NLP Assignment 1

Amisha Gupta - MT23225
Niteen Kumar - 2022336
Nitin Kumar - 2022337

TASK 1 (Niteen Kumar)

WordPiece Tokenizer Implementation

Introduction

This report outlines the implementation of a WordPiece tokenizer, designed to preprocess text data, construct a subword-based vocabulary, and tokenize sentences. The tokenizer handles unknown words by breaking them into subwords and supports processing data from JSON files.

Implementation Details

1. Class Structure

The `WordPieceTokenizer` class consists of the following key methods:

- `preprocess_data()`: Cleans and normalizes text.
- `construct_vocabulary()`: Builds a subword vocabulary from a corpus.
- `tokenize()`: Splits sentences into tokens using the vocabulary.
- `tokenize_json()`: Processes JSON input files and saves tokenized outputs.

---

2. Preprocessing

The preprocessing pipeline includes:

- **Lowercasing**: Converts all text to lowercase.
- **Special Character Removal**: Retains alphanumeric characters and spaces.
- **Whitespace Normalization**: Reduces multiple spaces to a single space

3. Vocabulary Construction

The vocabulary is built using a simplified WordPiece-like approach:

1. **Initial Subwords:** Extract individual characters from the corpus.
2. **Subword Expansion:** For each word, generate prefixes (e.g., `"t"`, `"th"` for `"this"`) and suffixes with `##` markers (e.g., `##his`, `##is`).
3. **Frequency Sorting:** Subwords are sorted by frequency (if `vocab_size` is specified, the top entries are selected).
4. **Special Tokens:** `[UNK]` (unknown tokens) and `[PAD]` (padding) are added by default.

```
[UNK]
[PAD]
i
feel
and
to
the
a
that
feeling
of
my
it
in
like
im
for
me
so
but
was
```

4. Tokenization Process

- **Known Words:** Directly mapped to vocabulary entries.
- **Unknown Words:** Split into subwords greedily, starting from the longest possible match. Subwords not at the start of a word are prefixed with `##`.

```
remembranc
##enyeri
##eemingly
##rtion
```

5. Handling JSON Data

- **Input:** Reads a JSON file (e.g., `test.json`) with samples containing `id` and `sentence` fields.
- **Output:** Generates a JSON file (e.g., `tokenized_01.json`) with `id` and `tokens` fields for each sample.

```json
[
    {
        "id": 0,
        "tokens": [
            "i",
            "cant",
            "help",
            "but",
            "als",
            "##o",
            "feel",
            "incredibl",
            "##y",
            "luck",
            "##y",
            "over",
            "how",
            "it",
            "all",
            "wen",
            "##t",
            "down",
            "and"
```

The tokenizer is initialized and used as follows:

```python
# Initialize the tokenizer with dynamic vocab size (e.g., 50)
tokenizer = WordPieceTokenizer(vocab_size=25000)

# Construct vocabulary with dynamic size
tokenizer.construct_vocabulary(
    corpus, vocab_size=25000, vocab_file=f"vocabulary_{group_no}.txt")
print("Vocabulary saved to vocabulary1.txt")

# Tokenize sentences from test.json
tokenizer.tokenize_json(input_file="test.json",
                        output_file=f"tokenized_{group_no}.json")
```

Strengths

- **Subword Handling**: Effectively breaks down unseen words into meaningful subwords.
- **JSON Support**: Streamlines batch processing of text data.
- **Custom Vocabulary Size**: Allows control over memory and computational requirements.

Limitations

- **Simplified Algorithm**: Unlike the standard WordPiece (which merges frequent token pairs iteratively), this implementation generates subwords through prefix/suffix splitting, potentially leading to suboptimal tokenization.
- **Special Characters**: Aggressive removal of punctuation may degrade performance in tasks requiring syntactic analysis.
- **Efficiency**: Generating all possible subwords during vocabulary construction may be computationally expensive for large corpus.

Conclusion

This implementation provides a functional WordPiece-style tokenizer suitable for basic NLP tasks. Key improvements could include adopting a merge-based strategy for vocabulary construction and preserving punctuation for specific use cases. The code demonstrates core concepts of subword tokenization and offers a foundation for further optimization.

