CS 533: Natural Language Processing **(Due: 03/31/20)**

Assignment 4

Name: Prakruti Joshi, *NetID:* phj15 *Students discussed with:* tn268,kd706

Problem 1: HMM

((1 + 6 + 6 = 13 points))

1. Emission probabilities $o(x|y)$:

$o(x\|y)$	Probability value
$the\|D$	1.0
$cut\|N$	0.167
$man\|N$	0.33
$saw\|N$	0.33
$the\|N$	0.167
$cut\|V$	0.5
$saw\|V$	0.5

Transition probabilities $o(x|y)$:

$t(y^{'}\|y)$	Probability value
$D\|*$	0.67
$D\|V$	1.0
$N\|*$	0.33
$N\|D$	1.0
$N\|N$	0.167
$STOP\|N$	0.5
$V\|N$	0.33

Non-zero emission probabilities for the word **cut** are:

- $o(cut|N) = 0.167$
- $o(cut|V) = 0.5$

2. Probability under the HMM that the third word is tagged with V conditioning on x(2) = ”the saw cut the man” is as follows:

$$\sum_{(y1,y2,y3,y4,y5)\in Y:y3=V} p(y1, y2, y3, y4, y5|\text{the saw cut the man})$$
$$= alpha(3, V) * beta(3, V)$$
$$= 0.03755 * 0.1667$$
$$= 0.00626$$

3. Probability that the fifth word is tagged with N conditioning on x(1) = "the man saw the cut" is as follows:

$$\begin{aligned}\sum_{(y1,y2,y3,y4,y5)\in Y:y5=N} & p(y1,y2,y3,y4,y5|\text{the man saw the cut}) \\ &= alpha(5,N) * beta(5,N) \\ &= 0.00626 * 0.5 \\ &= 0.00313\end{aligned}$$

Problem 2: PCFG ((1 + 6 + 6 = 13 points))

1. MLE parameter values of (u, b) estimated from the given corpus.
   Unary Rule Probabilities

$u(x\|y)$	Probability
$u(the\|D)$	0.67
$u(a\|D)$	0.33
$u(saw\|V)$	1.0
$u(with\|P)$	1.0
$u(boy\|N)$	0.33
$u(man\|N)$	0.33
$u(telescope\|N)$	0.33

   Binary Rule Probabilities

$b(X \to YZ)$	Probability
$b(S \to NP, VP)$	1.0
$b(PP \to P, NP)$	1.0
$b(NP \to D, N)$	0.857
$b(NP \to NP, PP)$	0.143
$b(VP \to VP, PP)$	0.33
$b(VP \to V, NP)$	0.67

2. Probability under the PCFG that **NP spans (4, 8)** (i.e., "the man with a telescope") conditioning on x is as follows:

$$\begin{aligned}\sum_{\tau\in GEN(x):root(\tau,4,8)=NP} & p(\tau|x) \\ &= alpha(4,8,NP) * beta(4,8,NP) \\ &= 0.00259 * 0.12698 \\ &= 0.000329\end{aligned}$$

3. Probability under the PCFG that **VP spans (3, 5)** (i.e., "saw the man") conditioning on x is as follows:

$$\begin{aligned}\sum_{\tau\in GEN(x):root(\tau,3,5)=VP} & p(\tau|x) \\ &= alpha(3,5,VP) * beta(3,5,VP) \\ &= 0.12698 * 0.00605 \\ &= 0.000768\end{aligned}$$

Problem 3: Programming (CRF) ((1 + 1 + 1 + 1 + 1 + 1 + 8 + 8 = 22 points))

1. (a) The word sequences are sorted in descending order of their length. While forming batches for training, testing and evaluation; the function *batchfy* in the class *tagging_dataset* forms the batches depending on the length word sequence. Word sequences with same length are put in the same batch. Hence, word sequences are sorted according to their length.

   (b) No, every batch size will not contain N sequences. This is because, the batch is created on two conditions:
   1. length of the word sequence
   2. maximum size of batch (N) : *batch_size*
   Thus, if there are $m <= N$, sequences of length l, then the batch will have m samples in it. The batch size over here defines the maximum sequences a batch can have.

   (c) No. There is no padding at the word sequence level. Since the batch has same length word sequences, no padding is required.

