

Appendix to Bayesuvius Chapter about SentenceAx

Robert R. Tucci
tucci@ar-tiste.com

February 14, 2024

The SentenceAx (Sax) software (at github repo Ref.[4]) is a complete re-write of the Openie6 (O6) software (at github repo Ref.[1]).

The O6 software is described by its creators in the paper Ref.[2], which we will henceforth refer to as the O6 paper.

Before reading this appendix, you should read the document entitled "Sentence Splitting with SentenceAx" (Ref.[2]) that is a chapter excerpt from my book Bayesuvius (Ref.[3]). I will henceforth refer to that chapter as the Sax chapter. The purpose of this Appendix is to record details about Sax that were deemed too fine-grained or ephemeral to be included in the Sax chapter.

1 PyTorch code for calculating Penalty Loss

The Sax chapter gives all the equations associated with Penalty Loss. But how does one code them with PyTorch? The O6 software does it masterfully. Here is the pertinent code snippet from Sax. It comes directly from the O6 software, modulus changes in notation.

```
1 @staticmethod
2 def sax_penalty_loss(x_d,
3                      llll_word_scoreT,
4                      con_to_weight):
5     """
6     similar to Openie6.model.constrained_loss()
7
8     This method is called inside sax_batch_loss(). It returns the
9     penalty loss.
10
11     Parameters
12     _____
13     x_d: OrderedDict
14     llll_word_scoreT: torch.Tensor
15     con_to_weight: dict[str, float]
```

```

16
17 Returns
18
19 float
20     penalty_loss
21
22 """
23 batch_size, num_depths, num_words, icode_dim = \
24     llll_word_scoreT.shape
25 penalty_loss = 0
26 llll_index = x_d["ll_osen_t_verb_loc"].\
27     unsqueeze(1).unsqueeze(3).repeat(1, num_depths, 1, icode_dim)
28 llll_verb_trust = torch.gather(
29     input=llll_word_scoreT,
30     dim=2,
31     index=llll_index)
32 lll_verb_rel_trust = llll_verb_trust[:, :, :, 2]
33 # (batch_size, depth, num_words)
34 lll_bool = (x_d["ll_osen_t_verb_loc"] != 0).unsqueeze(1).float()
35
36 lll_verb_rel_trust = lll_verb_rel_trust * lll_bool
37 # every head-verb must be included in a relation
38 if 'hvc' in con_to_weight:
39     ll_column_loss = \
40         torch.abs(1 - torch.sum(lll_verb_rel_trust, dim=1))
41     ll_column_loss = \
42         ll_column_loss[x_d["ll_osen_t_verb_loc"] != 0]
43     penalty_loss += con_to_weight['hvc'] * ll_column_loss.sum()
44
45 # extractions must have at least k-relations with
46 # a head verb in them
47 if 'hvr' in con_to_weight:
48     l_a = x_d["ll_osen_t_verb_bool"].sum(dim=1).float()
49     l_b = torch.max(lll_verb_rel_trust, dim=2)[0].sum(dim=1)
50     row_rel_loss = F.relu(l_a - l_b)
51     penalty_loss += con_to_weight['hvr'] * row_rel_loss.sum()
52
53 # one relation cannot contain more than one head verb
54 if 'hve' in con_to_weight:
55     ll_ex_loss = \
56         F.relu(torch.sum(lll_verb_rel_trust, dim=2) - 1)
57     penalty_loss += con_to_weight['hve'] * ll_ex_loss.sum()
58
59 if 'posm' in con_to_weight:
60     llll_index = \
61         x_d["ll_osen_t_pos_loc"].unsqueeze(1).unsqueeze(3).\
62         repeat(1, num_depths, 1, icode_dim)
63     llll_pred_trust = torch.gather(
64         input=llll_word_scoreT,
65         dim=2,
66         index=llll_index)

```

```

67     ll_pos_not_none_trust = \
68         torch.max(lll_pred_trust[:, :, :, 1:], dim=-1)[0]
69     ll_column_loss = \
70         (1 - torch.max(lll_pos_not_none_trust, dim=1)[0]) * \
71         (x_d["ll_osen_pos_loc"] != 0).float()
72     penalty_loss += con_to_weight['posm'] * ll_column_loss.sum()
73
74     return penalty_loss

```

2 Sax bnet

The Sax chapter gives a drawing of the Sax bnet, and a list of its structural equations. Both were produced with the texnn tool (Ref.[5])

In this section, we provide evidence that Sax does indeed implement that bnet correctly.

