CSE556
Natural Language Processing
Winter 2025

Report for Assignment 1

Shamik Sinha - 2022468
Shrutya Chawla - 2022487
Vansh Yadav - 2022559

Group 86

Task 1 - Implementing a WordPiece Tokenizer

WordPiece tokenization works by iteratively merging frequent character pairs into a single token. This allows the model to learn subword units that can effectively represent unknown or rare words. Unlike traditional word-level tokenization, it focuses on subword units, making it robust for languages with rich morphology and helps with out-of-vocabulary tokens.

Implementation Details

The implementation follows the main steps of **text preprocessing, vocabulary construction, pair merging, and tokenization** of input text.

1. **Preprocessing:**

   The text is first **converted to lowercase** and cleaned by **removing non-alphabetical characters** using regular expressions. The sentences were split into words and put in a list. The result is a list of words ready for tokenization.

2. **Vocabulary Construction:**

   Begins by processing a given corpus of text. Each word's frequency is recorded, and the characters are split into individual units.

   Special tokens such as "[PAD]" and "[UNK]" are added to handle padding and unknown words.

   An iterative approach is used to merge the most frequent character pairs into new subword tokens until the desired vocabulary size is reached. The merging process is guided by a frequency-based score computed for each pair of adjacent subword tokens.

3. **Tokenization:**

   Each **input word is split into the longest possible subword units** found in the vocabulary.

   If no valid subword is found, the word is replaced by the "[UNK]" token. The final tokenized output is a list of subword tokens.

4. **Helper Methods:**
    a. **Pair Score Computation:** computes a score for each possible character pair based on their frequencies and the overall frequency of the word in the corpus.
    b. **Best Pair Merging:** once the best pair is selected, it is merged into a new token, and the word splits are updated.
    c. **Subword Tokenization:** handles the tokenization of a word by matching the longest prefix in the vocabulary. Any leftover part of the word is prepended with "##" to indicate it's a subword.

Assumptions and Considerations

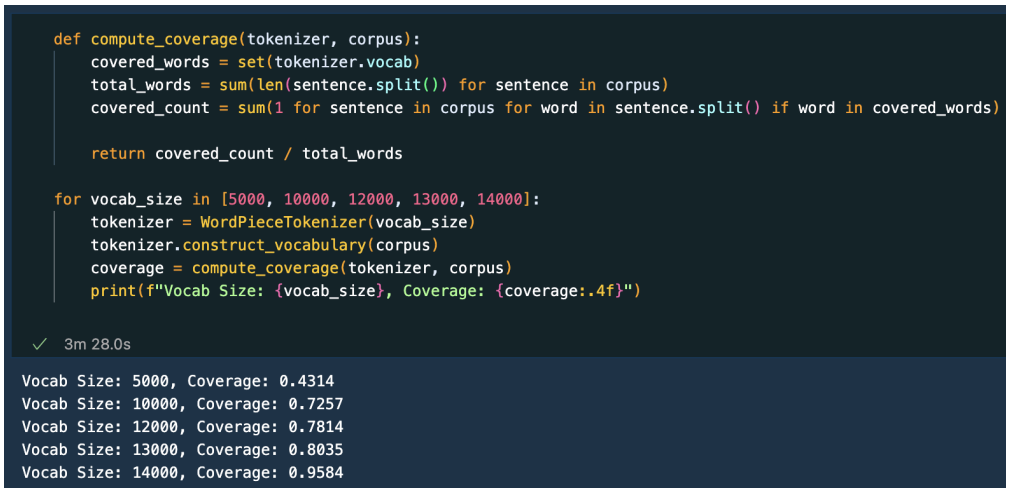
1. **Vocabulary Size:** In this implementation, the vocabulary size was set to 14,000, however, we tried working with vocabulary sizes in the ranges of 5000 to 14000. While selecting the size, we balanced the **trade-off in subword granularity** for optimal performance.

