Relational Graph Attention Network for Aspect-based Sentiment Analysis

-- Stargazers

Preprocessing of dataset

•Twitter (.raw):-

he might have got the nobel prize just for not being T
george bush
0

.raw -> .txt file : Read twitter data, extract the sentence and store it in txt file. It returns [sentence, aspect with its sentiment no., from_to]

•Semeval (.xml):-

```
<sentence id="2339">
  <text>I charge it at night and skip taking the cord with me because of the good battery life.</text>
  <aspectTerms>
  <aspectTerm term="cord" polarity="neutral" from="41" to="45"/>
  <aspectTerm term="battery life" polarity="positive" from="74" to="86"/>
  </aspectTerms>
</sentence>
```

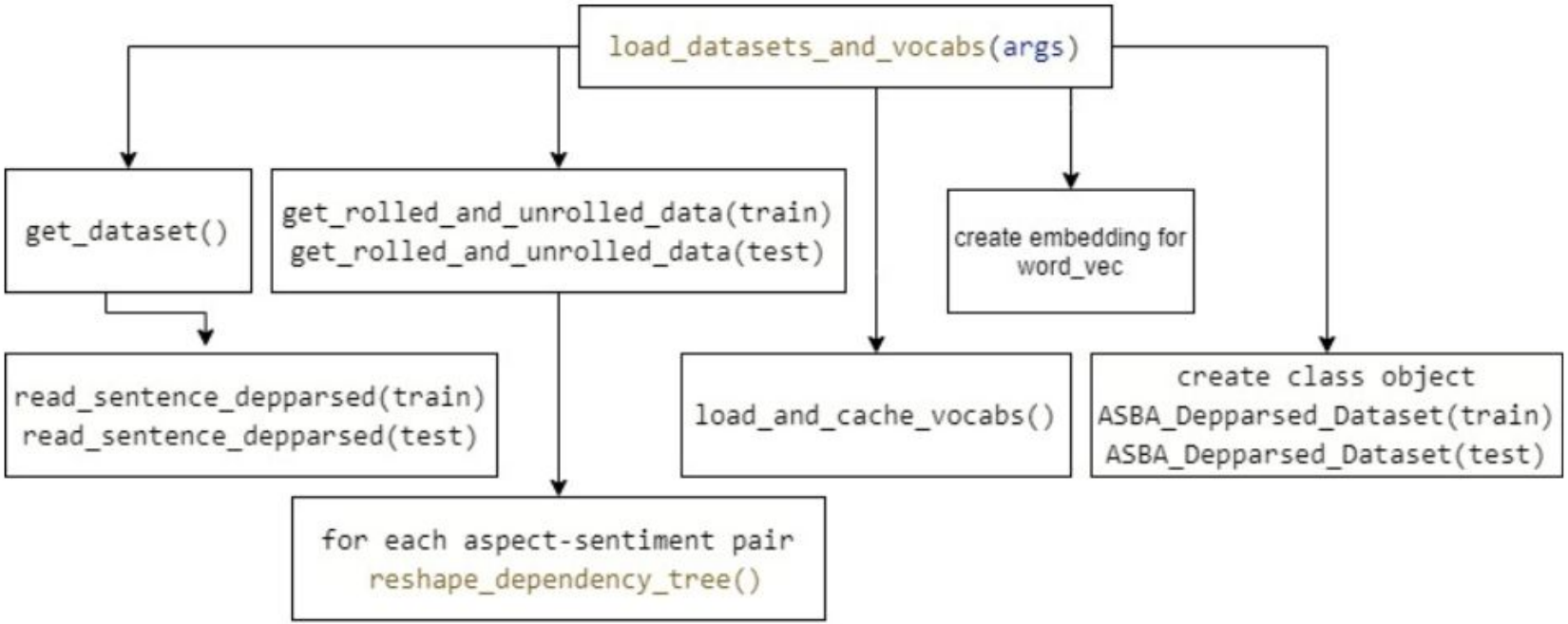
.xml -> .txt file : Read xml file and extract the sentences having aspect terms.

