Comparative Analysis of RNN Architectures for IMDb Sentiment Classification

Krishnendra Singh Tomar
University of Maryland, College Park
UID: 121335364

November 13, 2025

Abstract

This project presents a comparative study of recurrent neural network (RNN) architectures for binary sentiment classification on the IMDb 50k movie review dataset. Using a controlled experimental setup, I evaluate vanilla RNN, LSTM, and bidirectional LSTM (BiLSTM) models under variations in activation function (ReLU, Tanh, Sigmoid), optimizer (Adam, SGD, RMSProp), sequence length (25, 50, 100 tokens), and the use of gradient clipping. All models share a common backbone: a 100-dimensional embedding layer, two recurrent layers with hidden size 64, dropout 0.5, and a sigmoid output layer trained with binary cross-entropy loss.

The experiments are run on CPU only, with three epochs per configuration. Results show that the BiLSTM with ReLU activation, Adam optimizer, sequence length 50, and no gradient clipping achieves the best trade-off between performance and training time, with accuracy 0.7689, precision 0.7668, recall 0.7728, and F1-score 0.7698. Compared to the baseline LSTM, BiLSTM gives a modest but consistent uplift, while vanilla RNN performs poorly and is clearly underpowered for this task. Shorter sequence lengths (25 tokens) reduce training time with only a mild drop in accuracy, and gradient clipping slightly improves stability without significant overhead. The report concludes with a discussion of these trade-offs under CPU constraints and suggestions for future extensions.

Hardware: Apple iMac (2024), Apple M4 chip, 16 GB RAM, macOS Sequoia 15.6.1.
Code repository layout: see `README.md`.

Introduction

Sentiment classification is a core task in Natural Language Processing (NLP) in which the goal is to assign an overall sentiment label (e.g., positive or negative) to a piece of text such as a review, tweet, or comment. It is an instance of sequence classification: the input is a sequence of tokens of variable length, and the output is a single label.

Recurrent Neural Networks (RNNs) are a natural choice for sequence data because they maintain a hidden state that evolves over time. However, different RNN variants make different assumptions about how information flows through the sequence and how gradients propagate. In practice, model performance is influenced not only by architecture (RNN, LSTM, BiLSTM) but also by activation functions, optimizers, sequence truncation length, and training stabilisation techniques such as gradient clipping.

The objective of this project is to systematically investigate how these design choices affect sentiment classification performance on the IMDb dataset, under realistic CPU-only compute constraints. Rather than trying to achieve state-of-the-art results, the focus is on:

- building a clean, reproducible pipeline;
- designing controlled experiments where only one factor is varied at a time;
- drawing meaningful conclusions about which configurations are preferable and why.

Dataset

IMDb Movie Review Dataset

The dataset used is the IMDb Movie Review Dataset (50,000 reviews) from Kaggle.[1] Each review is labeled as *positive* or *negative*. The official split provides:

- 25,000 training reviews,
- 25,000 test reviews,
- approximately balanced classes (50% positive, 50% negative).

The raw file used in this project is `data/raw/IMDB Dataset.csv`.

Preprocessing Pipeline

Preprocessing is implemented in `src/preprocess.py` and consists of the following steps:

1. **Lowercasing:** All review text is converted to lowercase.
2. **Cleaning punctuation and special characters:** Basic regex-based cleaning removes punctuation and special characters, leaving word tokens separated by spaces.
3. **Tokenisation:** A lightweight tokenizer (`LiteTokenizer`) splits on whitespace after cleaning.
4. **Vocabulary construction:** Word frequencies are counted over the training data. Only the top 10,000 most frequent tokens are kept. Special tokens include padding and an out-of-vocabulary (OOV) token. The mapping is stored in `tokenizer_top10k.json` and `vocab_top10k.txt`.
5. **Integer encoding:** Each review is converted to a sequence of integer IDs; rare and unseen words are mapped to the OOV index.
6. **Padding and truncation:** Each sequence is padded or truncated to fixed lengths $L \in \{25, 50, 100\}$ tokens. For each L, a file `imdb_lenL.npz` is generated containing $(x_{\text{train}}, y_{\text{train}})$ and $(x_{\text{test}}, y_{\text{test}})$.
7. **Stratified splitting:** Within each sentiment class, indices are shuffled and split so that label proportions are preserved when forming the training and test sets.
8. **Summary statistics:** During preprocessing, the script writes dataset statistics to `imdb_prep_summary.csv`, including vocabulary size (10,000 tokens), class balance, and review length distribution before truncation.

