NLP Assignment-1

Aarya Gupta 2022006
Aditya Aggarwal 2022028
Arpan Verma 2022105

We pledge our Allegiance to the Ring!

Followed the coding practices/Nomenclature as written in the below Documentation and GitHub Repo:
https://github.com/pytorch/examples/blob/main/word_language_model/model.py
https://pytorch.org/docs/stable/generated/torch.nn.Module.html

TASK-1:

PREPROCESSING

Lowercasing – Converts all characters to lowercase for uniformity.

Removing non-alphabetic characters – Eliminates punctuation, special characters, and numbers, keeping only alphabets.

Removing extra whitespaces – Collapses multiple spaces, tabs, and newline characters into a single space for clean tokenization.

CONSTRUCTING VOCABULARY

The vocabulary construction process begins by splitting the corpus into individual words. Each word is then split into its characters. Every character except the first character of a word is prefixed with the delimiter `##`, representing subword units. Each individual-character token is added to the vocabulary. This forms our initial vocabulary. Two special tokens, `[UNK]` (for unknown words) and `[PAD]` (for padding sequences), are also included. This forms the initial vocabulary. For a large corpus, the size of initial vocabulary would be 52.

The vocabulary is then iteratively expanded by merging the most frequently occurring adjacent token pairs. The frequency of each token and token pair is computed using word occurrences in the corpus, and the most frequent pair, calculated using the below score function, is merged into a new token, which is added to the vocabulary. All the instances

in the split corpus where these merged tokens existed adjacent to each other are then replaced by the new token.

$$score = \frac{frequency\ of\ pair}{(frequency\ of\ first\ token) \times (frequency\ of\ second\ token)}$$

This process continues until the vocabulary reaches the specified size. During each iteration, words in the corpus are updated to replace occurrences of merged pairs with the newly created token. This approach ensures that frequently co-occurring character sequences are merged into meaningful subwords, effectively balancing the trade-off between word-level and character-level tokenization.

TOKENIZATION

The tokenization process begins by splitting the input text into individual words. Each word is then processed sequentially to generate subword tokens using the predefined vocabulary. The algorithm attempts to tokenize each word into the longest possible subword segments found in the vocabulary while preserving meaningful linguistic patterns.

For each word, the process starts from the beginning and searches for the longest matching subword in the vocabulary. If a match is found, the corresponding token is added to the token list, and the search continues for the remaining part of the word. If no valid subword match is found for a portion of the word, the `[UNK]` (unknown) token is assigned, indicating an out-of-vocabulary segment. This step ensures that even unseen tokens can be handled efficiently.

EXAMPLES

Input: `"i cant help but also feel incredibly lucky over how it all went down and the community around us\n"`

Output: `"i", "can", "##t", "h", "##e", "##l", "##p", "but", "a", "##l", "##s", "##o", "f", "##e", "##e", "##l", "i", "##n", "##c", "##r", "##e", "##d", "##i", "##b", "##l", "##y", "l", "##u", "##ck", "##y", "ov", "##e", "##r", "h", "##o", "##w", "i", "##t", "a", "##l", "##l", "w", "##e", "##n", "##t", "d", "##o", "##w", "##n", "a", "##n", "##d", "th", "##e", "community", "a", "##r", "##o", "##u", "##n", "##d", "us"`

Input: `"i feel the most overwhelmed\n"`

Output: "i", "f", "##e", "##e", "##l", "th", "##e", "m", "##o", "##s", "##t", "ov", "##e", "##rwh", "##e", "##l", "##m", "##e", "##d"

Input: "i go shopping i feel like julia roberts in pretty woman\n"

Output: "i", "g", "##o", "shopping", "i", "f", "##e", "##e", "##l", "lik", "##e", "ju", "##l", "##i", "##a", "r", "##o", "##b", "##e", "##r", "##t", "##s", "i", "##n", "p", "##r", "##e", "##t", "##t", "##y", "w", "##o", "##m", "##a", "##n"

TASK-2:

Word2Vec Model Training and Evaluation

Overview

The results presented here are from the training of a Word2Vec model using the Continuous Bag of Words (CBOW) approach. The model was trained for 100 epochs using the Stochastic Gradient Descent (SGD) optimizer. The dataset used for training is derived from the provided corpus, which contains sentences rich in emotional expressions. The goal of this task was to build a Word2Vec model that learns meaningful word embeddings, enabling semantic relationships between words to be captured effectively.