   (d) The characters in each batch are stored in *cseqs*. This consists of the characters of each word converted into the corresponding index mapping. The character list has *torch.LongTensors* representing the characters of a particular word.

   (e) Yes. The flattened list of character list is further padded with $< pad >$ as the padding value. The function *pad_sequence* from *torch.nn.utils.rnn* is used to pad the character sequence of words.

2. In the *evaluate* method in *BiLSTMTagger* class, each batch is evaluated and the accuracy reports are calculated. The method first runs a forward propagation with the given input and calculates the score by using *self.score*. The scores are decoded using *CRFLoss* which returns the indice of the tag with the maximum score. The accuracy is calculated by keeping count of number of total predictions and number of correctly predicted. Number of correctly predicted is calculated as follows:

$$num\_correct += \; (preds == Y).sum().item()$$

   If the tag sequences are in BIO format, then F1 score is calculated by keeping the counts of true positive (*tp*), false positive (*fp*), and false negatives (*fn*). These counts are kept for each entity. The entities in the input data and the target sequence is determined using *util.get_boundaries*. If the predicted entity is same as the target entity, then *tp* is incremented otherwise *fn* is incremented for that entity. If the entity is not present in the target sequence boundary but is present in the predicted boundary, then *fp* is incremented for that entity. The final F1 score for each entity is calculated based on the values of *tp,fp,fn*.

3. In the BiLSTM layer,

$$h^{(i)} = nn.Dropout(nn.Linear(nn.LSTM(x^{(i)})))$$

   The loss for a single point is calculated using the Crossentropy loss as follows:

$$loss(x^{(i)}, h^{(i)}, y^{(i)}) = -\log(\frac{\exp(h^{(i)}[y^{(i)}])}{\sum_{j=1}^{L} \exp(h^{(i)}[j])})$$

   The final loss is the average loss of the data points in the batch.

$$GreedyLoss = \frac{\sum_{i \in Batch} \text{loss}(x^{(i)}, h^{(i)}, y^{(i)})}{\text{batch\_size}}$$

4. (a) Given a word w, if character level information is to be incorporated, first a character embedding *cemb* is produced using *nn.Embedding* which takes the number of unique characters and the character emdedding dimension *dim_char* as parameters. Then, the character emdedding is passed into a bi-LSTM layer *wlstm* which is defined in *BiLSTMOverCharacters* class. The feedforward output of this bi-LSTM layer is contactenated with the word embeddings of the input to be passed as the input to the bi-LSTM layer over words. Thus, character level information is incorporated in the input.

(b) The final dimension of the input word representation is:

$$input\_dim = wdim + 2 * cdim$$

5. (a) The parameters of CRFLoss in *crf.py* are:
   i. **start**: *nn.Parameter* of length as number of tag types L
   ii. **T**: *nn.Parameter* of shape $L \times L$
   iii. **end**: *nn.Parameter* of length L

   (b) Score in terms of the CRF parameters:

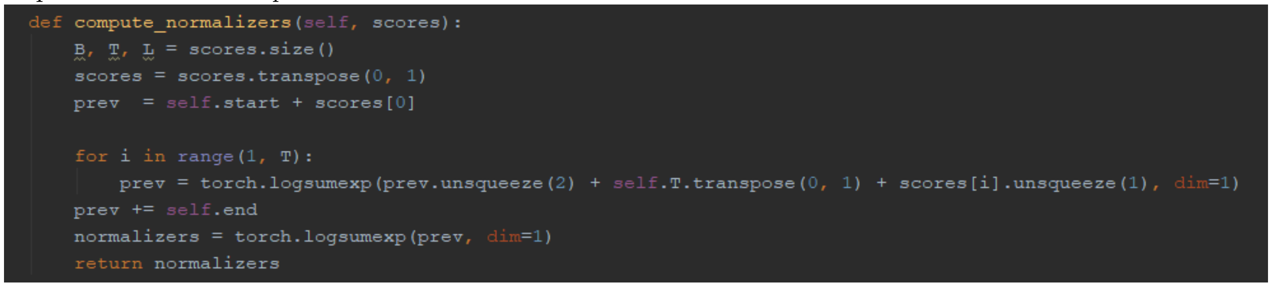
$$score(h,\, y) = \sum_{i=1}^{T} h_i[y_i] + \text{self.start}[y_0] + \sum_{i=2}^{T-1} \text{self.T}[y_i, y_{i-1}] + \text{self.end}[y_T]$$