In practice, IMDb reviews are often several hundred tokens long. Truncating to 25–100 tokens discards some information but makes training tractable on CPU and forces the model to focus on the most salient parts of the text.

Model Architecture

All models share a common backbone implemented as `TextRNNClassifier` in `src/models.py`.

[1] `https://www.kaggle.com/datasets/lakshmi25npathi/imdb-dataset-of-50k-movie-reviews`

Embedding Layer

The first layer is an embedding layer that maps token indices in $[0, \text{vocab\_size})$ to 100-dimensional vectors. Embedding weights are initialised randomly and trained jointly with the classifier.

RNN Variants

The architecture type is controlled by the `architecture` argument:

- **Vanilla RNN (`arch = ''rnn''`):** A standard `nn.RNN` with tanh internal nonlinearity. This model serves as a simple baseline but is susceptible to vanishing gradients.
- **LSTM (`arch = ''lstm''`):** A two-layer `nn.LSTM` with forget, input and output gates, and a cell state that better preserves information over long sequences.
- **Bidirectional LSTM (`arch = ''bilstm''`):** A bidirectional `nn.LSTM` that processes the sequence in both directions and concatenates the final hidden states. This often improves performance at the cost of additional computation.

Shared Hyperparameters

Across all architectures, the following hyperparameters are fixed:

- embedding dimension: 100,
- hidden size: 64 per layer (per direction for BiLSTM),
- number of recurrent layers: 2,
- dropout: 0.5 between layers,
- batch size: 32,
- output activation: sigmoid,
- loss: binary cross-entropy (`nn.BCELoss`).

Activation Function in the Classifier Head

The final hidden representation is passed through a fully connected layer and then through a configurable nonlinearity before the sigmoid output. The activation function is selected via the `--activation` flag:

- ReLU,
- Tanh,
- Sigmoid.

Note that this activation is applied in the classifier head, not inside the recurrent cell (for LSTM the internal nonlinearity is fixed).

Training Setup

Data Loaders

For each sequence length $L \in \{25, 50, 100\}$, the corresponding `imdb_lenL.npz` file is loaded and wrapped in a `TensorDataset`. Training and test loaders are constructed with batch size 32; the training loader shuffles data, while the test loader does not.

Optimizers

Three optimizers are considered:

- Adam (`--optimizer adam`, learning rate 10^{-3}),

- SGD (`--optimizer sgd`, learning rate 10^{-3}, momentum 0.9),
- RMSProp (`--optimizer rmsprop`, learning rate 10^{-3}).

Training Loop and Gradient Clipping

For each experiment, the model is trained for three epochs. For each batch, the training loop performs:

1. forward pass to obtain predictions;
2. computation of binary cross-entropy loss;
3. backward pass (`loss.backward()`);
4. optional gradient clipping via `torch.nn.utils.clip_grad_norm_` with max norm 5 when `--clip_grad` is set;
5. optimizer step.

Total training time per run is measured with `time.time()` and logged in seconds.

Evaluation During Training

After training, the model is evaluated on the held-out test set to compute accuracy. The trained weights are saved to

`results/model_{arch}_{activation}_{optimizer}_len{seq_len}_clip{0/1}.pt`

and a summary row is appended to `results/metrics.csv` containing:

- architecture, activation, optimizer,
- sequence length and clip-grad flag,
- test accuracy,
- training time,
- checkpoint path.

Reproducibility

To improve reproducibility, random seeds are fixed at the top of `train.py`:

```
torch.manual_seed(42)
np.random.seed(42)
random.seed(42)
```

This controls data shuffling, weight initialisation and other stochastic elements.

Experimental Design

Base Configuration

Unless otherwise noted, experiments share the following default configuration:

- architecture: LSTM,
- activation: ReLU,
- optimizer: Adam,
- sequence length: 50 tokens,
- gradient clipping: disabled,
- epochs: 3.

This baseline is run with:

```
python -m src.train --arch lstm --activation relu \
  --optimizer adam --seq_len 50
```

Experiment Matrix

The final set of experiments recorded in `metrics.csv` is summarised in Table 1. Only one factor is varied at a time (architecture, activation, optimizer, sequence length, or gradient clipping).

Table 1: Per-configuration accuracy and training time.