    Vocabulary Size vs. Generalization

    a. **Larger subwords** (e.g., whole words or common morphemes) → Reduce vocabulary size but may struggle with rare or out-of-vocabulary (OOV) words.
    b. **Smaller subwords** (e.g., character-level tokens) → Improve generalisation but lead to longer sequences, increasing computational cost.

    Metrics we used to determine a good vocabulary size:

    a. **Coverage:** Indicates the percentage of words in a given text that are present in the model's vocabulary. A coverage above 90%, ideally closer to 95% or higher, means that the vast majority of words in the text are known to the model.



```
def compute_coverage(tokenizer, corpus):
    covered_words = set(tokenizer.vocab)
    total_words = sum(len(sentence.split()) for sentence in corpus)
    covered_count = sum(1 for sentence in corpus for word in sentence.split() if word in covered_words)

    return covered_count / total_words

for vocab_size in [5000, 10000, 12000, 13000, 14000]:
    tokenizer = WordPieceTokenizer(vocab_size)
    tokenizer.construct_vocabulary(corpus)
    coverage = compute_coverage(tokenizer, corpus)
    print(f"Vocab Size: {vocab_size}, Coverage: {coverage:.4f}")
```

✓ 3m 28.0s

```
Vocab Size: 5000, Coverage: 0.4314
Vocab Size: 10000, Coverage: 0.7257
Vocab Size: 12000, Coverage: 0.7814
Vocab Size: 13000, Coverage: 0.8035
Vocab Size: 14000, Coverage: 0.9584
```

b. **Average subwords per word:** The average number of subwords per word tells us how aggressively the tokenizer is breaking down words. A high average suggests that even common words are being split into many pieces, which can be a sign of a too-small vocabulary.

   a reasonable range for an average subword count somewhere between 1.1 and 1.5.

```python
def compute_avg_subwords(tokenizer, corpus):
    total_words = 0
    total_subwords = 0

    for sentence in corpus:
        words = sentence.split()
        total_words += len(words)
        total_subwords += sum(len(tokenizer.tokenize(word)) for word in words)

    return total_subwords / total_words

for vocab_size in [5000, 10000, 12000, 13000, 14000]:
    tokenizer = WordPieceTokenizer(vocab_size)
    tokenizer.construct_vocabulary(corpus)
    avg_subwords = compute_avg_subwords(tokenizer, corpus)
    print(f"Vocab Size: {vocab_size}, Avg Subwords per Word: {avg_subwords:.2f}")
```

✓ 3m 45.3s

```
Vocab Size: 5000, Avg Subwords per Word: 2.52
Vocab Size: 10000, Avg Subwords per Word: 1.59
Vocab Size: 12000, Avg Subwords per Word: 1.43
Vocab Size: 13000, Avg Subwords per Word: 1.38
Vocab Size: 14000, Avg Subwords per Word: 1.07
```

We obtained our best results in the case of 14000 vocabulary size.

Examples from implementation:

1. **Initial Character Tokens (Before Merging)**

   At the start, we only have individual characters in the vocabulary:
   `s, u, p, p, y, o, r, t, h, a, c` (and more from the corpus).

2. **Merging Frequent Bigrams**

   WordPiece will start merging the most frequent adjacent characters first.
   For example:

   - `p` and `u` might frequently appear together → **"pu"**
   - `p` and `p` might frequently appear together → **"pp"**
   - `p` and `y` might frequently appear together → **"py"**

3. **Creating Common Subwords**

   If "puppy" appears frequently, it could have been built like this:

1. `p` + `u` → **"pu"`**
2. `pu` + `p` → **"pup"`**
3. `pup` + `p` → **"pupp"`**
4. `pupp` + `y` → **"puppy"`** (full word)

Since "puppy" is a frequent word, it likely becomes a full token.

