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 bnet

In Sax, I have replaced the original O6 bnet by a slightly different one. In this section, I describe the original O6 bnet. In the next section, I describe the current Sax bnet and explain why I changed it slightly.

This section describes the bnet in the O6 software in 3 part:

1. Sax code, now replaced, that reproduces the original O6 bnet.
2. Excerpt of print-out to console produced when I run the jupyter notebook for training the NN for task=ex.
3. Output of texnn tool (Ref.[5]) with drawing and structural equations for original O6 bnet.

2.1 Defunct Sax code that reproduces O6 bnet

```
def sax_get_llll_word_score(self, x_d, ttt, verbose=False):
"""

This method is used inside self.forward() and is the heart of that
method. It contains a while loop over depths that drives a batch
through the layers of the model and returns ‘llll_word_score‘.
Setting ‘verbose‘ to True prints out a detailed trail of what occurs
in this method. The following example was obtained from such a
verbose trail.

Assume:
batch_size= 24,
hidden_size= 768,
NUM_ILABELS= 6,
MERGE_DIM= 300
2 iterative layers and 5 depths.

lll_word_score is the output of the last ilabelling_layer for each
depth

llll_word_score is a list of lll_word_score
```

```

len( llll_word_score)= 5 = num_depths

Note that llll_word_scoreT = Ten(llll_word_score)

Parameters
----------
x_d: OrderedDict
ttt: str
verbose: bool

Returns
-------
list[torch.Tensor]
    llll_word_score

"""
# lll_label is similar to Openie6.labels
# first (outer) list over batch/sample of events
# second list over extractions
# third (inner) list over number of labels in a line
# after padding and adding the 3 unused tokens

# batch_size, num_depths, num_words = y_d["lll_ilabel"].shape
# sometimes num_depths will exceed max.
# This doesn't happen when training, because
# num_depths is specified when training.
# if ttt != 'train':
num_depths = get_num_depths(self.params.task)

# `loss_fun` is not used in this function anymore
# loss_fun, lstm_loss = 0, 0

# batch_text = " ".join(redoL(meta_d["l_orig_sent"]))
# starting_model_input = \
#     torch.Tensor(self.auto_tokenizer.encode(batch_text))
hstate_count = Counter(verbose, "lll_hidstate")
word_hstate_count = Counter(verbose, "lll_word_hidstate")
lll_hidstate, _ = self.starting_model(x_d["ll_osent_icode"])
hstate_count.new_one(reset=True)
if verbose:
    print()
    print("ll_osent_icode.shape", x_d["ll_osent_icode"].shape)
    print("after starting_model, lll_hidstate.shape",
          lll_hidstate.shape)

lll_word_score = Ten([0])  # this statement is unecessary
llll_word_score = []  # ~ Openie6.all_depth_scores
depth = 0
# loop over depths
while True:
    for ilay, layer in enumerate(self.iterative_transformer):
        comment(verbose,
                prefix="*********** Starting iterative layer",
                params_d={"ilay": ilay})
        # layer(lll_hidstate)[0] returns a copy
        # of the tensor lll_hidstate after transforming it
        # in some way
        # [0] chooses first component
        comment(
            verbose,
            prefix="Before iterative layer",
            params_d={
                "ilay": ilay,
                "depth": depth,
                "lll_hidstate.shape": lll_hidstate.shape})
        lll_hidstate = layer(lll_hidstate)[0]
        hstate_count.new_one()
        comment(
            verbose,
            prefix="After iterative layer",
            params_d={
                "ilay": ilay,
                "depth": depth,
                "lll_hidstate.shape": lll_hidstate.shape})
