Final Report

1. Dependency Parsing

Dependency parsing is the task of analyzing the grammatical structure of a sentence to establish the relationships between words. In a dependency parse, words are connected by directed links that represent grammatical relationships, such as subject-verb or modifier-modified relationships. However, dependency parsing faces several challenges:

- Ambiguity: Natural language is inherently ambiguous, and sentences can have multiple valid interpretations. Dependency parsers need to disambiguate between different syntactic structures to find the most appropriate parse for a given sentence.

- Non-local Dependencies: Dependencies in a sentence can span long distances, requiring the parser to consider information from across the entire sentence. Resolving non-local dependencies accurately is crucial for producing a correct parse.

- Crossing Dependencies: In some cases, dependencies between words may cross each other, leading to complex dependency structures. Handling crossing dependencies poses a challenge for dependency parsers, as they need to ensure that the resulting parse adheres to linguistic constraints.

- Parsing Errors Propagation: Dependency parsing is often an iterative process, where errors made earlier in the parsing process can propagate and affect subsequent decisions. This can lead to cascading errors, where a single mistake early in the parse can result in a completely incorrect analysis.

- Language-specific Challenges: Different languages exhibit varying word order, morphological complexity, and syntactic structures, which can pose challenges for dependency parsers. Parsers may need to be adapted or trained specifically for different languages to achieve optimal performance.

- Out-of-Vocabulary Words: Dependency parsers rely on pre-trained models or lexicons to recognize words and their properties. Out-of-vocabulary words, i.e., words not seen during training, can pose challenges for parsers, as they may struggle to accurately analyze the syntactic role of such words.

Addressing these challenges often involves the use of sophisticated algorithms, machine-learning techniques, and linguistic knowledge. Dependency parsing remains an active area of research, with ongoing efforts to improve accuracy, efficiency, and robustness across different languages and domains.

2. Transition-Based Parsing

Transition-based parsing is a common approach for dependency parsing that operates by iteratively applying transition actions to build a dependency tree for a given sentence. It's based on the concept of a transition system, where the parser moves through different states of parsing the input sentence until a complete parse tree is constructed.

Here's an overview of how transition-based parsing works:

- Initial State: The parsing process starts with an initial state where the parser is provided with the input sentence to be parsed. At the beginning, the parser has an empty dependency tree.

- Transition Actions: In transition-based parsing, the parser applies transition actions to move from one parsing state to another. Each transition action corresponds to a specific operation that modifies the parsing state. Common transition actions include:

    - Shift: Move the next word in the input sentence onto the parsing stack.

    - Reduce: Combine words on the parsing stack into a syntactic unit (e.g., a phrase) and add a dependency link between them.

    - Left-Arc: Add a dependency link between the top two words on the parsing stack, with the first word being the head and the second word being the dependent.

    - Right-Arc: Add a dependency link between the top two words on the parsing stack, with the second word being the head and the first word being the dependent.

- Transition Sequence: The parser applies a sequence of transition actions to gradually build the dependency tree. The sequence of actions is determined by a parsing algorithm, which may be rule-based or learned from annotated data using machine learning techniques such as neural networks.

- Termination: The parsing process continues until a termination condition is met. This condition could be reaching a specific parsing state (e.g., an empty input sentence and a complete dependency tree) or until all possible actions are exhausted.

However, transition-based parsing also has limitations. It may struggle with parsing sentences containing complex structures or handling cases of ambiguity,

3. Selection of Baseline

We have chosen a baseline model based on a research paper titled "A Fast and Accurate Dependency Parser using Neural Networks." A transition-based parser is used for dependency parsing in the paper. Transition-based parsing is selected for dependency parsing due to several reasons:

- Efficiency: Transition-based parsers often have a linear time complexity with respect to the length of the input sentence. This makes them efficient and suitable for parsing long sentences or large corpora, especially in real-time applications where parsing speed is crucial.
- Incremental Parsing: Transition-based parsers build the dependency tree incrementally, making decisions at each step based on the current parsing state. This incremental nature allows for fast parsing, as the parser does not need to consider all possible dependencies simultaneously.
- Simplicity: Transition-based parsing algorithms are relatively simple to implement compared to other parsing techniques, such as graph-based parsing. They typically involve a small set of transition actions that modify the parsing state, making them easier to understand and implement.
- Learning Ability: Transition-based parsers can be easily integrated with machine learning techniques, such as supervised learning or reinforcement learning, to learn parsing decisions from annotated data. This allows for the development of more accurate parsers by leveraging large annotated datasets.
- Adaptability: Transition-based parsers can be adapted and customized to different languages and domains by adjusting the feature set, training data, or transition system. This adaptability makes them suitable for parsing a wide range of languages and text types.
- State-of-the-Art Performance: Transition-based parsers have achieved state-of-the-art performance on many benchmark datasets and parsing tasks. They are competitive with other parsing techniques in terms of accuracy while offering advantages in terms of efficiency and simplicity.

