ASSIGNMENT - 1

CSE 556 - Natural Language Processing

Akshat Parmar

Rishi Pendyala

Vimal Jayant Subburaj

2.2.2025
Winter 2025

TASK 1

Lead : Rishi Pendyala (2022403)

WordPiece Tokenizer

The WordPieceTokenizer class implements a WordPiece tokenization algorithm, capable of building a subword vocabulary and tokenizing input text. The three main functions are:

preprocesses_data:

1. Converts to lowercase.
2. Removes punctuation.
3. Removes extra spaces.

construct_vocabulary:

1. Construct initial vocabulary
2. Repeat until vocab size is reached

   (i) Parse pairs of elements

   (ii) Find best pair

   (iii) Merge best pair

   (iv) Update elements

   (v) Update vocab and frequencies
3. Save final vocab

Reference: https://www.youtube.com/watch?v=qpv6ms_t_1A&t=1s
https://huggingface.co/learn/nlp-course/en/chapter6/6

tokenize:

Tokenizes a given sentence into a list of tokens

1. Word in vocab - directly added
2. Word not in vocab - split into chars and append

HELPER FUNCTIONS:

count_frequencies:

Makes a dict of element frequencies

parse_pairs:

Calculates freq of pairs of elements

First iter - pairs from corpus directly

Subsequent iters - account for merged pairs

is_valid_pair:

Checks if the pair is valid.

we cannot form pairs across words - invalid pair

two starting elements - invalid pair

else - valid pair

find_best_pair:

Finds the best pair of elements

Formula: score = freq of pair / freq of first element * freq of second element

merge_elements:

Merges best pair into a new single element

update_elements:

Updates the element list by accounting for merged pairs

A merged pair becomes a new element

save_vocabulary(self):

Save the resulting vocabulary in a text file named vocabulary_66.txt

Assumptions :

The test data is tokenized in lower case

TASK 2

Lead : Vimal Jayant Subburaj (2022571)

Word2VecDataset Class:

This class has 3 main functions: preprocess_data, __len__, __getitem__.

Word2VecModel Class:

Implements Word2Vec CBOW architecture from scratch using PyTorch. It mainly does the following :

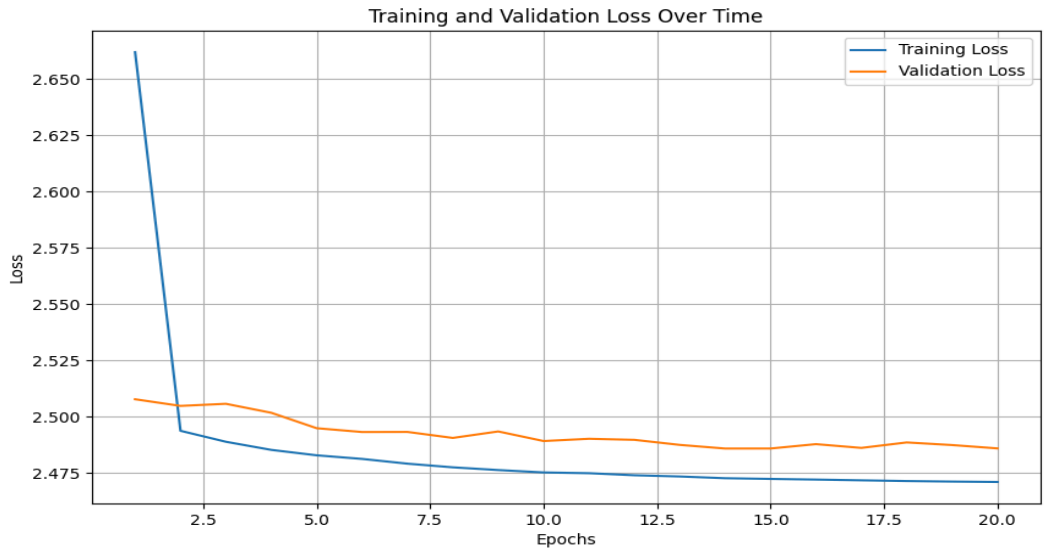
- Initializing the model : It has 2 layers:
    - Embedding layer: Converts word indices to word vectors
    - Linear layer: Linear transformation to vocab_size dimensions.
- Forward pass: Gets embeddings of context words, averages them and passes them through the linear layer.
- Get_word_embeddings: It gets the word embedding of a specified token. If it is not present it will give the embedding of the [UNK] token.

