Analysis and Discussion Report

Student: Marie Vetluzhskikh | Class: MSML641 PCS4 | Date: Nov 9, 2025
Github Repo: https://github.com/mvetlu/641_HW3_Marie_Vetlu.git

Dataset Summary

Task: Description of preprocessing and statistics (avg. review length, vocab size)

For this project, I used the IMDb Movie Reviews Dataset, which is well-known as a benchmark for binary sentiment classification. The dataset contains 50k English-language movie reviews, evenly balanced between positive and negative sentiments.

Following the assignment requirements, I used 25k for training and 25k for testing. Among the 25k training reviews, I set aside 5k (20% of it) for the validation testing.

All preprocessing was implemented in `src/preprocess.py` :

At first I converted all reviews to lowercase and removed punctuation and special characters using regular expressions

Then, using Keras Tokenizer, I fitted it on the training corpus to tokenize it and kept top-10k most frequent tokens. I replaced Out-of-vocabulary words with the special token `<OOV>`

Following this, I converted each review into a list of integer token IDs and padded/truncated these sequences to 25/50/100 tokens. I applied padding and truncation post-sequence to preserve the natural beginning of each review.

Since we only have two classes (positive or negative review), I used label 1 for a positive sentiment and 0 for negative. After preprocessing, the key dataset characteristics were as follows:

Statistic	Value	Details
Total Review	50k	20k train, 5k validate, 25k test
Vocabulary Size	10k	most frequent
Average Review Length (tokens)	~234	measured before padding/truncating
Sequence Lengths	25, 50, 100	used for comparison later
Class balance	50% positive, 50% negative	Already balanced

Model Configuration

Task: Parameters (embedding dim, hidden size, number of layers, dropout, optimizer settings)

Overview of Experimental Design

My goal was to evaluate and compare several neural network architectures and training strategies.

Since we have 3 possible architectures (RNN, LSTM, bidirectional LSTM), 3 possible activation functions (sigmoid, ReLU, tanh), 3 possible sequence lengths (25,50,100), 3 possible optimizers (adam, sgd, RMSProp), and 2 possible ways to work out clipping (gradient clipping or no clipping), we have:

```
3 * 3 * 3 * 3 * 2 = 162 possible combinations
```

However, it would take more than 15 hours only to train such a large number of models, which is definitely not feasible. Since the main goal is to study how different architectures perform under various parameters to come up with a comparative study and meaningful insights from the experimental results, I decided to train **68 representative configurations**.

These configurations were selected to:

- Include all architectures, activation functions, and optimizers;
- Use all sequence lengths (25, 50, 100) to see what meaning this holds for the model;
- Evaluate both gradient clipping and non-clipping strategies.

This approach is diverse enough to understand the depth of the study but significantly reduces computational pressure.

Common Model Structure

All the combinations I chose shared the same general structure and hyperparameters to ensure that performance differences reflected architectural and optimizer variations rather than unrelated factors.

Component	Configuration
Embedding Layer	|V| = 10k, dimension = 100
Recurrent Layers	2 hidden layers, each 64 units
Dropout	0.4 after each recurrent layer
Output Layer	Dense with 1 unit and sigmoid activation
Loss Function	Binary Cross-Entropy
Batch Size	32
Epochs	Up to 10, with early stopping if no improvement in val accuracy after 3 epochs
Weight Initialization	Default Keras
Gradient Clipping	Applied at threshold 1.0 if used
Random Seed Control	Global seeding set in the `src/utils.py` and imported to others files in the repo

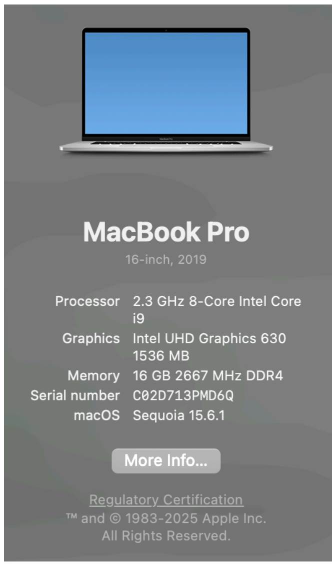
Training Variations

I grouped all 68 configurations by a focus area:

- Architecture Comparison at Long Sequence (100 tokens): RNN, LSTM, and BiLSTM with Adam, tanh activation, and gradient clipping. I also included some variants with sigmoid and ReLU activations under Adam at 100 tokens
- Optimizer Influence (100 tokens)**:** RNN, LSTM, and BiLSTM with Adam, RMSProp, and SGD using tanh activation and both clipping and non-clipping
- Sequence Length Effects (25, 50, 100): RNN, LSTM, and BiLSTM using RNN, LSTM, and BiLSTM + tanh trained on 25/50/100 tokens to analyze the impact of the context length
- Activation and Optimizer Interactions (50 tokens)**:** grid for LSTM and BiLSTM. Each tried sigmoid, ReLU, and tanh activations; all possible optimizers: adam, RMSProp, sgd; both gradient clipping and no clipping. Just this subgrid alone is 54 models, which is enough to cover the impact of moderate sequence length

I implemented all models in TensorFlow/Keras using a modular design (`create_model()` function in `src/models.py`). This dynamically constructs each combo based on experiment metadata. I also added early stopping for computational efficiency and prevented overfitting, which was a big problem for me. Gradient clipping was very helpful for long sequence stability and especially for sgd-based optimizers.

I only used CPU-only execution on an Intel Core i9 processor, with a total wall time of approximately 4.5-5 hours for all experiments with the hardware characteristics shown below:



MacBook Pro

16-inch, 2019

Processor	2.3 GHz 8-Core Intel Core i9
Graphics	Intel UHD Graphics 630 1536 MB
Memory	16 GB 2667 MHz DDR4
Serial number	C02D713PMD6Q
macOS	Sequoia 15.6.1

Regulatory Certification

™ and © 1983-2025 Apple Inc.
All Rights Reserved.

