10417/10617

Language Modeling Part I: Tokenization, Training Objective and Positional Encodings

VIT Large (the original vision transformer)

- Convert input image to 16x16 total patches (total 256 vectors).
- For each patch, apply an embedding layer to map it to dimension 1024.
- Apply 24 transformer blocks, each transformer block has a one-hidden-layer MLP of size 1024 -> 4096 -> 1024.
    - Each transformer block as a Multi-Head Attention layer with 16 heads.
- Total 307M parameters (very small for a transformer).

We learnt a transformer model

- Step 1: Map the input to a sequence of vectors.
- Step 2: Go through an embedding layer that changes the dimension of the vectors.
- Step 3: Apply MHA (Multi-Head Attention) + Residual Link
- Step 4: Apply MLP + Residual Link
- Step 5: Repeat 3, 4 for multiple times.
- Step 6: Apply another layer that maps the final embedding layer to the output.

From Image to Language

For vision application, we can cut it into patches to get a sequence of input vectors.

The training objective is classification/reconstruction.

What about language?

Given a sentence "Alice likes to swim, so she asks Bob to go fishing with her and then she jumps into the water."

How to break it into a sequence of vectors?

What is the training objective?

We learnt a transformer model

Today, we focus on Steps 1 and 6 for language modeling.

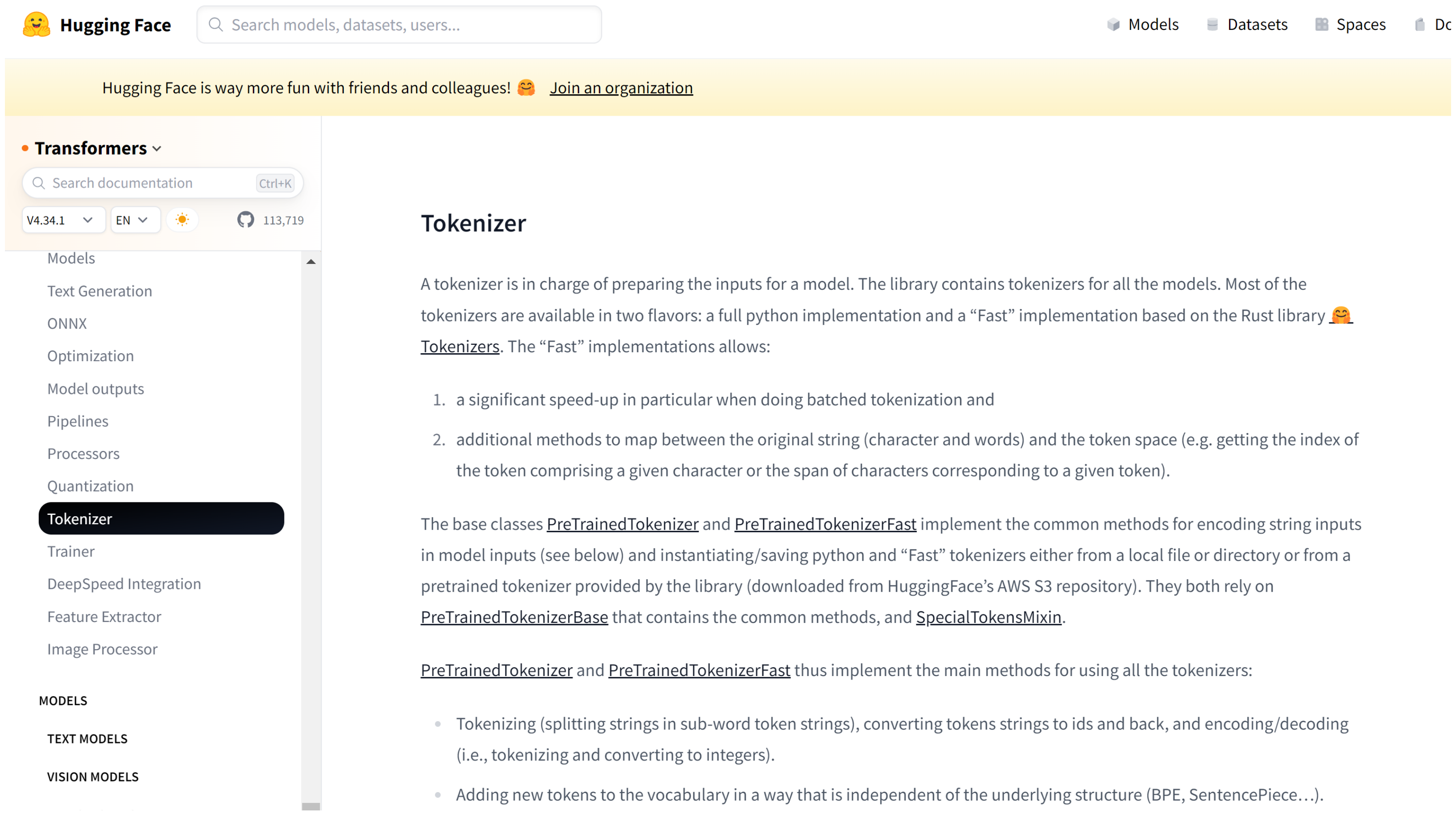
- Step 1: Map the input to a sequence of vectors.
- Step 6: Apply another layer that maps the final embedding layer to the output.

