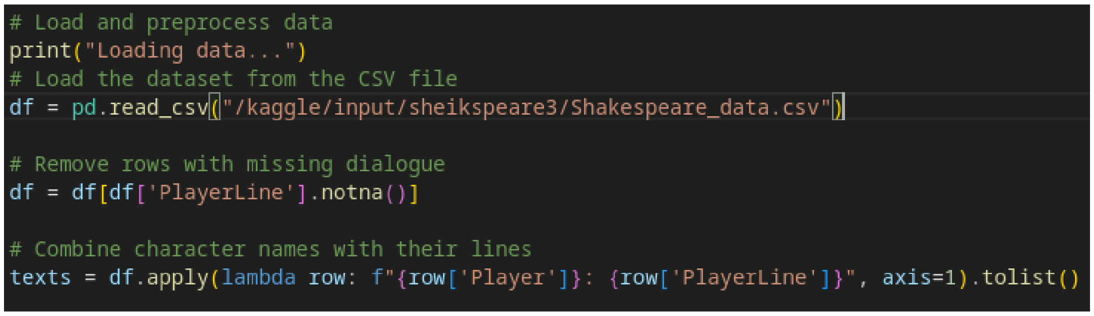
Daniel-Cristian-Marian Țăpuși

Neural Networks - HW3 report

Data organization and preprocessing:

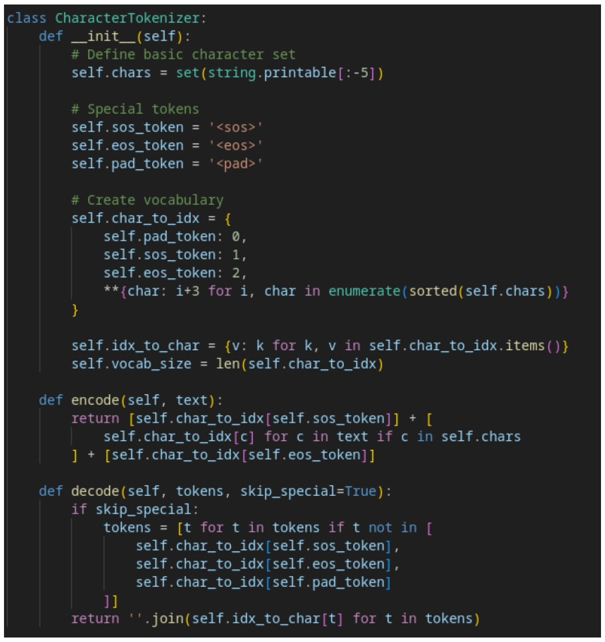
To create the list of playlines, I began by importing the data from the CSV file containing all of Shakespeare's novels playlines. After loading the data, I removed any rows with missing conversation to ensure that the dataset was clean and consistent. Each line of conversation was afterwards paired with the corresponding character's name, resulting in entries such as HAMLET: To be or not to be. This approach allows the model to easily understand not just the text but also who is speaking, providing context to the training data. The end result was a lengthy list of these formatted lines, suitable for tokenization and training.



```python
# Load and preprocess data
print("Loading data...")
# Load the dataset from the CSV file
df = pd.read_csv("/kaggle/input/sheikspeare3/Shakespeare_data.csv")

# Remove rows with missing dialogue
df = df[df['PlayerLine'].notna()]

# Combine character names with their lines
texts = df.apply(lambda row: f"{row['Player']}: {row['PlayerLine']}", axis=1).tolist()
```

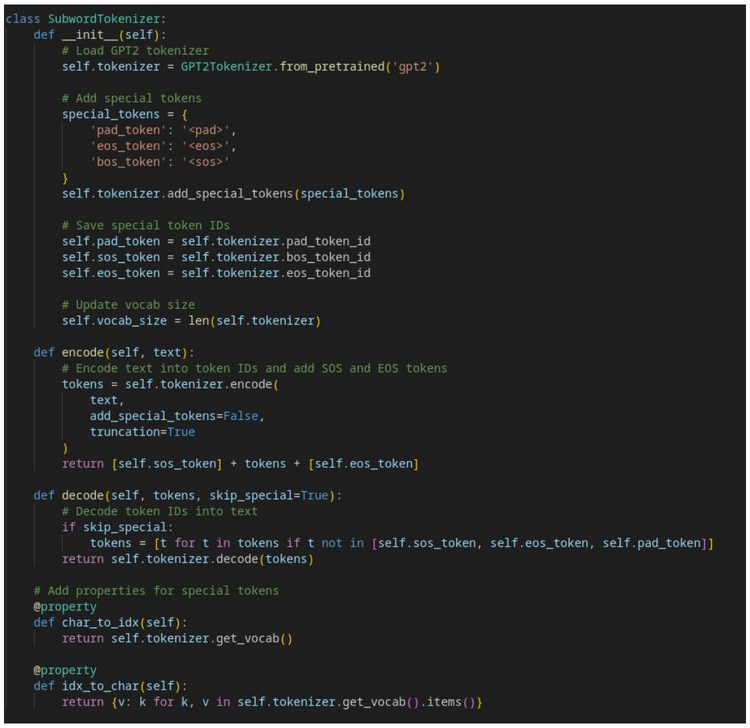
I didn't perform any further cleaning or normalization, like lowercasing or removing punctuation, because the tokenizers are designed to handle raw text. Special tokens like <sos> (start of sequence) and <eos> (end of sequence) were added during tokenization to mark the beginning and end of each line.

