

## Blog of Qing

### NLP-Dependency Parser

📅 2019-06-12 | 📁 [NLP](#)

Dependency structure of sentences shows which words depend on (modify or are arguments of) which other words. These binary asymmetric relations between the words are called dependencies and are depicted as arrows going from the head (or governor, superior, regent) to the dependent (or modifier, inferior, subordinate).

#### What is Dependency Parsing

Dependency parsing is the task of analyzing the syntactic dependency structure of a given input sentence  $S$ . The output of a dependency parser is a dependency tree where the words of the input sentence are connected by typed dependency relations. Formally, the dependency parsing problem asks to create a mapping from the input sentence with words  $S = w_0 w_1 \dots w_n$  (where  $w_0$  is the ROOT) to its dependency tree graph  $G$ .

To be precise, there are two subproblems in dependency parsing:

1. Learning: Given a training set  $D$  of sentences annotated with dependency graphs, induce a parsing model  $M$  that can be used to parse new sentences.
2. Parsing: Given a parsing model  $M$  and a sentence  $S$ , derive the optimal dependency graph  $D$  for  $S$  according to  $M$ .

#### How to train a Dependency Parsing

## Gready Deterministic Transition-Based Parsing

This transition system is a state machine, which consists of states and transitions between those states. The model induces a sequence of transitions from some initial state to one of several terminal states.

### States:

For any sentence  $S = w_0 w_1 \dots w_n$ , a state can be described with a triple  $c = (\alpha, \beta, A)$ :

1. a stack  $\alpha$  of words  $w_i$  from  $S$
2. a buffer  $\beta$  of words  $w_i$  from  $S$
3. a set of dependency arcs  $A$  of the form  $(w_i, r, w_j)$ , where  $r$  describes a dependency relation

It follows that for any sentence  $S = w_0 w_1 \dots w_n$ ,

1. an initial state  $c_0$  is of the form  $([w_0]_\alpha, [w_1, \dots, w_n]_{\beta}, \text{\textcolor{red}{empty}})$  (only the ROOT is on the stack, all other words are in the buffer and no actions have been chosen yet)
2. a terminate state has the form  $(\alpha, [], A)$

### Transitions:

There are three types of transitions between states:

1. Shift: Remove the first word in the buffer and push it on top of the stack. (Pre-condition: buffer has to be non-empty.)
2. LEFT-ARC: Add a dependency arc  $(w_j, r, w_i)$  to the arc set  $A$ , where  $w_i$  is the word second to the top of the stack and  $w_j$  is the word at the top of the stack. Remove  $w_i$  from the stack.

$[w_0, w_1, \dots, w_i \leftarrow w_j]$

3. RIGHT-ARC: Add a dependency arc  $(w_i, r, w_j)$  to the arc set  $A$ , where  $w_i$  is the word second to the top of the stack and  $w_j$  is the word at the top of the stack. Remove  $w_j$  from the stack.

$[w_0, w_1, \dots, w_i \rightarrow w_j]$

## Neural Dependency Parsing

The network is to predict the transitions between words, i.e., *Shift*, *Left – Arc*, *Right – Arc*.

