Practical Assignment 2 – Dependency Parsing

Natural Language Understanding
Master in Artificial Intelligence

Alberto Somoza
Emilio Somoza
Gianluca Lascaro

December 7, 2025

1 Description

This project implements a transition-based dependency parser following the arc-eager paradigm, coupled with a compact neural classifier. Data processing involves a `ConlluReader` that ensures compatibility with the arc-eager transition system by removing non-projective trees. Gold-standard parsing configurations and supervision transitions are obtained using an arc-eager oracle, extracting features from word forms and UPOS tags. The neural architecture, implemented via the `ParserMLP` class, uses 100- and 25-dimensional embeddings for word forms and POS tags, respectively, followed by two hidden layers. The model produces two softmax outputs: one over the four arc-eager transitions (SHIFT, LEFT-ARC, RIGHT-ARC, REDUCE) and one over the 44 dependency relations. Training is performed using the Adam optimizer with a joint loss, applying the dependency relation loss only to arc-creating transitions (LEFT-ARC and RIGHT-ARC). Model selection occurs on the development set, and the final evaluation on the test set involves the iterative application of the parser under arc-eager constraints, followed by a post-processing module to repair malformed structures.

2 User Manual

Environment uses Python 3.9.x with TensorFlow/Keras and NumPy. To execute run:

```
python main.py
```

The script prints dataset sizes, model summary, training/validation accuracies, and dev metrics, then saves model weights and dictionaries under `models/` and writes predictions to `data/output_clean.conllu`.

3 Model and Implementation

3.1 Arc-eager transition system

Our implementation (`src/algorithm.py`) mirrors standard arc-eager behaviour but enforces correctness via explicit preconditions. A configuration is a triple (σ, β, A): the stack `S`, buffer `B`, and set of arcs `A` with tuples `(head_id, deprel, dependent_id)`. Transitions are:

- **LEFT-ARC**: applicable when stack and buffer are non-empty, the stack top is not ROOT, and it has no head yet. It is correct when the buffer head is the gold head of the stack top and no buffer token has the stack top as gold head (to avoid removing parents with pending children). It adds $(b.id, dep, s.id)$, writes `s.head=b.id` and `s.dep=dep`, then pops the stack.
- **RIGHT-ARC**: applicable when stack and buffer are non-empty and the buffer head has no head yet. It is correct when the stack top is the gold head of the buffer head. It adds $(s.id, dep, b.id)$, writes `b.head=s.id` and `b.dep=dep`, then moves the buffer head to the stack.

- **REDUCE**: applicable when the stack top already has a head. It is correct if no buffer token has the stack top as gold head; it simply pops the stack.
- **SHIFT**: the fallback when no other valid/correct transition applies; it moves the buffer head onto the stack.

The oracle prioritises valid and correct actions in the order LEFT-ARC, RIGHT-ARC, REDUCE, otherwise SHIFT, and asserts that the arcs produced during simulation match the gold arcs. This sequence of state–transition pairs constitutes the training data.

3.2 Data and preprocessing

We use cleaned English ParTUT splits in `data/`: `en_partut-ud-train_clean.conllu` (45,285 lines), `en_partut-ud-dev_clean.conllu` (2,879), and `en_partut-ud-test_clean.conllu` (3,562). The reader (`src/conllu/conllu_reader.py`) discards multi-word tokens (ID ranges, e.g., 2–3) and empty nodes (IDs with dots), and injects a dummy ROOT at position 0. We remove non-projective sentences for train/dev via a crossing-arc check; test is parsed as-is. Exact sentence counts are printed by the script at runtime; the file sizes correspond roughly to $\sim$3k train, $\sim$200 dev, and $\sim$240 test sentences.

3.3 Vocabulary construction and sample encoding

To interface the arc-eager oracle with the neural classifier, we introduce two utility modules, `src/vocab.py` and `src/preprocessor.py`, that build the necessary vocabularies and convert oracle configurations into fixed-size feature vectors.

The `build_form_upos_deprel_vocabs` function in `src/vocab.py` scans the training trees and constructs three vocabularies: one for word forms (FORM), one for universal POS tags (UPOS) and one for dependency relations (DEPREL). Word forms are counted with a frequency threshold (`min_freq`), and two special symbols are reserved: <PAD> for padding and <UNK> for out-of-vocabulary items. UPOS and DEPREL vocabularies also include a <PAD> symbol at index zero. The function returns both forward and reverse mappings (`form2id/id2form`, `upos2id/id2upos`, `deprel2id/id2deprel`) to support ID-based encoding and human-readable inspection.

In addition, `build_action_only_vocab` defines a compact action inventory over the four arc-eager transitions (`SHIFT`, `LEFT-ARC`, `RIGHT-ARC`, `REDUCE`). This function returns two dictionaries (`action2id`, `id2action`) that are used to map oracle transitions to integer labels for the action prediction head of the network.

