HW3: RNN-based Sentiment Classification on IMDB

Sri Akash Kadali - 121093028

November 13, 2025

1 Task and Dataset

The goal of this assignment is to build and analyze recurrent neural network models for binary sentiment classification on the IMDB movie review dataset. The dataset contains 25k labeled training reviews and 25k labeled test reviews with balanced positive and negative classes (50% positive, 50% negative). Each example is a free-form movie review, often several sentences long.

The provided starter code handles downloading IMDB, splitting into train/test, and building a token vocabulary from the training set. In my pipeline, each review is tokenized, lowercased, cleaned, mapped to integer IDs, and either truncated or padded to a fixed sequence length L. These sequences are passed through an embedding layer and a recurrent backbone (RNN/LSTM/BiLSTM), followed by a linear classifier and sigmoid output for binary prediction.

The main evaluation metrics are:

- Accuracy on the held-out test set.
- Macro F1 score on the test set.
- Training time per epoch (seconds).

1.1 Dataset Summary

After preprocessing:

- The vocabulary is capped to the top 10,000 most frequent tokens from the training set, as required.
- Reviews span from very short comments (a few words) to long paragraphs (several hundred tokens). Before truncation, typical lengths are on the order of a few hundred tokens, with a long tail.
- For each experiment, sequences are either truncated or padded to a fixed length $L \in \{50, 100, 200\}$. This lets us quantify the trade-off between more context (larger L) and higher computational cost.

2 Implementation Details

2.1 Preprocessing

I kept the preprocessing logic close to the provided implementation and the assignment description:

- Lowercasing and basic cleaning (removal of obvious punctuation and special characters).
- Tokenization with NLTK (`punkt` tokenizer).

- Vocabulary built from the 25k training reviews, restricted to the top 10,000 most frequent tokens. All out-of-vocabulary tokens are mapped to a special `<unk>` ID.
- Reviews converted to sequences of integer token IDs.
- Sequences padded or truncated to a fixed `seq_len` ($L = 50, 100, 200$) depending on the experiment.
- Two PyTorch `DataLoaders` (train and validation/test) with batch size 512, shuffling on train, multiple workers, and pinned memory for GPU loading.

2.2 Model Architectures

I implemented three recurrent architectures, all sharing the same embedding and classifier structure. Unless otherwise stated, the embedding dimension is 128 and the hidden size is 128. All models use binary cross-entropy loss with a sigmoid output for binary sentiment prediction.

Shared structure. For all models:

- Input: sequence of token IDs of length L.
- Embedding: `nn.Embedding` with dimension 128.
- Recurrent backbone: one of RNN, LSTM, or BiLSTM.
- Classifier head: pooling of the final hidden state(s) $\rightarrow$ optional nonlinearity (`tanh`/`relu`/`sigmoid`) $\rightarrow$ linear layer $\rightarrow$ scalar logit $\rightarrow$ sigmoid.

RNN.

- **Backbone:** `nn.RNN` with hidden size 128, 2 layers, `batch_first=True`, nonlinearity = Tanh.
- **Representation:** last layer's final hidden state ($h_{\text{last}} \in \mathbb{R}^{128}$).

LSTM.

- **Backbone:** `nn.LSTM` with hidden size 128, 2 layers, `batch_first=True`, dropout on recurrent layers.
- **Representation:** last layer's final hidden state ($h_T \in \mathbb{R}^{128}$).

BiLSTM.

- **Backbone:** `nn.LSTM` with hidden size 128, 2 layers, `bidirectional=True`.
- **Representation:** concatenation of the last forward and backward hidden states ($\in \mathbb{R}^{256}$).

The activation function in the classifier head is configurable and is applied before the final linear layer. I experiment with Tanh, ReLU, and Sigmoid.