## Feature Selection:

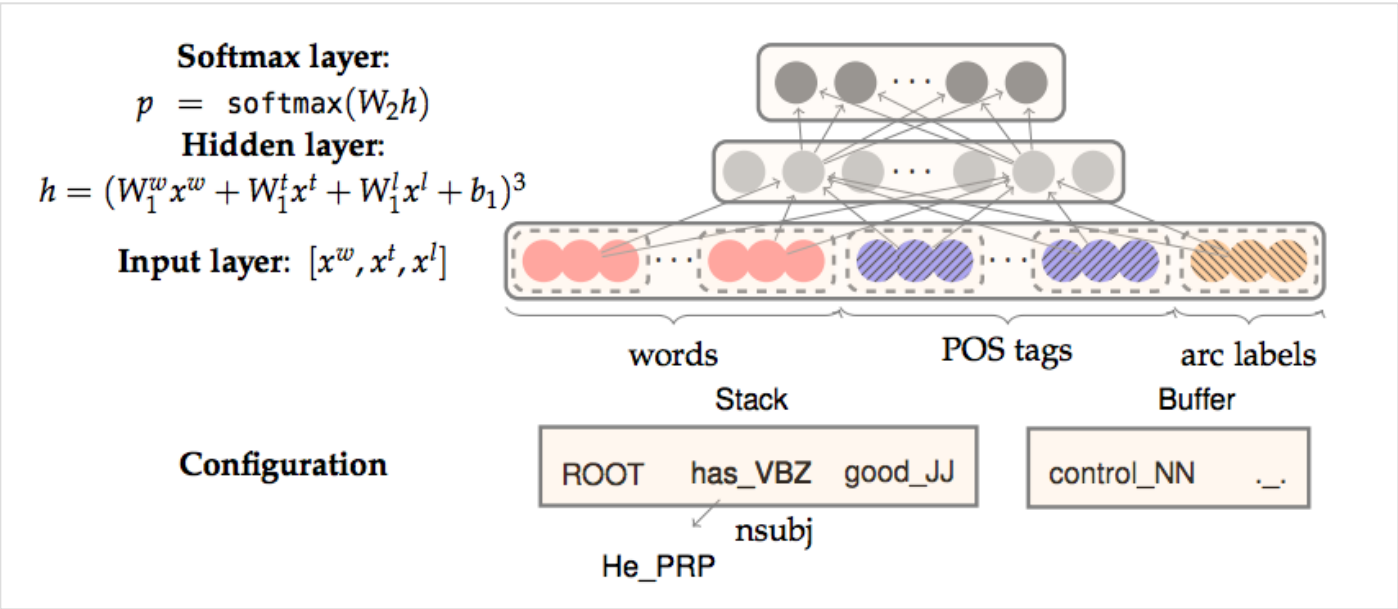
Depending on the desired complexity of the model, there is flexibility in defining the input to the neural network. The features for a given sentence  $S$  generally include some subset of:

1.  $S_{word}$ : Vector representations for some of the words in  $S$  (and their dependents) at the top of the stack  $\sigma$  and buffer  $\beta$ .
2.  $S_{tag}$ : Part-of-Speech (POS) tags for some of the words in  $S$ . POS tags comprise a small, discrete set:  $\mathcal{P} = \{NN, NNP, NNS, DT, JJ, \dots\}$
3.  $S_{label}$ : The arc-labels for some of the words in  $S$ . The arc-labels comprise a small, discrete set, describing the dependency relation:  $\mathcal{L} = \{amod, tmod, nsubj, csubj, dobj, \dots\}$

For each feature type, we will have a corresponding embedding matrix, mapping from the feature's one hot encoding, to a  $d$ -dimensional dense vector representation.

## Feedforward Neural Network Model:

The network contains an input layer  $[x^w, x^t, x^l]$ , a hidden layer, and a final softmax layer with a cross-entropy loss function. We can either define a single weight matrix in the hidden layer, to operate on a concatenation of  $[x^w, x^t, x^l]$ , or we can use three weight matrices  $[W_1^w, W_1^t, W_1^l]$ , one for each input type, as shown in Figure 3. We then apply a non-linear function and use one more affine layer  $[W_2]$  so that there are an equivalent number of softmax probabilities to the number of possible transitions (the output dimension).



Example

(a) (6 points) Go through the sequence of transitions needed for parsing the sentence “*I parsed this sentence correctly*”. The dependency tree for the sentence is shown below. At each step, give the configuration of the stack and buffer, as well as what transition was applied this step and what new dependency was added (if any). The first three steps are provided below as an example.

Stack	Buffer	New dependency	Transition
[ROOT]	[I, parsed, this, sentence, correctly]		Initial Configuration
[ROOT, I]	[parsed, this, sentence, correctly]		SHIFT
[ROOT, I, parsed]	[this, sentence, correctly]		SHIFT
[ROOT, parsed]	[this, sentence, correctly]	parsed→I	LEFT-ARC

**Answer:**

Stack	Buffer	New dependency	Transition
[ROOT]	[I, parsed, this, sentence, correctly]		Initial Configuration
[ROOT, I]	[parsed, this, sentence, correctly]		SHIFT
[ROOT, I, parsed]	[this, sentence, correctly]		SHIFT
[ROOT, parsed]	[this, sentence, correctly]	parsed → I	LEFT-ARC
[ROOT, parsed, this]	[sentence, correctly]		SHIFT
[ROOT, parsed, this, sentence]	[correctly]		SHIFT
[ROOT, parsed, sentence]	[correctly]	sentence → this	LEFT-ARC
[ROOT, parsed]	[correctly]	parsed → sentence	RIGHT-ARC
[ROOT, parsed, correctly]	[]		SHIFT
[ROOT, parsed]	[]	parsed → correctly	RIGHT-ARC
[ROOT]	[]	ROOT → parsed	RIGHT-ARC

A sentence containing  $n$  words will be parsed in  $2n$  steps. Because for each word, it will be pushed from buffer to stack, which result in  $n$  steps. Then each word in the stack has to be assigned a transition, i.e., LEFT-ARC or RIGHT-ARC, which results in another  $n$  steps. Therefore, the total steps are  $2n$ .

