CSE 291: Advanced Statistical NLP

Project 3: Neural CRFs for Constituency Parsing

Due Friday June 11th by 11:59pm PT

1 Overview

In this assignment, you will be building a neural conditional random field (CRF) for the task of constituency parsing. You only need to implement the inside algorithm for the CRF as part of your model's forward pass. The rest of the system is provided in the starter code.

After introducing the task and modeling approach in Sections 2–3, we provide implementation instructions in Section 4. Finally, we describe deliverables in Section 5 and optional extensions to the project in Section 7.

2 Constituency Parsing

Constituency Parsing [1] [1, 2] is a type of syntactic parsing, which is the task of assigning a syntactic structure to a sentence. For example:

`I shot an elephant in my pajamas.`

can be parsed in two ways as Figure 1 shown.

[1] `https://web.stanford.edu/~jurafsky/slp3/13.pdf`

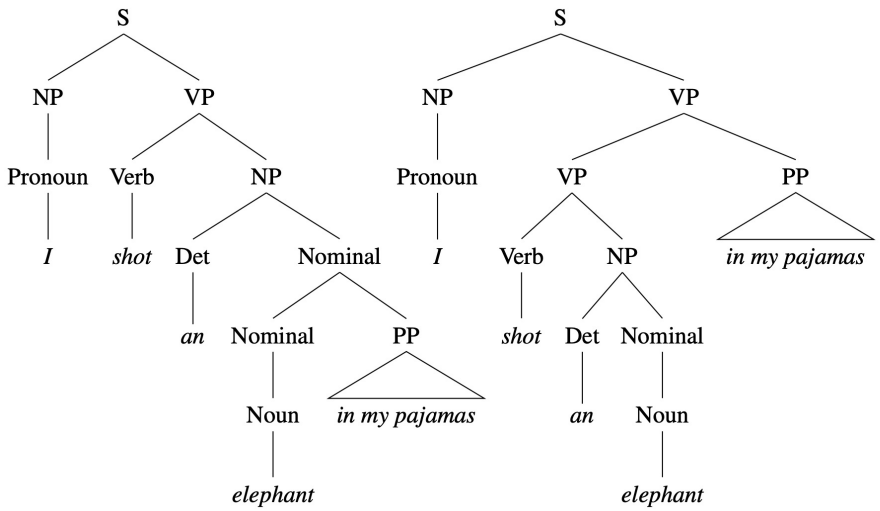

Figure 1: Two parse trees for an ambiguous sentence. The parse on the left corresponds to the humorous reading in which the elephant is in the pajamas, the parse on the right corresponds to the reading in which Captain Spaulding did the shooting in his pajamas.

For this assignment, you will be using data from the English WSJ Penn TreeBank (PTB) to learn a constituency parser. In the dataset itself, trees are represented as flattened strings with brackets and labels. For example,

```
(TOP (S (NP (NNP Ms.) (NNP Haag)) (VP (VBZ plays) (NP (NNP Elianti))) (. .)))
```

In the codebase, we provide a **`Dataloader()`** module to load the string trees and a **`draw_tree()`** method to visualize the trees (rendered as below).

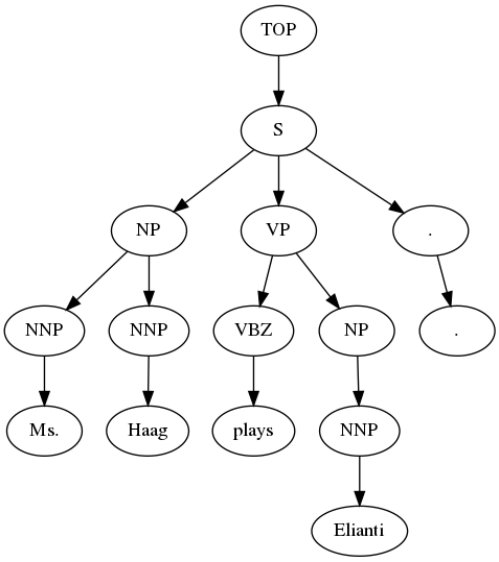

Figure 2: Constituency tree example.