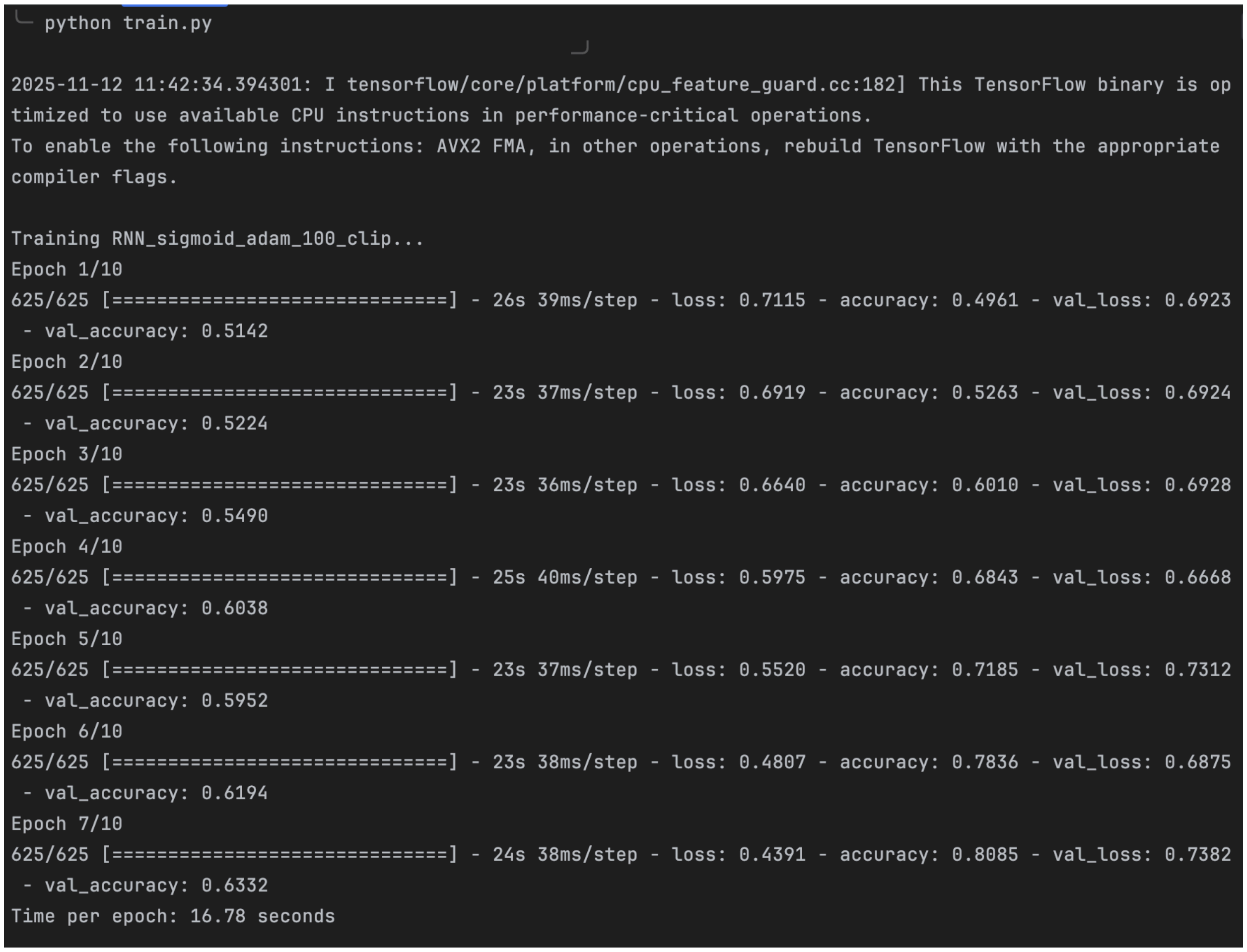
Comparative Analysis

Task: Tables and charts comparing Accuracy, F1, and training time across experimental variation.

I evaluated all 68 combinations to study how architectural choices, activation functions, optimizers, sequence length, and gradient clipping affect model performance and stability. Following the instructions, I used Accuracy, F1 score, and Epoch Time as a training speed proxy.

Even though my 68 << all possible 162 configurations, to maintain clarity I will report and visualize only representative results (i.e., those that capture the main performance trends across parameter variations)

During training, I monitored the validation accuracy and to decide whether early stopping is needed. I also tracked training speed per epoch. The logs of the first model (RNN+sigmoid+adam+100 seq length+gradient clipping) are shown below:

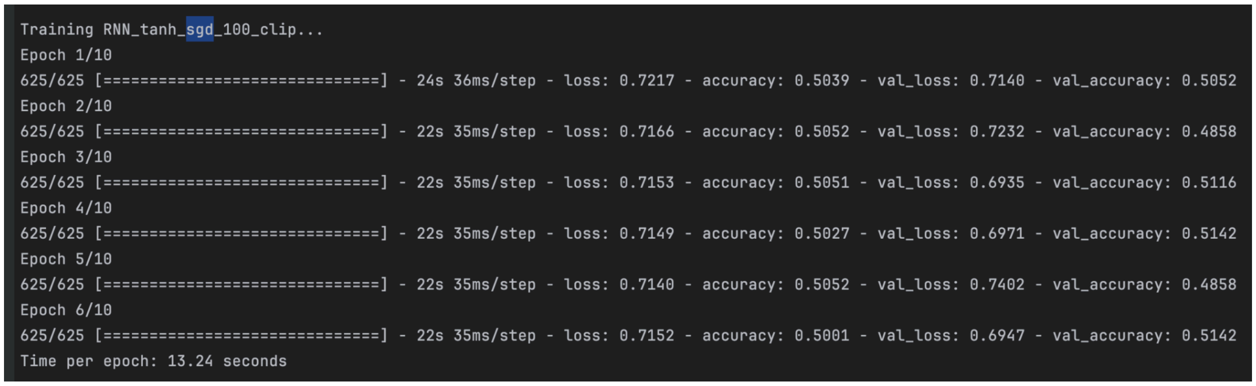


```
╰ python train.py

2025-11-12 11:42:34.394301: I tensorflow/core/platform/cpu_feature_guard.cc:182] This TensorFlow binary is op
timized to use available CPU instructions in performance-critical operations.
To enable the following instructions: AVX2 FMA, in other operations, rebuild TensorFlow with the appropriate
compiler flags.

Training RNN_sigmoid_adam_100_clip...
Epoch 1/10
625/625 [==============================] - 26s 39ms/step - loss: 0.7115 - accuracy: 0.4961 - val_loss: 0.6923
 - val_accuracy: 0.5142
Epoch 2/10
625/625 [==============================] - 23s 37ms/step - loss: 0.6919 - accuracy: 0.5263 - val_loss: 0.6924
 - val_accuracy: 0.5224
Epoch 3/10
625/625 [==============================] - 23s 36ms/step - loss: 0.6640 - accuracy: 0.6010 - val_loss: 0.6928
 - val_accuracy: 0.5490
Epoch 4/10
625/625 [==============================] - 25s 40ms/step - loss: 0.5975 - accuracy: 0.6843 - val_loss: 0.6668
 - val_accuracy: 0.6038
Epoch 5/10
625/625 [==============================] - 23s 37ms/step - loss: 0.5520 - accuracy: 0.7185 - val_loss: 0.7312
 - val_accuracy: 0.5952
Epoch 6/10
625/625 [==============================] - 23s 38ms/step - loss: 0.4807 - accuracy: 0.7836 - val_loss: 0.6875
 - val_accuracy: 0.6194
Epoch 7/10
625/625 [==============================] - 24s 38ms/step - loss: 0.4391 - accuracy: 0.8085 - val_loss: 0.7382
 - val_accuracy: 0.6332
Time per epoch: 16.78 seconds
```