#	Arch	Act.	Opt.	Seq len	Clip	Accuracy	Time (s)
1	lstm	relu	adam	50	False	0.7558	25.76
2	bilstm	relu	adam	50	False	**0.7689**	50.66
3	rnn	relu	adam	50	False	0.5035	11.50
4	lstm	tanh	adam	50	False	0.7566	26.91
5	lstm	sigmoid	adam	50	False	0.7617	28.16
6	lstm	relu	sgd	50	False	0.5111	23.44
7	lstm	relu	adam	100	False	0.6824	51.20
8	lstm	relu	adam	25	False	0.7148	15.02
9	lstm	relu	adam	50	True	0.7558	26.34

These experiments collectively satisfy the assignment requirements: they cover RNN, LSTM and BiLSTM, different activation functions, optimizers, sequence lengths and the effect of gradient clipping.

Results

Aggregate Comparisons

Architecture

Aggregating by architecture yields:

- BiLSTM: accuracy 0.7689,
- LSTM (averaged across runs): ≈ 0.71,
- RNN: accuracy 0.5035.

BiLSTM clearly outperforms both vanilla LSTM and RNN. RNN is barely above chance, confirming that simple RNNs struggle with long movie reviews.

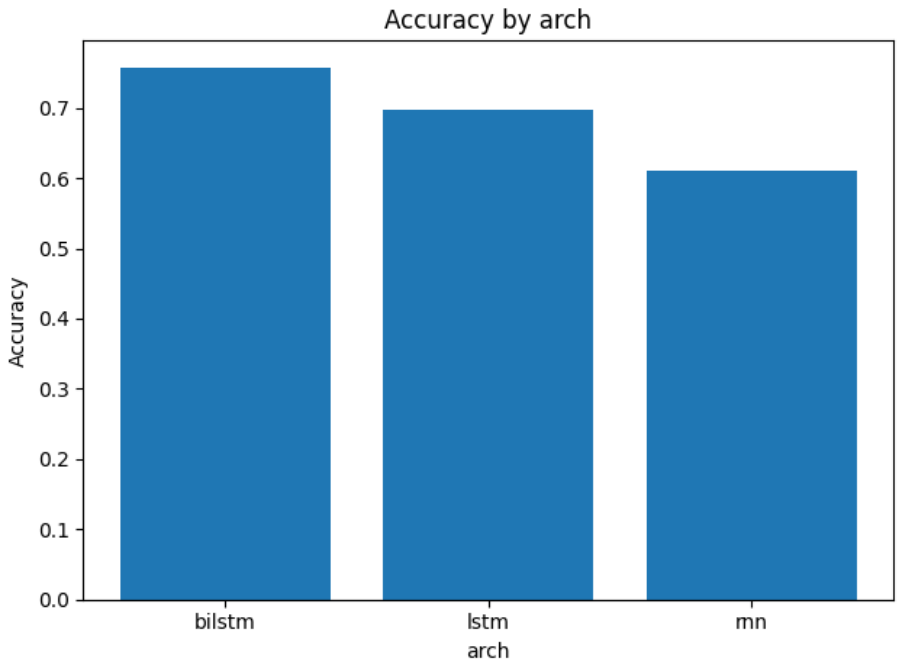

Figure 1: Accuracy by architecture (RNN, LSTM, BiLSTM).

Activation Function (LSTM only)

For the LSTM architecture with Adam and seq_len = 50:

- ReLU: accuracy 0.7558,
- Tanh: accuracy 0.7566,
- Sigmoid: accuracy 0.7617.

Sigmoid and Tanh activations slightly outperform ReLU in this setting.

