Avataar AI

Documentation for 'Word Embeddings from BERT':

You can find the code at (Private Repository)
https://github.com/avataarapp/Transformers/blob/master/BERT/BERT_WordEmbeddingsPipeline.py

```
1   import torch
2   import numpy as np
3
4   # !pip install pytorch-pretrained-bert
5   from pytorch_pretrained_bert import BertTokenizer, BertModel
6   tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
7
```

1. Install PyTorch using "`!pip install torch`".
2. Import the required libraries (PyTorch and Numpy).
3. Install pretrained transformer model (BERT) using "`!pip install pytorch-pretrained-bert`", if not installed previously.
4. Import BertTokenizer and BERT Model from the pretrained models.
5. Load Pretrained model Tokenizer (Vocabulary).

```
8   # Load pre-trained model (weights)
9   # Put the model in "evaluation" mode, meaning feed-forward operation.
10  model = BertModel.from_pretrained('bert-base-uncased')
11  model.eval()
12
```

6. Load the model with pretrained weights.
7. Put the model into evaluation mode. (Here we are not training the model)

```
12
13    def embeddingsPipeline(text):
```

8. A function that takes a cleaned sentence as input and returns the embeddings for each word in the sentence as a list.

BERT is a pretrained model that expects input data in a specific format. We will need:

- Special token **[CLS]** and **[SEP]** to mark the beginning and end of our sentence respectively.
- Tokens that conform with the fixed vocabulary used in BERT.
- **Token ID**s for the tokens, from BERT Tokenizer.
- **Mask ID**s to indicate which elements in the sequence are tokens and which are padding elements. (BERT require sentence of fixed number of tokens, so we will pad the sentence and mask them)
- **Segment ID**s to distinguish different sentences.

```
14
15        # Add the special tokens.
16        marked_text = "[CLS] " + text + " [SEP]"
17
18        # Split the sentence into tokens.
19        tokenized_text = tokenizer.tokenize(marked_text)
```

9. Special tokens **[CLS]** and **[SEP]** are added to the sentence.
10. Tokenize the sentence into tokens.

```
text = "Here is the sentence I want embeddings for."
marked_text = "[CLS] " + text + " [SEP]"


# Tokenize our sentence with the BERT tokenizer.
tokenized_text = tokenizer.tokenize(marked_text)

# Print out the tokens.
print (tokenized_text)
```

```
['[CLS]', 'here', 'is', 'the', 'sentence', 'i', 'want', 'em', '##bed', '##ding', '##s', 'for', '.', '[SEP]']
```

```
"""
The original word has been split into smaller subwords and characters.
The two hash signs preceding some of these subwords are just our tokenizer's way to
denote that this subword or character is part of a larger word and preceded by another subword.
"""

"""
After breaking the text into tokens,
we then have to convert the sentence from a list of strings to a list of vocabulary indeces.
"""
```

```
20
21        # Map the token strings to their vocabulary indeces.
22        indexed_tokens = tokenizer.convert_tokens_to_ids(tokenized_text)
23
24        # Mark each of the tokens as belonging to sentence "1".
25        segments_ids = [1] * len(tokenized_text)
```

11. Get all the indices of the tokens obtained from the tokenizer.
12. Create an array of 1's for Segment IDs (Here we are providing only one sentence as text, else alternate sentences will have 1s for all tokens of one sentence and 0s for all tokens of next sentence and then again pattern repeats).

```
31      # Predict hidden states features for each layer
32      with torch.no_grad():
33          encoded_layers, _ = model(tokens_tensor, segments_tensors)
34
35      """
36      encoded_layers object has four dimensions, in the following order:
37          1. The layer number (12 layers)
38          2. The batch number (1 sentence)
39          3. The word / token number (no. of tokens in our sentence) (let it be 'N')
40          4. The hidden unit / feature number (768 features)
41      """
```

13. 'torch.no_grad' tells PyTorch not to construct the compute graph during this forward pass (since we won't be running backprop here)–this just reduces memory consumption and speeds things up a little.

```
42
43      # Concatenate the tensors for all layers.
44      # We use `stack` here to create a new dimension in the tensor.
45      token_embeddings = torch.stack(encoded_layers, dim=0)
46      # Dimensions of token_embeddings = [12 x 1 x N x 768]
47
48      # Remove dimension 1, the "batches".
49      token_embeddings = torch.squeeze(token_embeddings, dim=1)
50      # Dimensions of token_embeddings = [12 x N x 768]
51
52      # Swap dimensions 0 and 1.
53      token_embeddings = token_embeddings.permute(1,0,2)
54      # Dimensions of token_embeddings = [N x 12 x 768]
55
```

Note: 'N' in the above code denotes the number of tokens in the input sentence.
14. Concatenate the tensors from all 12 layers and change the dimensions as required.

```
    # Creating the word vectors by summing together the last four layers.

    token_vecs_sum = []
    # Dimensions of token_vecs_sum will be [N x 768]

    for token in token_embeddings:
        # `token` is a [12 x 768] tensor
        # Sum the vectors from the last four layers.
        sum_vec = torch.sum(token[-4:], dim=0)

        token_vecs_sum.append(sum_vec)

```

15. Studies for Named Entity Recognition have shown that concatenation of the last four layers produced the best results on this specific task. Have a look at http://jalammar.github.io/images/bert-feature-extraction-contextualized-embeddings.png