## Pytorch Implementation

```

1 class PartialParse(object):
2     def __init__(self, sentence):
3         """Initializes this partial parse.
4
5         @param sentence (list of str): The sentence to be parsed as a list of words.
6
7         Your code should not modify the sentence.
8
9         """

```

```
8      # The sentence being parsed is kept for bookkeeping purposes. Do not alter it in yo
9      self.sentence = sentence
10
11     ### YOUR CODE HERE (3 Lines)
12
13     ### Your code should initialize the following fields:
14
15     ###     self.stack: The current stack represented as a list with the top of the sta
16
17     ###         last element of the list.
18
19     ###     self.buffer: The current buffer represented as a list with the first item o
20
21     ###         buffer as the first item of the list
22
23     ###     self.dependencies: The list of dependencies produced so far. Represented as
24
25     ###         tuples where each tuple is of the form (head, dependent).
26
27     ###         Order for this list doesn't matter.
28
29     ###
30
31     ### Note: The root token should be represented with the string "ROOT"
32
33     ###
34
35     self.stack = ['ROOT']
36
37     self.buffer = sentence.copy()
38
39     self.dependencies = []
40
41     ### END YOUR CODE
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
```

31

```
32     @param transition (str): A string that equals "S", "LA", or "RA" representing the s
33                               left-arc, and right-arc transitions. You can assume the pro
34                               transition is a legal transition.
```

```
35     """
```

```
36     ### YOUR CODE HERE (~7-10 Lines)
```

```
37     ### TODO:
```

```
38     ###     Implement a single parsing step, i.e. the logic for the following as
```

```
39     ###     described in the pdf handout:
```

```
40     ###         1. Shift
```

```
41     ###         2. Left Arc
```

```
42     ###         3. Right Arc
```

```
43     if transition=='S':
```

```
44         self.stack.append(self.buffer.pop(0))
```

```
45     elif transition=='LA':
```

```
46         stack_second = self.stack.pop(-2)
```

```
47         self.dependencies.append((self.stack[-1],stack_second))
```

```
48     else:
```

```
49         stack_top = self.stack.pop()
```

```
50         self.dependencies.append((self.stack[-1],stack_top))
```

```
51
```

```
52     ### END YOUR CODE
```

```
53
```

```
54     def parse(self, transitions):
```

```
55         """Applies the provided transitions to this PartialParse

56
57         @param transitions (list of str): The list of transitions in the order they should
58
59         @return ddependencies (list of string tuples): The list of dependencies produced wh
60
61
62         parsing the sentence. Represented a
63
64         tuples where each tuple is of the f
65
66         """
67
68         for transition in transitions:
69
70             self.parse_step(transition)
71
72         return self.dependencies
73
74
75
76
77
78 def minibatch_parse(sentences, model, batch_size):
79
80     """Parses a list of sentences in minibatches using a model.
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
```



79

80

```
81     @return dependencies (list of dependency lists): A list where each element is the dependen
82                                     list for a parsed sentence. Ordering sh
83                                     same as in sentences (i.e., dependencies
84                                     contain the parse for sentences[i]).
```

```
85     """
```

```
86     dependencies = []
```

87

```
88     ### YOUR CODE HERE (~8-10 Lines)
```

```
89     ### TODO:
```

```
90     ###     Implement the minibatch parse algorithm as described in the pdf handout
```

```
91     ###
```

```
92     ###     Note: A shallow copy (as denoted in the PDF) can be made with the "=" sign in p
```

```
93     ###         unfinished_pauses = partial_pauses[:].
```

```
94     ###         Here `unfinished_pauses` is a shallow copy of `partial_pauses`.
```

```
95     ###         In Python, a shallow copied list like `unfinished_pauses` does not cont
```

```
96     ###         of the object stored in `partial_pauses`. Rather both lists refer to th
```

```
97     ###         In our case, `partial_pauses` contains a list of partial parses. `unfin
```

```
98     ###         contains references to the same objects. Thus, you should NOT use the `
```

```
99     ###         to remove objects from the `unfinished_pauses` list. This will free the
```

```
100    ###         is being accessed by `partial_pauses` and may cause your code to crash.
```

