CSE556: Natural Language Processing
Assignment-1

Himanshu Raj (2022216) | Ishita (2022224) | Ritika Thakur (2022408)

February 2, 2025

1 Task 1

1.1 Data Pre-processing

Our corpus `corpus.txt` had over `45k` words which included all lower-case words with no number and punctuations. The corpus also included words like `http`, `src`, `img`, `href`, `www` which did not add anything to the contextual meaning of the words. The following pre-processing steps were applied:

- **Removing irrelevant words and trailing content**: Words such as `img`, `http`, `href`, and `src` along with everything after them were removed to eliminate unnecessary HTML-like tags and URLs. These words along with the trailing content were found to be irrelevant in the corpus and did not add to the context of the sentences.
- **Whitespace normalization**: Extra spaces were removed.

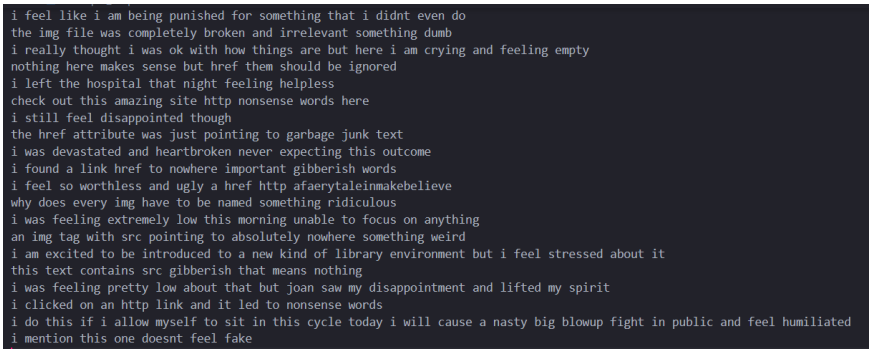

Figure 1: Example corpus before pre-processing

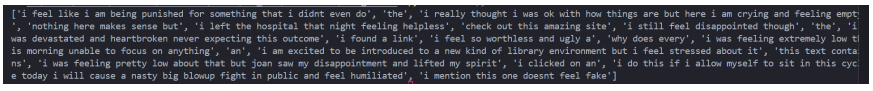

Figure 2: Example corpus after pre-processing

1.2 Vocabulary Generation

The vocabulary generation is based on the WordPiece subword tokenization approach (similar to BPE Algorithm).

1. **Splitting words into characters**: Each word is initially split into individual characters, with non-initial characters prefixed by `##`.

2. **Pair frequency calculation**: The frequency of adjacent character pairs is counted across the corpus.

3. **Pair scoring**: A score is calculated for each character pair based on its frequency relative to individual unit frequencies:
   `freq{pair} / (freq{word1} * freq{word2})`

4. **Merging pairs iteratively**: The highest-scoring pairs are merged until the vocabulary size reaches the predefined limit.

`[UNK]` and `[PAD]` are also added to the vocabulary.

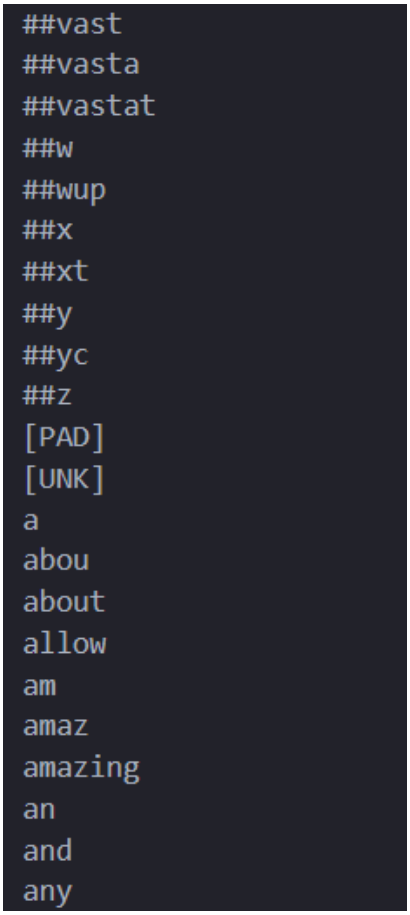

Figure 3: Snippet from vocabulary generated from example corpus with `size=500`

1.3 Tokenization

The tokenization process converts input text into subword units based on the generated vocabulary. After splitting words in the text into characters, as done during vocabulary generation, characters are merged based on the longest match in the vocabulary. If a word cannot be matched, it is replaced with `[UNK]`.