This section has 3 parts.

1. texnn output
2. Sax code that implements the bnet.
3. Excerpt of print-out to console produced when I run the jupyter notebook for training the warmup NN for task=ex. (The jupyter notebooks for warmup training have verbose=True. Those for non-warmup training have verbose=False).

2.1 texnn output

$$\begin{aligned}
 \underline{a}^{[86]} &: \text{ll_greedy_ilabel} \\
 \underline{B}^{[121],[768]} &: \text{lll_hidstate} \\
 \underline{d}^{[121],[768]} &: \text{lll_hidstate} \\
 \underline{E}^{[86],[768]} &: \text{lll_pred_code} \\
 \underline{G}^{[86],[768]} &: \text{lll_word_hidstate} \\
 \underline{I}^{[121],[768]} &: \text{lll_hidstate} \\
 \underline{L}^{[86],[6]} &: \text{lll_word_score} \\
 \underline{M}^{[86],[300]} &: \text{lll_word_hidstate} \\
 \underline{S}^{[86],[768]} &: \text{lll_word_hidstate} \\
 \underline{X}^{[86],[6]} &: \text{lll_word_score} \\
 \underline{a}^{[86]} &= \operatorname{argmax}(X^{[86],[6]}; \dim = -1) \\
 &: \text{ll_greedy_ilabel}
 \end{aligned} \tag{1a}$$

$$\begin{aligned}
 \underline{B}^{[121],[768]} &= \text{BERT}() \\
 &: \text{lll_hidstate}
 \end{aligned} \tag{1b}$$

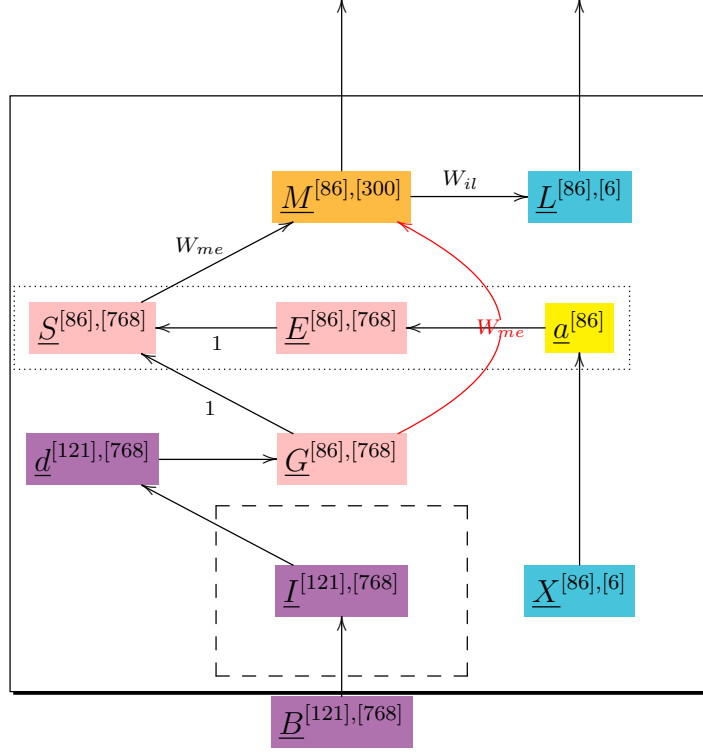


Figure 1: Sax bnet. 2 copies of dashed box are connected in series. 5 copies (5 depths) of plain box are connected in series. However, in the first of those 5 plain box copies, the dotted box is omitted and node \underline{G} feeds directly into node \underline{M} (indicated by red arrow). We display the tensor shape superscripts in the PyTorch L2R order. All tensor shape superscripts have been simplified by omitting a $[s_{ba}]$ from their left side, where $s_{ba} = 24$ is the batch size.

$$\begin{aligned} d^{[121],[768]} &= \text{dropout}(I^{[121],[768]}) \\ &: \text{lll_hidstate} \end{aligned} \tag{1c}$$

$$\begin{aligned} E^{[86],[768]} &= \text{embedding}(a^{[86]}) \\ &: \text{lll_pred_code} \end{aligned} \tag{1d}$$

$$\begin{aligned} G^{[86],[768]} &= \text{gather}(d^{[121],[768]}; \text{dim} = -2) \\ &: \text{lll_word_hidstate} \end{aligned} \tag{1e}$$

$$I^{[121],[768]} = [B^{[121],[768]} \mathbb{1}(\text{depth} = 0) + M^{[86],[300]} \mathbb{1}(\text{depth} > 0)]$$

