Byte latent transformer

Tokenization has previously been essential because directly training LLMs on bytes is prohibitively costly at scale due to long sequence lengths. To efficiently allocate compute, we propose a dynamic, learnable method for grouping bytes into patches and a new model architecture that mixes byte and patch information. Byte Latent Transformer (BLT), a new byte-level LLM architecture that, for the first time, matches tokenization-based LLM performance at scale with significant improvements in inference efficiency and robustness. BLT encodes bytes into dynamically sized patches, which serve as the primary units of computation. Patches are segmented based on the entropy of the next byte, allocating more compute and model capacity where increased data complexity demands it.

- BPE groups bytes into static set of tokens. It Requires understanding nuances at the individual character.
- Due to long sequence of lengths, it was costly to train.
- Allocates compute to every token.
- Group bytes into patches
- Arbitrary groups of bytes are mapped to latent patch representation using learned encoder decoder
- BLT segments data based on the entropy of the next byte prediction.
- Segmenting bytes into patches in the Byte Latent Transformer (BLT) framework allows the model to dynamically decide where to focus computational resources, adapting to the complexity and structure of the input rather than using a static rule like traditional tokenization.

We train a small byte level LM trained on the training data to produce prob distribution pe over the next byte given preceding bytes. For each byte position we calculate entropy, if entropy at that position is greater than global threshold
then place patch there. Regions with low entropy correspond to predictable, repetitive or simple content where bigger patches can be used.

$$H(x_i) = \sum_{v \in \mathcal{V}} p_e(x_i = v | \boldsymbol{x}_{<i}) \log p_e(x_i = v | \boldsymbol{x}_{<i})$$

Identifies positions where the entropy suddenly increases significantly compared to the previous byte, breaking a roughly monotonic decrease of entropy within a patch. It detects sudden rises in uncertainty indicating a boundary. A critical difference between patches and tokens is that with tokens, the model has no direct access to the underlying byte features. These patches are sent to transformer and then the transformer predicts bytes. The model would be causal and we would be taking the previous patches. The one hot vector is converted into embedding and then fed to transformer. We then n-gram information to it. Example, we take a particular token and add n-gram hash information

$$e_i = x_i + \sum_{n=3,\ldots,8} E_n^{hash}(\text{Hash}(g_{i,n}))$$

$$\text{where, Hash}(g_{i,n}) = \text{RollPolyHash}(g_{i,n})\% |E_n^{hash}|$$

Above is just byte sequence, to make patches, we need to use for cross attention.

BLT architecture

- An autoregressive language model generates text by predicting one element at a time, based on previous elements in sequence. Language Models helps in computing the probabilities of the patches.
- When generating, BLT needs to decide whether the current step in the byte sequence is at a patch boundary or not as this determines whether more compute is invoked via the Latent Transformer.
- It has a local encoder (bytes to patches) and a local decoder(patches to bytes).
- Operating directly on bytes is costly because language sequence are very long at byte level. By grouping bytes into patches, global model deals with much shorter sequences.

1. The **Byte Latent Transformer (BLT)** architecture is designed to process raw byte sequences efficiently by combining **local encoding/decoding** with a **global transformer**. The motivation is to handle long sequences of bytes (like text, code, DNA) in a scalable and expressive way.

It has three main components:

