Open in Colab

Homework 3: Language Models, Contextual Embedding and BERT

In this homework, we will explore implementations of various language models we saw in lecture. We will explore BERT and measure perplexity.

Set Up

If you're opening this Notebook on colab, you will probably need to install Transformers. Make sure your version of Transformers is at least 4.11.0

In [ ]:
```
! pip3.7 install transformers
```

```
Collecting transformers
  Using cached transformers-4.26.1-py3-none-any.whl (6.3 MB)
Collecting huggingface-hub<1.0,>=0.11.0
  Using cached huggingface_hub-0.12.1-py3-none-any.whl (190 kB)
Collecting tokenizers!=0.11.3,<0.14,>=0.11.1
  Downloading tokenizers-0.13.2-cp37-cp37m-macosx_10_11_x86_64.whl (3.8 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 3.8/3.8 MB 20.1 MB/s eta 0:00:0000:0100:01
Requirement already satisfied: tqdm>=4.27 in /Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-packages (from transformers) (4.64.1)
Collecting regex!=2019.12.17
  Downloading regex-2022.10.31-cp37-cp37m-macosx_10_9_x86_64.whl (294 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 294.4/294.4 kB 10.0 MB/s eta 0:00:00
Requirement already satisfied: pyyaml>=5.1 in /Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-packages (from transformers) (6.0)
Requirement already satisfied: importlib-metadata in /Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-packages (from transformers) (4.12.0)
Requirement already satisfied: filelock in /Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-packages (from transformers) (3.0.12)
Requirement already satisfied: numpy>=1.17 in /Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-packages (from transformers) (1.21.6)
Requirement already satisfied: packaging>=20.0 in /Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-packages (from transformers) (21.3)
Requirement already satisfied: requests in /Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-packages (from transformers) (2.25.1)
Requirement already satisfied: typing-extensions>=3.7.4.3 in /Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-packages (from huggingface-hub<1.0,>=0.11.0
->transformers) (3.7.4.3)
Requirement already satisfied: pyparsing!=3.0.5,>=2.0.2 in /Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-packages (from packaging>=20.0->transformers)
(2.4.6)
Requirement already satisfied: zipp>=0.5 in /Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-packages (from importlib-metadata->transformers) (2.1.0)
Requirement already satisfied: idna<3,>=2.5 in /Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-packages (from requests->transformers) (2.10)
Requirement already satisfied: chardet<5,>=3.0.2 in /Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-packages (from requests->transformers) (4.0.0)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in /Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-packages (from requests->transformers) (1.26.3)
Requirement already satisfied: certifi>=2017.4.17 in /Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-packages (from requests->transformers) (2020.12.5)
Installing collected packages: tokenizers, regex, huggingface-hub, transformers
Successfully installed huggingface-hub-0.12.1 regex-2022.10.31 tokenizers-0.13.2 transformers-4.26.1

[notice] A new release of pip available: 22.3.1 -> 23.0.1
[notice] To update, run: pip3.7 install --upgrade pip
```

In [ ]:
```
import transformers
print(transformers.__version__)
```

```
4.26.1
```

IMPORTANT: For this assignment, GPU is not necessary. The following code block should show "Running on cpu". Go to Runtime > Change runtime type > Hardware accelerator > None if otherwise.

In [ ]:
```
import torch
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Running on {}".format(device))
```

```
Running on cpu
```

Masking

One of the core ideas to wrap your head around with transformer-based language models (and PyTorch) is the concept of *masking*---preventing a model from seeing specific tokens in the input during training.

- BERT training relies on the concept of *masked language modeling*: masking a random set of input tokens in a sequence and attempting to predict them. Remember that BERT is *bidirectional*, so that it can use all of the other non-masked tokens in a sentence to make that prediction.
- The GPT class of models acts as a traditional left-to-right language model (sometimes called a "causal" LM) . This family also uses self-attention based transformers---but, when making a prediction for the word w_i at position i, it can only use information about words $w_1, \ldots, w_{i-1}$ to do so. All of the other tokens following position $i-1$ must be *masked* (hidden from view).

