Project name: Find Hidden Entities in Wikipedia Articles

Dataset: Wikidata5m

Reference: Bidirectional relation-guided attention network with semantics and knowledge for relational triple extraction.

Purpose:

- Extract relationship triples from Wikipedia articles
- Build a neural network and train it to extract the triples (head, relationship, tail) from unstructured text
- Knowledge graph that we extract where head representing usually the subject and tail representing the object

Approach for preparation:

We are using wikidata5m dataset which contains three files containing the data:

1- Transductive: triples of the knowledge graph where exists triples in the shape (head id, relation id, tail id)
2- Corpus: containing Wikipedia articles and each one is represented by id
3- Entity & relation aliases: containing for each query_id a list of aliases representing it and for each relationship_id a list of names for the relationship

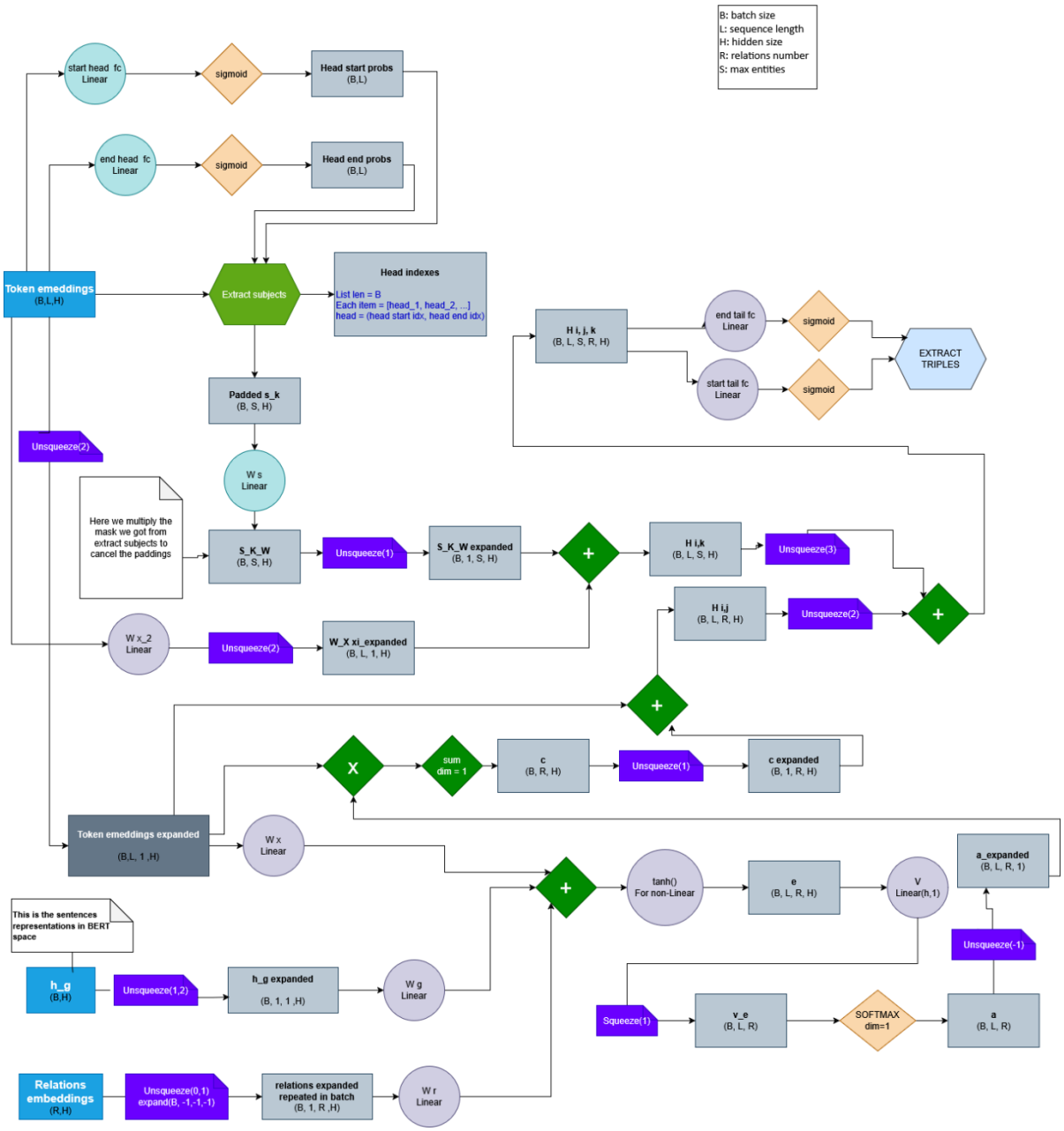
BRASK approach:

