# SentenceAx Appendix

Robert R. Tucci
tucci@ar-tiste.com

February 2, 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]). Sax is 99% identical algorithmically to O6 but it's packaged in what we hope is a friendlier form. The O6 software is described by its creators in the paper Ref.[2], which we will henceforth refer to as the O6 paper.

The original and primary documentation for sax is Ref.[2], which we will henceforth refer to as the O6 paper.

The main documentation for sax is the chapter entitled "Sentence Splitting with SentenceAx" in my text book Bayesuvius (Ref.[3]). The purpose of this Appendix is to record details about sax that were deemed too technical or ephemeral to be included in that chapter.

## 1 PyTorch code for calculating Penalty Loss

The sax chapter gives all the equations associated with Penalty Loss. But how to 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.

```
@staticmethod
def sax_penalty_loss(x_d,
                     llll_word_scoreT,
                     con_to_weight):
    """
    similar to Openie6.model.constrained_loss()

    This method is called inside sax_batch_loss(). It returns the
    penalty loss.

    Parameters
    ----------
    x_d: OrderedDict
    llll_word_scoreT: torch.Tensor
    con_to_weight: dict[str, float]
```

```

    Returns
    -------
    float
        penalty_loss

    """
    batch_size, num_depths, num_words, icode_dim = \
        llll_word_scoreT.shape
    penalty_loss = 0
    llll_index = x_d["ll_osent_verb_loc"].\
        unsqueeze(1).unsqueeze(3).repeat(1, num_depths, 1, icode_dim)
    llll_verb_trust = torch.gather(
        input=llll_word_scoreT,
        dim=2,
        index=llll_index)
    lll_verb_rel_trust = llll_verb_trust[:, :, :, 2]
    # (batch_size, depth, num_words)
    lll_bool = (x_d["ll_osent_verb_loc"] != 0).unsqueeze(1).float()

    lll_verb_rel_trust = lll_verb_rel_trust * lll_bool
    # every head-verb must be included in a relation
    if 'hvc' in con_to_weight:
        ll_column_loss = \
            torch.abs(1 - torch.sum(lll_verb_rel_trust, dim=1))
        ll_column_loss = \
            ll_column_loss[x_d["ll_osent_verb_loc"] != 0]
        penalty_loss += con_to_weight['hvc'] * ll_column_loss.sum()

    # extractions must have at least k-relations with
    # a head verb in them
    if 'hvr' in con_to_weight:
        l_a = x_d["ll_osent_verb_bool"].sum(dim=1).float()
        l_b = torch.max(lll_verb_rel_trust, dim=2)[0].sum(dim=1)
        row_rel_loss = F.relu(l_a - l_b)
        penalty_loss += con_to_weight['hvr'] * row_rel_loss.sum()

    # one relation cannot contain more than one head verb
    if 'hve' in con_to_weight:
        ll_ex_loss = \
            F.relu(torch.sum(lll_verb_rel_trust, dim=2) - 1)
        penalty_loss += con_to_weight['hve'] * ll_ex_loss.sum()

    if 'posm' in con_to_weight:
        llll_index = \
            x_d["ll_osent_pos_loc"].unsqueeze(1).unsqueeze(3).\
            repeat(1, num_depths, 1, icode_dim)
        llll_pred_trust = torch.gather(
            input=llll_word_scoreT,
            dim=2,
            index=llll_index)
```

```
        lll_pos_not_none_trust = \
            torch.max(llll_pred_trust[:, :, :, 1:], dim=-1)[0]
        ll_column_loss = \
            (1 - torch.max(lll_pos_not_none_trust, dim=1)[0]) * \
            (x_d["ll_osent_pos_loc"] != 0).float()
        penalty_loss += con_to_weight['posm'] * ll_column_loss.sum()

    return penalty_loss
```