3 LSTM-CRFs for Constituency Parsing

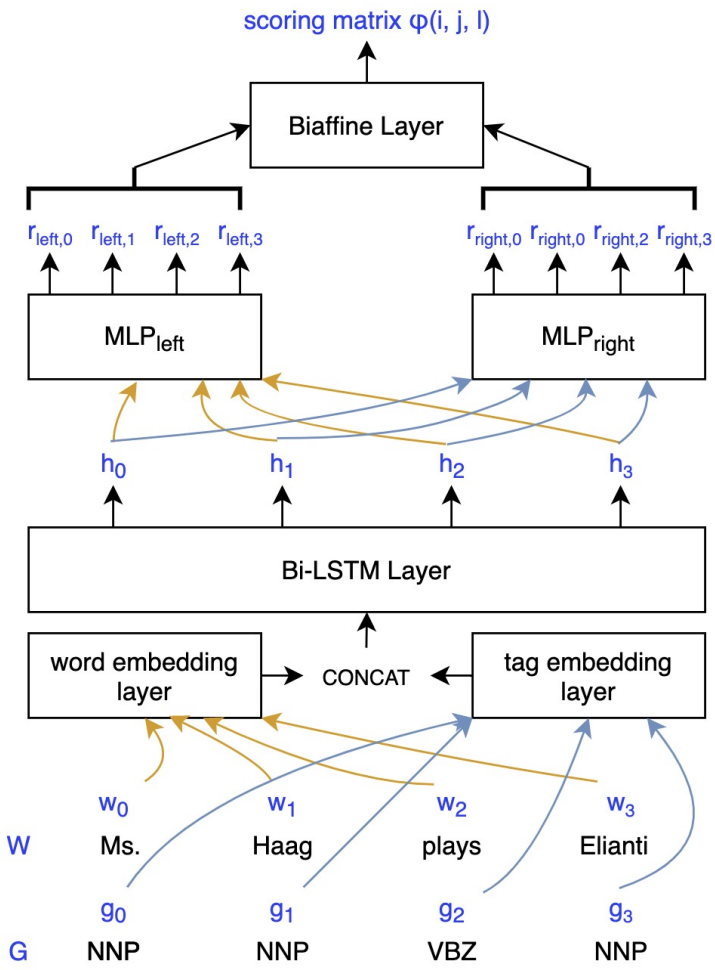

Figure 3: Neural parameterization overview.

An overview of the model that we ask you to build is shown in Figure 3. The model will predict a constituency parse tree given a sentence consisting of T words, $\boldsymbol{X} = \{\boldsymbol{W}, \boldsymbol{G}\}$, where $\boldsymbol{W} = w_0, w_1, \ldots, w_{T-1}$ are the words, and $\boldsymbol{G} = g_0, g_1, \ldots, g_{T-1}$ are the Part-of-Speech (POS) tags. A constituency parse tree is denoted as $\boldsymbol{Y}$, which is a set of triples, $(i, j, l) \in \boldsymbol{Y}$, that each denote a constituent spanning the words at positions i to j, inclusive, with syntactic label $l \in \mathcal{L}$.[2] For example, $(0, 1, \text{NP})$ is a constituent in Figure 2. *(Note: In lecture, some slides used i inclusive and j non-inclusive. This is just choice of notation – the algorithm is the same.)*

We use Bi-LSTMs, MLPs (multilayer perceptrons), and Biaffine functions to obtain a scoring function ϕ for the task. Then, similar to the previous project on Name Entity Recognition (NER), we use a Conditional Random Field (CRF) layer to incorporate constituency structure among words. We will describe each component and the decoding and learning algorithm in details in the following parts.

