COMS W4705 Spring 24

Homework 4 - Semantic Role Labelling with BERT

The goal of this assignment is to train and evaluate a PropBank-style semantic role labeling (SRL) system. Following (Collobert et al. 2011) and others, we will treat this problem as a sequence-labeling task. For each input token, the system will predict a B-I-O tag, as illustrated in the following example:

The	judge	scheduled	to	preside	over	his	trial	was	removed	from	the	ca
B-ARG1	I-ARG1	B-V	B-ARG2	I-ARG2	I-ARG2	I-ARG2	I-ARG2	O	O	O	O	O
		schedule.01										

Note that the same sentence may have multiple annotations for different predicates

The	judge	scheduled	to	preside	over	his	trial	was	removed	from	the	
B-ARG1	I-ARG1	I-ARG1	I-ARG1	I-ARG1	I-ARG1	I-ARG1	I-ARG1	O	B-V	B-ARG2	I-ARG2	I
									remove.01			

and not all predicates need to be verbs

The	judge	scheduled	to	preside	over	his	trial	was	removed	from	the	case	to
O	O	O	O	O	O	B-ARG1	B-V	O	O	O	O	O	O
							try.02						

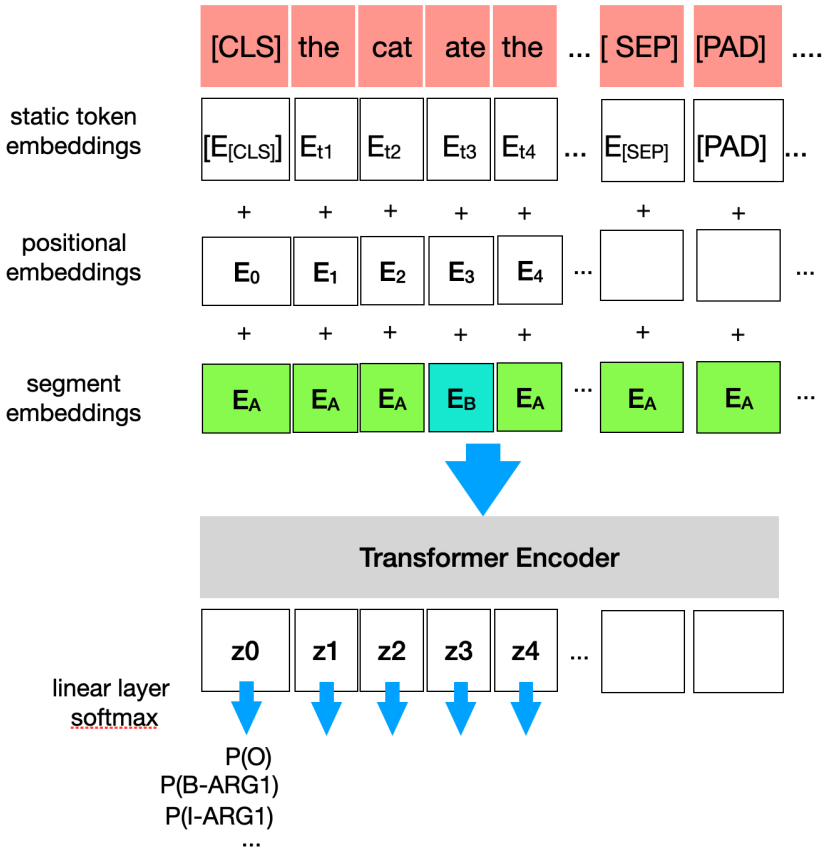
The SRL system will be implemented in PyTorch. We will use BERT (in the implementation provided by the Huggingface transformers library) to compute contextualized token representations and a custom classification head to predict semantic roles. We will fine-tune the pretrained BERT model on the SRL task.

Overview of the Approach

The model we will train is pretty straightforward. Essentially, we will just encode the sentence with BERT, then take the contextualized embedding for each token and feed it into a classifier to predict the corresponding tag.

Because we are only working on argument identification and labeling (not predicate identification), it is essentially that we tell the model where the predicate is. This can be accomplished in various ways. The approach we will choose here repurposes Bert's *segment embeddings*.

Recall that BERT is trained on two input sentences, seperated by [SEP], and on a next-sentence-prediction objective (in addition to the masked LM objective). To help BERT comprehend which sentence a given token belongs to, the original BERT uses a segment embedding, using A for the first sentene, and B for the second sentence 2. Because we are labeling only a single sentence at a time, we can use the segment embeddings to indicate the predicate position instead: The predicate is labeled as segment B (1) and all other tokens will be labeled as segment A (0).



Setup: GCP, Jupyter, PyTorch, GPU

To make sure that PyTorch is available and can use the GPU,run the following cell which should return True. If it doesn't, make sure the GPU drivers and CUDA are installed correctly.

GPU support is required for this assignment -- you will not be able to fine-tune BERT on a CPU.

In [1]:
```
import torch
torch.cuda.is_available()
```

Out[1]:
```
True
```

Dataset: Ontonotes 5.0 English SRL annotations

We will work with the English part of the Ontonotes 5.0 data. This is an extension of PropBank, using the same type of annotation. Ontonotes contains annotations other than predicate/argument structures, but we will use the PropBank style SRL annotations only. *Important*: This data set is provided to you for use in COMS 4705 only! Columbia is a subscriber to LDC and is allowed to use the data for educational purposes. However, you may not use the dataset in projects unrelated to Columbia teaching or research.

