SpaceByte: Towards Deleting Tokenization from Large Language Modeling

Kevin Slagle
Rice University
kevin.slagle@rice.edu

Abstract

Tokenization is widely used in large language models because it significantly improves performance. However, tokenization imposes several disadvantages, such as performance biases, increased adversarial vulnerability, decreased character-level modeling performance, and increased modeling complexity. To address these disadvantages without sacrificing performance, we propose SpaceByte, a novel byte-level decoder architecture that closes the performance gap between byte-level and subword autoregressive language modeling. SpaceByte consists of a byte-level Transformer model, but with extra larger transformer blocks inserted in the middle of the layers. We find that performance is significantly improved by applying these larger blocks only after certain bytes, such as space characters, which typically denote word boundaries. Our experiments show that for a fixed training and inference compute budget, SpaceByte outperforms other byte-level architectures and roughly matches the performance of tokenized Transformer architectures.

1 Introduction

Most language models are trained using tokenization, which partitions text into tokens that typically consist of words or subwords. Tokenization is useful because it significantly decreases the inference and training computational costs of large language models. However, tokenization also imposes several disadvantages, including a performance penalty on text distributions different from what the tokenizer was trained on [1]; increased vulnerability to adversarial attacks [2]; worse character-level modeling performance [3], and additional model complexity.[1]

Recently, MegaByte [4], MambaByte [5], and more [6, 7] have been proposed as new byte-level autoregressive language models that model bytes instead of tokens. (See [8–14] for encoder and encoder-decoder byte-level modeling.) To address the longer context size resulting from modeling bytes instead of tokens, MegaByte uses multiscale modeling (which is more challenging for autoregressive models than Transformer encoders [15, 16]), while MambaByte uses Mamba blocks [17] instead of Transformer blocks. But although MegaByte and MambaByte have been shown to perform better than a standard byte-level Transformer, to our knowledge, no byte-level autoregressive large language model architecture has been shown to match the performance of tokenized models when controlling for compute costs.

In this work, we study the performance of byte-level and subword-level autoregressive models when trained using a fixed compute budget. We measure the performance in terms of the cross entropy (measured in bits-per-byte), which has been shown to be a strong predictor of down-stream performance [18]. In addition to controlling for training compute, we also control for inference compute costs (measured in FLOPs). We find that byte-level Transformer and MegaByte models

[1] See also Andrej Karpathy's tweet twitter.com/karpathy/status/1657949234535211009 and video youtube.com/watch?v=zduSFxRajkE&t=6725s on the disadvantages of tokenization.

Preprint. Under review.

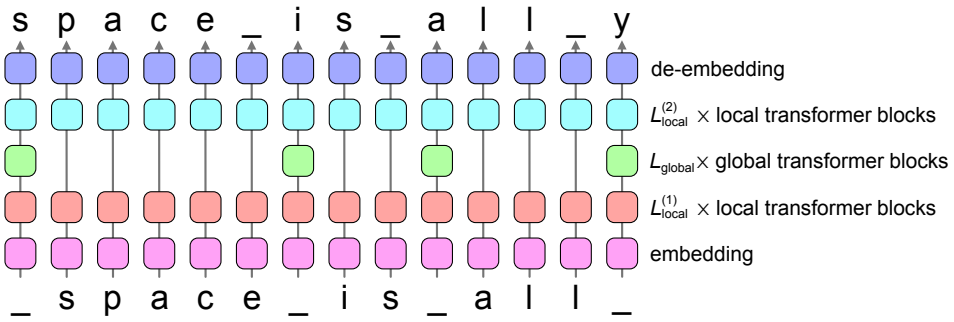

Figure 1: An overview of the SpaceByte architecture. The embedding, local transformer blocks, and de-embedding (i.e. a layer norm and linear) are the standard Transformer decoder layers. SpaceByte modifies the standard transformer by applying "global" transformer blocks only after certain bytes, such as space characters. The intuition is that the first character of a word is typically the hardest to predict; thus this positioning of the global blocks should make the best use of the global blocks (which use a larger model dimension).

can require roughly 10 times more training FLOPs to achieve the same performance as a subword-level Transformer. To close this substantial performance gap, we propose a new byte-level decoder architecture: SpaceByte.