[2]Note that we don't reason about unary rules or pre-terminal rules. Therefore, $(0, 0, \text{NNP})$ is not a span, but a POS tag.

Bi-LSTM features. As we've learned from lectures and the previous two assignments, recurrent neural networks (RNNs) are a natural choice for computing learned feature representations of sequences. Given a sentence with POS tags, $\boldsymbol{X} = \{\boldsymbol{W}, \boldsymbol{G}\}$, we encode it as sequence of word embedding vectors $\boldsymbol{w}_1 \dots \boldsymbol{w}_T$ and tag embedding vectors $\boldsymbol{g}_1 \dots \boldsymbol{g}_T$ before passing it to the model. The word embedding and tag embedding and the same position are simply concatenated.

$$\boldsymbol{x}_t = \text{CONCAT}(\boldsymbol{w}_t, \boldsymbol{g}_t), \tag{1}$$

We then apply a Bi-LSTM to this embedding sequence.

$$\overrightarrow{\boldsymbol{h}}_t = \tanh(\boldsymbol{W_{ih}}\boldsymbol{x}_t + \boldsymbol{b_{ih}} + \boldsymbol{W_{hh}}\boldsymbol{h}_{(t-1)} + \boldsymbol{b_{hh}}). \tag{2}$$

For bidirectional RNNs, hidden states $\overrightarrow{\boldsymbol{h}}_t$, $\overleftarrow{\boldsymbol{h}}_t$ are computed in forward and backward directions, respectively, and concatenated to obtain $\boldsymbol{h}_t = \left[\overrightarrow{\boldsymbol{h}}_t; \overleftarrow{\boldsymbol{h}}_t\right]$.

Boundary representation. We apply two separate MLPs, MLP_{left} and MLP_{right}, to h_t, to create feature representations of both left and right span endpoints.

$$\boldsymbol{r}_i^{left} = \text{MLP}_{left}(\boldsymbol{h}_t) \tag{3}$$

$$\boldsymbol{r}_i^{right} = \text{MLP}_{right}(\boldsymbol{h}_t) \tag{4}$$

Biaffine scoring function. The biaffine scoring layer was first introduce in [3]. Given the left and right span representations, $\boldsymbol{r}_i^{left}$, $\boldsymbol{r}_i^{right}$, for each position i, we score each candidate span (i, j) using a biaffine operation over the left boundary representation r_i and the right boundary representation r_j, using an intermediate parameter matrix, $\boldsymbol{w}_l^{label}$, that depends on the label l. Thus, the score a span (i, j) with label l is given by:

$$\phi(i, j, l) = \exp\left(\text{Biaffine}(\boldsymbol{r}_i^{left}, \boldsymbol{r}_j^{right}, l)\right) \tag{5}$$

$$= \exp\left(\boldsymbol{r}_i^{left}\boldsymbol{w}_l^{label}\boldsymbol{r}_j^{right}\right) \tag{6}$$

where $\boldsymbol{w}_l^{label}$ is a weight matrix for label l. And, accordingly, the vectorized version will be:

$$\phi(i, j, \boldsymbol{L}) = \exp\left(\boldsymbol{r}_i^{left}\boldsymbol{W}^{label}\boldsymbol{r}_j^{right}\right) \tag{7}$$

Note that, $\boldsymbol{W}^{label}$ is a tensor defined as $[\boldsymbol{w}_1^{label}, \boldsymbol{w}_2^{label}, .., \boldsymbol{w}_{|\mathcal{L}|}^{label}]$.