# 2 Original O6 software bnet

This section describes the bnet in the O6 software. We first

## 2.1 texnn print out

$\underline{a}^{[86]}$ : `ll_greedy_ilabel`
$\underline{B}^{[121],[768]}$ : `lll_hidstate`
$\underline{d}^{[121],[768]}$ : `lll_hidstate`
$\underline{E}^{[86],[768]}$ : `lll_pred_code`
$\underline{G}^{[86],[768]}$ : `lll_word_hidstate`
$\underline{L}^{[86],[6]}$ : `lll_word_score`
$\underline{M}^{[86],[300]}$ : `lll_word_hidstate`
$\underline{n}^{[121],[768]}$ : `lll_hidstate`
$\underline{S}^{[86],[768]}$ : `lll_word_hidstate`

$$A^{[121],[D]} = \text{Attention}(Q^{[121],[D]}, K^{[121],[D]}, V^{[121],[D]}) \tag{1a}$$

$$\begin{aligned} a^{[86]} &= \text{argmax}(G^{[86],[768]}; dim = -1) \\ &: \texttt{ll\_greedy\_ilabel} \end{aligned} \tag{1b}$$

$$\begin{aligned} B^{[121],[768]} &= \text{BERT}() \\ &: \texttt{lll\_hidstate} \end{aligned} \tag{1c}$$

$$\begin{aligned} d^{[121],[768]} &= \text{dropout}(n^{[121],[768]}) \\ &: \texttt{lll\_hidstate} \end{aligned} \tag{1d}$$

$$\begin{aligned} E^{[86],[768]} &= \text{embedding}(a^{[86]}) \\ &: \texttt{lll\_pred\_code} \end{aligned} \tag{1e}$$

