Training models in Deep Learning: A Formal basis

Samir El Karrat Moreno

December 12, 2025

Contents

1 Introduction

2 Input data

3 Tokenization

In DL we want to train our models by finding the minima of a chosen loss function. Let us denote $x^{(i)} \in \mathcal{D}_{\text{input}}$ be our input data $i = 1, \ldots, n$ in the input space $\mathcal{D}_{\text{input}}$ for, n being the amount of data that we are training our model with. Often times our input data $x^{(i)}$ can be highly dimensional, possibly reducing the context window of our models (see **??**). For example, CNNs work directly on the pixel tensor representing an image. However, for when having text as our input data several methods of compression have been developed, as we are going to describe in the following discussion.

In the case that we want to represent our data more condensedly, it is popular to take the following steps:

(1) **Tokenization**: It is a 'statistical' model for encoding input data following certain fixed rules and frequency optimizations to create condensed representations called **token sequences**. The latter consist of strings of **tokens**, statistically optimized encoded subdivisions of our original data. This process is bi-directional, and given any token sequence we can recover its input space representation.

During the training of a tokenizer, we look for efficient mappings between tokens in order to compress token sequences further. We denote the concept of **vocabulary** as the collection of all such mappings. It makes up for the entirety of the learnt parameters. However the **vocabulary dimension** is a hyper-parameter.

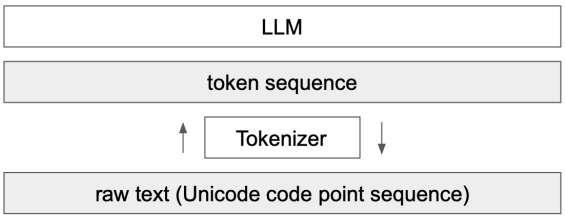

Figure 1: How tokenizer works

3.1 Byte Pair Encoding (BPE)

Byte Pair Encoding, or shortly BPE, is a basic example of text tokenization method. During its training stage, we first find the most frequent bigram (tuple of subsequent tokens) in the corresponding token sequence of our training text. Then we add it to our vocabulary, creating in this way a new **merge**. After this we replace all its occurrences with the correspnding mapping according to the vocabulary, i.e, its index. We perform the whole process iteratively, adding new merges until the desired vocabulary size is achieved. Note that merges containing tokens previously added to the vocabulary are a possiblity. A possible implementation could be as described in 1:

Algorithm 1 Byte Pair Encoding using UTF-8 (BPE) – Training

Require: Input text sequence T, vocabulary size V_{target}
Ensure: Learned merge rules M, Final token sequence S

Convert T into a sequence of UTF-8 bytes: $S \leftarrow [b_1, b_2, \ldots, b_n]$ where $b_i \in [0, 255]$
Initialize vocabulary size $V \leftarrow 256$
Initialize merge rules $M \leftarrow \emptyset$
while $V < V_{target}$ **do**
    $C \leftarrow \text{CountPairs}(S)$ ▷ Compute frequency of adjacent pairs
    if C is empty **then**
        break
    end if
    $(p_1, p_2) \leftarrow \arg\max_{(x,y)\in C} C[x, y]$ ▷ Find most frequent pair
    $idx \leftarrow V$
    $M[(p_1, p_2)] \leftarrow idx$ ▷ Record the new merge rule
    $S \leftarrow \text{Merge}(S, (p_1, p_2), idx)$ ▷ Replace all occurrences in sequence
    $V \leftarrow V + 1$
end while
return M, S

During BPE's inference stage, we encode/decode any token sequence from an inference text (the text we want to tokenize). For encoding, we simply just loop through each merge in our vocabulary, hierarchically from first merges to last merges, replacing any ocurrence of it with its assigned token. For decoding, we perform the same loop but in the opposite order, from last performed merges to the first ones, replacing them for their corresponding bigram according to the vocabulary. See a pseudocode implementation in 2.