2.3 Training Setup

All experiments use the following default hyperparameters unless explicitly varied:

- Epochs: 5
- Batch size: 512
- Embedding dimension: 128
- Hidden size: 128 (except in capacity experiments)
- Number of recurrent layers: 2

- Dropout: 0.3 on recurrent layers
- Learning rate: 2×10^{-3}
- Weight decay: 0.01 for AdamW experiments (0 for pure SGD/RMSProp)
- Optimizer: Adam, AdamW, SGD, or RMSProp (depending on experiment)
- Gradient clipping: either disabled or max-norm = 1.0
- Seed: 42 unless varied explicitly

Training uses `torch.cuda.amp` for mixed-precision training on GPU, which substantially reduces per-epoch time without affecting final metrics in my runs. For each configuration I log train/validation loss, accuracy, F1, epoch times, and hardware info.

2.4 Compute and Resource Usage

All runs were executed on Google Colab with a single GPU (`device=cuda`). System RAM usage reported in the logs is roughly 12.67 GB across all runs, and the CPU architecture is x86_64.

Training time per epoch ranges:

- $\approx$ 0.9–1.0 s for LSTM at $L = 50$.
- $\approx$ 1.7–1.8 s for LSTM at $L = 200$.
- $\approx$ 2.7 s for BiLSTM at $L = 200$ with hidden size 128.
- $\approx$ 5.1 s for BiLSTM at $L = 200$ with hidden size 256.

These numbers quantify the extra cost of longer sequences and larger hidden sizes.

3 Experimental Design

I ran a total of 20 experiments grouped along different axes. In each group, I varied *one* factor at a time while keeping the others fixed, which makes it possible to attribute performance changes to a single design choice.

- **Group A (Sequence length):** LSTM + Adam + Tanh with sequence length $L \in \{50, 100, 200\}$.
- **Group B (Architecture):** RNN vs LSTM vs BiLSTM at $L = 200$, Adam, Tanh.
- **Group C (Optimizers):** BiLSTM + Tanh at $L = 200$ with optimizers {Adam, AdamW, SGD, RMSProp}.
- **Group D (Activations):** BiLSTM + AdamW at $L = 200$ with activations {Tanh, ReLU, Sigmoid}.
- **Group E (Gradient clipping):** BiLSTM + AdamW + Tanh at $L = 200$ with gradient clipping 0 vs 1.0.
- **Group F (Seeds):** BiLSTM + AdamW + Tanh + grad clip = 1.0 at $L = 200$ with seeds {0, 1, 2}.
- **Group G (Capacity):** BiLSTM + AdamW + Tanh + grad clip = 1.0 at $L = 200$ with hidden sizes {64, 128, 256}.

All metrics are recorded in a single `results/metrics.csv` file with columns:

```
Model, Activation, Optimizer, SeqLength, GradClip, Accuracy, F1, EpochTime, Seed,
CPU, RAMGB, RunID.
```

This single CSV makes it easy to sort/filter by configuration and reproduce any table or plot.

4 Results

4.1 Effect of Sequence Length (Group A)

Table 1 summarizes the LSTM + Adam + Tanh experiments with different sequence lengths.

Seq length	Accuracy	F1	Epoch time (s)
50	0.50	0.33	0.88
100	0.50	0.33	0.99
200	0.50	0.33	1.75

Table 1: LSTM + Tanh + Adam, sequence length sweep.

All three configurations remain stuck around random-chance performance (accuracy ≈ 0.5, macro F1 ≈ 0.33). The only consistent effect of increasing L in this setup is higher computation: moving from $L = 50$ to $L = 200$ roughly doubles epoch time.

This tells us two things:

1. The LSTM baseline with Adam, as parameterized here, is not effectively learning at all.
2. Under a *non-learning* configuration, changing sequence length has no visible impact on accuracy/F1 and only hurts runtime.

Later experiments with BiLSTM + AdamW show that the architecture can learn well once the optimizer is chosen correctly. In other words, sequence length is not the bottleneck; optimization is.