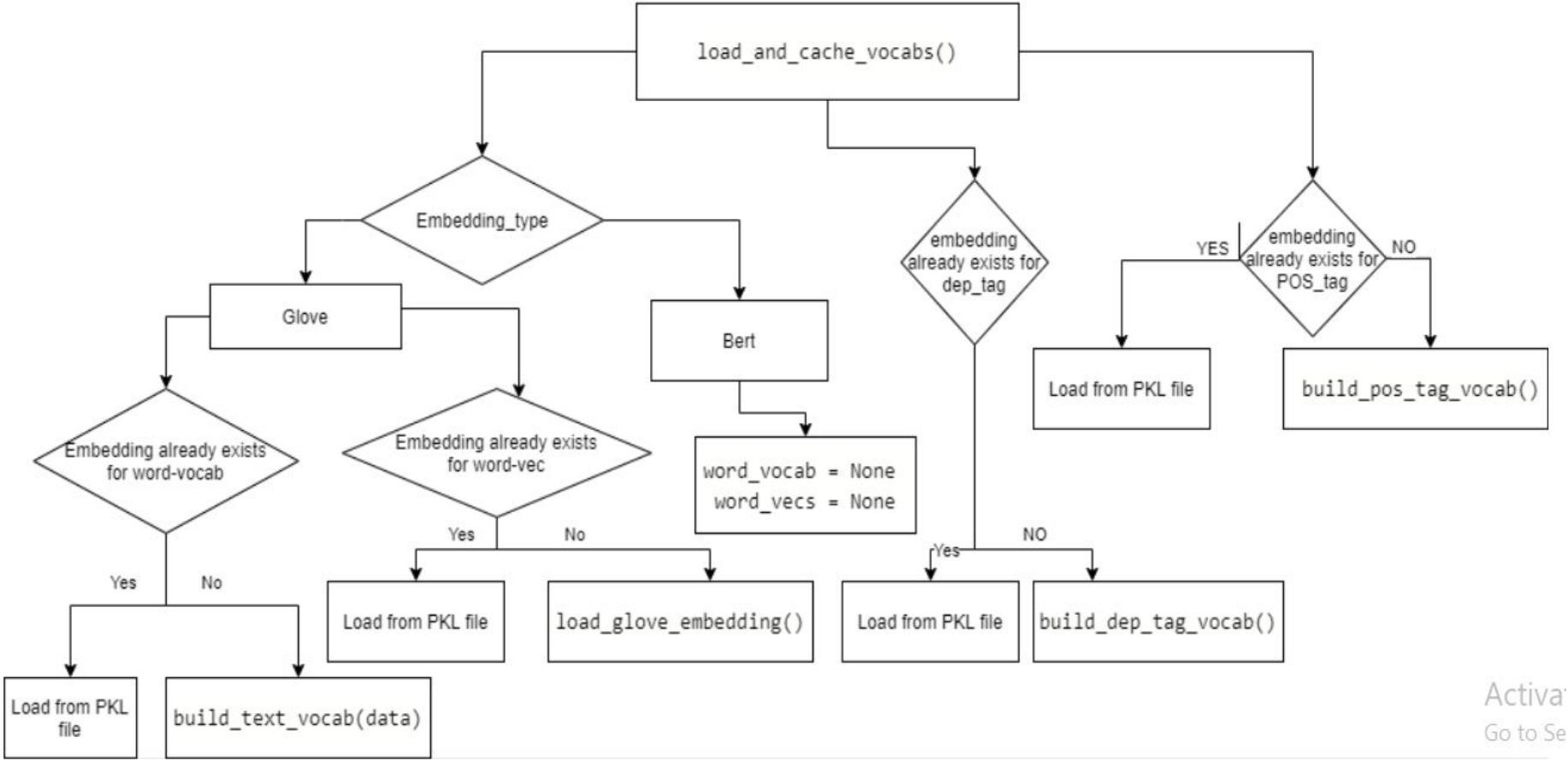
Preprocessing of dataset

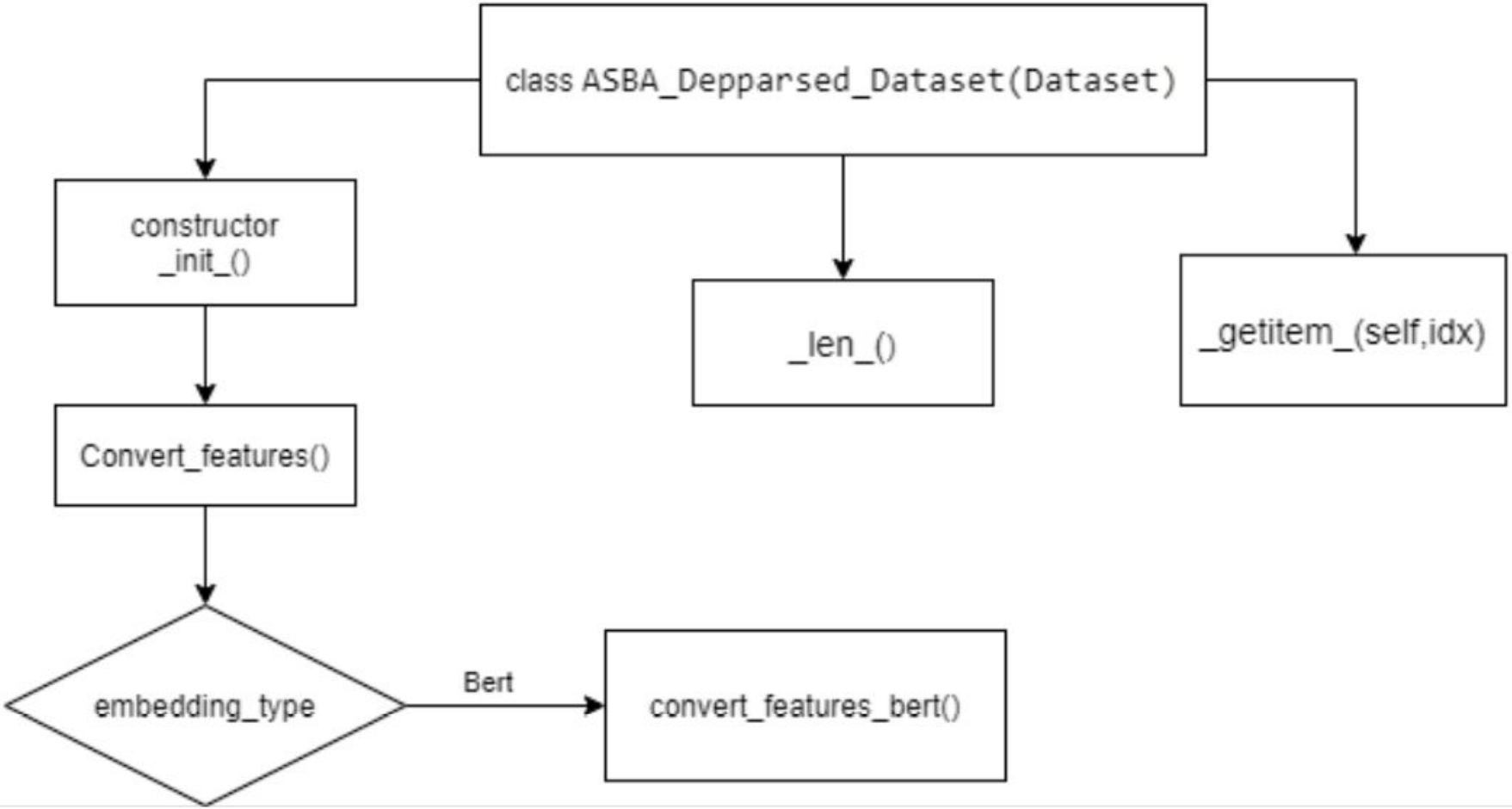
We then annotate the sentences from the text file using dependency parser. It produces for each sentence, various parameters like tokens, pos tags, predicted dependencies, predicted heads, etc.
Finally, we save it in a json file which becomes the dataset to work with.

{"sentence": "he might have got the nobel prize just for not being george bush",
"aspect_sentiment": [["george bush", "neutral"]], "from_to": [[11, 13]],
"tokens": ["he", "might", "have", "got", "the", "nobel", "prize", "just", "for", "not", "being", "george", "bush"],
"tags": ["PRP", "MD", "VB", "VBN", "DT", "JJ", "NN", "RB", "IN", "RB", "VBG", "VB", "NN"],
"predicted_dependencies": ["nsubj", "aux", "aux", "root", "det", "amod", "dobj", "advmod", "prep", "neg", "aux", "pcomp", "dobj"],
"dependencies": [["nsubj", 4, 1], ["aux", 4, 2], ["aux", 4, 3], ["root", 0, 4], ["det", 7, 5], ["amod", 7, 6], ["dobj", 4, 7], ["advmod", 9, 8], ["prep", 4, 9], ["neg", 12, 10],
["aux", 12, 11], ["pcomp", 9, 12], ["dobj", 12, 13]], "predicted_heads": [4, 4, 4, 0, 7, 7, 4, 9, 4, 12, 12, 9, 12]}

