Artificial Intelligence Final Project Part 1 Report

Ik Jun Lee
Soongsil University
Computer Science and Engineering
ikjoon0812@gmail.com

Abstract

This report investigates the fine-tuning of BERT using Python and PyTorch for sentiment classification on the IMDB dataset. The task involves predicting whether movie reviews express positive or negative sentiment. A `CustomBERTClassifier` was developed, extending the pre-trained `bert-base-uncased` model with a lightweight feed-forward neural network. Experimental trials were conducted using different hyperparameter configurations, with significant improvements achieved by optimizing sequence lengths and training epochs. The best configuration yielded a validation accuracy of 94% after two epochs.

1. Introduction

Sentiment analysis is an essential task in natural language processing (NLP). With the advent of large pre-trained models like BERT, performance on text classification tasks has significantly improved. This project aims to fine-tune BERT for binary sentiment classification on the IMDB dataset.

2. CustomBERTClassifier Model Description

The `CustomBERTClassifier` consists of the following components:

- **BERT Model:** The pre-trained `bert-base-uncased` model.
- **Hidden Layer:** A linear layer maps the 768-dimensional input to a 768-dimensional output.
- **Dropout:** Dropout is applied for prevent overfitting issues. ($p = 0.15$ or $p = 0.20$)
- **Output Layer:** A linear layer maps the hidden layer's output to two classes.

3. Trials

3.1. Trial 1

Table 1. Trial 1 Hyperparameter Settings

Hyperparameter	Value
Max Sequence Length	500
Batch Size	8
Number of Epochs	1
Learning Rate	2×10^{-5}
Weight Decay	0.01
Dropout	0.15

In the first trial, the BERT fine-tuning process was initialized with a baseline set of hyperparameters optimized for stable and efficient training. The hyperparameters were chosen based on prior research that highlights their relative importance in transformer-based models: learning rate, weight decay, batch size, warmup steps, and number of epochs.

The learning rate was set to 2×10^{-5}, a commonly used value for fine-tuning pre-trained BERT models. A weight decay of 0.01 was applied to reduce overfitting by penalizing large weights. The maximum sequence length was set to 500 tokens to accommodate the IMDB dataset's lengthy movie reviews, ensuring the model captures the full contextual information. Due to memory constraints caused by the long sequences, the batch size was limited to 8.

The number of training epochs was set to 1. Although a single epoch might seem insufficient, BERT's pre-trained knowledge and the small learning rate ensured that meaningful updates were made to the model's weights. Dropout with a rate of 0.15 was applied to further prevent overfitting.

Training took approximately 52 minutes, while inference required about 5 minutes. The model achieved a validation accuracy of 0.9411. This strong performance demonstrates BERT's effectiveness, even under minimal fine-tuning. Since the validation accuracy exceeded 94%, detailed metrics (Precision, Recall, and F1-Score) are presented in Table 2. Future trials will omit such tables unless

significant performance changes are observed.

Table 2. Trial 1 Metrics on Validation Set

Metric	Precision	Recall	F1-Score
Positive Class	0.94	0.94	0.94
Negative Class	0.94	0.94	0.94

This configuration served as the baseline to evaluate the performance of BERT on the IMDB sentiment classification task and laid the foundation for further hyperparameter optimization.

3.2. Trial 2

Table 3. Trial 2 Hyperparameter Settings

Hyperparameter	Value
Max Sequence Length	128
Batch Size	8
Number of Epochs	5
Learning Rate	2×10^{-5}
Weight Decay	0.01
Dropout	0.15

After analyzing the results of Trial 1, the training time for a single epoch with a maximum sequence length of 500 was determined to be excessive, even with the use of a CUDA T4 GPU on Colab. To address this issue, the maximum sequence length was reduced to 128 tokens. Although shorter sequences might lose some contextual information, increasing the number of epochs was expected to compensate for this and improve accuracy.

With this setup, training time was significantly reduced compared to Trial 1. However, the results per epoch revealed that the validation accuracy plateaued around the 90% mark after the second epoch. The detailed results are as follows:

- **Epoch 1:** Validation Accuracy = 0.8706
- **Epoch 2:** Validation Accuracy = 0.8941
- **Epoch 3:** Validation Accuracy = 0.8916
- **Epoch 4:** Validation Accuracy = 0.8969
- **Epoch 5:** Validation Accuracy = 0.8966

Although the validation accuracy improved initially, it stabilized after the second epoch, consistently hovering around 89% to 90%. This indicates that reducing the sequence length effectively reduced training time but limited further performance gains.