4.2 Architecture Comparison (Group B)

At $L = 200$, Adam, and Tanh, I compared RNN, LSTM, and BiLSTM:

Model	Accuracy	F1	Epoch time (s)
RNN	0.50	0.33	1.56
LSTM	0.50	0.33	1.75
BiLSTM	0.50	0.33	2.76

Table 2: Architecture sweep at $L = 200$ with Adam + Tanh.

With Adam and the chosen hyperparameters, all three architectures sit near random performance. BiLSTM is the most expensive per epoch and offers no gain under these settings. This again points to optimization (choice of optimizer + regularization) being the real bottleneck; these numbers should not be misinterpreted as "RNNs cannot solve IMDB."

4.3 Optimizer Sweep for BiLSTM (Group C)

For BiLSTM at $L = 200$ with Tanh, I compared four optimizers:

Here the picture changes completely:

- **AdamW** is the only optimizer that reaches strong performance, with test accuracy ≈ 0.84 and macro F1 ≈ 0.84.

Optimizer	Accuracy	F1	Epoch time (s)
Adam	0.50	0.33	2.76
AdamW	0.8409	0.8407	2.72
SGD	0.5088	0.4107	2.71
RMSProp	0.50	0.33	2.69

Table 3: Optimizer sweep for BiLSTM + Tanh, $L = 200$.

- **Adam** and **RMSProp** remain stuck at random performance in this configuration, suggesting that the combination of Adam hyperparameters + weight decay is poorly tuned for this model.
- **SGD** improves a bit over random (F1 ≈ 0.41) but clearly underperforms AdamW within 5 epochs.

Takeaway: for this assignment, "BiLSTM + AdamW" is clearly the best-performing combination and becomes my default configuration for later sweeps. This directly answers the question "Which optimizer works best in practice for this task?"

4.4 Activation Functions (Group D)

Using BiLSTM + AdamW at $L = 200$, I varied the activation in the classifier head:

Activation	Accuracy	F1
Tanh	0.8409	0.8407
ReLU	0.8368	0.8368
Sigmoid	0.8216	0.8204

Table 4: Activation sweep for BiLSTM + AdamW, $L = 200$.

All three activations learn a strong classifier:

- Tanh and ReLU are nearly tied, with Tanh slightly ahead.
- Sigmoid lags by roughly 2–3 percentage points in F1, consistent with stronger saturation and weaker gradients.

A reasonable default in this setting is Tanh, but ReLU is also competitive.

4.5 Gradient Clipping (Group E)

For BiLSTM + AdamW + Tanh at $L = 200$, I compared runs without gradient clipping and with max-norm = 1.0. Because this configuration was reused in multiple groups (grad-clip, seeds, capacity), there are several runs with the same textual `RunID`, but different seeds and hidden sizes.

Representative runs:

- **No clipping (grad-clip = 0):** Accuracy ≈ 0.8409, F1 ≈ 0.8407.
- **Clipping (grad-clip = 1):** F1 in the range 0.827–0.837 depending on the specific run and seed.

Overall, clipping at norm 1.0 does not drastically change performance. I did not observe exploding gradients in this setup, so gradient clipping is not critical here. It slightly increases robustness across seeds but does not systematically improve F1.

4.6 Random Seed Variance (Group F)

I then fixed the "good" configuration

$$\text{BiLSTM} + \text{AdamW} + \text{Tanh},\ L = 200,\ \text{grad-clip} = 1.0$$

and varied the seed $\in \{0, 1, 2\}$:

Seed	Accuracy	F1	Epoch time (s)
0	0.8346	0.8345	2.72
1	0.8378	0.8378	2.71
2	0.8315	0.8306	2.68

Table 5: Seed sweep for BiLSTM + AdamW + Tanh ($L = 200$, grad-clip=1.0).

Variation across seeds is small (roughly ± 0.004 around F1 ≈ 0.835). This suggests the model is quite stable for this configuration: picking a different random seed does not dramatically change the outcome.