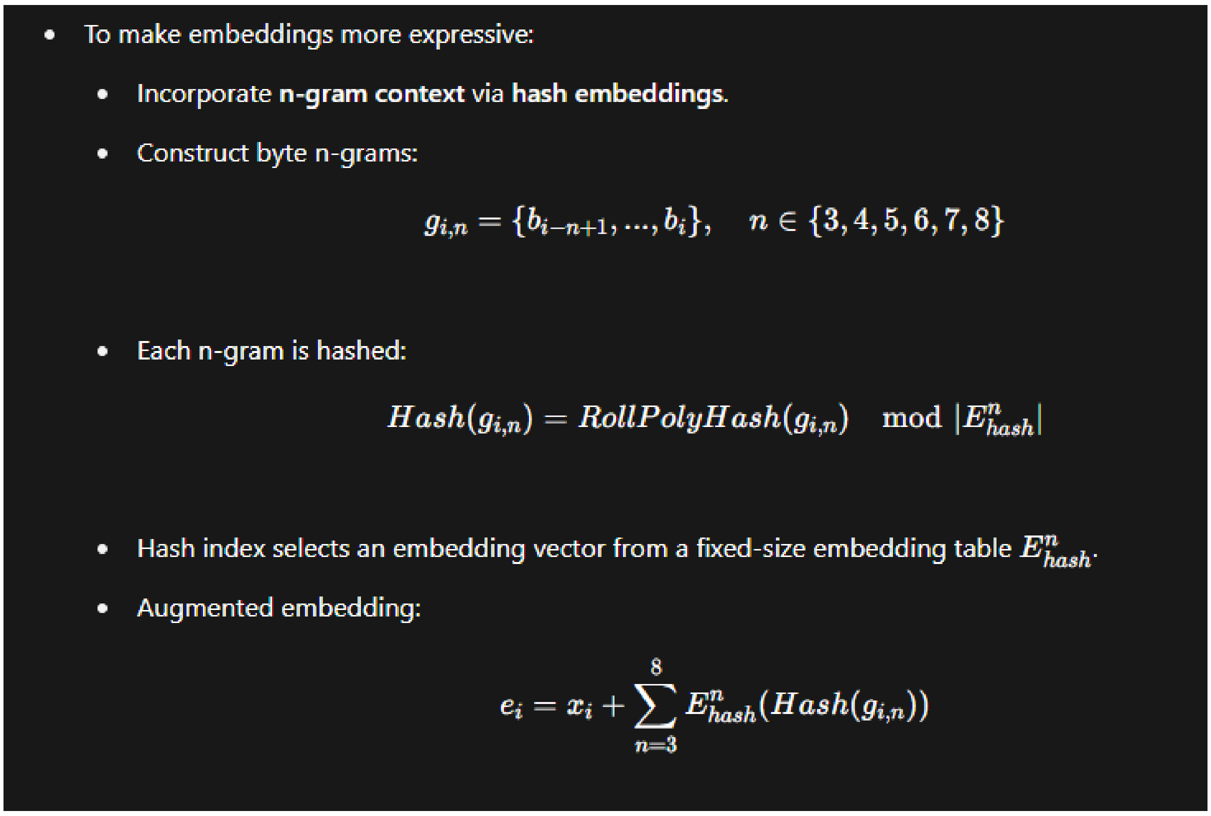
- **Local Encoder (E):** Maps raw byte sequences into expressive patch representations.
- **Global Transformer (G):** Operates autoregressively over patch representations.
- **Local Decoder (D):** Decodes patch representations back into raw byte sequences.

2. Local Encoder (E)

The **local encoder** is a lightweight transformer with significantly fewer layers than the global transformer (lE<<lGl_E << l_GlE<<lG). It converts **bytes → patches**.

2.1 Byte Embedding and Hash n-gram Embeddings

- Input: Byte sequence bib_ibi
- Raw bytes embedded via R256×hE\mathbb{R}^{256 \times h_E}R256×hE, giving embeddings xix_ixi.
- To make embeddings more expressive:
    - Incorporate **n-gram context** via **hash embeddings**.



- To make embeddings more expressive:
  - Incorporate **n-gram context** via **hash embeddings**.
  - Construct byte n-grams:

$$g_{i,n} = \{b_{i-n+1}, \ldots, b_i\}, \quad n \in \{3,4,5,6,7,8\}$$

  - Each n-gram is hashed:

$$Hash(g_{i,n}) = RollPolyHash(g_{i,n}) \mod |E^n_{hash}|$$

  - Hash index selects an embedding vector from a fixed-size embedding table E^n_{hash}.
  - Augmented embedding:

$$e_i = x_i + \sum_{n=3}^{8} E^n_{hash}(Hash(g_{i,n}))$$

  - Normalization is applied (dividing by number of n-gram sizes + 1).

