NLP Assignment - 1

Task 1: Preprocessing and Vocabulary Construction

1. Preprocessing Applied to the Corpus

Before constructing the vocabulary, the input text (corpus) undergoes a preprocessing phase to prepare it for tokenization. The preprocessing steps are as follows:

Step 1: Lowercasing

All text is converted to lowercase to ensure consistency. This avoids treating words with different cases (e.g., "Apple" and "apple") as distinct tokens.

- **Example:**
  - Original Text: "Apple and apple are the same fruit."
  - After Preprocessing: "apple and apple are the same fruit."

Step 2: Removing Unwanted Characters

All punctuation marks and special symbols are removed. This ensures that the tokenizer focuses on words only, ignoring symbols or non-alphanumeric characters.

- **Example:**
  - Original Text: "This is great! Isn't it?"
  - After Preprocessing: "this is great isnt it"

Step 3: Splitting into Words

The text is then split into individual words based on spaces. This gives us a list of words that are used to build the vocabulary.

- **Example:**
  - Original Text: "apple and orange are fruits"
  - After Preprocessing: ["apple", "and", "orange", "are", "fruits"]

This list of words is used to build the vocabulary and for further tokenization.

2. Building the Vocabulary

The vocabulary is constructed using the WordPiece algorithm, which iteratively merges the most frequent pairs of characters or subwords from the corpus. The steps for vocabulary construction are:

Step 1: Initial Word Splitting

Each word in the corpus is initially split into individual characters (or subwords), where each character is treated as a separate token. Special tokens like [PAD] and [UNK] are also added to the vocabulary for padding and unknown words.

- **Example:**
    - Word: "apple" → ['a', 'p', 'p', 'l', 'e']
    - Word: "banana" → ['b', 'a', 'n', 'a', 'n', 'a']

Step 2: Counting Word Frequencies

The frequency of each word in the corpus is calculated. This is crucial because the merging of pairs is guided by the frequency of the words and subword tokens.

- **Example:**
    - If the word "apple" appears 5 times, its frequency is 5.

Step 3: Pair Merging

The most frequent pairs of adjacent tokens (subwords) are merged into a new token. This merging continues iteratively until the vocabulary reaches a specified size (e.g., VOCAB_SIZE = 5000).

- **Example:**
    - Initial Tokens: ['a', 'p', 'p', 'l', 'e']
    - First Merge: ['a', 'pp', 'l', 'e']
    - Next Merge: ['a', 'ppl', 'e']

This process creates a vocabulary containing subword units, such as ['apple', 'ban', 'na', '##na'], where ## denotes a continuation of a previous subword.

Step 4: Vocabulary Construction

The vocabulary is built by merging the most frequent pairs until the desired size is reached. The final vocabulary includes both individual characters and subwords that were created through merging.

- **Example:**
    - Some possible vocabulary tokens might include: ['apple', 'banana', '##n', '##a', 'fruits', '##s']

Step 5: Final Vocabulary

Once the vocabulary is built, it is sorted alphabetically and saved to a file. This vocabulary is then used for encoding and decoding sentences.

Example: Tokenizing a Sentence

After building the vocabulary, the tokenizer can encode new sentences by breaking them into known tokens.

- **Input Sentence:** "apple and banana are fruits"
- **Preprocessing:** The sentence is converted to lowercase and split into words: ["apple", "and", "banana", "are", "fruits"]
- **Tokenization:** Each word is encoded into subwords using the vocabulary:
    - "apple" → ['apple']
    - "and" → ['and']
    - "banana" → ['banana']
    - "are" → ['are']
    - "fruits" → ['fruits']

Thus, the tokenized sentence is: ['apple', 'and', 'banana', 'are', 'fruits'].

Task 2: Word2Vec Model and Loss Evaluation

Implementation Summary

Vocabulary Construction

The script builds a subword vocabulary using the WordPiece algorithm based on the Byte Pair Encoding (BPE) method. It starts with individual characters and merges the most frequent pairs until the desired vocabulary size (VOCAB_SIZE) is reached. Special tokens like [PAD] and [UNK] are added.

Text Preprocessing

The script preprocesses input text by:

- Converting all characters to lowercase.
- Removing punctuation and special characters using regular expressions.
- Splitting the cleaned text into individual words.

Pair Scoring and Merging

After splitting words into characters, the script counts the frequency of each subword pair and merges the highest-scoring pair into a new subword. The merging continues until the vocabulary reaches the specified size.