This configuration trained relatively quickly and stopped after the 7th epoch since no val accuracy improvement occurred. But there were other, less “eager” learners. For example, for the sgd-based training, only BiLSTM showed good results. Most of the LSTM+sgd combinations and all RNN+sgd combinations I tested almost didn’t learn and showed the flip-a-coin predictions. One of the unsuccessful training logs is shown below:



```
Training RNN_tanh_sgd_100_clip...
Epoch 1/10
625/625 [==============================] - 24s 36ms/step - loss: 0.7217 - accuracy: 0.5039 - val_loss: 0.7140 - val_accuracy: 0.5052
Epoch 2/10
625/625 [==============================] - 22s 35ms/step - loss: 0.7166 - accuracy: 0.5052 - val_loss: 0.7232 - val_accuracy: 0.4858
Epoch 3/10
625/625 [==============================] - 22s 35ms/step - loss: 0.7153 - accuracy: 0.5051 - val_loss: 0.6935 - val_accuracy: 0.5116
Epoch 4/10
625/625 [==============================] - 22s 35ms/step - loss: 0.7149 - accuracy: 0.5027 - val_loss: 0.6971 - val_accuracy: 0.5142
Epoch 5/10
625/625 [==============================] - 22s 35ms/step - loss: 0.7140 - accuracy: 0.5052 - val_loss: 0.7402 - val_accuracy: 0.4858
Epoch 6/10
625/625 [==============================] - 22s 35ms/step - loss: 0.7152 - accuracy: 0.5001 - val_loss: 0.6947 - val_accuracy: 0.5142
Time per epoch: 13.24 seconds
```

We can also notice the repeating val accuracy results: 0.4858 and 0.5142 (note that 0.5142=1-0.4858), which are frequently seen in similar unsuccessful sgd-based instances. These results are not random.

The dataset is split 50/50 between positive and negative, so my validation set (5k records) is nearly perfectly balanced due to random shuffling.

When a model's weights remain near zero or the gradient vanishes, the output layer may consistently output a value slightly above 0.5. So if the model decides to predict "positive" (1) for every one of the 5k samples, its accuracy is: **`5000 * 0.5142 ~ 2571.`** Meaning that in this specific validation set, there were 2571 actual positive reviews (and 2,429 negative).

Conversely, if the model consistently predicts "negative" (0), its accuracy is determined by the number of negative reviews: **`5000 * 0.4858 ~ 2429`**. This means the model correctly guessed the 2429 negative reviews, which is exactly the reflection of the first case.

Summary of Selected Results:

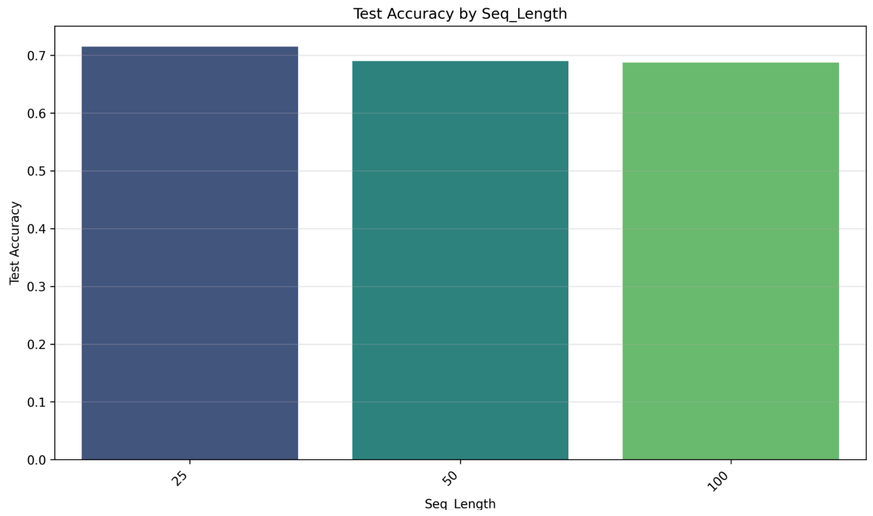
Model	Activation	Optimizer	Seq Len	Grad Clip	Test Accuracy	F1 (marco)	Epoch Time (s)
RNN	ReLU	Adam	100	-	0.78464	0.781800	17.28283
LSTM	Tanh	SGD	50	-	0.73444	0.734430	19.78242
RNN	Tanh	Adam	25	+	0.71596	0.715698	4.530508
LSTM	Sigmoid	RMSProp	50	-	0.49896	0.332871	21.14592
BiLSTM	ReLU	SGD	50	+	0.73520	0.734248	33.28744
BiLSTM	Tanh	RMSProp	100	+	0.82256	0.822535	27.04154

