Bucketing's Effect on Efficiency and Accuracy in KLUE/BERT-Base for NSMC

Kang Heebong
AIFFEL Research 13
kmasta.study@gmail.com

June 4, 2025

Abstract

Pretrained Transformer models such as BERT have achieved state-of-the-art performance on sentiment analysis tasks, but fine-tuning these large models remains computationally expensive due to redundant operations introduced by fixed-length padding. In this work, we evaluate two complementary strategies—(1) training-time bucketing (length-aware batching) and (2) inference-time length sorting—on `klue/bert-base` fine-tuning for the Korean NSMC dataset. Our experiments show that using length-aware bucketing during training reduces floating-point operations (FLOPs) by 80% and shortens training time by 60%, with only a 0.12% drop in validation accuracy. Furthermore, sorting test samples by token length before inference yields a 76.5% reduction in evaluation time, while maintaining test accuracy and F1. These optimizations are particularly effective for NSMC, where most reviews are short (average $\approx$22 tokens, max 120 in training and 122 in test). All code and results are publicly available for reproducibility[1].

1 Introduction

Pretrained Transformer models, particularly BERT [1], have demonstrated state-of-the-art performance on sentiment analysis and various other NLP tasks. However, fine-tuning these large models is computationally expensive: padding variable-length inputs to a fixed size incurs substantial redundant FLOPs, increases memory usage, and prolongs wall-clock time.

When fine-tuning `klue/bert-base` on NSMC [2], setting `max_length=120` forces even short reviews to include many padding tokens. In fact, most reviews are under 30 tokens (avg 22), so fixed padding wastes significant FLOPs (Figure 1).

In this paper, we propose and evaluate two complementary strategies for `klue/bert-base` fine-tuning on NSMC:

[1] `https://github.com/kmasta/AIFFEL_quest_rs/blob/master/GoingDeeper/Gd08/nsmc_with_klue_bert_base.ipynb`

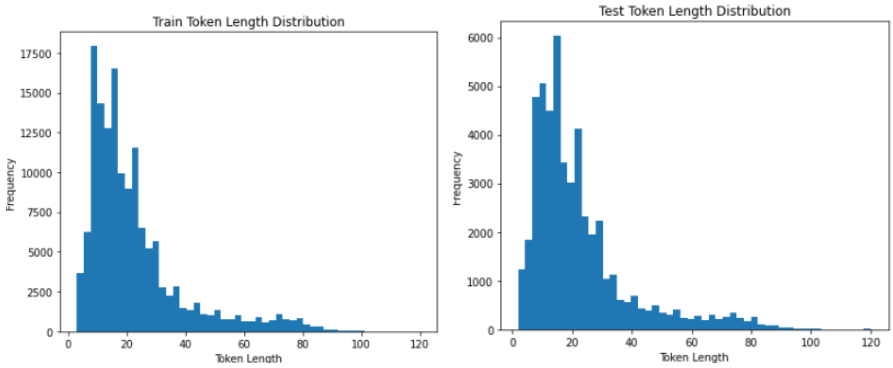

Figure 1: Training (left) and test (right) token length distributions. Most reviews are under 30 tokens, with mean ≈22.7.

- **Training-Time Bucketing:** Set `group_by_length=True` in `TrainingArguments` to form length-homogeneous batches, using `DataCollatorWithPadding` so that each batch is only padded to its longest sequence.
- **Inference-Time Length Sorting:** Sort test samples by token length before batching; use `DataCollatorWithPadding` to dynamically pad each sorted batch to its maximum length.

We compare these strategies against a fixed-padding baseline (`batch_size=128`, `max_length=120`). Our experiments show up to 80% reduction in FLOPs and 60% reduction in training time, with only a 0.12% drop in validation accuracy. Inference-time length sorting yields a 76.5% reduction in evaluation time, while preserving test accuracy and F1. These optimizations are particularly beneficial for NSMC, where short reviews dominate.

The remainder of the paper is organized as follows. Section 2 reviews related work on padding optimization and sequence packing. Section 3 details our dataset preprocessing, model configuration, and batching strategies. Section 4 presents experimental results and analysis. Section 5 discusses implications, limitations, and future work. Finally, Section 6 concludes the paper.

2 Related Work

Reducing the computational burden of pretrained Transformers has attracted increasing attention, with methods targeting both training and inference efficiency. We categorize related work into two main areas: (1) padding removal and length-aware batching, and (2) sequence packing.