Think about a mask as a matrix that's applied to every input w when generating an output o that determines whether an given o_i is allowed to access each token in w. For example, when passing a three-word input sequence through a transformer (to yield a three-word output sequence), a mask is a 3×3 matrix where the cells are essentially answering the following questions:

$$\begin{bmatrix} o_1 \text{ hide } w_1? & o_1 \text{ hide } w_2? & o_1 \text{ hide } w_3? \\ o_2 \text{ hide } w_1? & o_2 \text{ hide } w_2? & o_2 \text{ hide } w_3? \\ o_3 \text{ hide } w_1? & o_3 \text{ hide } w_2? & o_3 \text{ hide } w_3? \end{bmatrix}$$

In the masks we will consider below, 1 denotes that a position should be hidden; 0 denotes that it should be visible. Consider this mask:

$$\begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}$$

And consider this sequence:

$$\begin{bmatrix} \text{John} & \text{likes} & \text{dogs} \end{bmatrix}$$

When applying this mask to that sequence, we're saying that when we're generating the output for o_1 (*John*), we can only consider w_1 as an input (*John*). Likewise, when we generate the output for o_2 (*likes*), we can only consider w_2 as an input (*likes*), and so on. (This is a terrible mask! But illustrates what function a mask performs.)

The following code illustrates how this works for that particular mask.

In [ ]:
```
import numpy as np

def visualize_masking(sequences, mask):
  print(mask)
  for sequence in sequences:
    for i in range(len(sequence)):
      visible=[]
      for j in range(len(sequence)):
        if mask[i][j]==0:
          visible.append(sequence[j])
      print("for word %s, the following tokens are visible: %s" % (sequence[i], visible))
    print()
```

In [ ]:
```
sequences=[["This", "is", "a", "sentence", "that", "has", "exactly", "ten", "tokens", "."], ["Here's", "another", "sequence", "with", "10", "words", "like", "the", "last", ".

seq_length=len(sequences[0])

test_mask=np.ones((seq_length,seq_length))
for i in range(seq_length):
  test_mask[i,i]=0

visualize_masking(sequences, test_mask)
```

```
[[0. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 0. 1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 0. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 0. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 0. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 0. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 0. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 0. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1. 0. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1. 0.]]
for word This, the following tokens are visible: ['This']
for word is, the following tokens are visible: ['is']
for word a, the following tokens are visible: ['a']
for word sentence, the following tokens are visible: ['sentence']
for word that, the following tokens are visible: ['that']
for word has, the following tokens are visible: ['has']
for word exactly, the following tokens are visible: ['exactly']
for word ten, the following tokens are visible: ['ten']
for word tokens, the following tokens are visible: ['tokens']
for word ., the following tokens are visible: ['.']

for word Here's, the following tokens are visible: ["Here's"]
for word another, the following tokens are visible: ['another']
for word sequence, the following tokens are visible: ['sequence']
for word with, the following tokens are visible: ['with']
for word 10, the following tokens are visible: ['10']
for word words, the following tokens are visible: ['words']
for word like, the following tokens are visible: ['like']
for word the, the following tokens are visible: ['the']
for word last, the following tokens are visible: ['last']
for word ., the following tokens are visible: ['.']
```

Q1.

As we discussed in class, BERT masks a random set of words in the input and attempts to reconstruct those words as output. Create a mask that randomly masks token positions 2 and 7 (for an input sequence length of 10 tokens, with 0 being the position of the first token). For an input sequence of 10 tokens, you should generate output representations for all 10 tokens (i.e., $[o_1, \ldots, o_{10}]$ in the notation above, but each representation must ignore the same 2 input tokens.

In [ ]:
```
def create_bert_mask(seq_length):
  mask=np.zeroes((seq_length,seq_length))
  # implement BERT mask here

  # BEGIN SOLUTION
  indices_to_mask = [2, 7]
  for r in range(len(mask)):
    for i in indices_to_mask:
      mask[r][i] = 1
  # END SOLUTION

  return mask
```

Q2

A left-to-right language model (such as GPT) can only use information from input words $[w_1, \ldots, w_i]$ when generating the representation for output o_i. Encode this as a mask as well.