If you haven't done so already, you can download the data here:

In [2]:
```
! wget https://storage.googleapis.com/4705-bert-srl-data/ontonotes_srl.zip
```

```
--2024-05-01 15:41:30--  https://storage.googleapis.com/4705-bert-srl-data/o
ntonotes_srl.zip
Resolving storage.googleapis.com (storage.googleapis.com)... 172.217.13.155,
172.217.13.187, 172.217.13.219, ...
Connecting to storage.googleapis.com (storage.googleapis.com)|172.217.13.155
|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 12369688 (12M) [application/zip]
Saving to: ‘ontonotes_srl.zip.3’

ontonotes_srl.zip.3 100%[===================>]  11.80M  41.5MB/s    in 0.3s

2024-05-01 15:41:30 (41.5 MB/s) - ‘ontonotes_srl.zip.3’ saved [12369688/1236
9688]
```

In [3]:
```
! unzip ontonotes_srl.zip
```

```
Archive:  ontonotes_srl.zip
replace propbank_dev.tsv? [y]es, [n]o, [A]ll, [N]one, [r]ename: ^C
```

The data has been pre-processed in the following format. There are three files:

`propbank_dev.tsv` `propbank_test.tsv` `propbank_train.tsv`

Each of these files is in a tab-separated value format. A single predicate/argument structure annotation consists of four rows. For example

```
ontonotes/bc/cnn/00/cnn_0000.152.1
The     judge   scheduled       to      preside over    his
trial   was     removed from    the     case    today   /.
                schedule.01
B-ARG1  I-ARG1  B-V     B-ARG2  I-ARG2  I-ARG2  I-ARG2  I-
ARG2  O       O       O       O       O       O       O
```

- The first row is a unique identifier (1st annotation of the 152nd sentence in the file ontonotes/bc/cnn/00/cnn_0000).
- The second row contains the tokens of the sentence (tab-separated).
- The third row contains the probank frame name for the predicate (empty field for all other tokens).
- The fourth row contains the B-I-O tag for each token.

The file `rolelist.txt` contains a list of propbank BIO labels in the dataset (i.e. possible output tokens). This list has been filtered to contain only roles that appeared more than 1000 times in the training data. We will load this list and create mappings from numeric ids to BIO tags and back.

In [2]:
```
role_to_id = {}
with open("role_list.txt",'r') as f:
    role_list = [x.strip() for x in f.readlines()]
    role_to_id = dict((role, index) for (index, role) in enumerate(role_list
    role_to_id['[PAD]'] = -100

    id_to_role = dict((index, role) for (role, index) in role_to_id.items())
```

Note that we are also mapping the '[PAD]' token to the value -100. This allows the loss function to ignore these tokens during training.

Part 1 - Data Preparation

Before you can build the SRL model, you first need to preprocess the data.

1.1 - Tokenization

One challenge is that the pre-trained BERT model uses subword ("WordPiece") tokenization, but the Ontonotes data does not. Fortunately Huggingface transformers provides a tokenizer.

In [3]:
```
from transformers import BertTokenizerFast
tokenizer = BertTokenizerFast.from_pretrained('bert-base-uncased', do_lower_
tokenizer.tokenize("This is an unbelievably boring test sentence.")
```

Out[3]:
```
['this',
 'is',
 'an',
 'un',
 '##bel',
 '##ie',
 '##va',
 '##bly',
 'boring',
 'test',
 'sentence',
 '.']
```

TODO: We need to be able to maintain the correct labels (B-I-O tags) for each of the subwords. Complete the following function that takes a list of tokens and a list of B-I-O labels of the same length as parameters, and returns a new token / label pair, as illustrated in the following example.

```
>>> tokenize_with_labels("the fancyful penguin devoured yummy
fish .".split(), "B-ARG0 I-ARG0 I-ARG0 B-V B-ARG1 I-ARG1
O".split(), tokenizer)
(['the',
  'fancy',
  '##ful',
  'penguin',
  'dev',
  '##oured',
  'yu',
  '##mmy',
  'fish',
  '.'],
 ['B-ARG0',
  'I-ARG0',
  'I-ARG0',
  'I-ARG0',
  'B-V',
  'I-V',
  'B-ARG1',
  'I-ARG1',
  'I-ARG1',
  'O'])
```

To approach this problem, iterate through each word/label pair in the sentence. Call the tokenizer on the word. This may result in one or more tokens. Create the correct number of labels to match the number of tokens. Take care to not generate multiple B- tokens.

This approach is a bit slower than tokenizing the entire sentence, but is necessary to produce proper input tokenization for the pre-trained BERT model, and the matching target labels.