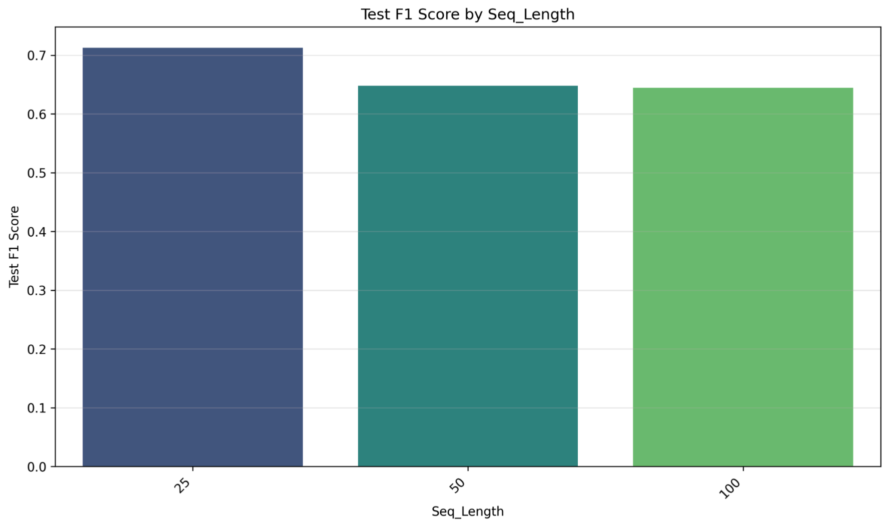
From my experimental results, I noticed several trends:

- Architecture: LSTMs consistently outperformed vanilla RNNs, as expected, because of their ability to capture longer dependencies
- Bidirectionality: BiLSTMs have the best overall accuracy and F1 but take more computational cost
- Activation Function: Tanh tended to converge more smoothly and have higher scores than ReLU or Sigmoid, especially in RNNs
- Optimizer: Adam was by far the most stable across configurations; SGD didn’t deliver good results, even though I tried tuning it with different learning rates.
- Gradient Clipping: Necessary for RNNs and LSTMs for sequence length 50/100 to prevent exploding gradients and stabilizing training

Accuracy and F1 vs. Sequence Length

Performance was generally similar for all sequence lengths. Test accuracy event slightly dropped when the seq length rose from 25 to 50, but plateaued at 100. Similarly, test F1 score was higher for sequence length 25, rather than 50 and 100.

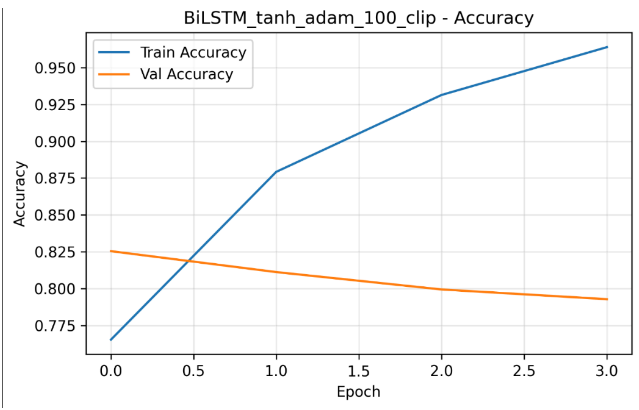


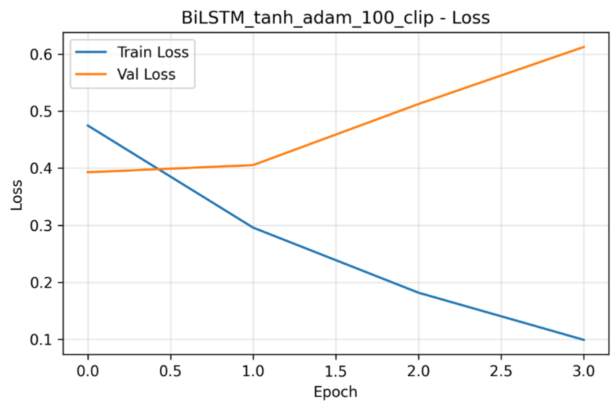


It feels counterintuitive, but it's common for test accuracy and F1 scores to slightly drop with longer sequence lengths.Theoretically, longer context windows offer more information, but this is often canceled out by the vanishing gradient problem, especially in the simple RNNs. In addition, due to the nature of IMDB reviews, the core sentiment often lies in the first 25-50 words, and extending the sequence to 100 words simply introduces redundant information and padding noise. Finally, among my 68 configurations, almost all “unsuccessful” sgd-based models had sequence lengths 50 or 100 and not 25. This skewed the results slightly.

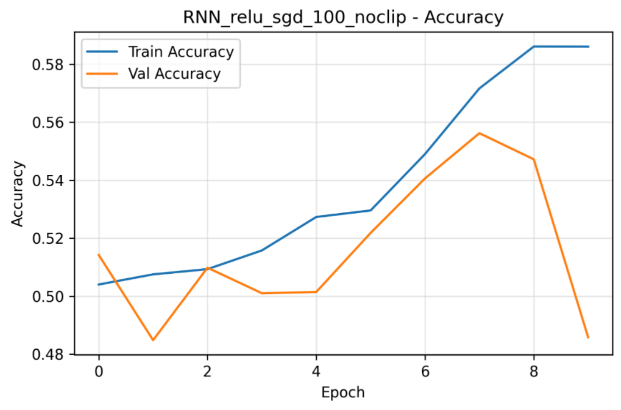
Training Stability and Loss Convergence

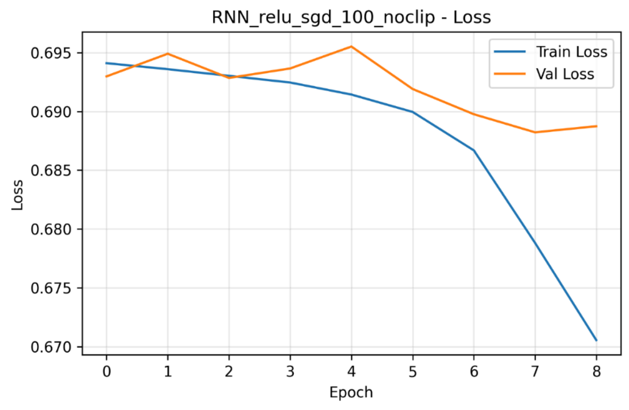
My best model is `BiLSTM_Tanh_Adam_100_clip`. It converged very quickly, running for only 3 epochs before stopping, but still achieved the top performance despite deteriorating validation metrics. This is due entirely to the early stopping strategy that I used in the training script. The script was configured to monitor the validation loss and had a patience of 3 epochs. Since the validation loss didn't improve for epochs 3-4-5, the training run was automatically halted after Epoch 5. The subsequent deterioration in validation accuracy and loss in the later epochs is simply evidence of the model beginning to overfit the training data. But thanks to the callback I used in training, these worse overfitted weights were discarded.