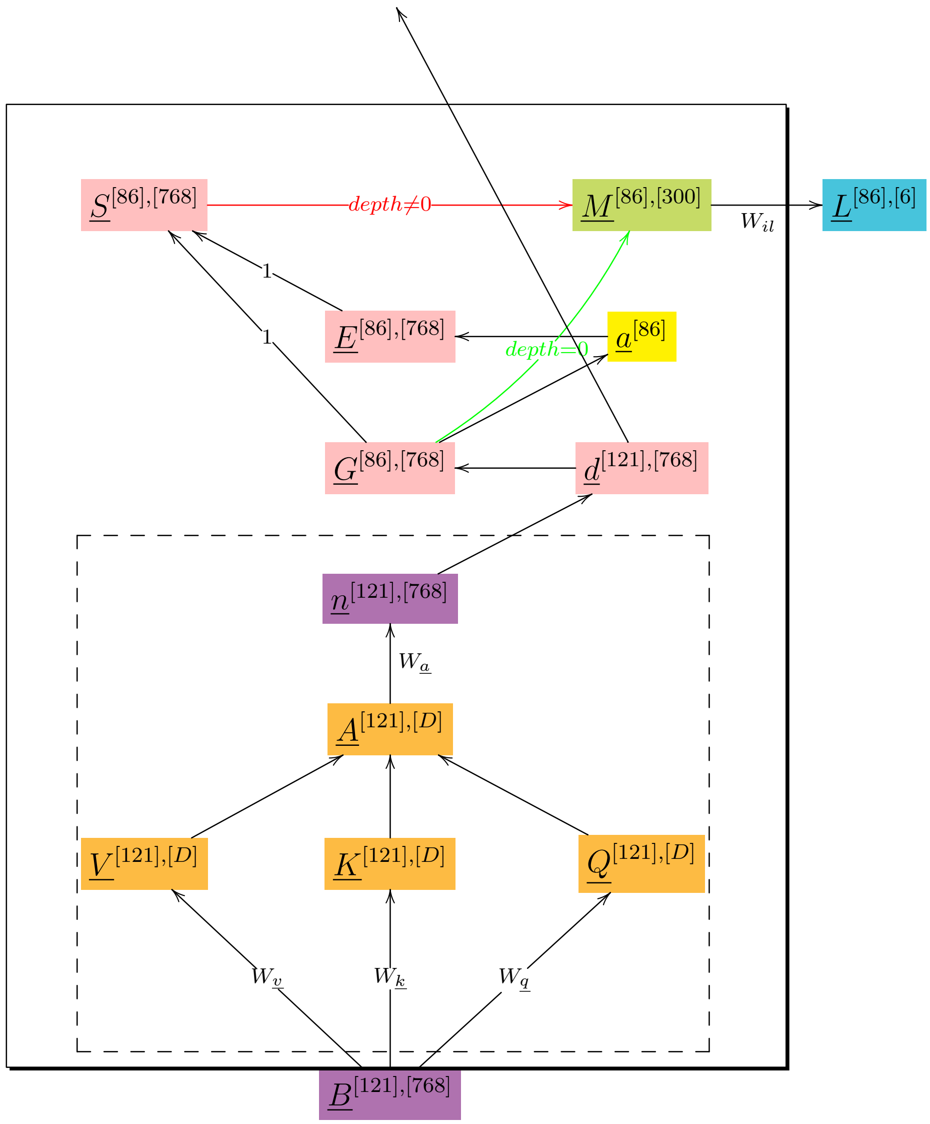


Figure 1: SentenceAx Bayesian network. 2 copies of dashed box are connected in series. 5 copies of plain box are connected in series. We display the tensor shape superscripts in the Linear Algebra R2L order. (PyTorch uses a L2R order instead). All tensor shape superscripts have been simplified by omitting a $[s_{ba}]$, where $s_{ba} = 24$ is the batch size. $D = dn_{\underline{h}}$ where $d = 768$ is the hidden dimension per head, and $n_{\underline{h}} = 12$ is the number of heads.

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

$$K^{[121],[D]} = B^{[121],[768]} W_{\underline{k}}^{[768],[D]} \tag{1g}$$

$$\begin{aligned} L^{[86],[6]} &= M^{[86],[300]} W_{il}^{[300],[6]} \\ &: \texttt{lll\_word\_score} \end{aligned} \tag{1h}$$

$$\begin{aligned} M^{[86],[300]} &= G^{[86],[768]} W_{il}^{[768],[300]} \\ &: \texttt{lll\_word\_hidstate} \end{aligned} \tag{1i}$$

$$\begin{aligned} n^{[121],[768]} &= A^{[121],[D]} W_{\underline{a}}^{[D],[768]} \\ &: \texttt{lll\_hidstate} \end{aligned} \tag{1j}$$

$$Q^{[121],[D]} = B^{[121],[768]} W_{\underline{q}}^{[768],[D]} \tag{1k}$$

$$\begin{aligned} S^{[86],[768]} &= E^{[86],[768]} + G^{[86],[768]} \\ &: \texttt{lll\_word\_hidstate} \end{aligned} \tag{1l}$$

$$V^{[121],[D]} = B^{[121],[768]} W_{\underline{v}}^{[768],[D]} \tag{1m}$$

$\underline{a}^{[86]}$ : `ll_greedy_ilabel`
$\underline{B}^{[121],[768]}$ : `lll_hidstate`
$\underline{d}^{[121],[768]}$ : `lll_hidstate`
$\underline{E}^{[86],[768]}$ : `lll_pred_code`
$\underline{G}^{[86],[768]}$ : `lll_word_hidstate`
$\underline{L}^{[86],[6]}$ : `lll_word_score`
$\underline{M}^{[86],[300]}$ : `lll_word_hidstate`
$\underline{n}^{[121],[768]}$ : `lll_hidstate`
$\underline{S}^{[86],[768]}$ : `lll_word_hidstate`

$$A^{[121],[D]} = \mathrm{Attention}(Q^{[121],[D]}, K^{[121],[D]}, V^{[121],[D]}) \tag{2a}$$

$$\begin{aligned} a^{[86]} &= \mathrm{argmax}(G^{[86],[768]}; dim = -1) \\ &: \texttt{ll\_greedy\_ilabel} \end{aligned} \tag{2b}$$