    comment(verbose,
            prefix="Before dropout",
            params_d={
                "depth": depth,
                "lll_hidstate.shape": lll_hidstate.shape})
    lll_hidstate = self.dropout_fun(lll_hidstate)
    hstate_count.new_one()
    comment(verbose,
            prefix="After dropout",
            params_d={
                "depth": depth,
                "lll_hidstate.shape": lll_hidstate.shape})
    lll_loc = x_d["ll_osent_wstart_loc"].unsqueeze(2). \
        repeat(1, 1, lll_hidstate.shape[2])
    lll_word_hidstate = torch.gather(
        input=lll_hidstate,
        dim=1,
        index=lll_loc)
    comment(
        verbose,
        prefix="Gather's 2 inputs, then output",
        params_d={
            "lll_hidstate.shape": lll_hidstate.shape,
            "lll_loc.shape": lll_loc.shape,
            "lll_word_hidstate.shape": lll_word_hidstate.shape})
    word_hstate_count.new_one(reset=True)
    if depth != 0:
        comment(
            verbose,
            prefix="before argmax",
            params_d={"lll_word_score.shape": lll_word_score.shape})
        ll_greedy_ilabel = torch.argmax(lll_word_score, dim=-1)
        comment(
            verbose,
            prefix="after argmax",
            params_d={"ll_greedy_ilabel.shape":
                          ll_greedy_ilabel.shape})
        # not an integer code/embedding
        comment(
            verbose,
            prefix="before embedding",
            params_d={"ll_greedy_ilabel.shape":
                          ll_greedy_ilabel.shape})
        lll_pred_code = self.embedding(ll_greedy_ilabel)
        comment(
            verbose,
            prefix="after embedding",
            params_d={"lll_word_hidstate.state":
                          lll_word_hidstate.shape})
        lll_word_hidstate += lll_pred_code
        word_hstate_count.new_one()
        comment(
            verbose,
            prefix="just summed two signals with this shape",
            params_d={
                "depth": depth,
                "lll_word_hidstate.shape": lll_word_hidstate.shape})
    comment(verbose,
            prefix="Before merge layer",
            params_d={
                "depth": depth,
                "lll_word_hidstate.shape": lll_word_hidstate.shape})
    lll_word_hidstate = self.merge_layer(lll_word_hidstate)
    comment(
        verbose,
        prefix="After merge layer",
        params_d={
            "depth": depth,
            "lll_word_hidstate.shape": lll_word_hidstate.shape})
    comment(
        verbose,
        prefix="Before ilabelling",
        params_d={
            "depth": depth,
            "lll_word_hidstate.shape": lll_word_hidstate.shape})
    lll_word_score = self.ilabelling_layer(lll_word_hidstate)
    comment(
        verbose,
        prefix="After ilabelling",
        params_d={
            "depth": depth,
            "lll_word_score.shape": lll_word_score.shape})
    llll_word_score.append(lll_word_score)

    depth += 1
    if depth >= num_depths:
        break

    if ttt != 'train':
        ll_pred_ilabel = torch.max(lll_word_score, dim=2)[1]
        valid_extraction = False
        for l_pred_ilabel in ll_pred_ilabel:
            if is_valid_label_list(
                    l_pred_ilabel, self.params.task, "ilabels"):
                valid_extraction = True
                break
        if not valid_extraction:
            break
comment(
    verbose,
    params_d={
        "len(llll_word_score)": len(llll_word_score),
        "llll_word_score[0].shape": llll_word_score[0].shape})
return llll_word_score
```