16. Create a list which stores the embeddings for tokens, by summing up the tensors from the last four layers.

```

    # Converting embeddings into numpy array for further processing.
    numpy_embeddings = []
    for embedding in token_vecs_sum:
        numpy_embeddings.append(embedding.numpy())


    # Now we have to deal with the split words.
    # Example: 'embeddings' is tokenized into ['em', '##bed', '##ding', '##s']
    # We will average the embeddings of split words to get the embeddings of the original word.

    # Put all split-word-embeddings (if any) of every original word in a list 'p'.
    # If the word is not a split word, make a list with one elment i.e,
    # the respective word-embedding and append this list to 'p'.
    p = []
    for i,token in enumerate(tokenized_text):
        if token[0] == '#' and token[1] == '#':
            p[-1].append(numpy_embeddings[i])
        else:
            p.append([numpy_embeddings[i]])

```

17. Convert extracted embeddings into numpy arrays, and take the average of the sub tokens (that are starting with '##') to get the final word embedding.

```
94
95      # Averaging the embeddings of split words to obtain the embedding of original word.
96      final_embeddings = []
97      for i in p:
98          temp = np.mean( np.array([l for l in i]), axis=0 )
99          final_embeddings.append(temp)


100
101     # Remove first and last embeddings (i.e., Embeddings for "[CLS]" and "[SEP]")
102     final_embeddings = final_embeddings[1:-1]
103
104     return final_embeddings
```

18. Remove the embeddings for special tokens (these tokens may be necessary during the further process in the machine learning model, so this step may not be performed if required) and return the final embeddings for each word in the sentence.

==========
==========

Documentation for 'Fine Tuning BERT for sequence to sequence task':

You can find the code at (Private Repository)
https://github.com/avataarapp/Models/blob/master/FineTuning_BERT/pretrainedBERT.ipynb

```
In [1]: from google.colab import drive
        drive.mount('/content/drive')

        Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id=947318989803-6bn6qk8qdgf4
        n4g3pfee6491hc0brc4i.apps.googleusercontent.com&redirect_uri=urn%3aietf%3awg%3aoauth%3a2.0%3aoob&response_
        type=code&scope=email%20https%3a%2f%2fwww.googleapis.com%2fauth%2fdocs.test%20https%3a%2f%2fwww.googleapi
        s.com%2fauth%2fdrive%20https%3a%2f%2fwww.googleapis.com%2fauth%2fdrive.photos.readonly%20https%3a%2f%2fww
        w.googleapis.com%2fauth%2fpeopleapi.readonly

        Enter your authorization code:
        ··········
        Mounted at /content/drive

In [2]: cd /content/drive/My Drive/bert

        /content/drive/My Drive/bert
```

1. Mount the google drive (avataar.intern@gmail.com) in google colab and navigate to the folder named 'bert'.

In [3]:
```
import torch
import torch.nn as nn

import pandas as pd
import numpy as np
import time

! pip install transformers
from transformers import BertModel, BertTokenizer
bert_model = BertModel.from_pretrained('bert-base-uncased')
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')

from torch.utils.data import Dataset
from torch.utils.data import DataLoader
```

```
Collecting transformers
```

2. Import required libraries and modules (install transformers library, if not yet installed)
3. Load pre trained BERT model 'bert-base-uncased' and BERT Tokenizer.

In [0]:
```
class MakeDataset(Dataset):

    def __init__(self, filename, maxlen):

        self.df = pd.read_csv(filename)

        # Reducing data size for faster testing
        size = 2000
        if(len(self.df) > size):
            self.df = self.df[:size]

        self.tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
        self.maxlen = maxlen

    def __len__(self):
        return len(self.df)

    def __getitem__(self, index):

        sentence = self.df.loc[index, 'tokenized_sents']

        tokens = self.tokenizer.tokenize(sentence)
        tokens = ['[CLS]'] + tokens + ['[SEP]']
        if len(tokens) < self.maxlen:
            tokens = tokens + ['[PAD]' for _ in range(self.maxlen - len(tokens))]
        else:
            tokens = tokens[:self.maxlen-1] + ['[SEP]']

        tokens_ids = self.tokenizer.convert_tokens_to_ids(tokens)
        tokens_ids_tensor = torch.tensor(tokens_ids)

        #Obtaining the attention mask i.e a tensor containing 1s for no padded tokens and 0s for padded on
es
        attn_mask = (tokens_ids_tensor != 0).long()

        return tokens_ids_tensor, attn_mask
```

4. Make a class named 'MakeDataset' which will be used to make datasets for training and validation. When we create an instance of this MakeDataset, we will be passing the path to the data file and the maximum numbers of tokens in a sentence.

5. We will read the data which is in csv format using pandas read_csv method. If the number of samples in the dataset are more than 2000, we are reducing them to 2000 inorder to reduce the training time during model testing. We have to remove this part during original training on the cloud.
6. Then we are tokenizing the sentences from the dataset into tokens using BERT tokenizer and then add special tokens "[CLS]" and "[SEP]" at the beginning and end of each sentence.
7. If the number of tokens in a sentence is less than the maximum number of tokens, we will add the padding tokens at the end to make the number of tokens equal to the maximum number of tokens.
8. Else if the number of tokens in a sentence is greater than or equal to maximum number of tokens, we will discard some tokens in the add so that the final number of tokens will become the maximum number of tokens after adding the special "[SEP]" token.
9. Then we will convert the tokens to indices using a predefined (in bert tokenizer) dictionary of words and indices.
10. We then prepare a list as an attention mask, i.e, the element would be 0 if the token present at the corresponding index in tokens list else it would be 1.
11. We then convert these token-ids and the attention mask as tensors.