In [4]:
```
def tokenize_with_labels(sentence, text_labels, tokenizer):
    """
    Word piece tokenization makes it difficult to match word labels
    back up with individual word pieces.
    """

    tokenized_sentence = []
    labels = []

    for word, label in zip(sentence, text_labels):

        # tokenize words
        sw = tokenizer.tokenize(word)
        if len(sw) > 1:
            sw_labels = []
            sw_labels.append(label)

            for i in range(1, len(sw)):
                if label.startswith('B-'):
                    sw_labels.append(label.replace('B-', 'I-'))
                else:
                    sw_labels.append(label)
        else:
            sw_labels = [label]


        tokenized_sentence.extend(sw)
        labels.extend(sw_labels)

    return tokenized_sentence, labels
```

In [5]:
```
tokenize_with_labels("the fancyful penguin devoured yummy fish .".split(), "
```

Out[5]:
```
(['the',
  'fancy',
  '##ful',
  'penguin',
  'dev',
  '##oured',
  'yu',
  '##mmy',
  'fish',
  '.'],
 ['B-ARG0',
  'I-ARG0',
  'I-ARG0',
  'I-ARG0',
  'B-V',
  'I-V',
  'B-ARG1',
  'I-ARG1',
  'I-ARG1',
  'O'])
```

1.2 Loading the Dataset

Next, we are creating a PyTorch Dataset class. This class acts as a contained for the training, development, and testing data in memory. You should already be familiar with Datasets and Dataloaders from homework 3.

1.2.1 **TODO**: Write the __init__(self, filename) method that reads in the data from a data file (specified by the filename).

For each annotation you start with the tokens in the sentence, and the BIO tags. Then you need to create the following

1. call the `tokenize_with_labels` function to tokenize the sentence.
2. Add the (token, label) pair to the self.items list.

1.2.2 **TODO**: Write the __len__(self) method that returns the total number of items.

1.2.3 **TODO**: Write the __getitem__(self, k) method that returns a single item in a format BERT will understand.

- We need to process the sentence by adding "[CLS]" as the first token and "[SEP]" as the last token. The need to pad the token sequence to 128 tokens using the "[PAD]" symbol. This needs to happen both for the inputs (sentence token sequence) and outputs (BIO tag sequence).
- We need to create an *attention mask*, which is a sequence of 128 tokens indicating

the actual input symbols (as a 1) and [PAD] symbols (as a 0).

- We need to create a *predicate indicator* mask, which is a sequence of 128 tokens with at most one 1, in the position of the "B-V" tag. All other entries should be 0. The model will use this information to understand where the predicate is located.

- Finally, we need to convert the token and tag sequence into numeric indices. For the tokens, this can be done using the `tokenizer.convert_tokens_to_ids` method. For the tags, use the `role_to_id` dictionary. Each sequence must be a pytorch tensor of shape (1,128). You can convert a list of integer values like this `torch.tensor(token_ids, dtype=torch.long)`.

To keep everything organized, we will return a dictionary in the following format

```
{'ids': token_tensor,
 'targets': tag_tensor,
 'mask': attention_mask_tensor,
 'pred': predicate_indicator_tensor}
```

(Hint: To debug these, read in the first annotation only / the first few annotations)

In [6]:
```
from torch.utils.data import Dataset, DataLoader

class SrlData(Dataset):

    def __init__(self, filename):
        super(SrlData, self).__init__()

        self.max_token_length = 128
        self.tokenizer = BertTokenizerFast.from_pretrained('bert-base-uncase
        self.dataset_entries = self.load_data_from_file(filename)

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

    def __getitem__(self, index):
        tokenized_sentence, aligned_labels = self.dataset_entries[index]

        token_ids = self.tokenizer.convert_tokens_to_ids(tokenized_sentence)
        input_ids = self.pad_sequence(token_ids, self.max_token_length)

        label_ids = self.convert_labels_to_ids(aligned_labels)
        label_tensor = self.pad_sequence(label_ids, self.max_token_length, p

        attention_mask = [1] * len(token_ids) + [0] * (self.max_token_length
        predicate_indicator = self.create_predicate_indicator(aligned_labels

        return {
            'ids': torch.tensor(input_ids, dtype=torch.long),
```

```
            'mask': torch.tensor(attention_mask, dtype=torch.long),
            'targets': torch.tensor(label_tensor, dtype=torch.long),
            'pred': torch.tensor(predicate_indicator, dtype=torch.long)
        }

    def load_data_from_file(self, filename):
        data_entries = []
        with open(filename, 'r') as file:
            while True:
                identifier = file.readline().strip()
                if not identifier:
                    break
                sentence = file.readline().strip().split()
                predicate = file.readline().strip()
                labels = file.readline().strip().split()


                tokenized_sentence, token_labels = tokenize_with_labels(sent
                tokenized_sentence = ['[CLS]'] + tokenized_sentence[:self.ma
                token_labels = ['O'] + token_labels[:self.max_token_length -

                data_entries.append((tokenized_sentence, token_labels))
        return data_entries

    def pad_sequence(self, sequence, max_length, pad_value=0):
        return sequence + [pad_value] * (max_length - len(sequence))

    def convert_labels_to_ids(self, labels):
        return [-100] + [role_to_id.get(label, -100) for label in labels[1:-

    def create_predicate_indicator(self, labels):
        predicate_indicator = [0] * self.max_token_length
        if 'B-V' in labels:
            verb_position = labels.index('B-V')
            if verb_position < self.max_token_length:
                predicate_indicator[verb_position] = 1
        return predicate_indicator
```

In [7]:
```
# Reading the training data takes a while for the entire data because we pre
data = SrlData("propbank_train.tsv")
```

2. Model Definition

In [8]:
```
from torch.nn import Module, Linear, CrossEntropyLoss
from transformers import BertModel
```

We will define the pyTorch model as a subclass of the torch.nn.Module class. The code for the model is provided for you. It may help to take a look at the documentation to remind you of how Module works. Take a look at how the huggingface BERT model simply becomes another sub-module.