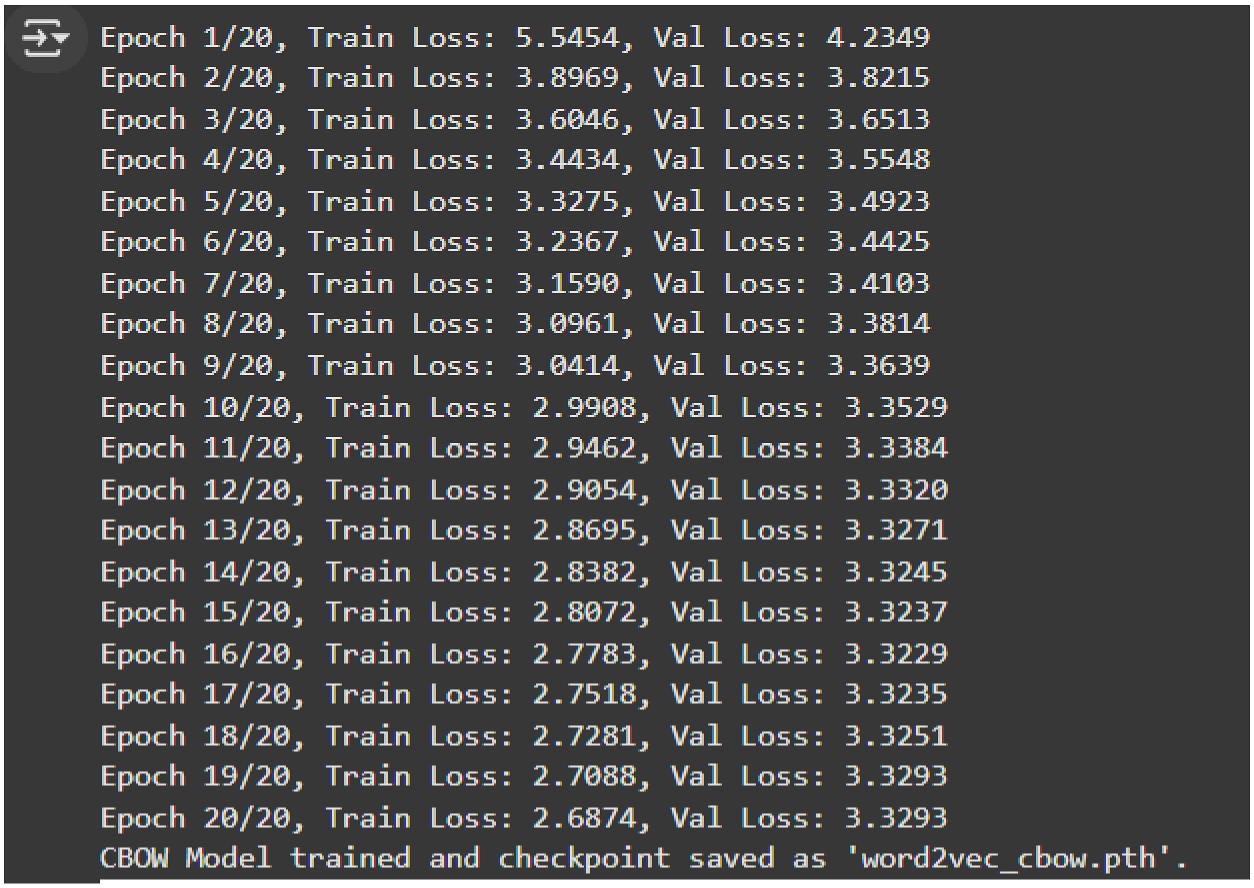
Word Encoding

Once the vocabulary is built, each word is encoded into subwords based on the vocabulary. Words that cannot be fully represented by known subwords are replaced with the [UNK] token.