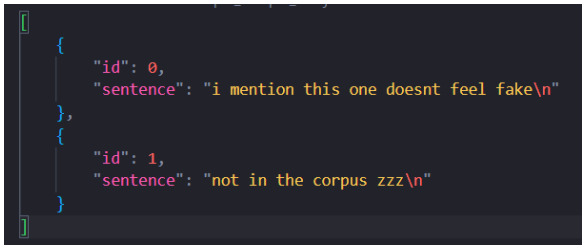

Figure 4: Test json file

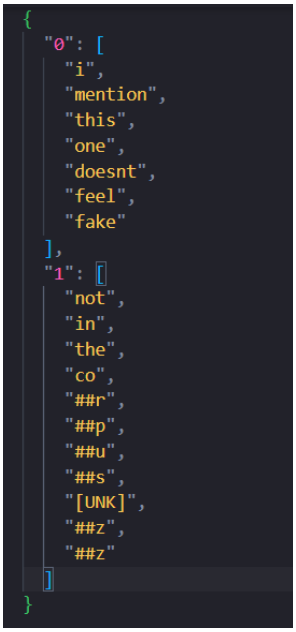

Figure 5: Tokenized json file

2 Task 2

2.1 Introduction

We build a Word2Vec model using the Continuous Bag of Words (CBOW) approach. The goal is to preprocess the input text corpus, generate CBOW pairs, and use them to train the Word2Vec model. Also, we find similar and dissimilar word triplets using cosine similarity.

2.2 Dataset Preparation - Word2VecDataset Class

The **Word2VecDataset class** preprocesses the input text corpus for training the Word2Vec model using the **Continuous Bag of Words (CBOW)** approach.

1. **Initialization** : It initializes the window size for context words, vocabulary, and mappings between words and indices (word2idx and idx2word). It also initializes CBOW training pairs.
2. **Pre-processes Data** :
   (a) Uses the WordPieceTokenizer from Task 1 to generate the vocabulary from the input text.

(b) Tokenizes the input text corpus using the WordPieceTokenizer from Task 1.

(c) Generates mapping from each word to a unique index and vice versa

(d) Generates CBOW training pairs based on the tokenized corpus.

Figure 6: Tokenized Corpus

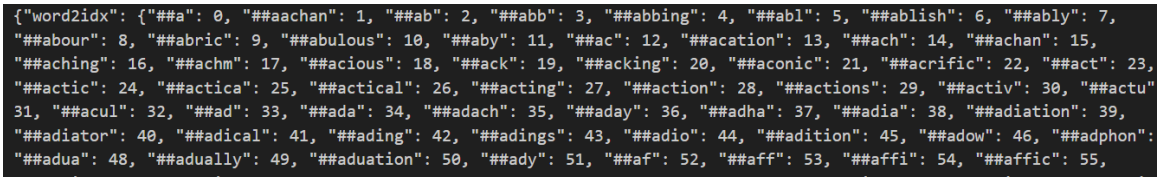

Figure 7: Word to Index Mapping

Figure 8: Index to Word Mapping

2.3 Model Architecture - Word2VecModel Class

1. **Initialization** : Initializes the vocabulary size, embedding dimension for each word and weights using a uniform distribution. Bias terms are set to zero.

2. **Architecture Details**

   - **Embedding Layer**: Maps words to their corresponding vector representations.
   - **Linear Layer**: Predicts the target word given the context of word embeddings.

- **Log Softmax Activation**: Outputs probability distribution over the entire vocabulary.

3. **Forward Function** : The forward pass gets the embeddings for the context words, averages them, and sends the result to the output layer. The softmax function is then applied to give a probability distribution over all the words in the vocabulary.

4. **Calculates Cosine Similarity** : Calculates cosine similarity between two vectors and also identifies the most and least similar vector to a given vector.

2.4 Training

1. **Initialization** : Initializes Adam optimizer, NLLLoss() for calculating loss and moves the model to the appropriate device (CPU/GPU). At the same time it also initializes history to keep track of loss and accuracy.

2. **Training** : The function trains the Word2Vec CBOW model. In every iteration, it decreases the learning rate and processes batches of context-target pairs. For each batch, the model performs a forward pass, computes the loss, and applies backpropagation to update the weights. Training progress, including batch loss and accuracy, is displayed using the progress bar.