In [9]:
```
class SrlModel(Module):

    def __init__(self):

        super(SrlModel, self).__init__()

        self.encoder = BertModel.from_pretrained("bert-base-uncased")

        # The following two lines would freeze the BERT parameters and allow
        # We are fine-tuning the model, so you can leave this commented out!
        # for param in self.encoder.parameters():
        #    param.requires_grad = False

        # The linear classifier head, see model figure in the introduction.
        self.classifier = Linear(768, len(role_to_id))


    def forward(self, input_ids, attn_mask, pred_indicator):

        # This defines the flow of data through the model

        # Note the use of the "token type ids" which represents the segment
        # In our segment encoding, 1 indicates the predicate, and 0 indicate
        bert_output =  self.encoder(input_ids=input_ids, attention_mask=attn

        enc_tokens = bert_output[0] # the result of encoding the input with
        logits = self.classifier(enc_tokens) #feed into the classification l

        # Note that we are only interested in the argmax for each token, so
        # to a probability distribution using softmax. The CrossEntropyLoss
        # It essentially computes the softmax first and then computes the ne
        return logits
```

In [10]:
```
model = SrlModel().to('cuda') # create new model and store weights in GPU me
```

```
Some weights of the model checkpoint at bert-base-uncased were not used when
initializing BertModel: ['cls.seq_relationship.weight', 'cls.seq_relationshi
p.bias', 'cls.predictions.transform.dense.bias', 'cls.predictions.transform.
dense.weight', 'cls.predictions.transform.LayerNorm.weight', 'cls.prediction
s.transform.LayerNorm.bias', 'cls.predictions.bias']
- This IS expected if you are initializing BertModel from the checkpoint of
a model trained on another task or with another architecture (e.g. initializ
ing a BertForSequenceClassification model from a BertForPreTraining model).
- This IS NOT expected if you are initializing BertModel from the checkpoint
of a model that you expect to be exactly identical (initializing a BertForSe
quenceClassification model from a BertForSequenceClassification model).
```

Now we are ready to try running the model with just a single input example to check if it is working correctly. Clearly it has not been trained, so the output is not what we expect. But we can see what the loss looks like for an initial sanity check.

TODO:

- Take a single data item from the dev set, as provided by your Dataset class defined above. Obtain the input token ids, attention mask, predicate indicator mask, and target labels.
- Run the model on the ids, attention mask, and predicate mask like this:

In [11]:
```
# pick an item from the dataset. Then run

data = SrlData("propbank_dev.tsv")
example = data[0]
ids, mask, pred = example['ids'].unsqueeze(0).cuda(), example['mask'].unsque


model = SrlModel().to('cuda')
outputs = model(ids, mask, pred)
```

```
Some weights of the model checkpoint at bert-base-uncased were not used when
initializing BertModel: ['cls.seq_relationship.weight', 'cls.seq_relationshi
p.bias', 'cls.predictions.transform.dense.bias', 'cls.predictions.transform.
dense.weight', 'cls.predictions.transform.LayerNorm.weight', 'cls.prediction
s.transform.LayerNorm.bias', 'cls.predictions.bias']
- This IS expected if you are initializing BertModel from the checkpoint of
a model trained on another task or with another architecture (e.g. initializ
ing a BertForSequenceClassification model from a BertForPreTraining model).
- This IS NOT expected if you are initializing BertModel from the checkpoint
of a model that you expect to be exactly identical (initializing a BertForSe
quenceClassification model from a BertForSequenceClassification model).
```

In [12]:
```
print(outputs)
```

```
tensor([[[-0.5272, -0.1824,  0.2601,  ..., -0.2194, -0.6609,  0.2998],
         [-0.4737,  0.2553,  0.3648,  ..., -0.0649, -0.1399,  0.2702],
         [-0.3644, -0.0799,  0.5314,  ..., -0.1487,  0.4253,  0.3470],
         ...,
         [-0.0466, -0.2391,  0.2214,  ...,  0.2220, -0.1415, -0.0128],
         [-0.1060, -0.3233,  0.3101,  ...,  0.1383, -0.0104, -0.0050],
         [-0.0518, -0.2785,  0.1834,  ...,  0.1917, -0.0802,  0.0438]]],
       device='cuda:0', grad_fn=<AddBackward0>)
```

TODO: Compute the loss on this one item only. The initial loss should be close to -ln(1/num_labels)

Without training we would assume that all labels for each token (including the target label) are equally likely, so the negative log probability for the targets should be approximately

$$-\ln(\frac{1}{\text{num\_labels}}).$$

This is what the loss function should return on a single example. This is a good sanity check to run for any multi-class prediction problem.