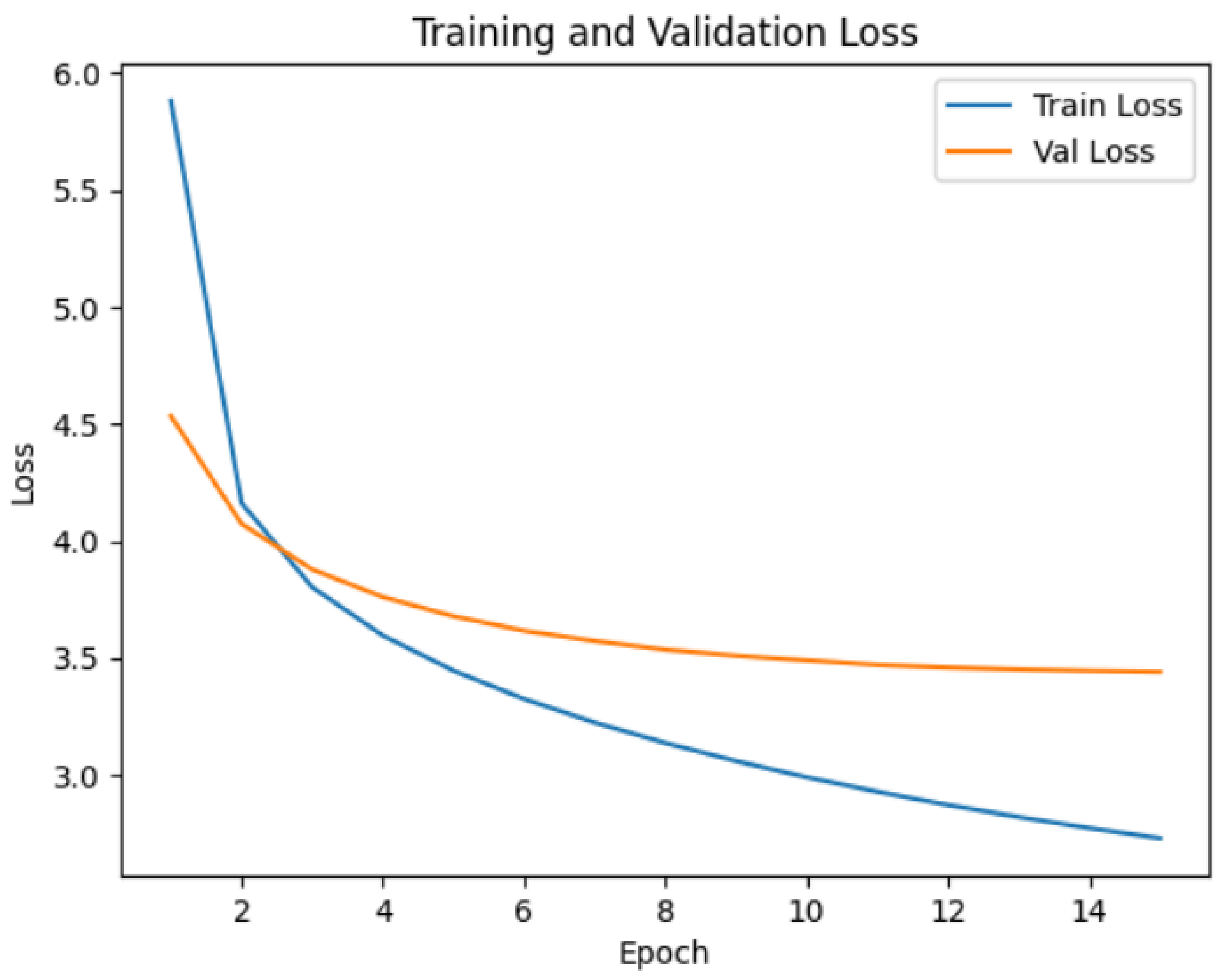
Sentence Tokenization

The script tokenizes entire sentences by breaking each word into subword tokens based on the vocabulary. Padding is handled by adding [PAD] tokens to maintain consistent sentence lengths.

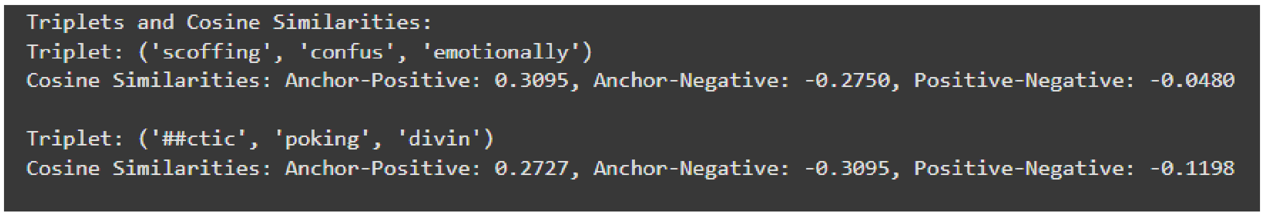
Train and Validation Loss vs. Epoch Graphs



```
Epoch 1/20, Train Loss: 5.5454, Val Loss: 4.2349
Epoch 2/20, Train Loss: 3.8969, Val Loss: 3.8215
Epoch 3/20, Train Loss: 3.6046, Val Loss: 3.6513
Epoch 4/20, Train Loss: 3.4434, Val Loss: 3.5548
Epoch 5/20, Train Loss: 3.3275, Val Loss: 3.4923
Epoch 6/20, Train Loss: 3.2367, Val Loss: 3.4425
Epoch 7/20, Train Loss: 3.1590, Val Loss: 3.4103
Epoch 8/20, Train Loss: 3.0961, Val Loss: 3.3814
Epoch 9/20, Train Loss: 3.0414, Val Loss: 3.3639
Epoch 10/20, Train Loss: 2.9908, Val Loss: 3.3529
Epoch 11/20, Train Loss: 2.9462, Val Loss: 3.3384
Epoch 12/20, Train Loss: 2.9054, Val Loss: 3.3320
Epoch 13/20, Train Loss: 2.8695, Val Loss: 3.3271
Epoch 14/20, Train Loss: 2.8382, Val Loss: 3.3245
Epoch 15/20, Train Loss: 2.8072, Val Loss: 3.3237
Epoch 16/20, Train Loss: 2.7783, Val Loss: 3.3229
Epoch 17/20, Train Loss: 2.7518, Val Loss: 3.3235
Epoch 18/20, Train Loss: 2.7281, Val Loss: 3.3251
Epoch 19/20, Train Loss: 2.7088, Val Loss: 3.3293
Epoch 20/20, Train Loss: 2.6874, Val Loss: 3.3293
CBOW Model trained and checkpoint saved as 'word2vec_cbow.pth'.
```



Triplets identified and explanation of the cosine similarities.



```
Triplets and Cosine Similarities:
Triplet: ('scoffing', 'confus', 'emotionally')
Cosine Similarities: Anchor-Positive: 0.3095, Anchor-Negative: -0.2750, Positive-Negative: -0.0480

Triplet: ('##ctic', 'poking', 'divin')
Cosine Similarities: Anchor-Positive: 0.2727, Anchor-Negative: -0.3095, Positive-Negative: -0.1198
```

Triplet 1: ('scoffing', 'confus', 'emotionally')

- **Anchor-Positive Cosine Similarity (scoffing, confus): 0.3095**
  - The cosine similarity between "scoffing" and "confus" is 0.3095, which is a moderate positive value. This means these two words are somewhat contextually similar in the learned embedding space. "Scoffing" typically refers to mocking or ridiculing, and "confus" (likely referring to "confused") is related to a state of being

perplexed or bewildered. There seems to be some overlapping context between these words, as both could appear in situations involving negative or emotionally charged expressions.