2.1 Padding Removal and Length-Aware Batching

Zhang et al. [3] show that curriculum learning can progressively remove padding tokens and speed up BERT training without accuracy loss. Nagarajan and Raghunathan [4] further combine bucketing with token-level regularization (TokenDrop), achieving over 10× throughput gains on various English tasks.

2.2 Sequence Packing

Kosec et al. [5] formulate batch construction as bin-packing to eliminate padding entirely, yielding nearly 2× throughput gains in long-sequence pretraining. Although their work targets pretraining, the same principle motivates our length-sorting approach during inference.

3 Methodology

This section describes our approach for fine-tuning `klue/bert-base` on NSMC, using three distinct batching strategies: fixed padding (baseline), length-aware bucketing, and length-sorted inference. We detail dataset preprocessing, model/tokenizer configuration, and each pipeline's design without discussing empirical results.

3.1 Dataset Preprocessing

- **Dataset:** Naver Sentiment Movie Corpus (NSMC) [2].
  - Training: 150,000 labeled Korean movie reviews.
  - Test: 50,000 labeled reviews.
- **Duplicate Removal:** Remove exact-duplicate review texts in the training split:

$$\text{train\_no\_duplicates} = \text{train.select}(\text{get\_unique\_indices}(\text{train})).$$

- **Invalid Sample Filtering:** Discard any training example if its review text is empty or null, or its label is missing:

$$\text{train\_nan} = \text{train\_no\_duplicates.filter}(\text{is\_valid\_sample}).$$

- **Length Filtering:** Tokenize each remaining review without truncation, and exclude any sample whose token count exceeds 120:

$$\text{train} = \text{train\_nan.filter}(\text{is\_valid\_length}).$$

  This ensures that both fixed and dynamic padding pipelines operate on the same subset of training samples (146,176 out of 150,000).
- **Test Split:** The original 50,000 reviews are tokenized only (no filtering), preserving all samples including those up to 122 tokens.

3.2 Model and Tokenizer Configuration

- **Pretrained Model:** `klue/bert-base` [6, 7].
  - Load tokenizer and classification model:

```
tokenizer = AutoTokenizer.from_pretrained("klue/bert-base")
model     = AutoModelForSequenceClassification.from_pretrained(
                "klue/bert-base", num_labels=2
            )
```

  - All inputs are truncated to a maximum of 120 tokens, though test samples up to 122 tokens are allowed.
- **Tokenization Functions:**
  - `encode_fixed(sample)`: Tokenizes `sample["document"]` with `padding="max_length"` and `max_length=120` [8].
  - `encode_dynamic(sample)`: Tokenizes `sample["document"]` with `padding=False`, `truncation=True` (max_length=120) [8].

3.3 Pipeline Designs

We implement three pipelines—baseline, bucketing, and sorted inference—differing only in padding and batching methods. Each pipeline uses the same model architecture, optimizer, and evaluation metrics.

Baseline (Fixed Padding) Tokenize all reviews with `padding='max_length', max_length=120`. Batch size = 128 for both training and evaluation.

Bucketing (Dynamic Padding + Length-Aware Batching) Tokenize with `padding=False`. Set `group_by_length=True`, use `DataCollatorWithPadding`, and achieve an effective batch size of 128 via `per_device_train_batch_size=64` with `gradient_accumulation_steps=2`. During evaluation, apply dynamic padding without bucketing.

Sorted Inference (Bucketing Model + Length Sorting + Dynamic Padding) Use the model trained under the Bucketing setup to perform inference. Tokenize with `padding=False`, add a "length" field, sort all test samples in ascending order by length, and create batches of size 128 using `DataCollatorWithPadding` so that each sorted batch is padded only to its own maximum length.

In summary, all three pipelines share the same model, optimizer, and hyperparameters (Section 4.1), differing only in tokenization, padding, and batching. These designs enable an isolated assessment of padding overhead's impact on computation and runtime.

4 Experiments & Results

This section presents our experimental setup, followed by training and inference results for each pipeline. We include quantitative metrics.

4.1 Experimental Setup

- **Compute Environment:** All experiments were conducted on a single cloud GPU instance. Training and evaluation were performed using the HuggingFace `Trainer` API [9].
- **Hyperparameters:**
  - Epochs: 3
  - Optimizer: AdamW
  - Learning Rate: 2×10^{-5}
  - Weight Decay: 0.01
  - Warmup Ratio: 0.1
  - Max Sequence Length: 120 (training), test samples up to 122 allowed.
  - Train/Validation Split: 80/20% stratified on the filtered 146,176-sample training set
  - Batch Sizes:
    - Baseline: 128
    - Bucketing: per-device 64 with gradient accumulation 2 (effective 128)
    - Sorted Inference: 128
  - Metrics: Accuracy and weighted F1 (computed on validation and test sets)
  - Random Seed: 42
- **Measurement:**
  - *Throughput*, *Wall-Clock Time* and *FLOPs* were obtained directly from `Trainer` logs.