2.2 statements printed to console

```
"""
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's 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])
"""
```

2.3 texnn output for original O6 bnet

$\underline{a}^{[86]}$:	`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`
$\underline{I}^{[121],[768]}$:	`lll_hidstate`
$\underline{L}^{[86],[6]}$:	`lll_word_score`
$\underline{M}^{[86],[300]}$:	`lll_word_hidstate`
$\underline{S}^{[86],[768]}$:	`lll_word_hidstate`

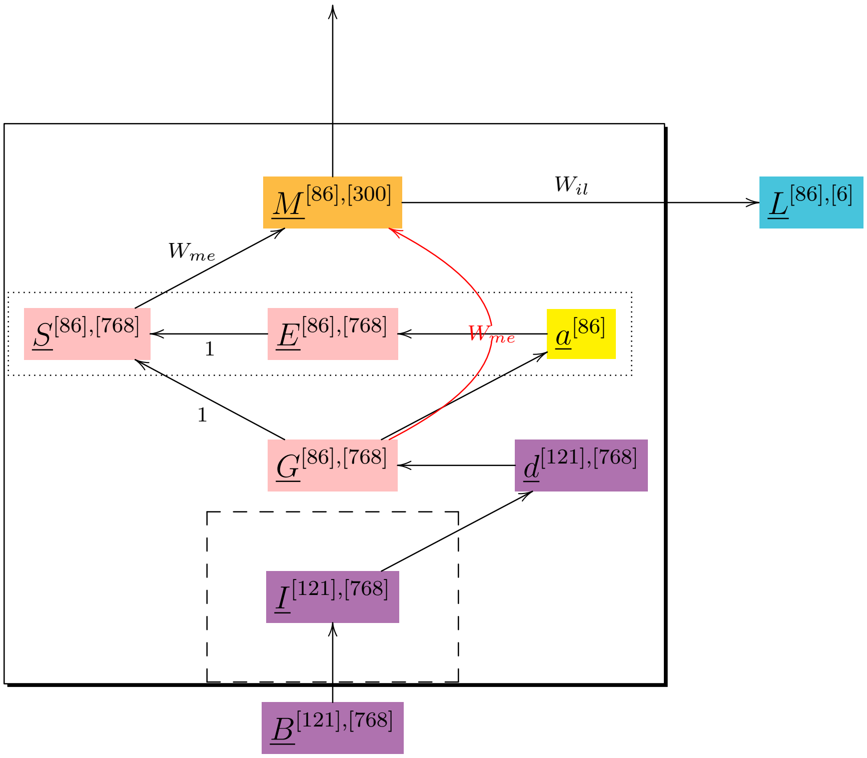

$\underline{M}^{[86],[300]}$
W_{il}
$\underline{L}^{[86],[6]}$
W_{me}
$\underline{S}^{[86],[768]}$
$\underline{E}^{[86],[768]}$
W_{me}
$\underline{a}^{[86]}$
1
1
$\underline{G}^{[86],[768]}$
$\underline{d}^{[121],[768]}$
$\underline{I}^{[121],[768]}$
$\underline{B}^{[121],[768]}$

Figure 1: O6 bnet. (Slightly different from 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 omited 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. $D = dn_{\underline{h}}$ where $d = 768$ is the hidden dimension per head, and $n_{\underline{h}} = 12$ is the number of heads.

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

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

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

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

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

$$\begin{gathered} I^{[121],[768]} = \left[B^{[121],[768]} \mathbb{1}(depth = 0) + M^{[86],[300]} \mathbb{1}(depth \neq 0)\right] \\ : \texttt{lll\_hidstate} \end{gathered} \tag{1f}$$

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

$$\begin{gathered} M^{[86],[300]} = \left[S^{[86],[768]} \mathbb{1}(depth \neq 0) + G^{[86],[768]} \mathbb{1}(depth = 0)\right] W_{mer}^{[768],[300]} \\ : \texttt{lll\_word\_hidstate} \end{gathered} \tag{1h}$$

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

3 Sax Code for current Sax bnet

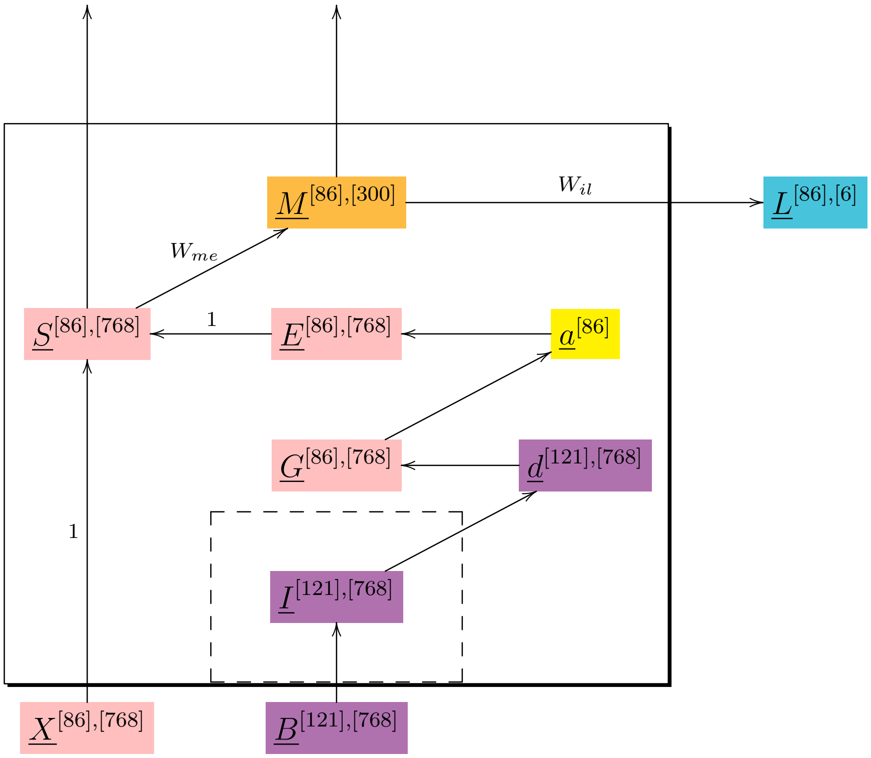
I changed the O6 bnet to the current one because the O6 bnet treats the first extraction ($depth = 0$) differently from the higher depth extractions. (the dotted box is only used for $depth \neq 0$). In the current Sax bnet, all 5 depths are treated the same.