- **Anchor-Negative Cosine Similarity (scoffing, emotionally): -0.2750**
  - The cosine similarity between "scoffing" and "emotionally" is negative, at -0.2750. This indicates that these two words are somewhat dissimilar in context. "Scoffing" is a specific action (mocking or ridiculing), whereas "emotionally" is an adverb that describes a general state. They belong to different parts of speech and have less overlap in the types of contexts they might appear in.
- **Positive-Negative Cosine Similarity (confus, emotionally): -0.0480**
  - The cosine similarity between "confus" and "emotionally" is also negative, but very close to zero (-0.0480). This suggests that there is little to no relationship between these two words in the embedding space. "Confus" refers to a mental state (being confused), while "emotionally" relates to emotional expression or states. They are contextually distant from each other, even though both are associated with human experiences.

Triplet 2: ('##ctic', 'poking', 'divin')

- **Anchor-Positive Cosine Similarity (##ctic, poking): 0.2727**
  - The cosine similarity between "##ctic" (likely a subword token) and "poking" is 0.2727, indicating a moderate positive similarity. "##ctic" is a part of a word like "pictic" or "kinetic", a subword token that can be found as part of many larger words. "Poking" refers to physically nudging or prodding. The connection here may be less direct, but both could appear in contexts where movement or actions are described.
- **Anchor-Negative Cosine Similarity (##ctic, divin): -0.3095**
  - The cosine similarity between "##ctic" and "divin" is negative (-0.3095), suggesting that they are contextually distant. "##ctic" is a part of a word related to motion or kinetics, while "divin" might refer to something related to "divine" or spiritual concepts. These two concepts are likely to appear in very different contexts, hence the negative similarity.
- **Positive-Negative Cosine Similarity (poking, divin): -0.1198**
  - The cosine similarity between "poking" and "divin" is slightly negative (-0.1198), meaning that "poking" (a physical action) and "divin" (likely relating to the divine or something spiritual) are not very similar in meaning. They belong to different conceptual spaces, one grounded in physical actions and the other in metaphysical or spiritual contexts.

Task 3: Neural Language Model Implementation

Overview:

The script trains a **Neural Language Model (LM)** using **Word2Vec embeddings** and a **WordPiece tokenizer** to process the input text. The model aims to predict the next word in a sequence given the context of preceding words. It includes multiple neural network architectures, trains them on a tokenized corpus, and tracks performance using loss curves.

Key Components:

1. **Dataset (NeuralLMDataset)**:
    - **Tokenization**: The input text (corpus) is tokenized using the WordPieceTokenizer from Task 1. The tokenizer splits sentences into subword units (tokens) and maps them to integer indices.
    - **Context-Target Pair Generation**: For each sentence, a sliding window is used to create **context-target pairs**:
        - **Context**: The previous context_size words are used as input to predict the next word in the sequence.
        - **Target**: The next word in the sequence, which the model attempts to predict based on the context.
    - The context words are converted to **Word2Vec embeddings**, which represent each word as a dense vector. These embeddings are used as the input to the model during training.
2. **Neural Language Models**: Three different neural network architectures are implemented:
    - **NeuralLM1**: A single hidden layer (256 units) with **ReLU** activation.
    - **NeuralLM2**: Two hidden layers (256 and 512 units) with **ReLU** activation.
    - **NeuralLM3**: A single hidden layer (256 units) with **Tanh** activation.
3. The architecture takes the flattened embeddings of the context words as input and outputs a probability distribution over the vocabulary for predicting the next word. The output layer has as many units as the vocabulary size, and the activation function is chosen based on the architecture (ReLU or Tanh).
4. **Training the Models**:
    - **Loss Function**: **CrossEntropyLoss** is used because the model is performing multi-class classification, where each word in the vocabulary is a class. The predicted output is a probability distribution, and the actual target word is compared against this distribution.
    - **Optimizer**: **Adam optimizer** is used for updating model parameters. It's popular because it adapts learning rates during training.
    - **Training Loop**:

        - The training process iterates over the dataset for the specified number of epochs (EPOCHS_LM).
        - In each epoch, the model's predictions are compared to the target words, and the loss is computed.
        - The loss is back propagated to update the model's parameters.
        - Both training and validation losses are tracked and reported at the end of each epoch.

5. **Data Loading and Splitting**:
    - The dataset is split into **training** (90%) and **validation** (10%) sets using random_split. This allows the model to be trained on one portion of the data and validated on another portion to monitor its generalization performance.
    - **DataLoader** is used to efficiently load data in batches (BATCH_SIZE_LM = 64) and shuffle the training data at each epoch.
6. **Model Evaluation and Loss Plotting**:
    - After training, the model's performance is evaluated based on both training and validation loss.
    - Loss curves are plotted to visualize how the model's performance changes across epochs. This helps to identify potential overfitting or underfitting.

Justify the design choices for each of the three architectures and explain how each modification impacted the model's performance.

1. NeuralLM1: Single Hidden Layer with 256 Neurons and ReLU Activation

Architecture:

- **Input Layer**: The input consists of a concatenated vector of embeddings for the context words (size: embedding_dim * context_size).
- **Hidden Layer**: A single hidden layer with 256 neurons.
- **Activation Function**: **ReLU (Rectified Linear Unit)** is applied to the output of the hidden layer. It is a widely used activation function that helps to introduce non-linearity and mitigates the vanishing gradient problem.
- **Output Layer**: The output layer predicts the probability distribution over the vocabulary using a softmax-like output (handled internally by **CrossEntropyLoss**).