(1f)

: lll_hidstate

$$L^{[86],[6]} = M^{[86],[300]} W_{il}^{[300],[6]}$$

(1g)

: lll_word_score

$$M^{[86],[300]} = [G^{[86],[768]} \mathbb{1}(\text{depth} = 0) + S^{[86],[768]} \mathbb{1}(\text{depth} > 0)] W_{me}^{[768],[300]}$$

(1h)

: lll_word_hidstate

$$S^{[86],[768]} = E^{[86],[768]} + G^{[86],[768]}$$

(1i)

: lll_word_hidstate

$$X^{[86],[6]} = L^{[86],[6]} \mathbb{1}(\text{depth} > 0)$$

(1j)

: lll_word_score

2.2 Sax code

```

1 def sax_get_lll_word_score(self, x_d, ttt, verbose=False):
2     """
3
4     This method is used inside self.forward() and is the heart of that
5     method. It contains a while loop over depths that drives a batch
6     through the layers of the model and returns 'lll_word_score'.
7     Setting 'verbose' to True prints out a detailed trail of what occurs
8     in this method. The following example was obtained from such a
9     verbose trail.
10
11     Assume:
12     batch_size= 24,
13     hidden_size= 768,
14     NUM_ILABELS= 6,
15     MERGE_DIM= 300
16     2 iterative layers and 5 depths.
17
18     lll_word_score is the output of the last ilabelling_layer for each
19     depth
20
21     llll_word_score is a list of lll_word_score
22
23     len(llll_word_score)= 5 = num_depths

```

```

24
25 Note that llll_word_scoreT = Ten(llll_word_score)
26
27 Parameters
28 -----
29 x_d: OrderedDict
30 ttt: str
31 verbose: bool
32
33 Returns
34 -----
35 list[torch.Tensor]
36     llll_word_score
37
38 """
39 # lll_label is similar to Openie6.labels
40 # first (outer) list over batch/sample of events
41 # second list over extractions
42 # third (inner) list over number of labels in a line
43 # after padding and adding the 3 unused tokens
44
45 # batch_size, num_depths, num_words = y_d["lll_ilabel"].shape
46 # sometimes num_depths will exceed max.
47 # This doesn't happen when training, because
48 # num_depths is specified when training.
49 num_depths = get_num_depths(self.params.task)
50
51 # 'loss_fun' is not used in this function anymore
52 # loss_fun, lstm_loss = 0, 0
53
54 # batch_text = " ".join(redoL(meta_d["l_orig_sent"]))
55 # base_model_input = \
56 #     torch.Tensor(self.auto_tokenizer.encode(batch_text))
57 hstate_count = Counter(verbose, "lll_hidstate")
58 word_hstate_count = Counter(verbose, "lll_word_hidstate")
59 lll_hidstate, _ = self.base_model(x_d["ll_osent_icode"])
60 hstate_count.new_one(reset=True)
61 comment(
62     verbose,
63     prefix="after base_model",
64     params_d={
65         "ll_osent_icode.shape": x_d["ll_osent_icode"].shape,
66         "lll_hidstate.shape": lll_hidstate.shape})
67 lll_word_score = Ten([0]) # this statement is unnecessary
68 llll_word_score = [] # ~ Openie6.all_depth_scores
69 depth = 0
70 # loop over depths
71 while True:
72     for ilay, layer in enumerate(self.iterative_transformer):
73         comment(verbose,
74             prefix="***** Starting iterative layer",