TASK 2 (Nitin Kumar)

Word2Vec Implementation with Negative Sampling and Subsampling

Introduction

This report details a Word2Vec implementation using Continuous Bag-of-Words (CBOW) architecture with key enhancements for improved performance and evaluation. The implementation features advanced techniques including subsampling of frequent words, negative sampling, and a novel triplet analysis method for embedding evaluation.

Key Features & Improvements

1. **Enhanced Dataset Handling**
   - Context window of ±2 words
   - Subsampling of frequent words using (f(w)/Σf)^0.75 formula
   - Add-one smoothing for frequency estimation
   - Dynamic negative sampling based on word frequencies
2. **Model Architecture**
   - Dual embedding system (word + context embeddings)
   - Dropout regularization (p=0.1)
   - Xavier-style initialization
   - Negative sampling loss implementation
3. **Training Optimization**
   - Early stopping with patience=3
   - ReduceLROnPlateau learning rate scheduler
   - Adam optimizer (lr=0.001)
   - Batch training (size=32)
4. **Novel Evaluation**
   - Cosine similarity matrix computation
   - Triplet analysis (similar pair + dissimilar word)
   - Similarity threshold filtering (0.5)

Methodology

1. Data Preprocessing

- **Tokenization**: Uses WordPiece tokenizer from previous task
- **Subsampling**: Implements word-frequency based probability

```
# Apply subsampling formula (t^0.75)
frequencies = np.power(frequencies, 0.75)
```

- **Negative Sampling**: 5 samples per context using smoothed distribution

2. Model Structure

```python
class Word2VecModel(nn.Module):
    def __init__(self, vocab_size, embedding_dim):
        super(Word2VecModel, self).__init__()
        self.embeddings = nn.Embedding(vocab_size, embedding_dim)
        self.context_embeddings = nn.Embedding(vocab_size, embedding_dim)
        self.dropout = nn.Dropout(0.1)

        # Initialize embeddings
        initrange = 0.5 / embedding_dim
        self.embeddings.weight.data.uniform_(-initrange, initrange)
        self.context_embeddings.weight.data.uniform_(-0, 0)
```

3. Loss Calculation

Combined positive and negative loss:

```python
positive_loss = F.logsigmoid(positive_score)
negative_loss = F.logsigmoid(-negative_scores).sum(1)

return -(positive_loss + negative_loss).mean()
```

4. Triplet Analysis Algorithm

1. Compute pairwise cosine similarities
2. Identify similar pairs (similarity ≥ 0.5)
3. Find least similar word using averaged similarity scores

Results & Evaluation

Training Performance

- Loss curves tracked for 100 epochs maximum
- Early stopping typically occurs around epoch 15-20
- Example output:

```
Epoch 1, Train Loss: 3.0703, Val Loss: 2.2974
Epoch 2, Train Loss: 2.0911, Val Loss: 2.0871
Epoch 3, Train Loss: 1.8814, Val Loss: 1.9696
Epoch 4, Train Loss: 1.6992, Val Loss: 1.8616
Epoch 5, Train Loss: 1.5164, Val Loss: 1.7678
Epoch 6, Train Loss: 1.3470, Val Loss: 1.7094
Epoch 7, Train Loss: 1.2031, Val Loss: 1.6745
Epoch 8, Train Loss: 1.0829, Val Loss: 1.6655
Epoch 9, Train Loss: 0.9828, Val Loss: 1.6628
Epoch 10, Train Loss: 0.9002, Val Loss: 1.6741
Epoch 11, Train Loss: 0.8273, Val Loss: 1.6871
Epoch 12, Train Loss: 0.7631, Val Loss: 1.7099
Early stopping at epoch 12
```