In [ ]:
```
def create_causal_mask(seq_length):
  mask=np.ones((seq_length,seq_length))
  # implement causal mask here

  # BEGIN SOLUTION
  for r in range(len(mask)):
    for c in range(len(mask[r])):
      if c > r:
        mask[r][c] = 0
  # END SOLUTION

  return mask
```

Now let's go ahead and embed these masks within a model. First, we'll load some textual data (from Austen's *Pride and Prejudice*).

In [ ]:
```
!wget https://www.gutenberg.org/files/1342/1342-0.txt
```

```
--2023-02-19 16:40:02--  https://www.gutenberg.org/files/1342/1342-0.txt
Resolving www.gutenberg.org (www.gutenberg.org)... 152.19.134.47, 2610:28:3090:3000:0:bad:cafe:47
Connecting to www.gutenberg.org (www.gutenberg.org)|152.19.134.47|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 772145 (754K) [text/plain]
Saving to: ‘1342-0.txt.1’

1342-0.txt.1        100%[===================>] 754.05K   434KB/s    in 1.7s

2023-02-19 16:40:04 (434 KB/s) - ‘1342-0.txt.1’ saved [772145/772145]
```

In [ ]:
```
import nltk
from nltk import word_tokenize
from collections import Counter
```

In [ ]:
```
nltk.download('punkt')
```

```
[nltk_data] Downloading package punkt to
[nltk_data]     /Users/andersontsai/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
```

Out[ ]: True

Let's read in the data and tokenize it; for this homework, we'll only work with the first 10,000 tokens of that book; we'll keep only the most frequent 1,000 word types (all other tokens will be mapped to an [UNK] token).

In [ ]:
```
def read_data(filename):
  with open(filename) as file:
    data=file.read().lower()
    first10K=' '.join(data.split(" ")[:10000])
    toks=nltk.word_tokenize(first10K)[:10000]
    vocab={"[PAD]":0, "[UNK]":1}
    counts=Counter()
    for tok in toks:
      counts[tok]+=1
    for v, _ in counts.most_common(1000):
      vocab[v]=len(vocab)
    tokids=[]
    for tok in toks:
      tokid=1
      if tok in vocab:
        tokid=vocab[tok]

      tokids.append(tokid)

    return tokids, vocab
```

Now let's specify our model in PyTorch.

In [ ]:
```
from torch import nn
import torch

class MaskedLM(nn.Module):
    def __init__(self, vocab, mask, d_model=512):
        super().__init__()
        self.vocab=vocab
        self.mask=mask
        vocab_size=len(vocab)
        self.embeddings=nn.Embedding(1002,512)
        encoder_layer = nn.TransformerEncoderLayer(d_model, nhead=8, batch_first=True)
        self.transformer_encoder = nn.TransformerEncoder(encoder_layer, num_layers=6)
        self.linear=torch.nn.Linear(d_model, vocab_size)
        self.rev_vocab={vocab[k]:k for k in vocab}

    def forward(self, input):
        # first we pass the input word IDS through an embedding layer to get embeddings for them
        input=self.embeddings(input)
        # then we pass those embeddings through a transformer to get contextual representations, masking the input where appropriate
        out = self.transformer_encoder.forward(input, mask=self.mask)
        # finally we pass those embeddings through a linear layer to transform it into the output space (the size of our vocabulary)
        h=self.linear(out)
        return h
```

In [ ]:
```
def get_batches(xs, ys, batch_size=32):
    batch_x=[]
    batch_y=[]
    for i in range(0, len(xs), batch_size):
        batch_x.append(torch.LongTensor(xs[i:i+batch_size]).to(device))
        batch_y.append(torch.LongTensor(ys[i:i+batch_size]).to(device))
    return batch_x, batch_y
```