Train Function:

The parameters used to train:

- Loss : Cross Entropy Loss
- Optimizer : Adam Optimizer
- Dataset is divided into train and val with a 80-20 split.
- Padding is also done in this function

Training loss and validation loss versus epochs graph:



Cosine Similarity calculation:

This function was designed to find the similarity between two tokens using cosine similarity.

The formula used is:

$$\text{Cosine Similarity}(A, B) = (A \cdot B) / (||A|| * ||B||)$$

Where:

- **A** and **B** are the word embeddings of token1 and token2 respectively.
- torch.dot(A, B) computes the dot product between the embeddings.
- torch.norm(A) and torch.norm(B) compute the Euclidean norms (magnitudes) of the embeddings.

Find_triplet function:

The function uses the following to identify the triplets:

- **Anchor Token**: A randomly selected token.
- **Most Similar Token**: The token with the highest cosine similarity to the anchor.
- **Least Similar Token**: The token with the lowest cosine similarity to the anchor.

Method Reasoning:

The reason we resorted to a method of having 1 randomly selected anchor token and then finding the most similar token and least similar token with respect to the anchor token is because our vocabulary is of size 10,000 tokens. If we were to compute cosine similarity for all possible pairs we would require:

(10000 x (10000-1)) = 99990000 comparisons. Then comes the computation cost of comparing the similar ones and dissimilar ones and it is very costly computationally.

Triplet 1:

- Anchor: kicking
- Most Similar: lessons (Cosine Similarity: 0.3783)
- Least Similar: christ (Cosine Similarity: -0.3410)

Triplet 2:

- Anchor: ##ationship
- Most Similar: appriciat (Cosine Similarity: 0.3459)
- Least Similar: cit (Cosine Similarity: -0.3515)

References:

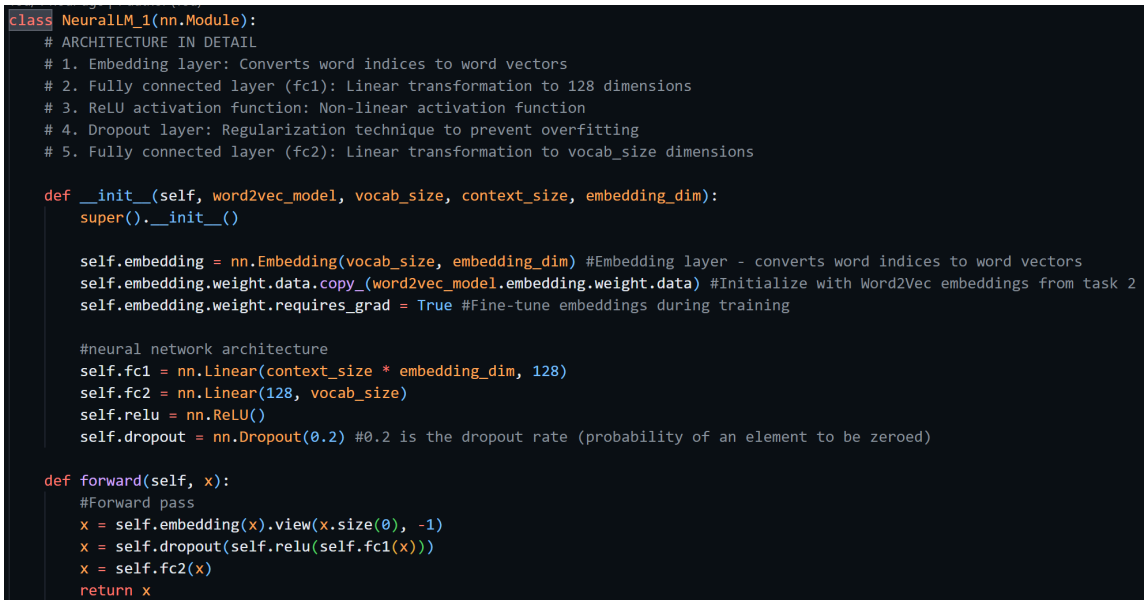
- **https://www.youtube.com/watch?v=Rqh4SRcZuDA : Video taken as reference to understand the architecture**
- **https://medium.com/towards-data-science/word2vec-with-pytorch-implementing-original-paper-2cd7040120b0 :**
- **https://stackoverflow.com/questions/67683406/difference-between-dataset-and-tensordataset-in-pytorch**