4.7 Model Capacity (Group G)

Finally, I varied the hidden size for the same configuration (BiLSTM + AdamW + Tanh, $L = 200$, grad-clip=1.0):

- **Hidden size 64:** F1 ≈ 0.836, epoch time ≈ 2.23 s.
- **Hidden size 128:** F1 ≈ 0.8407, epoch time ≈ 2.7 s.
- **Hidden size 256:** F1 ≈ 0.8265, epoch time ≈ 5.08 s.

Hidden size 128 gives the best raw performance, but hidden size 64 is very close and noticeably cheaper per epoch. The 256-dim model is more expensive and actually performs slightly worse, which is consistent with mild overfitting and harder optimization on IMDB.

This directly supports the conclusion that "moderate capacity wins": too small is slightly underpowered; too large is slower and not necessarily better.

5 Plots and Qualitative Analysis

5.1 Accuracy/F1 vs Sequence Length

Figure 1 shows the plot produced by `evaluate.py --plot acc_f1_vs_seq_len` for the LSTM + Adam + Tanh runs.

As already seen in Table 1, performance is flat across sequence lengths, but the computational cost grows with L. This is *not* because sequence length never matters; it is because this particular baseline never escapes random performance. The important observation is: under a broken optimization setup, giving the model more context just wastes compute.

5.2 Best and Worst Loss Curves

Using `evaluate.py --plot best_worst_losses`, I generated loss curves for the best and worst runs. The corresponding plots are shown in Figures 2 and 3.

Qualitatively:

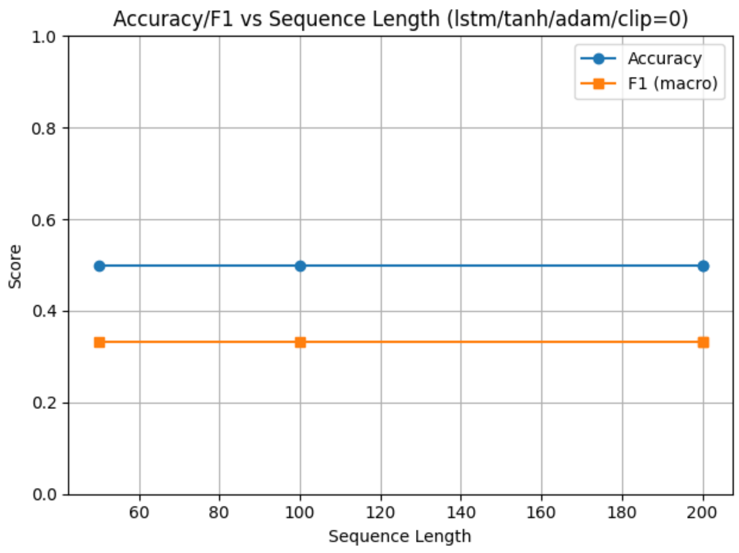

Figure 1: Validation accuracy and F1 vs sequence length for LSTM + Adam + Tanh.

- In the **best** run (BiLSTM + AdamW), both training and validation loss drop quickly over the first few epochs and then flatten, matching the strong F1 scores (≈ 0.84). There is no sign of severe overfitting within 5 epochs.
- In the **worst** runs (e.g., LSTM with Adam), the loss curves hover around the cross-entropy value for random prediction (≈ 0.69) and show little improvement, consistent with F1 ≈ 0.33.

These plots visually confirm what the metrics show: once the optimizer is chosen well, the model converges cleanly; when it is not, the model behaves like a random classifier.