Factorizing the production rules. In the neural CRF we present here in this project, all labels are independent of one-another given X. This is not true in PCFGs. In this project, the CRF structure is just for enforcing tree-shape, while the BiLSTM will predict labels for spans independently. This means our factorization assumptions are simpler. For a full sentence X, its score is the product of all constituent spans' scores:

$$\Phi(\boldsymbol{X}, \boldsymbol{Y}) = \prod_{(i,j,l)\in\boldsymbol{Y}} \phi(i, j, l) \tag{8}$$

And the conditional log likelihood under the model is:

$$\log P(\boldsymbol{Y}|\boldsymbol{X}) = \log \frac{\Phi(\boldsymbol{X}, \boldsymbol{Y})}{\sum_{\boldsymbol{Y}' \in \mathcal{O}(\boldsymbol{X})} \Phi(\boldsymbol{X}, \boldsymbol{Y}')} \tag{9}$$

Here, $\mathcal{O}(\boldsymbol{X})$ denotes the set of valid binary trees over sentence $\boldsymbol{X}$.

CYK Decoding. Given the log values of the local scoring potentials we computed earlier, $\log \phi(i, j, l)$, we can use the Cocke–Younger–Kasami (CYK) algorithm [3] to obtain the best scoring constituency tree $\hat{\boldsymbol{Y}}$ (i.e. whose score is maximum among all the possible parse trees):

$$\hat{\boldsymbol{Y}} = \text{argmax}_{\boldsymbol{Y} \in \mathcal{O}(\boldsymbol{X})} P(\boldsymbol{Y}|\boldsymbol{X}) \tag{10}$$

$$= \text{argmax}_{\boldsymbol{Y} \in \mathcal{O}(\boldsymbol{X})} \Phi(\boldsymbol{X}, \boldsymbol{Y}) \tag{11}$$

The original CYK is a dynamic programming algorithm with $O(n^3 \cdot |\mathcal{R}|)$ complexity, where $\mathcal{R}$ is the set of all possible production rules in the underlying PCFG. However, in a neural CRF parser, we actually make the parent label and child label independent in our parametrization (see Eq. 8). Therefore, the CYK decoding algorithm in this assignment is just $O(n^3)$ time complexity (if the maxes in the pseudo-code below are pre-computed and cached). We provide an implementation of the algorithm in the CSE291 repository, and the pseudo-code (non-batched version) is as below.

```
Input:

    Score: a scoring matrix of shape [T, T, L]
            # stores the logs of the phi scores computed earlier
    T: length of sentence
    L: the number of labels

Output:

    Result: a result matrix of shape [T, T]
    # stores the scores of best sub-trees for each span

Algorithm:

    # all the data are zero-indexed
    Initialize matrix Result with -inf
    For i = 0 to T - 1:
        Result[i, i] = 0

    For d = 2 to T:             # for each span length
        For i = 0 to T - d:     # for each start
            j := i + d - 1      # end of span
            For k = i to j - 1: # for each split point of span
                Result[i, j] := max(Result[i, j],
                        max(Score[i, j, *]) + Result[i, k] + Result[k+1, j])

    Return Result
```

[3] https://en.wikipedia.org/wiki/CYK_algorithm

Learning. In order to estimate model parameters θ, we minimize the negative log likelihood over *entire sequences* in the dataset $\{\boldsymbol{\mathcal{X}}, \boldsymbol{\mathcal{Y}}^*\}$:

$$\text{NLL} = \sum^{\boldsymbol{\mathcal{X}}, \boldsymbol{\mathcal{Y}}^*} -\log p(\boldsymbol{Y}^* \mid \boldsymbol{X}; \theta) = \sum^{\boldsymbol{\mathcal{X}}, \boldsymbol{\mathcal{Y}}^*} -\Big[\log \Phi(\boldsymbol{X}, \boldsymbol{Y}^*) - \log \sum_{\boldsymbol{Y}' \in \mathcal{O}(\boldsymbol{X})} \Phi(\boldsymbol{X}, \boldsymbol{Y}')\Big] \tag{12}$$

To compute the partition function, $\log \sum_{\boldsymbol{Y} \in \mathcal{O}(\boldsymbol{X})} \Phi(\boldsymbol{X}, \boldsymbol{Y})$, we can use an algorithm similar to the CKY algorithm we defined above for decoding from the model.

