grad():
        test_loss = 0
```

```
        for inputs, targets in test_dataloader:
            inputs = inputs.to(device)
            targets = targets.to(device)
            outputs = model(inputs)
            test_loss += criterion(outputs, targets).item()

        avg_test_loss = test_loss / len(test_dataloader)
        print(f"Test␣Loss␣for␣{name}:␣{avg_test_loss:.4f}")
    return avg_test_loss
```

TRAIN AND TEST FUNCTION

```
def train_and_test_model(name, input_size=12, hidden_size=64, dropout=0.2, lr= 0.01, num_epochs
    =100):
    model = train_model(name, input_size=input_size, hidden_size=hidden_size, dropout=dropout,
        lr=lr, num_epochs=num_epochs)
    return test_model(name, model, test_dataloader)
```

TESTS

```
test_results["512␣|␣0.2␣|␣0.01␣|␣100"] =  train_and_test_model("512␣|␣0.2␣|␣0.01␣|␣100",
    input_size=12, hidden_size=512, dropout=0.2, lr=0.01, num_epochs=100)
```

```
test_results["512␣|␣0.2␣|␣0.001␣|␣100"] =  train_and_test_model("512␣|␣0.2␣|␣0.001␣|␣100",
    input_size=12, hidden_size=512, dropout=0.2, lr=0.001, num_epochs=100)
```

```
test_results["512␣|␣0.001␣|␣300"] =  train_and_test_model("512␣|␣0.001␣|␣300", input_size=12,
    hidden_size=512, dropout=0.2, lr=0.001, num_epochs=300)
```

```
test_results["1024␣|␣0.2␣|␣0.001␣|␣100"] =  train_and_test_model("1024␣|␣0.2␣|␣0.001␣|␣100",
    input_size=12, hidden_size=1024, dropout=0.2, lr=0.001, num_epochs=100)
```