SpaceByte also utilizes multiscale modeling to improve efficiency by grouping bytes into patches. But unlike MegaByte, which uses a fixed patch size, SpaceByte uses a simple rule to dynamically partition the bytes into patches that are aligned with word and other language boundaries. (A similar technique was also explored by Thawani et al. [6].) Our experiments show that this simple modification is crucial for performance, allowing SpaceByte to outperform other byte-level architectures and roughly match the performance of subword Transformers across a variety of text modalities.

Our experiments are performed on datasets consisting of English books, LaTeX formatted arXiv papers, and open-source code. For other data modalities, SpaceByte with our simple patching rule might not be as effective.

2 SpaceByte

The SpaceByte architecture is summarized in Figure 1. In a nutshell, SpaceByte can be thought of as a byte-level Transformer model, but with extra "global" transformer blocks (with a larger model dimension) inserted in the middle, which are only applied a fraction of the time. While the MegaByte architecture applies the global transformer blocks every $P \sim 8$ bytes, we hypothesize that this fixed spacing hinders performance. Our intuition is that the first character of a word is typically significantly harder to predict than the following characters. We therefore expect that performance can be improved by applying the global blocks primarily at word boundaries.

Global Block Insertion Rule In this work, we consider a very simple rule to dynamically decide when to apply the global blocks. We assume that the text bytes are encoded using the UTF-8 encoding. We define a byte to be *spacelike* if the byte does not encode a letter, number, or UTF-8 continuation byte[2]. We apply the global blocks after any spacelike byte that is not preceded by another spacelike byte (and after any BOS token). See Figure 2 for examples.

The most common spacelike byte is the space character. Thus, the global blocks are applied most frequently to predict the first character of a word, which we expect is the hardest character to predict in a given word. With fixed patch size (e.g. as in MegaByte), the global blocks are typically inserted in the middle a word, which we expect is inefficient because predicting the rest of the word could likely be more efficiently accomplished using the local blocks. We define continuation bytes to be spacelike so that languages that do not use spaces between words can still benefit from the global blocks between multi-byte characters (e.g. Chinese characters consists of three bytes in UTF-8).

[2]UTF-8 uses a variable number of bytes to encode a character. English letters or numbers consist of a single byte. Characters that are encoded using multiple bytes are encoded using a leading byte (which is spacelike by our definition) followed by one or more continuation bytes (which are not spacelike).

PG-19:

arXiv:

Github:

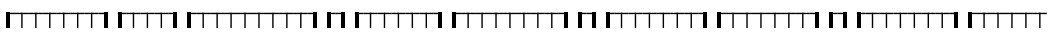

Figure 2: Examples of patch boundaries from datasets that we study. Spacelike bytes are underlined and colored blue. Patches boundaries are drawn above the text. Each patch ends after a spacelike byte that is not preceded by another spacelike byte. Consequently, each patch begins with zero or more spacelike bytes, followed by one or more non-spacelike bytes, and ends with a single spacelike byte. The global blocks predict the first character of each patch. The downward arrow (↓) denotes a newline byte. The left and right quotation characters, (“) and (”) in the PG-19 example, are encoded using three bytes in UTF-8. The first of the three bytes is spacelike, while the later two bytes are UTF-8 continuation bytes, which are not spacelike and are each denoted using a bullet point (•) above.

Although this very simple “spacelike” rule is likely not the optimal rule, we find that it works surprisingly well in practice for English text, LaTeX formatted papers, and code. Nevertheless, a critical future direction is to optimize better rules using data rather than our simple heuristic.

Important Details Since the global blocks are not applied as often as the local transformer blocks, it is advantageous to use a larger model dimension for the global transformer blocks. To increase the dimensions of an activation vector before the global blocks, we simply pad the activation vector with zeros. To decrease the dimension, we truncate the activation vector.

In our experiments, we use a significantly larger context size than the model dimension D_{local} of the local transformer blocks. To prevent the attention mechanism from dominating the compute costs for the local model, we use an attention window [19–21] of length D_{local} for the local transformer blocks.