Triplet Analysis Examples

Top 5 results from similarity analysis:

```
Top 5 Similar Word Pairs with their Dissimilar Words:

Pair 1:
Similar words: fetis - psycho (similarity: 1.000)
Dissimilar word: cynica (similarities: -0.466, -0.466)

Pair 2:
Similar words: continuousl - empath (similarity: 0.962)
Dissimilar word: uncertaintie (similarities: -0.732, -0.725)

Pair 3:
Similar words: empath - panick (similarity: 0.950)
Dissimilar word: uncertaintie (similarities: -0.725, -0.652)

Pair 4:
Similar words: empath - unnecessar (similarity: 0.938)
Dissimilar word: uncertaintie (similarities: -0.725, -0.669)

Pair 5:
Similar words: imbibe - tow (similarity: 0.935)
Dissimilar word: jack (similarities: -0.560, -0.575)
```

Visualization

Training and validation loss showing stable convergence pattern

Conclusion & Recommendations

This implementation demonstrates effective word embedding learning through:

- Proper handling of word frequencies via subsampling
- Robust negative sampling strategy
- Effective regularization with dropout
- Meaningful similarity relationships in results

TASK 3 (Amisha Gupta)

Introduction

This report details the training of Neural Language Model- An MLP based model using PyTorch.

Implementation Details -
Classes -

1. NeuralLMDataset - responsible for handling the data preparation for training the language model. It ensures that the data is preprocessed and formatted particularly for the next-word prediction task.
2. NeuralLM1 - It is the simplest architecture of the three variations. It consists of an embedding layer, a hidden linear layer, and an output layer.
3. NeuralLM2 - NeuralLM2 builds upon NeuralLM1 by adding a second hidden layer and using a Leaky ReLU activation for better gradient flow.
4. NeuralLM3 - integrates convolutional layers

Functions -

1. compute_accuracy - computes the accuracy of the model by comparing the predicted tokens with the actual target tokens.
2. compute_perplexity - computes the perplexity from the given loss value.
3. train - This function contains all the training logic that is required to train all the three models.
4. predict - makes predictions for the next three tokens

Results-

Model	**Architecture**	**Accuracy**	**Perplexity**	**Rationale**
NeuralLM1	Basic neural network with one hidden layer, ReLU activation, and fully connected layers.	0.70	3.16	Simple architecture, fails to capture complex dependencies
NeuralLM2	Deeper architecture with two hidden layers and Leaky ReLU activation.	0.56	5.81	Added complexity may led to overfitting, therefore resulted in performance drop.
NeuralLM3	Convolutional layers with ReLU activation, followed by fully connected layer for output	0.90	1.4	Conv layers generalizes well for next token prediction task as it was able to learn local

				patterns

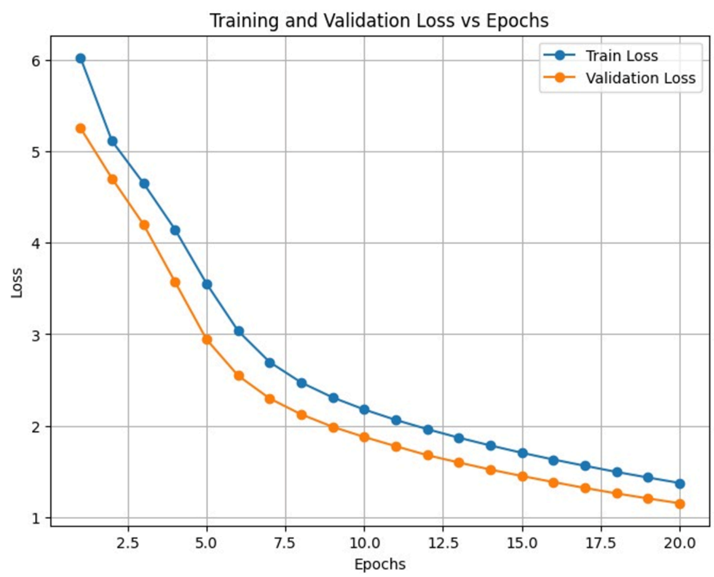
Performance-

Models were trained for 20 epochs.

