Pretokenization in LLM Training: Implementation and Analysis

Introduction

Pretokenization moves the text-to-token conversion process from the training loop to a preprocessing stage. This approach aims to reduce computational overhead during training by removing redundant tokenization operations. In *Assignment 2* we saw two approaches for tokenization: a simple one with *padding* , and a more advanced *padding-free* one creating a continuous stream of useful tokens that one can easily collate. I create a pretokenization for *both* approaches. Finally, I extensively benchmark the result and judge whether its a useful integration or not.

Implementation Details

Pretokenization Approach

We implemented two pretokenization formats:

1. **Padded Format**: Tokenizes text with fixed-length sequences and attention masks (remaining tokens are padded), exactly matching the format used by the original `ParquetDataset`.
2. **Token-List Format**: Stores only token IDs without padding, preserving the continuous token stream needed for the padding-free approach.

Both formats are saved as Parquet files with Snappy compression using `row_group_size=1000`, matching the structure of the original training data. This parameter choice came from inspecting the metadata of the original dataset.

Format	Size	Efficiency
Original text	2.3 GB	Baseline
Pretokenized padded	2.6 GB	Larger due to both tokens and attention masks
Pretokenized token-list	1.7 GB	Smaller due to no padding or attention masks

The token-list format is more space-efficient because it stores only the essential token IDs and doesn't need attention masks.

Technical Implementation

We have a preprocessing step which tokenizes the data – which has to be executed once (see `src/preprocessing/pretokenize.py`). It implements specialized tokenization for each format. While pretokenization moves computation from training time to a preprocessing stage, it's important to consider this one-time computational cost. Our benchmarks show:

- Token-list format: Tokenization completed in 266.93 seconds (2944.20 docs/sec)
- Padded format: Tokenization completed in 337.58 seconds (2328.06 docs/sec)

We minimally adapted the existing DataLoader classes:

- `PreTokenizedDataset` (padding approach): Directly loads and return the pretokenized data instead of tokenizing on-the-fly.
- `IterablePreTokenizedDataset` (padding-free token-list approach): Reads raw token lists instead of text and maintains identical BOS token handling and loss masking logic

Both implementations maintain exact compatibility with the original code, only changing the data loading step while preserving all other processing logic. The implementation can be found in `src/data/pretokenized_dataset.py`, and also in the Appendix.

Next, modified the training pipeline `train.py` to conditionally use pretokenized datasets with minimal code changes. One can give a new flag `--pretokenized` along the training script, together with the version one wants to use – either `--dataset-type padded` or `--dataset-type token-list`. This design maintains complete backward compatibility while allowing direct performance comparisons.

Tests

We rigorously tested our pretokenization (see also `analysis/pretokenization_verification.py`) by making sure the pretokenization matches on-the-fly tokenization. We also inspected the losses during traiing (see Appendix) to make sure pretokenization does not hurt performance.

Results and Analysis

Raw Processing Speedup

In isolated benchmarks, pretokenization showed significant improvements (measured for 1000 samples):