4. Handling Other Words Like 'Support'

`s` + `u` → **"su"`**

`su` + `p` → **"sup"`**

`sup` + `p` → **"supp"`**

`supp` + `o` → **"suppo"`**

`suppo` + `r` → **"suppor"`**

`suppor` + `t` → **"support"`**

Since "support" is a common word, it appears as a full token.

5. Handling Word Variants Using '##'

Once full words exist, the tokenizer starts **splitting less frequent variations** into subwords:

- "puppy" is frequent, so it remains a full token.
- "##ppy" is created because it appears in words like **"happy"** and **"crappy"** but not as a standalone word.
- "##upp" appears in words like **"puppy"**, **"suppose"**, and **"support"**.

Examples for "happy" and "crappy"

- Since "puppy" is a word, "ppy" could appear as **"##ppy"** (a subword that appears inside words).
- "happy" is common, so it gets its own token.
- "crappy" is less common, so it is split:
  - "c" + "r" → "cr"
  - "cr" + "appy" → "crappy" (or sometimes split as "cr" + "##appy").

References:

WordPiece Tokenization

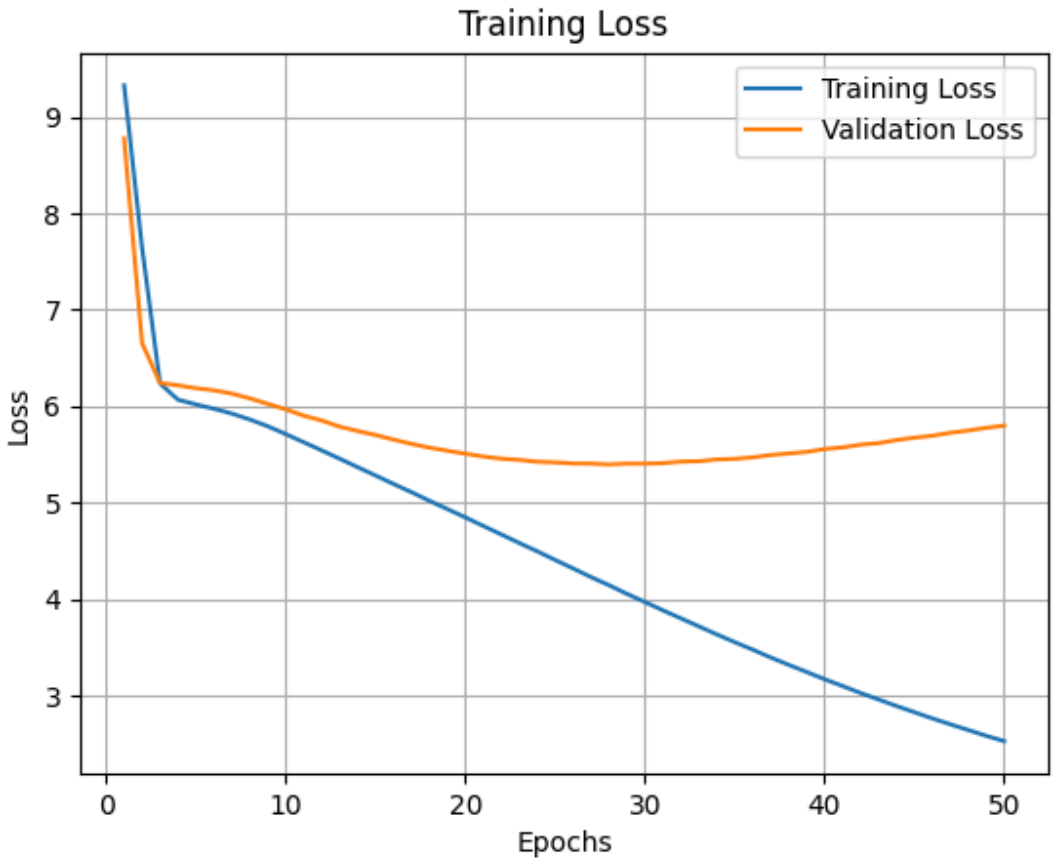
https://huggingface.co/learn/nlp-course/en/chapter6/6