Given all the log scores scores of labeled spans with from the biaffine function (technically, this will be a 3 dimensions matrix of shape `[seq_length, seq_length, n_labels]`), we can define pseudo-code for computing the partition function:

```
Input:

    Score: a scoring matrix of shape [T, T, L]
           # stores the logs of the phi scores computed earlier
    T: length of sentence
    L: the number of labels

Output:

   S: a result matrix of shape [T, T], where S[0, T - 1] is the log partition function

Algorithm:
    # all the data are zero-indexed
    Initialize matrix S with -inf
    for i = 0 to T - 1:
        S[i, i] = 0

    For d = 2 to T:             # for each span length
        For i = 0 to T - d:     # for each start
            j := i + d - 1      # end of the span
            For k = i to j - 1: # for each split point of span
                S[i, j] := logSumExp(S[i, j],
                        logSumExp(Score[i, j, *]) + S[i, k] + S[k+1, j])

    Return S
```

In the starter code, we also provide some comments to help you understand how to implement the algorithm. All you need is to fill in the missing part of function **`inside()`**. You are allowed to implement it in your own way. However, we would suggest that you implement the vectorized version of it. Otherwise, your training process would significantly slow down on colab.

To implement the vectorized version, you will need first understand how the CYK decoding is implemented. You will find the helper function **`stripe()`**[4] and the PyTorch's Tensor operator **`diag()`**[5] helpful in implementing the vectorized version.

[4] See it in the course repository.

[5] `https://pytorch.org/docs/stable/generated/torch.diag.htmls`

Evaluation. The constituency trees in PTB data are n-array trees. However, our model actually reasons about Chomsky Normal Form (CNF)[6] trees. In the provided evaluation script, we first convert the prediction back to an n-array tree and then compute the recall, precision, and F_1 score.

4 Implementation Instructions

Implement the inside algorithm of the CRF layer defined above and use it for the task of Constituency Parsings. You are encouraged to clone and use the starter code from the CSE291 repository[7], which includes portions of the PTB data train/dev splits, data loading functionality, and evaluation using (labeled/unlabeled) span-based precision, recall, and F_1 score. The only part you need to implement is to add the required Inside algorithm for CRF negative log likelihood learning.

Please note that the added dependency structure of the CRF layer results in a longer runtime for both learning and prediction. We encourage you to use Google Colab[8] if you do not have access to a GPU. Feel free to report results on a subset of the data in your report if you do not have enough compute to train on the entire training set. We have already prepared two subsets under the folder **`./data`** for your convenience. Folder **`./data/PTB_LE10`**[9] includes 2000 sentences whose length are less than 10, while folder **`./data/PTB_first_2000`** includes the first 2000 sentences in the original PTB trainset. If you would like to have the full training data, you can download it from here[10].

Here's the best scoring evaluation result on **`./data/PTB_first_2000`** (A full training log is also attached in the appendix for reference):

```
dev:    loss: 0.6538 - UCM: 18.76% LCM: 17.35%
        UP: 84.74% UR: 86.06% UF: 85.40%
        LP: 82.59% LR: 83.87% LF: 83.23%

where

UCM/LCM stands for Unlabeled Complete Match,
UP/LP stands for Unlabeled/Labeled Precision,
UR/LR stands for Unlabeled/Labeled Recall,
UF/LF stands for Unlabeled/Labeled F1.
```

As a reminder, we will not be grading based on the absolute performance of your method as long as it performs reasonably well, indicating it is implemented correctly. If you approximately match the F1 numbers we provide above, you should expect you've implemented the algorithm correctly.