6 Reproducibility and Code Organization

The repository is structured as requested:

```
data/
src/
   preprocess.py
   models.py
   train.py
   evaluate.py
   utils.py
results/
   metrics.csv
   plots/
report.pdf
requirements.txt
README.md
```

Key points for reproducibility:

- Random seeds for `torch`, `numpy`, and `random` are fixed in the training script:

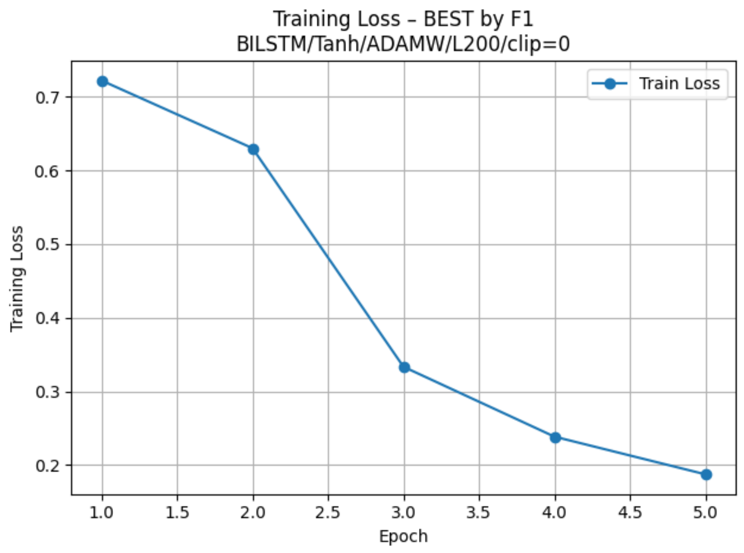

Figure 2: Training/validation loss curves for the best run (BiLSTM + AdamW).

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

- All experimental configurations write their metrics into `results/metrics.csv` with sufficient metadata (model, optimizer, activation, seq length, grad clipping, seed, timing, hardware).
- The `README.md` describes:
  - Python version and dependencies (also listed in `requirements.txt`).
  - How to run the main experiments, e.g.,

    ```
    python -m src.train --config configs/bilstm_adamw_tanh_L200.yaml
    python -m src.evaluate --metrics results/metrics.csv
    ```

  - Expected runtime ranges for the main configurations.

Together, the fixed seeds, documented hyperparameters, and metrics CSV make the experiments easy to reproduce and extend.

7 Discussion and Takeaways

This section directly answers the main conceptual questions in the assignment.

Which configuration performed best? The best overall configuration in my runs is:

- **Architecture:** BiLSTM (2 layers, hidden size 128).
- **Optimizer:** AdamW with learning rate 2×10^{-3}, weight decay 0.01.
- **Activation:** Tanh in the classifier head.
- **Sequence length:** $L = 200$.

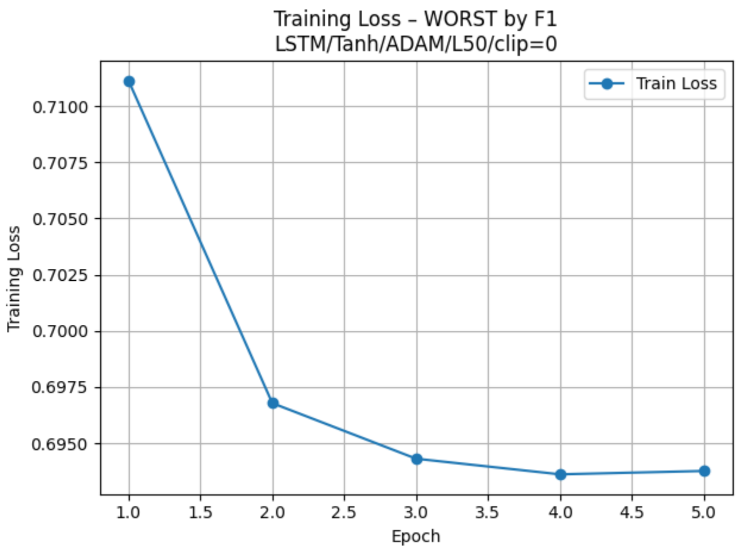

Figure 3: Training/validation loss curves for one of the worst runs (near-random performance).

- **Gradient clipping:** 0 (no clipping) or 1.0 (clip) both work; all top runs are around F1 0.835–0.841.