BRASK aims to extract correct triples, doing 2 extractions (forward & backward) and take the intersection of those triples, this approach will solve also overlapping problems

The bidirectional extraction let us use attention guidance in both directions:

1- Semantic general language knowledge for the forward extraction representing general knowledge and based on BERT
2- Domain knowledge (knowledge graph) for the backward extraction where we train transE on our existing knowledge graph

Neural network graph:



Preparation:

Dictionary preparation:

Implemented through prep.py

In this step, the text files of the dataset are converted into python dictionaries which are saved in .pkl files.

.pkl files paths are saved in data.py

Dictionaries generated:

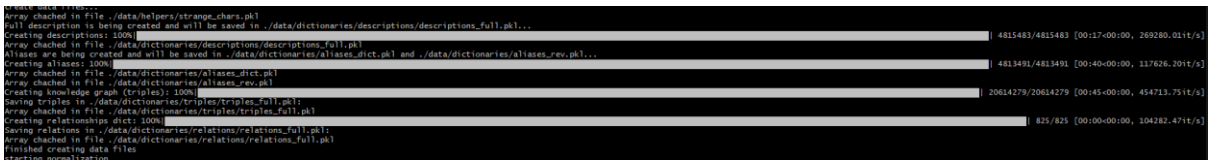
1- Description texts: including the entity_id and the entity_description (entity text)
2- Aliases: including the entity_id and the entity_aliases (list of strings or titles referring to the entity_id)
3- Relationships: including relationship_id and relationship_names (list of strings)
4- Triples – knowledge graph: including triples (head_id, relationship_id, tail_id) where the head_id & tail_id referring to the entity_id in 1&2 dictionaries and relationship_id referring to 4

Also, in prep.py, we are preparing minimized dictionaries for descriptions, aliases, relationships triples depending on the size chosen (10, 100, 1k, 1m, full)

The minimization process is done with the following steps:

- K descriptions chosen randomly
- Get the triples of those k descriptions
- Add tails ids into the descriptions dictionary
- Extract relationships of those triples
- Get aliases of all descriptions

The descriptions dictionary length will not be k exactly but will be 10 + descriptions of the triples' tails



1- Normalization:

Implemented through Normalize.py

Parallel normalization for descriptions.

Normalization steps:

a- Replace special chars with their English version using re compiled patterns
    a. For example ắặằẳẵăâấậầ would be converted to 'a'

b. The preparation of the compiled pattern to do this step is done in file prep.py

b- Remove any non-english chars (Chinese, Arabic...)

c- Remove multiple white spaces

No more normalization steps are done (e.g. removing punctuation marks, make the description small letters) to not loose the meaning of the sentences.

Train transE neural network:

Implemented through: TransE.py

TransE is an energy-based model that produces knowledge base embeddings. It models relationships by interpreting them as translations operating on the low-dimensional embeddings of the entities.

Algorithm 1 Learning TransE

input Training set $S = \{(h, \ell, t)\}$, entities and rel. sets E and L, margin γ, embeddings dim. k.

initialize $\boldsymbol{\ell} \leftarrow \text{uniform}(-\frac{6}{\sqrt{k}}, \frac{6}{\sqrt{k}})$ for each $\ell \in L$

$\boldsymbol{\ell} \leftarrow \boldsymbol{\ell} / \|\boldsymbol{\ell}\|$ for each $\ell \in L$

$\mathbf{e} \leftarrow \text{uniform}(-\frac{6}{\sqrt{k}}, \frac{6}{\sqrt{k}})$ for each entity $e \in E$

loop

  $\mathbf{e} \leftarrow \mathbf{e} / \|\mathbf{e}\|$ for each entity $e \in E$

  $S_{batch} \leftarrow$ sample(S, b) // sample a minibatch of size b

  $T_{batch} \leftarrow \emptyset$ // initialize the set of pairs of triplets

  for $(h, \ell, t) \in S_{batch}$ **do**

    $(h', \ell, t') \leftarrow$ sample$(S'_{(h,\ell,t)})$ // sample a corrupted triplet