In [0]:
```
MAX_LEN = 50
BATCH_SIZE = 2
BERT_VOCAB_SIZE = len(list(tokenizer.vocab.keys()))

train_set = MakeDataset(filename = 'train.csv', maxlen = MAX_LEN)
val_set = MakeDataset(filename = 'val.csv', maxlen = MAX_LEN)

train_loader = DataLoader(train_set, batch_size = BATCH_SIZE, drop_last=True)
val_loader = DataLoader(val_set, batch_size = BATCH_SIZE, drop_last=True)
```

12. We store the maximum number of tokens in a sentence, batch size and the vocabulary size of BERT in variables MAX_LEN, BATCH_SIZE, BERT_VOCAB_SIZE respectively.
13. We then create the data loaders for iterating through the training and validation datasets.

In [0]:
```
class Seq2Seq(nn.Module):

    def __init__(self, max_len, bert_vocab_size, freeze_bert = True):
        super(Seq2Seq, self).__init__()
        self.bert_layer = BertModel.from_pretrained('bert-base-uncased')
        self.bert_vocab_size = bert_vocab_size

        if freeze_bert:
            for p in self.bert_layer.parameters():
                p.requires_grad = False

        self.fc1 = nn.Linear(768, self.bert_vocab_size)
        self.soft_max = nn.Softmax(dim=2)

    def forward(self, seq, attn_masks):
        encoded_layers, _ = self.bert_layer(seq, attention_mask = attn_masks)
        f1 = self.fc1(encoded_layers)
        output = self.soft_max(f1)
        return output
```

14. We create the model Seq2Seq which will take MAX_LEN, BERT_VOCAB_SIZE and a boolean for freezing model parameters, during the instantiation of the model.
15. First part of the model would be the pre-trained BERT which we imported in the beginning. Then the output from this model will be passed through a fully connected linear layer.
16. We then pass the output through a softmax layer and return it.

```
In [0]: net = Seq2Seq(MAX_LEN, BERT_VOCAB_SIZE, freeze_bert=False)
```

17. Create an instance of the model.

```
In [0]: def train(net, train_loader, val_loader, num_epochs, batch_size, max_len, bert_vocab_size):

            device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
            net.to(device)

            criterion = nn.BCELoss()
            optimizer = torch.optim.Adam(net.parameters(), lr = 0.0002)
```

18. Create a train function for the fine tuning of the BERT model. Check for availability of GPU and move the model to GPU if available. Define criterion and optimizer.

```
for epoch in range(num_epochs):

    start_time = time.time()
    train_epoch_loss = 0
    net.train()
    for it, (seq, attn_masks) in enumerate(train_loader):

        seq = seq.to(device)
        attn_masks = attn_masks.to(device)

        output = net(seq, attn_masks)

        target = torch.zeros(batch_size, max_len, bert_vocab_size)
        for i in range(batch_size):
            for j in range(max_len):
                target[i][i][seq[i][j]] = 1

        output = output.to(device)
        target = target.to(device)

        # print(output.size())
        # print(target.size())

        optimizer.zero_grad()
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()

        train_epoch_loss += loss.item()
```

19. For each epoch, extract the token ids and attention masks from the data loader and obtain the output from the model.
20. Create target tensor from the token-ids i.e., one-hot encoding form and then pass the output and target to the loss criterion.
21. Back propagate the loss and update the weights of the model.

```
    val_epoch_loss = 0
    net.eval()
    for it, (seq, attn_masks) in enumerate(val_loader):
        seq = seq.to(device)
        attn_masks = attn_masks.to(device)

        output = net(seq, attn_masks)

        target = torch.zeros(batch_size, max_len, bert_vocab_size)
        for i in range(batch_size):
            for j in range(max_len):
                target[i][i][seq[i][j]] = 1

        output = output.to(device)
        target = target.to(device)

        # print(output.size())
        # print(target.size())

        loss = criterion(output, target)
        val_epoch_loss += loss.item()
```

22. Put the model into evaluation mode and follow the same steps for validation data except back propagation and updation of model weights.

```
        train_epoch_loss /= len(train_loader)
        val_epoch_loss /= len(val_loader)
        finish_time = time.time()
        time_taken = finish_time-start_time

        print("Epoch[{:02}/{:02}]: Train_loss={} Validation_loss={} Time: {:.3f}" .format(epoch+1, num_epo
chs, train_epoch_loss, val_epoch_loss, time_taken))
```

23. Calculate and print out the required items such as epoch number, train loss, validation loss and time taken for the epoch.