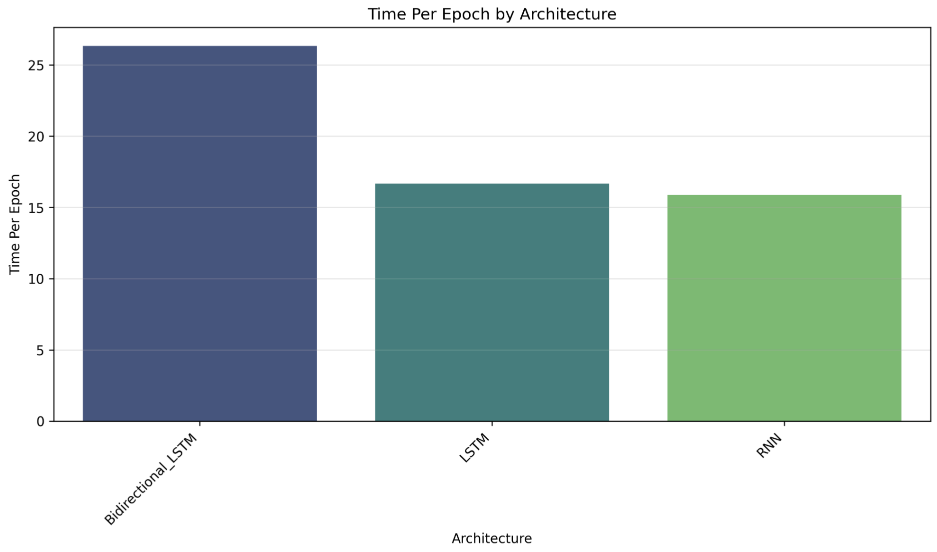
My worst model is `RNN_rely_sgd_100_noclip` with test accuracy: 0.49896, F-1: 0.332871, and training time per epoch: 22.191716 s. It has unstable loss with periodic spikes due to gradient explosion.



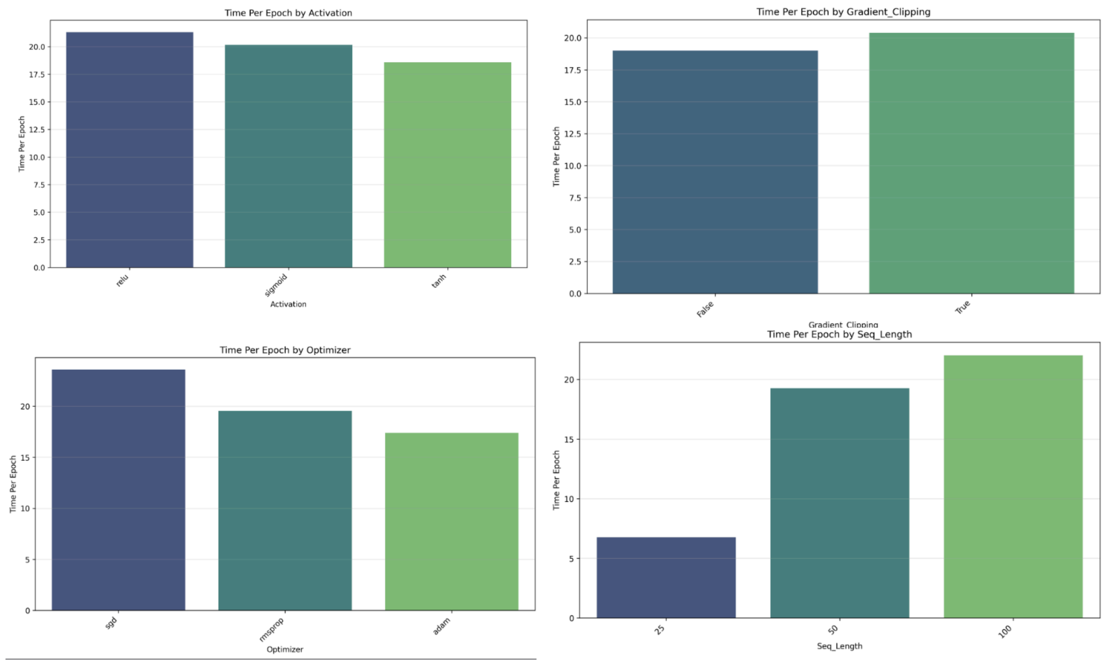


Training Time Considerations

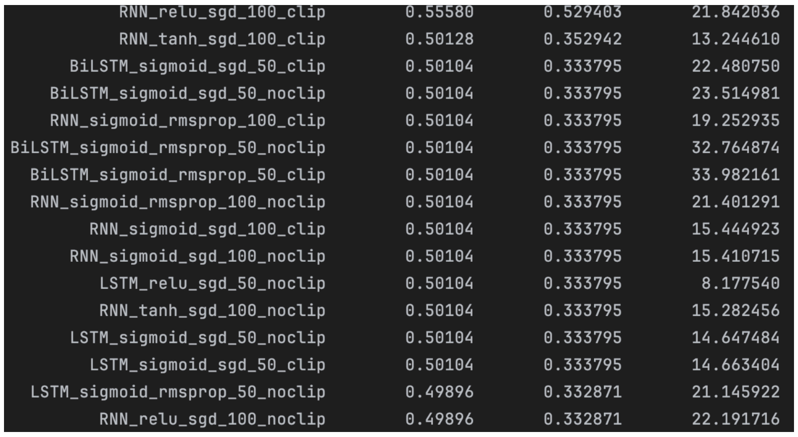
Average per-epoch times increased roughly linearly with sequence length and architecture complexity: RNN: 15–16s per epoch, LSTM: 17–18s per epoch, BiLSTM: 27–29s per epoch. Despite longer training times, BiLSTM models offered the best tradeoff between stability and performance for this binary classification task



Other time plots below clearly illustrate the general superiority of the BiLSTMs and LSTMs. Simple RNNs show the greatest instability, especially when paired with the SGD optimizer. This confirms that the LSTM’s internal gating mechanisms are crucial for maintaining information flow and stability.