In [13]:
```
import math
-math.log(1 / len(role_to_id), math.e)
```

Out[13]:
```
3.970291913552122
```

In [14]:
```
loss_function = CrossEntropyLoss(ignore_index = -100, reduction='mean')

# complete this. Note that you still have to provide a (batch_size, input_pc
# tensor for each parameter, where batch_size =1

# outputs = model(ids, mask, pred)
# loss = loss_function(...)
# loss.item()   #this should be approximately the score from the previous ce

targets = example['targets'].unsqueeze(0).cuda()
loss = loss_function(outputs.view(-1, len(role_to_id)), targets.view(-1))
print(loss.item())
```

```
3.9902141094207764
```

TODO: At this point you should also obtain the actual predictions by taking the argmax over each position. The result should look something like this (values will differ).

```
tensor([[ 1,  4,  4,  4,  4,  4,  5, 29, 29, 29,  4, 28,  6,
32, 32, 32, 32, 32,
         32, 32, 30, 30, 32, 30, 32,  4, 32, 32, 30,  4, 49,
4, 49, 32, 30,  4,
         32,  4, 32, 32,  4,  2,  4,  4, 32,  4, 32, 32, 32,
32, 30, 32, 32, 30,
         32,  4,  4, 49,  4,  4,  4,  4,  4,  4,  4,  4,  4,
4,  6,  6, 32, 32,
         30, 32, 32, 32, 32, 32, 30, 30, 30, 32, 30, 49, 49,
32, 32, 30,  4,  4,
          4,  4, 29,  4,  4,  4,  4,  4,  4, 32,  4,  4,  4,
32,  4, 30,  4, 32,
         30,  4, 32,  4,  4,  4,  4,  4, 32,  4,  4,  4,  4,
4,  4,  4,  4,  4,
          4,  4]], device='cuda:0')
```

Then use the id_to_role dictionary to decode to actual tokens.

```
['[CLS]', 'O', 'O', 'O', 'O', 'O', 'B-ARG0', 'I-ARG0', 'I-
ARG0', 'I-ARG0', 'O', 'B-V', 'B-ARG1', 'I-ARG2', 'I-ARG2',
'I-ARG2', 'I-ARG2', 'I-ARG2', 'I-ARG2', 'I-ARG2', 'I-ARG1',
'I-ARG1', 'I-ARG2', 'I-ARG1', 'I-ARG2', 'O', 'I-ARG2', 'I-
ARG2', 'I-ARG1', 'O', 'I-ARGM-TMP', 'O', 'I-ARGM-TMP', 'I-
ARG2', 'I-ARG1', 'O', 'I-ARG2', 'O', 'I-ARG2', 'I-ARG2', 'O',
'[SEP]', 'O', 'O', 'I-ARG2', 'O', 'I-ARG2', 'I-ARG2', 'I-
ARG2', 'I-ARG2', 'I-ARG1', 'I-ARG2', 'I-ARG2', 'I-ARG1', 'I-
ARG2', 'O', 'O', 'I-ARGM-TMP', 'O', 'O', 'O', 'O', 'O', 'O',
'O', 'O', 'O', 'O', 'B-ARG1', 'B-ARG1', 'I-ARG2', 'I-ARG2',
'I-ARG1', 'I-ARG2', 'I-ARG2', 'I-ARG2', 'I-ARG2', 'I-ARG2',
'I-ARG1', 'I-ARG1', 'I-ARG1', 'I-ARG2', 'I-ARG1', 'I-ARGM-
TMP', 'I-ARGM-TMP', 'I-ARG2', 'I-ARG2', 'I-ARG1', 'O', 'O',
'O', 'O', 'I-ARG0', 'O', 'O', 'O', 'O', 'O', 'O', 'I-ARG2',
'O', 'O', 'O', 'I-ARG2', 'O', 'I-ARG1', 'O', 'I-ARG2', 'I-
ARG1', 'O', 'I-ARG2', 'O', 'O', 'O', 'O', 'O', 'I-ARG2', 'O',
'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O']
```

For now, just make sure you understand how to do this for a single example. Later, you will write a more formal function to do this once we have trained the model.