In [9]:
```
NUM_EPOCHS = 10
train(net, train_loader, val_loader, NUM_EPOCHS, BATCH_SIZE, MAX_LEN, BERT_VOCAB_SIZE)
```

```
Epoch[01/10]: Train_loss=0.0001767641440455918 2 Validation_loss=0.000171609992849127 Time: 106.078
Epoch[02/10]: Train_loss=0.000166638184520707 Validation_loss=0.00016700179073369676 Time: 106.411
Epoch[03/10]: Train_loss=0.00015955338733328973 Validation_loss=0.00016398020755353405 Time: 106.643
Epoch[04/10]: Train_loss=0.00015226766893465537 Validation_loss=0.00016255338992319336 Time: 106.907
Epoch[05/10]: Train_loss=0.00014491071562224535 Validation_loss=0.00016216104028533035 Time: 106.469
Epoch[06/10]: Train_loss=0.00013752039645623882 Validation_loss=0.0001620276135252418 Time: 106.696
Epoch[07/10]: Train_loss=0.00013038430753294962 Validation_loss=0.00016340724919763628 Time: 107.334
Epoch[08/10]: Train_loss=0.00012341415107221 11 Validation_loss=0.00016422183873954614 Time: 106.470
Epoch[09/10]: Train_loss=0.00011768421006127028 Validation_loss=0.00016544819161695722 Time: 107.185
Epoch[10/10]: Train_loss=0.00011382855837655 37 Validation_loss=0.00016658889179665428 Time: 107.053
```

24. Train the model for the required number of epochs.

In [16]:
```
for it, (seq, attn_masks) in enumerate(train_loader):
    s1 = seq
    a1 = attn_masks
    break

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
a1 = a1.to(device)
s1 = s1.to(device)

output = net(s1, a1)
print("Dimensions of output : ", output.size())

example = output[0]
print("Dimensions of example: ", example.size())

result = tokenizer.convert_ids_to_tokens(torch.argmax(example, 1))
len(result)

original = tokenizer.convert_ids_to_tokens(s1[0])
```

```
Dimensions of output :  torch.Size([2, 50, 30522])
Dimensions of example:  torch.Size([50, 30522])
```

25. Test with an example how the sentences are being produced from the model. First get the output from the model for a batch and consider one example from the batch.
26. Since the output was last passed through a softmax layer, the outputs would be probability for occurrence of each word. We will consider the highest probable word.

In [17]:
```
print(original)
print(result)
```

```
['[CLS]', 'it', 'never', 'once', 'occurred', 'to', 'me', 'that', 'the', 'fu', '##mbling', 'might', 'be',
'a', 'mere', 'mistake', '[SEP]', '[PAD]', '[PAD]', '[PAD]', '[PAD]', '[PAD]', '[PAD]', '[PAD]', '[PAD]',
'[PAD]', '[PAD]', '[PAD]', '[PAD]', '[PAD]', '[PAD]', '[PAD]', '[PAD]', '[PAD]', '[PAD]', '[PAD]', '[PA
D]', '[PAD]', '[PAD]', '[PAD]', '[PAD]', '[PAD]', '[PAD]', '[PAD]', '[PAD]', '[PAD]', '[PAD]', '[PAD]',
'[PAD]', '[PAD]']
['to', 'me', 'to', 'to', 'a', 'one', 'be', 'to', 'that', 'to', 'to', 'to', 'one', '[CLS]', 'to', 'to', 'mi
ght', 'prove', 'time', 'imagine', 'prove', 'imagine', '##ably', 'imagine', 'prove', 'prove', 'prove', 'pro
ve', 'imagine', 'prove', 'prove', '##ably', '##ably', '##ably', 'imagine', '##ably', '##ably', '##ably',
'prove', '##ably', 'prove', 'prove', '##ably', 'prove', 'imagine', '##ably', 'prove', '##ably', '##ably',
'prove']
```

27. Print out the original sentence and the output sentence produced by the model.

Documentation for 'Generative Adversarial Network (GAN) ':

You can find the code at (Private Repository)
https://github.com/avataarapp/Models/tree/master/GAN

File 'Encoder.py':
https://github.com/avataarapp/Models/blob/master/GAN/Encoder.py

```python
import torch
import torch.nn as nn
import numpy as np

class Encoder(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers):
        super(Encoder, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.num_layers = num_layers

        self.dropout = nn.Dropout(p=0.2)
        self.lstm = nn.LSTM(input_size, hidden_size, num_layers, batch_first=True, bidirectional=True)
        self.lrelu = nn.LeakyReLU()

        # initializing weights
        nn.init.xavier_normal(self.lstm.weight_ih_l0, gain=np.sqrt(2))
        nn.init.xavier_normal(self.lstm.weight_hh_l0, gain=np.sqrt(2))

    def forward(self, input):
        input = self.dropout(input)
        encoded_input, hidden = self.lstm(input)
        encoded_input = self.lrelu(encoded_input)
        return encoded_input
```

1. Encoder model will take 'input_size' (size of embeddings), 'hidden_size' (latent dimension) and the 'num_layers'(number of lstm layers in encoder) during its instantiation.
2. During forward pass, the embeddings move through a dropout layer. (This is useful for masking some words in the sentence).
3. Then the masked sentence passes through the lstm layers and gives the output of size twice the 'hidden_size'. We are enabling the 'bidirectional' parameter to true in lstm layers. This is the reason we are obtaining the output of dimension twice the 'hidden_dimension'.
4. We then pass the obtained output through the 'leakyReLU' layer which serves as an activation function.