5 Deliverables

Please submit a 1–2 page report along with the source code on Gradescope. We suggest using the ACL style files,[11] which are also available as an Overleaf template.[12] The core of this project is about implementing the inside algorithm for a neural CRF parser. Thus, we don't require as much analysis

[6] https://en.wikipedia.org/wiki/Chomsky_normal_form

[7] https://github.com/tberg12/cse291spr21/assignment3

[8] https://colab.research.google.com/

[9] If you decide to implement the non-vertorized version, we would suggest you use this subset to train your model.

[10] https://raw.githubusercontent.com/nikitakit/self-attentive-parser/master/data/02-21.10way.clean

[11] https://2021.aclweb.org/downloads/acl-ijcnlp2021-templates.zip

[12] https://www.overleaf.com/latex/templates/instructions-for-acl-ijcnlp-2021-proceedings/mhxffkjdwymb

as in past projects. We ask you to briefly describe the overall approach, e.g. paraphrase the model description and math in your own words. Then we ask you briefly describe any specific design choices you made (e.g. how you vectorized your inside algorithm if you choose to do so). Then we ask you to present your evaluation results and runtime. Finally, we ask you to present two example predicted test trees and comment on errors you see vs. the ground-truth trees for the same examples. Shorter reports are fine for this project, e.g. 1-2 pages, though feel free to write more if you want to do additional analysis, try additional features (not required for full credit).

6 Useful References

While you can use the Parser package[13] as a reference, the code in your implementation must be your own. Note that the model of constituency parsing in the package is different from that of our assignment.

7 Optional Extensions

You may choose to carry out any or none of the following extra credit options:

- Simplify the neural architecture and compare it with our default model. For example, you can remove the tagging embedding layer, or use the same MLP layer (instead of two different MLP layers) to model the boundary representation $\boldsymbol{r}$.
- Choose a pre-trained language model (like BERT or GloVe) and use its embeddings as input to the CRF model. Or use a char-level LSTM layer to encode words instead of directly using an embedding layer.
- Add a discrete grammar (e.g. X-bar or one read directly off the treebank) to constrain your neural CRF parser – e.g. not all triples of parent label, left child label, and right child label are permitted under real discrete grammars.
- Change the biaffine scoring function to capture interactions between parent and child labels – i.e. learn a full neural grammar!
- Choose your own adventure! Propose and implement your own analysis or extension.

Collaboration Policy

You are allowed to discuss the assignment with other students and collaborate on developing algorithms – you're even allowed to help debug each other's code! However, every line of your write-up and the new code you develop must be written by your own hand.

References

[1] Greg Durrett and Dan Klein. Neural CRF parsing. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 302–312, Beijing, China, July 2015. Association for Computational Linguistics.

[13] `https://github.com/yzhangcs/parser`

[2] Yu Zhang, Houquan Zhou, and Zhenghua Li. Fast and accurate neural CRF constituency parsing. In *Proceedings of IJCAI*, pages 4046–4053, 2020.

[3] Timothy Dozat and Christopher D. Manning. Deep biaffine attention for neural dependency parsing. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

