STA5007 Assignment 1

Li Xuanran
12312110

October 24, 2025

1 Q1

Tokenization is an important step in modern NLP pipelines. In this task,

1. (3 pts) Explain the Byte Pair Encoding (BPE) algorithm. Then illustrate with code to show how BPE works.
2. (2 pts) Write code to train a BPE model (vocabulary size = 10000) and apply the tokenizer to the dataset. You can use the Subword-NMT toolkit. Please refer to link for the training data. You need to write python code to extract the data from the `conversations` data in the jsonl file and follow the ChatML format.

Note you can refer to the following code MinBPE and Subword-NMT, as well as the original paper Byte pair encoding. You can also ask for help from GPTs, however, you are responsible for checking the correctness of the responses.

Solution.

1. BPE algorithm steps:
   (a) Initialize the vocabulary from basic tokens (i.e. letter or other kind of chars).
   (b) Calculate the frequencies of every nearby token-pairs in the corpus.
   (c) Merge the adjacent tokens which has the highest frequency, and add it to the vocabulary.
   (d) Follow the step repeatedly, until we reach the limit number of tokens.

   The following main functions show how it works.

```python
# Transfer every text into a list of token with '</w>'.
def extract_frequencies(texts):
    tokens = Counter()
    for text in texts:
        text = ' '.join(text) + ' </w>'
        tokens.update(text.split())
    return tokens

# Calculate the frequency of adjacent tokens.
```

```python
def frequency_of_pairs(frequencies):
    pairs = Counter()
    for token, freq in frequencies.items():
        symbols = token.split()
        for i in range(len(symbols) - 1):
            pairs[symbols[i], symbols[i + 1]] += freq
    return pairs

# Merge the pair and add it into vocab.
def merge_vocab(pair, vocab):
    bigram = re.escape(' '.join(pair))
    merged = ''.join(pair)
    new_vocab = Counter()
    for token in vocab:
        new_token = token.replace(bigram, merged)
        new_vocab[new_token] = vocab[token]
    return new_vocab

# Extract the original frenquencies, then merge the most-frequency pair iteratively.
def encode_with_bpe(texts, num_merges):
    vocab = extract_frequencies(texts)
    for _ in range(num_merges):
        pairs = frequency_of_pairs(vocab)
        if not pairs:
            break
        most_frequent = pairs.most_common(1)[0][0]
        vocab = merge_vocab(most_frequent, vocab)
    return vocab
```

Listing 1: BPE Algorithm

2. Simply run `Q1_BPE.py`. The text following ChatML format is stored in `chatml_data.txt`, and the token-frequency pairs are stored in `bpe.codes`.

□

2 Q2

(3 pts) Complete the missing parts of `transformer_model.py`. Search for **TODO** in the code. There are 5 **TODOs** that correspond to key functions. Note that you can ask for help from GPTs, however, you are responsible for checking the correctness of the responses.

Solution.

Positional encoding: We use sin and cos function to set positional vectors:

$$\mathrm{PE}(\mathrm{pos}, 2i) = \sin\left(\frac{\mathrm{pos}}{10000^{\frac{2i}{d_m}}}\right),$$
$$\mathrm{PE}(\mathrm{pos}, 2i+1) = \cos\left(\frac{\mathrm{pos}}{10000^{\frac{2i}{d_m}}}\right),$$

where pos is the position, i is the dimension, d_m is the hidden state dimension of the model.

Then we add PE$(\cdot)$ to embedding vectors and then do drop-out, which can let the model identify the order information.

```python
class PositionalEncoding(nn.Module):
    def __init__(self, embed_dim: int, max_len: int, dropout: float):
        super().__init__()
        self.dropout = nn.Dropout(dropout)

        pe = torch.zeros(max_len, embed_dim)
        pos = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, embed_dim, 2).float() * (-
    math.log(10000.0) / embed_dim))

        pe[:, 0::2] = torch.sin(pos * div_term)
        pe[:, 1::2] = torch.cos(pos * div_term)
        pe = pe.unsqueeze(0)
        self.register_buffer('pe', pe)

    def forward(self, x):
        # TODO: Add positional encoding self.pe to input x, then apply
    dropout
        x = x + self.pe[:, :x.size(1)]
        return self.dropout(x)
```

