Project Title: Biaffine scoring Neural Dependency Model with Valence (Neural DMV)

Pragya Paramita Pal (7071771)

The Original Idea:

In their 2004 paper "Corpus-Based Induction of Syntactic Structure: Models of Dependency and Constituency", Klein and Manning introduced the Dependency Model with Valence (DMV).

The main feature of this model is "valence" - whether a head has already taken arguments in a particular direction. This allows capturing important linguistic patterns like verbs taking exactly one subject.

1. **Start at ROOT**: Begin with a special ROOT node
2. **Generate Main Head**: ROOT generates a single dependent (usually the main verb)
3. **Recursive Expansion**: For each head word:
    - First generate dependents to the right until deciding to stop
        1. P_{STOP} decides whether stopping or choosing a dependent
        2. P_{CHOOSE} decides which dependent is chosen based on the current head POS
    - Then generate dependents to the left until deciding to stop (same as above)
    - Each newly generated dependent recursively follows the same process

DMV includes explicit probabilistic STOP decisions in its model. Special token STOP is to end generation in this direction otherwise, continue and generate another dependent.

P_{STOP}(STOP|h, dir, adj)

This is the probability of stopping the generation of dependents in a particular direction.

Parameters:

- **h**: The head word (typically represented by POS tag)
- **dir**: Direction (LEFT or RIGHT)
- **adj**: Adjacency status (TRUE/FALSE). Signifies whether it has already generated a dependent (Valence)
- **STOP**: The decision (STOP or ¬STOP)

Linguistic Interpretation:

- P_{STOP}(STOP|VERB, LEFT, TRUE): Probability a verb takes no subjects
- P_{STOP}(STOP|VERB, LEFT, FALSE): Probability a verb stops after taking one or more subjects
- P_{STOP} (STOP|NOUN, RIGHT, TRUE): Probability a noun takes no right modifiers

$P_{CHOOSE}(a|h, dir)$

This is the probability of choosing a specific dependent given a head.

Parameters:

- **a**: The dependent word/POS
- **h**: The head word/POS
- **dir**: Direction (LEFT or RIGHT)

Linguistic Interpretation:

- P_{CHOOSE}(DET|NOUN, LEFT): Probability of a noun taking a determiner as left dependent
- P_{CHOOSE}(NOUN|VERB, RIGHT): Probability of a verb taking a noun as right dependent

The DMV model learns through an unsupervised Expectation-Maximization (EM) algorithm that iteratively refines the model's parameters.

E-Step (Expectation)

- Use the **inside-outside algorithm** to compute expected counts:
    - For each sentence, calculate probabilities over all possible dependency trees
    - Compute expected frequencies of each attachment and stop decision

M-Step (Maximization)

- Update parameters using maximum likelihood estimates:
    - PSTOP(STOP|h,dir,adj) = (Expected count of stopping) / (Expected count of decision opportunities)
    - PCHOOSE(a|h,dir) = (Expected count of h→a attachments) / (Expected count of all h attachments in dir)

My Problem Statement:

I wanted to combine the above DMV with a neural model having Biaffine scoring. Using this, the probabilities explained above would be calculated by a neural model and it can incorporate richer contextual information with the embeddings. I hoped the richer representations would likely lead to significant improvements in parsing accuracy while maintaining the linguistically-motivated structure of DMV.

Datasets used: English Web Treebank and Hindi UD Treebank

The data processing pipeline:

I created a **Vocabulary class** with the purpose of mapping tokens to numerical indices for the neural model. It sets up dictionaries with special tokens and counts token frequencies during preprocessing. It then creates final vocabulary, filtering by frequency and size of the data.

Next a **ConllLoader class** was created to parse CoNLL-U formatted dependency treebanks. It parses dependency treebank and extracts word forms, POS tags, and head information.

SimpleDataModule class helps implement the data preprocessing:

- Text cleaning (removing punctuation, normalizing numbers)
- Limited sentences to 10 words or less (following Klein & Manning's approach)
- Built separate vocabularies for words and POS tags

ConllDataset class extends PyTorch's Dataset class and maps words and POS tags to indices, returning tensors for model training.

There are also two supporting functions:

A **custom_collate_fn** to handle variable-length sequences that uses pad_sequence to create uniform-length tensors.