TASK 3

Lead : Akshat Parmar(2022050)

Model Architectures

NeuralLM_1:



```python
class NeuralLM_1(nn.Module):
    # ARCHITECTURE IN DETAIL
    # 1. Embedding layer: Converts word indices to word vectors
    # 2. Fully connected layer (fc1): Linear transformation to 128 dimensions
    # 3. ReLU activation function: Non-linear activation function
    # 4. Dropout layer: Regularization technique to prevent overfitting
    # 5. Fully connected layer (fc2): Linear transformation to vocab_size dimensions

    def __init__(self, word2vec_model, vocab_size, context_size, embedding_dim):
        super().__init__()

        self.embedding = nn.Embedding(vocab_size, embedding_dim) #Embedding layer - converts word indices to word vectors
        self.embedding.weight.data.copy_(word2vec_model.embedding.weight.data) #Initialize with Word2Vec embeddings from task 2
        self.embedding.weight.requires_grad = True #Fine-tune embeddings during training

        #neural network architecture
        self.fc1 = nn.Linear(context_size * embedding_dim, 128)
        self.fc2 = nn.Linear(128, vocab_size)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(0.2) #0.2 is the dropout rate (probability of an element to be zeroed)

    def forward(self, x):
        #Forward pass
        x = self.embedding(x).view(x.size(0), -1)
        x = self.dropout(self.relu(self.fc1(x)))
        x = self.fc2(x)
        return x
```

1. **Embedding Layer:** Pre-trained Word2Vec embeddings, fine-tuned during training.
2. **Fully Connected Layers:** Two layers (128 and vocabulary size) for feature transformation and classification.
3. **ReLU Activation & Dropout:** Enables non-linear feature learning and prevents overfitting.

I implemented a pretty basic model just to test the outputs and experiment with different activation functions.

NeuralLM_2:

```python
class NeuralLM_2(nn.Module):
    # ARCHITECTURE IN DETAIL
    # 1. Embedding layer: Converts word indices to word vectors
    # 2. Fully connected layer (fc1): Linear transformation to 256 dimensions
    # 3. Layer normalization: Normalize the output of fc1
    # 5. Dropout layer: Regularization technique to prevent overfitting
    # 6. Fully connected layer (fc2): Linear transformation to 128 dimensions
    # 7. Tanh activation function: Non-linear activation function
    # 9. Fully connected layer (fc3): Linear transformation to vocab_size dimensions

    def __init__(self, word2vec_model, vocab_size, context_size, embedding_dim):
        super().__init__()

        self.embedding = nn.Embedding(vocab_size, embedding_dim) #Embedding layer - converts word indices to word vectors
        self.embedding.weight.data.copy_(word2vec_model.embedding.weight.data) #Initialize with Word2Vec embeddings from task 2
        self.embedding.weight.requires_grad = True #Fine-tune embeddings during training

        #neural network architecture
        self.fc1 = nn.Linear(context_size * embedding_dim, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, vocab_size)
        self.tanh = nn.Tanh()
        self.dropout = nn.Dropout(0.3)
        self.layer_norm = nn.LayerNorm(256)

    def forward(self, x):
        #Forward pass
        x = self.embedding(x).view(x.size(0), -1)
        x = self.dropout(self.layer_norm(self.tanh(self.fc1(x))))
        x = self.dropout(self.tanh(self.fc2(x)))
        x = self.fc3(x)
        return x
```

1. **Embedding Layer:** Same as NeuralLM_1.
2. **Fully Connected Layers:** Three layers (256 → 128 → vocabulary size) for richer context representation and compression.
3. **Layer Normalization & Tanh Activation:** Stabilizes training and ensures smoother gradients.
4. **Dropout:** Higher rate (0.3) to handle increased complexity.

I tried using a deeper architecture to improve representation while addressing gradient issues.