Step 1: How to map the input sentence to a sequence of vectors.

- Tokenizer + Positional Encoding.

Step 6: What is the training objective?

Tokenizer



Tokenizer

A tokenizer is in charge of preparing the inputs for a model. The library contains tokenizers for all the models. Most of the tokenizers are available in two flavors: a full python implementation and a "Fast" implementation based on the Rust library 🤗 Tokenizers. The "Fast" implementations allows:

1. a significant speed-up in particular when doing batched tokenization and
2. additional methods to map between the original string (character and words) and the token space (e.g. getting the index of the token comprising a given character or the span of characters corresponding to a given token).

The base classes PreTrainedTokenizer and PreTrainedTokenizerFast implement the common methods for encoding string inputs in model inputs (see below) and instantiating/saving python and "Fast" tokenizers either from a local file or directory or from a pretrained tokenizer provided by the library (downloaded from HuggingFace's AWS S3 repository). They both rely on PreTrainedTokenizerBase that contains the common methods, and SpecialTokensMixin.

PreTrainedTokenizer and PreTrainedTokenizerFast thus implement the main methods for using all the tokenizers:

- Tokenizing (splitting strings in sub-word token strings), converting tokens strings to ids and back, and encoding/decoding (i.e., tokenizing and converting to integers).
- Adding new tokens to the vocabulary in a way that is independent of the underlying structure (BPE, SentencePiece…).

Tokenizer

Tokenizer maps an input sentence (of any length, in any language) to a list (the length may vary depending on the sentence).

Example: GPT2-Tokenizer(“Alice likes to swim, so she asks BoBBB to go fishing with her and then she jumps into the water”) = [44484 7832 284 9422 11 523 673 7893 3248 15199 33 284 467 12478 351 607 290 788 673 18045 656 262 1660]

How are these numbers computed?

Tokenizer

A tokenizer is consistent of the following pieces:

A vocab.json, which maps subwords into integers

- "groupon": 14531, "\u0120jokes": 14532, "\u0120Benjamin": 14533, "\u0120Random": 14534, "frame": 14535, "\u0120Lions": 14536, "\u0120highlighted": 14537, ...

A pre-tokenization rule.

Pre-tokenization

Given a sentence like “Alice likes to swim, so she asks BoBBB to go fishing with her and then she jumps into the water.”

Pre-tokenization split the sentence into words according to a prescribed list of special symbols.

Typically, those are space + punctuations

- Period (.)
- Comma (,)
- Question mark (?)

…

Pre-tokenization

Given input sentence “Alice likes to swim, so she asks BoBBB to go fishing with her and then she jumps into the water.”

(1). Split according to punctuation. So it becomes

[Alice likes to swim] [,] [so she asks BoBBB to go fishing with her and then she jumps into the water] [.]

(2). Split each chunk according to space, but merge the space with the word after it. So the first chunk is splitted to

[Alice], [\u0120likes], [\u0120to], [\u0120swin]

\u0120 represents space.

Pre-tokenization

Split each chunk according to space, but merge the space with the word after it.

