**CFG-Based Sentence Parsing using Neural Network POS Tagging**

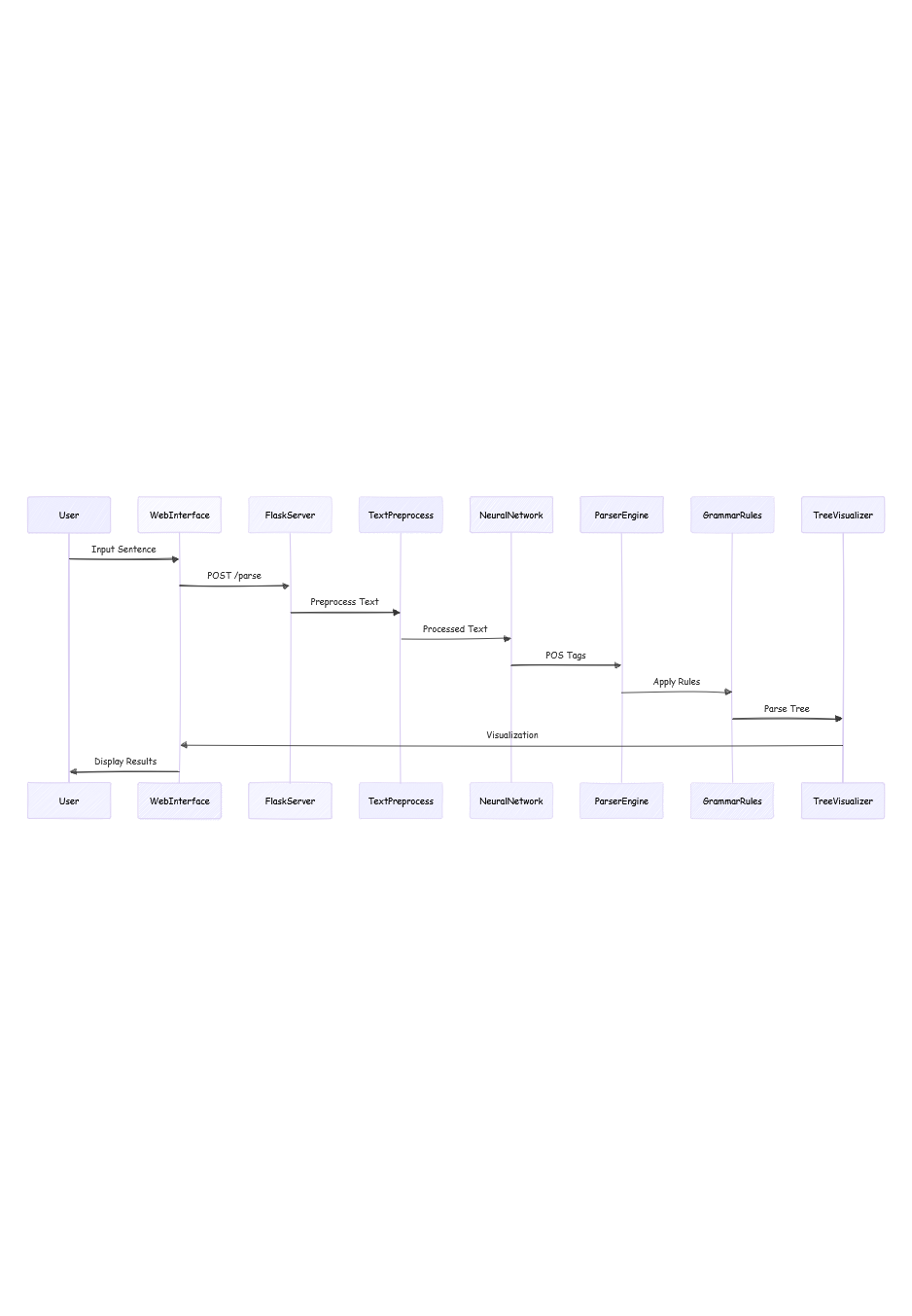
**1. Introduction**

Parsing natural language syntax is a core task in Natural Language Processing (NLP). This project implements a hybrid sentence parser that combines deep learning-based POS tagging with rule-based Context-Free Grammar (CFG) parsing.

The system performs the following steps:

1. Takes a user-input sentence
2. Tags each word with its POS using a trained bidirectional LSTM model
3. Applies a defined CFG to generate a parse tree
4. Visualizes the tree using Graphviz

This approach leverages the strengths of both statistical and rule-based NLP methods.



**Key Concepts and Tools Used:**

1. **Tokenization**:

* Can use different tokenization strategies (word-level, subword-level, character-level)
* Important for handling out-of-vocabulary words
* Affects model's ability to understand rare words

1. **Padding**:

* Essential for batch processing in neural networks
* Can use different padding strategies (pre-padding, post-padding)
* Often combined with masking to ignore padding tokens

1. **Embedding Layer**:

* Learns semantic relationships between words
* Can be pre-trained (Word2Vec, GloVe) or trained from scratch
* Dimensionality affects model capacity and training time

1. **Bidirectional LSTM**:

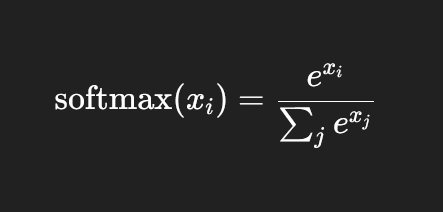
* Captures context from both directions
* Useful for tasks requiring full sentence understanding
* More parameters than unidirectional LSTM

1. **Softmax Layer**:

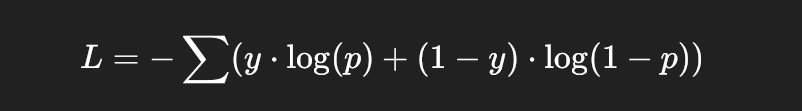
* Converts logits to probabilities
* Used for multi-class classification
* Often combined with cross-entropy loss for training

**Formulae:**

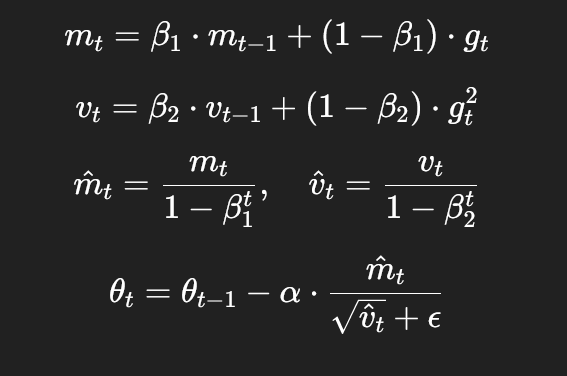
1. **Softmax Function**:



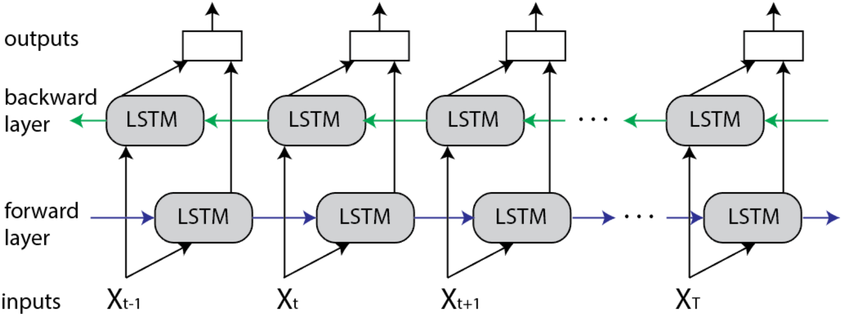
1. **Cross-Entropy Loss**:



1. **Adam Optimizer Update Rules**:



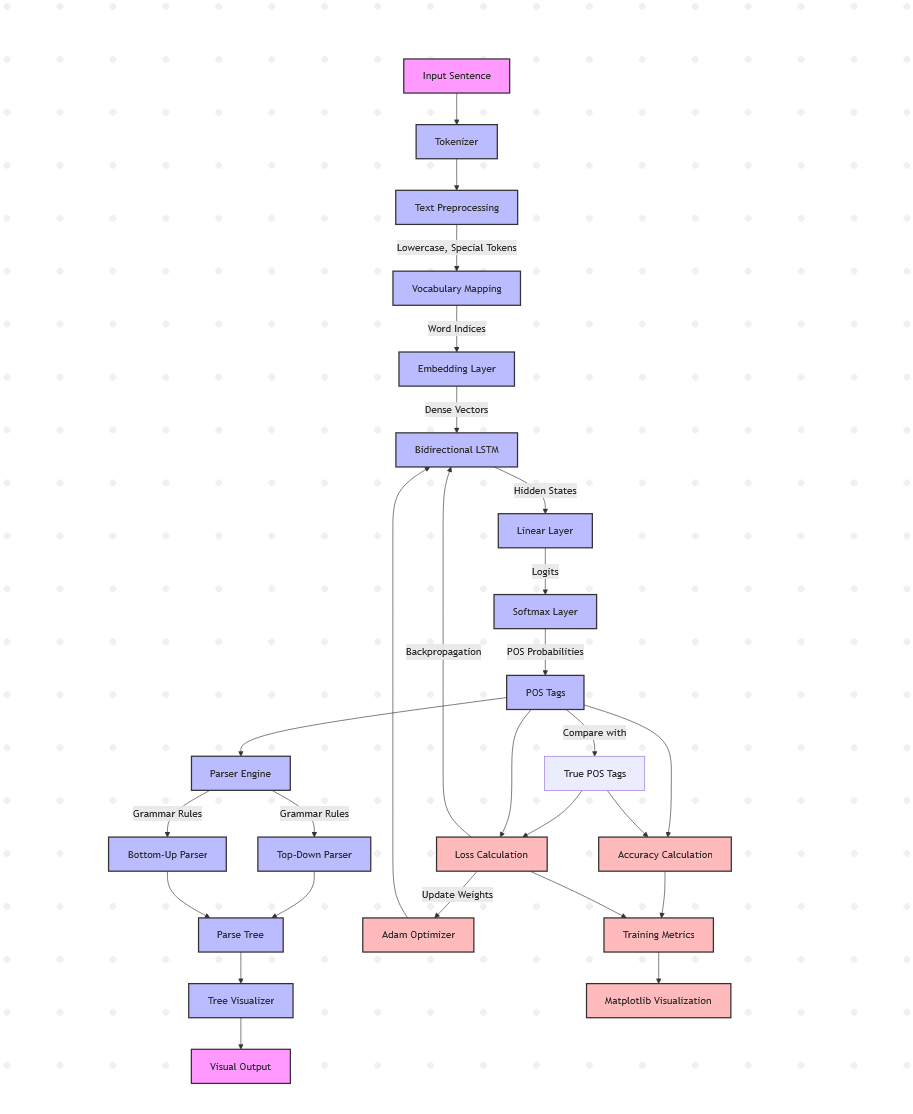
**BiLSTM Architecture:**



Our neural network model for POS tagging utilizes categorical cross-entropy loss to measure prediction errors and accuracy as the primary performance metric. We employ matplotlib to visualize the training progress through loss and accuracy curves, while Graphviz is used to generate the hierarchical parse tree visualization that represents the syntactic structure of parsed sentences.

This sentence captures:

1. The use of categorical cross-entropy for our multi-class POS tagging task.
2. Accuracy as our main evaluation metric
3. Matplotlib for training visualization (loss/accuracy plots)
4. Graphviz for parse tree visualization



**2.Dataset:**

**1. Source & Format**

The dataset is sourced from the Penn Treebank via NLTK, offering rich linguistic structure. It contains raw text, part-of-speech (POS) tags, and syntactic parse trees, all represented using a bracketed tree format to preserve hierarchical structure.

**2. Processing Pipeline**

Preprocessing involves NLTK's Punkt sentence tokenizer followed by word-level tokenization. Each word is converted to lowercase for uniformity. Special tokens like <unk> (unknown) and <pad> (padding) are introduced for out-of-vocabulary handling and batch alignment. Padding ensures that sequences are of equal length for batch processing.

**3.Tag Mapping System**

A consistent mapping system is implemented to convert both words and POS tags into index values, which facilitates efficient model training.

**4.Statistics**

The dataset consists of approximately 40,000 sentences, containing around 10,000 unique words and 47 unique POS tags. The average sentence length ranges from 15 to 20 words, with the maximum length capped at 100 words (padded as needed).

**5. Data Splits**

The dataset is divided into three parts: 80% for training, 10% for validation, and 10% for testing, ensuring robust model evaluation.

**6. Tag Distribution**

The POS tags are distributed across all parts of the dataset, covering a wide range of grammatical categories for comprehensive learning.

**7. Quality & Coverage**

The data is professionally annotated and sourced from multiple domains, including news articles, technical documents, and general text. It effectively handles special cases like proper nouns, numerical data, dates, punctuation, and abbreviations, contributing to its high linguistic

coverage.

**8. Storage**

All relevant mappings are stored in JSON files: word\_to\_idx.json and tag\_to\_idx.json. Model-related data such as weights, mappings, and training statistics are saved as checkpoints for easy reusability and evaluation.

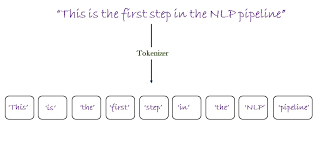
**3. Text Processing**

**1.Preprocessing:**

* All input sentences are converted to lowercase to ensure consistency and avoid treating the same word in different cases (e.g., The vs the) as different tokens.
* Special tokens like <unk> are used to handle out-of-vocabulary (OOV) words that were not seen during training, while <pad> is used to maintain consistent input length.
* Punctuation marks and special characters that do not contribute to the syntactic structure are removed to reduce noise in the data and help the model generalize better.

**2. Tokenization:**

* Sentences are split into individual words or tokens using standard tokenizers like those provided by NLTK.
* These tokens are then mapped to their respective indices using a vocabulary dictionary created during training.
* This step translates the textual input into a numerical format suitable for neural network processing.



**3. POS Mapping:**

* The raw POS tags from the dataset (like VBP, NNPS, etc.) are often too detailed or inconsistent with CFG rules, so they are mapped to a reduced, compatible tag set.
* For example, VBP (verb, non-3rd person singular present) is converted to VBZ (verb, 3rd person singular present), and NNS (plural noun) is mapped to NN (singular noun).
* This mapping ensures that the neural network's output aligns with the parser's grammar structure.