Justification:

- A **single hidden layer** is a simpler architecture that might capture basic relationships between context and target words.
- **256 neurons** is a moderate size for the hidden layer, balancing the complexity of the model and computational efficiency.

- **ReLU** was chosen because it is computationally efficient and effective at preventing the vanishing gradient problem, which is particularly important when training deep networks.

Impact on Performance:

- The single hidden layer allows the model to capture basic dependencies in the data. This architecture is a good starting point for modeling language, as it provides a balance between simplicity and capacity to learn from context.
- With this configuration, the model performs decently for relatively simple language tasks but may struggle with more complex sequences and long-term dependencies because of its limited depth and the absence of more advanced features like multiple hidden layers or regularization techniques.

---

2. NeuralLM2: Two Hidden Layers (256 and 512 Neurons) with ReLU Activation

Architecture:

- **Input Layer**: Similar to NeuralLM1, the input is a concatenated vector of word embeddings for the context words.
- **First Hidden Layer**: The first hidden layer consists of **256 neurons**, followed by a **ReLU activation**.
- **Second Hidden Layer**: The second hidden layer has **512 neurons**, allowing the model to capture more complex interactions between context words.
- **Output Layer**: The output layer is similar to the previous model, predicting the next word's probability distribution over the vocabulary.

Justification:

- **Two hidden layers** provide more capacity to capture complex relationships between context words and the target word. By adding a second layer, the model is better able to learn non-linear mappings from context to target.
- The first layer uses **256 neurons**, which keeps the model lightweight but still powerful enough to learn meaningful representations.
- The second hidden layer with **512 neurons** increases the model's capacity to handle more complex relationships and interactions between context words.
- **ReLU activation** is used to maintain the non-linearity between layers and ensure efficient training.

Impact on Performance:

- The addition of a second hidden layer improves the model's ability to capture higher-level abstractions from the input data, which could potentially lead to better performance on tasks involving more complex language dependencies.
- This deeper architecture is likely to perform better than the single hidden layer architecture, especially for tasks where relationships between words are more intricate.
- However, it may also be prone to overfitting on smaller datasets due to the increase in the number of parameters.

---

3. NeuralLM3: Single Hidden Layer with 256 Neurons and Tanh Activation

Architecture:

- **Input Layer**: The input is a concatenated vector of word embeddings for the context words, similar to the other models.
- **Hidden Layer**: A single hidden layer with **256 neurons**.
- **Activation Function**: Instead of **ReLU**, this model uses **Tanh** activation. Tanh maps the outputs to a range of [-1, 1], which can lead to a more balanced and smoother transition in the hidden layer output.
- **Output Layer**: The output layer is the same as in the other models, producing a distribution over the vocabulary using softmax.

Justification:

- **Tanh activation** was chosen to examine if the model benefits from a more balanced activation function, as Tanh produces outputs in a bounded range, which may help with gradient flow during training and prevent large updates to weights.
- Using a **single hidden layer** keeps the architecture simple while allowing for the exploration of the impact of activation function changes without adding unnecessary complexity.
- The **256 neurons** in the hidden layer are kept the same as in NeuralLM1 to maintain the same capacity and focus on how the activation function influences performance rather than the number of neurons.

Impact on Performance:

- **Tanh** can lead to more stable training in certain situations, especially when the model requires a balanced output from the hidden layers. However, it can also suffer from the vanishing gradient problem, especially with deeper networks. In this case, the impact on a single-layer architecture may be minimal.
- This architecture might not outperform NeuralLM1 or NeuralLM2 in capturing complex patterns due to the simpler structure and the more conservative activation function. However, it could provide advantages in terms of stability and robustness, particularly in scenarios where the model is prone to large gradients.