Special Notice: If we have things like [\u0120][,], we also merge it to [\u0120,]

Tokenization

“Alice likes to swim, so she asks BoBBB to go fishing with her and then she jumps into the water.” -- After pretokenization, we get a list

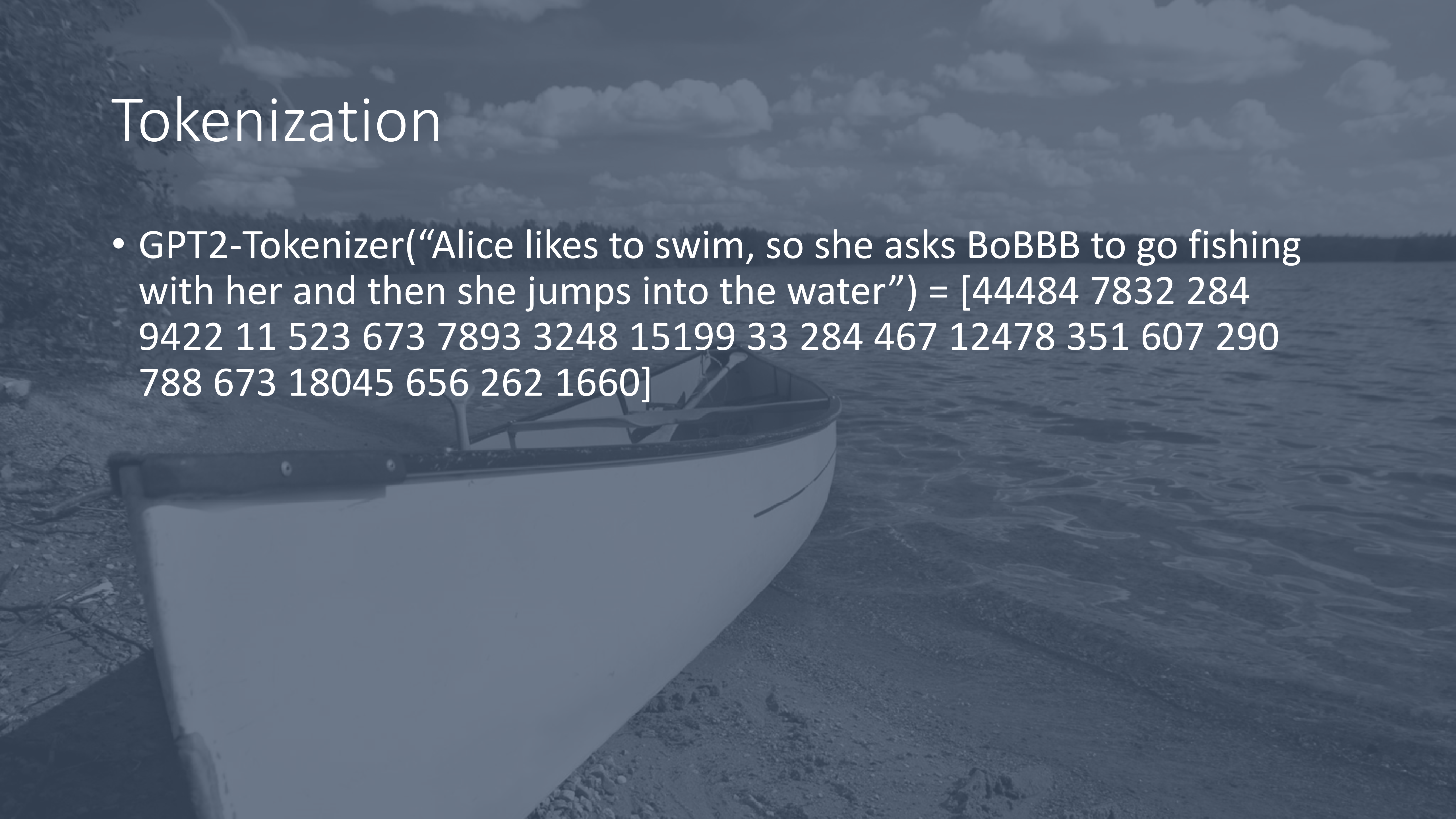
[Alice], [\u0120likes], [\u0120to], [\u0120swin], [,] [so], [\u0120she]…

For each word in the list, we find the longest subword in the vocab.json that matches the prefix of the word.

- “Alice”: 44484, “\u0120likes”: 7832, “\u0120to”: 284, …
- So we map them to [44484, 7832, 284, …]
- “\u0120BoBBB” is not in the vocab.json. Longest prefix is “\u0120Bo”: 3248
- Then we have “BBB”, “BBB” is not in the vocab.json, longest prefix is “BB”: 15199, then we have “B”: 33
- So we map “\u0120BoBBB” to [3248, 15199, 33]

Tokenization

- GPT2-Tokenizer(“Alice likes to swim, so she asks BoBBB to go fishing with her and then she jumps into the water”) = [44484 7832 284 9422 11 523 673 7893 3248 15199 33 284 467 12478 351 607 290 788 673 18045 656 262 1660]



Vocab.json

How was the vocab.json constructed?