NeuralLM_3:

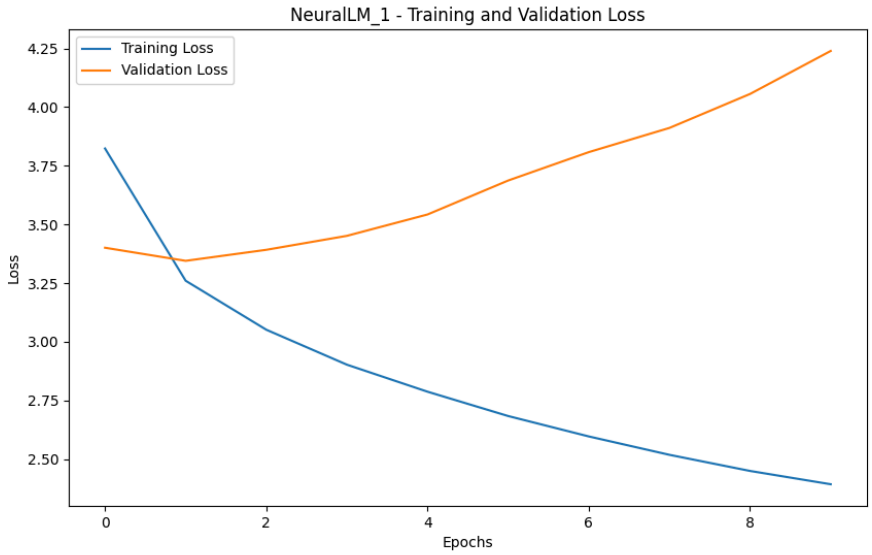
```python
class NeuralLM3(nn.Module):
    # ARCHITECTURE IN DETAIL
    # 1. Embedding layer: Converts word indices to word vectors
    # 2. Fully connected layer (fc1): Linear transformation to 512 dimensions
    # 3. Layer normalization: Normalize the output of fc1
    # 5. Dropout layer: Regularization technique to prevent overfitting
    # 6. Fully connected layer (fc2): Linear transformation to 512 dimensions
    # 7. Layer normalization: Normalize the output of fc2
    # 9. Residual connection: Add the output of fc1 to the output of fc2
    # 10. Fully connected layer (fc3): Linear transformation to 256 dimensions
    # 12. Fully connected layer (fc4): Linear transformation to vocab_size dimensions

    def __init__(self, word2vec_model, vocab_size, context_size, embedding_dim):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.embedding.weight.data.copy_(word2vec_model.embedding.weight.data)
        self.embedding.weight.requires_grad = True
        self.fc1 = nn.Linear(context_size * embedding_dim, 512)
        self.fc2 = nn.Linear(512, 512)
        self.fc3 = nn.Linear(512, 256)
        self.fc4 = nn.Linear(256, vocab_size)
        self.leaky_relu = nn.LeakyReLU(0.1)
        self.dropout = nn.Dropout(0.4)
        self.layer_norm1 = nn.LayerNorm(512)
        self.layer_norm2 = nn.LayerNorm(512)

    def forward(self, x):
        '''
        Forward pass of the model with residual connections
        '''
        x = self.embedding(x).view(x.size(0), -1)
        identity = self.fc1(x)
        x = self.dropout(self.layer_norm1(self.leaky_relu(self.fc1(x))))
        x = self.dropout(self.layer_norm2(self.leaky_relu(self.fc2(x))))
        x = x + identity  #Residual connection

        x = self.dropout(self.leaky_relu(self.fc3(x)))
        x = self.fc4(x)
        return x
```

1. **Embedding Layer:** Similar initialization as other models.
2. **Fully Connected Layers:** Four layers (512 → 512 → 256 → vocabulary size) for high-capacity learning.
3. **Residual Connections:** Improves gradient flow and mitigates vanishing gradients.
4. **Leaky ReLU & Dropout:** Prevents dead neurons and reduces overfitting in the large parameter space.

It is designed for complex representation learning with robust gradient handling and generalization.