```

```

75         params_d={"ilay": ilay})
76     # layer(l1l_hidstate)[0] returns a copy
77     # of the tensor l1l_hidstate after transforming it
78     # in some way. [0] chooses first component
79     comment(
80         verbose,
81         prefix="Before iterative layer",
82         params_d={
83             "ilay": ilay,
84             "depth": depth,
85             "l1l_hidstate.shape": l1l_hidstate.shape})
86     l1l_hidstate = layer(l1l_hidstate)[0]
87     hstate_count.new_one()
88     comment(
89         verbose,
90         prefix="After iterative layer",
91         params_d={
92             "ilay": ilay,
93             "depth": depth,
94             "l1l_hidstate.shape": l1l_hidstate.shape})
95     comment(verbose,
96         prefix="Before dropout",
97         params_d={
98             "depth": depth,
99             "l1l_hidstate.shape": l1l_hidstate.shape})
100    l1l_hidstate = self.dropout_fun(l1l_hidstate)
101    hstate_count.new_one()
102    comment(verbose,
103        prefix="After dropout",
104        params_d={
105            "depth": depth,
106            "l1l_hidstate.shape": l1l_hidstate.shape})
107    l1l_loc = x_d["l1l_osent_wstart_loc"].unsqueeze(2). \
108        repeat(1, 1, l1l_hidstate.shape[2])
109    l1l_word_hidstate = torch.gather(
110        input=l1l_hidstate,
111        dim=1,
112        index=l1l_loc)
113    comment(
114        verbose,
115        prefix="Gather's 2 inputs, then output",
116        params_d={
117            "l1l_hidstate.shape": l1l_hidstate.shape,
118            "l1l_loc.shape": l1l_loc.shape,
119            "l1l_word_hidstate.shape": l1l_word_hidstate.shape})
120    word_hstate_count.new_one(reset=True)
121    if depth != 0:
122        comment(
123            verbose,
124            prefix="before argmax",
125            params_d={"l1l_word_score.shape": l1l_word_score.shape})

```

```

126     ll_greedy_ilabel = torch.argmax(lll_word_score, dim=-1)
127     comment(
128         verbose,
129         prefix="after argmax",
130         params_d={"ll_greedy_ilabel.shape":
131                 ll_greedy_ilabel.shape})
132     # not an integer code/embedding
133     comment(
134         verbose,
135         prefix="before embedding",
136         params_d={"ll_greedy_ilabel.shape":
137                 ll_greedy_ilabel.shape})
138     lll_pred_code = self.embedding(ll_greedy_ilabel)
139     comment(
140         verbose,
141         prefix="after embedding",
142         params_d={"lll_word_hidstate.state":
143                 lll_word_hidstate.shape})
144     lll_word_hidstate += lll_pred_code
145     word_hstate_count.new_one()
146     comment(
147         verbose,
148         prefix="just summed two signals with this shape",
149         params_d={
150             "depth": depth,
151             "lll_word_hidstate.shape": lll_word_hidstate.shape})
152     comment(verbose,
153             prefix="Before merge layer",
154             params_d={
155                 "depth": depth,
156                 "lll_word_hidstate.shape": lll_word_hidstate.shape})
157     lll_word_hidstate = self.merge_layer(lll_word_hidstate)
158     comment(
159         verbose,
160         prefix="After merge layer",
161         params_d={
162             "depth": depth,
163             "lll_word_hidstate.shape": lll_word_hidstate.shape})
164     comment(
165         verbose,
166         prefix="Before ilabelling",
167         params_d={
168             "depth": depth,
169             "lll_word_hidstate.shape": lll_word_hidstate.shape})
170     lll_word_score = self.ilabelling_layer(lll_word_hidstate)
171     comment(
172         verbose,
173         prefix="After ilabelling",
174         params_d={
175             "depth": depth,
176             "lll_word_score.shape": lll_word_score.shape})

```



```

177     llll_word_score.append(lll_word_score)
178
179     depth += 1
180     if depth >= num_depths:
181         break
182
183     if ttt != 'train':
184         ll_pred_ilabel = torch.max(lll_word_score, dim=2)[1]
185         valid_extraction = False
186         for l_pred_ilabel in ll_pred_ilabel:
187             if is_valid_label_list(
188                 l_pred_ilabel, self.params.task, "ilabels"):
189                 valid_extraction = True
190                 break
191             if not valid_extraction:
192                 break
193     comment(
194         verbose,
195         params_d={
196             "len(llll_word_score)": len(llll_word_score),
197             "llll_word_score[0].shape": llll_word_score[0].shape})
198     return llll_word_score

```

2.3 jupyter notebook print-out

```

1  """
2  Entering Model.training_step method, batch_idx=0
3  'lll_hidstate' count changed: 0->1
4  after base_model
5      ll_osent_icode.shape=torch.Size([4, 121])
6      lll_hidstate.shape=torch.Size([4, 121, 768])
7  ***** Starting iterative layer
8      ilay=0
9  Before iterative layer
10     ilay=0
11     depth=0
12     lll_hidstate.shape=torch.Size([4, 121, 768])
13 'lll_hidstate' count changed: 1->2
14 After iterative layer
15     ilay=0
16     depth=0
17     lll_hidstate.shape=torch.Size([4, 121, 768])
18 ***** Starting iterative layer
19     ilay=1
20 Before iterative layer
21     ilay=1
22     depth=0
23     lll_hidstate.shape=torch.Size([4, 121, 768])
24 'lll_hidstate' count changed: 2->3
25 After iterative layer