1. Training and Validation Loss Analysis:

Nature of Output:

Training Loss : This represents the error (loss) on the training dataset during each epoch. It measures how well the model is fitting the training data.
Validation Loss : This represents the error on a separate validation dataset, providing an indication of how well the model generalizes to unseen data.

Observations:

Initial Epochs (1–20) :
Both training and validation losses start high (e.g., Train Loss: 9.5814, Val Loss: 9.5253 at Epoch 1).

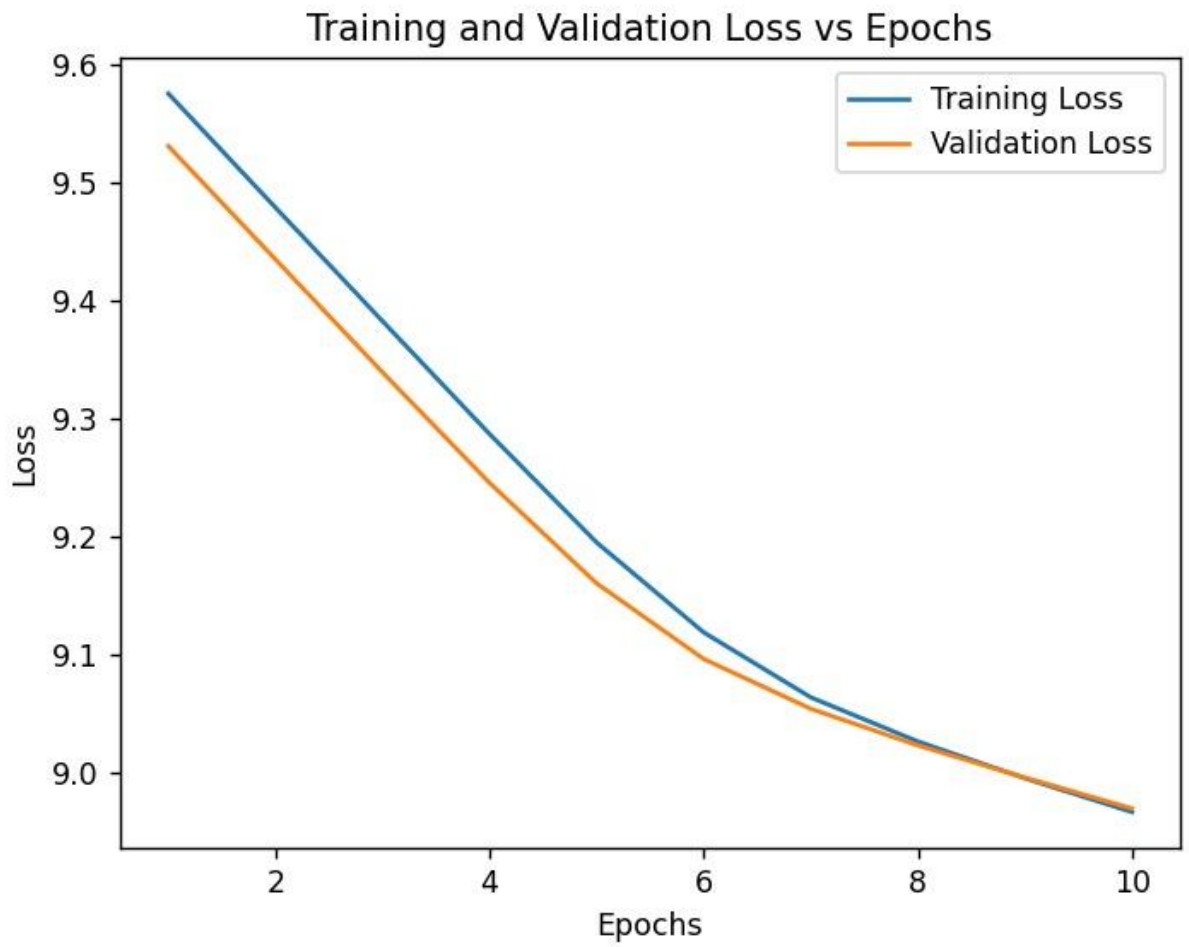
The losses decrease rapidly during the first few epochs, indicating that the model is learning basic patterns in the data.

By Epoch 20, the training loss has reduced to 8.7031, and the validation loss to 8.6995. This sharp decline suggests that the model is making significant progress in capturing word relationships.

Mid-Range Epochs (21–60) :
The rate of decrease in both losses slows down, but they continue to drop steadily.
For example, by Epoch 40, the training loss is 8.2916, and the validation loss is 8.3029. This indicates that the model is refining its understanding of the data. The gap between training and validation losses remains small, suggesting that the model is not overfitting.