Training and Validation Results

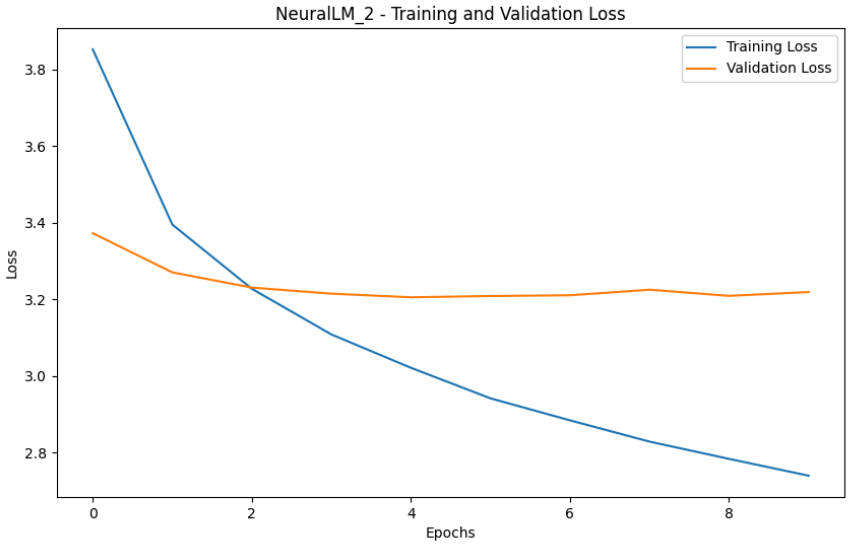
NeuralLM_1



- **Training Loss:** Dropped steadily (3.84 → 2.39).
- **Validation Loss:** Increased consistently after epoch 2 (3.35 → 4.21).
- **Train Accuracy:** 49.10%
- **Validation Accuracy:** 40.81%
- **Perplexity Scores:** Train: 8.24, Validation: 67.15

Despite decent training loss improvement, NeuralLM_1 overfits heavily, as reflected by rising validation loss and perplexity.

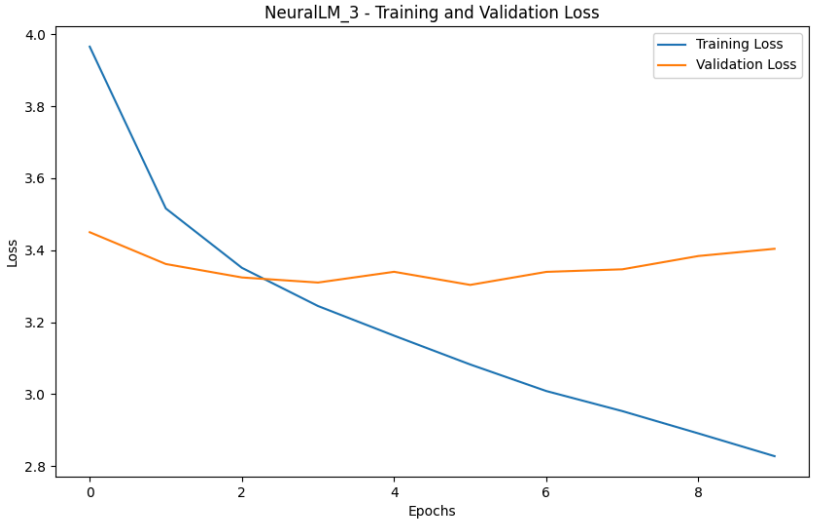
NeuralLM_2



- **Training Loss:** Declined smoothly (3.88 → 2.73).
- **Validation Loss:** Stabilized around 3.22 after epoch 5.
- **Train Accuracy:** 47.63%
- **Validation Accuracy:** 42.21%
- **Perplexity:** Train: 9.90, Validation: 25.08

NeuralLM_2 achieves the best balance between training and validation performance. Validation perplexity indicates better generalization than other models.

NeuralLM_3



- **Training Loss:** Reduced gradually (3.98 → 2.83).
- **Validation Loss:** Was constant and then slightly increased after epoch 6, ending at 3.39.
- **Train Accuracy:** 45.88%
- **Validation Accuracy:** 39.03%
- **Perplexity:** Train: 10.54, Validation: 29.54

NeuralLM_3 struggles with slight underfitting, as training and validation losses are closer but overall accuracy and perplexity lag behind NeuralLM_2.