File 'Decoder.py':
https://github.com/avataarapp/Models/blob/master/GAN/Decoder.py

```python
import torch
import torch.nn as nn
import numpy as np

class Decoder(nn.Module):
    def __init__(self, hidden_size, output_size, num_layers):
        super(Decoder, self).__init__()
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.num_layers = num_layers

        self.lstm = nn.LSTM(hidden_size*2, output_size, num_layers, batch_first=True, bidirectional=True)
        self.fc1 = nn.Linear(output_size*2, output_size)

        # initializing weights
        nn.init.xavier_normal(self.lstm.weight_ih_l0, gain=np.sqrt(2))
        nn.init.xavier_normal(self.lstm.weight_hh_l0, gain=np.sqrt(2))

    def forward(self, encoded_input):
        output, hidden = self.lstm(encoded_input)
        decoded_output = self.fc1(output)
        return decoded_output
```

5. Decoder follows the similar architecture of Encoder with some minor changes.
6. Decoder requires 'hidden_size', 'output_size' (size of embeddings) and 'num_layers' during its instantiation.
7. Here the output from the encoder will be passed through the decoder lstm layers and the outputs of dimension twice the required size are obtained. (Since the bidirectional is set to true.)
8. Hence we pass the output through a linear fully connected layer to obtain embeddings of required dimension and return them.

File 'noTeacherForcingGenerator.py':
https://github.com/avataarapp/Models/blob/master/GAN/noTeacherForcingGenerator.py

```python
import torch
import torch.nn as nn

class Generator(nn.Module):
    def __init__(self, encoder, decoder):
        super(Generator, self).__init__()
        self.encoder = encoder
        self.decoder = decoder

    def forward(self, input):
        encoded_input, original_input = self.encoder(input)
        decoded_output = self.decoder(encoded_input)
        return decoded_output
```

9. noTeacherForcingGenerator takes encoder and decoder as parameters during instantiation and as the name says it generates embeddings without any Teacher forcing implementation.
10. The input to the generator will be passed through the encoder and then the encoded input will be passed through the decoder and then we return the generated embeddings.

File 'Generator.py' :
https://github.com/avataarapp/Models/blob/master/GAN/Generator.py

```
class Generator(nn.Module):
    def __init__(self, encoder, decoder, max_words, hidden_size, embedding_length):
        super(Generator, self).__init__()
        self.encoder = encoder
        self.decoder = decoder
        self.max_words = max_words
        self.hidden_size = hidden_size
        self.emb_length = embedding_length

    def forward(self, input, teacher_forcing_ratio=0.5):
        encoded_input, origianl_input = self.encoder(input)

        batch_size = encoded_input.size(0)
        outputs = torch.zeros(batch_size, self.max_words, self.emb_length)

        for batch_index, batch in enumerate(encoded_input):
            decoded = torch.zeros(self.max_words, self.emb_length)

            input = batch[0]

            for t in range(1, batch_size):

                hidden_embedding = batch[t]
                decoder_out = self.decoder(hidden_embedding.view(1,1,self.hidden_size))

                decoded[t] = decoder_out.view(self.emb_length)
                teacher_force = random.random() < teacher_forcing_ratio

                if(teacher_force):
                    input = origianl_input[batch_index][t]
                else:
                    input = decoder_out

            outputs[batch_index] = decoded
        return outputs
```

11. This is the Generator which is implementing the teacher forcing method. At the time of instantiation, in addition to encoder and decoder, this version of generator will also take 'max_words' (maximum number of tokens in a sentence), 'hidden_size' (dimension of latent representation), and 'embedding_lenght'.
12. Here we are passing the embedding of each word as input to the decoder to determine the next embedding. Depending on the teacher forcing ratio we will decide whether to send the last

obtained output or the corresponding original word embedding, as input to find the next embedding.
13. Since we are passing word by word to the decoder rather than the entire batch of sentences, this method consumes a very large amount of time during training (approximately 1 second for each sentence.

File 'Discriminator.py' :
https://github.com/avataarapp/Models/blob/master/GAN/Discriminator.py

```python
import torch
import torch.nn as nn

class Discriminator(nn.Module):
    def __init__(self, input_size, num_layers, max_words, batch_size):
        super(Discriminator, self).__init__()
        self.lstm = nn.LSTM(input_size, 1, num_layers, batch_first=True)
        self.fc1 = nn.Linear(batch_size*max_words, batch_size)
        self.sigmoid = nn.Sigmoid()

    def forward(self, input):
        output, hidden = self.lstm(input)
        output = torch.flatten(output)
        output = self.fc1(output)
        output = self.sigmoid(output)
        return output
```

14. Discriminator will be trained to distinguish between real and generated embeddings.
15. Our discriminator will be taking 'input_size' (size of word embeddings), 'num_layers', 'max_words' (maximum number of tokens in a sentence) and 'batch_size' as parameters during the instantiation.
16. The input to the discriminator will be a batch of embeddings. They will be passed through lstm layers and then through a fully connected linear layer of outsize equal to batch size.
17. The output obtained will be passed through a sigmoid layer so that the values would be restricted between 0 and 1. (0 denotes a fake sentence and 1 denotes a real sentence)

Implementation and training:
https://github.com/avataarapp/Models/blob/master/GAN/T5.ipynb

18. Mount the google drive in colab and navigate to the folder containing the train and test datasets.

19. Install pytorch_pretrained_bert module, if not yet installed.

```
In [4]: import torch
        import torch.nn as nn

        import spacy
        import numpy as np
        import pandas as pd

        import random
        import math
        import time

        from BERT_WordEmbeddingsPipeline import embeddingsPipeline
        from sklearn.model_selection import train_test_split

        import warnings
        warnings.filterwarnings("ignore")

        SEED = 25

        random.seed(SEED)
        np.random.seed(SEED)
        torch.manual_seed(SEED)
        torch.cuda.manual_seed(SEED)
        torch.backends.cudnn.deterministic = True
        torch.backends.cudnn.benchmark = False
```

```
100%|██████████| 231508/231508 [00:00<00:00, 259888.44B/s]
100%|██████████| 407873900/407873900 [00:41<00:00, 9931739.76B/s]
```