And a **create_dataloaders function** that builds PyTorch DataLoaders for train, validation and test sets.

The Neural Model for the DMV:

It must have main 3 probabilities:

Attach/ P_{CHOOSE}: P(j|i,dir) - Probability of word j being a dependent of word i in direction dir

- Shape: [batch,seq_len,seq_len,2]

Decision/ P_{STOP}: P(stop|i,dir,val) - Probability of stopping/continuing for a head

- Shape: [batch,seq_len,2,2,2]

Root: P(i=root) - Probability of word i being the root

- Shape: [batch,seq_len]

Uses a **Biaffine Scoring**: Borrowed from supervised dependency parsing (Dozat & Manning, 2017). The arc scoring between head and dependent would be more expressive using this.

Neural Parameterization of DMV Probabilities:

POS embeddings and Valence embeddings jointly represent h_{joint}

$h_{joint}=[e_{pos}(x);e_{val}(v)]$

Head Projection is given by: $h\_head = MLP\_head(h\_joint)$ where MLP_head does as follows

$$MLP\_head(h\_joint) = W\_head \cdot h\_joint + b\_head$$

Similarly, Dependent Projection is given by: $h\_dep = MLP\_dep(h\_joint)$

Now, the arc scoring between h_head and h_dep is done with a Biaffine scoring function as:

$$s\_arc(i,j) = Biaffine(h\_head(\mathrm{i}), h\_dep(j))$$

So there is a single scoring function with directional masks. I used triangular masks to enforce directionality constraints:

$$s\_left(i,j) = s\_arc(i,j) \cdot 1 \text{ if } j < i + (-\infty) \cdot 1 \text{ if } j \geq i$$

And

$$s\_\mathrm{right}(i,j) = s\_arc(i,j) \cdot 1 \text{ if } j > i + (-\infty) \cdot 1 \text{ if } j \leq i$$

P_left and P_right are calculated by doing softmax of the above such that it has a probability distributed across all possible dependents for each head word.

Next P_{STOP} probabilities are implemented which depend on:

- Direction (left/right)
- Valence (0 or 1+ dependents already taken)
- Decision (stop/continue)

$$MLPdecision(h) = W2 \cdot ReLU(W1 \cdot h + b1) + b2$$

- Where h consists of:

  Flattened embedding (POS+valence combinations)

  [POS; valence] = 2*pos_emb_size dimensions

  To represent this for BOTH valence states (0 deps, 1+ deps):

  2 valence states × 2*pos_emb_size = 4*pos_emb_size

- Output is reshaped to [b,l,2,2,2]

  The 8-dimensional output is reshaped to represent:

  - b: batch size
  - l: sequence length (word positions)

- 2: direction (left/right)
- 2: valence (0 or 1+ dependents)
- 2: decision (stop/continue)

- The output models $P\text{stop}(stop \mid dir, val, dec)$ after applying softmax on the decision scores.

$$Pstop(dir, val, dec) = softmax(MLPdecision(hjoint))$$

Similarly, a dedicated MLP for root selection. This allows learning which POS tags are likely to be sentence roots

$$MLProot(r) = W2 \cdot ReLU(W1 \cdot r + b1) + b2$$

This gives scores for each POS tag being selected as root, which are converted to probabilities using softmax.

$$Proot(pos) = softmax(MLProot(eroot))$$

Inside algorithm and EM for learning

For the **inside algorithm** we will be using a chart **alpha** where first 3 dimensions refer to

tree side [A/B], completeness [C/I], direction [L/R] and then inside each is 4D tensor consisting of [batch, head_position, span_size, valence]

So the base case is drawn up like:

alpha[A][C][L][:, :, 0, :] = decision[:, :, LEFT, :, STOP] (Complete tree, looking leftward with span size 0 will take on the stopping score in the right direction (b,l,2,2,2→ for all batches, all word positions, left, all valences, only STOP))

alpha[B][C][L][:, :, -1, :] = decision[:, :, LEFT, :, STOP] (Complete tree, looking leftward with the largest span size will take on the stopping score in the left direction (from right edge, after tree is complete, max span can only give STOP probability in Leftward direction)

alpha[A][C][R][:, :, 0, :] = decision[:, :, RIGHT, :, STOP] (same logic as above edge cases)

alpha[B][C][R][:, :, -1, :] = decision[:, :, RIGHT, :, STOP]

Recursively computes scores for left attachments and right attachments. Then combines scores from smaller subproblems using max operations.