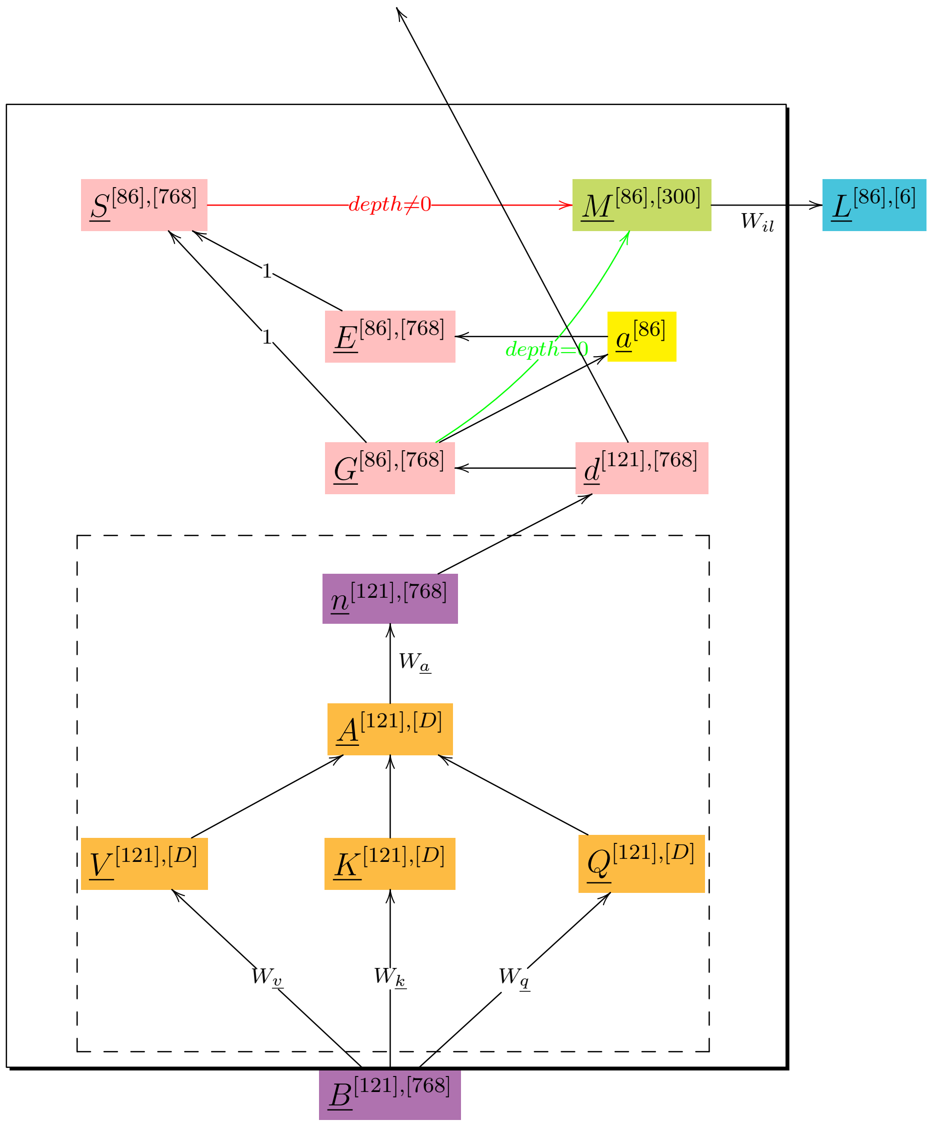


Figure 2: SentenceAx Bayesian network. 2 copies of dashed box are connected in series. 5 copies of plain box are connected in series. We display the tensor shape superscripts in the Linear Algebra R2L order. (PyTorch uses a L2R order instead). All tensor shape superscripts have been simplified by omitting a $[s_{ba}]$, where $s_{ba} = 24$ is the batch size. $D = dn_{\underline{h}}$ where $d = 768$ is the hidden dimension per head, and $n_{\underline{h}} = 12$ is the number of heads.

$$\begin{aligned} B^{[121],[768]} &= \text{BERT}() \\ &: \texttt{lll\_hidstate} \end{aligned} \tag{2c}$$

$$d^{[121],[768]} = \text{dropout}(n^{[121],[768]}) \\ : \texttt{lll\_hidstate} \tag{2d}$$

$$E^{[86],[768]} = \text{embedding}(a^{[86]}) \\ : \texttt{lll\_pred\_code} \tag{2e}$$

$$G^{[86],[768]} = \text{gather}(d^{[121],[768]}; dim = -2) \\ : \texttt{lll\_word\_hidstate} \tag{2f}$$

$$K^{[121],[D]} = B^{[121],[768]} W_{\underline{k}}^{[768],[D]} \tag{2g}$$

$$L^{[86],[6]} = M^{[86],[300]} W_{il}^{[300],[6]} \\ : \texttt{lll\_word\_score} \tag{2h}$$

$$M^{[86],[300]} = G^{[86],[768]} W_{il}^{[768],[300]} \\ : \texttt{lll\_word\_hidstate} \tag{2i}$$

$$n^{[121],[768]} = A^{[121],[D]} W_{\underline{a}}^{[D],[768]} \\ : \texttt{lll\_hidstate} \tag{2j}$$