Task 2 Implementing Word2Vec

References:

Datasets & DataLoaders — PyTorch Tutorials 2.6.0+cu124 documentation
Build the Neural Network — PyTorch Tutorials 2.6.0+cu124 documentation
Automatic Differentiation with torch.autograd — PyTorch Tutorials 2.6.0+cu124 documentation
torch.optim — PyTorch 2.6 documentation
torch.nn.functional — PyTorch 2.6 documentation
cosine_similarity — scikit-learn 1.6.1 documentation
11 — Word2Vec Approaches: Continuous Bag of Words (CBOW) & Skip-Gram | by Aysel Aydin | Medium

Losses:



Model shows steady decrease in training loss but at 50 epochs the validation loss starts to increase so we stop training it.

Cosine Similarity

Cosine similarity is calculated by the following formula :

$$cos\Theta = A \cdot B \,/\, ||A||||B||$$

A cosine similarity from 0 to 1 means words are similar to each other and from -1 to 0 means they are dissimilar

Identified triplets

```
delighted

similar words:
word:  honoured   with similarity:  0.8310107
word:  lucky   with similarity:  0.8068399
word:  privileged   with similarity:  0.8048269
Dissimilar: unhealed -0.60767937
```

Delighted, honoured, lucky and privileged are positive words/ situations while unhealed is a state of being hurt / negative state which has a low cosine similarity

```
 hope

 similar words:
 word:  thought   with similarity:  0.8000118
 word:  dream   with similarity:  0.74726975
 word:  wish   with similarity:  0.7442607
 Dissimilar: dread -0.49034765
```

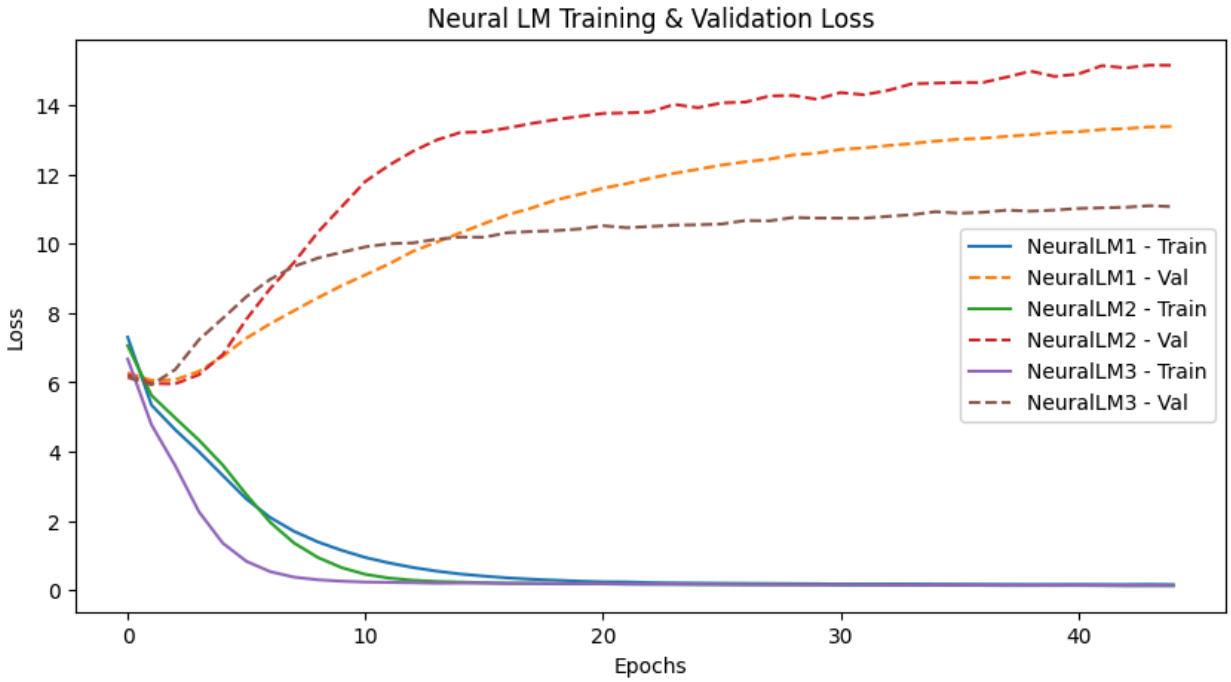
Hope , thought , wish, dream are related positive thoughts dread is a state of fear and has low cosine similarity

```
annoyed

similar words:
word:  frustrated   with similarity:  0.6656821
word:  resentful   with similarity:  0.610801
word:  shocked   with similarity:  0.6085851
Dissimilar: unconsciously -0.40372226
```