It computes logZ which represents the total probability mass of all possible dependency trees for the sentence. It contains the sum of probabilities for all possible parse trees starting from position 0 of

ROOT symbol, generating a complete right-branching tree, with initial valence 0 and covers the entire sentence.

These output gradients are going to be used as the expected counts of each rule in the model:

- grad_attach: Expected attachment counts between heads and dependents
- grad_decision: Expected counts of STOP/GO decisions
- grad_root: Expected counts of root attachments

These expected counts represent how often each rule would be used under the current model parameters.

The returned gradients are used in the M-step to update the neural network parameters toward configurations that maximize the likelihood of the observed data.

The loss function is the negative expected log-likelihood (since PyTorch minimizes loss). The loss is computed as a weighted sum of the model's scores (rules) by their expected counts. In training mode, this loss is backpropagated to update the model's parameters.

Hyperparameter Tuning for Neural DMV

I experimented with various batch sizes (32, 16, and 8) revealed that a batch size of 8 yielded the best performance. This may be because the chart computation is very memory expensive. So, a smaller batch size might help.

In the training by EM function, I used gradient clipping to prevent exploding gradients. clip_grad_norm_() rescales the gradients if their global norm exceeds a specified threshold 4.5 (out of 1, 2.5, 4.5, 5, 6.5).

The learning rate scheduler works by will be halving the learning rate when triggered, but waits 2 epochs before it acts. It acts on the avg_loss. If the loss stops improving, the lr is halved and helps the model not fall into false minima.

The learning rate is kept at (1e-5). It worked with the adaptive optimizer AdamW I used. A greater value of lr is causing larger errors, perhaps overshooting. So small changes is better for this.

Evaluation and Results

The Training Loss graph shows a sharper decrease in the first couple of epochs and then the NLL (negative log likelihood) decreases only slightly (sometimes fluctuating by around 0.02). 10 epochs are calculated and after that it seems to be plateauing out. The validation loss also drops initially and then only slightly fluctuates over the later epochs.

For evaluation, I calculated the **Unlabelled Attachment Score (UAS)** and **Directed Dependency Accuracy (DDA).** With that I could directly compare it to the paper's score and compare to see how my model does. I reused the Chu-Liu Edmonds algorithm from the assignment to get the dependency trees. It outputs the head word for each token in a sentence.

UAS = Number of correct head predictions/ Total number of non-root words

DDA is like UAS but excludes the root token)

DDA = Number of correct non-root head predictions/ Total number of non-root words

My model:

Test set UAS: 0.2969

Test set DDA: 0.3703

Paper's scores: English (WSJ10): 43.2% directed dependency accuracy, 63.7% undirected

I could not access the English (WSJ10). Thus, I did my evaluation on the English Web Treebank (EWT) and the Hindi UD treebank (UD_Hindi-HDTB)

For English the **test set DDA** are comparable for my model and the paper's but the **UAS (highest: 30.37%)** is still low—training does not converge very well, perhaps underfitting. However, UAS scores remain similar across various runs, meaning it learns a stable grammar, although incorrect.

For the Hindi UD Treebank it shows better convergence, reaches lower loss values, and demonstrates consistent improvement throughout training. It starts with a higher loss though, signifying the dataset may be more diverse or have ambiguous syntactic structures. However, it may contain more learnable patterns or the model architecture is potentially better suited to the linguistic properties of the Hindi Treebank. For this dataset, I failed to get the UAS scores.

Differences due to which comparison of my model with the paper's is not perfect:

The undirected dependency accuracy (63.7%) is essentially equivalent to UAS (Unlabeled Attachment Score), but calculated while ignoring the direction of the dependencies. UAS normally measures the percentage of words that are correctly attached to their head, regardless of the dependency label, but considering the direction.

Reasons for underfitting and areas to improve:

No linguistic priors were given to the model and perhaps that is why it underperforms. More nuanced rules['attach'] are required with a deeper LSTM.

The Left and Right attach scores being stacked together may be causing issues. Implementing them discretely.

The valence states just have NOCHILD and HASCHILD which may not account for valences of 1+ properly.