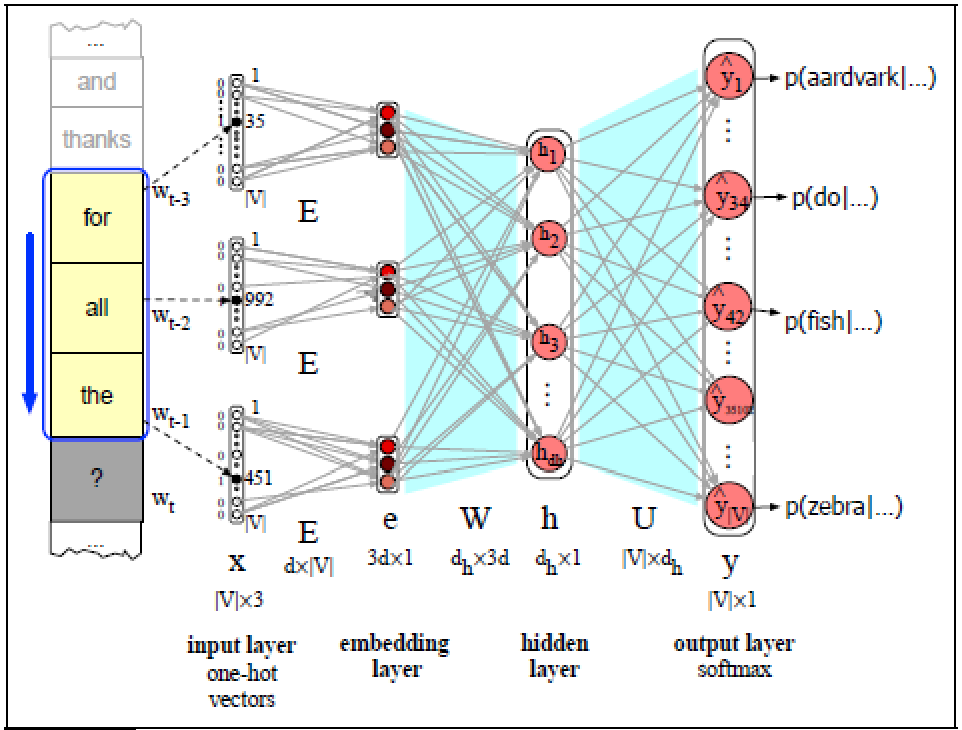
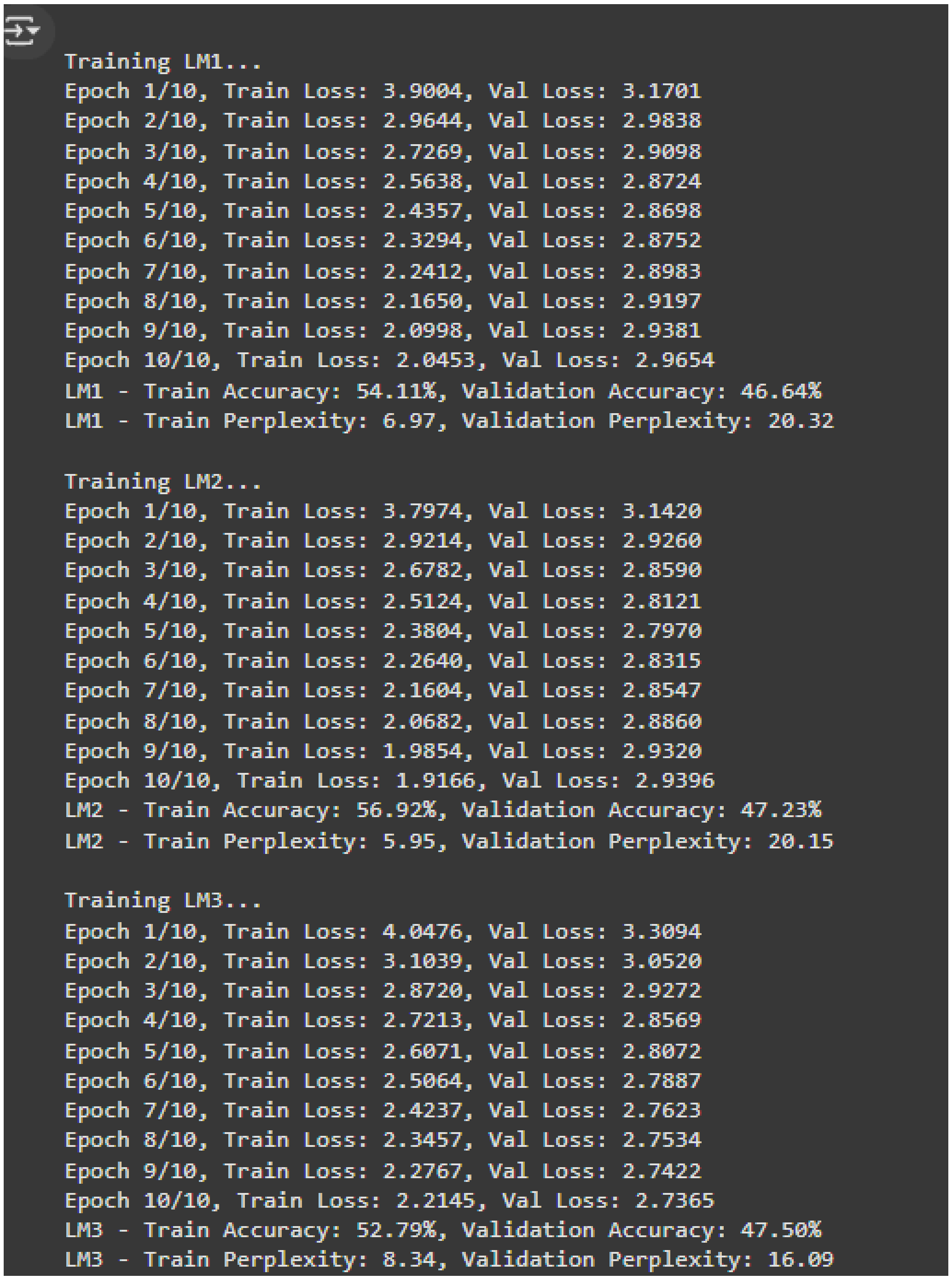


Figure 7.17 Forward inference in a feedforward neural language model. At each timestep t the network computes a d-dimensional embedding for each context word (by multiplying a one-hot vector by the embedding matrix $\mathbf{E}$), and concatenates the 3 resulting embeddings to get the embedding layer $\mathbf{e}$. The embedding vector $\mathbf{e}$ is multiplied by a weight matrix $\mathbf{W}$ and then an activation function is applied element-wise to produce the hidden layer $\mathbf{h}$, which is then multiplied by another weight matrix $\mathbf{U}$. Finally, a softmax output layer predicts at each node i the probability that the next word w_t will be vocabulary word V_i.