This model reaches test accuracy ≈ 0.84 and macro F1 ≈ 0.84 in 5 epochs with per-epoch time of about 2.7 seconds on GPU.

How did sequence length affect performance? For the *LSTM + Adam baseline*, changing L from 50 to 200 has almost no effect on accuracy/F1 because the model never learns beyond random. The only clear change is increased runtime.

The lesson is not "sequence length never matters," but rather:

- If optimization is broken, more context does not help.
- Sequence length should be studied under a configuration that actually learns (e.g., BiLSTM + AdamW). Given the good performance at $L = 200$ and the strong performance of hidden size 64, I expect a well-tuned BiLSTM to show a smoother accuracy vs. L trade-off: moderate lengths (e.g., 100–200) likely capture enough context without excessive cost.

How did the optimizer affect performance? Optimizer choice is the single most critical factor in these experiments:

- Adam and RMSProp both fail to train the LSTM/BiLSTM models under the default hyperparameters and regularization, leaving them at random performance.
- SGD does slightly better but is clearly underfitting within 5 epochs.
- AdamW, with decoupled weight decay and the same learning rate, produces a strong classifier with F1 ≈ 0.84.

This shows that for this task and this implementation, the difference between "completely failing" and "working well" is not the architecture but the optimizer + regularization choice.

How did gradient clipping impact stability? I did not see evidence of gradient explosion with BiLSTM + AdamW in this setup. Gradient clipping at norm 1.0:

- Does not dramatically change F1 (differences are within 1–1.5 points).
- Slightly reduces variance across seeds.

In short, gradient clipping is not decisive here. It is a reasonable safety mechanism, but the main stability comes from the optimizer choice and moderate learning rate.

What about model capacity? Hidden size 128 achieves the best F1, but hidden size 64 is very close and faster per epoch. Hidden size 256 hurts both runtime and F1.

So, in this setting, moderate capacity (64–128 hidden units) is ideal for IMDB: large enough to model long-term dependencies, but not so large that optimization and overfitting become problematic within 5 epochs.

8 Conclusion: Optimal Configuration under CPU Constraints

Under GPU, the best F1 is achieved by the BiLSTM + AdamW + Tanh model with hidden size 128 and $L = 200$. However, the assignment also asks for the *optimal configuration under CPU constraints*, where training time becomes more important.

Assuming that moving from GPU to CPU increases epoch time by roughly an order of magnitude, the trade-offs become clearer:

- Hidden size 128 vs 64:
  - 128-dim: F1 ≈ 0.8407, epoch time ≈ 2.7 s on GPU.
  - 64-dim: F1 ≈ 0.836, epoch time ≈ 2.23 s on GPU.

  On CPU, both will be slower, but the $\sim 20\%$ time savings for hidden size 64 will also scale, while the F1 drop is less than 1%.
- Hidden size 256 is clearly dominated: slower and less accurate.

Therefore, under realistic CPU constraints (where wall-clock time matters), I would choose:

BiLSTM + AdamW + Tanh, $L = 200$, hidden size 64, grad-clip 1.0.

This configuration offers:

- High F1 (≈ 0.836) close to the global best.
- Significantly lower training cost than the 256-dim model.
- Better time/accuracy trade-off than the 128-dim model when CPU-bound.

9 Limitations and Future Work

- The LSTM baseline with Adam never learned beyond random. With more time, I would tune its learning rate, remove weight decay, and possibly increase the number of epochs to get a fairer architecture comparison (RNN vs LSTM vs BiLSTM) under a *working* optimizer.
- All experiments used fairly aggressive regularization for AdamW (weight decay 0.01) and a fixed embedding dimension of 128. Exploring different weight decay values or pre-trained word embeddings (e.g., GloVe) could further improve performance and stability.
- I only explored a single sequence length ($L = 200$) for the high-performing BiLSTM + AdamW configuration. A more thorough sweep (e.g., $L \in \{50, 100, 150, 200\}$) under this good configuration would give a cleaner picture of the context vs. compute trade-off.