1- NeuralLM1

```
Training NeuralLM1...

Epoch 1: Train Loss: 6.0275, Train Acc: 0.0968, Train PPL: 414.6814
Val Loss: 5.2574, Val Acc: 0.1237, Val PPL: 191.9892
Epoch 2: Train Loss: 5.1142, Train Acc: 0.1309, Train PPL: 166.3700
Val Loss: 4.7053, Val Acc: 0.1481, Val PPL: 110.5304
Epoch 3: Train Loss: 4.6525, Train Acc: 0.1485, Train PPL: 104.8466
Val Loss: 4.2018, Val Acc: 0.1713, Val PPL: 66.8045
Epoch 4: Train Loss: 4.1463, Train Acc: 0.1702, Train PPL: 63.2001
Val Loss: 3.5732, Val Acc: 0.2266, Val PPL: 35.6313
Epoch 5: Train Loss: 3.5537, Train Acc: 0.2299, Train PPL: 34.9428
Val Loss: 2.9429, Val Acc: 0.3328, Val PPL: 18.9707
Epoch 6: Train Loss: 3.0370, Train Acc: 0.3112, Train PPL: 20.8435
Val Loss: 2.5478, Val Acc: 0.4071, Val PPL: 12.7786
Epoch 7: Train Loss: 2.6958, Train Acc: 0.3717, Train PPL: 14.8170
Val Loss: 2.2977, Val Acc: 0.4540, Val PPL: 9.9514
Epoch 8: Train Loss: 2.4722, Train Acc: 0.4132, Train PPL: 11.8488
Val Loss: 2.1236, Val Acc: 0.4884, Val PPL: 8.3612
Epoch 9: Train Loss: 2.3085, Train Acc: 0.4446, Train PPL: 10.0591
Val Loss: 1.9871, Val Acc: 0.5165, Val PPL: 7.2946
Epoch 10: Train Loss: 2.1765, Train Acc: 0.4696, Train PPL: 8.8155
Val Loss: 1.8760, Val Acc: 0.5410, Val PPL: 6.5277
Epoch 11: Train Loss: 2.0625, Train Acc: 0.4932, Train PPL: 7.8655
Val Loss: 1.7751, Val Acc: 0.5623, Val PPL: 5.9011
Epoch 12: Train Loss: 1.9616, Train Acc: 0.5141, Train PPL: 7.1106
Val Loss: 1.6768, Val Acc: 0.5839, Val PPL: 5.3484
Epoch 13: Train Loss: 1.8682, Train Acc: 0.5344, Train PPL: 6.4765
Val Loss: 1.5969, Val Acc: 0.6013, Val PPL: 4.9377
Epoch 14: Train Loss: 1.7832, Train Acc: 0.5513, Train PPL: 5.9490
Val Loss: 1.5193, Val Acc: 0.6200, Val PPL: 4.5690
Epoch 15: Train Loss: 1.7029, Train Acc: 0.5698, Train PPL: 5.4897
Val Loss: 1.4488, Val Acc: 0.6370, Val PPL: 4.2579
Epoch 16: Train Loss: 1.6293, Train Acc: 0.5866, Train PPL: 5.1001
Val Loss: 1.3826, Val Acc: 0.6519, Val PPL: 3.9853
Epoch 17: Train Loss: 1.5608, Train Acc: 0.6028, Train PPL: 4.7627
Val Loss: 1.3179, Val Acc: 0.6666, Val PPL: 3.7355
Epoch 18: Train Loss: 1.4936, Train Acc: 0.6175, Train PPL: 4.4529
Val Loss: 1.2585, Val Acc: 0.6827, Val PPL: 3.5201
Epoch 19: Train Loss: 1.4322, Train Acc: 0.6338, Train PPL: 4.1877
Val Loss: 1.2034, Val Acc: 0.6959, Val PPL: 3.3313
Epoch 20: Train Loss: 1.3727, Train Acc: 0.6464, Train PPL: 3.9461
Val Loss: 1.1508, Val Acc: 0.7088, Val PPL: 3.1608
```