The module `src/preprocessor.py` provides the `samples_to_arrays` function, which converts a list of oracle-generated `Sample` objects into NumPy arrays suitable for Keras. For each configuration, we first call `state_to_feats` to obtain a fixed window of features: word forms and UPOS tags from two stack positions and two buffer positions. These symbols are mapped to integer IDs using the vocabularies, with <PAD> and <UNK> handling missing or out-of-range values. The function then produces four arrays: `X_words` and `X_pos` as input matrices, and `y_action` and `y_deprel` as label vectors. For non-arc transitions (`SHIFT` and `REDUCE`), the dependency-label target is set to `-1`, which is later used as a mask when computing the loss for the deprel prediction head.

3.4 Neural architecture

The classifier (`src/model.py`) is a two-head MLP fed by word and UPOS features from the parser state, with $n = m = 2$ elements from stack and buffer. `Sample.state_to_feats` yields four word forms and four UPOS tags (padded with <PAD>, forms include <UNK>); these are mapped to IDs via vocabularies and embedded with `mask_zero=True`. Word embeddings have dimension 100, UPOS embeddings 25.

We concatenate flattened embeddings and pass them through two ReLU-activated dense layers (64 units each). The network has two softmax outputs: action over 4 classes (SHIFT, LEFT-ARC, RIGHT-ARC, REDUCE) and deprel over the label inventory learned from training trees. Regularisation is implicit via padding masks; we do not add dropout or L2. Because deprels are undefined for SHIFT/REDUCE, the training masks the deprel loss on those samples via sample weights.

Table 1: Summary of the MLP architecture.

Component	Specification
Inputs	Word features, POS features
Word embeddings	100 dimensions
POS embeddings	25 dimensions
Hidden layers	2 $\times$ Dense(64, ReLU)
Action output	4-way softmax (SHIFT, LA, RA, REDUCE)
Deprel output	Softmax over all dependency labels

3.5 Training setup

Training is configured in `main.py` and compiled in `model.py`. We optimise with Adam (default learning rate), batch size 64, for 10 epochs. Losses are sparse categorical cross-entropy for both outputs; metrics track accuracy for actions and deprels. Early stopping is not used. The script reports final train and validation accuracies and then computes detailed dev metrics by argmaxing the two heads and combining them into transition, deprel, and joint accuracies.

4 Results

4.1 Evaluation

After training, the model parses the test set by iteratively extracting features from active states, predicting actions and deprels, ordering actions by probability, and applying the first valid transition. The predictions are written back into token fields and saved to `data/output.conllu`; then a post-processor fixes multi-root scenarios and assigns heads to any remaining headless tokens, producing `data/output_clean.conllu`. LAS/UAS can be computed against the cleaned gold test split using the official `conll18_ud_eval.py`.

Table 2: Evaluation metrics of the dependency parser (values in %).

Metric	Precision	Recall	F1 Score	AligndAcc
...	...	...	...	...
UAS	75.56	75.56	75.56	75.56
LAS	67.05	67.05	67.05	67.05
...	...	...	...	...

4.2 Analysis

The evaluation on the cleaned UD English-ParTUT test set shows that the parser correctly preserves all token-level and tagging information: Tokens, Sentences, Words, UPOS, XPOS, UFeats, AllTags and Lemmas all reach 100% precision, recall and F1. This behaviour is expected, since the system does not attempt to predict segmentation or morphological analysis, but instead reuses the gold CoNLL-U

annotations and only predicts syntactic heads and dependency labels. The relevant syntactic metrics are therefore the attachment scores: the model achieves a UAS of 75.56% and a LAS of 67.05%, with CLAS, MLAS and BLEX in the range of 51–56%, although this last values are out of the scope for this project.

These scores confirm that the parser is able to recover the majority of head–dependent relations, while still making a substantial number of errors on more complex structures and fine-grained labels. The gap between UAS and LAS indicates that predicting the correct head is easier than assigning the exact UD dependency relation, and the lower CLAS/MLAS/BLEX values suggest additional difficulties when lexical and morphological agreement are taken into account.

First, we restrict training to projective trees and use an arc-eager transition system with a narrow local feature window (two positions from the stack and two from the buffer), which limits the amount of long-distance contextual information available for each decision.

Second, the model makes all predictions based on a single fixed representation of the current configuration, without any recurrent or self-attentive mechanism to encode sentence-level structure or transition history. As a result, the parser handles local attachments reasonably well but tends to struggle with prepositional phrases, coordination and other constructions that require broader context.

Overall, the results are consistent with what can be expected from a compact MLP-based transition parser: they demonstrate that the architecture is able to learn meaningful syntactic patterns from relatively simple features, while also highlighting clear avenues for improvement, such as richer feature sets or more expressive encoders.

5 Conclusion

We presented a compact transition-based dependency parser that combines a standard arc-eager algorithm with a two-head MLP classifier. The system focuses on simplicity and clarity: parsing configurations are generated by an explicit oracle, features are extracted from a small window over the stack and buffer, and vocabularies and samples are handled by lightweight preprocessing utilities. On top of this, the neural architecture uses modest word and POS embeddings and two hidden layers to jointly predict the next transition and, when applicable, the corresponding dependency label. This design keeps the overall model easy to implement, debug and interpret, while still enforcing strong algorithmic constraints that guarantee structurally well-formed outputs after post-processing.