Based on these findings, it was determined that additional modifications to the hyperparameters or training strategy were necessary. Therefore, Trial 3 was conducted with the goal of addressing these limitations and further improving the model's performance.

3.3. Trial 3

Table 4. Trial 3 Hyperparameter Settings

Hyperparameter	Value
Max Sequence Length	500
Batch Size	8
Number of Epochs	2
Learning Rate	2×10^{-5}
Weight Decay	0.01
Dropout	0.15

To build upon the insights gained from the previous trials, the sequence length was restored to 500 tokens to leverage the full contextual information, while the number of epochs was limited to 2 to mitigate excessive training time. The hyperparameters for Trial 3 were configured as follows:

This configuration aimed to balance the trade-off between training time and model performance. The validation accuracy achieved during the two epochs is as follows:

- **Epoch 1:** Validation Accuracy = 0.9321
- **Epoch 2:** Validation Accuracy = 0.9440

The results demonstrate a significant improvement, with a final validation accuracy of 94.4%. The combination of a longer sequence length and an increased number of epochs allowed the model to better capture the contextual nuances of the input data.

Despite the longer training time caused by the increased sequence length, the results suggest that extending the number of epochs beyond 2 has the potential to further improve accuracy. This insight has led to the design of Trial 4, where additional epochs will be explored to maximize performance.

3.4. Trial 4

Table 5. Trial 4 Hyperparameter Settings

Hyperparameter	Value
Max Sequence Length	500
Batch Size	8
Number of Epochs	5
Learning Rate	2×10^{-5}
Weight Decay	0.01
Dropout	0.15

To further investigate the effect of training for additional epochs, the number of epochs was increased to 5 while

maintaining the same hyperparameter configuration as Trial 3.

The validation accuracy achieved across all epochs is summarized below:

- **Epoch 1:** Validation Accuracy = 0.9387
- **Epoch 2:** Validation Accuracy = 0.9405
- **Epoch 3:** Validation Accuracy = 0.9417
- **Epoch 4:** Validation Accuracy = 0.9419
- **Epoch 5:** Validation Accuracy = 0.9419

The results indicate that while validation accuracy improved slightly up to the third epoch, performance plateaued at around 94.2%, showing minimal gains beyond this point. Notably, the final accuracy achieved in this trial was lower than the 94.4% obtained in Trial 3.

Despite increasing the number of epochs from 2 to 5, the model appeared to hit an early performance ceiling, suggesting that the current configuration has a limited capacity for further improvement.

3.5. Trial 5

Table 6. Trial 5 Hyperparameter Settings

Hyperparameter	Value
Max Sequence Length	500
Batch Size	8
Number of Epochs	5
Learning Rate	3×10^{-5}
Weight Decay	0.02
Dropout	0.2

In Trial 5, I plan to adjust key hyperparameters, including **dropout**, **weight decay**, and **learning rate**. If the resulting model surpasses the 94.4% validation accuracy achieved in Trial 3, it will be selected as the final submission model. Else, finally submit the model with 94.4% accuracy obtained from Trial 3. The validation accuracy achieved during the five epochs is as follows:

- **Epoch 1:** Validation Accuracy = 0.9320
- **Epoch 2:** Validation Accuracy = 0.9390
- **Epoch 3:** Validation Accuracy = 0.9413
- **Epoch 4:** Validation Accuracy = 0.9407
- **Epoch 5:** Validation Accuracy = 0.9410

The validation accuracy in Trial 5 showed a slight improvement over the first few epochs, peaking at 94.13% in the third epoch. However, performance stagnated and even slightly regressed in subsequent epochs. These results closely mirror those of Trial 4, where the validation accuracy plateaued at approximately 94.2%. Despite adjusting key hyperparameters, the performance did not surpass the 94.4% accuracy achieved in Trial 3.