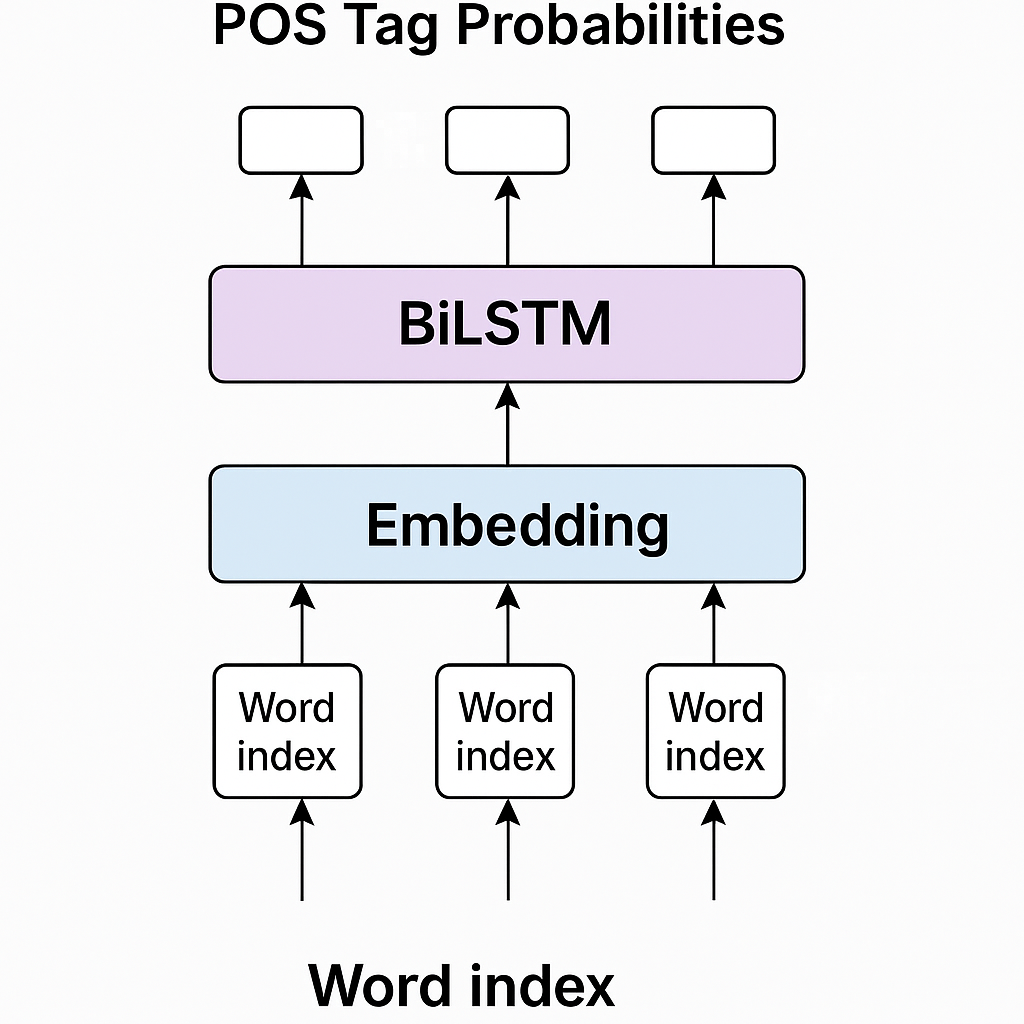
**4. Sequence Padding:**

* Neural networks require input sequences to have the same length for batch processing, so shorter sequences are padded with the <pad> token.
* This allows efficient training by batching sentences together and maintaining alignment across time steps in the LSTM.
* Padding is usually added at the end of the sentence (post padding) to preserve the beginning of the context.

**4. Parsing Algorithm**

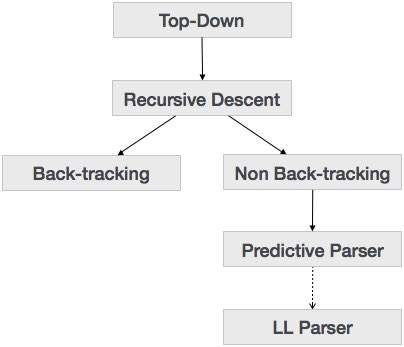
**POS Tagging Model:**

* The model starts by converting input word indices into dense vectors using an embedding layer that captures semantic similarity.
* These embeddings are passed through a bidirectional LSTM that learns contextual relationships in both forward and backward directions.
* Finally, a linear layer with softmax outputs the probability distribution over possible POS tags for each word in the sentence.

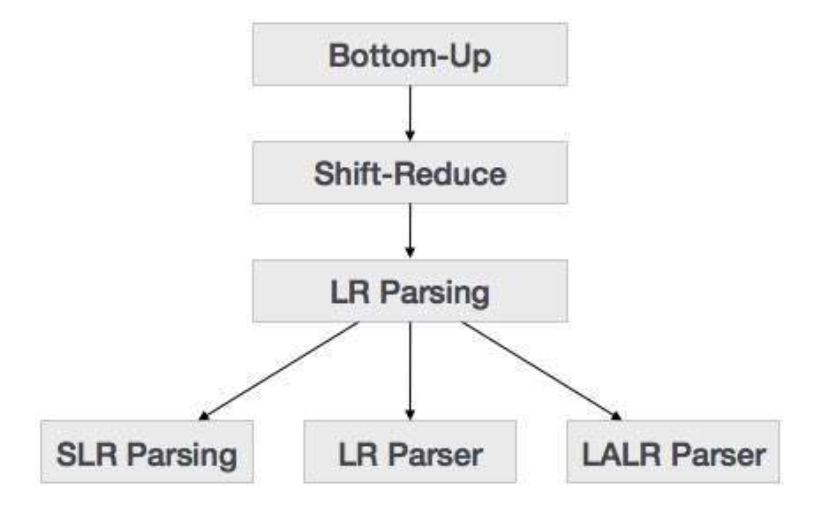


**Parsing Approaches:**

* **Top-Down Parsing**: This strategy begins with the start symbol 'S' and recursively applies grammar rules to expand non-terminal symbols until the terminal symbols match the input tokens. It is intuitive but may involve backtracking if a chosen rule path fails.



* **Bottom-Up Parsing:** This strategy starts with the input words and attempts to combine them into higher-level structures by matching sequences to grammar rules. It uses a shift-reduce mechanism to build the parse tree from leaves to root and is often more efficient for certain sentence types.



**5. Code Overview**

The implementation is structured as a Flask-based web application that integrates PyTorch for neural network inference and Graphviz for visualization. The code is divided into two main components: the POS tagger and the CFG parser logic.

**Backend Structure**

* parser\_trainer.py handles data loading, preprocessing, model definition, training, and saving of the POS tagger model.
* app.py serves as the main Flask backend that loads the model, receives user input, processes the input, and returns parsed results.

**POS Tagging Module**

* Loads the trained PyTorch model and vocabulary mappings (word\_to\_ix, tag\_to\_ix) from disk.
* Input sentence is tokenized, converted to indices, and passed to the model to get predicted POS tags.
* Uses a softmax layer to determine the most probable tag for each word.