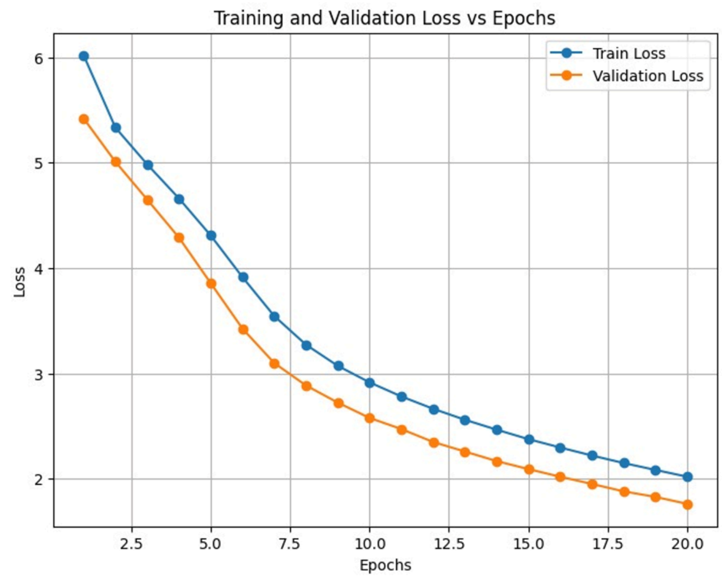


Training and validation loss showing stable convergence pattern

2- NeuralLM2

```
Training NeuralLM2...

Epoch 1: Train Loss: 6.0192, Train Acc: 0.0896, Train PPL: 411.2292
Val Loss: 5.4197, Val Acc: 0.1160, Val PPL: 225.8002
Epoch 2: Train Loss: 5.3325, Train Acc: 0.1253, Train PPL: 206.9513
Val Loss: 5.0083, Val Acc: 0.1437, Val PPL: 149.6438
Epoch 3: Train Loss: 4.9842, Train Acc: 0.1451, Train PPL: 146.0894
Val Loss: 4.6503, Val Acc: 0.1626, Val PPL: 104.6151
Epoch 4: Train Loss: 4.6605, Train Acc: 0.1597, Train PPL: 105.6937
Val Loss: 4.2887, Val Acc: 0.1783, Val PPL: 72.8690
Epoch 5: Train Loss: 4.3086, Train Acc: 0.1717, Train PPL: 74.3380
Val Loss: 3.8561, Val Acc: 0.2060, Val PPL: 47.2810
Epoch 6: Train Loss: 3.9135, Train Acc: 0.2025, Train PPL: 50.0733
Val Loss: 3.4247, Val Acc: 0.2728, Val PPL: 30.7146
Epoch 7: Train Loss: 3.5442, Train Acc: 0.2554, Train PPL: 34.6132
Val Loss: 3.0980, Val Acc: 0.3209, Val PPL: 22.1544
Epoch 8: Train Loss: 3.2719, Train Acc: 0.2940, Train PPL: 26.3610
Val Loss: 2.8851, Val Acc: 0.3531, Val PPL: 17.9046
Epoch 9: Train Loss: 3.0713, Train Acc: 0.3237, Train PPL: 21.5691
Val Loss: 2.7224, Val Acc: 0.3834, Val PPL: 15.2175
Epoch 10: Train Loss: 2.9146, Train Acc: 0.3472, Train PPL: 18.4412
Val Loss: 2.5769, Val Acc: 0.4062, Val PPL: 13.1568
Epoch 11: Train Loss: 2.7808, Train Acc: 0.3698, Train PPL: 16.1323
Val Loss: 2.4712, Val Acc: 0.4240, Val PPL: 11.8367
Epoch 12: Train Loss: 2.6639, Train Acc: 0.3887, Train PPL: 14.3527
Val Loss: 2.3487, Val Acc: 0.4467, Val PPL: 10.4723
Epoch 13: Train Loss: 2.5595, Train Acc: 0.4043, Train PPL: 12.9297
Val Loss: 2.2579, Val Acc: 0.4612, Val PPL: 9.5633
Epoch 14: Train Loss: 2.4655, Train Acc: 0.4218, Train PPL: 11.7690
Val Loss: 2.1671, Val Acc: 0.4810, Val PPL: 8.7332
Epoch 15: Train Loss: 2.3756, Train Acc: 0.4385, Train PPL: 10.7571
Val Loss: 2.0912, Val Acc: 0.4949, Val PPL: 8.0946
Epoch 16: Train Loss: 2.2962, Train Acc: 0.4528, Train PPL: 9.9368
Val Loss: 2.0186, Val Acc: 0.5105, Val PPL: 7.5278
Epoch 17: Train Loss: 2.2200, Train Acc: 0.4676, Train PPL: 9.2073
Val Loss: 1.9501, Val Acc: 0.5240, Val PPL: 7.0292
Epoch 18: Train Loss: 2.1487, Train Acc: 0.4809, Train PPL: 8.5733
Val Loss: 1.8795, Val Acc: 0.5384, Val PPL: 6.5504
Epoch 19: Train Loss: 2.0831, Train Acc: 0.4922, Train PPL: 8.0297
Val Loss: 1.8272, Val Acc: 0.5493, Val PPL: 6.2164
Epoch 20: Train Loss: 2.0189, Train Acc: 0.5049, Train PPL: 7.5298
Val Loss: 1.7610, Val Acc: 0.5633, Val PPL: 5.8185
```