For the character-level tokenizer I created a vocabulary of all printable characters and added special tokens like <sos>, <eos>, and <pad>. For example, the text HAMLET: To be is tokenized as ['<sos>', 'H', 'A', 'M', 'L', 'E', 'T', ':', ' ', 'T', 'o', ' ', 'b', 'e', '<eos>']. While this approach is straightforward, it results in longer sequences compared to subword tokenization.



```python
class CharacterTokenizer:
    def __init__(self):
        # Define basic character set
        self.chars = set(string.printable[:-5])

        # Special tokens
        self.sos_token = '<sos>'
        self.eos_token = '<eos>'
        self.pad_token = '<pad>'

        # Create vocabulary
        self.char_to_idx = {
            self.pad_token: 0,
            self.sos_token: 1,
            self.eos_token: 2,
            **{char: i+3 for i, char in enumerate(sorted(self.chars))}
        }

        self.idx_to_char = {v: k for k, v in self.char_to_idx.items()}
        self.vocab_size = len(self.char_to_idx)

    def encode(self, text):
        return [self.char_to_idx[self.sos_token]] + [
            self.char_to_idx[c] for c in text if c in self.chars
        ] + [self.char_to_idx[self.eos_token]]

    def decode(self, tokens, skip_special=True):
        if skip_special:
            tokens = [t for t in tokens if t not in [
                self.char_to_idx[self.sos_token],
                self.char_to_idx[self.eos_token],
                self.char_to_idx[self.pad_token]
            ]]
        return ''.join(self.idx_to_char[t] for t in tokens)
```

The subword tokenizer uses the GPT-2 tokenizer from the Hugging Face library, which breaks text into smaller subword units (e.g., "playing" becomes ['play', 'ing']). I added special tokens like <sos>, <eos>, and <pad> to the tokenizer's vocabulary. This method is more efficient than character-level tokenization because it reduces sequence length while still preserving the meaning of the text.



```python
class SubwordTokenizer:
    def __init__(self):
        # Load GPT2 tokenizer
        self.tokenizer = GPT2Tokenizer.from_pretrained('gpt2')

        # Add special tokens
        special_tokens = {
            'pad_token': '<pad>',
            'eos_token': '<eos>',
            'bos_token': '<sos>'
        }
        self.tokenizer.add_special_tokens(special_tokens)

        # Save special token IDs
        self.pad_token = self.tokenizer.pad_token_id
        self.sos_token = self.tokenizer.bos_token_id
        self.eos_token = self.tokenizer.eos_token_id

        # Update vocab size
        self.vocab_size = len(self.tokenizer)

    def encode(self, text):
        # Encode text into token IDs and add SOS and EOS tokens
        tokens = self.tokenizer.encode(
            text,
            add_special_tokens=False,
            truncation=True
        )
        return [self.sos_token] + tokens + [self.eos_token]

    def decode(self, tokens, skip_special=True):
        # Decode token IDs into text
        if skip_special:
            tokens = [t for t in tokens if t not in [self.sos_token, self.eos_token, self.pad_token]]
        return self.tokenizer.decode(tokens)

    # Add properties for special tokens
    @property
    def char_to_idx(self):
        return self.tokenizer.get_vocab()

    @property
    def idx_to_char(self):
        return {v: k for k, v in self.tokenizer.get_vocab().items()}
```

Models’ architecture:

1. **Small Model with Character-Level Tokenization**

   Model Parameters:

   d_model = 384, num_heads = 6, num_layers = 6, d_ff = 1536, dropout = 0.2, max_seq_length = 256

   Training Loop Parameters:

   batch_size = 8, accumulation_steps = 4, learning_rate = 1e-3, num_epochs = 30, val_split = 0.1. For the optimizer AdamW with betas = (0.9, 0.98), eps = 1e-6 and weight_decay = 0.005 were used. The scheduler OneCycleLR with max_lr = 1e-3, pct_start = 0.2, div_factor = 25, final_div_factor = 1000 and anneal_strategy = ’cos’.

2. **Small Model with Subword-Level Tokenization**

   Model Parameters:

   d_model = 384, num_heads = 6, num_layers = 6, d_ff = 1536, dropout = 0.2, max_seq_length = 256.