$$Q^{[121],[D]} = B^{[121],[768]} W_{\underline{q}}^{[768],[D]} \tag{2k}$$

$$S^{[86],[768]} = E^{[86],[768]} + G^{[86],[768]} \\ : \texttt{lll\_word\_hidstate} \tag{2l}$$

$$V^{[121],[D]} = B^{[121],[768]} W_{\underline{v}}^{[768],[D]} \tag{2m}$$

## 2.2 statements printed to console while running the warmup model

```
"""
after starting_model, lll_hidstate.shape torch.Size([4, 121, 768])
*********** Starting iterative layer
    ilay=0
Before iterative layer
    ilay=0
    depth=0
    lll_hidstate.shape=torch.Size([4, 121, 768])
After iterative layer
    ilay=0
    depth=0
    lll_hidstate.shape=torch.Size([4, 121, 768])
*********** Starting iterative layer
    ilay=1
Before iterative layer
    ilay=1
    depth=0
    lll_hidstate.shape=torch.Size([4, 121, 768])
After iterative layer
    ilay=1
    depth=0
    lll_hidstate.shape=torch.Size([4, 121, 768])
Before dropout
    depth=0
    lll_hidstate.shape=torch.Size([4, 121, 768])
After dropout
    depth=0
    lll_hidstate.shape=torch.Size([4, 121, 768])
gather 2 inputs, then output
    lll_hidstate.shape=torch.Size([4, 121, 768])
    lll_loc.shape=torch.Size([4, 86, 768])
    lll_word_hidstate.shape=torch.Size([4, 86, 768])
Before merge layer
    depth=0
    lll_word_hidstate.shape=torch.Size([4, 86, 768])
After merge layer
    depth=0
    lll_word_hidstate.shape=torch.Size([4, 86, 300])
Before ilabelling
    depth=0
    lll_word_hidstate.shape=torch.Size([4, 86, 300])
After ilabelling
    depth=0
    lll_word_score.shape=torch.Size([4, 86, 6])
*********** Starting iterative layer
    ilay=0
Before iterative layer
```

```
    ilay=0
    depth=1
    lll_hidstate.shape=torch.Size([4, 121, 768])
After iterative layer
    ilay=0
    depth=1
    lll_hidstate.shape=torch.Size([4, 121, 768])
*********** Starting iterative layer
    ilay=1
Before iterative layer
    ilay=1
    depth=1
    lll_hidstate.shape=torch.Size([4, 121, 768])
After iterative layer
    ilay=1
    depth=1
    lll_hidstate.shape=torch.Size([4, 121, 768])
Before dropout
    depth=1
    lll_hidstate.shape=torch.Size([4, 121, 768])
After dropout
    depth=1
    lll_hidstate.shape=torch.Size([4, 121, 768])
gather 2 inputs, then output
    lll_hidstate.shape=torch.Size([4, 121, 768])
    lll_loc.shape=torch.Size([4, 86, 768])
    lll_word_hidstate.shape=torch.Size([4, 86, 768])
before argmax
    lll_word_score.shape=torch.Size([4, 86, 6])
after argmax
    ll_greedy_ilabel.shape=torch.Size([4, 86])
before embedding
    ll_greedy_ilabel.shape=torch.Size([4, 86])
after embedding
    lll_word_hidstate.state=torch.Size([4, 86, 768])
just summed two signals with this shape
    depth=1
    lll_word_hidstate.shape=torch.Size([4, 86, 768])
Before merge layer
    depth=1
    lll_word_hidstate.shape=torch.Size([4, 86, 768])
After merge layer
    depth=1
    lll_word_hidstate.shape=torch.Size([4, 86, 300])
Before ilabelling
    depth=1
    lll_word_hidstate.shape=torch.Size([4, 86, 300])
After ilabelling
    depth=1
    lll_word_score.shape=torch.Size([4, 86, 6])
*********** Starting iterative layer
```

```
    ilay=0
Before iterative layer
    ilay=0
    depth=2
    lll_hidstate.shape=torch.Size([4, 121, 768])
After iterative layer
    ilay=0
    depth=2
    lll_hidstate.shape=torch.Size([4, 121, 768])
"""
```

# 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.