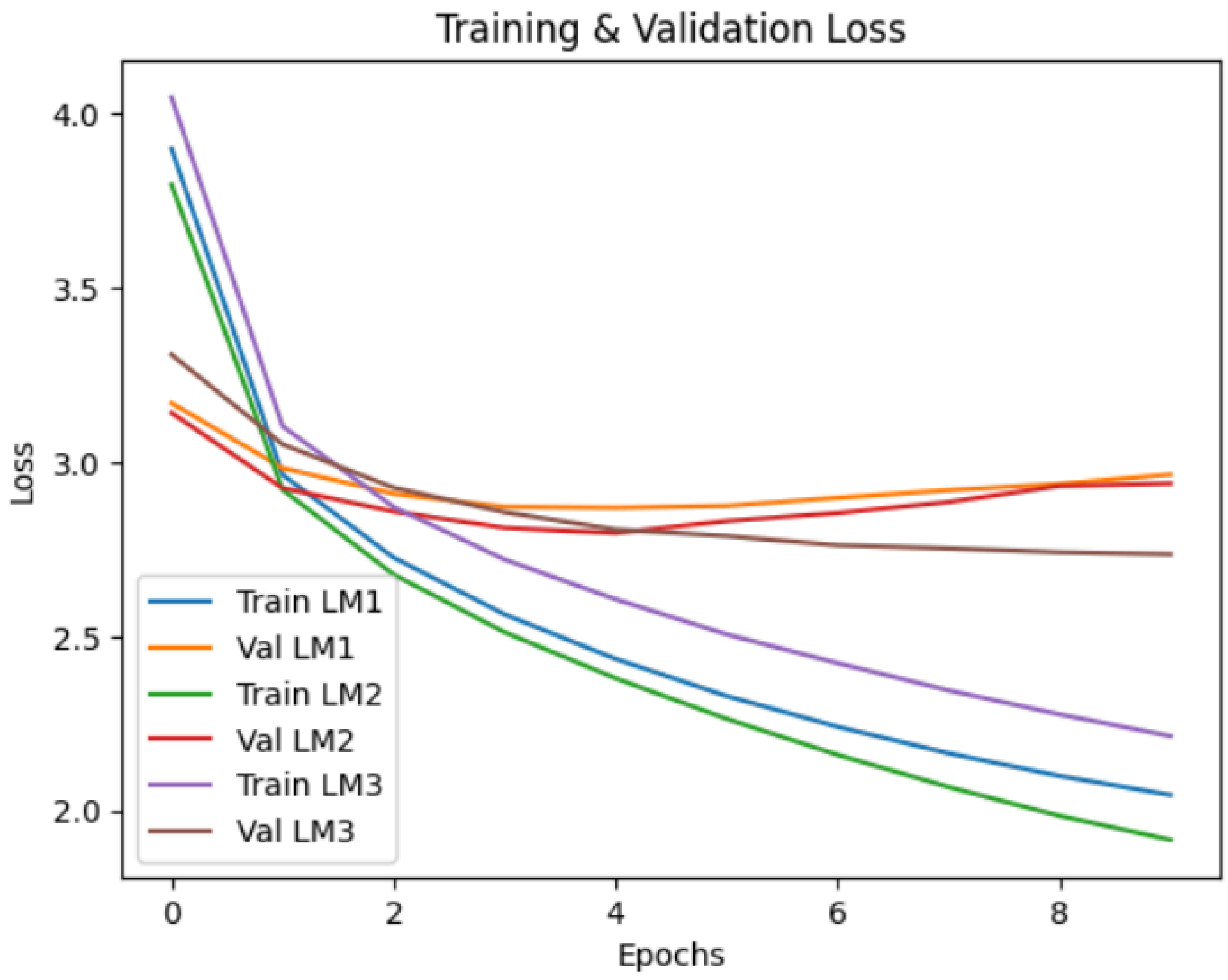
Train and Validation Loss v/s Epoch graphs.



```
Training LM1...
Epoch 1/10, Train Loss: 3.9004, Val Loss: 3.1701
Epoch 2/10, Train Loss: 2.9644, Val Loss: 2.9838
Epoch 3/10, Train Loss: 2.7269, Val Loss: 2.9098
Epoch 4/10, Train Loss: 2.5638, Val Loss: 2.8724
Epoch 5/10, Train Loss: 2.4357, Val Loss: 2.8698
Epoch 6/10, Train Loss: 2.3294, Val Loss: 2.8752
Epoch 7/10, Train Loss: 2.2412, Val Loss: 2.8983
Epoch 8/10, Train Loss: 2.1650, Val Loss: 2.9197
Epoch 9/10, Train Loss: 2.0998, Val Loss: 2.9381
Epoch 10/10, Train Loss: 2.0453, Val Loss: 2.9654
LM1 - Train Accuracy: 54.11%, Validation Accuracy: 46.64%
LM1 - Train Perplexity: 6.97, Validation Perplexity: 20.32

Training LM2...
Epoch 1/10, Train Loss: 3.7974, Val Loss: 3.1420
Epoch 2/10, Train Loss: 2.9214, Val Loss: 2.9260
Epoch 3/10, Train Loss: 2.6782, Val Loss: 2.8590
Epoch 4/10, Train Loss: 2.5124, Val Loss: 2.8121
Epoch 5/10, Train Loss: 2.3804, Val Loss: 2.7970
Epoch 6/10, Train Loss: 2.2640, Val Loss: 2.8315
Epoch 7/10, Train Loss: 2.1604, Val Loss: 2.8547
Epoch 8/10, Train Loss: 2.0682, Val Loss: 2.8860
Epoch 9/10, Train Loss: 1.9854, Val Loss: 2.9320
Epoch 10/10, Train Loss: 1.9166, Val Loss: 2.9396
LM2 - Train Accuracy: 56.92%, Validation Accuracy: 47.23%
LM2 - Train Perplexity: 5.95, Validation Perplexity: 20.15

Training LM3...
Epoch 1/10, Train Loss: 4.0476, Val Loss: 3.3094
Epoch 2/10, Train Loss: 3.1039, Val Loss: 3.0520
Epoch 3/10, Train Loss: 2.8720, Val Loss: 2.9272
Epoch 4/10, Train Loss: 2.7213, Val Loss: 2.8569
Epoch 5/10, Train Loss: 2.6071, Val Loss: 2.8072
Epoch 6/10, Train Loss: 2.5064, Val Loss: 2.7887
Epoch 7/10, Train Loss: 2.4237, Val Loss: 2.7623
Epoch 8/10, Train Loss: 2.3457, Val Loss: 2.7534
Epoch 9/10, Train Loss: 2.2767, Val Loss: 2.7422
Epoch 10/10, Train Loss: 2.2145, Val Loss: 2.7365
LM3 - Train Accuracy: 52.79%, Validation Accuracy: 47.50%
LM3 - Train Perplexity: 8.34, Validation Perplexity: 16.09
```