Usually, we do not work with a raw string of text. We instead format it in a general way that allows us to uniformly treat basically any string of text we can ever encounter. This can be done in various forms, like for example formatting to bytes using an UTF-8 encoder. The reference library for it is OpenAI's official library for inference-only tokenization, **tiktoken**. We see a small snippet in Listing 1:

```python
import tiktoken

# GPT-2 (does not merge spaces)
enc = tiktoken.get_encoding("gpt2")
print(enc.encode("    hello world!!!"))
# output: [220, 220, 220, 23748, 995, 10185]
# GPT-4 (merges spaces)
enc = tiktoken.get_encoding("cl100k_base")
print(enc.encode("    hello world!!!"))
# output: [262, 24748, 1917, 12340]
```

Listing 1: Example usage of the tiktoken library, for GPT-2 and GPT-4 tokenizers.

However, this is not the most common library. Instead it is **sentencepiece**, a widely used BPE tokenizer that works directly on the text's Unicode *code points*, i.e., the characters of the text seen as a string. Note that exclusively building the vocabulary using code points (CPs from now on) and its mergings lacks generalization. CPs not seen during training would lack an entry in our vocabulary, a problem we would encounter using UTF-8 pre-encoding. Sentencepiece overcomes the challenge by providing a *character coverage*. It is a hyper-parameter that determines what percentage of the training CPs won't be treated as unseen. There are two approaches for dealing with unseen CPs:

(1) 'UNK' Mapping: Mapping them to the special token 'UNK'.

(2) Byte Fallback: Pre-encoding them using UTF-8, previously introducing bits in the vocabulary.

```python
import sentencepiece as spm

# the settings here are (best effort) those used for training Llama 2
options = dict(
  # input spec
  input="toy.txt",
  input_format="text",
  # output spec
```

```
  model_prefix="tok400", # output filename prefix
  # algorithm spec
  # BPE alg
  model_type="bpe",
  vocab_size=400,
  # normalization
  normalization_rule_name="identity", # ew, turn off normalization
  remove_extra_whitespaces=False,
  input_sentence_size=200000000, # max number of training sentences
  max_sentence_length=4192, # max number of bytes per sentence
  seed_sentencepiece_size=1000000,
  shuffle_input_sentence=True,
  # rare word treatment
  character_coverage=0.99995,
  byte_fallback=True,
  # merge rules
  split_digits=True,
  split_by_unicode_script=True,
  split_by_whitespace=True,
  split_by_number=True,
  max_sentencepiece_length=16,
  add_dummy_prefix=True,
  allow_whitespace_only_pieces=True,
  # special tokens
  unk_id=0, # the UNK token MUST exist
  bos_id=1, # the others are optional, set to -1 to turn off
  eos_id=2,
  pad_id=-1,
  # systems
  num_threads=os.cpu_count(), # use ~all system resources
)

spm.SentencePieceTrainer.train(**options)
```

Listing 2: Example usage of the sentencepiece library, reproducing LLama2 tokenizer

As we can see in Listing 2, we will be working with text files. We define different hyper-parameters: vocabulary size & character coverage with byte fallback. Moreover we set different merging rules for segmentation of the text and add special tokens.

3.1.1 Regex patterns & special tokens

After this talk about regex motivated by token issues. Continue with special tokens (useful for meta data, used in finetuning like adapting to browser)

4 Embeddings

https://web.stanford.edu/class/archive/cs/cs224n/cs224n.1246/

4.1 Text embeddings

In what follows we will overview NLP Theory.

We want to encapsulate de meaning of a word w by encoding it as an **embedding** or word representation $w \in \mathbb{R}^n$, for some relatively low dimensional n. Ideally, we want words that are close to each other in meaning, i.e., **semantically similar** words, to have corresponding vectors that are "close in distance".

First attempts of generating embeddings consist comprise the so called *one hot encoding* technique. Its approach is to assign words to each of the different components of the vectors in space. We are using **localist representations** for words, as the j-th component of any given vector represents an specific word, so we can say the vector e_j is that exact word. Following this reasoning, the canonical base of our space is formed by all the embeddings of our words in the vocabulary. Specifically, this means all the embeddings are orthogonal, meaning there is no point in encoding meaning as their dot product or distance. In general, any efforts of encoding meaning in this framework are not of interest inefficient to perform.

Current trends involve **distributed representations** for words. Embeddings take the form of **dense vectors**, which allows use to measure their semantic similarity via their dot product. Unlike one hot

encoding, we no longer encode words to components. We pay attention to the first efforts on creaing such representations using the now dated framework called *word2vec*. It will be a good intuition for understanding advanced embedding frameworks like BERT and ELMo.