Trial 5 demonstrated that hyperparameter tuning alone is insufficient to break through the observed performance ceiling of approximately 94.4%. The minimal improvement suggests that the current model and data configuration may require additional techniques to achieve substantial gains in accuracy. As a result, the model from Trial 3 remains the best-performing configuration and has been selected as the final submission model.

4. Conclusion

This project aimed to fine-tune a BERT-based model for sentiment classification on the IMDB dataset, focusing on optimizing hyperparameters and designing a custom classifier architecture to achieve the best performance.

The custom architecture, `CustomBERTClassifier`, extended the pre-trained `bert-base-uncased` model with a feed-forward network comprising a hidden linear layer, a dropout layer for regularization, and an output layer for binary classification. Dropout rates of 0.15 and 0.20, along with a weight decay regularizer, were tested to mitigate overfitting and stabilize training.

A series of trials with various hyperparameter configurations produced the following key insights:

- **Trial 1:** A baseline setup (500 sequence length, 1 epoch) achieved 94.1% validation accuracy but lacked sufficient training optimization.
- **Trial 2:** Reducing sequence length to 128 improved efficiency but lowered accuracy to 89.6%.
- **Trial 3:** Restoring sequence length to 500 and training for 2 epochs achieved the highest validation accuracy of 94.4%, emphasizing the importance of contextual information in longer sequences.
- **Trial 4:** Increasing epochs to 5 plateaued accuracy at 94.2%, suggesting diminishing returns from additional training.
- **Trial 5:** Further fine-tuning of dropout, weight decay, and learning rate produced a peak accuracy of 94.13%, similar to Trial 4, underscoring the performance ceiling with the current configuration.

In conclusion, Trial 3 emerged as the most effective setup, achieving a validation accuracy of 94.4%. These

results demonstrate the significance of balancing sequence length, epochs, and regularization in fine-tuning BERT for sentiment analysis.

Future Work: To surpass the observed accuracy ceiling, the following strategies will be explored in future iterations:

- Data Augmentation: Techniques like back-translation, paraphrasing, and synonym replacement to enhance data diversity.
- Model Architecture Modification: Introducing task-specific layers or enhancing the classifier head for better representation learning.
- Automated Hyperparameter Optimization: Utilizing tools such as Optuna or Ray Tune to achieve efficient and optimal hyperparameter tuning.
- Alternative Pre-trained Models: Experimenting with advanced models like RoBERTa or ALBERT for potential performance improvements.

References

[1] M. Vaswani, et al. *Hyperparameter Optimization for Transformers: A Guide*. Medium, 2021. Available: https://medium.com/distributed-computing-with-ray/hyperparameter-optimization-for-transformers-a-guide-c4e32c6c989b.

[2] Haren Lin, et al. *IMDB Sentiment Analysis Using BERT Fine-Tuning*. GitHub Repository, 2021. Available: https://github.com/harenlin/IMDB-Sentiment-Analysis-Using-BERT-Fine-Tuning/blob/main/BERT_Fine_Tune.ipynb.

[3] Wakaka. *BERT Fine-Tuned on IMDB Dataset*. Hugging Face, 2023. Available: https://huggingface.co/Wakaka/bert-finetuned-imdb.

Artificial Intelligence Final Project Part 2 Report

Ik Jun Lee
Soongsil University
Computer Science and Engineering
ikjoon0812@gmail.com

Abstract

This report provides a detailed analysis and implementation of various algorithms to solve Markov Decision Processes (MDPs). explore Value Iteration, Asynchronous Value Iteration, Prioritized Sweeping Value Iteration, and Q-Learning. Using provided problem specifications, I implemented solutions in Python and verified their correctness through autograder. Each section details the methodology, parameter selection, and results.

1. Value Iteration(Q1)

Objective: Compute the optimal value function for an MDP using the Value Iteration algorithm.

Methodology:

1. **Initialization:** Initialize the value of all states to zero.
2. **Update Rule:** Perform k iterations, updating each state's value based on the Bellman equation:

$$V(s) = \max_a \sum_{s'} P(s'|s,a)\big[R(s,a,s') + \gamma V(s')\big], \tag{1}$$

   where $P(s'|s,a)$ is the transition probability, $R(s,a,s')$ is the reward, and γ is the discount factor.
3. **Batch Update:** Ensure that the updates use values from the previous iteration (batch update style).