OUTPUTS:

```
Training NeuralLM_1
Epoch 1/10
Train Loss: 3.8238, Val Loss: 3.4009
Epoch 2/10
Train Loss: 3.2603, Val Loss: 3.3452
Epoch 3/10
Train Loss: 3.0516, Val Loss: 3.3921
Epoch 4/10
Train Loss: 2.9025, Val Loss: 3.4516
Epoch 5/10
Train Loss: 2.7874, Val Loss: 3.5427
Epoch 6/10
Train Loss: 2.6837, Val Loss: 3.6874
Epoch 7/10
Train Loss: 2.5968, Val Loss: 3.8084
Epoch 8/10
Train Loss: 2.5184, Val Loss: 3.9115
Epoch 9/10
Train Loss: 2.4497, Val Loss: 4.0564
Epoch 10/10
Train Loss: 2.3931, Val Loss: 4.2396

NeuralLM_1 Results:
Train Accuracy: 0.48960429242119385
Validation Accuracy: 0.4080117195393009
Train Perplexity: 8.28085708618164
Validation Perplexity: 69.44660186767578
```

```
Training NeuralLM_2
Epoch 1/10
Train Loss: 3.8518, Val Loss: 3.3725
Epoch 2/10
Train Loss: 3.3950, Val Loss: 3.2704
Epoch 3/10
Train Loss: 3.2275, Val Loss: 3.2307
Epoch 4/10
Train Loss: 3.1084, Val Loss: 3.2148
Epoch 5/10
Train Loss: 3.0215, Val Loss: 3.2054
Epoch 6/10
Train Loss: 2.9418, Val Loss: 3.2086
Epoch 7/10
Train Loss: 2.8844, Val Loss: 3.2107
Epoch 8/10
Train Loss: 2.8290, Val Loss: 3.2251
Epoch 9/10
Train Loss: 2.7840, Val Loss: 3.2091
Epoch 10/10
Train Loss: 2.7401, Val Loss: 3.2190

NeuralLM_2 Results:
Train Accuracy: 0.4778537894030852
Validation Accuracy: 0.42175186906445744
Train Perplexity: 9.993494033813477
Validation Perplexity: 25.0150089263916
```

```
Training NeuralLM_3
Epoch 1/10
Train Loss: 3.9659, Val Loss: 3.4500
Epoch 2/10
Train Loss: 3.5161, Val Loss: 3.3616
Epoch 3/10
Train Loss: 3.3510, Val Loss: 3.3243
Epoch 4/10
Train Loss: 3.2448, Val Loss: 3.3101
Epoch 5/10
Train Loss: 3.1624, Val Loss: 3.3400
Epoch 6/10
Train Loss: 3.0823, Val Loss: 3.3035
Epoch 7/10
Train Loss: 3.0084, Val Loss: 3.3397
Epoch 8/10
Train Loss: 2.9526, Val Loss: 3.3469
Epoch 9/10
Train Loss: 2.8905, Val Loss: 3.3839
Epoch 10/10
Train Loss: 2.8273, Val Loss: 3.4039

NeuralLM_3 Results:
Train Accuracy: 0.461046277665996
Validation Accuracy: 0.40134370579915135
Train Perplexity: 10.245501518249512
Validation Perplexity: 30.087905883789062
```

PREDICTIONS:

```
Predicting next tokens for test sentences:

NeuralLM_1 predictions for: i felt like earlier this year i was starting to feel emotional that it
Next three tokens: ['was', 'a', 'lot']

NeuralLM_2 predictions for: i felt like earlier this year i was starting to feel emotional that it
Next three tokens: ['was', 'a', 'good']

NeuralLM_3 predictions for: i felt like earlier this year i was starting to feel emotional that it
Next three tokens: ['is', 'a', 'couple']

NeuralLM_1 predictions for: i do need constant reminders when i go through lulls in feeling submiss
Next three tokens: ['and', 'i', 'f']

NeuralLM_2 predictions for: i do need constant reminders when i go through lulls in feeling submiss
Next three tokens: ['and', 't', '##h']

NeuralLM_3 predictions for: i do need constant reminders when i go through lulls in feeling submiss
Next three tokens: ['and', 't', '##h']

NeuralLM_1 predictions for: i was really feeling crappy even after my awesome
Next three tokens: ['w', '##e', '##r']

NeuralLM_2 predictions for: i was really feeling crappy even after my awesome
Next three tokens: ['was', 't', '##h']

NeuralLM_3 predictions for: i was really feeling crappy even after my awesome
Next three tokens: ['i', 'f', '##e']

NeuralLM_1 predictions for: i finally realise the feeling of being hated and its after effects are
Next three tokens: ['so', 'much', 'i']

NeuralLM_2 predictions for: i finally realise the feeling of being hated and its after effects are
Next three tokens: ['and', 'f', '##e']

NeuralLM_3 predictions for: i finally realise the feeling of being hated and its after effects are
Next three tokens: ['a', 'lot', 'and']

NeuralLM_1 predictions for: i am feeling unhappy and weird
Next three tokens: ['t', '##h', '##e']

NeuralLM_2 predictions for: i am feeling unhappy and weird
Next three tokens: ['t', '##h', '##e']

NeuralLM_3 predictions for: i am feeling unhappy and weird
Next three tokens: ['and', 'i', 'f']
```