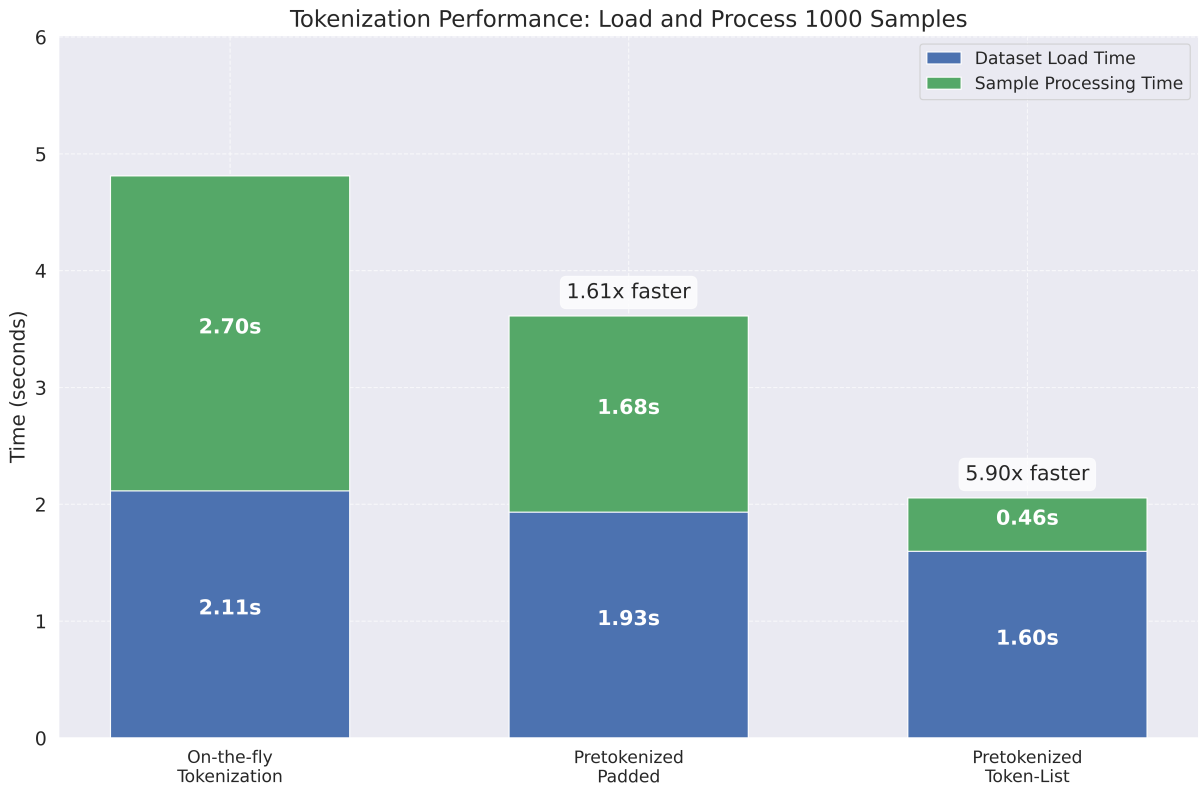

Figure 1: Tokenization Benchmark

The *token-list* format shows particularly impressive gains in processing time – reducing it from 2.70s to just 0.46s. This clearly demonstrates the theoretical efficiency gains possible by eliminating redundant tokenization operations during the training stage.
However, as we see in the next section, these dramatic processing speedups do not translate to meaningful improvements in actual training throughput.

Training Performance

We benchmarked the actual benefit (in terms of throughput, MFU, and achieved TFLOPS) in model training, using batch size 1 and 2 (more was not possible due to memory limitations on the GPU). Despite the impressive tokenization speedups, training throughput showed minimal differences:

Configuration	Tokens/sec	MFU %	TFLOPs
Baseline padded (batch_size=1)	6544.40	31.97	316.23
Baseline padded (batch_size=2)	7873.05	38.47	380.43

Configuration	Tokens/sec	MFU %	TFLOPs
Pretokenized padded (batch_size=1)	6333.60	30.94	306.04
Pretokenized padded (batch_size=2)	7605.06	37.16	367.48
Baseline padding-free (batch_size=1)	6902.55	35.97	355.77
Baseline padding-free (batch_size=2)	7410.96	36.21	358.10
Pretokenized token-list (batch_size=1)	6224.24	30.41	300.76
Pretokenized token-list (batch_size=2)	7475.30	36.52	361.21

Interestingly, our data suggests that pretokenization benefits may scale with batch size. For the token-list approach, pretokenization went from 9.8% slower at batch_size=1 to 0.9% faster at batch_size=2. This trend suggests that with larger batch sizes (which we were unable to test due to GPU memory constraints), pretokenization could provide more significant performance benefits.