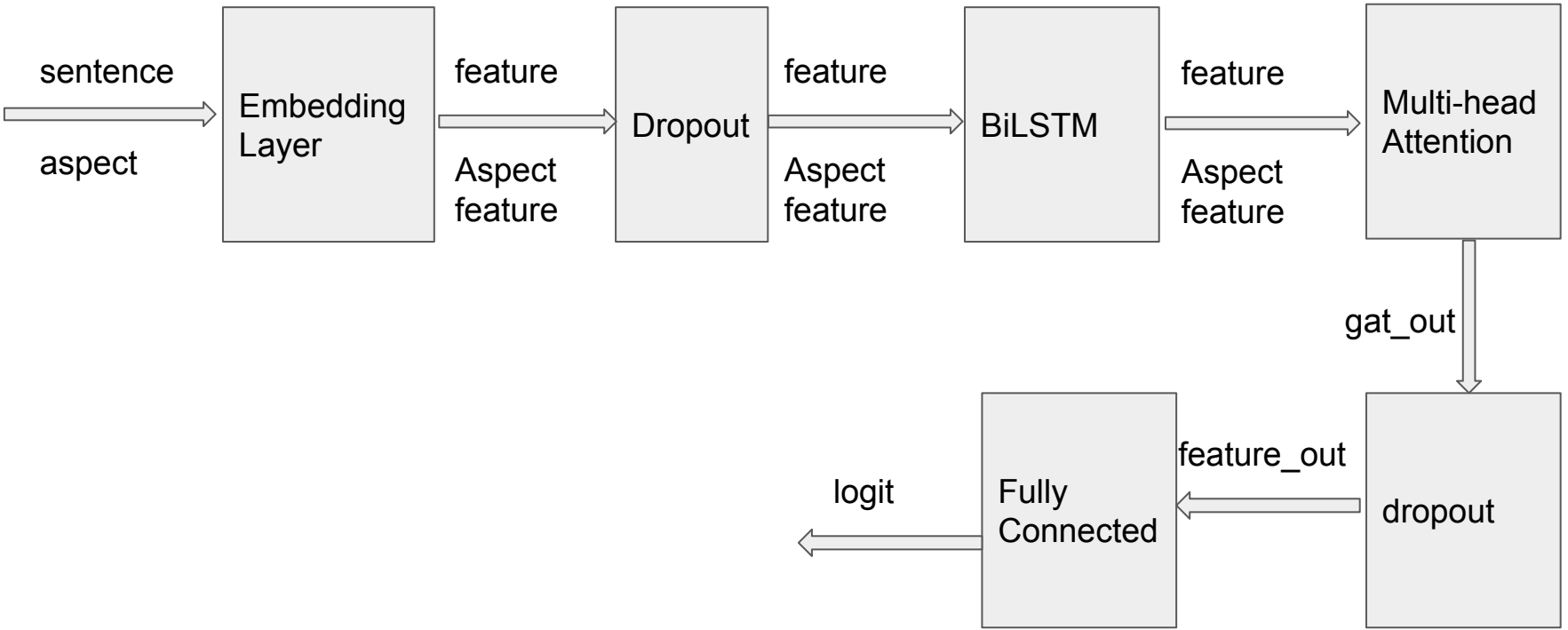
Datasets.py


load_datasets_and_vocabs(args)
get_dataset()
get_rolled_and_unrolled_data(train)
get_rolled_and_unrolled_data(test)
create embedding for word_vec
read_sentence_depparsed(train)
read_sentence_depparsed(test)
load_and_cache_vocabs()
create class object
ASBA_Depparsed_Dataset(train)
ASBA_Depparsed_Dataset(test)
for each aspect-sentiment pair
reshape_dependency_tree()


load_and_cache_vocabs()
Embedding_type
Glove
Bert
Embedding already exists for word-vocab
Embedding already exists for word-vec
word_vocab = None
word_vecs = None
Yes
No
Load from PKL file
build_text_vocab(data)
load_glove_embedding()
embedding already exists for dep_tag
YES
NO
build_dep_tag_vocab()
embedding already exists for POS_tag
build_pos_tag_vocab()


class ASBA_Depparsed_Dataset(Dataset)
constructor
init()
len()
getitem(self,idx)
Convert_features()
embedding_type
Bert
convert_features_bert()

Aspect Text GAT only



```
feature = self.embed(sentence)  # (N, L, D)
aspect_feature = self.embed(aspect) # (N, L', D)
```

Embedding

```
feature = self.dropout(feature)
aspect_feature = self.dropout(aspect_feature)
```