On one hand, we want the prefix to be as long as possible, so we tokenize each chunk into fewer tokens.

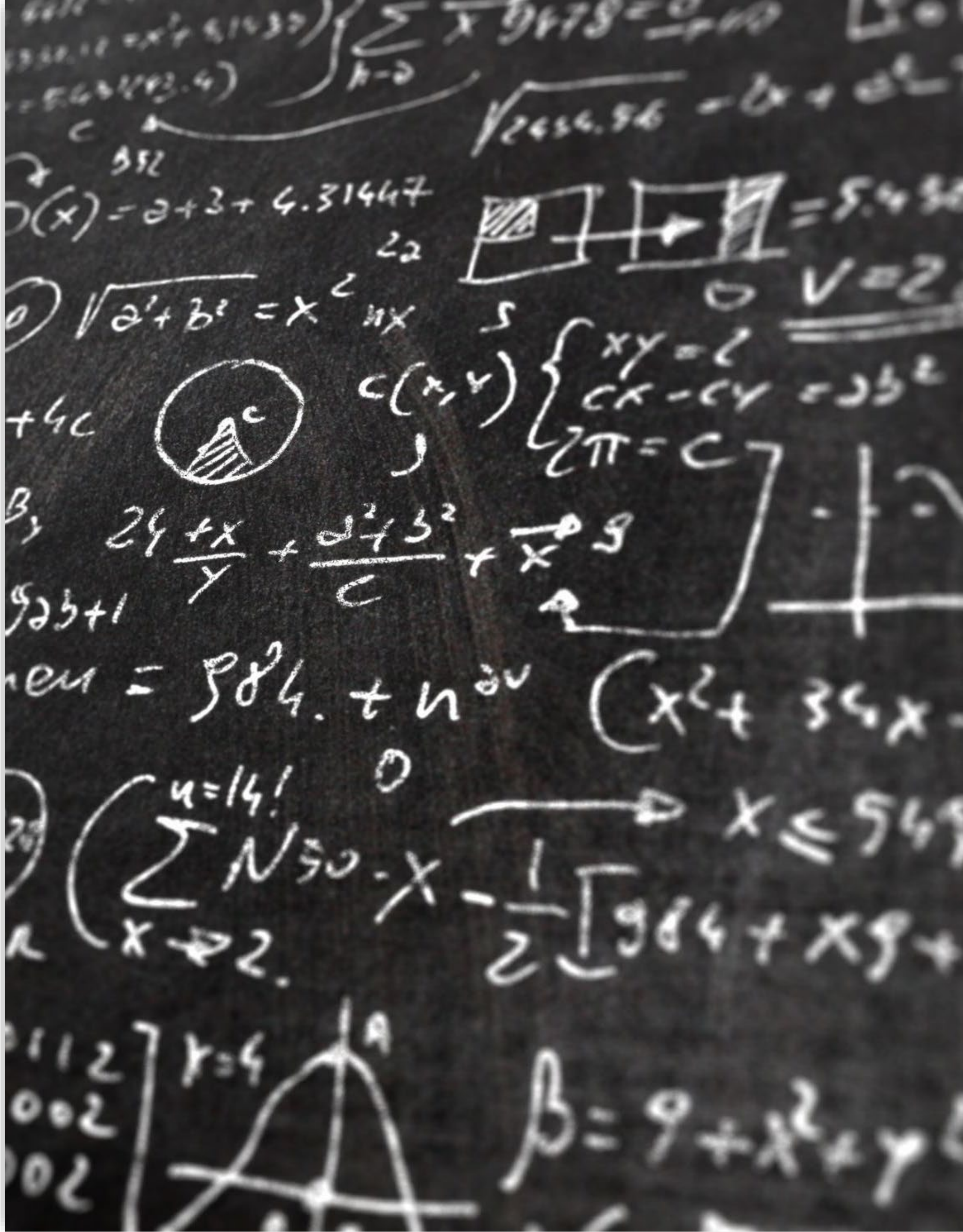
On the other hand, we don't want the vocab size to be too large (one token per every possible chunk, even things like [%WIUORQNCXKZJYTQ%VHKSJVASTXN%])

Given a training corpus, we want to solve the following problem:

- Minmize the length of the tokenization of the corpus, subject to the vocab size <= M.