```

```

26     ilay=1
27     depth=0
28     lll_hidstate.shape=torch.Size([4, 121, 768])
29 Before dropout
30     depth=0
31     lll_hidstate.shape=torch.Size([4, 121, 768])
32 'lll_hidstate' count changed: 3->4
33 After dropout
34     depth=0
35     lll_hidstate.shape=torch.Size([4, 121, 768])
36 Gather's 2 inputs, then output
37     lll_hidstate.shape=torch.Size([4, 121, 768])
38     lll_loc.shape=torch.Size([4, 86, 768])
39     lll_word_hidstate.shape=torch.Size([4, 86, 768])
40 'lll_word_hidstate' count changed: 0->1
41 Before merge layer
42     depth=0
43     lll_word_hidstate.shape=torch.Size([4, 86, 768])
44 After merge layer
45     depth=0
46     lll_word_hidstate.shape=torch.Size([4, 86, 300])
47 Before ilabelling
48     depth=0
49     lll_word_hidstate.shape=torch.Size([4, 86, 300])
50 After ilabelling
51     depth=0
52     lll_word_score.shape=torch.Size([4, 86, 6])
53 ***** Starting iterative layer
54     ilay=0
55 Before iterative layer
56     ilay=0
57     depth=1
58     lll_hidstate.shape=torch.Size([4, 121, 768])
59 'lll_hidstate' count changed: 4->5
60 After iterative layer
61     ilay=0
62     depth=1
63     lll_hidstate.shape=torch.Size([4, 121, 768])
64 ***** Starting iterative layer
65     ilay=1
66 Before iterative layer
67     ilay=1
68     depth=1
69     lll_hidstate.shape=torch.Size([4, 121, 768])
70 'lll_hidstate' count changed: 5->6
71 After iterative layer
72     ilay=1
73     depth=1
74     lll_hidstate.shape=torch.Size([4, 121, 768])
75 Before dropout
76     depth=1

```

```

77     lll_hidstate.shape=torch.Size([4, 121, 768])
78 'lll_hidstate' count changed: 6->7
79 After dropout
80     depth=1
81     lll_hidstate.shape=torch.Size([4, 121, 768])
82 Gather's 2 inputs, then output
83     lll_hidstate.shape=torch.Size([4, 121, 768])
84     lll_loc.shape=torch.Size([4, 86, 768])
85     lll_word_hidstate.shape=torch.Size([4, 86, 768])
86 'lll_word_hidstate' count changed: 0->1
87 before argmax
88     lll_word_score.shape=torch.Size([4, 86, 6])
89 after argmax
90     ll_greedy_ilabel.shape=torch.Size([4, 86])
91 before embedding
92     ll_greedy_ilabel.shape=torch.Size([4, 86])
93 after embedding
94     lll_word_hidstate.state=torch.Size([4, 86, 768])
95 'lll_word_hidstate' count changed: 1->2
96 just summed two signals with this shape
97     depth=1
98     lll_word_hidstate.shape=torch.Size([4, 86, 768])
99 Before merge layer
100     depth=1
101     lll_word_hidstate.shape=torch.Size([4, 86, 768])
102 After merge layer
103     depth=1
104     lll_word_hidstate.shape=torch.Size([4, 86, 300])
105 Before ilabelling
106     depth=1
107     lll_word_hidstate.shape=torch.Size([4, 86, 300])
108 After ilabelling
109     depth=1
110     lll_word_score.shape=torch.Size([4, 86, 6])
111 """

```

References

- [1] Data Analytics and IIT Delhi Intelligence Research (DAIR) Group. Openie6. <https://github.com/dair-iitd/openie6>.
- [2] Keshav Kolluru, Vaibhav Adlakha, Samarth Aggarwal, Mausam, and Soumen Chakrabarti. Openie6: Iterative grid labeling and coordination analysis for open information extraction. <https://arxiv.org/abs/2010.03147>.
- [3] Robert R. Tucci. Bayesuvius (book). <https://github.com/rrtucci/Bayesuvius/raw/master/main.pdf>.
- [4] Robert R. Tucci. SentenceAx. <https://github.com/rrtucci/SentenceAx>.

[5] Robert R. Tucci. texnn. <https://github.com/rrtucci/texnn>.