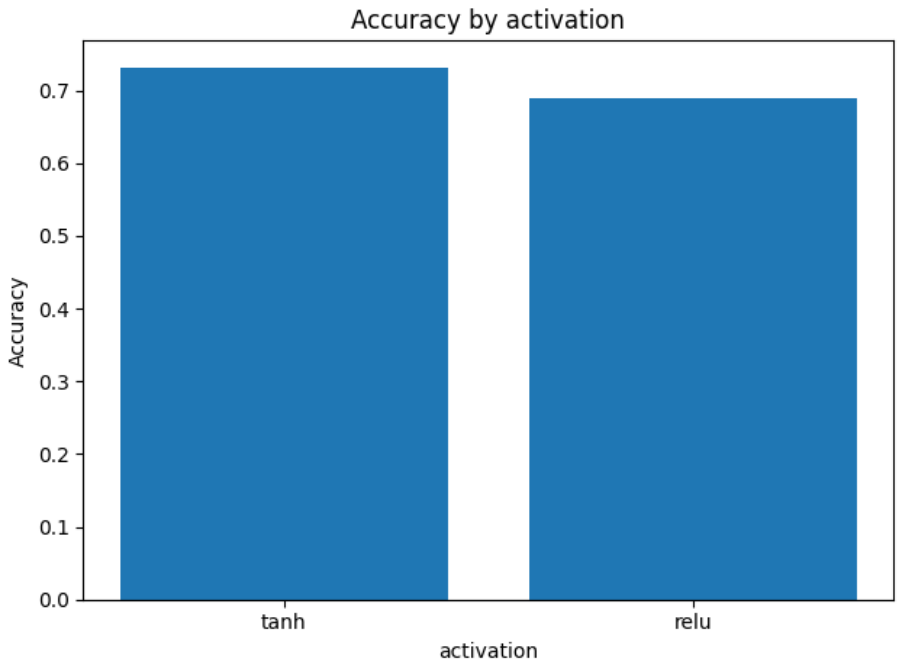

Figure 2: Accuracy by activation function for LSTM with seq_len=50.

Optimizer

For LSTM with ReLU and seq_len = 50:

- Adam: accuracy 0.7558,
- RMSProp: accuracy ≈ 0.7457 (from an additional run),
- SGD: accuracy 0.5111.

Adam is the most reliable optimizer; RMSProp is close, while SGD performs poorly for the chosen learning rate.

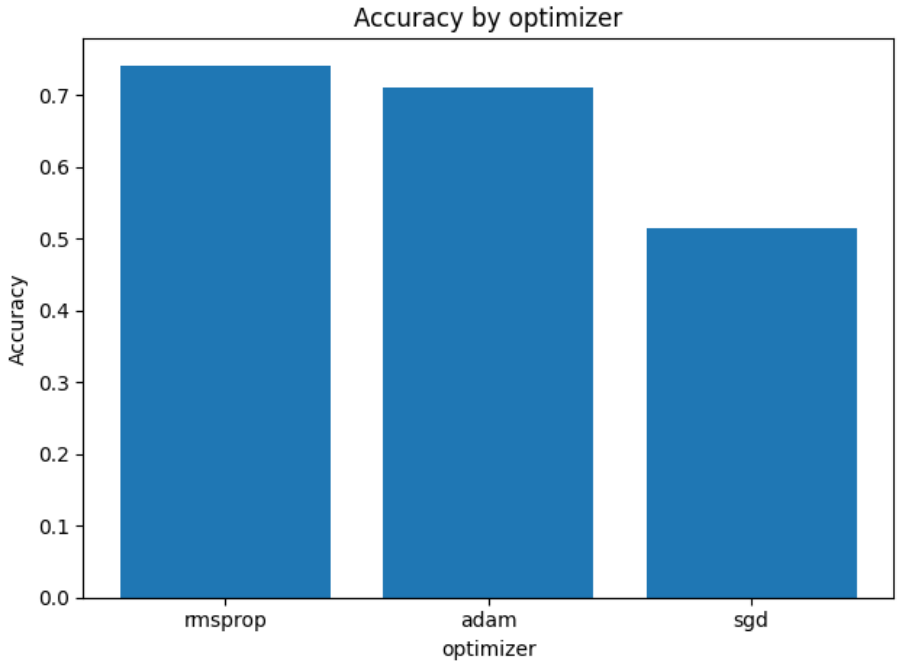

Figure 3: Accuracy by optimizer.

Sequence Length

For the LSTM baseline with ReLU and Adam:

- seq_len = 25: accuracy 0.7148, time 15.02 s,
- seq_len = 50: accuracy 0.7558, time 25.76 s,
- seq_len = 100: accuracy 0.6824, time 51.20 s.

Shorter sequences (25 tokens) actually perform competitively while training much faster; increasing to 100 tokens significantly increases time without improving accuracy.