In [ ]:
```
tokids, vocab=read_data("1342-0.txt")
```

In [ ]:
```
def train(mask, data_function, tokids, vocab):

    mask=torch.BoolTensor(mask).to(device)

    num_labels=len(vocab)
    model=MaskedLM(vocab, mask).to(device)
    optimizer=torch.optim.Adam(model.parameters(), lr=0.001, weight_decay=1e-5)
    cross_entropy=nn.CrossEntropyLoss()
    losses=[]

    xs, ys=data_function(tokids)

    batch_x, batch_y=get_batches(xs, ys)

    for epoch in range(1):
        model.train()

        for x, y in list(zip(batch_x, batch_y)):
            x, y = x.to(device), y.to(device)
            y_pred=model.forward(x)
            loss=cross_entropy(y_pred.view(-1, num_labels), y.view(-1))
            losses.append(loss.item())
            print(loss)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
```

Our model and training process are now all defined; all that remains is to pass our inputs and outputs through it to train. Your job here is to create the correct inputs (x) and outputs (y) to train a left-to-right (causal) language model.

Q3

Write a function that takes in a sequence of token ids $[w_1, \ldots, w_n]$ and segments it into 8-token chunks -- e.g., $x_1 = [w_1, \ldots, w_8]$, $x_2 = [w_9, \ldots, w_{16}]$, etc. For each x_i, also create its corresponding y_i. Given this language modeling specification, each y_i should also contain 8 values (for each token in x_i). Keep in mind this is a left-to-right causal language model; your job is to figure out the values of y that respects this design. At token position i, when a model has access to $[w_1, \ldots, w_i]$, which is the true y_i for that position? Each element in y should be a word ID (i.e., an integer).

In [ ]:
```
def get_causal_xy(data, max_len=8):
    xs=[]
    ys=[]

    # BEGIN SOLUTION
    xs = [ data[i:i+max_len] for i in range(0, len(data), max_len) ]
    for row in xs:
      yr = [0] * max_len
      for i in range(0, max_len-1):
        yr[i] = row[i+1]
      ys.append(yr)
    # # END SOLUTION

    return xs, ys
```

In [ ]:
```
seq_length=8

train(create_causal_mask(seq_length=seq_length), get_causal_xy, tokids, vocab)
```

```
tensor(nan, grad_fn=<NllLossBackward0>)
tensor(nan, grad_fn=<NllLossBackward0>)
tensor(nan, grad_fn=<NllLossBackward0>)
tensor(nan, grad_fn=<NllLossBackward0>)
tensor(nan, grad_fn=<NllLossBackward0>)
tensor(nan, grad_fn=<NllLossBackward0>)
tensor(nan, grad_fn=<NllLossBackward0>)
tensor(nan, grad_fn=<NllLossBackward0>)
tensor(nan, grad_fn=<NllLossBackward0>)
tensor(nan, grad_fn=<NllLossBackward0>)
tensor(nan, grad_fn=<NllLossBackward0>)
tensor(nan, grad_fn=<NllLossBackward0>)
tensor(nan, grad_fn=<NllLossBackward0>)
tensor(nan, grad_fn=<NllLossBackward0>)
tensor(nan, grad_fn=<NllLossBackward0>)
tensor(nan, grad_fn=<NllLossBackward0>)
tensor(nan, grad_fn=<NllLossBackward0>)
tensor(nan, grad_fn=<NllLossBackward0>)
tensor(nan, grad_fn=<NllLossBackward0>)
tensor(nan, grad_fn=<NllLossBackward0>)
tensor(nan, grad_fn=<NllLossBackward0>)
tensor(nan, grad_fn=<NllLossBackward0>)
tensor(nan, grad_fn=<NllLossBackward0>)
tensor(nan, grad_fn=<NllLossBackward0>)
```

Q4 (Write-up)

In this model, as implemented, does the following equivalence hold?

$$P(y_4 \mid w_1 = \text{go}, w_2 = \text{ahead}, w_3 = \text{make}, w_4 = \text{my}) = P(y_4 \mid w_1 = \text{ahead}, w_2 = \text{my}, w_3 = \text{make}, w_4 = \text{go})$$

Why or why not?

No, since the model learns from the order of words from left to right, and generated probabilities for a word to occur based on this order. Since the words are in different order, the probability of the fifth token is different, which means the predicted fifth token can be different.