**Parsing Logic**

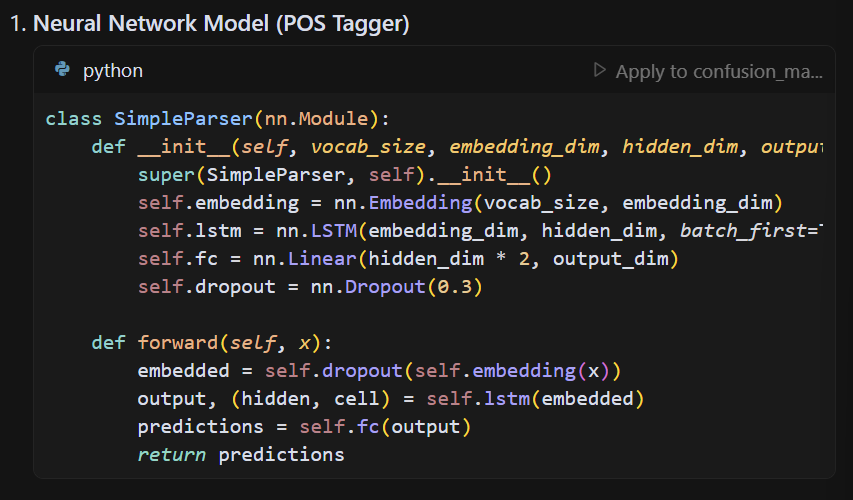
* Once tags are predicted, the parser uses CFG rules to attempt both top-down and bottom-up parsing.
* Each strategy tries to construct a valid parse tree from the sequence of POS tags based on the defined grammar.
* Recursive functions and stack-based logic are used to build and traverse the tree structures.

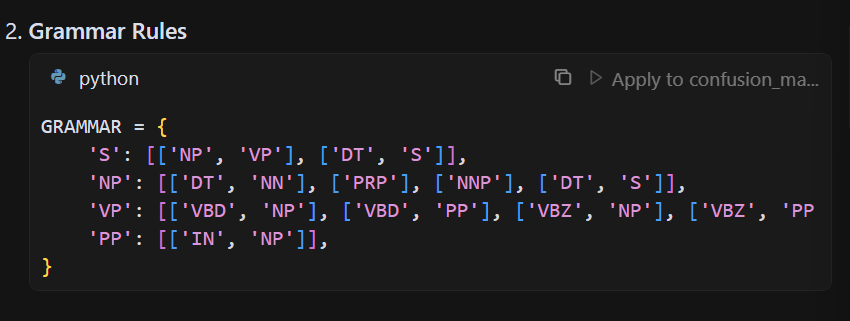
**Visualization**

* A recursive algorithm generates a .dot representation of the parse tree structure.
* Graphviz is used to render this .dot file into a PNG image, which is served to the frontend.
* The user interface then displays the generated parse tree for visual understanding.