101

```
102    #initialization
```

```
103     partial_parsers = []

104     for sentence in sentences:

105         partial_parsers.append(PartialParse(sentence))

106     unfinished_parsers = partial_parsers[:]

107

108     while unfinished_parsers:

109         minibatch = unfinished_parsers[:batch_size]

110         transitions = model.predict(minibatch)

111         for i,parses in enumerate(minibatch):

112             parses.parse([transitions[i]])

113             if len(parses.buffer)==0 and len(parses.stack)==1:

114                 dependencies.append(parses.dependencies)

115                 unfinished_parsers.remove(parses)

116     ### END YOUR CODE

117     return dependencies
```

## Train Parser Network

```
1  import torch

2  import torch.nn as nn

3  import torch.nn.functional as F

4

5  class ParserModel(nn.Module):

6      """ Feedforward neural network with an embedding layer and single hidden layer.

7          The ParserModel will predict which transition should be applied to a
```

```
8     given partial parse configuration.

9

10    PyTorch Notes:

11        - Note that "ParserModel" is a subclass of the "nn.Module" class. In PyTorch all ne
12            are a subclass of this "nn.Module".

13        - The "__init__" method is where you define all the layers and their respective par
14            (embedding layers, linear layers, dropout layers, etc.).

15        - "__init__" gets automatically called when you create a new instance of your class
16            when you write "m = ParserModel()".

17        - Other methods of ParserModel can access variables that have "self." prefix. Thus,
18            you should add the "self." prefix layers, values, etc. that you want to utilize
19            in other ParserModel methods.

20        - For further documentation on "nn.Module" please see https://pytorch.org/docs/stab

21    """

22    def __init__(self, embeddings, n_features=36,

23                hidden_size=200, n_classes=3, dropout_prob=0.5):

24        """ Initialize the parser model.

25

26        @param embeddings (Tensor): word embeddings (num_words, embedding_size)

27        @param n_features (int): number of input features

28        @param hidden_size (int): number of hidden units

29        @param n_classes (int): number of output classes

30        @param dropout_prob (float): dropout probability
```

```
31      """
32
33      super(ParserModel, self).__init__()
34
35      self.n_features = n_features
36
37      self.n_classes = n_classes
38
39      self.dropout_prob = dropout_prob
40
41      self.embed_size = embeddings.shape[1]
42
43      self.hidden_size = hidden_size
44
45      self.pretrained_embeddings = nn.Embedding(embeddings.shape[0], self.embed_size)
46
47      self.pretrained_embeddings.weight = nn.Parameter(torch.tensor(embeddings))
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
```

```
55     ### - After you create a linear layer you can access the weight
56     ### matrix via:
57     ### linear_layer.weight
58     ###
59     ### Please see the following docs for support:
60     ### Linear Layer: https://pytorch.org/docs/stable/nn.html#torch.nn.Linear
61     ### Xavier Init: https://pytorch.org/docs/stable/nn.html#torch.nn.init.xavier_u
62     ### Dropout: https://pytorch.org/docs/stable/nn.html#torch.nn.Dropout
63
64     self.embed_to_hidden = nn.Linear(self.embed_size*self.n_features,self.hidden_size)
65     nn.init.xavier_uniform_(self.embed_to_hidden.weight)
66     self.dropout = nn.Dropout(self.dropout_prob)
67     self.hidden_to_logits = nn.Linear(hidden_size,self.n_classes)
68     nn.init.xavier_uniform_(self.hidden_to_logits.weight)
69
70     ### END YOUR CODE
71
72     def embedding_lookup(self, t):
73         """ Utilize `self.pretrained_embeddings` to map input `t` from input tokens (integers)
74             to embedding vectors.
75
76             PyTorch Notes:
77
78             - `self.pretrained_embeddings` is a torch.nn.Embedding object that we defined in the
79             constructor.
80             - Here `t` is a tensor where each row represents a list of features. Each feature is an integer
81             representing a word in the vocabulary.
82             - We use the embedding lookup operation to map each word to its corresponding embedding vector.
83             - The output is a tensor of shape (batch_size, hidden_size) where each row represents the hidden
84             state for a given input sequence.
85             - We use the hidden state to calculate the logits for the next time step.
86             - We use the logits to calculate the loss for the current time step.
87             - We use the loss to calculate the gradient of the loss with respect to the parameters of the
88             model.