    $T_{batch} \leftarrow T_{batch} \cup \{((h, \ell, t), (h', \ell, t'))\}$

  end for

  Update embeddings w.r.t. $\sum_{((h,\ell,t),(h',\ell,t')) \in T_{batch}} \nabla \left[\gamma + d(\boldsymbol{h} + \boldsymbol{\ell}, \boldsymbol{t}) - d(\boldsymbol{h'} + \boldsymbol{\ell}, \boldsymbol{t'})\right]_+$

end loop

The idea behind TransE is to process triples (head, relation, tail) represented in some space, TransE makes sure that:

$$\overrightarrow{Head\ vecotr} + \overrightarrow{Relation\ vector} \approx \overrightarrow{tail\ vector}$$

TransE is used to embed the knowledge graph relations, to create domain knowledge and use it in the backward extraction.

The code in transE.py uses the triples dictionary and train transE neural network to generate rel_embeddings which will be used in the backward extraction.

I did 140 epochs with 52224 as the batch_size and run the model on cuda and it results in perfect training for the neural network and saved the resulted relationship embeddings.

Prepare relations embeddings:

Implemented in file: Prep_rels.py

"To integrate the semantics and knowledge into our model, we utilize semantic and knowledge relations as guidance in bidirectional extraction. Semantic relations are acquired with the pretrained language model BERT."

In our dictionaries, we have a dictionary called relationships, in this dictionary we have relation_id and a list of relationship names.

Ex. We have relationship with id p_1, with the list = ['name_1_token_1, name_1_token_2', 'name_2_token_1]

1- We do BERT embedding of each name for the tokens and we do pool average for the token embeddings
2- For each name average embeddings we take the last two layers of the BERT embeddings (to generalize)
3- We do pool average for all the names of the relationship and we have the embedding representing the relation

This is representing the general semantic knowledge of our model. Which we will use in our relation semantic guidance

Also, there is an optimized version of the function extracting the relation embeddings for the gpu use

Ground truth preparation:

Implemented by: Prep_ground_truth.py

For extracting subjects/objects in forward/backward extractions, we are using binary tagging system. Which means that for every token in our description text, we will have either 0 or 1. 1 representing if the token is start/end of head/tail.

Output: silver spans dictionary containing:

1- Head start: 1s for tokens representing head start
2- Head end: 1s for tokens representing head end
3- Tail start
4- Tail end

Those silver spans are used to calculate the loss when training the neural network by using BCE.

Ex.

We have 2 descriptions (entities - texts) in our descriptions dictionary:

a- "q1" → Europe is a continent of the world continents
b- "q2" → Italy is a country in Europe

And we have the following aliases:

a- "q1" → ["europe", "oropa", "eu"]
b- "q2" → ["Italy", "italia" , "it"]

And we have those triples:

a- "q1" → [(), ... other triples]
b- "q2" → [(q2, r1, q1), ...other triples] (considering r1 is relationship for "located_in")

I have triple (q2, r1, q1) representing that "Italy is located in Europe"

From this triple, we have to create silver spans for the q2 which will show where the head (Italy) starts and ends, and where is the tail (Europe) starts and ends.

Notes:

- for simplicity this example head's and tail's are one token but they might be 5 tokens
- for simplicity we are showing a raw text, in real life, we are tokenizing the sentences so one token here might be multiple tokens after tokenization using BERT tokenizer)

We try to search for the head/tail aliases in the description and we decide the start/end index for each one of them.

In our example, the triple is (q1, _, q2), the aliases for head are [“europe”, “oropa”, “eu”] and the aliases for tail are [“Italy”, “italia” , “it”], we can find from our description text “Italy is a country in Europe”, that the head “italy” starting at index 0 and ending at index 0, and for our tail “europe” starting at index 5 and ending at index 5 (considering here tokenization is based on space split).

We know that for q2, the triple (q2, _, q1):

- Head’s start index = 0
- Head’s end index = 0
- Tail’s start index = 5
- Tail’s end index = 5

We are using re patterns to search for the alias string in the description.