Implementation Details:

- `runValueIteration`: Iteratively computes values for all states.
- `computeQValueFromValues`: Calculates the Q-value for a given state-action pair by summing over all possible next states.
- `computeActionFromValues`: Determines the optimal action by selecting the action with the highest Q-value.

Results: The implementation was validated using the autograder, successfully producing optimal values and policies for various grid configurations. The approach ensures convergence to the correct solution.

2. Bridge Crossing Analysis(Q2)

Objective: Modify parameters such that the agent crosses a narrow bridge in the grid world.

Problem Analysis: The BridgeGrid layout has a high-reward terminal state across a narrow bridge and a high negative-reward chasm. The default settings prevent the agent from crossing due to the risk of unintended transitions caused by noise.

Solution:

- Reduce the `noise` parameter to 0.002, ensuring minimal unintended transitions.
- Keep the `discount` at 0.9, encouraging the agent to aim for long-term rewards.

Implementation Details:

- Adjusted the parameters in `question2()` of `analysis.py`.

Results: With reduced noise, the agent reliably crossed the bridge, achieving the desired behavior as confirmed by the autograder.

3. Policies (Q3)

Objective: Adjust parameters to produce distinct optimal policies in the DiscountGrid layout.

Methodology:

- The DiscountGrid has two positive-reward exits (+1 and +10) and a cliff region (-10). The agent's behavior is controlled by the `discount`, `noise`, and `livingReward` parameters.
- Different combinations of these parameters were tested to achieve specific behaviors.

Parameter Configurations:

- **Q3a:** Prefer the close exit (+1), risking the cliff.
  - `discount` = 0.5, `noise` = 0, `livingReward` = -1
- **Q3b:** Prefer the close exit (+1), avoiding the cliff.
  - `discount` = 0.2, `noise` = 0.2, `livingReward` = -1
- **Q3c:** Prefer the distant exit (+10), risking the cliff.
  - `discount` = 0.95, `noise` = 0, `livingReward` = -1
- **Q3d:** Prefer the distant exit (+10), avoiding the cliff.
  - `discount` = 0.2, `noise` = 0.5, `livingReward` = 2
- **Q3e:** Avoid both exits and the cliff.
  - `discount` = 0.95, `noise` = 0, `livingReward` = 1000

Results: Each behavior was successfully achieved, with parameter configurations validated by the autograder.

4. Asynchronous Value Iteration (Q4)

Objective: Implement an agent that performs asynchronous updates for value iteration.

Methodology:

1. Update the value of one state per iteration, cycling through all states in order.
2. Skip updates for terminal states.

Implementation Details:

- Implemented in `runValueIteration()` of `AsynchronousValueIterationAgent`.
- Leveraged state indexing to ensure cyclic updates.

Results: The implementation passed all test cases, demonstrating correct and efficient convergence.

5. Prioritized Sweeping Value Iteration (Q5)

Objective: Focus updates on states most likely to change the policy using prioritized sweeping.

Methodology:

1. Compute predecessors for all states.
2. Use a priority queue to order state updates by the magnitude of value change.
3. Iteratively update values, adjusting priorities for predecessor states.

Implementation Details:

- Predecessors were stored as sets to avoid duplicates.
- Used `util.PriorityQueue` for efficient priority management.

Results: The algorithm converged efficiently, producing optimal policies as validated by the autograder.

6. Q-Learning (Q6)

Objective: Implement a model-free reinforcement learning agent using Q-learning.

Methodology:

- Use an ϵ-greedy policy for exploration.
- Update Q-values using the Bellman equation:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha\big[R + \gamma \max_{a'} Q(s',a')\big]. \tag{2}$$

- Handle unseen actions with a default Q-value of zero.

Implementation Details:

- Implemented Q-value updates in the `update` method of `QLearningAgent`.
- Ensured exploration by breaking ties randomly in `computeActionFromQValues`.

Results: The Q-learning agent successfully learned optimal policies, validated through simulation and testing.

Conclusion

This report demonstrates the implementation of key algorithms for solving MDPs and reinforcement learning problems. The results highlight the correctness and efficiency of the solutions in various grid world scenarios.