4.2 Training Results

Table 1 summarizes training performance for the baseline and bucketing pipelines. Bucketing uses dynamic padding with `group_by_length=True` and gradient accumulation to achieve an effective batch size of 128.
Observation: Bucketing reduces FLOPs by approximately 80% and train time by 60%, with only a 0.12% drop in validation accuracy compared to baseline.

Pipeline	FLOPs ($\times 10^{15}$)	Train Time (s)	Val Accuracy	Val F1
Baseline	21.6	3,589	0.9054	0.9053
Bucketing	4.32	1,453	0.9042	0.9042

Table 1: Training results on NSMC. FLOPs and train time are reported by `Trainer`. Validation metrics are computed on the 20% holdout of the training set.

Method	Test Accuracy	Test F1	Eval Time (s)	Throughput (samples/sec)
Baseline	0.90184	0.90183	430.7	116.1
Bucketing	0.90188	0.90187	326.0	153.4
Sorted	0.90188	0.90187	101.1	494.6

Table 2: Inference results on the NSMC test set. Eval time and throughput are reported by `Trainer`.

4.3 Inference Results

Table 2 reports test performance for baseline, bucketing inference, and sorted inference methods. Evaluation time, FLOPs, and throughput are directly from `Trainer` logs.
Observation:

- Bucketing inference reduces evaluation time by 24.3% over baseline, with negligible change in accuracy.
- Sorted inference reduces evaluation time by 76.5%, preserving identical test metrics.

4.4 Padding Statistics

- **Baseline:** Every sequence is padded to 120 tokens.
- **Bucketing Inference:** Dynamic padding without sorting results in batches whose maximum lengths cluster around 80–100 tokens, yielding moderate padding.
- **Sorted Inference:** Sorting by length before batching reduces average padding per batch to under 5 tokens.

For detailed batch-wise padding statistics, please refer to the figures in the Appendix.

5 Discussion

Our results confirm that minimizing padding via bucketing (during training) and length sorting (during inference) yields clear efficiency gains without harming classification accuracy.

5.1 Implications

- **Training Efficiency:** Length-aware bucketing reduced FLOPs by 80% and training time by 60%, with only a 0.12% drop in validation accuracy.
- **Inference Speed:** Sorting test samples by length yields a 76.5% reduction in evaluation time while maintaining test accuracy and F1.
- These strategies are particularly beneficial for Korean text, where token lengths vary widely: NSMC's average token length is 22.7, and most reviews are under 30 tokens.

5.2 Limitations

- **Single Dataset:** We only evaluated on NSMC. Other Korean NLP tasks (e.g., question answering, named entity recognition) may require different batching considerations.
- **Hardware Specificity:** Experiments were conducted on a single cloud GPU instance; results may vary on different hardware configurations.
- **Qualitative Scope:** We removed the qualitative observation section since no extensive misclassification analysis beyond the numerical metrics was performed.

5.3 Future Work

- **Generalization to Other Tasks:** Apply these padding strategies to additional Korean tasks (e.g., QA, NER).
- **Hybrid Optimization:** Combine length-aware batching with token-level pruning (e.g., TokenDrop [4]) for further efficiency gains.
- **Adaptive Bucketing:** Investigate dynamic bucket boundaries that adjust during training to evolving token-length distributions.

6 Conclusion

We present and evaluate three batching strategies for fine-tuning `klue/bert-base` on NSMC: (1) fixed padding (baseline), (2) length-aware bucketing (dynamic padding during training), and (3) length-sorted inference. Our experiments show:

- **Bucketing Training:** 80% reduction in FLOPs and 60% faster training, with negligible accuracy loss (-0.0012).
- **Sorted Inference:** 76.5% faster inference time while preserving test accuracy and F1.

These practical optimizations can be readily adopted in resource-constrained environments to accelerate Korean BERT workflows. All code and results are publicly available for reproducibility[2].

References

[1] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 4171–4186, Minneapolis, MN, 2019.

[2] Yoonho Park and Jihun Lee. Naver sentiment movie corpus (nsmc). `https://github.com/e9t/nsmc`, 2019.