Dropout

```
feature, _ = self.bilstm(feature) # (N,L,D)
aspect_feature, _ = self.bilstm(aspect_feature) #(N,L,D)
aspect_feature = aspect_feature.mean(dim = 1) # (N, D)
```

BiLSTM

```
self.gat = [DotprodAttention().to(args.device) for i in range(args.num_heads)]
gat_out = [g(feature, aspect_feature, fmask).unsqueeze(1) for g in
self.gat]
gat_out = torch.cat(gat_out, dim=1)
gat_out = gat_out.mean(dim=1)
```

Multihead Attention

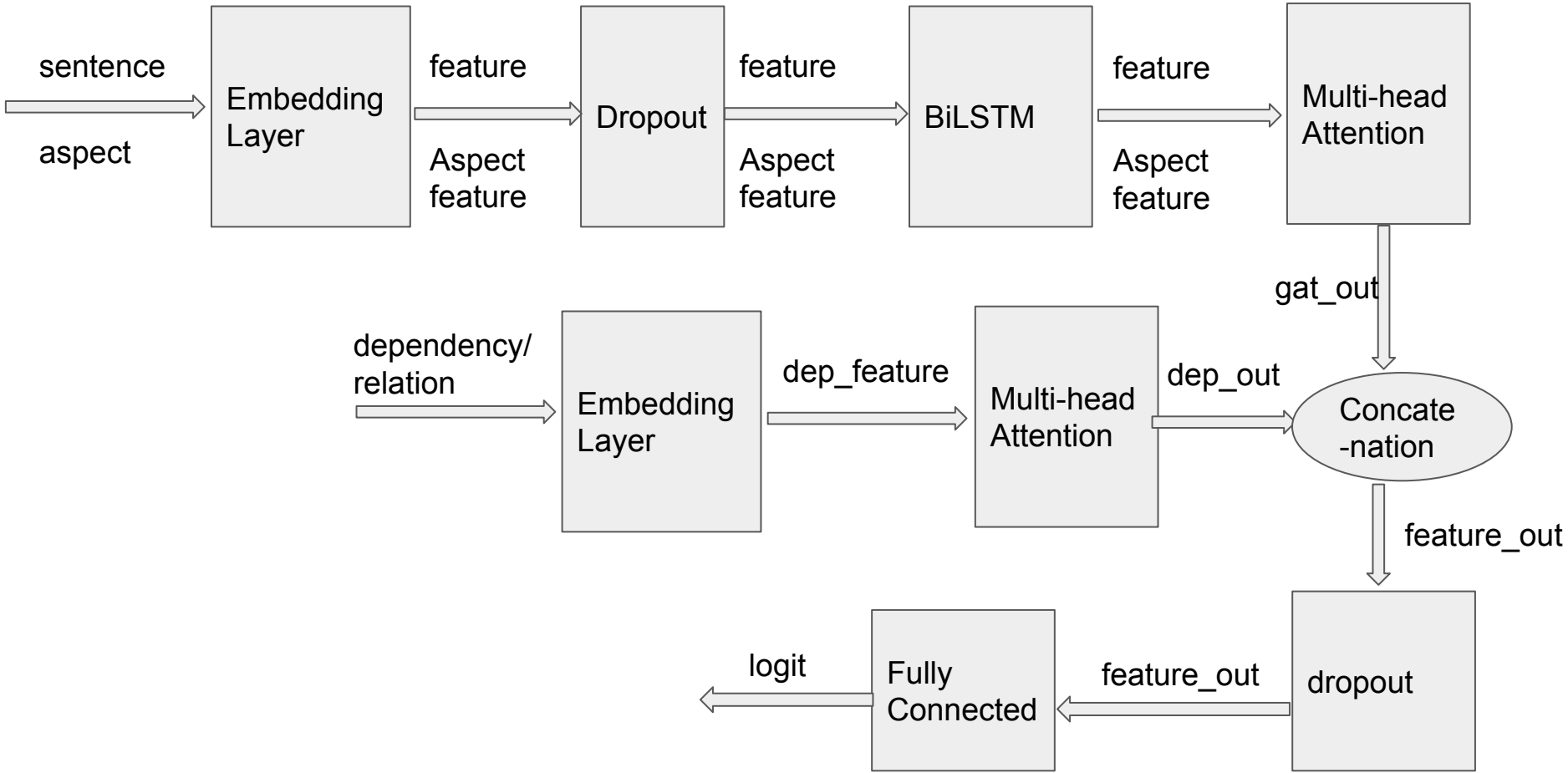
```
feature_out = gat_out # (N, D')
x = self.dropout(feature_out)
```