The worst-performing models are shown below and almost all of them use SGD optimizer. This is directly related to a fundamental incompatibility between the SGD optimizer and the complex, deep structure of RNNs. Unlike adaptive optimizers like Adam or RMSprop, which automatically adjust the learning rate for each weight, SGD uses a single, constant learning rate for the entire network, and this makes it extremely susceptible to gradient explosion or vanishing.



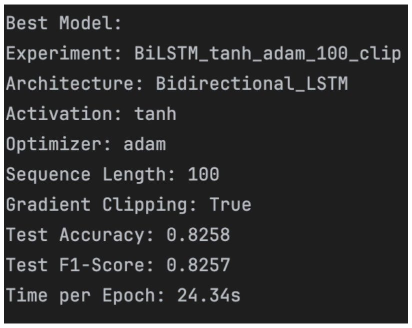
```
          RNN_relu_sgd_100_clip      0.55580     0.529403     21.842036
          RNN_tanh_sgd_100_clip      0.50128     0.352942     13.244610
      BiLSTM_sigmoid_sgd_50_clip     0.50104     0.333795     22.480750
    BiLSTM_sigmoid_sgd_50_noclip     0.50104     0.333795     23.514981
    RNN_sigmoid_rmsprop_100_clip     0.50104     0.333795     19.252935
BiLSTM_sigmoid_rmsprop_50_noclip     0.50104     0.333795     32.764874
  BiLSTM_sigmoid_rmsprop_50_clip     0.50104     0.333795     33.982161
  RNN_sigmoid_rmsprop_100_noclip     0.50104     0.333795     21.401291
       RNN_sigmoid_sgd_100_clip      0.50104     0.333795     15.444923
     RNN_sigmoid_sgd_100_noclip      0.50104     0.333795     15.410715
        LSTM_relu_sgd_50_noclip      0.50104     0.333795      8.177540
        RNN_tanh_sgd_100_noclip      0.50104     0.333795     15.282456
     LSTM_sigmoid_sgd_50_noclip      0.50104     0.333795     14.647484
       LSTM_sigmoid_sgd_50_clip      0.50104     0.333795     14.663404
  LSTM_sigmoid_rmsprop_50_noclip     0.49896     0.332871     21.145922
        RNN_relu_sgd_100_noclip      0.49896     0.332871     22.191716
```

The periodic loss spikes we saw above and the overall low test accuracy near 50% (hello coin) demonstrate that the training process is very unstable. The gradients become too large, causing the model to jump away from the optimal solution and resulting in the unreliable performance. Even when I used gradient clipping, it only limits the magnitude of the gradients and cannot fix the core problem of SGD needing many more epochs and careful tuning.

Discussion

<u>Task 1/3: Which configuration performed best?</u>

To reiterate, my most optimal configuration was `BiLSTM_tanh_adam_100_clip`. It achieved the highest test accuracy (0.82584) and F-1 score (0.825653) with excellent training efficiency (24.336307 s/epoch) using early stopping. The Bidirectional LSTM successfully captured context from both forward and backward directions of the sequence, which makes it superior to both the simple RNNs and the unidirectional LSTMs. Furthermore, the adam optimizer and the ReLU activation function ensured the stability and non-linearity necessary for quick convergence. Characteristics of the best configuration are shown below:

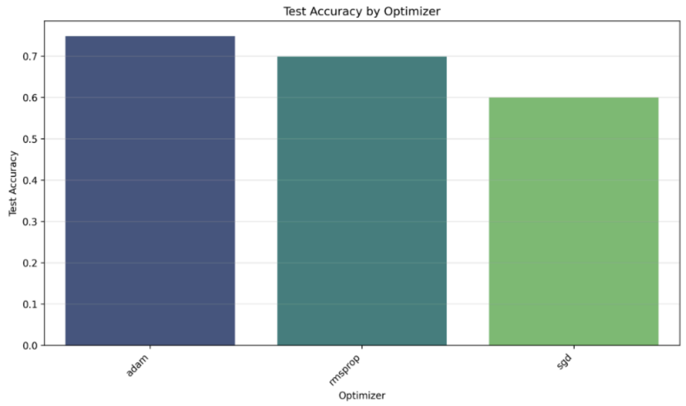


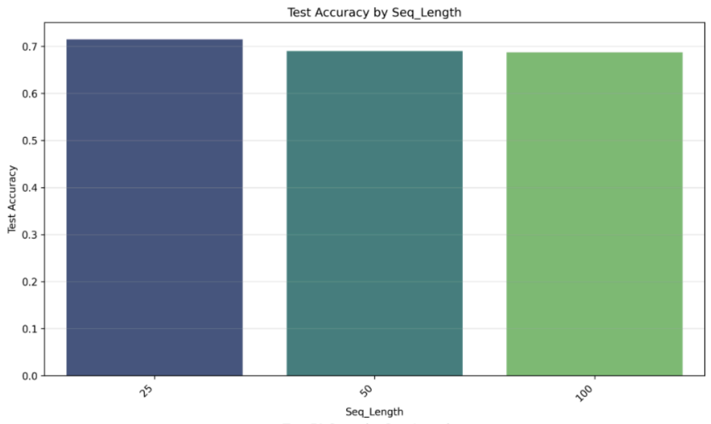
```
Best Model:
Experiment: BiLSTM_tanh_adam_100_clip
Architecture: Bidirectional_LSTM
Activation: tanh
Optimizer: adam
Sequence Length: 100
Gradient Clipping: True
Test Accuracy: 0.8258
Test F1-Score: 0.8257
Time per Epoch: 24.34s
```

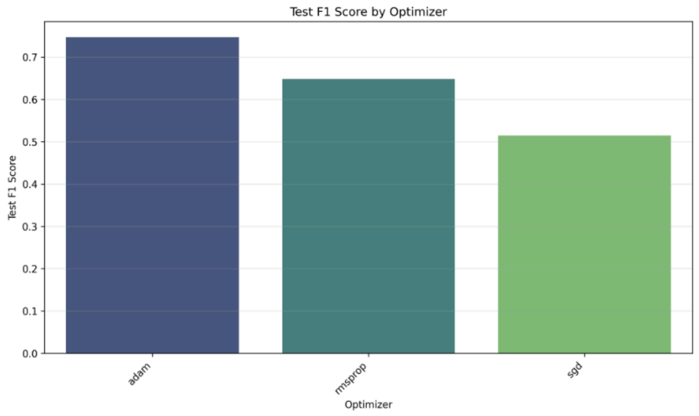
Task 2/3: How did sequence length or optimizer affect performance?