Discuss any differences in performances across the architectures.

Model Comparison Based on Key Metrics:

1. **Loss:**
    - **LM1** showed the highest training and validation losses at the beginning (around 3.9 and 3.2 respectively), and it decreased more slowly compared to the other models.
    - **LM2** had a similar trajectory but slightly better performance in terms of both training and validation loss.
    - **LM3**, despite starting with a higher training loss (4.04), showed a smoother decrease, resulting in lower validation loss by the end (around 2.74).

2. This suggests that **LM3** generally converged faster, and despite the initial higher loss, it stabilized more effectively on both the training and validation sets.
3. **Accuracy:**
    - **LM2** performed best, with **56.92% training accuracy** and **47.23% validation accuracy**.
    - **LM1** followed closely with **54.11% training accuracy** and **46.64% validation accuracy**.
    - **LM3** had the lowest accuracy with **52.79% training accuracy** and **47.50% validation accuracy**.
4. Despite LM3's faster loss reduction, its accuracy was lower than LM2's. However, LM3's validation accuracy slightly outperformed LM1, which could be indicative of better generalization.
5. **Perplexity:**
    - **LM2** achieved the best perplexity score with **5.95** on the training set and **20.15** on the validation set.
    - **LM1** had slightly higher perplexity values: **6.97** on the training set and **20.32** on the validation set.
    - **LM3** displayed higher perplexity, with **8.34** on the training set and **16.09** on the validation set.
6. Lower perplexity typically correlates with better model performance, and once again, **LM2** consistently had the lowest perplexity across both training and validation.

Insights from Model Architectures:

1. **NeuralLM1 (Single Hidden Layer [256], ReLU)**:
    - This model was relatively simple with only one hidden layer. Although it showed decent accuracy and loss reduction, it could benefit from more complexity to capture more intricate patterns in the data. The validation performance seems to plateau somewhat early compared to the other models, which may indicate that it doesn't have the capacity to model the data as well as the others.
2. **NeuralLM2 (Two Hidden Layers [256, 512], ReLU)**:
    - The addition of a second hidden layer seems to provide the model with better capacity, as it resulted in both lower loss and better accuracy/perplexity scores. The deeper architecture allows the model to capture more complex relationships in the data. This made it the most effective overall, especially in terms of validation accuracy and perplexity.
3. **NeuralLM3 (Single Hidden Layer [256], Tanh)**:
    - The use of Tanh activation instead of ReLU might have contributed to more stable gradients, but the performance was not as strong as LM2's. Despite this, it showed faster convergence and slightly better generalization (validation loss/accuracy) compared to LM1. This suggests that Tanh may be more effective in certain contexts, though overall the deeper architecture in LM2 still outperforms it.

Conclusions:

- **LM2 (Two Hidden Layers)** was the best performing model in terms of both training and validation metrics, including loss, accuracy, and perplexity. The additional complexity of the two hidden layers helped the model capture more nuanced patterns, resulting in better performance.
- **LM1 (Single Hidden Layer)**, while simpler, provided good results but seemed to hit a performance plateau more quickly, indicating it might not be deep enough to fully learn the complexity of the language data.
- **LM3 (Single Hidden Layer with Tanh)** had slightly better generalization than LM1, but its performance was still not on par with LM2. The Tanh activation worked reasonably well, but the extra hidden layer in LM2 likely helped it capture more meaningful features.

Ultimately, the **architecture with two hidden layers (LM2)** is the optimal choice in this scenario, as it strikes a good balance between complexity and performance.

Individual Contributions

- **Krishna Shukla:** Understanding the logic and workflow for all task and coding in Task 2 and Task 3
- **Harsh Rajput:** Understanding the logic and workflow for all task and coding Task 1 and Task 2
- **Varun Kumar:** Understanding the logic and workflow for all task and coding Task 3 and Task 1.

References

1. WordPiece Tokenizer on Hugging Face
2. Jurafsky & Martin's Speech and Language Processing
3. PyTorch Documentation
4. CBOW implementation - YouTube