3. **Model Validation** : After each epoch it computes validation loss, validation accuracy and average cosine similarity for the batch and updates the history.

4. **Updates Final Checkpoint and Saves the Final Model**

2.5 Finding Triplets

The **find_triplets** function creates triplets consisting of two similar words and one dissimilar word based on cosine similarity. It randomly selects a word, retrieves the two most similar words, and identifies the least similar word using the Word2Vec model.

Triplet 1:

- **High positive similarity:** "#run" has a similarity of **0.8747** with "haul" and **0.8696** with "#ashion" ighlighting their frequent occurrence in similar contexts.

- **Negative similarity:** "#run" and "uhur" have a similarity of **-0.9269**, suggesting they occur in opposite contexts or are rarely used together.

Triplet 2:

- **Very high positive similarity:** "#logic" has a similarity of **0.9312** with "##rful" and **0.9026** with "entry," highlighting their frequent occurrence in similar contexts.

- **Negative similarity:** "#logic" and "#aomi" have a similarity of **-0.8795**,suggesting they occur in opposite contexts or are rarely used together.

```
Triplet 1:
Token - ##run (index: 2296)
Similar tokens - haul (index: 5482), ##ashion (index: 285)
Dissimilar token - uhur (index: 8051)
Cosine similarity (##run, haul): 0.8747
Cosine similarity (##run, ##ashion): 0.8696
Cosine similarity (##run, uhur): -0.9269

Triplet 2:
Token - logic (index: 6054)
Similar tokens - ##rful (index: 2054), entry (index: 4856)
Dissimilar token - ##aomi (index: 211)
Cosine similarity (logic, ##rful): 0.9312
Cosine similarity (logic, entry): 0.9026
Cosine similarity (logic, ##aomi): -0.8795
```

Figure 9: Identifying Triplets

2.6 Experimental Results

After experimenting with the values of hyperparameters, the following hyperparameters gave the best results:

WINDOW_SIZE = 4

EMBEDDING_DIM = 10

BATCH_SIZE = 256

NUM_EPOCHS = 15

LEARNING_RATE = 0.02

TRAIN_SPLIT = 0.8

VOCAB_SIZE = 8500

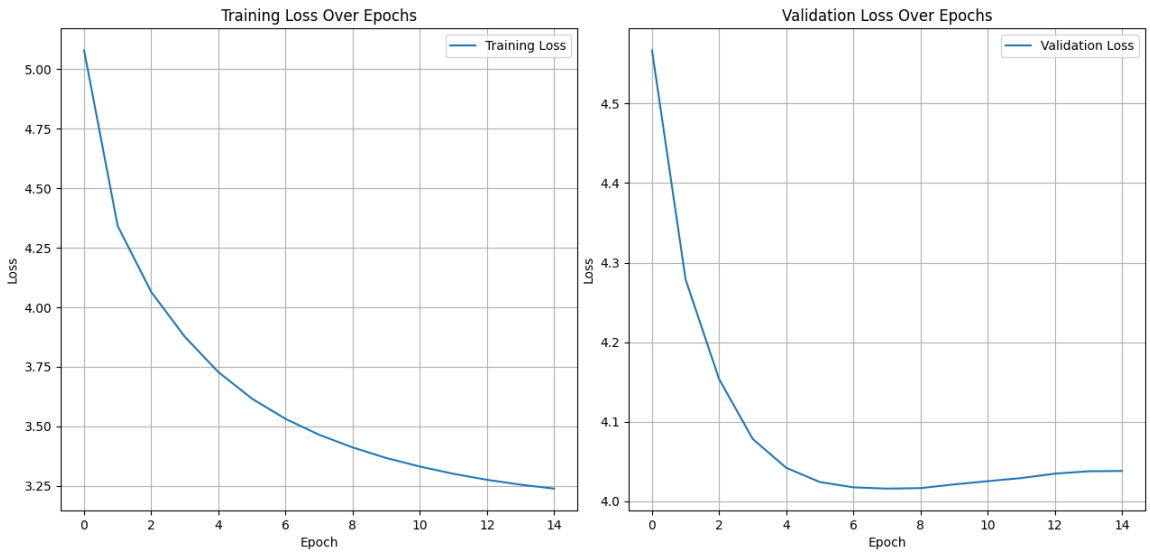

Figure 10: Training Loss and Validation Loss

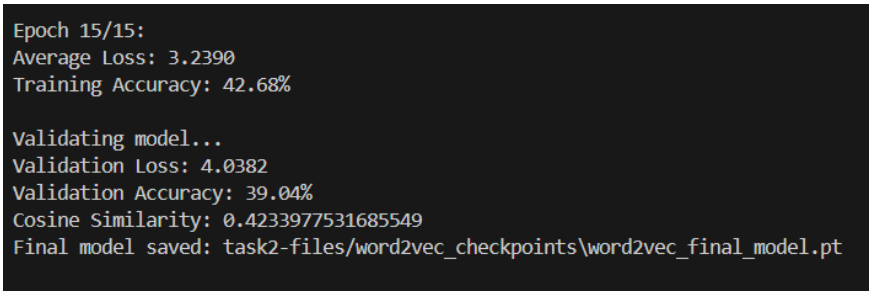

Figure 11: Best Validation Accuracy using the given hyperparameters

3 Task 3

3.1 Introduction

For all architectures, input layer contains 40 neurons (contextwindow x embeddingdimension = 4x10) and output layer contains 8500 neurons (the vocab size used to train Word2Vec model). All models were trained for 5 epochs with a learning rate of 0.01 with a batch size of 1024 samples over a training set with 60,000+ samples. A common observation over these models was that validation loss increased significantly after 6th epoch, even with smaller learning rates. The training time and compute increased as we go from NeuralLM1 to NeuralLM2 to NeuralLM3.

3.2 Design Justifications and Performance comparisons

NeuralLM1: 1 hidden layer with 1024 neurons and tanh activation

Justification: This is a basic MLP based Neural Language Model architecture proposed by Bengio in 2003. This serves as a good baseline model to start with when training neural language models.

NeuralLM2: 1 hidden layer with 1024 neurons and ReLU activation and skip connection

Justification: The tanh activation outputs cap at a value of 1 which can lead to less/slower updates in learnable parameters. The ReLU activation output is unbounded and hence is used in an attempt to improve accuracy and perplexity. Skip connections help in tackling vanishing gradients, proposed by Bengio in 2003 but they were optional.

Improvement: This led to better training and validation accuracy than NeuralLM1. Training perplexity decreased, but validation perplexity increased signifying model is unable to generalize on unseen data.

NeuralLM3: 2 hidden layers with 512 and 2048 neurons and ReLU activation

Justification: In previous 2 architectures, validation loss increases after 6th epoch indicating model is unable to generalize on unseen data. Hence, another hidden layer is introduced in an attempt to capture the features in a better way, so that the model can generalise well on unseen data. We are not adding skip connection like NeuralLM2 because we saw an increase in validation perplexity because of which model is unable to generalize well on unseen data.

Improvement: This led to better training and validation accuracy than NeuralLM2. The validity perplexity is even less than NeuralLM1, signifying the best performance on unseen data out of all architectures. Training perplexity is a bit more than NeuralLM2, but still less than NeuralLM1.