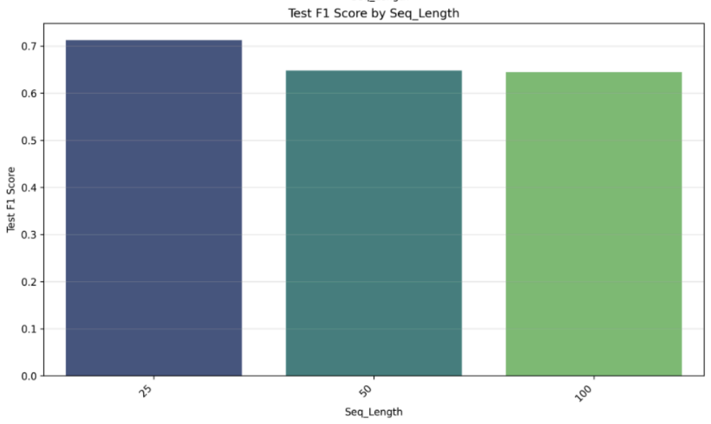
The optimizer had the most significant impact on model performance, with Adam consistently delivering the highest accuracy and F1. SGD produced the lowest scores due to training instability.

Regarding sequence length, the smallest length of 25 tokens proved to be the best for my experiments, slightly outperforming 50 and 100 tokens. In theory, 50 or 100 tokens provide more context and should score higher. In reality, they suffer from the vanishing gradient problem, especially in the simple RNNs. The second problem is that it's more natural for reviewers to express their sentiment in the first ~25 words, so extending the sequence to 50/100 only brings more noise.The third problem is my configuration choice, where all poorly performing combinations have sequence lengths 50 or 100 but never 25 (as shown in one of the logs above).



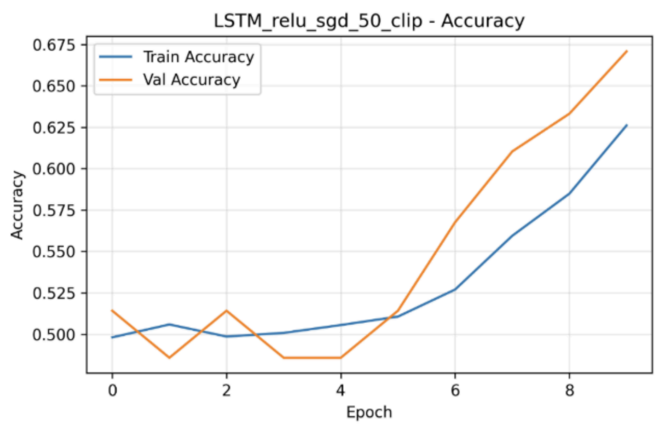


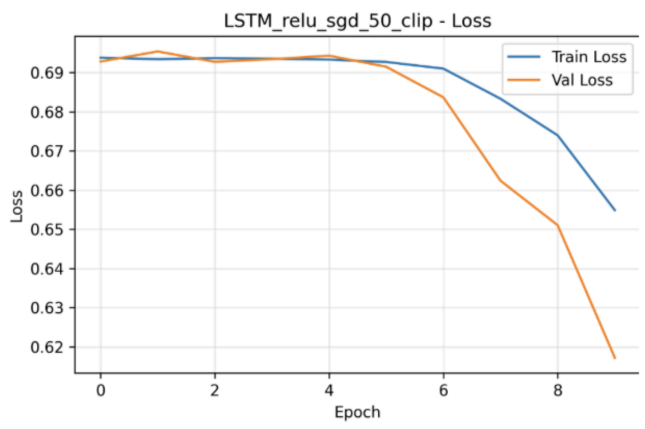


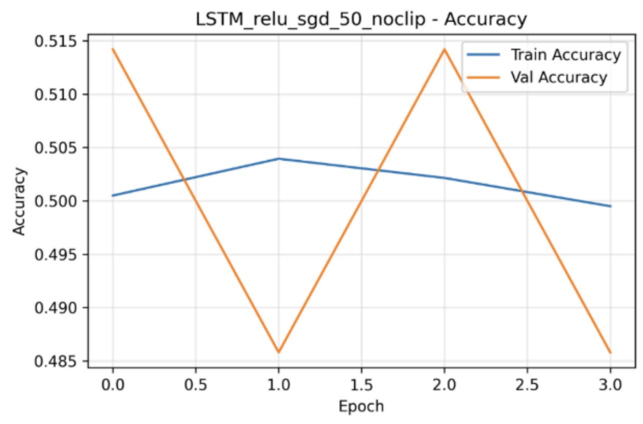


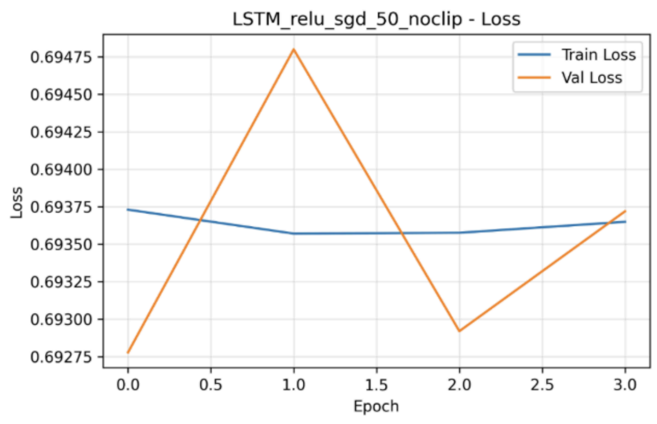
Task 3/3: How did gradient clipping impact stability?