See Appendix C for pseudocode. Additional details specific to our experiments are provided in Sections 4.1 and 4.2 and Appendix A.

3 Related Work

The most straight-forward consequence of modeling bytes instead of subword tokens is that the length of a sequence typically increases by about a factor of four. This increased sequence length increases the training and inference compute cost for modeling a given long sequence of text for a Transformer due to the quadratic scaling of attention.

MegaByte The MegaByte architecture strives to use multiscale Transformer modeling to lessen these performance issues. In particular, MegaByte groups bytes into patches of a fixed patch size P. Each patch of bytes is vectorized and then fed into a “global” Transformer model. The output of the global model is then fed into a “local” Transformer model that autoregressively outputs byte-level logits. [4]

For a context size of T bytes, MegaByte’s global Transformer model compresses the context into only T/P patches, which can significantly decrease the compute cost for modeling long sequences. Similar to Yu et al. [4], we also find that MegaByte outperforms a standard byte-level Transformer. However, we find that MegaByte’s performance is remarkably close to a stronger byte-level Transformer baseline that simply uses a sliding window attention mechanism [19–21] to increase the context size without increasing the compute costs.

Yu et al. [4] do not compare MegaByte to subword-level Transformer in compute controlled experiments. In our compute controlled experiments, we find that MegaByte’s performance significantly lags behind a subword-level Transformer.

Compared to MegaByte, SpaceByte makes the crucial change that patches are dynamically sized to be commensurate with the text, e.g. with word boundaries. We also add an additional local model before the global model (while MegaByte only utilizes a single local model after the global model) to help the model deal with the dynamical patch sizes. We also use significantly longer attention windows for our local models. We find that these changes allow SpaceByte to significantly improve upon the performance of MegaByte and roughly match the performance of subword-level Transformers.

MambaByte The MambaByte architecture [5] takes an alternative approach to avoiding the quadratic compute scaling of the attention mechanism by replacing the Transformer block with a Mamba block [17]. Wang et al. [5] find that their byte-level MambaByte models outperform byte-level Transformer and byte-level MegaByte models. They perform one controlled experiment with a subword model, where they find that MambaByte slightly outperforms Mamba (using tokens) when controlling for the amount of model parameters and training data (which was 14 epochs of the PG-19 dataset). But this experiment was not controlled for compute as MambaByte was trained using roughly four times as much compute than Mamba. We view the Mamba and MambaByte architectures as complementary to our work, as the Mamba block could be integrated into SpaceByte (or MegaByte) in place of Transformer blocks.

Layer Skipping SpaceByte could be though of as a Transformer that employs a novel kind of text-dependent layer skipping [22–28] on the middle layers.

Word Boundary Using word boundaries to partition patches in autoregressive language modeling was explored by Thawani et al. [6] (and also Edman et al. [12] for encoder-decoder modeling). However, their experiment methodology differed very significantly from ours. While we study large language model perplexity (via bits-per-byte) in compute-limited settings, Thawani et al. [6] study models trained on small datasets ($\sim$ 5MB) over 100 epochs and evaluated using word prediction accuracies. The design of their local blocks also differ from SpaceByte as their local blocks do not attend across word boundaries and they use non-causal attention for the local blocks that preceed the global blocks.

4 Experiment Setup

Our experiments compare the performance of our byte-level SpaceByte architecture to subword-level Transformer and byte-level Transformer and MegaByte architectures. To fairly compare the performance between the byte and subword level models, we measure the cross-entropy of the test dataset in terms of bits-per-byte.[3] Given the substantial variation in inference compute costs across the models we study, we also compare their inference compute costs to provide a more comprehensive evaluation. We use FLOPs-per-byte as a simple software and hardware–independent proxy for inference compute costs, which is the number of FLOPs (see Appendix A.1) required to model a byte of text.[4]

Note that by controlling for both total training compute and FLOPs-per-byte, we are also controlling for the amount of training data since $(\text{bytes trained}) = (\text{training FLOPs})/(\text{training FLOPs-per-byte})$. The FLOPs-per-byte during training is equal to three times the FLOPs-per-byte during inference (due to the backwards pass during training).