Perplexity

To evaluate how good our language model is, we use a metric called perplexity. The perplexity of a language model (PP) on a test set is the inverse probability of the test set, normalized by the number of words. Let $W = w_1 w_2 \dots w_N$. Then,

$$PP(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i|w_1 \dots w_{i-1})}}$$

However, since these probabilities are often small, taking the inverse and multiplying can be numerically unstable, so we often first compute these values in the log domain and then convert back. So this equation looks like:

$$\ln PP(W) = \frac{1}{N} \sum_{i=1}^{N} -\ln P(w_i|w_1 \dots w_{i-1})$$

$$\implies PP(W) = e^{\frac{1}{N} \sum_{i=1}^{N} -\ln P(w_i|w_1 \dots w_{i-1})}$$

Here we want to calculate the perplexity of pretrained BERT model on text from different sources. When calculating perplexity with BERT, we'll use a related measure of pseudo-perplexity, which allow us to condition on the bidirectional context (and not just the left context, as in standard perplexity):

$$PP(W) = e^{\frac{1}{N} \sum_{i=1}^{N} -\ln P(w_i|w_1 \dots w_{i-1}, w_{i+1}, \dots, w_n)}$$

First, let's instantiate a BERT model, along with its WordPiece tokenizer.

In [ ]:
```
from transformers import AutoModelForMaskedLM, AutoTokenizer
import torch
import numpy as np

model_name = 'bert-base-uncased'
model = AutoModelForMaskedLM.from_pretrained(model_name)
tokenizer = AutoTokenizer.from_pretrained(model_name)
model=model.to(device)
```

```
Downloading (…)lve/main/config.json:   0%|          | 0.00/570 [00:00<?, ?B/s]
Downloading (…)"pytorch_model.bin";:   0%|          | 0.00/440M [00:00<?, ?B/s]
Some weights of the model checkpoint at bert-base-uncased were not used when initializing BertForMaskedLM: ['cls.seq_relationship.weight', 'cls.seq_relationship.bias']
- This IS expected if you are initializing BertForMaskedLM from the checkpoint of a model trained on another task or with another architecture (e.g. initializing a BertForSequenceClassification model from a BertForPreTraining model).
- This IS NOT expected if you are initializing BertForMaskedLM from the checkpoint of a model that you expect to be exactly identical (initializing a BertForSequenceClassification model from a BertForSequenceClassification model).
Downloading (…)okenizer_config.json:   0%|          | 0.00/28.0 [00:00<?, ?B/s]
Downloading (…)solve/main/vocab.txt:   0%|          | 0.00/232k [00:00<?, ?B/s]
Downloading (…)/main/tokenizer.json:   0%|          | 0.00/466k [00:00<?, ?B/s]
```

Let's see how the BERT tokenizer tokenizes a sentence into a sequence of WordPiece ids. Note how BERT tokenization automatically wraps an input sentences with [CLS] and [SEP] tags.

In [ ]:
```
sentence = "A dog landed on Mars"
tensor_input = tokenizer(sentence, return_tensors="pt")
print(tensor_input)
tensor_input_ids = tensor_input["input_ids"]
print(tensor_input_ids)
print(tokenizer.convert_ids_to_tokens(tensor_input_ids[0]))
```

```
{'input_ids': tensor([[ 101, 1037, 3899, 5565, 2006, 7733,  102]]), 'token_type_ids': tensor([[0, 0, 0, 0, 0, 0, 0]]), 'attention_mask': tensor([[1, 1, 1, 1, 1, 1, 1]])}
tensor([[ 101, 1037, 3899, 5565, 2006, 7733,  102]])
['[CLS]', 'a', 'dog', 'landed', 'on', 'mars', '[SEP]']
```

Now let's see how we can calculate output probabilities using this model. The output of each token position i gives us $P(w_i \mid w_1, \dots, w_n)$---the probability of the word at that position over our vocabulary, given *all* of the words in the sentence.