Overall, the efficiency, simplicity, learning ability, adaptability, and performance of transition-based parsing make it a popular choice for dependency parsing in both research and practical applications.

4. Dataset

The dataset we are utilizing comprises annotated sentences, where each sentence is segmented into words. For each word in the sentence, there exists corresponding information, including a part-of-speech (POS) tag, the head of the word in the dependency tree which is indicated by the index of the word in the sentence, and the label indicating the dependency relationship. Additionally, each word is assigned a numerical position(index) representing its sequential placement within the sentence.

In our dataset, there are two columns dedicated to POS tags: one column contains standard POS tags, while the other column contains fine-grained POS tags. The fine-grained POS tags offer a more detailed classification of word categories, encompassing a total of 45 unique tags within the training set. To accommodate potential unseen or undefined tags, we have incorporated an "unknown" tag and a "null" tag into our implementation.

Fine-grained POS tags provide more detailed and specific information about the syntactic role and category of words in a sentence compared to standard POS tags. While standard POS tags typically categorize words into broader classes such as nouns, verbs, adjectives, and adverbs, fine-grained POS tags offer more granularity by distinguishing between different subtypes within each category.

Here are some examples:

- NN: Singular noun
- NNS: Plural noun
- NNP: Proper noun, singular
- NNPS: Proper noun, plural
- PRP: Personal pronoun
- PRP$: Possessive pronoun

A noteworthy observation within our dataset is that over 99% of the dependency trees are projective, suggesting a relatively structured and well-formed syntactic representation. Furthermore, the dataset has been divided into three subsets: training, development, and testing. The training set consists of 39,832 sentences, while the development and test sets contain 2,416 and 1,700 sentences, respectively. These partitions facilitate the evaluation and refinement of our model across distinct stages of development.

Dependency relations define the syntactic relationships between words in a sentence, providing valuable information for understanding sentence structure and meaning. They are essential for building accurate dependency parse trees and analyzing linguistic phenomena.

After examining the distribution of the dependency relations, an analysis was conducted:

- Most Common Relations: The most common dependency relation is 'punct', indicating punctuation marks. This suggests that punctuation plays a significant role in structuring sentences. Other common relations include 'case', 'nmod' (nominal modifier), 'compound', and 'det' (determiner), which are fundamental for specifying grammatical relationships and modifying nouns.

- Subject-Verb Relations: Relations such as 'nsubj' (nominal subject) and 'nsubjpass' (passive nominal subject) indicate the subjects of sentences, while 'aux' and 'auxpass' denote auxiliary verbs. These relations are crucial for determining the syntactic roles of subjects and verbs in sentences.

- Modifiers and Complements: Relations like 'amod' (adjectival modifier), 'advmod' (adverbial modifier), 'nummod' (numeric modifier), and 'appos' (appositional modifier) are common modifiers that provide additional information about nouns, verbs, or other modifiers. Complement relations such as 'dobj' (direct object), 'ccomp' (clausal complement), and 'xcomp' (open clausal complement) represent syntactic units that complete the meaning of verbs and other predicates.

- Subordinate Clauses: Relations like 'advcl' (adverbial clause modifier), 'acl' (adjectival clause modifier), and 'acl:relcl' (relative clause modifier) indicate the presence of subordinate clauses, which add complexity to sentence structures.

- Rare Relations: Some relations, such as 'expl' (expletive), 'iobj' (indirect object), and 'csubj' (clausal subject), occur less frequently in the dataset. These relations represent specific syntactic constructions that are less common in typical sentences.

5. Summary of Implementation

The implementation consists primarily of three main files: the model file, the main training file, and the utility functions file.

Main training file:

1. Imports necessary libraries and modules: ParserModel, Utils, and Sentence_Parser classes from custom modules (model.py and utils.py). numpy, torch, and other required modules for data processing and training.
2. Defines a custom dataset class MyDataset inheriting from torch.utils.data.Dataset. This class is used to organize input embeddings and labels for training.
3. Defines a training function train that takes the model, training data loader, number of epochs, development data, and utility object as input. Inside this function:
    - It iterates over the specified number of epochs and trains the model on the training dataset using mini-batch gradient descent.
    - Prints the training loss after each epoch.
    - Evaluates the model's performance on the development dataset by parsing sentences and calculating the Unlabeled Attachment Score (UAS).

4. Defines a testing function test that evaluates the trained model on the test dataset. Inside this function:
    - It parses sentences from the test dataset and calculates both UAS and Labeled Attachment Score (LAS).
5. Defines the main execution block where the following steps are performed:
    - Initializes a utility object (obj) for preprocessing data.
    - Preprocesses the data, obtaining training, testing, and development datasets along with word embeddings (Ew).
    - Initializes random tag and label embeddings (Et and El).
    - Initializes the parser model using ParserModel class.
    - Sets hyperparameters such as number of epochs, learning rate, batch size, and regularization term.
    - Defines loss function (CrossEntropyLoss) and optimizer (Adam).
    - Creates a data loader for the training dataset.
    - Calls the train function to train the model.
    - Calls the test function to evaluate the trained model on the test dataset.
    - Saves the trained model's state dictionary to a file (model.pt).
    - Saves the utility object to a pickle file (file.pkl).