Because in our model, we are doing binary tagging system in our subject/object extractions, we have to create silver spans tensors that each one have the shape (batch_size, sequence_length), where each value in those tensors is 1 if the token is (head_start, head_end, tail_start, tail_end)

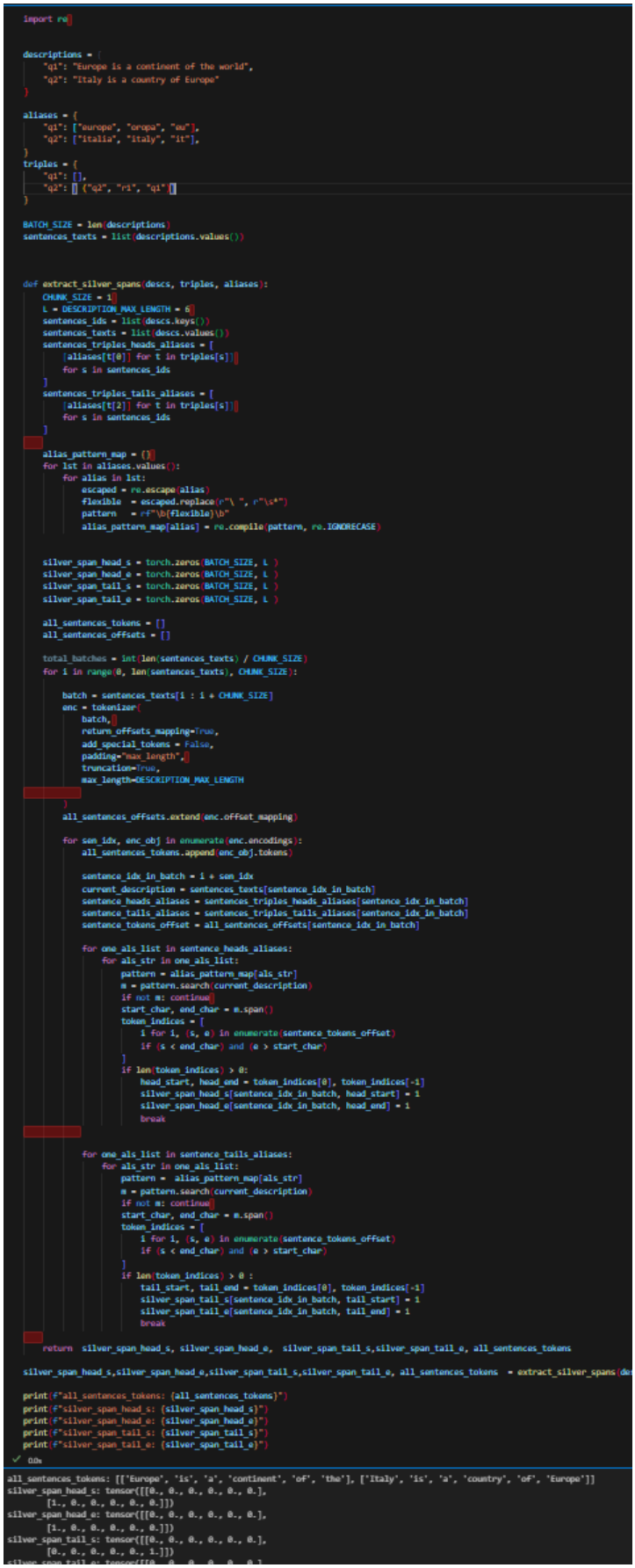


Figure 1 - Code screenshot for running the example on test notebook and executing the function extract_silver_spans -

```
all_sentences_tokens: [['Europe', 'is', 'a', 'continent', 'of', 'the'], ['Italy', 'is', 'a', 'country', 'of', 'Europe']]
silver_span_head_s: tensor([[0., 0., 0., 0., 0., 0.],
        [1., 0., 0., 0., 0., 0.]])
silver_span_head_e: tensor([[0., 0., 0., 0., 0., 0.],
        [1., 0., 0., 0., 0., 0.]])
silver_span_tail_s: tensor([[0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 1.]])
silver_span_tail_e: tensor([[0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 1.]])
```

The code gave result about the tokens (fortunately for this small example BERT tokenizer is not tokenizing multiple tokens for one word).