In [15]:
```
pred = torch.argmax(outputs, dim=-1)
decoded_labels = [id_to_role[index.item()] for index in pred[0]]
print(pred)
print(decoded_labels)
```

```
tensor([[24, 43, 31,  2, 16, 52, 20,  6, 42, 33, 26, 23, 43, 24, 24, 51,  6,
43,
         43, 43, 31, 27,  6, 51, 51, 51,  6, 41, 31,  6,  6,  6,  6,  6,  6,
31,
         45, 31,  6,  6, 31, 23, 33, 45,  6,  6,  6,  6,  6, 33,  6,  6,  6,
6,
          6,  6,  6,  6,  6, 16, 16, 33, 45, 31, 45, 33, 23, 23, 31, 31, 31,
6,
          6,  6,  6,  6,  6,  6,  6,  6,  6,  6,  6,  6,  6,  6,  6,  6,  2,
29,
         33, 45, 31,  6,  6, 31, 23, 31, 31, 31,  6,  6,  6,  6,  6, 31,  6,
6,
          6, 16,  6,  6,  6,  6,  6,  6,  6,  6,  6,  6,  6,  6,  6,  6,  6,
31,
          6,  6]], device='cuda:0')
['B-ARGM-PRR', 'I-ARGM-MNR', 'I-ARG1-DSP', '[SEP]', 'B-ARGM-EXT', 'I-V', 'B-
ARGM-MOD', 'B-ARG1', 'I-ARGM-LOC', 'I-ARG3', 'B-ARGM-CXN', 'B-ARGM-PRP', 'I-
ARGM-MNR', 'B-ARGM-PRR', 'B-ARGM-PRR', 'I-ARGM-LVB', 'B-ARG1', 'I-ARGM-MNR',
'I-ARGM-MNR', 'I-ARGM-MNR', 'I-ARG1-DSP', 'B-ARGM-LVB', 'B-ARG1', 'I-ARGM-LV
B', 'I-ARGM-LVB', 'I-ARGM-LVB', 'B-ARG1', 'I-ARGM-GOL', 'I-ARG1-DSP', 'B-ARG
1', 'B-ARG1', 'B-ARG1', 'B-ARG1', 'B-ARG1', 'B-ARG1', 'I-ARG1-DSP', 'I-ARGM-
NEG', 'I-ARG1-DSP', 'B-ARG1', 'B-ARG1', 'I-ARG1-DSP', 'B-ARGM-PRP', 'I-ARG
3', 'I-ARGM-NEG', 'B-ARG1', 'B-ARG1', 'B-ARG1', 'B-ARG1', 'B-ARG1', 'I-ARG
3', 'B-ARG1', 'B-ARG1', 'B-ARG1', 'B-ARG1', 'B-ARG1', 'B-ARG1', 'B-ARG1', 'B
-ARG1', 'B-ARG1', 'B-ARGM-EXT', 'B-ARGM-EXT', 'I-ARG3', 'I-ARGM-NEG', 'I-ARG
1-DSP', 'I-ARGM-NEG', 'I-ARG3', 'B-ARGM-PRP', 'B-ARGM-PRP', 'I-ARG1-DSP', 'I
-ARG1-DSP', 'I-ARG1-DSP', 'B-ARG1', 'B-ARG1', 'B-ARG1', 'B-ARG1', 'B-ARG1',
'B-ARG1', 'B-ARG1', 'B-ARG1', 'B-ARG1', 'B-ARG1', 'B-ARG1', 'B-ARG1', 'B-ARG
1', 'B-ARG1', 'B-ARG1', 'B-ARG1', 'B-ARG1', '[SEP]', 'I-ARG0', 'I-ARG3', 'I-
ARGM-NEG', 'I-ARG1-DSP', 'B-ARG1', 'B-ARG1', 'I-ARG1-DSP', 'B-ARGM-PRP', 'I-
ARG1-DSP', 'I-ARG1-DSP', 'I-ARG1-DSP', 'B-ARG1', 'B-ARG1', 'B-ARG1', 'B-ARG
1', 'B-ARG1', 'I-ARG1-DSP', 'B-ARG1', 'B-ARG1', 'B-ARG1', 'B-ARGM-EXT', 'B-A
RG1', 'B-ARG1', 'B-ARG1', 'B-ARG1', 'B-ARG1', 'B-ARG1', 'B-ARG1', 'B-ARG1',
'B-ARG1', 'B-ARG1', 'B-ARG1', 'B-ARG1', 'B-ARG1', 'B-ARG1', 'B-ARG1', 'I-ARG
1-DSP', 'B-ARG1', 'B-ARG1']
```

3. Training loop

pytorch provides a DataLoader class that can be wrapped around a Dataset to easily use the dataset for training. The DataLoader allows us to easily adjust the batch size and shuffle the data.

In [16]:
```
from torch.utils.data import DataLoader
loader = DataLoader(data, batch_size = 32, shuffle = True)
```

The following cell contains the main training loop. The code should work as written and report the loss after each batch, cumulative average loss after each 100 batches, and print out the final average loss after the epoch.

TODO: Modify the training loop belowso that it also computes the accuracy for each batch and reports the average accuracy after the epoch. The accuracy is the number of correctly predicted token labels out of the number of total predictions. Make sure you exclude [PAD] tokens, i.e. tokens for which the target label is -100. It's okay to include [CLS] and [SEP] in the accuracy calculation.

In [17]:
```
loss_function = CrossEntropyLoss(ignore_index = -100, reduction='mean')

LEARNING_RATE = 1e-05
optimizer = torch.optim.AdamW(params=model.parameters(), lr=LEARNING_RATE)

device = 'cuda'

def train():
    """
    Train the model for one epoch.
    """
    tr_loss = 0
    nb_tr_examples, nb_tr_steps = 0, 0
    total_loss = 0
    correct_predictions, total_predictions = 0, 0

    enumerated_loader = enumerate(loader)
    for step, batch in enumerated_loader:

        input_ids, attention_mask, labels, predicate_mask = (
            batch['ids'].to(device, dtype=torch.long),
            batch['mask'].to(device, dtype=torch.long),
            batch['targets'].to(device, dtype=torch.long),
            batch['pred'].to(device, dtype=torch.long)
        )


        logits = model(input_ids=input_ids, attn_mask=attention_mask, pred_i
        loss = loss_function(logits.transpose(2, 1), labels)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        total_loss += loss.item()


        is_not_padding = labels != -100
        predictions = torch.argmax(logits, dim=2)
        correct = (predictions[is_not_padding] == labels[is_not_padding]).su
        correct_predictions += correct
```

```
        total_predictions += is_not_padding.sum().item()


        nb_tr_steps += 1
        nb_tr_examples += labels.size(0)
        if nb_tr_steps % 100 == 0:
            avg_loss = total_loss / nb_tr_steps
            accuracy = correct_predictions / total_predictions if total_pred
            print(f"Step {nb_tr_steps}, average loss: {avg_loss:.4f}, accura


    epoch_loss = total_loss / nb_tr_steps
    epoch_accuracy = correct_predictions / total_predictions if total_predic
    print(f"Training loss  epoch: {epoch_loss:.4f}, Training accuracy epoch:
```