Appendix

The full training log is as below.

```
!cse291ass3 python main.py --batch-size 128
Building the fields
Namespace(batch_size=128, bos_index=2, buckets=32, clip=5.0,
delete={'', '.', ',', "''", '?', '!', 'S1', ':', '-NONE-', '``', 'TOP'},
dev='data/ptb/dev.pid', device='cuda', embed=None, encoder='lstm', eos_index=3,
epochs=10, eps=1e-12, equal={'ADVP': 'PRT'}, feat=None, lr=0.002, max_len=None,
mu=0.9, n_embed=100, n_labels=71, n_tags=45, n_words=3511, nu=0.9, pad_index=0,
test='data/ptb/test.pid', train='data/ptb/train.pid',
unk='unk', unk_index=1, weight_decay=1e-05)
Building the model
Model(
  (word_embed): Embedding(3511, 100)
  (tag_embed): Embedding(45, 100)
  (embed_dropout): Dropout(p=0.33, inplace=False)
  (encoder): LSTM(200, 400, num_layers=3, dropout=0.33, bidirectional=True)
  (encoder_dropout): Dropout(p=0.33, inplace=False)
  (mlp_l): MLP(n_in=800, n_out=100, dropout=0.33)
  (mlp_r): MLP(n_in=800, n_out=100, dropout=0.33)
  (feat_biaffine): Biaffine(n_in=100, n_out=71, bias_x=True, bias_y=True)
  (crf): CRFConstituency()
  (criterion): CrossEntropyLoss()
)

train: Dataset(n_sentences=2000, n_batches=409, n_buckets=32)
dev:   Dataset(n_sentences=1700, n_batches=342, n_buckets=32)
test:  Dataset(n_sentences=2416, n_batches=484, n_buckets=32)

Epoch 1 / 10:
0 iter of epoch 1, loss: 4.9900
50 iter of epoch 1, loss: 1.7499
100 iter of epoch 1, loss: 1.0927
150 iter of epoch 1, loss: 1.3784
200 iter of epoch 1, loss: 0.6478
250 iter of epoch 1, loss: 0.8427
300 iter of epoch 1, loss: 1.0710
350 iter of epoch 1, loss: 0.6919
400 iter of epoch 1, loss: 0.9706
dev:  loss: 0.8141 - UCM:  8.71% LCM:  7.41%
UP: 77.13% UR: 80.77% UF: 78.91%
LP: 74.12% LR: 77.63% LF: 75.84%
Epoch 2 / 10:
0 iter of epoch 2, loss: 0.9249
```

```
50 iter of epoch 2, loss: 0.5275
100 iter of epoch 2, loss: 0.8728
150 iter of epoch 2, loss: 0.7860
200 iter of epoch 2, loss: 0.8848
250 iter of epoch 2, loss: 1.0504
300 iter of epoch 2, loss: 0.5526
350 iter of epoch 2, loss: 1.1311
400 iter of epoch 2, loss: 0.8681
dev:  loss: 0.6986 - UCM: 12.47% LCM: 10.76%
UP: 81.49% UR: 83.45% UF: 82.46%
LP: 78.86% LR: 80.76% LF: 79.79%
Epoch 3 / 10:
0 iter of epoch 3, loss: 0.7196
50 iter of epoch 3, loss: 0.3954
100 iter of epoch 3, loss: 1.0758
150 iter of epoch 3, loss: 0.3884
200 iter of epoch 3, loss: 0.7023
250 iter of epoch 3, loss: 0.6243
300 iter of epoch 3, loss: 0.5182
350 iter of epoch 3, loss: 0.5139
400 iter of epoch 3, loss: 0.5725
dev:  loss: 0.6529 - UCM: 14.88% LCM: 13.59%
UP: 83.02% UR: 84.66% UF: 83.83%
LP: 80.62% LR: 82.20% LF: 81.40%
Epoch 4 / 10:
0 iter of epoch 4, loss: 0.6344
50 iter of epoch 4, loss: 0.4191
100 iter of epoch 4, loss: 0.6893
150 iter of epoch 4, loss: 0.5572
200 iter of epoch 4, loss: 0.9136
250 iter of epoch 4, loss: 1.1216