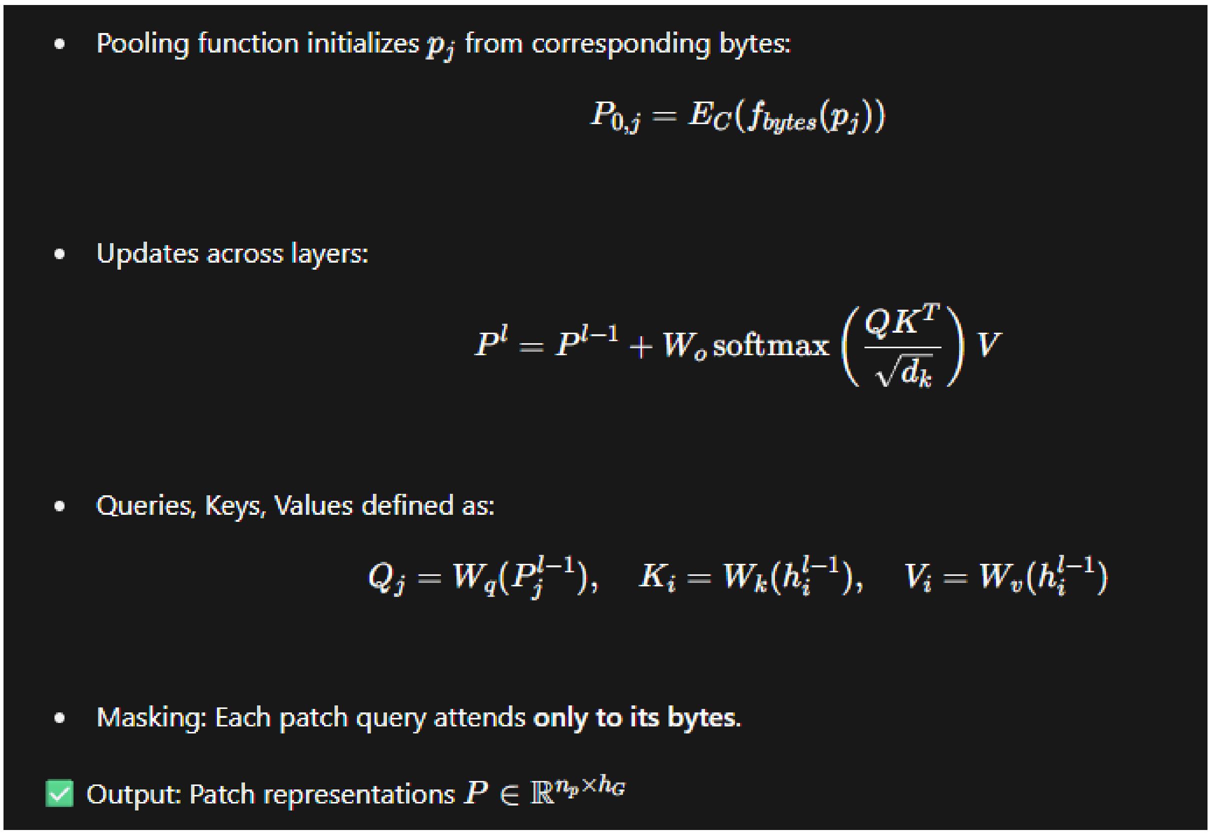
This allows each byte embedding to carry both local byte identity and n-gram context.

2.2 Encoder Transformer + Cross Attention

- Alternating layers:
  1. Local transformer layer (with block causal mask).
  2. Cross-attention layer (pooling bytes → patch).
- Block causal mask: Each byte attends to at most wEw_EwE preceding bytes (can cross patch boundaries but not document boundaries).

Cross-Attention Block

- Inspired by Perceiver IO.



- Pooling function initializes p_j from corresponding bytes:

$$P_{0,j} = E_C(f_{bytes}(p_j))$$

- Updates across layers:

$$P^l = P^{l-1} + W_o \operatorname{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V$$

- Queries, Keys, Values defined as:

$$Q_j = W_q(P_j^{l-1}), \quad K_i = W_k(h_i^{l-1}), \quad V_i = W_v(h_i^{l-1})$$

- Masking: Each patch query attends **only to its bytes**.

✅ Output: Patch representations $P \in \mathbb{R}^{n_p \times h_G}$

3. Global Transformer (G)

- Operates on **patch representations** from encoder.
- Uses **causal attention mask** (autoregressive).
- Much deeper than local encoder/decoder.
- Outputs processed patch sequence representations ojo_joj.