Later Epochs (61–100) :
The losses continue to decrease, albeit at a slower pace.
At Epoch 100, the training loss is 7.4845, and the validation loss is 7.5387. The consistent reduction in both losses implies that the model continues to improve even after many iterations.

The small difference between training and validation losses (approximately 0.05) suggests good generalization.



Inferences:

Model Learning :

The gradual reduction in training and validation losses demonstrates that the model is effectively learning meaningful representations of words in the corpus.

The initial rapid decrease in losses indicates that the model quickly captures basic relationships, while the slower decline in later epochs reflects fine-tuning of these relationships.

Generalization :

The small gap between training and validation losses suggests that the model generalizes well to unseen data. This is crucial for ensuring that the learned embeddings are applicable beyond the training dataset.

Convergence :

The losses stabilize as the number of epochs increases, indicating that the model is converging. However, further training might yield marginal improvements, depending on the complexity of the dataset and the model's capacity.

2. Cosine Similarity Analysis:

Nature of Output:

Cosine similarity measures the cosine of the angle between two vectors in a high-dimensional space. It ranges from -1 (completely dissimilar) to +1 (completely similar), with 0 indicating no correlation. In this case, cosine similarity is computed between word embeddings generated by the trained Word2Vec model.

Results:

Similar Pairs :

'happy' and 'joyful' : Cosine similarity = 0.0075
These words are semantically related (both describe positive emotions), so their embeddings should ideally have a high cosine similarity.
The low value (close to 0) suggests that the model has not fully captured the relationship between these words. This could be due to insufficient training or limited context in the corpus.

'lonely' and 'isolated' : Cosine similarity = 0.0043

These words are also semantically related (both describe feelings of disconnection). Again, the low similarity score indicates that the model struggles to capture their relationship.

Dissimilar Pairs :

'happy' and 'sad' : Cosine similarity = -0.0049
These words are opposites, so their embeddings should ideally have a negative cosine similarity. The near-zero value suggests that the model does not strongly distinguish between these antonyms.

'lonely' and 'loved' : Cosine similarity = -0.1772
These words are opposites, and the negative cosine similarity aligns with expectations. However, the magnitude (-0.1772) is relatively small, indicating weak differentiation.
Inferences:

Semantic Relationships :

The low cosine similarity scores for similar pairs ('happy' vs. 'joyful', 'lonely' vs. 'isolated') suggest that the model has not fully captured nuanced semantic relationships. This could be due to:
Insufficient training epochs.

A lack of diverse contexts in the corpus for these words.
Suboptimal hyperparameters (e.g., embedding size, learning rate).

Antonym Differentiation :

The model shows some ability to differentiate antonyms ('lonely' vs. 'loved'), but the strength of differentiation is weak. This indicates that the model's understanding of polarity (positive vs. negative emotions) is limited.

Improvement Opportunities :

Increasing the number of epochs or adjusting the learning rate may help the model refine its embeddings. Using a larger and more diverse corpus could provide richer contexts for the model to learn from. Experimenting with different architectures (e.g., Skip-Gram instead of CBOW) might yield better results.

3. Overall Conclusions:

Strengths:

Loss Reduction :

The consistent reduction in training and validation losses demonstrates that the model is learning effectively and generalizing well.

Basic Semantic Understanding :

The model shows some ability to differentiate between antonyms, indicating a rudimentary understanding of word relationships.

Weaknesses:

1) Limited Semantic Capture :

The low cosine similarity scores for similar pairs suggest that the model struggles to capture nuanced semantic relationships.

2) Weak Antonym Differentiation :

While the model can differentiate antonyms to some extent, the strength of differentiation is not strong enough.

Recommendations:

Hyperparameter Tuning :

Experiment with different learning rates, embedding sizes, and batch sizes to optimize performance.

Dataset Expansion :

Use a larger and more diverse corpus to expose the model to a wider range of contexts.