300 iter of epoch 4, loss: 0.6391
350 iter of epoch 4, loss: 0.5298
400 iter of epoch 4, loss: 0.8414
dev:  loss: 0.6712 - UCM: 15.41% LCM: 14.18%
UP: 81.03% UR: 84.04% UF: 82.51%
LP: 78.68% LR: 81.60% LF: 80.11%
Epoch 5 / 10:
0 iter of epoch 5, loss: 0.4830
50 iter of epoch 5, loss: 0.5475
100 iter of epoch 5, loss: 0.7482
150 iter of epoch 5, loss: 0.9239
200 iter of epoch 5, loss: 0.3832
250 iter of epoch 5, loss: 0.6287
300 iter of epoch 5, loss: 0.4790
350 iter of epoch 5, loss: 0.4887
400 iter of epoch 5, loss: 0.5246
dev:  loss: 0.6426 - UCM: 17.24% LCM: 16.00%
UP: 84.04% UR: 85.02% UF: 84.53%
LP: 81.96% LR: 82.92% LF: 82.44%
Epoch 6 / 10:
0 iter of epoch 6, loss: 0.6358
50 iter of epoch 6, loss: 0.3243
100 iter of epoch 6, loss: 0.4326
```

```
150 iter of epoch 6, loss: 0.2156
200 iter of epoch 6, loss: 0.2075
250 iter of epoch 6, loss: 0.3221
300 iter of epoch 6, loss: 0.6566
350 iter of epoch 6, loss: 0.3273
400 iter of epoch 6, loss: 0.4782
dev:  loss: 0.6857 - UCM: 17.71% LCM: 16.24%
UP: 85.12% UR: 84.78% UF: 84.95%
LP: 83.04% LR: 82.71% LF: 82.88%
Epoch 7 / 10:
0 iter of epoch 7, loss: 0.6639
50 iter of epoch 7, loss: 0.2892
100 iter of epoch 7, loss: 0.4901
150 iter of epoch 7, loss: 0.4327
200 iter of epoch 7, loss: 0.3621
250 iter of epoch 7, loss: 0.4680
300 iter of epoch 7, loss: 0.3519
350 iter of epoch 7, loss: 0.3976
400 iter of epoch 7, loss: 0.2718
dev:  loss: 0.6737 - UCM: 18.00% LCM: 16.65%
UP: 84.99% UR: 84.80% UF: 84.89%
LP: 83.04% LR: 82.86% LF: 82.95%
Epoch 8 / 10:
0 iter of epoch 8, loss: 0.2797
50 iter of epoch 8, loss: 0.4043
100 iter of epoch 8, loss: 0.4003
150 iter of epoch 8, loss: 0.3425
200 iter of epoch 8, loss: 0.5250
250 iter of epoch 8, loss: 0.6620
300 iter of epoch 8, loss: 0.2005
350 iter of epoch 8, loss: 0.7108
400 iter of epoch 8, loss: 0.5317
dev:  loss: 0.6942 - UCM: 18.65% LCM: 17.41%
UP: 84.95% UR: 85.95% UF: 85.45%
LP: 82.77% LR: 83.75% LF: 83.26%
Epoch 9 / 10:
0 iter of epoch 9, loss: 0.5051
50 iter of epoch 9, loss: 0.7551
100 iter of epoch 9, loss: 0.5349
150 iter of epoch 9, loss: 0.3717
200 iter of epoch 9, loss: 0.3090
250 iter of epoch 9, loss: 0.5121
300 iter of epoch 9, loss: 0.8158
350 iter of epoch 9, loss: 0.4875
400 iter of epoch 9, loss: 0.5440
dev:  loss: 0.6900 - UCM: 20.24% LCM: 19.00%
UP: 84.65% UR: 86.30% UF: 85.47%
LP: 82.43% LR: 84.04% LF: 83.23%
Epoch 10 / 10:
0 iter of epoch 10, loss: 0.3559
50 iter of epoch 10, loss: 0.2764
100 iter of epoch 10, loss: 0.2985
150 iter of epoch 10, loss: 0.3805
200 iter of epoch 10, loss: 0.4140
```

```
250 iter of epoch 10, loss: 0.4455
300 iter of epoch 10, loss: 0.8627
350 iter of epoch 10, loss: 0.4133
400 iter of epoch 10, loss: 0.6942
dev:  loss: 0.6538 - UCM: 18.76% LCM: 17.35%
UP: 84.74% UR: 86.06% UF: 85.40%
LP: 82.59% LR: 83.87% LF: 83.23%
```