This bnet change was achieved easily by

1. Calling the output of node $\underline{S}$, `lll_pred_code0`.

   Initializing variable `lll_pred_code0` to zero before going through any layers.

2. Eliminating the line `if depth != 0:` Note that whereas in the O6 bnet, feedback occurred solely through the node $\underline{M}$, in the new Sax bnet, feedback occurs via nodes $\underline{S}$ and $\underline{M}$.



$\underline{M}^{[86],[300]}$
W_{il}
$\underline{L}^{[86],[6]}$
W_{me}
$\underline{S}^{[86],[768]}$
1
$\underline{E}^{[86],[768]}$
$\underline{a}^{[86]}$
$\underline{G}^{[86],[768]}$
$\underline{d}^{[121],[768]}$
1
$\underline{I}^{[121],[768]}$
$\underline{X}^{[86],[768]}$
$\underline{B}^{[121],[768]}$

Figure 2: Sax bnet. (Slightly different from O6 bnet). 2 copies of dashed box are connected in series. 5 copies (5 depths) of plain box are connected in series. 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. $D = dn_{\underline{h}}$ where $d = 768$ is the hidden dimension per head, and $n_{\underline{h}} = 12$ is the number of heads.

3.1 texnn output for current Sax bnet

$\underline{a}^{[86]}$	:	`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`
$\underline{I}^{[121],[768]}$	:	`lll_hidstate`
$\underline{L}^{[86],[6]}$	:	`lll_word_score`
$\underline{M}^{[86],[300]}$	:	`lll_merge_hidstate`
$\underline{S}^{[86],[768]}$	:	`lll_pred_code0`
$\underline{X}^{[86],[768]}$	:	`lll_pred_code0`

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

$$\begin{gathered} B^{[121],[768]} = \mathrm{BERT}() \\ : \texttt{lll\_hidstate} \end{gathered} \tag{2b}$$

$$\begin{gathered} d^{[121],[768]} = \mathrm{dropout}(I^{[121],[768]}) \\ : \texttt{lll\_hidstate} \end{gathered} \tag{2c}$$

$$\begin{gathered} E^{[86],[768]} = \mathrm{embedding}(a^{[86]}) \\ : \texttt{lll\_pred\_code} \end{gathered} \tag{2d}$$

$$\begin{gathered} G^{[86],[768]} = \mathrm{gather}(d^{[121],[768]}; dim = -2) \\ : \texttt{lll\_word\_hidstate} \end{gathered} \tag{2e}$$

$$\begin{gathered} I^{[121],[768]} = \left[B^{[121],[768]}\mathbb{1}(depth = 0) + M^{[86],[300]}\mathbb{1}(depth \neq 0)\right] \\ : \texttt{lll\_hidstate} \end{gathered} \tag{2f}$$

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

$$\begin{gathered} M^{[86],[300]} = S^{[86],[768]}W_{me}^{[768],[300]} \\ : \texttt{lll\_merge\_hidstate} \end{gathered} \tag{2h}$$

$$\begin{gathered} S^{[86],[768]} = E^{[86],[768]} + X^{[86],[768]} \\ : \texttt{lll\_pred\_code0} \end{gathered} \tag{2i}$$

$$\begin{gathered} X^{[86],[768]} = S^{[86],[768]}\mathbb{1}(depth \neq 0) \\ : \texttt{lll\_pred\_code0} \end{gathered} \tag{2j}$$

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.