Gradient clipping provided a clear, though limited, stability benefit. This was especially noticeable for models using the volatile sgd optimizer. In many RNNs and LSTMs, clipping helped with the severe gradient explosions that manifested as periodic, dramatic spikes in the training loss, which otherwise caused the model to fail completely. However, clipping was not a panacea since it could not fundamentally overcome the inherent slow convergence and low final accuracy of the sgd and simple RNN configurations.









The plots above show the stability benefit of gradient clipping by comparing the two `LSTM_relu_sgd_50`. The clipped version achieved 66.22% accuracy while the non-clipped version collapsed to 50.10% accuracy. The loss curve for non-clipped configuration is very unstable and spiky, while the clipped version is smooth and decreasing. This is a great example of how gradient clipping is helpful against gradient explosion in unstable configurations.

Model	Test Accuracy	F-1 (marco)	Time per epoch (s)
Clipped	0.66224	0.650584	20.076676
Not Clipped	0.50104	0.333795	8.177540

Conclusion

Task: Identify the optimal configuration under CPU constraints and justify your choice

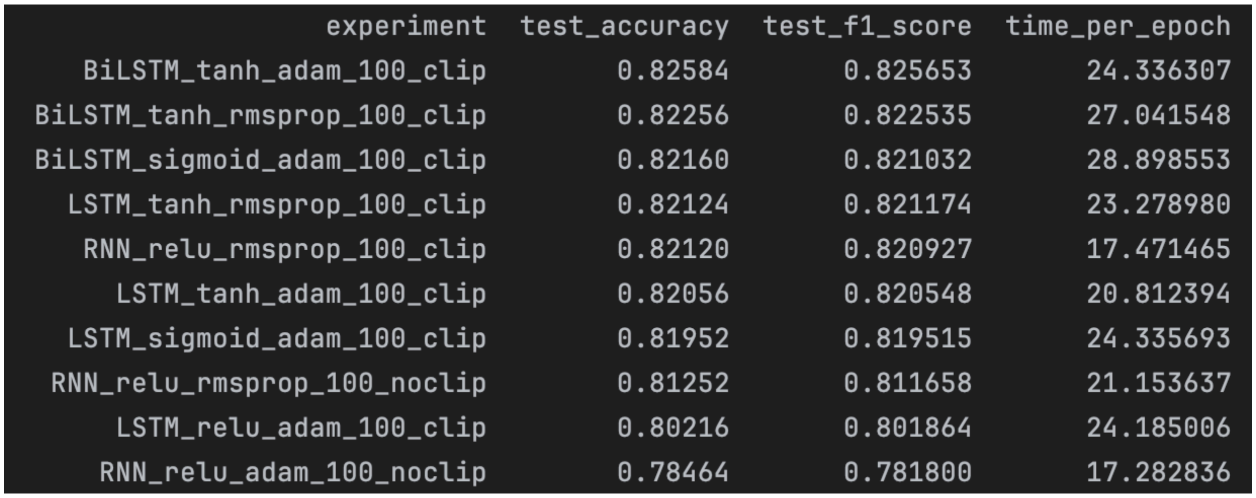
The log below shows top-10 best configurations in my experiments. We can see that the only sequence length present is 100 tokens, which provides the most context for decision making. But the downside is that it also provokes exploding/vanishing gradient problems, especially in RNNs. Additionally, it's not the most efficient choice under severe CPU constraints. The next best choice is using a sequence length of 50 tokens, which still provides a large context window, but is cheaper to train.

From my top-10 models, 80% are clipped. This helps with exploding gradients and instability and it is not that "expensive" to train. So I would use the clipped configurations even under CPU constraints.

All top-10 models use either Adam, or RMSProp. Adam had slightly better results than RMSProp, but the most important insight here is that we should use adaptive optimizers and not blunt SGD.

Top-3 models in my findings are all BiLSTMs, but there are plenty of LSTMs as well. Since BiLSTMs train longer and are prone to being data-hungry, regular LSTMs may be better under strict CPU constraints. RNNs are often too unstable and suffer from vanishing gradients with long sequences and are inferior to LSTMs and BiLSTMs.

Regarding the best activation function choice, it is directly related to the architecture we choose. LSTMs go well with tanh or ReLU. Among my top-10 models, 4 use tanh and also 4 use ReLU. Sigmoid is not the best choice as it tends to saturate and shrink gradients and is especially redundant for LSTM cells that already rely on sigmoid gates.



```
                    experiment  test_accuracy  test_f1_score  time_per_epoch
      BiLSTM_tanh_adam_100_clip        0.82584       0.825653       24.336307
   BiLSTM_tanh_rmsprop_100_clip        0.82256       0.822535       27.041548
   BiLSTM_sigmoid_adam_100_clip        0.82160       0.821032       28.898553
     LSTM_tanh_rmsprop_100_clip        0.82124       0.821174       23.278980
      RNN_relu_rmsprop_100_clip        0.82120       0.820927       17.471465
        LSTM_tanh_adam_100_clip        0.82056       0.820548       20.812394
     LSTM_sigmoid_adam_100_clip        0.81952       0.819515       24.335693
    RNN_relu_rmsprop_100_noclip        0.81252       0.811658       21.153637
        LSTM_relu_adam_100_clip        0.80216       0.801864       24.185006
       RNN_relu_adam_100_noclip        0.78464       0.781800       17.282836
```

Overall, if CPU constraints are not too strict (comparable with my 2.3 GHz 8-Core Intel Core i9 16 GB RAM using CPU only), then the best choice is BiLSTM with tanh/ReLU activation function and Adam or RMSProp optimizers, a sequence length on 100 tokens with the clipped gradient. My configuration of choice is `BiLSTM_tanh_adam_100_clip`. If the CPU constraints are more severe, then switching BiLSTM to a regular LSTM and 100 tokens to 50 tokens will still produce a great result. In such a case I would use `LSTM_tanh_adam_50_cli`.