In [ ]:
```
with torch.no_grad():
  output = model(tensor_input_ids)
  logits = output.logits
  # logits here are the unnormalized scores, so let's pass them through the softmax
  # to get a probability distribution
```

```
  softmax = torch.nn.functional.softmax(logits, dim = -1)
  # for one input sequence, the shape of the resulting distribution is:
  # 1 x [length of input, in WordPiece tokens] x (the size of the BERT vocabulary)
  print(softmax.shape) # [1, 7, 30522]
  input_ints=tensor_input_ids.numpy()[0]
  # Let's print the probability of the true inputs
  wp_tokens=tokenizer.convert_ids_to_tokens(input_ints)
  for i in range(len(input_ints)):
    prob=softmax[0][i][input_ints[i]].numpy()
    print("%s\t%s\t%.5f" % (wp_tokens[i], input_ints[i], prob))
```

```
torch.Size([1, 7, 30522])
[CLS]   101     0.00000
a       1037    0.99281
dog     3899    0.99052
landed  5565    0.99809
on      2006    0.99874
mars    7733    0.00133
[SEP]   102     0.00000
```

Note that w_i is in the range $[w_1, \ldots, w_n]$ -- clearly the probability of a word is going to be high when we can observe it in the input! Let's do some masking to calculate $P(w_i \mid w_1, \ldots w_{i-1}, w_{i+1}, w_n)$. Now annoyingly, BERT's `attention_mask` function only works for padding tokens; to mask input tokens, we need to intervene in the input and replace a WordPiece token that we're predicting with a special [MASK] token (BERT tokenizer word id `103`).

In [ ]:
```
import copy

with torch.no_grad():
  # let's make a copy of the original word ids so we can mask one of the tokens
  masked_input_ids=copy.deepcopy(tensor_input_ids)
  # we'll mask the second word
  masked_input_ids[0][1]=tokenizer.convert_tokens_to_ids("[MASK]")

  print("The second word here now is [MASK] token ID '103': ", masked_input_ids)

  # now let's run that through BERT in the same way we did before
  output = model(masked_input_ids)
  logits = output.logits

  softmax = torch.nn.functional.softmax(logits, dim = -1)
  input_ints=tensor_input_ids.numpy()[0]

  wp_tokens=tokenizer.convert_ids_to_tokens(input_ints)
  i=1
  prob=softmax[0][i][input_ints[i]].numpy()
  print("%s\t%s\t%.5f" % (wp_tokens[i], input_ints[i], prob))
```

```
The second word here now is [MASK] token ID '103':  tensor([[ 101,  103, 3899, 5565, 2006, 7733,  102]])
a       1037    0.13965
```

You can see the probability of "a" as the second token has gone down to 0.13965 when we mask it. This is the $P(w_1 = \text{a} \mid w_0, w_2, \ldots, w_n)$. At this point you should have everything you need to calculate the BERT pseudo-perplexity of an input sentence.

Q5

Implement the pseudo-perplexity measure described above, calculating the perplexity for a given model, tokenizer, and sentence.

The function calculates the average probability of each token in the sentence given all the other tokens. We need to predict the probability of each word in a sentence by masking the one word to predict. Note that you should not include the probabilities of the [CLS] and [SEP] tokens in your perplexity equation -- those tokens are not part of the original test sentence.

In [ ]:
```
# This function calculates the perplexity of a language model, given a sentence and its corresponding tokenizer

# Inputs:
# model: language model being used to calculate the perplexity
# tokenizer: tokenizer that is used to preprocess the input sentence
# sentence: input sentence string for which perplexity is to be calculated

# Outputs:
# returns perplexity of the input sentence

def perplexity(model, tokenizer, sentence):

    # hints: you'll need to:
    # encode the input sentence using the tokenizer
    # for each WordPiece token in the sentence (except [CLS] and [SEP]), mask that single token and
    # calculate the probability of that true word at the masked position
    # don't calculate perplexity for the [CLS] and [SEP] tokens (which are not part of the original test sentence).

    perplexity=0
    # BEGIN SOLUTION
    token_ids = tokenizer(sentence, add_special_tokens=False, return_tensors="pt")["input_ids"]
    logits = model(token_ids).logits
    probs = torch.nn.functional.softmax(logits, dim=-1)

    with torch.no_grad():
        input_ints = token_ids.numpy()[0]
        for i in range(1, len(input_ints)-1):
            masked_token_ids = token_ids.clone()
            masked_token_ids[0, i] = tokenizer.mask_token_id
            true_word_prob = probs[0, i, input_ints[i]]
            perplexity -= np.log(true_word_prob)
    perplexity = np.exp(perplexity / len(input_ints))
    # END SOLUTION

    return perplexity
```