Now let's train the model for one epoch. This will take a while (up to a few hours).

In [18]:
```
train()
```

```
Step 100, average loss: 1.9888, accuracy: 0.59
Step 200, average loss: 1.7627, accuracy: 0.61
Step 300, average loss: 1.6378, accuracy: 0.62
Step 400, average loss: 1.5278, accuracy: 0.64
Step 500, average loss: 1.4350, accuracy: 0.66
Step 600, average loss: 1.3601, accuracy: 0.67
Step 700, average loss: 1.2908, accuracy: 0.69
Step 800, average loss: 1.2271, accuracy: 0.70
Step 900, average loss: 1.1643, accuracy: 0.72
Step 1000, average loss: 1.1119, accuracy: 0.73
Step 1100, average loss: 1.0648, accuracy: 0.74
Step 1200, average loss: 1.0226, accuracy: 0.75
Step 1300, average loss: 0.9837, accuracy: 0.76
Training loss  epoch: 0.9606, Training accuracy epoch: 0.76
```

In [19]:
```
train()
```

```
Step 100, average loss: 0.4937, accuracy: 0.87
Step 200, average loss: 0.4635, accuracy: 0.88
Step 300, average loss: 0.4528, accuracy: 0.88
Step 400, average loss: 0.4510, accuracy: 0.88
Step 500, average loss: 0.4439, accuracy: 0.88
Step 600, average loss: 0.4389, accuracy: 0.88
Step 700, average loss: 0.4320, accuracy: 0.88
Step 800, average loss: 0.4256, accuracy: 0.88
Step 900, average loss: 0.4215, accuracy: 0.88
Step 1000, average loss: 0.4169, accuracy: 0.88
Step 1100, average loss: 0.4140, accuracy: 0.89
Step 1200, average loss: 0.4089, accuracy: 0.89
Step 1300, average loss: 0.4035, accuracy: 0.89
Training loss  epoch: 0.4023, Training accuracy epoch: 0.89
```

In [20]:
```
train()
```

```
Step 100, average loss: 0.3070, accuracy: 0.91
Step 200, average loss: 0.3024, accuracy: 0.91
Step 300, average loss: 0.3042, accuracy: 0.91
Step 400, average loss: 0.3060, accuracy: 0.91
Step 500, average loss: 0.3027, accuracy: 0.91
Step 600, average loss: 0.3006, accuracy: 0.91
Step 700, average loss: 0.2983, accuracy: 0.92
Step 800, average loss: 0.2967, accuracy: 0.92
Step 900, average loss: 0.2961, accuracy: 0.92
Step 1000, average loss: 0.2947, accuracy: 0.92
Step 1100, average loss: 0.2923, accuracy: 0.92
Step 1200, average loss: 0.2919, accuracy: 0.92
Step 1300, average loss: 0.2911, accuracy: 0.92
Training loss  epoch: 0.2908, Training accuracy epoch: 0.92
```

In [21]:
```
train()
```

```
Step 100, average loss: 0.2537, accuracy: 0.93
Step 200, average loss: 0.2372, accuracy: 0.93
Step 300, average loss: 0.2314, accuracy: 0.93
Step 400, average loss: 0.2310, accuracy: 0.93
Step 500, average loss: 0.2290, accuracy: 0.93
Step 600, average loss: 0.2304, accuracy: 0.93
Step 700, average loss: 0.2287, accuracy: 0.93
Step 800, average loss: 0.2283, accuracy: 0.93
Step 900, average loss: 0.2290, accuracy: 0.93
Step 1000, average loss: 0.2296, accuracy: 0.93
Step 1100, average loss: 0.2297, accuracy: 0.93
Step 1200, average loss: 0.2294, accuracy: 0.93
Step 1300, average loss: 0.2291, accuracy: 0.93
Training loss  epoch: 0.2284, Training accuracy epoch: 0.93
```

In [23]:
```
train()
```

```
Step 100, average loss: 0.1822, accuracy: 0.95
Step 200, average loss: 0.1892, accuracy: 0.95
Step 300, average loss: 0.1875, accuracy: 0.95
Step 400, average loss: 0.1870, accuracy: 0.95
Step 500, average loss: 0.1839, accuracy: 0.95
Step 600, average loss: 0.1862, accuracy: 0.95
Step 700, average loss: 0.1855, accuracy: 0.95
Step 800, average loss: 0.1855, accuracy: 0.95
Step 900, average loss: 0.1850, accuracy: 0.95
Step 1000, average loss: 0.1849, accuracy: 0.95
Step 1100, average loss: 0.1839, accuracy: 0.95
Step 1200, average loss: 0.1842, accuracy: 0.95
Step 1300, average loss: 0.1841, accuracy: 0.95
Training loss  epoch: 0.1844, Training accuracy epoch: 0.95
```

In my experiments, I found that two epochs are needed for good performance.