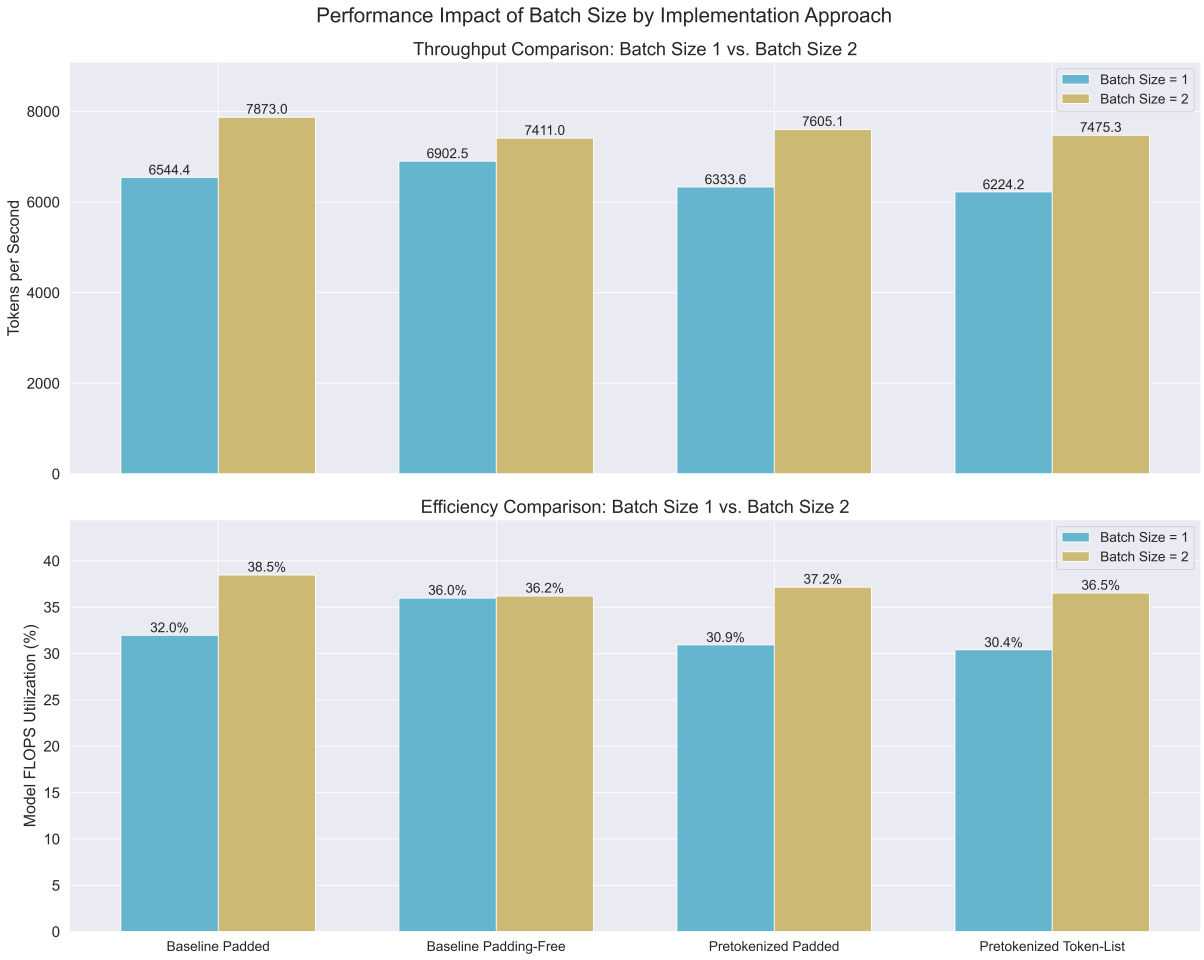

Figure 2: Bar chart comparing tokens per second across tokenization approaches with batch sizes 1 and 2, showing pretokenized token-list improving from slower to slightly faster as batch size increases.

Analysis and Insights

In this section we try to analyze, why the actual benefit of pretokenization is not really evident.

- **Tokenization is not the bottleneck**: Raw tokenization processes tokens at $\approx$ $774K$ tokens/second (see `analysis/benchmark_dataset.py` script), while our training pipeline achieves only $\approx 7K$ tokens/second throughput. This $100\times$ difference indicates that tokenization represents a very small fraction of the overall training time. The main training bottleneck is GPU computation, not tokenization.
- PyTorch's DataLoader already efficiently prefetches and processes data asynchronously on CPU while the GPU performs computation, effectively hiding most tokenization overhead.

- All methods improved with larger batch sizes. Unfortunately, we could only try up to batch_size=2, due to memory constraints. In future work, for example when implemented Data Distributed Parallel, one could test the effect of pretokenization again with larger batch sizes.

These findings align with *Amdahl's Law*, which states that the potential speedup of a system is limited by the fraction of time spent in the part being optimized. Since tokenization represents such a tiny fraction of overall training time, even eliminating it entirely would yield minimal overall improvement – at least for the small batch sizes we tested.

Conclusion

Pretokenization offers significant speedups for the tokenization process itself (up to $5.9\times$ faster) but doesn't translate to proportional training throughput improvements. Our key findings include:

- The token-list format provides better storage efficiency (1.7 GB) compared to the original text data (2.3 GB) and the padded format (2.6 GB).
- Despite theoretical benefits, pretokenization had no significant impact on training performance metrics (tokens/second, MFU, TFLOPs).

The complete implementation, including all source code, benchmark scripts, and analysis tools used, is available at: https://github.com/RamonKaspar/Large-Scale-AI-Engineering-Project.