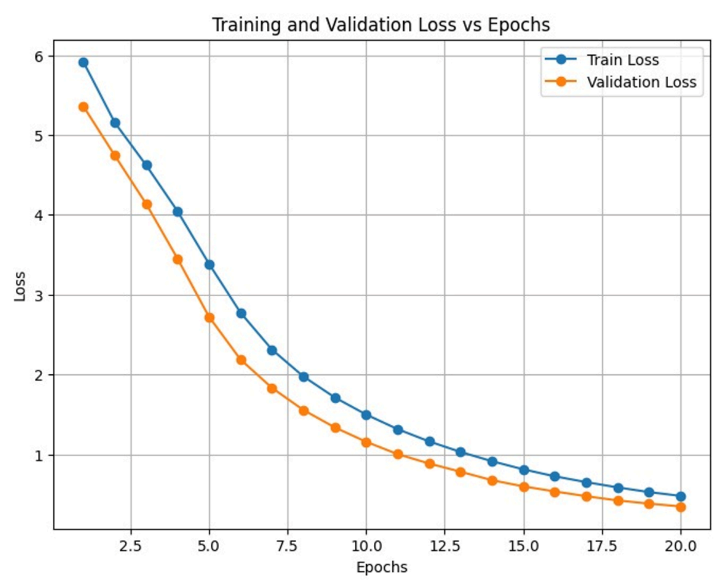


Training and validation loss showing stable convergence pattern

3- NeuralLM3

```
Training NeuralLM3...

Epoch 1: Train Loss: 5.9176, Train Acc: 0.0907, Train PPL: 371.5123
Val Loss: 5.3614, Val Acc: 0.1217, Val PPL: 213.0332
Epoch 2: Train Loss: 5.1558, Train Acc: 0.1334, Train PPL: 173.4287
Val Loss: 4.7440, Val Acc: 0.1503, Val PPL: 114.8895
Epoch 3: Train Loss: 4.6214, Train Acc: 0.1523, Train PPL: 101.6338
Val Loss: 4.1376, Val Acc: 0.1736, Val PPL: 62.6550
Epoch 4: Train Loss: 4.0443, Train Acc: 0.1715, Train PPL: 57.0741
Val Loss: 3.4501, Val Acc: 0.2224, Val PPL: 31.5039
Epoch 5: Train Loss: 3.3866, Train Acc: 0.2263, Train PPL: 29.5644
Val Loss: 2.7231, Val Acc: 0.3501, Val PPL: 15.2281
Epoch 6: Train Loss: 2.7757, Train Acc: 0.3251, Train PPL: 16.0500
Val Loss: 2.1885, Val Acc: 0.4557, Val PPL: 8.9219
Epoch 7: Train Loss: 2.3131, Train Acc: 0.4132, Train PPL: 10.1059
Val Loss: 1.8332, Val Acc: 0.5298, Val PPL: 6.2539
Epoch 8: Train Loss: 1.9760, Train Acc: 0.4810, Train PPL: 7.2142
Val Loss: 1.5553, Val Acc: 0.5924, Val PPL: 4.7363
Epoch 9: Train Loss: 1.7127, Train Acc: 0.5403, Train PPL: 5.5438
Val Loss: 1.3374, Val Acc: 0.6414, Val PPL: 3.8091
Epoch 10: Train Loss: 1.4981, Train Acc: 0.5896, Train PPL: 4.4733
Val Loss: 1.1568, Val Acc: 0.6876, Val PPL: 3.1798
Epoch 11: Train Loss: 1.3153, Train Acc: 0.6338, Train PPL: 3.7257
Val Loss: 1.0023, Val Acc: 0.7269, Val PPL: 2.7247
Epoch 12: Train Loss: 1.1619, Train Acc: 0.6730, Train PPL: 3.1960
Val Loss: 0.8849, Val Acc: 0.7579, Val PPL: 2.4228
Epoch 13: Train Loss: 1.0299, Train Acc: 0.7068, Train PPL: 2.8008
Val Loss: 0.7822, Val Acc: 0.7853, Val PPL: 2.1862
Epoch 14: Train Loss: 0.9138, Train Acc: 0.7384, Train PPL: 2.4938
Val Loss: 0.6757, Val Acc: 0.8117, Val PPL: 1.9654
Epoch 15: Train Loss: 0.8117, Train Acc: 0.7657, Train PPL: 2.2516
Val Loss: 0.5981, Val Acc: 0.8338, Val PPL: 1.8187
Epoch 16: Train Loss: 0.7241, Train Acc: 0.7886, Train PPL: 2.0630
Val Loss: 0.5348, Val Acc: 0.8520, Val PPL: 1.7071
Epoch 17: Train Loss: 0.6516, Train Acc: 0.8090, Train PPL: 1.9186
Val Loss: 0.4752, Val Acc: 0.8669, Val PPL: 1.6083