89             - We use the gradient to update the parameters of the model using the Adam optimizer.
90             - We repeat the above steps for a fixed number of epochs.
91             - We return the final loss and the parameters of the model.
92         """
93         # Embedding lookup
94         embed_to_hidden = self.embed_to_hidden
95         hidden_to_logits = self.hidden_to_logits
96         dropout = self.dropout
97         # Calculate the hidden state
98         hidden_state = embed_to_hidden(t)
99         hidden_state = nn.Dropout(dropout)(hidden_state)
100        # Calculate the logits
101        logits = hidden_to_logits(hidden_state)
102        # Calculate the loss
103        loss = nn.CrossEntropyLoss()(logits, target)
104        # Calculate the gradient
105        loss.backward()
106        # Update the parameters
107        optimizer.step()
108        # Return the final loss and parameters
109        return loss, model.parameters()
```

```
79         - In PyTorch the Embedding object, e.g. `self.pretrained_embeddings`, allow
80           go from an index to embedding. Please see the documentation (https://py
81           to learn how to use `self.pretrained_embeddings` to extract the embeddi
82
83     @param t (Tensor): input tensor of tokens (batch_size, n_features)
84
85     @return x (Tensor): tensor of embeddings for words represented in t
86
87           (batch_size, n_features * embed_size)
88
89     """
90     ### YOUR CODE HERE (~1-3 Lines)
91
92     ### TODO:
93
94     ### 1) Use `self.pretrained_embeddings` to lookup the embeddings for the input
95
96     ### 2) After you apply the embedding lookup, you will have a tensor shape (batch_size, n_features * embed_size)
97
98     ### Use the tensor `view` method to reshape the embeddings tensor to (batch_size, embed_size)
99
100    ### Note: In order to get batch_size, you may need use the tensor .size() function:
101
102    ### https://pytorch.org/docs/stable/tensors.html#torch.Tensor.size
103
104    ### Please see the following docs for support:
105
106    ### Embedding Layer: https://pytorch.org/docs/stable/nn.html#torch.nn.Embedding
107
108    ### View: https://pytorch.org/docs/stable/tensors.html#torch.Tensor.view
109
110    batch_size = t.shape[0]
111
112    x = self.pretrained_embeddings(t).view(batch_size, -1)
113
114    ### END YOUR CODE
```

```
103         return x

104
105
106     def forward(self, t):
107         """ Run the model forward.
108
109         Note that we will not apply the softmax function here because it is included in
110
111         PyTorch Notes:
112
113         - Every nn.Module object (PyTorch model) has a `forward` function.
114
115         - When you apply your nn.Module to an input tensor `t` this function is app
116
117         For example, if you created an instance of your ParserModel and applied
118
119         the `forward` function would called on `t` and the result would be stor
120
121         model = ParserModel()
122
123         output = model(t) # this calls the forward function
124
125         - For more details checkout: https://pytorch.org/docs/stable/nn.html#torch.
126
127
128     @param t (Tensor): input tensor of tokens (batch_size, n_features)
129
130
131
132     @return logits (Tensor): tensor of predictions (output after applying the layers of
133
134         without applying softmax (batch_size, n_classes)
135
136     """
137
138     ### YOUR CODE HERE (~3-5 lines)
139
140     ### TODO:
```

```

127      ###      1) Apply `self.embedding_lookup` to `t` to get the embeddings

128      ###      2) Apply `embed_to_hidden` linear layer to the embeddings

129      ###      3) Apply relu non-linearity to the output of step 2 to get the hidden units

130      ###      4) Apply dropout layer to the output of step 3.

131      ###      5) Apply `hidden_to_logits` layer to the output of step 4 to get the logits

132      ###

133      ### Note: We do not apply the softmax to the logits here, because

134      ### the loss function (torch.nn.CrossEntropyLoss) applies it more efficiently.

135      ###

136      ### Please see the following docs for support:

137      ###      ReLU: https://pytorch.org/docs/stable/nn.html?highlight=relu#torch.nn.funct

138      features_embeddings = self.embedding_lookup(t)

139      o = F.relu(self.embed_to_hidden(features_embeddings))


140      o = self.dropout(o)

141      logits = self.hidden_to_logits(o)

142      ### END YOUR CODE

143      return logits

```



```

1  def train(parser, train_data, dev_data, output_path, batch_size=1024, n_epochs=10, lr=0.0005

2      """ Train the neural dependency parser.

3

4      @param parser (Parser): Neural Dependency Parser

5      @param train_data ():

6      @param dev_data ():