[3] Xin Zhang, Wei Li, Hao Chen, and Rui Yang. Reducing bert computation by padding removal and curriculum learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1234–1243, 2021.

[4] Amrit Nagarajan and Anand Raghunathan. Tokendrop + bucketsampler: Toward efficient padding-free fine-tuning of language models. *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 1234–1245, 2023.

[5] Matej Kosec, Sergio P. Perez, and Andrew Fitzgibbon. Packing: Towards 2× nlp bert acceleration. In *NeurIPS 2021 AI for Science Workshop*, 2021.

[6] Hugging Face. klue/bert-base. `https://huggingface.co/klue/bert-base`, 2020.

[7] Jaewoo Park, Doyun Han, Seung Kim, and *et al.* Klue: Korean language understanding evaluation benchmark. `https://github.com/KLUE-benchmark/KLUE`, 2021.

[8] Hugging Face. Padding and truncation. `https://huggingface.co/docs/transformers/v4.30.0/en/main_classes/tokenizer#padding-and-truncation`, 2024.

[9] Hugging Face. Trainer api. `https://huggingface.co/docs/transformers/main_classes/trainer`, 2024.

[2] `https://github.com/kmasta/AIFFEL_quest_rs/blob/master/GoingDeeper/Gd08/nsmc_with_klue_bert_base.ipynb`

Appendix: Comprehensive Padding Statistics

Below are comprehensive batch-wise statistics for token lengths, dynamic padding, and fixed-length (120) padding on each data split (Train, Validation, Test, and Sorted Test). Each subfigure corresponds to one data split:

- **(a) Train split**
- **(b) Validation split**
- **(c) Test split**
- **(d) Sorted Test split (dynamic padding & length-sorted)**

Within each subfigure, a 3×3 grid of histograms is shown:

- **Row 1 (Token Length Distributions):**
  - Column 1: Minimum token length per batch
  - Column 2: Maximum token length per batch
  - Column 3: Average token length per batch
- **Row 2 (Dynamic Padding Distributions):**
  - Column 1: Minimum number of padding tokens per sequence (batch-wise)
  - Column 2: Maximum number of padding tokens per sequence (batch-wise)
  - Column 3: Average number of padding tokens per sequence (batch-wise)
- **Row 3 (Fixed 120-Length Padding Distributions):**
  - Column 1: Minimum number of padding tokens assuming fixed length 120
  - Column 2: Maximum number of padding tokens assuming fixed length 120
  - Column 3: Average number of padding tokens assuming fixed length 120

Since most NSMC reviews are short (average 22 tokens), applying dynamic padding (rows 2) greatly reduces the total padding compared to fixed-length padding (rows 3). In particular, when sorting the test split by length (subfigure d), the average padding per batch further decreases.

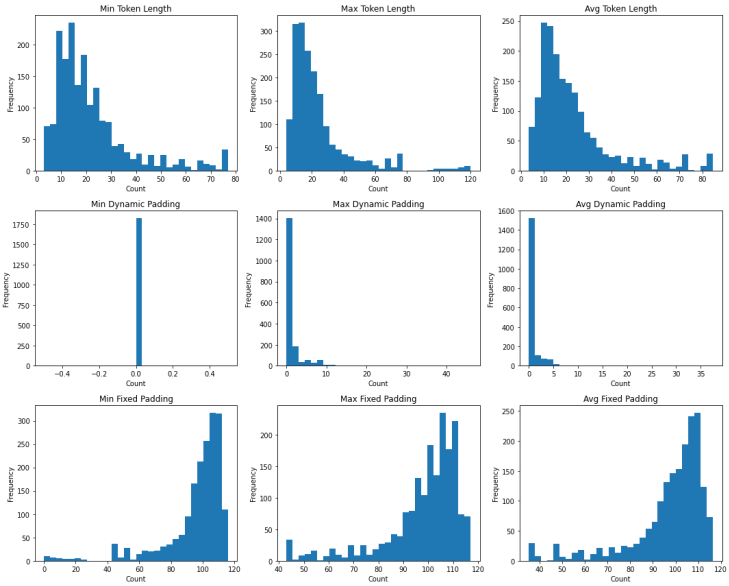

Figure 2: Train split: Token / Dynamic Padding / Fixed 120-length Padding (rows top to bottom).