   (c) Loss is computed by calculating the mean of the individul errors:

$$loss = Mean(\text{normalizers} - \text{target\_scores})$$

6. The method *compute_normalizers_brutes* calculates sequence scores for all possible target sequences generated by *itertools.product*.A final normalised score is produced by taking the log of summation of each of exponential of the scores.
The method *decode_brute* iterates over all possible combinations of target sequences of labels, stores these sequences and calculates the score for each of the sequence. Finally it returns the maximum score produced by a target sequence along with the corresponding sequence by using *torch.stack* and *torch.max*.
Time complexity:
Both the functions iterate over all possible target sequences and hence take $O(L^T)$ time complexity for this step. To compute scores using *score_targets*, it takes O(T) time. Thus, the total time complexity for both the functions is $O(T.L^T)$

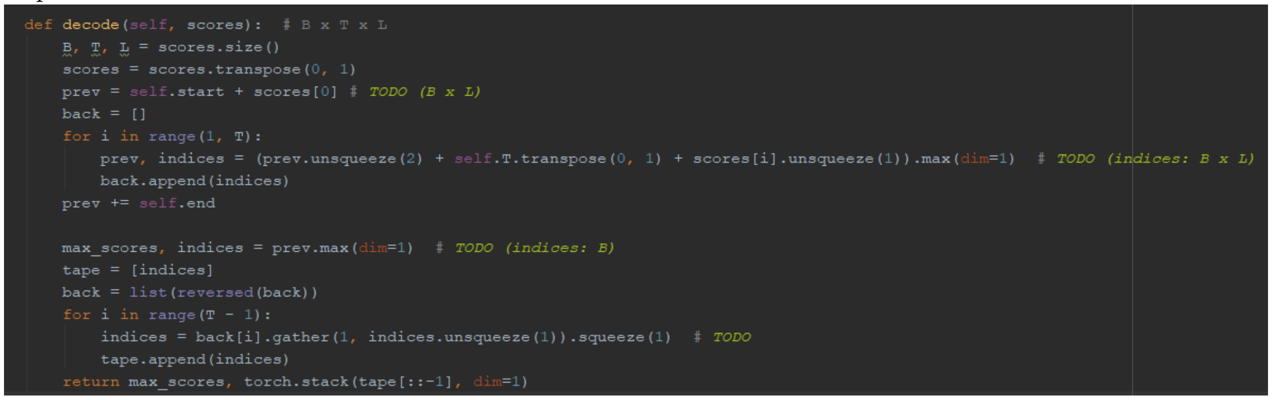
7. Implementation of *compute_ normalizers*:



```
def compute_normalizers(self, scores):
    B, T, L = scores.size()
    scores = scores.transpose(0, 1)
    prev  = self.start + scores[0]

    for i in range(1, T):
        prev = torch.logsumexp(prev.unsqueeze(2) + self.T.transpose(0, 1) + scores[i].unsqueeze(1), dim=1)
    prev += self.end
    normalizers = torch.logsumexp(prev, dim=1)
    return normalizers
```

8. Implementation of *decode*:



```
def decode(self, scores):  # B x T x L
    B, T, L = scores.size()
    scores = scores.transpose(0, 1)
    prev = self.start + scores[0] # TODO (B x L)
    back = []
    for i in range(1, T):
        prev, indices = (prev.unsqueeze(2) + self.T.transpose(0, 1) + scores[i].unsqueeze(1)).max(dim=1)  # TODO (indices: B x L)
        back.append(indices)
    prev += self.end

    max_scores, indices = prev.max(dim=1)  # TODO (indices: B)
    tape = [indices]
    back = list(reversed(back))
    for i in range(T - 1):
        indices = back[i].gather(1, indices.unsqueeze(1)).squeeze(1)  # TODO
        tape.append(indices)
    return max_scores, torch.stack(tape[::-1], dim=1)
```

Passed all the three tests in *test_crf.py*