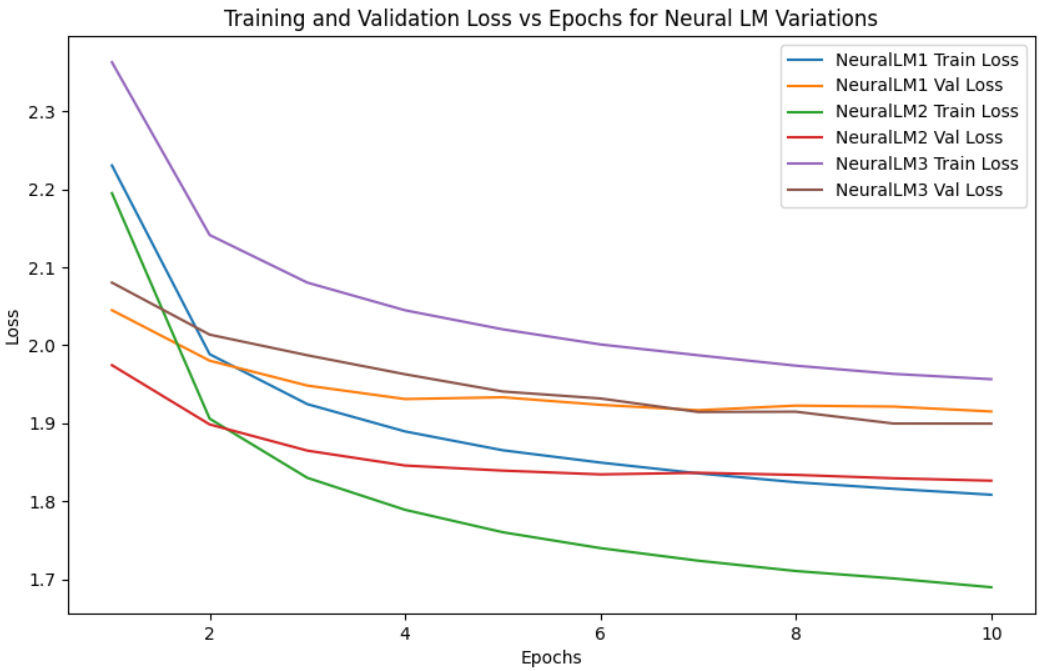
Architecture Exploration :

Try alternative architectures like Skip-Gram or GloVe to see if they yield better results.

Extended Training :

Increase the number of epochs to allow the model more time to refine its embeddings.

TASK-3




```
Next-Token Predictions on Test Data (using NeuralLM3):
Input Sentence: i felt like earlier this year i was starting to feel emotional that it
Predicted next tokens: ['w', '##h', '##e']

Input Sentence: i do need constant reminders when i go through lulls in feeling submiss
Predicted next tokens: ['##e', '##d', 'a']

Input Sentence: i was really feeling crappy even after my awesome
Predicted next tokens: ['##t', '##h', '##i']

Input Sentence: i finally realise the feeling of being hated and its after effects are
Predicted next tokens: ['a', '##n', '##d']

Input Sentence: i am feeling unhappy and weird
Predicted next tokens: ['i', '##n', 'th']
```

```
NeuralLM1 - Accuracy: Train = 49.17%, Val = 46.92%
NeuralLM1 - Perplexity: Train = 5.80, Val = 6.78

NeuralLM2 - Accuracy: Train = 51.56%, Val = 48.23%
NeuralLM2 - Perplexity: Train = 5.11, Val = 6.21

NeuralLM3 - Accuracy: Train = 47.66%, Val = 46.25%
NeuralLM3 - Perplexity: Train = 6.20, Val = 6.68
```

NeuralLM1:

NeuralLM1 serves as a baseline model with a straightforward one-hidden-layer MLP architecture. It leverages pre-trained embeddings and a simple ReLU activation to map context tokens to a target token. This simplicity makes NeuralLM1 computationally efficient, but its limited depth constrains its ability to capture more complex linguistic patterns, often resulting in lower overall performance compared to deeper architectures.

NeuralLM2:

NeuralLM2 introduces a deeper architecture with three hidden layers, each employing the Tanh activation function and interleaved with dropout for regularization. The added depth allows the model to learn richer, more nuanced representations of language, while dropout helps mitigate overfitting by randomly deactivating neurons during training. This architecture strikes a balance between complexity and generalization, leading to improved performance metrics over the baseline.

NeuralLM3:

NeuralLM3 further enhances the model's depth by using three fully connected layers with LeakyReLU activations and dropout regularization. The LeakyReLU activation addresses issues like the "dying ReLU" problem by allowing a small gradient when inactive, ensuring robust learning across all layers.

The performance difference can be clearly seen in the metric results and plots!

Contribution:

We all worked as a Team and split up the work. We also helped each other with logical thinking for each individual task.

Task-1: Aditya Aggarwal
Task-2: Aarya Gupta
Task-3: Arpan Verma