annoyed , frustrated, resentful and shocked are negative emotions which share the feeling of stress and unconsciously which is a physical state is completely unrelated to emotional states

Task 3 Train a Neural LM



1. Design Choices for Each Neural LM Architecture

NeuralLM1: Baseline Model

- A simple feedforward neural network with:
  - An embedding layer to map words to dense vectors.
  - A single hidden layer (128 units) with ReLU activation.
  - A fully connected output layer predicting the next word.
- Justification**:**
  - This model serves as a baseline with minimal complexity to evaluate basic performance.
  - ReLU activation helps introduce non-linearity for better feature extraction.

NeuralLM2: Deeper Network with Dropout

- Extended NeuralLM1 by adding:
    - An extra hidden layer (256 units) for deeper feature extraction.
    - Dropout (0.3) to reduce overfitting.
- Justification**:**
    - A deeper architecture is expected to capture more complex language patterns.
    - Dropout helps prevent over-reliance on certain neurons, improving generalization.

NeuralLM3: Batch Normalization for Stability

- Similar to NeuralLM1 but with:
    - Increased hidden units (512) for richer representations.
    - Batch normalization before activation to stabilize training.
- Justification**:**
    - Batch normalization helps normalize activations, improving training stability and reducing overfitting.
    - Increased hidden units allow for more expressive feature learning

2. Impact of Design Changes

- Training Accuracy & Perplexity: All three models achieve over 96% train accuracy with extremely low training perplexity (~1.1), indicating successful learning on the training set.
- Validation Accuracy & Perplexity:
    - NeuralLM1: 11.65% validation accuracy, 645,961 perplexity
    - NeuralLM2: 11.84% validation accuracy, 3,751,163 perplexity (highest, indicating worst generalization)
    - NeuralLM3: 13.47% validation accuracy, 63,946 perplexity (best validation performance)
- Observations:
    - All models overfit the training data due to memorization rather than learning general patterns.
    - NeuralLM2 performs the worst, possibly due to excessive complexity leading to instability.
    - NeuralLM3 shows marginal improvement, suggesting that its modifications (e.g., regularization techniques) slightly enhanced generalization.

3. Discussion of Performance Differences

- The large gap between training and validation metrics suggests the models struggle with unseen data.
- Possible reasons:

The corpus used for training may have contributed to overfitting due to the following reasons:

- Limited Vocabulary & Diversity: A small or repetitive dataset may lead to memorization rather than generalization.
- Mismatch Between Training & Validation Data: If the validation set contains linguistic variations not present in training, the model struggles with unseen data.
- Effect of WordPiece Tokenization: Rare subword tokens may not be well-represented, affecting generalization.

4. Conclusion

- NeuralLM3 performed the best, but all models suffered from severe overfitting.

References:

https://www.geeksforgeeks.org/word-embeddings-in-nlp/
https://www.geeksforgeeks.org/feedforward-neural-network/
https://www.geeksforgeeks.org/data-preprocessing-in-pytorch/

Individual Contributions:

Shamik : Task 1
Shrutya : Task 2
Vansh : Task 3