```



```

7      @param output_path (str): Path to which model weights and results are written.

8      @param batch_size (int): Number of examples in a single batch

9      @param n_epochs (int): Number of training epochs

10     @param lr (float): Learning rate

11     """

12     best_dev_UAS = 0

13

14

15     ### YOUR CODE HERE (~2-7 lines)

16     ### TODO:

17     ###     1) Construct Adam Optimizer in variable `optimizer`

18     ###     2) Construct the Cross Entropy Loss Function in variable `loss_func`

19     ###

20     ### Hint: Use `parser.model.parameters()` to pass optimizer

21     ###     necessary parameters to tune.

22     ### Please see the following docs for support:

23     ###     Adam Optimizer: https://pytorch.org/docs/stable/optim.html

24     ###     Cross Entropy Loss: https://pytorch.org/docs/stable/nn.html#crossentropyloss

25     optimizer = optim.Adam(parser.model.parameters(),lr=lr)

26     loss_func = nn.CrossEntropyLoss()

27

28     ### END YOUR CODE

29

30     for epoch in range(n_epochs):

```

```

30     for epoch in range(n_epochs):

31         print("Epoch {:} out of {:}".format(epoch + 1, n_epochs))

32         dev_UAS = train_for_epoch(parser, train_data, dev_data, optimizer, loss_func, batch_

33         if dev_UAS > best_dev_UAS:

34             best_dev_UAS = dev_UAS

35             print("New best dev UAS! Saving model.")

36             torch.save(parser.model.state_dict(), output_path)

37         print("")

38

39

40 def train_for_epoch(parser, train_data, dev_data, optimizer, loss_func, batch_size):

41     """ Train the neural dependency parser for single epoch.

42

43     Note: In PyTorch we can signify train versus test and automatically have

44     the Dropout Layer applied and removed, accordingly, by specifying

45     whether we are training, `model.train()`, or evaluating, `model.eval()`

46

47     @param parser (Parser): Neural Dependency Parser

48     @param train_data ():

49     @param dev_data ():

50     @param optimizer (nn.Optimizer): Adam Optimizer

51     @param loss_func (nn.CrossEntropyLoss): Cross Entropy Loss Function

52     @param batch_size (int): batch size

53     @param lr (float): learning rate

```

```

55     @return dev_UAS (float): Unlabeled Attachment Score (UAS) for dev data
56
57     parser.model.train() # Places model in "train" mode, i.e. apply dropout layer
58
59     n_minibatches = math.ceil(len(train_data) / batch_size)
60
61     loss_meter = AverageMeter()
62
63     dev_UAS, _ = parser.parse(dev_data)
64
65
66     with tqdm(total=(n_minibatches)) as prog:
67
68         for i, (train_x, train_y) in enumerate(minibatches(train_data, batch_size)):
69
70             optimizer.zero_grad() # remove any baggage in the optimizer
71
72             loss = 0. # store loss for this batch here
73
74             train_x = torch.from_numpy(train_x).long()
75
76             train_y = torch.from_numpy(train_y.nonzero()[1]).long()
77
78
79             ### YOUR CODE HERE (~5-10 lines)
80
81             ### TODO:
82
83             ###      1) Run train_x forward through model to produce `logits`
84
85             ###      2) Use the `loss_func` parameter to apply the PyTorch CrossEntropyLoss
86
87             ###          This will take `logits` and `train_y` as inputs. It will output the
88
89             ###          between softmax(`logits`) and `train_y`. Remember that softmax(`logi
90
91             ###          are the predictions ( $y^{\wedge}$  from the PDF).
92
93             ###      3) Backprop losses
94
95             ###      4) Take step with the optimizer
96
97
98             ### Please see the following docs for support:

```

```
78     ### PLEASE SEE THE FOLLOWING DOCS FOR SUPPORT:
79     ###     Optimizer Step: https://pytorch.org/docs/stable/optim.html#optimizer-ste
80     logits = parser.model(train_x)
81     loss = loss_func(logits,train_y)
82     loss.backward()
83     optimizer.step()
84     ### END YOUR CODE
85     prog.update(1)
86     loss_meter.update(loss.item())
87
88     print ("Average Train Loss: {}".format(loss_meter.avg))
89
90     print("Evaluating on dev set",)
91     parser.model.eval() # Places model in "eval" mode, i.e. don't apply dropout layer
92     dev_UAS, _ = parser.parse(dev_data)
93     print("- dev UAS: {:.2f}".format(dev_UAS * 100.0))
94     return dev_UAS
```

[# Deep Learning](#) [# NLP](#)

◀ DP-Neural Network

NLP-Seq2Seq ▶