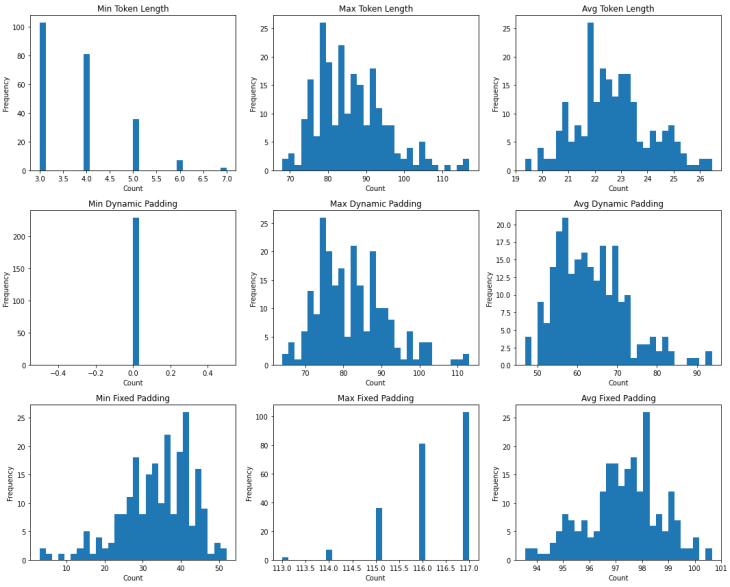

Figure 3: Validation split: Token / Dynamic Padding / Fixed 120-length Padding (rows top to bottom).

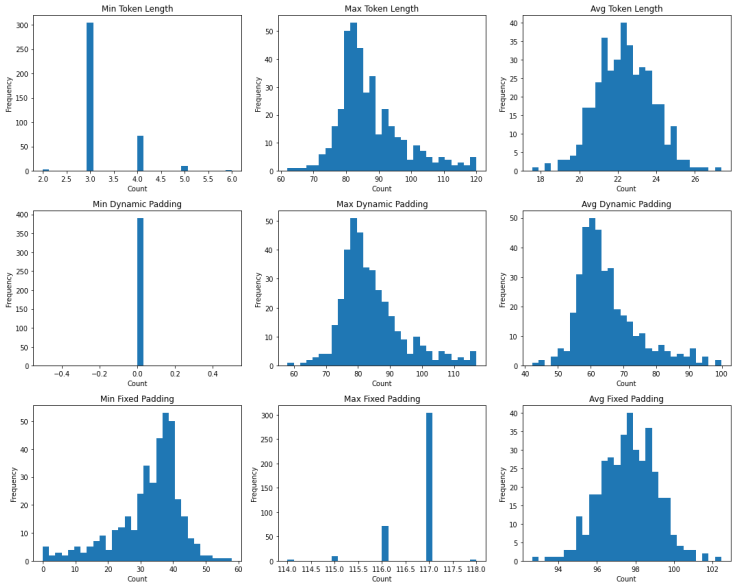

Figure 4: Test split: Token / Dynamic Padding / Fixed 120-length Padding (rows top to bottom).

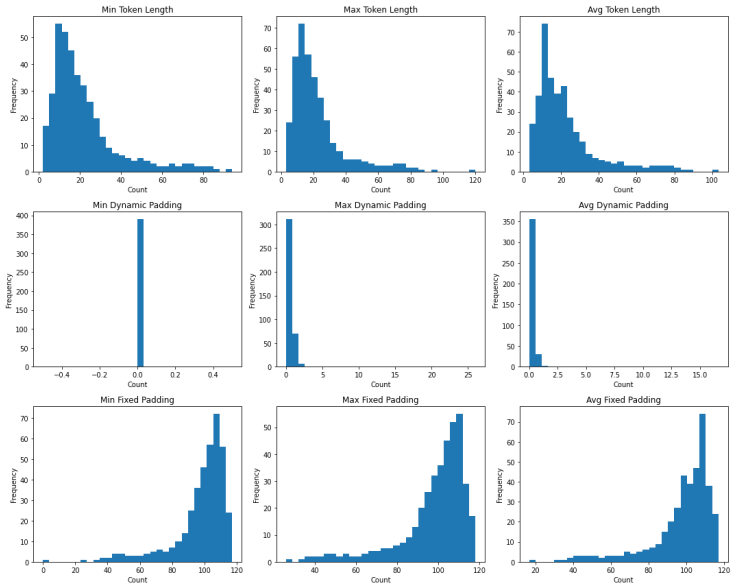

Figure 5: Sorted Test split: Token / Dynamic Padding / Fixed 120-length Padding (rows top to bottom). Since we sorted test samples by length before applying dynamic padding, the average padding per batch further decreases.