20. Import the required libraries and modules along with the 'embeddingsPipeline' function which was created by us previously. Set all the seeds to a specific number so that the results can be reproduced.

```
In [0]: data = pd.read_csv('hpl.csv')
        data = data[:3500]

        embedding_length = 768
        max_words = 100

        zero_embedding = [0 for i in range(embedding_length)]

        emb = []
        for i in data['tokenized_sents']:
            e = embeddingsPipeline(i)
            while(len(e) < max_words):
                e.append(zero_embedding)
            e = e[:max_words]
            emb.append(e)
        data['embeddings'] = emb

        train_data = data
```

```
In [0]: data = pd.read_csv('eap.csv')
        data = data[:100]

        embedding_length = 768
        max_words = 100

        zero_embedding = [0 for i in range(embedding_length)]

        emb = []
        for i in data['tokenized_sents']:
            e = embeddingsPipeline(i)
            while(len(e) < max_words):
                e.append(zero_embedding)
            e = e[:max_words]
            emb.append(e)
        data['embeddings'] = emb

        test_data = data
```

21. Import the train and test datasets from the csv file and obtain the embeddings for each word in the sentence.
22. If the number of embeddings in a sentence is less than the maximum number of tokens, the append zero embedding as padding, so that the total number of tokens become the maximum number of tokens.
23. If they are greater than the maximum number of tokens, then discard the ending embeddings so that the number of embeddings is equal to the maximum number of tokens.

In [7]:
```
print(train_data.shape)
print(test_data.shape)
```

```
(3500, 2)
(100, 2)
```

In [0]:
```
INPUT_SIZE = embedding_length
HIDDEN_SIZE = 512
NUM_LAYERS = 2
OUTPUT_SIZE = embedding_length
BATCH_SIZE = 32
```

In [0]:
```
train_emb = torch.Tensor(train_data['embeddings'])
train_dataSet = torch.utils.data.TensorDataset(train_emb)
train_dataLoader = torch.utils.data.DataLoader(train_dataSet, batch_size=BATCH_SIZE, drop_last=True)
```

In [0]:
```
test_emb = torch.Tensor(test_data['embeddings'])
test_dataSet = torch.utils.data.TensorDataset(test_emb)
test_dataLoader = torch.utils.data.DataLoader(test_dataSet, batch_size=BATCH_SIZE, drop_last=True)
```

24. Look at the size of datasets.
25. Declare the parameters such as input size, hidden size, number of layers and batch size.
26. Convert the training and test embeddings into the tensors and load them into the data loaders with the help of 'TensorDataset'. Make sure that drop_last parameter is set to true.

In [0]:
```
from Encoder import Encoder
from Decoder import Decoder
from noTeacherForcingGenerator import Generator
from Discriminator import Discriminator
```

In [12]:
```
enc = Encoder(INPUT_SIZE, HIDDEN_SIZE, NUM_LAYERS)
dec = Decoder(HIDDEN_SIZE, OUTPUT_SIZE, NUM_LAYERS)
# gen = Generator(enc, dec, max_words, HIDDEN_SIZE, embedding_length)
gen = Generator(enc, dec)
print(gen)

dis = Discriminator(INPUT_SIZE, NUM_LAYERS, max_words)
print(dis)
```

```
Generator(
  (encoder): Encoder(
    (dropout): Dropout(p=0.2, inplace=False)
    (lstm): LSTM(768, 512, num_layers=2, batch_first=True, bidirectional=True)
    (fc1): Linear(in_features=1024, out_features=512, bias=True)
    (lrelu): LeakyReLU(negative_slope=0.01)
  )
  (decoder): Decoder(
    (lstm): LSTM(512, 768, num_layers=2, batch_first=True, bidirectional=True)
    (fc1): Linear(in_features=1536, out_features=768, bias=True)
  )
)
Discriminator(
  (lstm): LSTM(768, 1, num_layers=2, batch_first=True)
  (fc1): Linear(in_features=100, out_features=1, bias=True)
  (sigmoid): Sigmoid()
)
```