But we notice:

- silver_span_head_s[1][0] is 1 because Italy is the start of head
- silver_span_head_e[1][0] is 1 because Italy is the end of head
- silver_span_tail_s[1][-1] is 1 because Europe is the start of tail
- silver_span_tail_e[1][-1] is 1 because Europe is the end of tail

Model:

Implemented by: brask_model.py

Dataset:

The dataset that we will prepare for our model is a dataset including the description dictionary (only ones that have triples), we will batch those descriptions along with their silver spans (head start, head end, tail start, tail end) into batches through pytorch dataloader. Also, we will extract h_g for each description which is representing the embedding of the text, we will get this by pool average of the tokens inside the text using BERT.

$$h_g = avg\{x_1, x_2, \dots, x_l\} \tag{5}$$

Forward extraction:

1- Create probabilities of the start/end of the subjects (head) for the forward extraction, start/end of object (tail) for the backward extraction:

$$p_{i,sub\_s} = Sigmoid(W_{sub\_s}x_i + b_{sub\_s}) \tag{2}$$

$$p_{i,sub\_e} = Sigmoid(W_{sub\_e}x_i + b_{sub\_e}) \tag{3}$$

This will be generated by feeding the token embeddings into fully connected neural network to get probabilities of distinguishing each token in the input sequence.

Then from those probabilities we will get the spans (0,1) for those probabilities:

```python
#forward
f_sub_s_logits = self.f_start_sub_fc(token_embs)
f_sub_e_logits = self.f_end_sub_fc(token_embs)
f_sub_start_probs = self.sigmoid(f_sub_s_logits).squeeze(-1) # (b, l)
f_sub_end_probs = self.sigmoid(f_sub_e_logits).squeeze(-1)
#backward

b_obj_s_logits = self.b_start_obj_fc(token_embs)
b_obj_e_logits = self.b_end_obj_fc(token_embs)
b_obj_start_probs = self.sigmoid(b_obj_s_logits).squeeze(-1)
b_obj_end_probs = self.sigmoid(b_obj_e_logits).squeeze(-1) # (b,l)
```

Each probability tensor has shape (Batch_size, sequence length)

Then we will use function to extract the spans.

Example:

We see from this example if we had batch_size = 2, sequence_length=4, and fixed probabilities, we would have the indexes of head/tail from the probabilities.

```python
from utils.model_helpers import  extract_first_embeddings

head_start_probs = torch.tensor([
    [.8, .1, .1, .1],
    [.8, .2, .2, .1],
])
head_end_probs = torch.tensor([
    [.2, .8, .2, .2],
    [.8, .2, .3, .1],
])


tail_start_probs = torch.tensor([
    [.1, .2, .1, .8],
    [.1, .2, .8, .2],
])
tail_end_probs = torch.tensor([
    [.1, .1, .2, .8],
    [.2, .1, .1, .8],
])


token_embs = torch.randn(2,4,100)

f_padded_subj_embs, f_mask_subj_embs, f_head_idxs = extract_first_embedding
                token_embs, head_start_probs, head_end_probs, 4, threshold=.5)



b_padded_obj_embs, b_mask_obj_embs, b_tail_idxs = extract_first_embeddings(
    token_embs, tail_start_probs, tail_end_probs, 4, threshold=.5)


print(f"f_head_idxs: {f_head_idxs}")
print(f"b_tail_idxs: {b_tail_idxs}")
```

✓ 0.0s

```
f_head_idxs: [[(0, 1)], [(0, 0)]]
b_tail_idxs: [[(3, 3)], [(2, 3)]]
```

We have similar function to extract triples (head, relation, tail) from the tail/head probabilities in the end of forward/backward extraction.

Neural network forward:

1- Create head start/end probabilities from passing token embeddings into fully-connected neural network and pass them into sigmoid activation

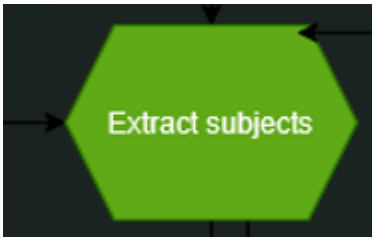
$$p_{i,sub\_s} = Sigmoid(W_{sub\_s}x_i + b_{sub\_s})$$