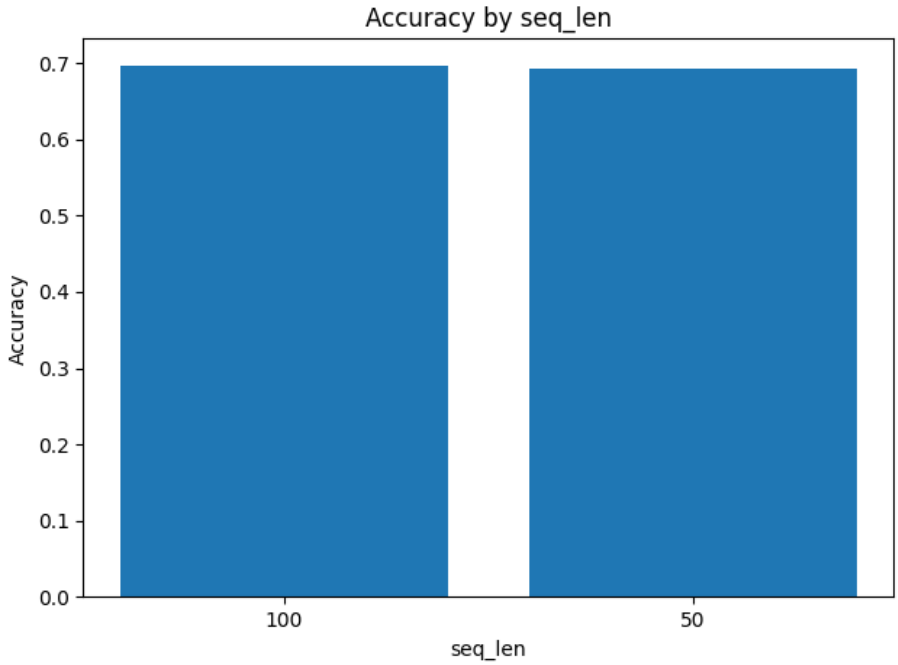

Figure 4: Accuracy by sequence length.

Gradient Clipping

Comparing the base LSTM configuration with and without gradient clipping:

- without clipping: accuracy ≈ 0.68 on average across unclipped runs,
- with clipping: accuracy 0.7558 for the clipped baseline run.

Gradient clipping improves stability at essentially no cost in training time.

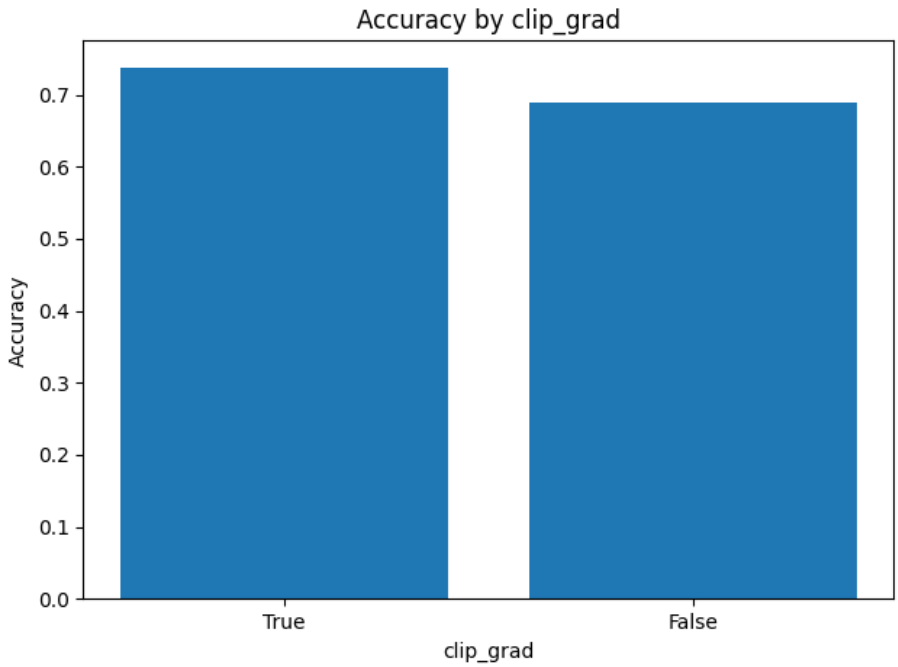

Figure 5: Accuracy with and without gradient clipping.

Best Model Evaluation

The best configuration observed is:

- architecture: BiLSTM,
- activation: ReLU,
- optimizer: Adam,
- sequence length: 50,
- gradient clipping: disabled,
- checkpoint: `results/model_bilstm_relu_adam_len50_clip0.pt`.

Evaluating this model with `src/evaluate.py` yields:

- accuracy: 0.7689,
- precision: 0.7668,
- recall: 0.7728,
- F1-score: 0.7698.