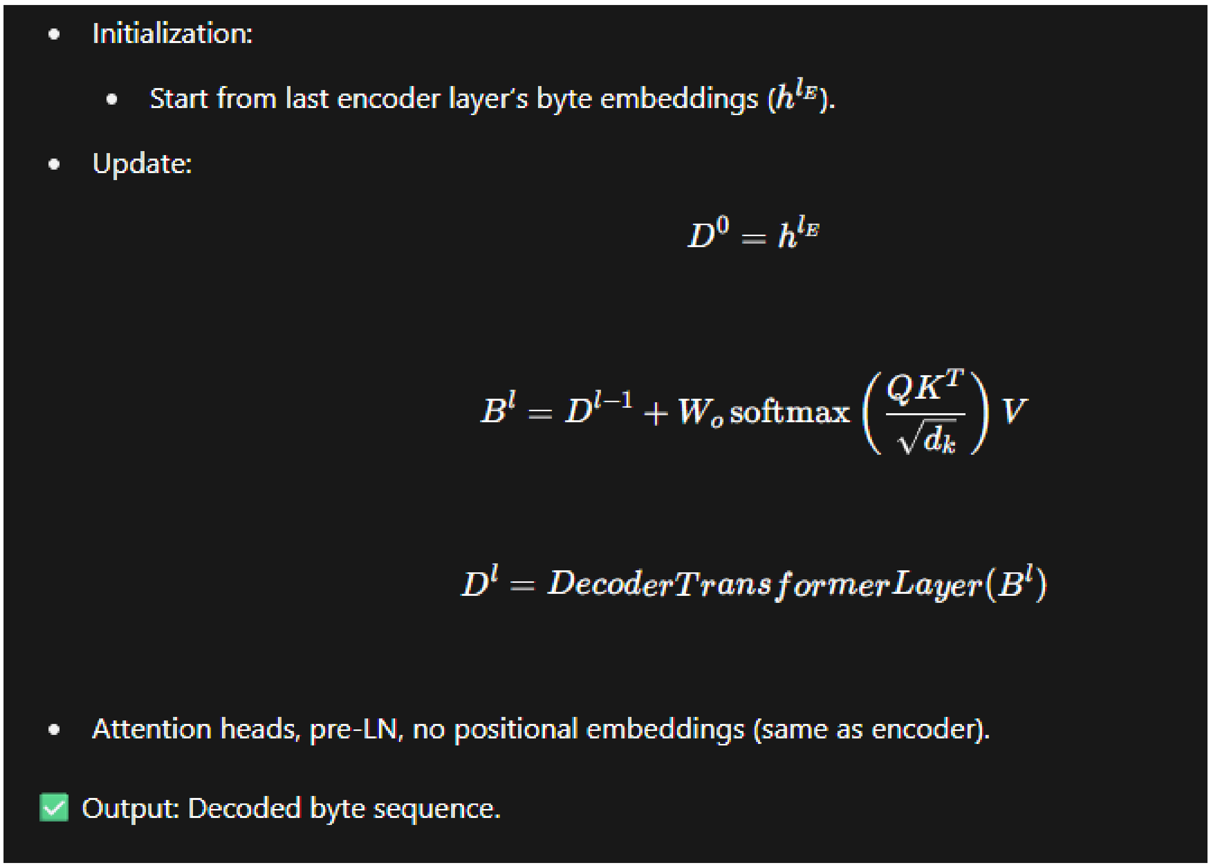
4. Local Decoder (D)

The **local decoder** reconstructs raw bytes from global patch representations.

- Lightweight transformer .
- Alternating layers of **cross-attention + transformer**.

4.1 Decoder Cross-Attention

- Roles **reversed** compared to encoder:
  - Queries = **bytes**.
  - Keys/Values = **patch representations**.
- Initialization:



- Initialization:
  - Start from last encoder layer's byte embeddings (h^{l_E}).
- Update:

$$D^0 = h^{l_E}$$

$$B^l = D^{l-1} + W_o \,\mathrm{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V$$

$$D^l = DecoderTransformerLayer(B^l)$$

- Attention heads, pre-LN, no positional embeddings (same as encoder).

✅ Output: Decoded byte sequence.

- Attention heads, pre-LN, no positional embeddings (same as encoder).

5. Key Intuitions

- Bytes → Local Encoder → Patches → Global Transformer → Patches → Local Decoder → Bytes.
- Cross-attention serves as the bridge:
  - Encoder: Pool bytes → patches.
  - Decoder: Expand patches → bytes.
- Hash n-grams provide efficient contextual embeddings.
- Global transformer only needs to model dependencies between patches (scales better than byte-level transformers).

References:

- https://drive.google.com/file/d/1BBYwr5botkuvI8CkjarIFNiN6B7uliWr/view
- https://drive.google.com/file/d/1B5BdO9FtmxTJiWwVJ3Wa-v3pqaRdbWMh/view
- https://arxiv.org/pdf/2412.09871