$$p_{i,sub\_e} = Sigmoid(W_{sub\_e}x_i + b_{sub\_e})$$

2- Extract subject set S representing a vector of the subjects

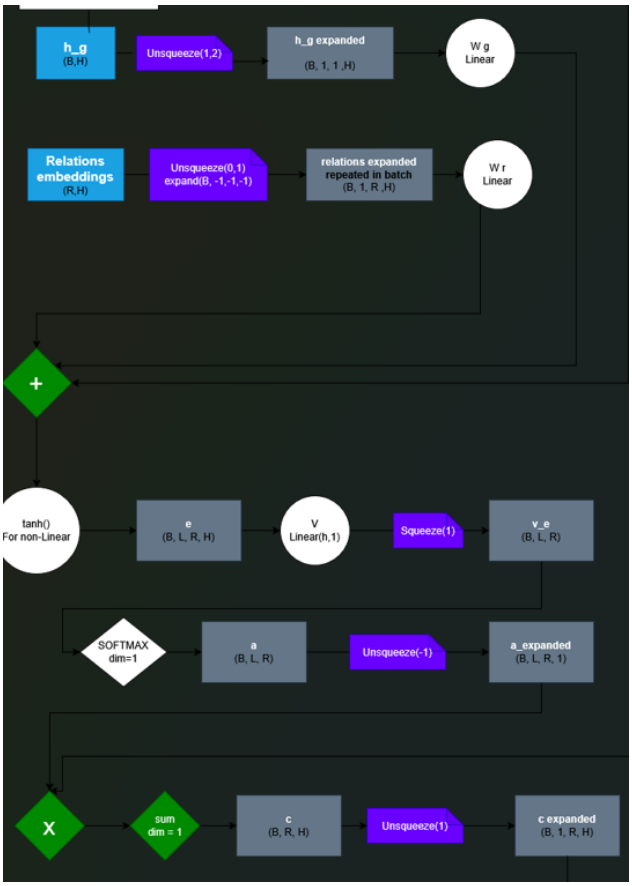
Where each S_k represents a subject by averaging the embeddings of start token and end token

$$s_k = avg(x_{k,sub\_s},\ x_{k,sub\_e})$$



3- We use predefined R which is relation embeddings got in the file prep_relations where we save a tensor representing the relationships embeddings in BERT space (by taking last two hidden states of all tokens and use average poolings

4- For each subject extracted, we can extract the relations using semantic relation guidance, BRASK adopts specific attention mechanisms to compute attention scores that reflect a different role of each token. h_g is pre-defined before representing the sentences in BERT space. $e_{i,j}$ attention scores that measure different role played by each token. c_j represents fine-grained sentence representations.



$$h_g = avg\{x_1, x_2, \ldots, x_l\}$$

$$e_{i,j} = V^T tanh(W_r r_j + W_g h_g + W_x x_i)$$

$$a_{i,j} = softmax(e_{i,j})$$

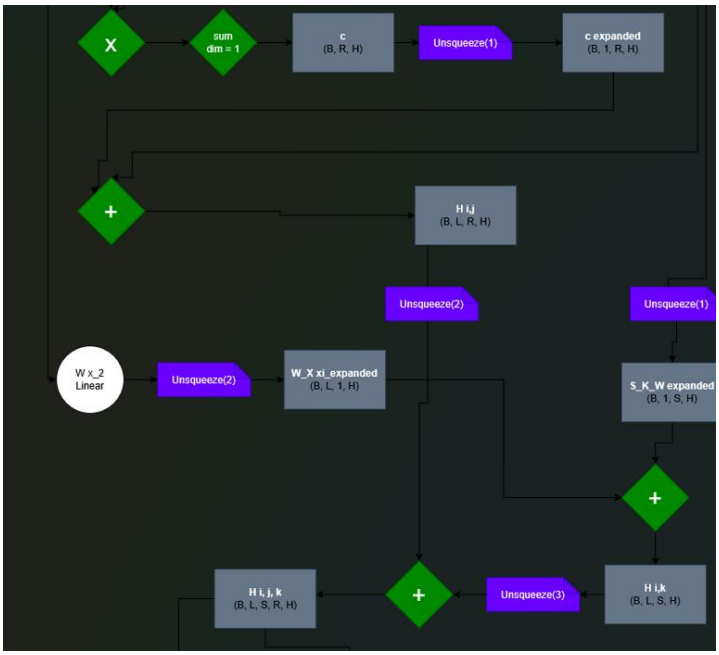
$$c_j = \sum_{i=1}^{l} a_{i,j} x_i$$

5- After we extract the objects. We do this by fusing: Subject representations and fine-grained sentence expressions of the semantic relations