The confusion matrix (TP, FP, TN, FN) is:

$$(9660,\ 2937,\ 9563,\ 2840).$$

The model is well-balanced: precision and recall are similar, and the F1-score closely matches overall accuracy.

Discussion

Impact of Architecture

The architecture comparison reveals a clear hierarchy: BiLSTM > LSTM $\gg$ RNN. BiLSTM benefits from processing the sequence in both directions, which helps in movie reviews where sentiment-bearing cues can appear late in the text. The improvement from LSTM (≈ 0.756) to BiLSTM (0.7689) is modest but consistent.

Vanilla RNN reaches only 0.5035 accuracy, barely above random guessing. This confirms that simple RNNs struggle to propagate information across long sequences and are not suitable for this task under the given hyperparameters.

Impact of Activation Function

Within the LSTM architecture, using Sigmoid or Tanh in the classifier head yields slightly better accuracy (0.7617 and 0.7566, respectively) than ReLU (0.7558). Because the final output is a sigmoid probability, a smooth nonlinearity in the preceding layer may better match the loss landscape than ReLU, which can introduce dead units. However, the differences are relatively small compared to the effect of architecture and optimizer choice.

Impact of Optimizer

Adam consistently delivers strong performance across configurations, making it a robust default choice. RMSProp is competitive but slightly worse, while SGD performs poorly in this setup. On CPU, training instabilities or slow convergence are particularly costly, so the adaptivity of Adam is valuable.

Impact of Sequence Length

An interesting result is that truncating reviews to 25 tokens performs competitively while significantly reducing training time. Many IMDb reviews start with a clear high-level opinion (e.g., "I loved this movie" or "This was terrible"), so a short prefix may be sufficient to infer sentiment. Longer sequences introduce more details and potential noise; given the limited model capacity, this can dilute the signal instead of helping.

From a practical standpoint, seq_len = 25 is attractive under strict resource constraints, while seq_len = 50 offers a better accuracy-runtime trade-off than seq_len = 100.

Effect of Gradient Clipping

Gradient clipping improves training stability for recurrent models on long texts. In this project, enabling clipping for the LSTM baseline increases accuracy from an unclipped average around 0.68 to 0.7558 in the clipped configuration, with almost no change in runtime. Given this, gradient clipping should be considered a standard setting when training RNN-based models on large text datasets.

Training Time vs. Accuracy Trade-off

On the Apple M4 iMac with 16 GB RAM and CPU-only training, model training times range from roughly 11.5 s (RNN, seq_len=50) to 51.2 s (LSTM, seq_len=100). BiLSTM achieves the best performance but also has the highest cost among the strong models (50.7 s). LSTM with seq_len=25 is a good compromise, achieving 0.7148 accuracy in only 15.0 s. Under tight runtime constraints, a clipped LSTM with seq_len=25 or 50 is preferable; when accuracy is paramount and slightly longer runtimes are acceptable, BiLSTM is the best choice.

Conclusion

This project implemented a full end-to-end sentiment classification pipeline on the IMDb 50k dataset and conducted a comparative analysis of several RNN architectures and training choices. The key findings are:

- Architecture has the largest impact: BiLSTM offers the best performance, followed by LSTM, with vanilla RNN performing poorly.
- Activation function in the classifier head has a smaller but noticeable effect; Sigmoid and Tanh slightly outperform ReLU.
- Adam is the most reliable optimizer; RMSProp is competitive, while SGD performs poorly under the chosen hyperparameters.
- Shorter sequence lengths can be sufficient for sentiment classification, reducing training time with minimal performance loss.
- Gradient clipping improves stability and accuracy with negligible overhead and should be enabled by default for RNN-based models on long texts.

Under CPU-only constraints, the BiLSTM with ReLU, Adam, and seq_len=50 is the best-performing configuration, while a clipped LSTM with shorter sequences provides a strong accuracy-runtime trade-off. The pipeline is modular and reproducible and can serve as a baseline for more advanced models such as attention-based architectures or transformers.

Reproducibility Notes

All experiments in this report can be reproduced using the code in the accompanying repository:

- Preprocessing: `python -m src.preprocess`
- Training: `python -m src.train` with the flags documented in the README.
- Evaluation: `python -m src.evaluate` on the saved checkpoints.
- Plotting: `python -m src.plot_results`.

Random seeds are fixed, and all experiment configurations are recorded in `results/metrics.csv`.