Byte-Pair Encoding

- One of the algorithm to find the vocab.json given a training corpus
- Maintain a list of subwords.
- Initially, the list of subwords are all the bytes (chars like a, b, c, d, …)
- Loop:
  - Merge the two subwords that are most likely to appear next to each other in the corpus into one and create a new subword. Until the total number of subwords = M.
- [you][would][like][to][take][some][cake][or][pick][some][meat]
  - Merge k, e to ke
  - Merge m, e to me
  - Merge a, ke to ake
  - Merge o, me to ome
  - …



After tokenization

After tokenization, we map a sentence into a list of integers like [3260 11241 1634 11 356 3975 257 6827 656 257 1351 286 37014 220]

Each integer is called a token.

How do we train a language model on a giant list of integers (>1T tokens)?

Training objective

How do we train a language model on a giant list of integers (>1T tokens)?

Step 1: Split the giant list into chunks of length context_length (typically 512/1024/2048/4096/8192/16K/32K/128K).

Over each chunk, there are two objectives we can use:

- Mask Language Modeling Objective.
- Autoregressive Language Modeling Objective.

Mask Langage Modeling Objective

Given a chunk like X = [3260 11241 1634 11 356 3975 257 6827 656 257 1351 286 37014 220]

Randomly mask 15% of the tokens to <mask>, so we get things like

X’ = [3260 11241 1634 <mask> 356 3975 257 6827 656 <mask> 1351 286 37014 220]

Taking X’ as the input, the objective is to predict the label, which is X.

Alice is a <mask> student, she always scores 0 on her exams. On the other <mask>, Bob is a good <mask>.

Autoregressive Language Modeling Objective

- The more commonly used training objective is Autoregressive Language Modeling Objective.
- Given a list of integers X of length context_length, for every $i \in [context\_length - 1]$, we want to predict X[i + 1] given X[0:i+1]
- Alice is a horrible student, she always scores _ (should predict something corresponding to a low score).
- In this way, we can train a generative model like GPT-4.

Training Transformer Based Models

So the input is a list of integers, and the label is another (list of) integers.

How do we use a transformer model on this task?

Labels: (List of) Integers – This is fine, we can use cross entropy loss.

Input: List of integers.

Embedding Layer

Given a list of integers of length L.

We first map it into a list of one-hot vectors, each of dimension vocab_size.

Then, we apply an (trainable) embedding layer.

A linear function of vocab_size -> emb_dim to map each vector to emb_dim.

So we get L vectors, each of emb_dim.

Positional Encoding

So we get L vectors, each of emb_dim.

How can we keep the positional information of those vectors?

“Alice likes Bob” is totally different from “Bob likes Alice”.

We will use the so-called positional encoding.

Absolute Positional Encoding

- Given a list of L vectors $x_1, x_2, \dots, x_L$, each of dimension emb_dim.
- For each l, we add p_l (trainable positional encoding vector) to x_l and get another vector. Then we apply Layer Normalization on each vector.
- We feed the new list of vectors to the transformer.

Rotary Positional Encoding

- Given a list of L vectors $x_1, x_2, \dots, x_L$, each x is of dimension emb_dim
- Rotary positional encoding (not trainable) maps each x_l to $rot(x_l) = x_l'$:
  - $x_l'[2k] = x_l[2k]\cos(\theta_k l) \ - x_l[2k+1]\sin(\theta_k l)$
  - $x_l'[2k+1] = x_l[2k]\sin(\theta_k l) + x_l[2k+1]\cos(\theta_k l)$
  - If we view $z_{l,k} = \ x_l[2k] + i x_l[2k+1]$, then $z_{l,k}' = z_{l,k} e^{i\theta_k l}$.
  - For every $k \ \in [d_{rot}]$, where $2d_{rot}$ is the rotary embedding dimension (typically emb_dim//2).
- Where $\theta_k = \frac{1}{10000^{-k/d_{rot}}}$.

Rotary Positional Encoding

- $z'_{l,k} = z_{l,k}e^{i\theta_k l}$
  - We know that $< z'_{l,k} \quad , z'_{l',k} >$ = $z_{l,k}\ z_{l\prime,k}e^{i\theta_k(l\ -l')}$
- We apply Rotary Positional Encoding to the MHA layer directly.
- Instead of computing $\{< Qx_l, Kx_{l'} >\}_{l'\in[context\_length]}$ for the softmax, we compute $\{< rot(Qx_l), rot(Kx_{l'}) >\}_{l'\in[context\_length]}$
- Key observation: If $x_l = x_h, x_{l+p} = x_{h+p}$, then (shift invariant)
  - $< rot(Qx_l), rot\left(Kx_{l+p}\right) > =< rot(Qx_h), rot(Kx_{h+p}) >$