In [ ]:
```
print(perplexity(sentence='London is the capital of the United Kingdom.', model=model, tokenizer=tokenizer))
```

```
tensor(1.8598)
```

No credit.

We provide texts from 4 different sources (Wikipedia, Yelp, Fiction, Twitter) collected from open-source datasets. Each category has 125 entries.

In [ ]:
```
!wget https://people.ischool.berkeley.edu/~dbamman/text_from_different_sources.txt
```

```
--2023-02-19 17:45:47--  https://people.ischool.berkeley.edu/~dbamman/text_from_different_sources.txt
Resolving people.ischool.berkeley.edu (people.ischool.berkeley.edu)... 128.32.78.16
Connecting to people.ischool.berkeley.edu (people.ischool.berkeley.edu)|128.32.78.16|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 61117 (60K) [text/plain]
Saving to: 'text_from_different_sources.txt.1'

text_from_different 100%[===================>]  59.68K  --.-KB/s    in 0.03s

2023-02-19 17:45:48 (1.71 MB/s) - 'text_from_different_sources.txt.1' saved [61117/61117]
```

In [ ]:
```
text_by_genre={}
with open('text_from_different_sources.txt') as file:
  file.readline()
  for line in file:
    cols=line.rstrip().split("\t")
    genre=cols[0]
    text=cols[1]

    if genre not in text_by_genre:
      text_by_genre[genre]=[]
    text_by_genre[genre].append(text)

for genre in text_by_genre:
  print(genre, len(text_by_genre[genre]))
```

```
Wikipedia 125
Yelp 125
Fiction 125
Twitter 125
```

Calculate perplexity on each genre over all of the words present within it; each line contains exactly one sentence for each genre.

The output perplexity_by_genre = {} is a dictionary mapping genre to a list of perplexities for each sentence in that genre. For computational purpose, we only take the first 25 sentences as an example (still this can take up to 10 minutes to run), feel free to change 25 to smaller numbers.

e.g. perplexity_by_genre['Wikipedia'] should be a list of 25 perplexities (one for each Wikipedia row in the input file).

In [ ]:
```
import numpy as np
def calculate_perplexity_by_genre(text_by_genre):
  perplexity_by_genre = {}
  for genre in text_by_genre:
    perplexity_by_genre[genre] = []
    for text in text_by_genre[genre][:25]: # change 25 to smaller numbers if necessary
      p = perplexity(sentence=text, model=model, tokenizer=tokenizer)
      perplexity_by_genre[genre].append(p)
  return perplexity_by_genre
```

In [ ]:
```
# running this might take up to 10 minutes
perplexity_by_genre = calculate_perplexity_by_genre(text_by_genre)
for genre in perplexity_by_genre:
  print("Genre:",genre,", mean perplexity:",np.mean(perplexity_by_genre[genre]))
```

```
Genre: Wikipedia , mean perplexity: 1.8357157
Genre: Yelp , mean perplexity: 3.3893404
Genre: Fiction , mean perplexity: 3.6056504
Genre: Twitter , mean perplexity: 1.888831
```

Question:

What do you think are the reasons for the wide variation in perplexity of different categories of corpus? (hint: think about the training data of the pre-trained BERT model)

Which of these is a true language model, and why?

There is a wide variation in the perplexity between different corpus, since the pre-trained BERT model was trained on Wikipedia and BookCorpus, which means it should predict more accurately words from Wikipedia and relatively modern English texts, which we see from the results above. All of the sources above are true language models, since they are all variations in ways that English is written.