3.3 Loss vs Epoch graphs

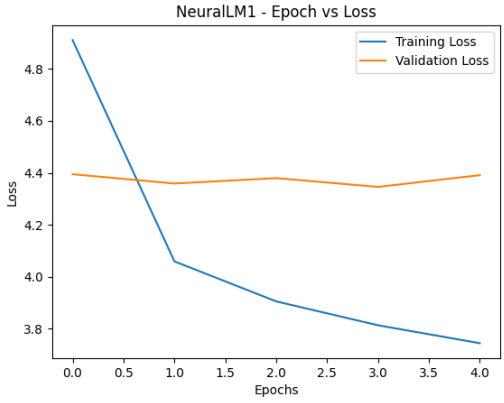

Figure 12: Loss vs Epoch curve for NeuralLM1

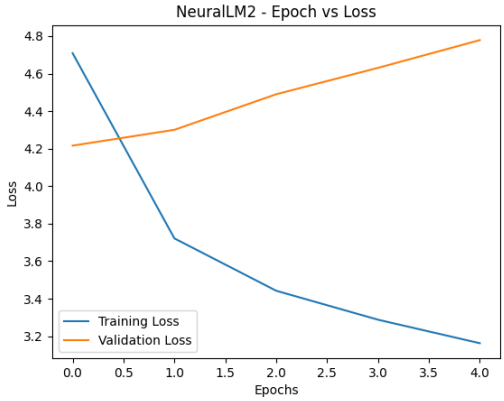

Figure 13: Loss vs Epoch curve for NeuralLM2

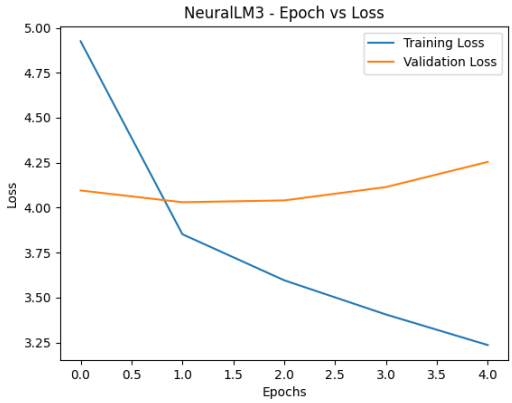

Figure 14: Loss vs Epoch curve for NeuralLM3

3.4 Accuracies and Perplexity scores

Model	Accuracy		Perplexity	
	Training	Validation	Training	Validation
NeuralLM1	36.01%	35.97%	42.27	80.71
NeuralLM2	39.73%	37.81%	23.63	118.94
NeuralLM3	39.17%	39.37%	25.43	70.43

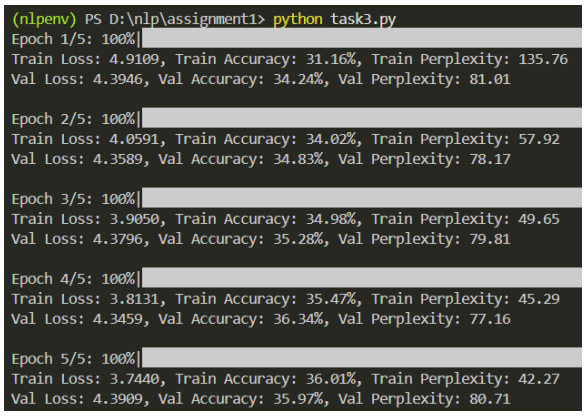

Figure 15: Loss, Accuracy and Perplexity over epochs for NeuralLM1

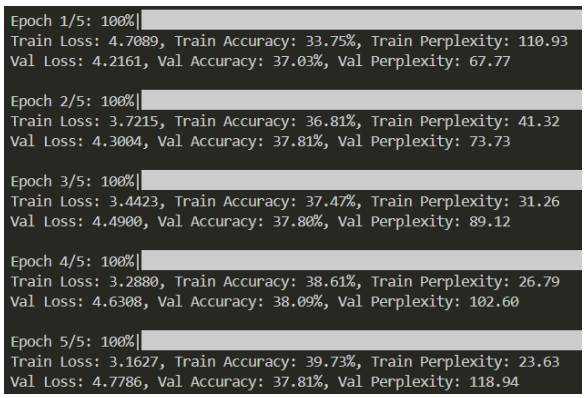

Figure 16: Loss, Accuracy and Perplexity over epochs for NeuralLM2

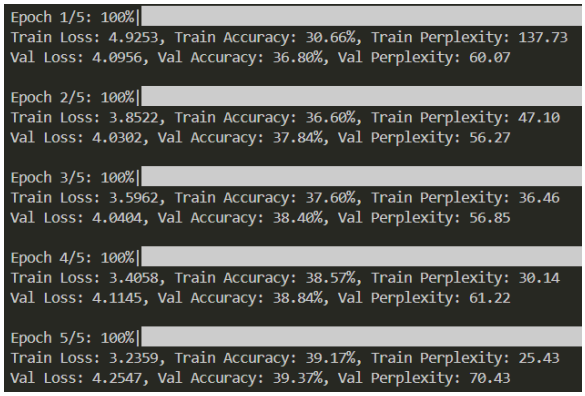

Figure 17: Loss, Accuracy and Perplexity over epochs for NeuralLM3

4 Individual Contribution

- Himanshu Raj: Task 3 implementation and report
- Ishita: Task 2 implementation and report
- Ritika Thakur: Task 1 implementation and report

5 References

- WordPieceTokenizer Tutorial
- Word2Vec paper
- Custom dataset Tutorial
- PyTorch Implementations
- Skip-connections
- Lecture Slides - Language Modelling, Word Representation