Listing 2: Positional encoding

In a FFL, for every token $\mathbf{x}_i$, we have

$$\mathrm{FFN}(\mathbf{x}_i) = \mathrm{Dropout}(\mathrm{ReLU}(\mathbf{x}_i\mathbf{W}_1 + \mathbf{b}_1))\mathbf{W}_2 + \mathbf{b}_2.$$

```python
class FeedForward(nn.Module):
    def __init__(self, embed_dim: int, hidden_dim: int, dropout: float):
        super().__init__()
        self.fc1 = nn.Linear(embed_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, embed_dim)
        self.relu = nn.ReLU()
        self.drop = nn.Dropout(dropout)

    def forward(self, x):
        # TODO: Implement two linear layers with ReLU and Dropout
        return self.fc2(self.drop(self.relu(self.fc1(x))))
```

Listing 3: Feed forward

In multi-head self-attention, we first calculate the attention score α:

$$\alpha = \frac{QK^{\mathrm{T}}}{\sqrt{d_k}},$$

then mask out all the coordination that $\texttt{mask} = 0$, then

$$\mathrm{Attention}(Q, K, V) = \mathrm{Softmax}(\alpha)V.$$

```python
class MultiHeadAttention(nn.Module):
    def __init__(self, embed_dim: int, num_heads: int, dropout: float):
        super().__init__()
        assert embed_dim % num_heads == 0
        self.d_k = embed_dim // num_heads
        self.num_heads = num_heads
        self.q_linear = nn.Linear(embed_dim, embed_dim)
        self.k_linear = nn.Linear(embed_dim, embed_dim)
        self.v_linear = nn.Linear(embed_dim, embed_dim)
        self.out_linear = nn.Linear(embed_dim, embed_dim)
        self.drop = nn.Dropout(dropout)

    def forward(self, q, k, v, mask=None):
        B = q.size(0)
        Q = self.q_linear(q).view(B, -1, self.num_heads, self.d_k).transpose(1, 2)
        K = self.k_linear(k).view(B, -1, self.num_heads, self.d_k).transpose(1, 2)
        V = self.v_linear(v).view(B, -1, self.num_heads, self.d_k).transpose(1, 2)

        # TODO: Compute attention scores
        scores = Q @ K.transpose(-2, -1) / math.sqrt(self.d_k)
        if mask is not None:
            scores = scores.masked_fill(mask == 0, -1e9)
        attn = torch.softmax(scores, dim=-1)
        attn = self.drop(attn)
        out = attn @ V
        out = out.transpose(1, 2).contiguous().view(B, -1, self.num_heads * self.d_k)
        return self.out_linear(out)
```

Listing 4: Multi-head attention

Do LayerNorm to the input, then going through the sub-layer (Attention or FFN), and then add the residual:

$$\mathbf{x}^l = \mathbf{x}^{l-1} + \text{block}(\mathbf{x}^{l-1}).$$

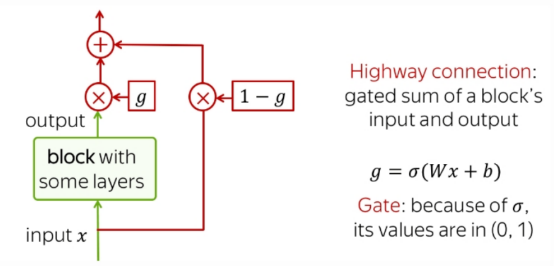

Figure 1: Residual connection

```python
class Residual(nn.Module):
```

```
    def __init__(self, dropout: float):
        super().__init__()
        self.norm = LayerNorm()
        self.drop = nn.Dropout(dropout)

    def forward(self, x, sublayer):
        # TODO: Apply LayerNorm on sublayer(x), then add residual connection with input x
        return x + self.drop(sublayer(self.norm(x)))
```

Listing 5: Residual connection

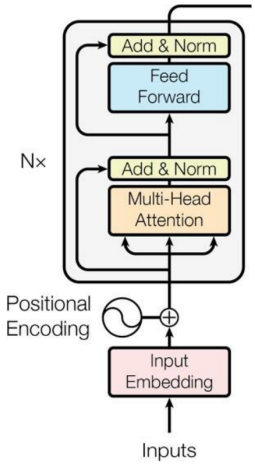

Figure 2: Encoder

According to Figure 2, every Encoder layer has two sub-layers: Layer 1 is self-attention with residual connection, Layer 2 is FFN with residual connection.

```
class EncoderLayer(nn.Module):
    def __init__(self, embed_dim: int, num_heads: int, hidden_dim: int, dropout: float):
        super().__init__()
        self.self_attn = MultiHeadAttention(embed_dim, num_heads, dropout)
        self.ffn = FeedForward(embed_dim, hidden_dim, dropout)
        self.res_layers = nn.ModuleList([Residual(dropout) for _ in range(2)])

    def forward(self, x, mask):
        # TODO: Apply self-attention + residual, then feed-forward + residual
        x = self.res_layers[0](x, lambda x: self.self_attn(x, x, x, mask))
        x = self.res_layers[1](x, self.ffn)
        return x
```