4.1.1 Word2vec

Word2vec is a framework comprising models that create embeddings as distributed representations. The main assumption of this model is that words that appear often in similar contexts, i.e., often surrounded by a similar set of words around them, then they must be semantically similar. Despite slight differences, algorithmically the models share the same following structure:

We start with a **corpus** which is our main body text and fix a vocabulary with all of the words that appear in it (wihout repetition). Then, we iterate through each word with position t in our corpus w_t, considering a word context window of size s. Together with w_t, we conider its set of surrounding words $\{w_{t+j}, j = \pm 1, \pm 2, ..., \pm s\}$. Depending on the model,

The main two models are the *Continuous Bag Of Words* and *Skip-Gram*. They

They propose different objective functions.

Desribe idea behind word2vec Math: correspondance to MLE and the approach to model the probabilities

Information Theory: High Frequency = Low Information (ALSO IN THIS WEEK'S PAPER!)

4.2 Image embeddings (put after NN)

Focus on ViT and maybe compare to some CNN approach

training by distillation and finetuning

5 Datasets & Data Loading in Pytorch

6 Deep neural networks: mathematical formulation

6.1 Bias & Affine transformation

6.2 Activation functions

6.3 Logits & Softmax

7 Types of neural networks

8 Training

8.1 Mathematical motivation (via Statistics)

8.1.1 Maximum Likelihood Estimation

The **Maximum Likelihood Estimation** is an *statistical approach* to the following problem: given some IID (independent and identically distributed) from which we know what distribution they follow but not its parameters, how can we *estimate* them.

MLE solves this approach by maximizing the so called **parameter likelihood**. In mathematical terms, let $p(x^{(1)}, \dots, x^{(n)}|\theta)$ be the joint density function for our IID data samples $x^{(1)}, \dots, x^{(n)}$ and some parameters θ. Then this translates to finding $\arg\max_\theta p(x^{(1)}, \dots, x^{(n)}|\theta) = \prod_i^n p(x^{(i)}|\theta)$. Note how we are considering our function fixed on the data points as we are trying to find the "best fit" for them.

Computationally, to avoid numerical instability and to be able to conduct optimiziation, we typically aim to minimize quantities. We then can transform our problem: we can take $\text{NLL}(\theta) = -\log p(x^1, \dots, x^n|\theta)$. Finding the maximum parameter likelihood is then equivallent (due to logarithmic injectivity) to finding $\arg\min_\theta \text{NLL}(\theta)$.

Observation: This statistical concept is closely tied to typical the optimization process in DL, for which we use a loss function to update the weights of our models. The NLL is a bridge between statistics and DL.

8.2 Loss functions

8.2.1 Cross-Entropy Loss

Before we get into detail, let us borrow a very important concept from statistics that help us understand better what cross-entropy is:

Kullback-Leibler Divergence

The **Kullback-Leibler Divergence** $D_{KL}(P||Q)$ is a measure of the information lost when Q is used to approximate P. In more of a mathematical approach we say it is the "distance" between both probability distributions. The formula for discrete distributions is:

$$D_{KL}(P||Q) = \sum_{i \in \mathcal{X}} P(i) \log\left(\frac{P(i)}{Q(i)}\right)$$

Let us motivate it through a concrete example. Given the two discrete probability distributions on the sample space $\mathcal{X} = \{0, 1\}$:

- **True Distribution P (Fair Coin X):** $P(X = 0) = P(0) = p_1$ $P(X = 1) = P(1) = q_1$
- **Approximating Distribution Q (Biased Coin Y):** $P(Y = 0) = Q(0) = p_2$ $P(Y = 1) = Q(1) = q_2$

We want to compare for a same event the probability of it happening. This in general cannot be done as they need not share the same sample space, but none the less it is a good way to visualize it.