Epoch 18: Train Loss: 0.5855, Train Acc: 0.8282, Train PPL: 1.7959
Val Loss: 0.4222, Val Acc: 0.8822, Val PPL: 1.5254
Epoch 19: Train Loss: 0.5264, Train Acc: 0.8440, Train PPL: 1.6929
Val Loss: 0.3822, Val Acc: 0.8931, Val PPL: 1.4655
Epoch 20: Train Loss: 0.4793, Train Acc: 0.8594, Train PPL: 1.6150
Val Loss: 0.3476, Val Acc: 0.9017, Val PPL: 1.4157
```



Training and validation loss showing stable convergence pattern

Predictions-

1- NeuralLM1-

```
Input: i felt like earlier this year i was starting to feel emotional that it
Predicted next words: was ##s over

Input: i do need constant reminders when i go through lulls in feeling submiss
Predicted next words: ##l i want

Input: i was really feeling crappy even after my awesome
Predicted next words: week ##s workout

Input: i finally realise the feeling of being hated and its after effects are
Predicted next words: so big i

Input: i am feeling unhappy and weird
Predicted next words: im confiden ##d
```

2- NeuralLM2

```
Input: i felt like earlier this year i was starting to feel emotional that it
Predicted next words: s be strange

Input: i do need constant reminders when i go through lulls in feeling submiss
Predicted next words: ##r polic trie

Input: i was really feeling crappy even after my awesome
Predicted next words: week ##s workout

Input: i finally realise the feeling of being hated and its after effects are
Predicted next words: so big i

Input: i am feeling unhappy and weird
Predicted next words: an confiden ##d
```

3- NeuralLM3

```
Input: i felt like earlier this year i was starting to feel emotional that it
Predicted next words: was all over

Input: i do need constant reminders when i go through lulls in feeling submiss
Predicted next words: ##l and is

Input: i was really feeling crappy even after my awesome
Predicted next words: week of workout

Input: i finally realise the feeling of being hated and its after effects are
Predicted next words: so big i

Input: i am feeling unhappy and weird
Predicted next words: im confiden ##t
```

References-

UMass CS685 (Advanced NLP) F20: Implementing a neural language model in PyTorch -
UMass CS685 (Advanced NLP) F20: Implementing a neural language model in PyTorch