Utility function file: There are two important classes present in this file.

1. Utils Class:
    - Initializes with paths to data files and embedding files.
        - Provides methods for:
        - Reading data from CoNLL format files (get_data).
        - Vectorizing dataset by converting words, tags, and labels to unique IDs (vectorize).
        - Generating transition actions based on stack, buffer, and sentence information (get_transition).
        - Extracting features for each transition action (get_features).
        - Creating instances for training, where each instance consists of input features and corresponding output labels (create_instances).

    - Performs preprocessing of training, test, and development datasets:
        - Reads data from files.
        - Converts words, tags, and labels to unique IDs.
        - Vectorizes datasets.
        - Retrieves word embeddings from the embedding file and initializes word embeddings matrix.

2. Sentence_Parser Class:
    - Initializes with a sentence object containing words, tags, heads, and labels.
    - Provides methods for:

    - Parsing a sentence using a given utility object and model (parse).
    - Extracting parsed dependencies and calculating UAS (Unlabeled Attachment Score) or LAS (Labeled Attachment Score) (get_pred).

Model file: It defines a neural network model for dependency parsing called ParserModel

- Inherits from nn.Module and initializes with various parameters such as embedding sizes, input sizes, hidden size, number of classes, and dropout probability.
- Sets up embedding layers for words, tags, and labels using nn.Embedding.
- Defines input layers for words, tags, and labels using nn.Linear.
- Defines the output layer using nn.Linear.
- Initializes the parameters of the input and output layers using Xavier initialization (nn.init.xavier_uniform_).
- Defines a method get_embeddings to obtain embeddings for words, tags, and labels.
- Implements the forward method, which:
    - Retrieves embeddings for words, tags, and labels.
    - Passes them through the input layers and applies activation function ReLU.
    - Applies dropout regularization.
    - Passes the result through the output layer to get the final output.

6. Results

In the context of dependency parsing, UAS (Unlabeled Attachment Score) and LAS (Labeled Attachment Score) are commonly used evaluation metrics to assess the accuracy of parsing algorithms. Here's an explanation of the results you provided:

1. UAS (Unlabeled Attachment Score): 90.13%
    - UAS measures the percentage of words in the input sentences for which the dependency arcs are correctly predicted, regardless of the specific dependency label attached to each arc.
    - An UAS of 90.13% indicates that, on average, 90.13% of the dependency arcs in the parsed sentences are correctly predicted.

2. LAS (Labeled Attachment Score): 88.03%
    - LAS measures the percentage of words in the input sentences for which both the dependency arcs and their corresponding labels are correctly predicted.

- An LAS of 88.03% indicates that, on average, 88.03% of the dependency arcs and their labels in the parsed sentences are correctly predicted.

Qualitative Analysis:

1. 'the complex financing plan in the s&l bailout law includes raising $ 30 billion from debt issued by the newly created rtc .'

```
plan  ->  financing  :  compound
plan  ->  complex  :  amod
plan  ->  the  :  det
law  ->  bailout  :  compound
law  ->  s&l  :  compound
law  ->  the  :  det
law  ->  in  :  case
plan  ->  law  :  nmod
includes  ->  plan  :  nsubj
billion  ->  30  :  compound
$  ->  billion  :  nummod
debt  ->  from  :  case
created  ->  newly  :  advmod
rtc  ->  created  :  amod
rtc  ->  the  :  det
rtc  ->  by  :  case
issued  ->  rtc  :  nmod
debt  ->  issued  :  acl
$  ->  debt  :  nmod
raising  ->  $  :  dobj
includes  ->  raising  :  xcomp
includes  ->  .  :  punct
.  ->  includes  :  root
```

2. 'this financing system was created in the new law in order to keep the bailout spending from swelling the budget deficit .'

```
system  ->  financing  :  compound
system  ->  this  :  det
created  ->  was  :  auxpass
created  ->  system  :  nsubjpass
law  ->  new  :  amod
law  ->  the  :  det
law  ->  in  :  case
created  ->  law  :  nmod
in  ->  order  :  mwe
keep  ->  to  :  mark
keep  ->  in  :  mark
bailout  ->  the  :  det
spending  ->  bailout  :  nsubj
keep  ->  spending  :  dobj
swelling  ->  from  :  mark
deficit  ->  budget  :  compound
deficit  ->  the  :  det
swelling  ->  deficit  :  dobj
keep  ->  swelling  :  advcl
created  ->  keep  :  advcl
created  ->  .  :  punct
.  ->  created  :  root
```