Consider that we flip the coin N times. Then our sample space is completely determined by strings of this same length of the type $101\ldots1$. For any given event $e \in \Omega$, we can define the ratio:

$$\frac{P(e)}{Q(e)} = \frac{p_1^n q_1^m}{p_2^n q_2^m}$$

Now we can take the log and divide by N:

$$\begin{aligned}
\frac{1}{N}\log\left(\frac{P(e)}{Q(e)}\right) &= \frac{1}{N}\log\left(\frac{p_1^n q_1^m}{p_2^n q_2^m}\right) \\
&= \frac{1}{N}\left[\log(p_1^n) + \log(q_1^m) - \log(p_2^n) - \log(q_2^m)\right] \\
&= \frac{1}{N}\left[n\log(p_1) + m\log(q_1) - n\log(p_2) - m\log(q_2)\right] \\
&= \frac{n}{N}\left[\log(p_1) - \log(p_2)\right] + \frac{m}{N}\left[\log(q_1) - \log(q_2)\right] \\
&= \frac{n}{N}\log\left(\frac{p_1}{p_2}\right) + \frac{m}{N}\log\left(\frac{q_1}{q_2}\right)
\end{aligned}$$

In the limit as $N \to \infty$, by the Law of Large Numbers, the frequency $\frac{n}{N}$ converges in probability to the true probability $P(1) = p_1$, and $\frac{m}{N}$ converges to $P(0) = q_1$.

Thus, taking the expectation over the sequence of N events, $\mathbb{E}_P\left[\frac{1}{N}\log\left(\frac{P(e)}{Q(e)}\right)\right]$, we find that the limit of this average log-ratio is the Kullback-Leibler Divergence:

$$\lim_{N\to\infty} \mathbb{E}_P\left[\frac{1}{N}\log\left(\frac{P(e)}{Q(e)}\right)\right] = p_1\log\left(\frac{p_1}{p_2}\right) + q_1\log\left(\frac{q_1}{q_2}\right)$$

This is the general formula for the KL-Divergence between two Bernoulli distributions:

$$D_{KL}(P||Q) = \sum_{i \in \{0,1\}} P(i) \log\left(\frac{P(i)}{Q(i)}\right)$$

where $P(1) = p_1, P(0) = q_1$, $Q(1) = p_2, Q(0) = q_2$.

8.3 Regularization

8.4 Optimization methods

9 Extra material

9.1 BPE

Here I include pseudocode for some of the functions in Algorithm 1, together with a python implementation of it.

Algorithm 2 Further functions for BPE Training

procedure CountPairs(S)
  $Counts \leftarrow$ empty map
  for $i \leftarrow 0$ **to** $|S| - 2$ **do**
    $pair \leftarrow (S[i], S[i+1])$
    $Counts[pair] \leftarrow Counts[pair] + 1$
  end for
  return $Counts$
end procedure
procedure Merge($S, pair, new\_token$)
  $S_{new} \leftarrow []$
  $i \leftarrow 0$
  while $i < |S|$ **do**
    if $i < |S| - 1$ **and** $S[i] == pair[0]$ **and** $S[i+1] == pair[1]$ **then**
      Append new_token to S_{new}
      $i \leftarrow i + 2$
    else
      Append $S[i]$ to S_{new}
      $i \leftarrow i + 1$
    end if
  end while
  return S_{new}
end procedure

```python
    @staticmethod
    def get_stats(ids): # Get the most frequent pair in the byte-encoded text
            counts = {}
            for pair in zip(ids, ids[1:]):
                counts[pair] = counts.get(pair, 0) + 1
            return counts # Dictionary of the form "pair -> times_appeared"
    @staticmethod
    def merge(ids, pair, idx): # Update the token sequence according to a new merge
        newids = []
        i = 0
        while i < len(ids):
            if i < len(ids) - 1 and ids[i] == pair[0] and ids[i+1] == pair[1]:
                newids.append(idx)
                i += 2
            else:
                newids.append(ids[i])
                i += 1
        return newids

    def train(self, text, verbose=False):
        tokens = text.encode("utf-8") # raw bytes
        tokens = list(map(int, tokens)) # convert to a list of integers in range 0..255
    for convenience

        # --
        num_merges = self.vocab_size - 256
        ids = list(tokens) # copy so we don't destroy the original list

        merges = {} # (int, int) -> int
        for i in range(num_merges):
            stats = self.get_stats(ids)
            pair = max(stats, key=stats.get)
            idx = 256 + i
            if verbose:
```

```
                print(f"merging {pair} into a new token {idx}")
            ids = self.merge(ids, pair, idx)
            merges[pair] = idx
        self.merges= merges

```

Listing 3: An implementation of BPE in Python (adapted code from Andrej Karpathy's github)

10 Annex

10.1 Pseudo-likelihood