   Training Loop Parameters:

   batch_size = 8, accumulation_steps = 4, learning_rate = 1e-3, num_epochs = 30, val_split = 0.1. For the optimizer AdamW with betas = (0.9, 0.98), eps = 1e-6 and weight_decay = 0.005 were used. The scheduler OneCycleLR with max_lr = 1e-3, pct_start = 0.2, div_factor = 25, final_div_factor = 1000 and anneal_strategy = ’cos’.

3. **Large Model with Character -Level Tokenization**

   Model Parameters:

   d_model = 1024, num_heads = 16, num_layers = 16, d_ff = 4096, dropout = 0.2, max_seq_length = 512.

Training Loop Parameters:

batch_size = 8, accumulation_steps = 2, learning_rate = 3e-4, num_epochs = 10, val_split = 0.1. For the optimizer AdamW with betas = (0.9, 0.98), eps = 1e-6 and weight_decay = 0.1 were used. The scheduler OneCycleLR with max_lr = 3e-4, pct_start = 0.15, div_factor = 20, final_div_factor = 50 and anneal_strategy = ’cos’.

4. **Large Model with Subword-Level Tokenization**

   Model Parameters:

   d_model = 1024, num_heads = 16, num_layers = 16, d_ff = 4096, dropout = 0.2, max_seq_length = 512.

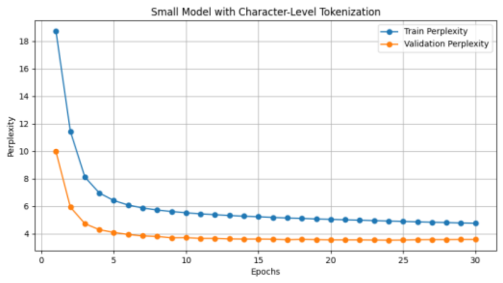
   Training Loop Parameters:

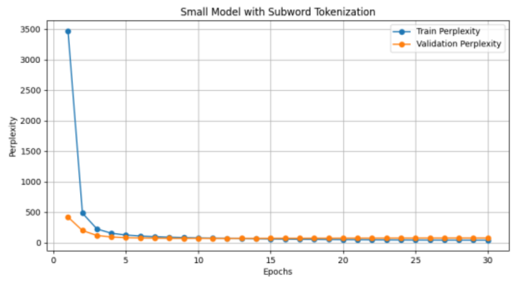
   batch_size = 8, accumulation_steps = 2, learning_rate = 3e-4, num_epochs = 10, val_split = 0.1. For the optimizer AdamW with betas = (0.9, 0.98), eps = 1e-6 and weight_decay = 0.1 were used. The scheduler OneCycleLR with max_lr = 3e-4, pct_start = 0.15, div_factor = 20, final_div_factor = 50 and anneal_strategy = ’cos’.

Results:

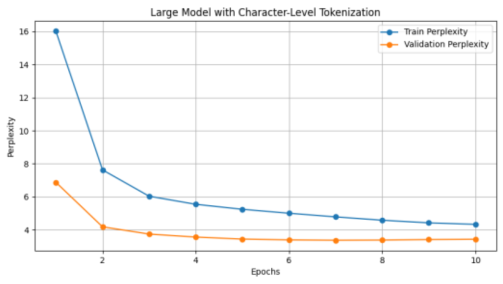
Strangely, the validation perplexity is consistently smaller than the training perplexity across all setups. This could be due to the training set containing more diverse and challenging examples, while the validation set consists of cleaner, more representative samples. It may also indicate that the models generalize well to unseen data, as they perform better on the validation set than on the training set.

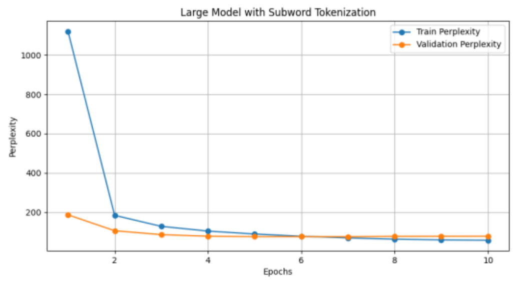
The small model with character-level tokenization started with a training perplexity of 18.76 and validation perplexity of 9.99, improving to 4.77 and 3.59, respectively, after 30 epochs. The small model with subword tokenization began with much higher perplexity (training: 3471.71, validation: 421.49) due to the larger vocabulary but converged to 43.27 and 78.30 by the end of training.





The large model with character-level tokenization achieved lower perplexity faster, starting at 16.01 (training) and 6.88 (validation) and reaching 4.34 and 3.44 after just 10 epochs. Finally, the large model with subword tokenization also started high (training: 1120.70, validation: 186.56) but improved to 57.37 and 77.69. Overall, character-level tokenization led to faster convergence and lower perplexity, while subword tokenization required more epochs to achieve comparable results due to its complexity.


Large Model with Character-Level Tokenization
Train Perplexity
Validation Perplexity
Perplexity
Epochs


Large Model with Subword Tokenization
Train Perplexity
Validation Perplexity
Perplexity
Epochs