Appendix

Implementation of the DataLoaders

The `PreTokenizedDataset` handles the padded version, simply loading the pretokenized data from the parquet file:

```python
class PreTokenizedDataset(Dataset):
    """Dataset for pretokenized padded data."""
    def __init__(self, parquet_file, sequence_length, training_samples):
        self.parquet_ds = pq.read_table(parquet_file, memory_map=True)
        self.real_length = len(self.parquet_ds)
        self.sequence_length = sequence_length
        self.training_samples = training_samples
```

```
    def __len__(self):
        return self.training_samples

    def __getitem__(self, idx):
        actual_idx = idx % self.real_length
        # Direct access to pretokenized data - key optimization point
        return {
            "input_ids": self.parquet_ds["input_ids"][actual_idx].as_py(),
            "attention_mask":
            ↪  self.parquet_ds["attention_mask"][actual_idx].as_py()
        }
```

The `IterablePreTokenizedDataset` handles padding-free streaming, carefully preserving the original token sequencing logic:

```
class IterablePreTokenizedDataset(IterableDataset):
    """Streams pretokenized tokens without padding."""
    def __init__(self, parquet_file, sequence_length, bos_token_id=1):
        self.parquet_ds = pq.read_table(parquet_file, memory_map=True)
        self.real_length = len(self.parquet_ds)
        self.sequence_length = sequence_length
        self.bos_token_id = bos_token_id
        self.current_index = 0
        self.token_buffer = []

    def __next__(self):
        # Fill buffer until we have enough tokens
        while len(self.token_buffer) < self.sequence_length + 1:
            if self.current_index >= self.real_length:
                self.current_index = 0

            # Direct access to pretokenized tokens
            tokens = self.parquet_ds["tokens"][self.current_index].as_py()
            self.current_index += 1

            # Preserve BOS token handling logic from original
            ↪  implementation
            if not self.token_buffer or self.token_buffer[-1] !=
            ↪  self.bos_token_id:
                self.token_buffer.append(self.bos_token_id)

            self.token_buffer.extend(tokens)

        token_sample = self.token_buffer[:self.sequence_length + 1]
```

```
        if len(self.token_buffer) > self.sequence_length + 1:
            self.token_buffer = self.token_buffer[self.sequence_length:]
        else:
            self.token_buffer = []

        inputs = torch.tensor(token_sample[:-1])
        labels = torch.tensor(token_sample[1:])

        # Mask loss for tokens after BOS
        bos_positions = (inputs ==
↪   self.bos_token_id).nonzero(as_tuple=True)[0]
        for pos in bos_positions:
            if pos < len(labels):
                labels[pos] = -100

        return inputs, labels
```

Training Loss

The following plot shows the training loss over time, using different on-the-fly and pretokenization approaches. We can see that pretokenization does not hurt performance.

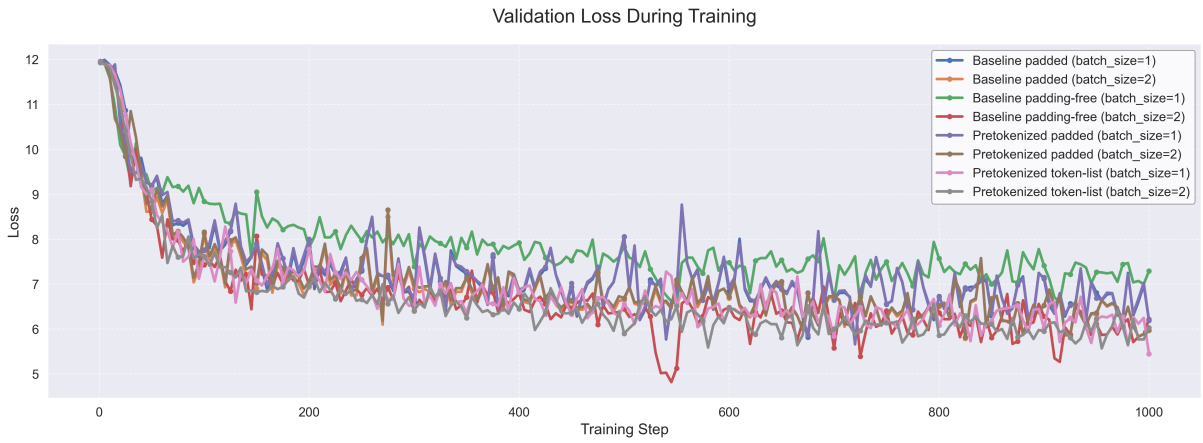

Figure 3: Training loss over time

Training Performance Over Time

While the results section presents average performance metrics, examining how these metrics evolve during training provides additional insights. These charts confirm that the relative performance

differences between approaches remain consistent throughout training, with no significant degradation or improvement over time.

Figure 4: MFU over time

Figure 5: Tokens per second over time