We therefore study the Pareto frontier of lowest bits-per-byte and lowest FLOPs-per-byte. We train all models using a compute-controlled setup, using either 10^{18} or 10^{19} FLOPs. In order to effectively explore this Pareto frontier, we train models using a grid of different model dimensions and numbers of layers, as specified in Appendix B.3.

Datasets Following the MegaByte [4] and MambaByte [5] experiments, we benchmark our models on a diverse range of long-form datasets: PG-19 (English-language books written before 1919) [31]; arXiv (papers from ArXiv written in LaTeX, extracted from the arXiv component of The Pile [32]); and Github (open-source code repositories, extracted from the Github component of The Pile [32]).

[3] The bits-per-byte (BPB) is equal to $\text{BPB} = \text{XE}\, N_{\text{tokens}}/(N_{\text{bytes}} \ln 2)$, i.e. the cross entropy per token (XE) times the number of tokens per byte ($N_{\text{tokens}}/N_{\text{bytes}}$) divided by $\ln 2$ (to convert nats to bits). N_{tokens} and N_{bytes} are the number of tokens and bytes in the dataset, respectively. For byte-level models, $N_{\text{tokens}} = N_{\text{bytes}}$ since bytes are used in place of tokens.

[4] We note that in order for memory bandwidth to not be a bottleneck during inference, the batch size must be sufficiently large and e.g. grouped-query attention [29, 30] must be used.

Listing 1: Pytorch pseudocode for SpaceByte

```
def forward(self, tokens, targets=None):
    B, T = tokens.shape # batch size, context size
    T_global = self.global_context_size
    D_local = self.local_model_dimension
    D = self.global_model_dimension

    # embedding:
    x = self.token_embedding(tokens)
    x = x + self.local_position_encoding
    # initial local transformer blocks:
    for block in self.initial_blocks:
        x = block(x)

    # global block insertion rule:
    use_global = ( # not a letter, number, or UTF-8 continuation byte
        (tokens < ord('0')) |
        ((ord('9') < tokens) & (tokens < ord('A'))) |
        ((ord('Z') < tokens) & (tokens < ord('a'))) |
        ((ord('z') < tokens) & (tokens < 0b1000_0000)) |
        (0b1100_0000 <= tokens) )
    use_global[:, 1:] &= use_global[:, :-1].bitwise_not() # not
       preceded by another spacelike byte
    use_global |= tokens == self.BOS_token # always use global blocks
       after BOS tokens

    # calculate global block indices:
    num_global = torch.full((B,), -1) # number of global blocks used
    global_idx = torch.full((B, T_global), T-1) # global block indices
    for b in range(B):
        idx, = use_global[b].nonzero(as_tuple=True)
        if targets is not None and len(idx) > T_global:
            # ignore targets with insufficient global blocks:
            targets[b, idx[T_global]:] = -1
        num_global[b] = len(idx[:T_global])
        global_idx[b, :num_global[b]] = idx[:T_global]

    # select activations for global blocks:
    y = x.gather(1, global_idx[:,:,None].expand(B, T_global, D_local))
    # expand model dimension by padding with zeros:
    y = torch.cat([torch.zeros(B, T_global, D - D_local), y], -1)

    # global transformer blocks:
    y = y + self.global_position_encoding
    for block in self.global_blocks:
        y = block(y)

    # add global block activations to local blocks:
    x = torch.stack([
        x[b].index_add(0, global_idx[b, :n], y[b, :n, -D_local:])
        for b, n in enumerate(num_global) ])

    # final local transformer blocks:
    for block in self.final_blocks:
        x = block(x)
    # de-embedding:
    logits = self.logits_linear(self.layer_norm(x))

    cross_entropy_loss = None
    if targets is not None:
        cross_entropy_loss = torch.nn.functional.cross_entropy(
            logits.view(B*T, 256), targets.view(B*T),
            ignore_index=-1).view(B, T)
    return logits, cross_entropy_loss
```