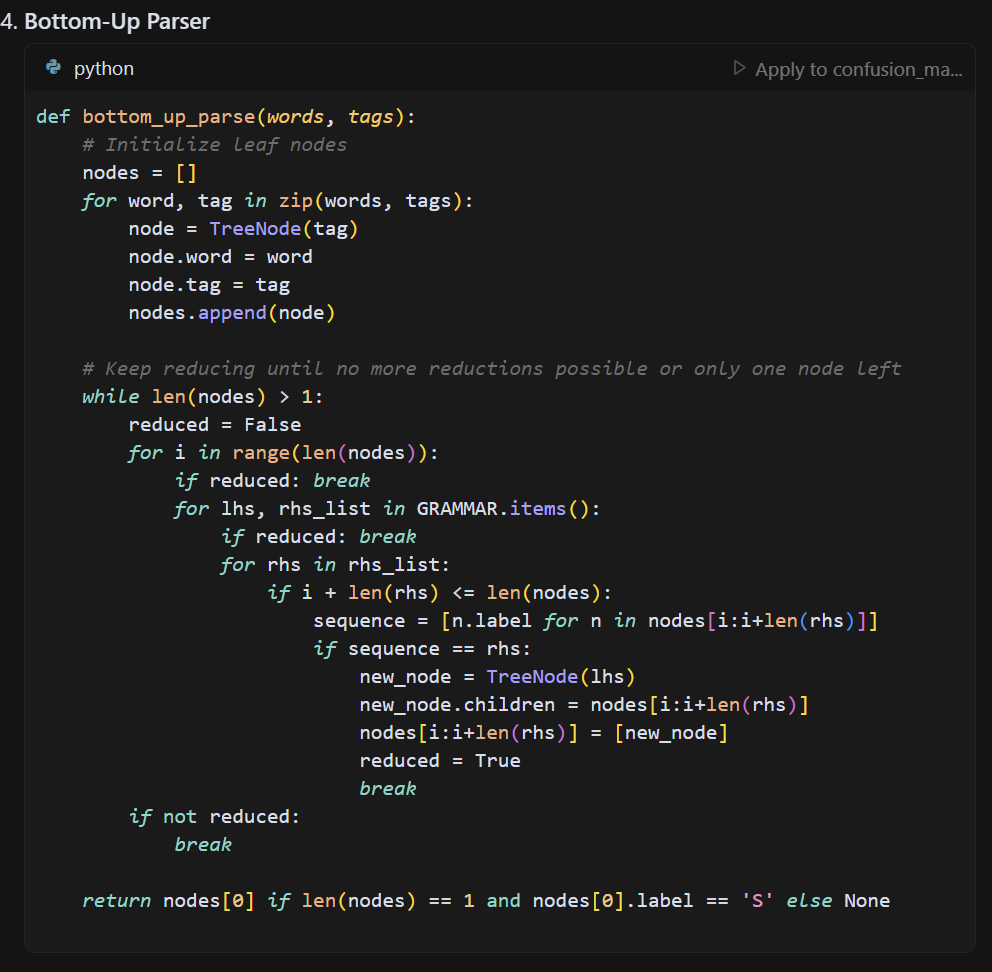
**Frontend Interaction (via Flask)**

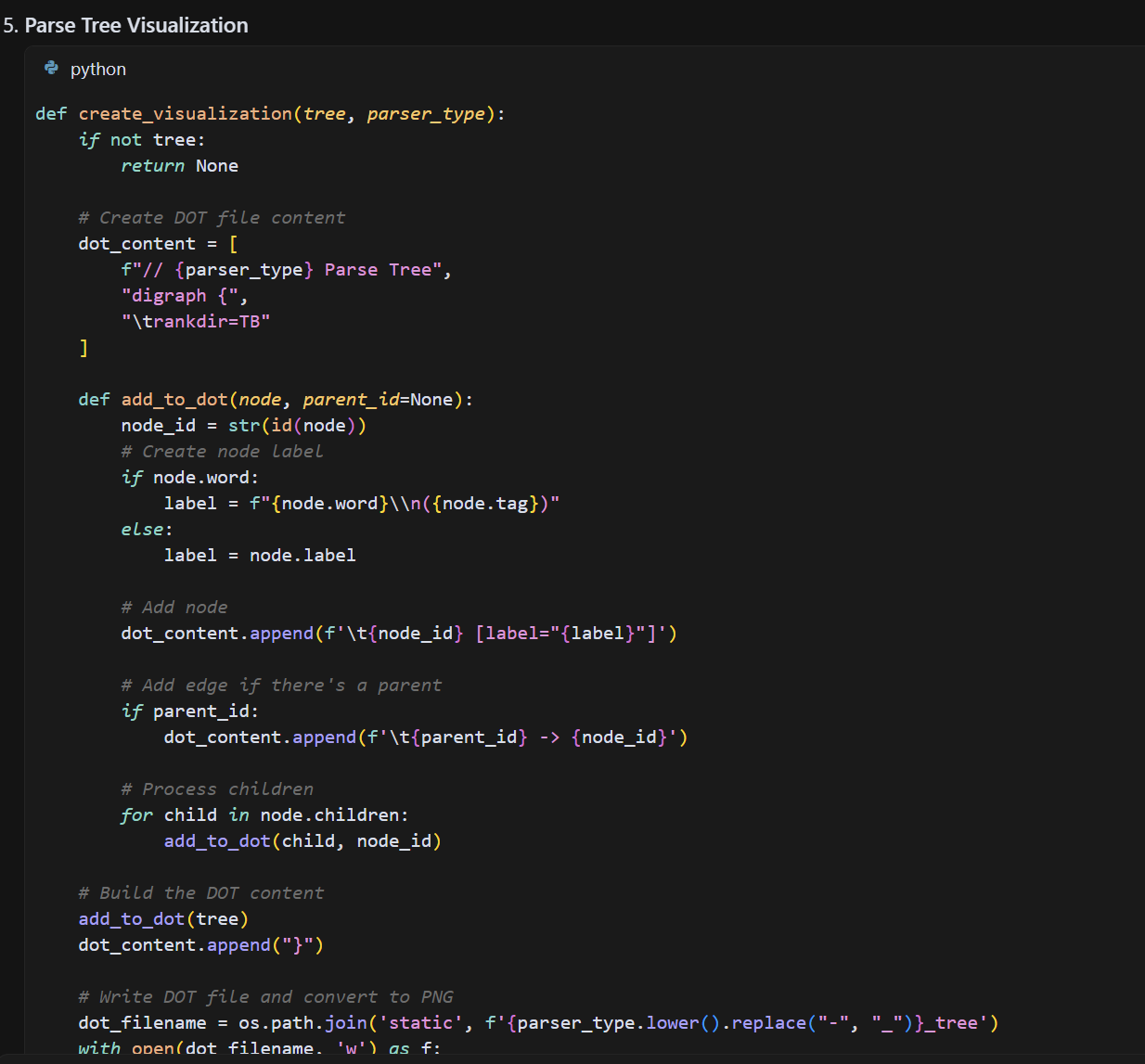
* Users input a sentence through a web interface.
* Flask routes the request to the parser, which processes it and returns the parse result and tree image.