Listing 6: Encoder layer

□

3 Q3

ImageBind-LLM: Multi-modality Instruction Tuning is a paper about large language model. The Figure 3 shows a bind network which consists of some feedforward network.

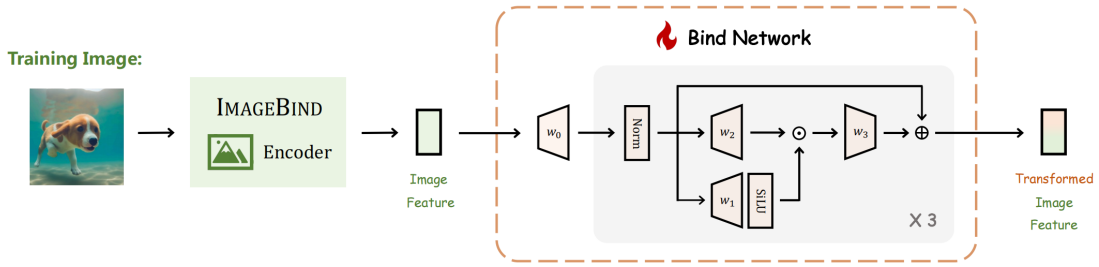

Fig. 3 Details of the Bind Network. Referring to the Feed-Forward Network (FFN) in LLaMA [11], we adopt cascaded blocks of RMSNorm [43], SiLU activation functions [44], and residual connections [45]. This aims to align the image feature from ImageBind [12] with LLaMA's word embeddings.

Figure 3: Details of the Bind Network

(2 pts) Write the pytorch code for the bind network, the input is `image_feature` and output is `transformered_image_feature`. Define the `__init__()` and `forward()` function. Note that the bind network has three blocks ($\times 3$).

Solution. According to the paper, the main architecture would be:

Input: $F_I \in \mathbb{R}^{1\times C_I}$: global image feature extracted by the ImageBind encoder.

Output: $T_I \in \mathbb{R}^{1\times C}$: transformed image feature aligned to LLaMA's feature dimension C

Structure: Linear projection ($F_I' = F_I W_0$), projecting ($C_I \to C$). Then pass through 3 repeated blocks, each composed of:

1. RMSNorm
2. Two linear projections (w_1, w_2, w_3)
3. SiLU activation
4. Residual connection

Computation for each block i:

$$F_I^{i+1} = F_I^i + (F_I^i W_2 \cdot \mathrm{SiLU}(F_I^i W_1))W_3$$

where $W_1, W_2 \in \mathbb{R}^{C\times C_h}$, $W_3 \in \mathbb{R}^{C_h\times C}$.

The `__init__()` and `forward()` function are as below.

```
class BindNetwork(nn.Module):
    def __init__(self, input_dim, output_dim, hidden_dim=None, num_blocks
    =3):
        super(BindNetwork, self).__init__()
```

```
        self.input_dim = input_dim
        self.output_dim = output_dim
        self.hidden_dim = hidden_dim or output_dim * 4
        self.num_blocks = num_blocks

        self.proj = nn.Linear(input_dim, output_dim)
        self.blocks = nn.ModuleList([
            nn.ModuleDict({
                "norm": nn.RMSNorm(output_dim),
                "w1": nn.Linear(output_dim, self.hidden_dim),
                "w2": nn.Linear(output_dim, self.hidden_dim),
                "w3": nn.Linear(self.hidden_dim, output_dim)
            })
            for _ in range(num_blocks)
        ])

    def forward(self, image_feature):
        x = self.proj(image_feature)
        for blk in self.blocks:
            h = blk["norm"](x)
            h1 = blk["w1"](h)
            h2 = blk["w2"](h)
            h_mul = h2 * F.silu(h1)
            h3 = blk["w3"](h_mul)
            x = x + h3
        return x
```

Listing 7: Bind Network

□