Dropout

```
x = self.fcs(x)
logit = self.fc_final(x)
```

Fully Connected

Aspect Text RGAT



```
dep_feature = self.dep_embed(dep_tags)
```

Embedding for dep

```
dep_out = [g(feature, dep_feature, fmask).unsqueeze(1) for g in
self.gat_dep]  #(N, 1, D)*num_heads
dep_out = torch.cat(dep_out, dim = 1) # (N, H, D)
dep_out = dep_out.mean(dim = 1) # (N, D)
```

Multihead Attention for dep

```
feature_out = torch.cat([dep_out,  gat_out], dim = 1) # (N, D')
```

Concatenation

```
x = self.dropout(feature_out)
```

Dropout

```
x = self.fcs(x)
logit = self.fc_final(x)
```

Fully Connected

Aspect Bert Only

BERT : Bidirectional Encoder Representations from Transformers

Bert is used for sequence classification.

Bert class constructor takes : args(argument) and Hidden_size as parameter.

Bert class have a forward(input_ids, token_type_ids) method.

Constructor

```
  def __init__(self, args, hidden_size=256):
       super(Pure_Bert, self).__init__()


       config = BertConfig.from_pretrained(args.bert_model_dir) //Instantiate the Bert Model
       self.tokenizer = BertTokenizer.from_pretrained(args.bert_model_dir) //Construct a Bert tokenizer.
       self.bert = BertModel.from_pretrained(
           args.bert_model_dir, config=config)    // creating the Bert model.


       self.dropout = nn.Dropout(config.hidden_dropout_prob)  //Defining the dropout parameter of the model.
       layers = [nn.Linear(
           config.hidden_size, hidden_size), nn.ReLU(), nn.Linear(hidden_size, args.num_classes)] //Applies a linear
transformation to the incoming data.
       self.classifier = nn.Sequential(*layers) //collection of all the layers created above.
```

forward():

```
def forward(self, input_ids, token_type_ids):
       outputs = self.bert(input_ids, token_type_ids=token_type_ids)
       pooled_output = outputs[1]
       pooled_output = self.dropout(pooled_output)
       logits = self.classifier(pooled_output)
       return logits
```

This function will take two arguments : input_ids and token_type_ids.These inputs are provided to the bert model and bert model will give two outputs. We take the second output that is pooled output. And then we pass this pooled_output to our classifier which is created in constructor. and Return the result which is a logistic regression.

Aspect Bert GAT

In this method we use R-GAT with Bert.

This class also has almost same method and functionality with some additional work.

This class contains :

Constructor.

Forward():

rnn_zero_state

Constructor:

Constructor of this class take 2 arguments : dep_tag_num,pos_tag_num

```
self.gat_dep = [RelationAttention(in_dim=args.embedding_dim).to(args.device) for i in range(args.num_heads)]
       self.dep_embed = nn.Embedding(dep_tag_num, args.embedding_dim)
       last_hidden_size = args.embedding_dim * 2
       layers = [nn.Linear(last_hidden_size, args.final_hidden_size), nn.ReLU()]
       for _ in range(args.num_mlps - 1):
           layers += [nn.Linear(args.final_hidden_size,args.final_hidden_size), nn.ReLU()]
       self.fcs = nn.Sequential(*layers)
       self.fc_final = nn.Linear(args.final_hidden_size, args.num_classes)
```

We first get all the dependencies from the relation attention using the num_heads. And after that we perform dependency embedding. Then again we make a linear classifier on hidden layer. And then finally make a classifier.

forward():

We use both the outputs from the bert model and that collectively is named as feature output

Using that feature output we make a logistic regression and return that.

```
feature_out = torch.cat([dep_out,  pool_out], dim=1)  # (N, D')
       # feature_out = gat_out
       ###############################################################################################
       x = self.dropout(feature_out)
       x = self.fcs(x)
       logit = self.fc_final(x)
       return logit
```

rnn_zero_state()

```
def rnn_zero_state(batch_size, hidden_dim, num_layers, bidirectional=True, use_cuda=True):
   total_layers = num_layers * 2 if bidirectional else num_layers
   state_shape = (total_layers, batch_size, hidden_dim)
   h0 = c0 = Variable(torch.zeros(*state_shape), requires_grad=False)
   if use_cuda:
       return h0.cuda(), c0.cuda()
   else:
       return h0, c0
```

This method is used only to define the zeroth state of rnn.