27. Import and instantiate Encoder, Decoder, Generator and Discriminator.

In [0]:
```
def train(dataLoader, test_dataLoader, gen, dis, num_epochs, max_words, batch_size, embedding_length):
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    gen.to(device)
    dis.to(device)

    criterion = nn.BCELoss()
    d_optimizer = torch.optim.Adam(dis.parameters(), lr=0.0002)
    g_optimizer = torch.optim.Adadelta(gen.parameters(), lr=0.0004)

    for epoch in range(num_epochs):
        start_time = time.time()

        dis.train()
        gen.train()

        real_labels = torch.ones(1, batch_size).to(device)
        fake_labels = torch.zeros(1, batch_size).to(device)

        #-------------------------------------------------#
        # Training Discriminator
        #-------------------------------------------------#
        total_discriminator_loss = 0
        iterator = iter(dataLoader)

        for i in range(len(dataLoader)):
            batch_data = next(iterator)
            # batch_data is a one-element list containing a single batch in the form of Tensor.
            batch = batch_data[0]

            batch = batch.to(device)

            dis_output = torch.zeros(batch_size)
            for i, embedding in enumerate(batch):
                dis_output[i] = dis(embedding.view(1, max_words, embedding_length))

            dis_output = dis_output.to(device)
            d_loss_real = criterion(dis_output, real_labels)

            fake_batch = gen(batch)
            fake_batch = fake_batch.to(device)

            dis_output = torch.zeros(batch_size)
            for i, embedding in enumerate(fake_batch):
                dis_output[i] = dis(embedding.view(1, max_words, embedding_length))

            dis_output = dis_output.to(device)
            d_loss_fake = criterion(dis_output, fake_labels)

            d_loss = d_loss_real + d_loss_fake
            total_discriminator_loss += d_loss.item()

            d_optimizer.zero_grad()
            d_loss.backward()
            d_optimizer.step()

        #-------------------------------------------------#
        # Training Generator
        #-------------------------------------------------#
        total_generator_loss = 0
        iterator = iter(dataLoader)
```

```
#-----------------------------------------------------#
# Training Generator
#-----------------------------------------------------#
total_generator_loss = 0
iterator = iter(dataLoader)

for i in range(len(dataLoader)):
    batch_data = next(iterator)
    # batch_data is a one-element list containing a single batch in the form of Tensor.
    batch = batch_data[0]
    batch = batch.to(device)

    generated_batch = gen(batch)
    generated_batch = generated_batch.to(device)

    dis_output = torch.zeros(batch_size)
    for i, embedding in enumerate(generated_batch):
        dis_output[i] = dis(embedding.view(1, max_words, embedding_length))

    dis_output = dis_output.to(device)
    g_loss = criterion(dis_output, real_labels)
    total_generator_loss += g_loss.item()

    g_optimizer.zero_grad()
    g_loss.backward()
    g_optimizer.step()


#-----------------------------------------------------#
# On Test data
#-----------------------------------------------------#
dis.eval()
gen.eval()

generator_loss_on_test = 0
discriminator_loss_on_test = 0
iterator = iter(test_dataLoader)

for i in range(len(test_dataLoader)):
    batch_data = next(iterator)
    # batch_data is a one-element list containing a single batch in the form of Tensor.
    batch = batch_data[0]
    batch = batch.to(device)

    dis_output = torch.zeros(batch_size)
    for i, embedding in enumerate(batch):
        dis_output[i] = dis(embedding.view(1, max_words, embedding_length))

    dis_output = dis_output.to(device)
    d_loss = criterion(dis_output, real_labels)
    discriminator_loss_on_test += d_loss.item()

    generated_batch = gen(batch)
    generated_batch = generated_batch.to(device)
    dis_output = torch.zeros(batch_size)

    for i, embedding in enumerate(generated_batch):
        dis_output[i] = dis(embedding.view(1, max_words, embedding_length))

    dis_output = dis_output.to(device)
    g_loss = criterion(dis_output, real_labels)
    d_loss = criterion(dis_output, fake_labels)

    discriminator_loss_on_test += d_loss.item()
    generator_loss_on_test += g_loss.item()
```

28. Create a train function to train the model. Find the availability of gpu (device) and move the generator and discriminator to the device. Initialize loss criterion and optimizer for both generator and discriminator.

29. For each epoch and each batch, first put the generator and discriminator in training mode. Pass the original embeddings to the discriminator and find the loss by comparing with the real labels. Then generate a batch of embeddings using the generator and pass them to the discriminator. Then find the loss by comparing with the fake labels. Add both the losses and back propagate through the discriminator and update the weights.

30. Then generate a batch of embeddings using the generator and pass them to the discriminator. Find the loss by comparing output with the real labels, so that the generator is trained to produce more realistic embeddings and fool the discriminator. Back propagate the loss and update the weights of the generator.

31. Repeat the same process for test data except back propagating and updating the parameters.

```
        finish_time = time.time()
        avg_d_loss = total_discriminator_loss/len(dataLoader)
        avg_g_loss = total_generator_loss/len(dataLoader)
        d_loss_test = discriminator_loss_on_test/len(test_dataLoader)
        g_loss_test = generator_loss_on_test/len(test_dataLoader)
        time_taken = finish_time-start_time

        print("Epoch[{:02}/{}]: avg_d_Loss: {:.4f}, avg_g_Loss: {:.4f}, d_loss_test: {:.4f}, g_loss_test: {:.4f}, Time : {:.2f}"
              .format(epoch+1, num_epochs, avg_d_loss, avg_g_loss, d_loss_test, g_loss_test, time_taken))
```

```
In [14]: train(train_dataLoader, test_dataLoader, gen, dis, 10, max_words, BATCH_SIZE, embedding_length)

Epoch[01/10]: avg_d_Loss: 1.3741, avg_g_Loss: 0.7248, d_loss_test: 1.3586, g_loss_test: 0.7246, Time : 109.36
Epoch[02/10]: avg_d_Loss: 1.3182, avg_g_Loss: 0.7800, d_loss_test: 1.2804, g_loss_test: 0.7786, Time : 111.97
Epoch[03/10]: avg_d_Loss: 1.1840, avg_g_Loss: 0.8890, d_loss_test: 1.1338, g_loss_test: 0.8828, Time : 112.58
Epoch[04/10]: avg_d_Loss: 0.9925, avg_g_Loss: 1.0314, d_loss_test: 0.9765, g_loss_test: 1.0122, Time : 113.17
Epoch[05/10]: avg_d_Loss: 0.8112, avg_g_Loss: 1.1693, d_loss_test: 0.8450, g_loss_test: 1.1367, Time : 115.79
Epoch[06/10]: avg_d_Loss: 0.6723, avg_g_Loss: 1.2962, d_loss_test: 0.7375, g_loss_test: 1.2699, Time : 115.09
Epoch[07/10]: avg_d_Loss: 0.5620, avg_g_Loss: 1.4378, d_loss_test: 0.6421, g_loss_test: 1.4237, Time : 114.09
Epoch[08/10]: avg_d_Loss: 0.4677, avg_g_Loss: 1.6035, d_loss_test: 0.5548, g_loss_test: 1.5964, Time : 114.30
Epoch[09/10]: avg_d_Loss: 0.3874, avg_g_Loss: 1.7840, d_loss_test: 0.4800, g_loss_test: 1.7803, Time : 114.02
Epoch[10/10]: avg_d_Loss: 0.3212, avg_g_Loss: 1.9692, d_loss_test: 0.4160, g_loss_test: 1.9672, Time : 113.75
```