$$H_{i,k} = W_s s_k + W_x x_i$$

$$H_{i,j} = c_j + x_i$$

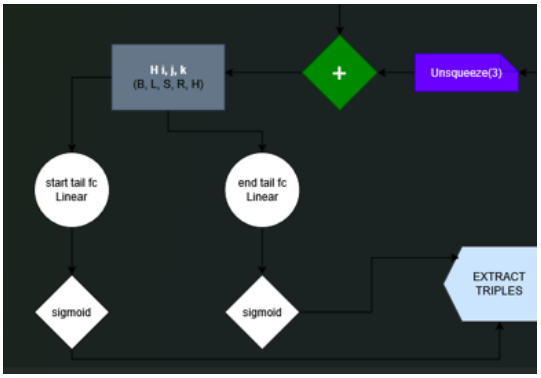
$$H_{i,j,k} = H_{i,k} + H_{i,j}$$



6- We feed H_{ijk} into fully connected neural network to obtain the start and end probabilities of each corresponding object

$$p_{i,obj\_s} = Sigmoid(W_{obj\_s} H_{i,j,k} + b_{obj\_s})$$

$$p_{i,obj\_e} = Sigmoid(W_{obj\_e} H_{i,j,k} + b_{obj\_e})$$



Backward triple extraction:

$$e'_{i,j} = V^T tanh(W'_r r'_j + W'_g h_g + W'_x x_i)$$

$$a'_{i,j} = softmax(e'_{i,j})$$

$$c'_j = \sum_{i=1}^{l} a'_{i,j} x_i$$

We do the same as forward extraction but:

- instead of using semantics relation guidance (BERT embeddings for relation), we use knowledge relation guidance (extract knowledge relations from the training of the TransE to our knowledge graph)
- Instead of extracting the subject and using guidance to extract objects, we extract objects and use knowledge to extract subjects

Loss calculations:

The model total loss is composed of $\mathcal{L}$ for forward extraction loss and $\mathcal{L}'$ for backward loss extraction.

$$\mathcal{L}_{total} = \sum_{d=1}^{|D|} (\mathcal{L} + \mathcal{L}')$$

The loss for forward extraction is calculating by summing the loss of the subject extraction and the loss of object extraction under all relations.

The loss of subject extraction is calculated using **Binary Cross Entropy** function which compares the predicted triples with the silver spans created in previous steps.

Minimizing the loss is through **Adam stochastic gradient descent**

```python
def compute_loss(out, batch, bce_losses):
    L = 0.0

    L += bce_losses['f_sub_s'](out['forward']['sub_s'].squeeze(-1),
                               batch['labels_head_start'])
    L += bce_losses['f_sub_e'](out['forward']['sub_e'].squeeze(-1),
                               batch['labels_head_end'])
    L += bce_losses['f_obj_s'](out['forward']['obj_s'].squeeze(-1),
                               batch['labels_tail_start'])
    L += bce_losses['f_obj_e'](out['forward']['obj_e'].squeeze(-1),
                               batch['labels_tail_end'])

    L += bce_losses['b_obj_s'](out['backward']['obj_s'].squeeze(-1),
                               batch['labels_tail_start'])
    L += bce_losses['b_obj_e'](out['backward']['obj_e'].squeeze(-1),
                               batch['labels_tail_end'])
    L += bce_losses['b_sub_s'](out['backward']['sub_s'].squeeze(-1),
                               batch['labels_head_start'])
    L += bce_losses['b_sub_e'](out['backward']['sub_e'].squeeze(-1),
                               batch['labels_head_end'])
    return L / 8.0
def build_bce_loss_dict(pos_weights):
    pw = {k: torch.tensor(v, dtype=torch.float32).to(device)
      for k, v in pos_weights.items()}
    bce_losses = {
        # forward
        'f_sub_s': nn.BCEWithLogitsLoss(pos_weight=pw['labels_head_start']),
        'f_sub_e': nn.BCEWithLogitsLoss(pos_weight=pw['labels_head_end']),
        'f_obj_s': nn.BCEWithLogitsLoss(pos_weight=pw['labels_tail_start']),
        'f_obj_e': nn.BCEWithLogitsLoss(pos_weight=pw['labels_tail_end']),
    }
    # backward re-uses the same four weights:
    bce_losses.update({
        'b_obj_s': bce_losses['f_obj_s'],
        'b_obj_e': bce_losses['f_obj_e'],
        'b_sub_s': bce_losses['f_sub_s'],
        'b_sub_e': bce_losses['f_sub_e'],
    })
    return bce_losses
```