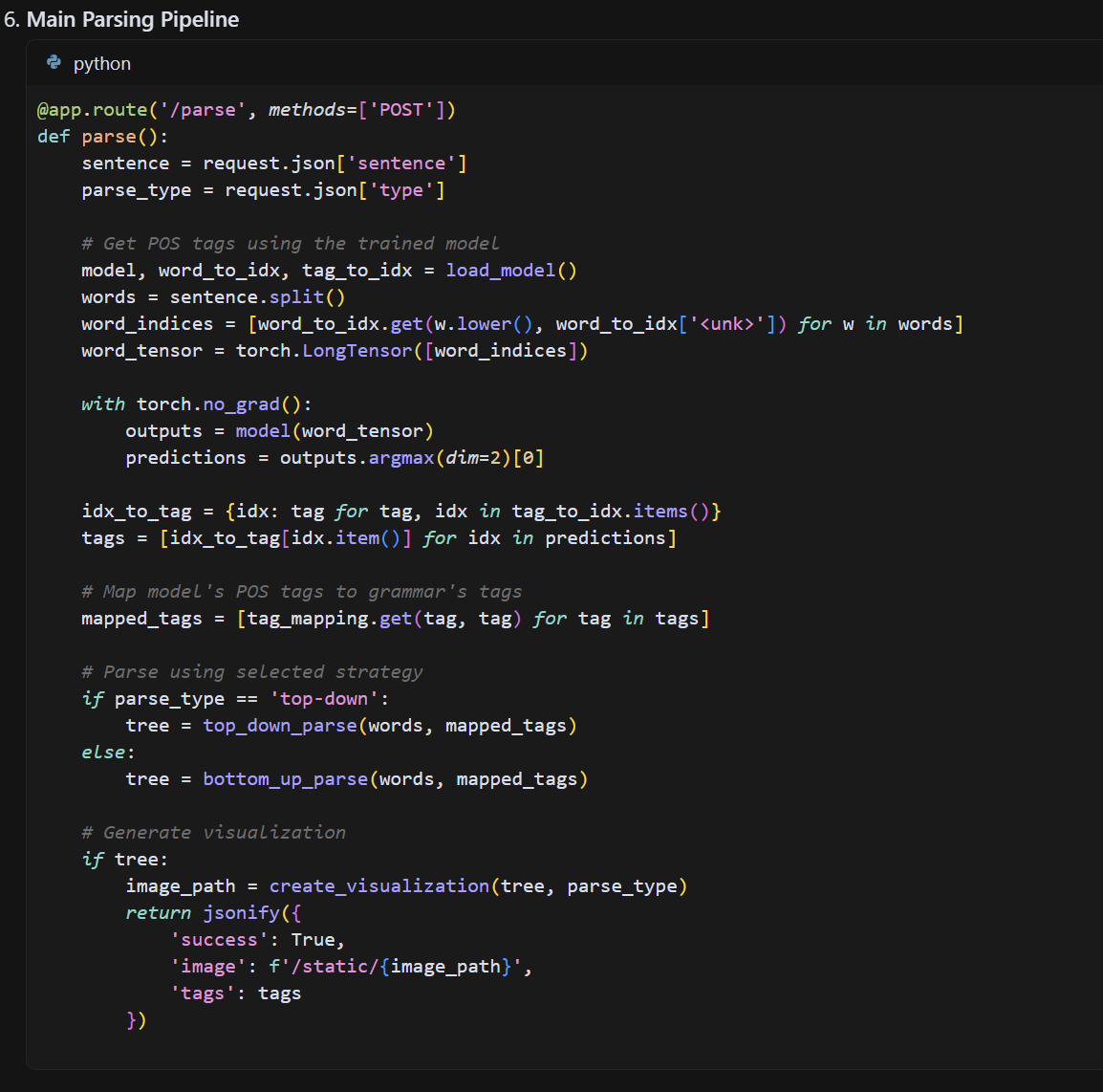






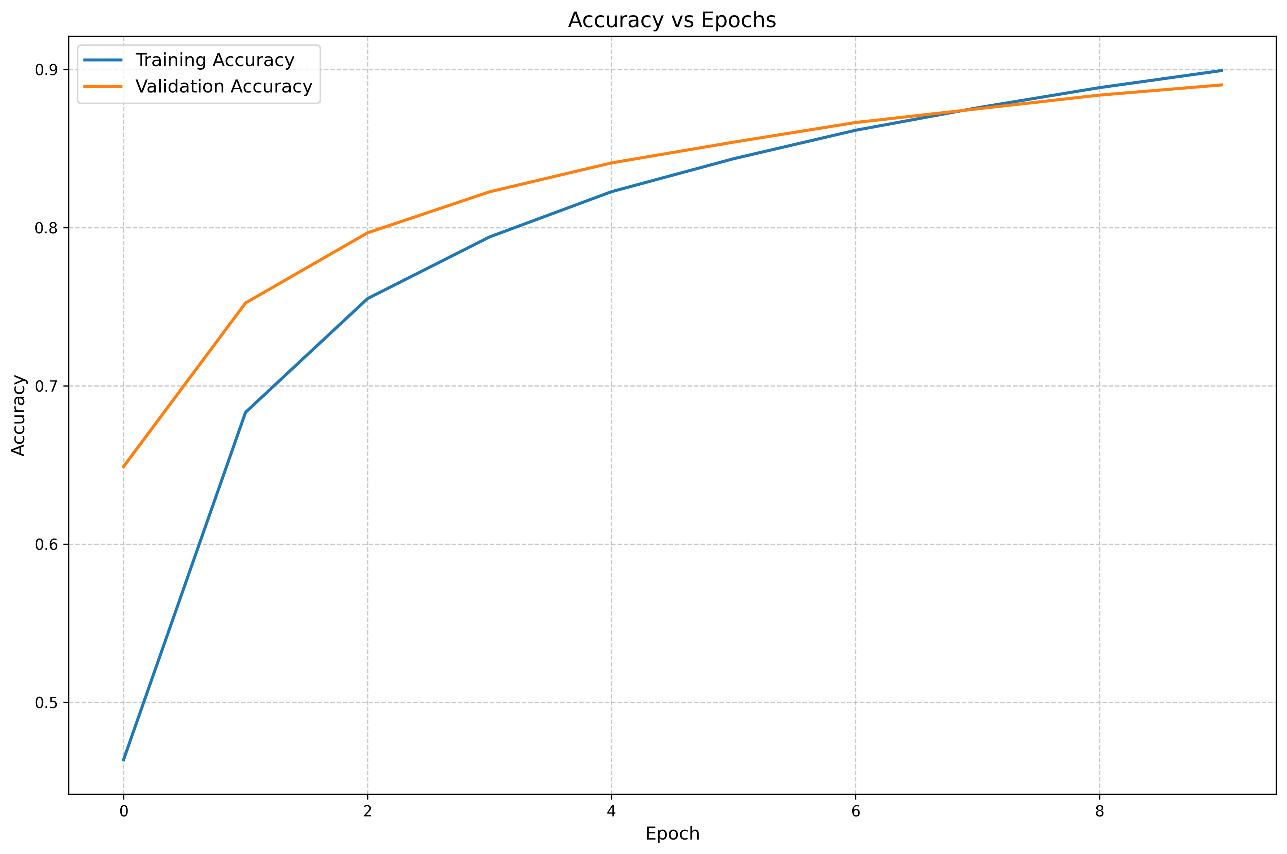


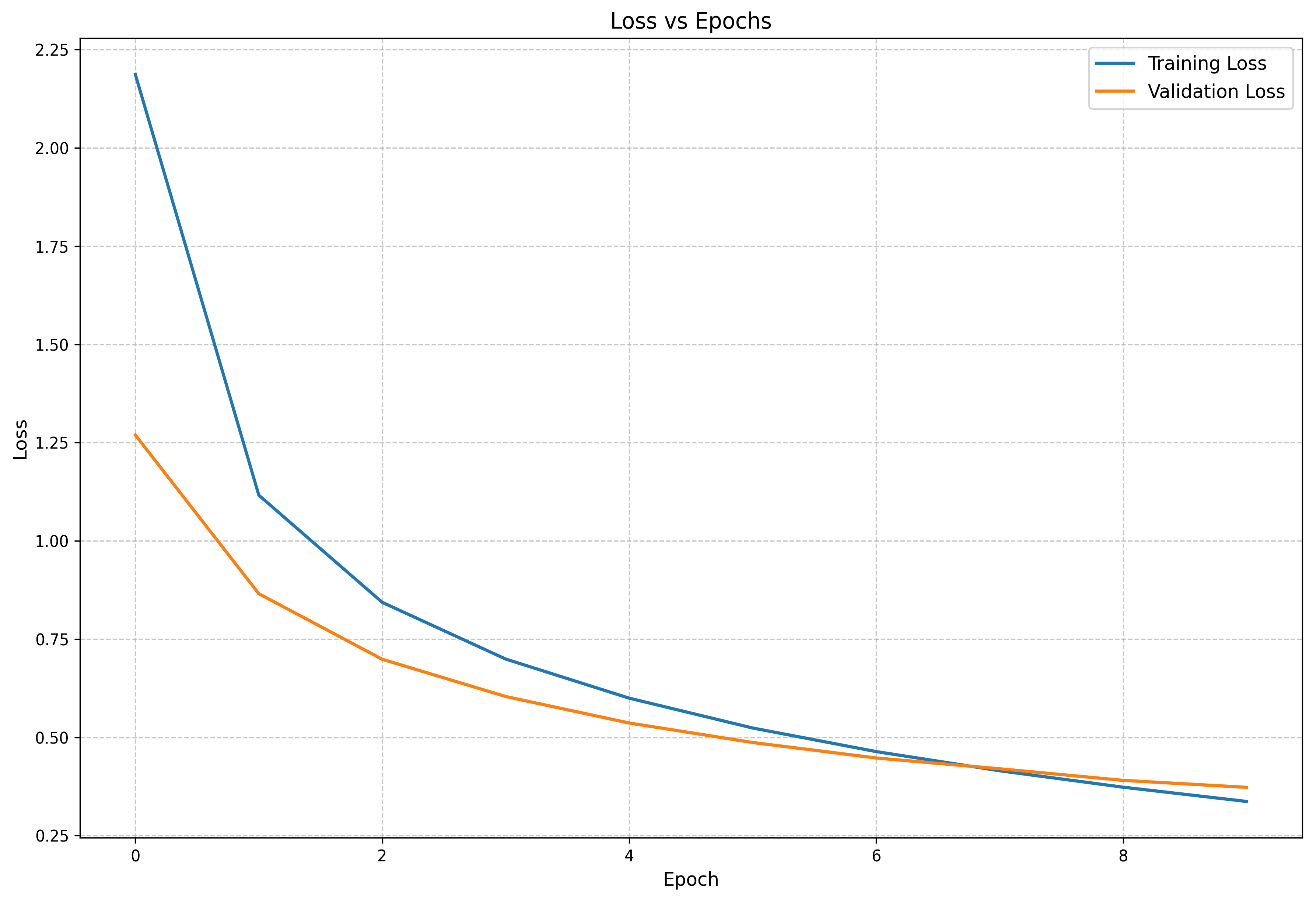




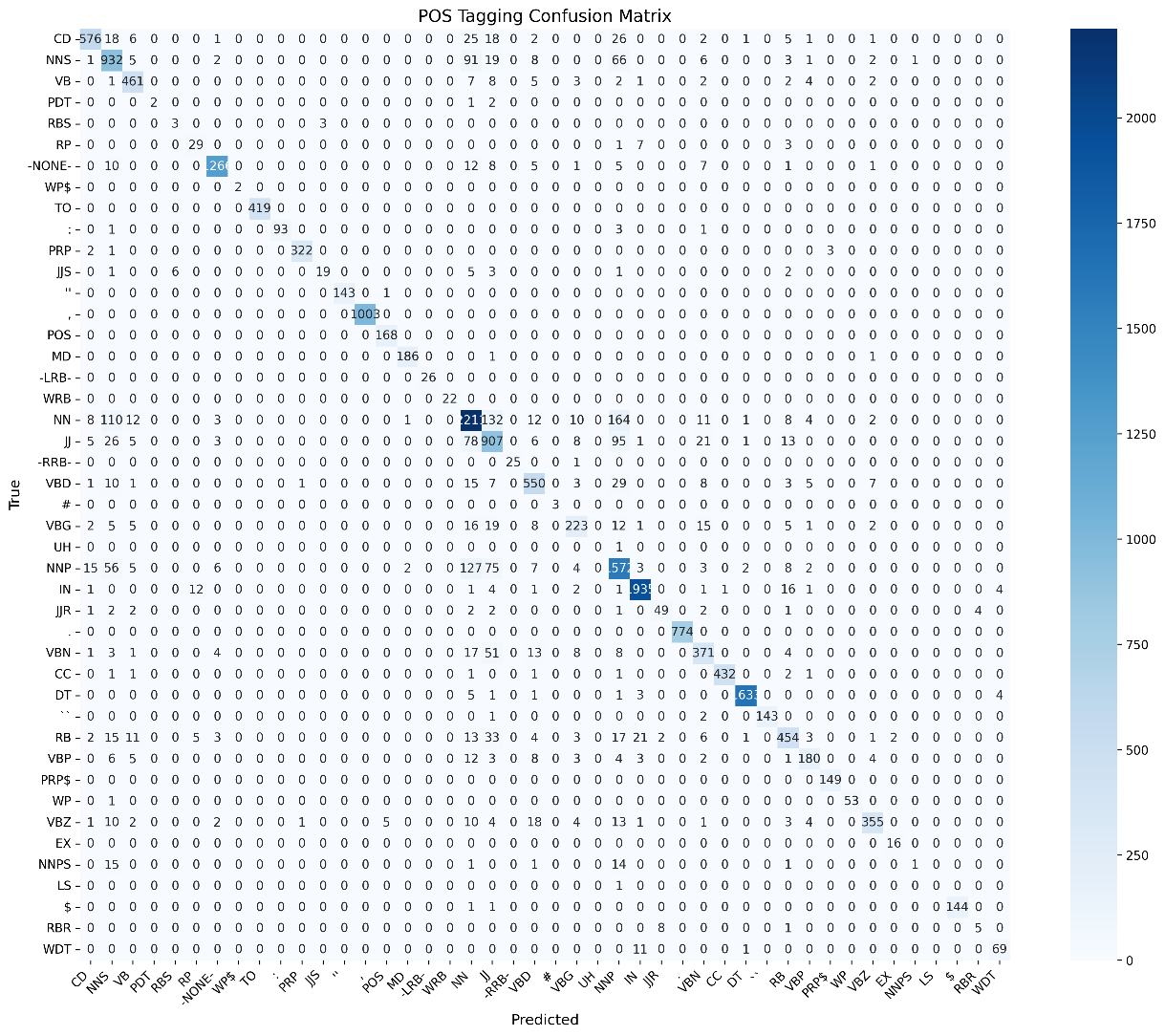
**6.Results**

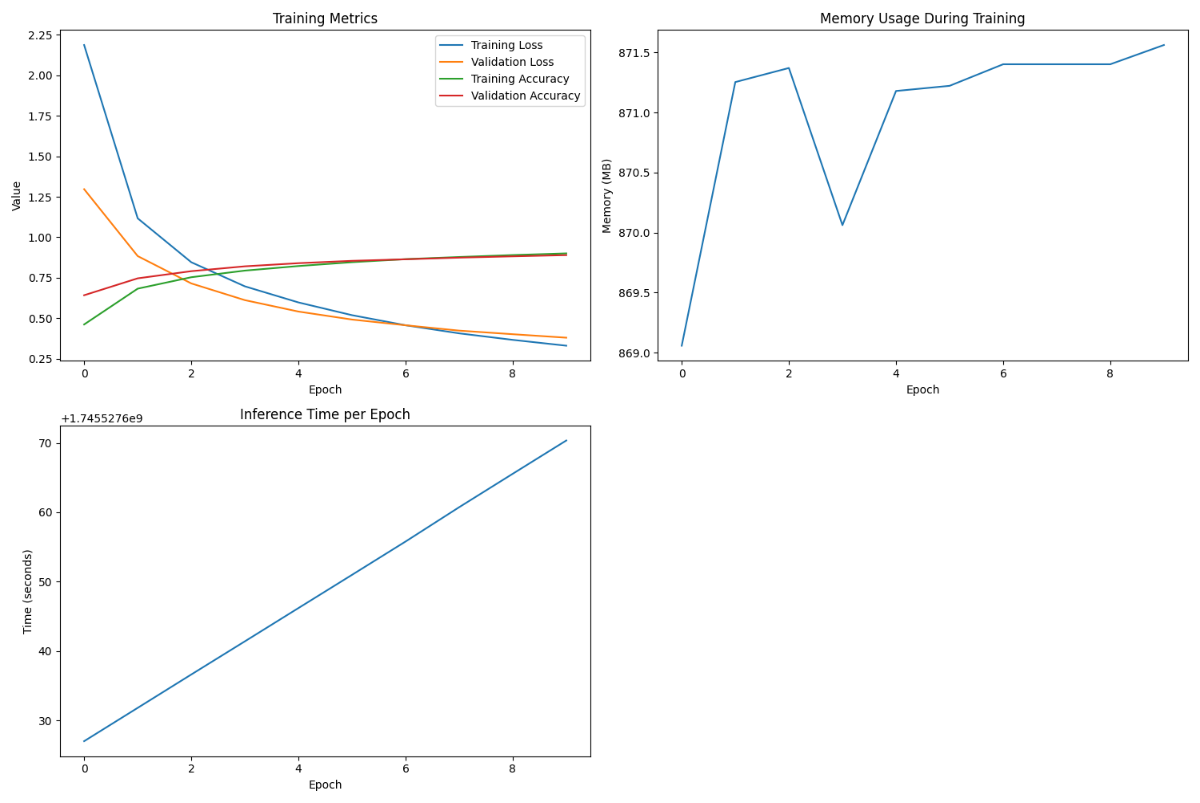
**Accuracy and Validation Graphs:**

****

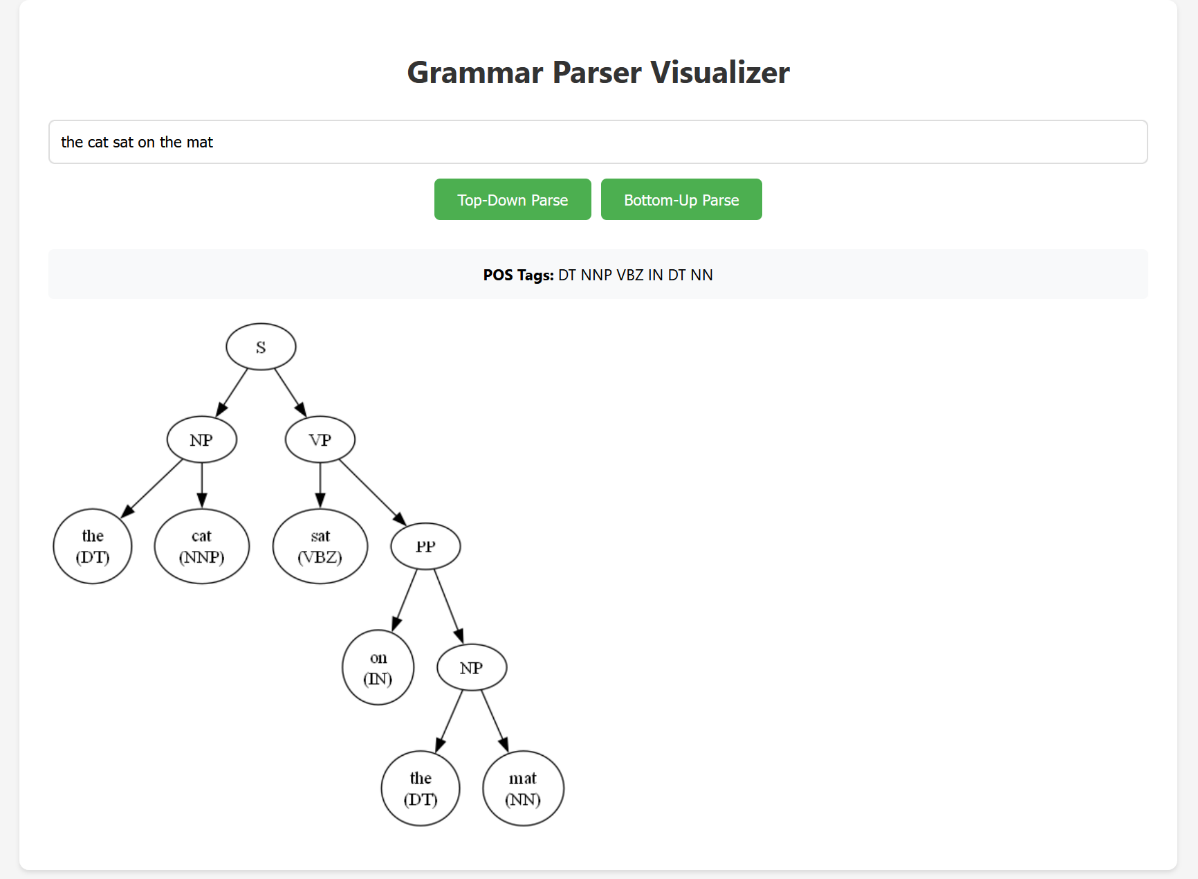
****

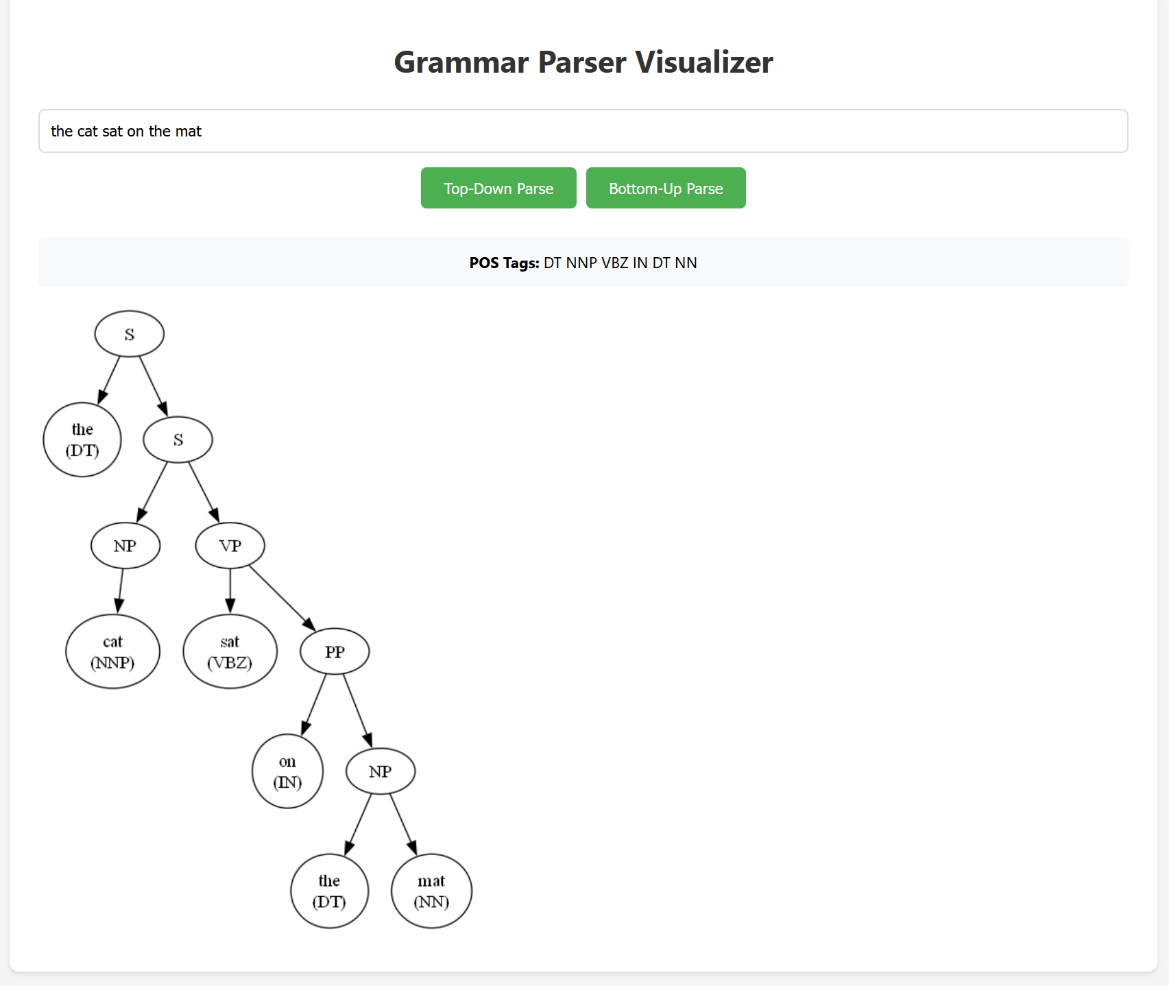
**Confusion Matrix:**

****

****

**Output:**

****

****

**8. Conclusion**

This project successfully demonstrates a hybrid approach to natural language parsing by combining deep learning-based POS tagging with rule-based Context-Free Grammar (CFG) parsing. By leveraging a bidirectional LSTM model trained on the Penn Treebank dataset, we were able to effectively predict part-of-speech tags for each word in a sentence with strong accuracy.

Once tagged, the CFG parser uses syntactic rules to build meaningful parse trees that reflect the grammatical structure of the sentence. This dual-layer system provides both flexibility from machine learning and explainability from rule-based logic—making it a robust solution for syntactic analysis in NLP.

The integration of tools like PyTorch, NLTK, Flask, and Graphviz creates a complete pipeline that goes from raw user input to a visual representation of the parse tree. The system is designed to be extendable, interpretable, and interactive, making it ideal for educational tools, research applications, or foundational NLP platforms.

**Future Enhancements**

* **Model Improvements**: Replacing BiLSTM with transformer-based models like BERT for improved contextual tagging.
* **Grammar Expansion**: Enhancing CFG rules to support questions, compound and complex sentences.
* **UI Features**: Introducing interactive tree visualization with node-level explanations and error highlighting.
* **Real-time Feedback**: Adding live corrections or parse suggestions based on incorrect grammar or unparsed sentences.