32. At the end of the train function, calculate and print the required criteria such as epoch number, average discriminator and generator loss on training and testing data.
33. Train the model for the required number of epochs.
34. There is another function named 'additional_generator_training' which can be used to provide extra training to the generator without training the discriminator. It is almost similar to the 'train' function except back propagating and updating weights of the discriminator. (Here in our model, the discriminator outperforms the generator, so we tried additional generator training to see if the results are improved.)

Documentation for 'Transformer based model ':

You can find the code at (Private Repository)
https://github.com/avataarapp/Models/blob/master/Transformer/Attention.ipynb

The code is almost similar to the code available at the https://github.com/bentrevett/pytorch-seq2seq/blob/master/6%20-%20Attention%20is%20All%20You%20Need.ipynb with small changes. Here I am explaining only about these changes. Complete explanation can be found here.

1. We have created a dataset containing two columns where the second column is nothing but the duplicate of the first column. We split this dataset into train, test and valid sets in csv format.

```
In [0]: spacy_en = spacy.load('en_core_web_sm')

        def tokenize_en(text):
            # Tokenizes English text from a string into a list of strings
            return [tok.text for tok in spacy_en.tokenizer(text)]


        SRC = Field(tokenize = tokenize_en,
                    init_token = '<sos>',
                    eos_token = '<eos>',
                    lower = True,
                    batch_first = True)

        TRG = Field(tokenize = tokenize_en,
                    init_token = '<sos>',
                    eos_token = '<eos>',
                    lower = True,
                    batch_first = True)
```

2. Both the SRC and TRG fields have the same tokenizer (english).

```
In [0]: fields = [('src', SRC), ('trg', TRG)]

        train_data = TabularDataset(
            path='train.csv',
            format='csv',
            fields=fields,
            skip_header=True
        )

        valid_data = TabularDataset(
            path='val.csv',
            format='csv',
            fields=fields,
            skip_header=True
        )

        test_data = TabularDataset(
            path='test.csv',
            format='csv',
            fields=fields,
            skip_header=True
        )
```

3. We load the datasets using the 'TabularDataset'.
We can try out with the BERT tokenizer instead of spacy tokenizer, by making a few changes in the above code. These changes are shown below:

In [3]:
```
# spacy_en = spacy.load('en_core_web_sm')
from pytorch_pretrained_bert import BertTokenizer
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')

def tokenize_en(text):
    # Tokenizes English text from a string into a list of strings
    return [tok for tok in tokenizer.tokenize(text)]


SRC = Field(tokenize = tokenize_en,
            init_token = '[CLS]',
            eos_token = '[SEP]',
            lower = True,
```

In [18]:
```
def translate_sentence(sentence, src_field, trg_field, model, tokenizer, device, max_len = 50):

    model.eval()

    if isinstance(sentence, str):
        tokens = [token.text.lower() for token in tokenizer.tokenize(sentence)]
    else:
        tokens = [token.lower() for token in sentence]

    tokens = [src_field.init_token] + tokens + [src_field.eos_token]
    src_indexes = [src_field.vocab.stoi[token] for token in tokens]
```

Documentation for 'BERT Consumption Time for word Embeddings ':

You can find the code at (Private Repository)
https://github.com/avataarapp/BERT-Testing/blob/master/BERT_pipelineTesting.py

```
t0 = time.time()

for text in corpus:
    # text = preprocess(text)
    # text = text[0]
    # print("Processed Text: {}" .format(text))

    # word_count = len(text.split())
    # print("word count: {}" .format(word_count))

    embeddings = embeddingsPipeline(text)
    # print("Number of Embeddings received: {}" .format(len(embeddings)))
    # print("-----------------------------")

t1 = time.time()
print("Time taken by BERT for all {} sentences: {:.3f} seconds" .format(len(corpus), t1-t0))
```

We will obtain the embeddings for each word in 100 sentences and print out the time taken for this task.

Output:

```
In [3]: runcell(0, 'C:/Users/chait/Learn PyTorch/Testing/BERT_pipelineTesting.py')
Time taken by BERT for all 100 sentences: 5.679 seconds
```

Machine details: CPU (intel i7 8th gen- 16GB RAM):