I ended up with a training loss of about 0.19 and a training accuracy of 0.94. Specific values may differ.

At this point, it's a good idea to save the model (or rather the parameter dictionary) so you can continue evaluating the model without having to retrain.

In [24]:
```
torch.save(model.state_dict(), "srl_model_fulltrain_2epoch_finetune_1e-05.pt
```

4. Decoding

In [ ]:
```
# Optional step: If you stopped working after part 3, first load the trained

model = SrlModel().to('cuda')
model.load_state_dict(torch.load("srl_model_fulltrain_2epoch_finetune_1e-05.
model = model.to('cuda')
```

TODO (this is the fun part): Now that we have a trained model, let's try labeling an unseen example sentence. Complete the functions decode_output and label_sentence below. decode_output takes the logits returned by the model, extracts the argmax to obtain the label predictions for each token, and then translate the result into a list of string labels.

label_sentence takes a list of input tokens and a predicate index, prepares the model input, call the model and then call decode_output to produce a final result.

Note that you have already implemented all components necessary (preparing the input data from the token list and predicate index, decoding the model output). But now you are putting it together in one convenient function.

In [69]:
```
tokens = "A U. N. team spent an hour inside the hospital , where it found ev
```

In [70]:
```
def decode_output(logits):
    """
    Given the model output logits, return a list of string labels for each t
    """
    predictions = torch.argmax(logits, dim=2)
    squeezed_predictions = predictions.squeeze()
    label_ids = squeezed_predictions.tolist()
    labels = []
    for id in label_ids:
        labels.append(id_to_role[id])
    return labels
```

In [71]:
```
def label_sentence(tokens, pred_idx):

    # complete this function to prepare token_ids, attention mask, predicate
    # Decode the output to produce a list of labels.

    # tokenize and pad
    ids = tokenizer.convert_tokens_to_ids(['[CLS]'] + tokens + ['[SEP]'])
    ids_padded = ids + [0] * (128 - len(ids))

    #attention and predicate mask
    mask = [1] * len(ids) + [0] * (128 - len(ids))
    predicate_mask = [0] * 128
    predicate_mask[pred_idx + 1] = 1  # Offset for [CLS]

    # model inference
    with torch.no_grad():
        output_logits = model(
            input_ids=torch.tensor(ids_padded, dtype=torch.long).unsqueeze(0
            attn_mask=torch.tensor(mask, dtype=torch.long).unsqueeze(0).to('
            pred_indicator=torch.tensor(predicate_mask, dtype=torch.long).un
        )

    # decode output and take out CLS and SEP tokens
    predicted_labels = decode_output(output_logits)
    return predicted_labels[1:len(tokens)+1]
```

In [72]:
```
# Now you should be able to run

label_test = label_sentence(tokens, 13) # Predicate is "found"
zip(tokens, label_test)
list(zip(tokens, label_test))
```

Out[72]:
```
[('A', 'O'),
 ('U.', 'O'),
 ('N.', 'O'),
 ('team', 'O'),
 ('spent', 'O'),
 ('an', 'O'),
 ('hour', 'O'),
 ('inside', 'O'),
 ('the', 'B-ARGM-LOC'),
 ('hospital', 'I-ARGM-LOC'),
 (',', 'O'),
 ('where', 'B-ARGM-LOC'),
 ('it', 'B-ARG0'),
 ('found', 'B-V'),
 ('evident', 'B-ARG1'),
 ('signs', 'I-ARG1'),
 ('of', 'I-ARG1'),
 ('shelling', 'I-ARG1'),
 ('and', 'I-ARG1'),
 ('gunfire', 'I-ARG1'),
 ('.', 'O')]
```

The expected output is somethign like this:

```
('A', 'O'),
('U.', 'O'),
('N.', 'O'),
('team', 'O'),
('spent', 'O'),
('an', 'O'),
('hour', 'O'),
('inside', 'O'),
('the', 'B-ARGM-LOC'),
('hospital', 'I-ARGM-LOC'),
(',', 'O'),
('where', 'B-ARGM-LOC'),
('it', 'B-ARG0'),
('found', 'B-V'),
('evident', 'B-ARG1'),
('signs', 'I-ARG1'),
('of', 'I-ARG1'),
('shelling', 'I-ARG1'),
('and', 'I-ARG1'),
('gunfire', 'I-ARG1'),
('.', 'O'),
```

5. Evaluation 1: Token-Based Accuracy

We want to evaluate the model on the dev or test set.

In [73]:
```
dev_data = SrlData("propbank_dev.tsv") # Takes a while because we preprocess
```

In [74]:
```
from torch.utils.data import DataLoader
loader = DataLoader(dev_data, batch_size = 1, shuffle = False)
```

In [ ]:
```
# Optional: Load the model again if you stopped working prior to this step.
# model = SrlModel()
# model.load_state_dict(torch.load("srl_model_fulltrain_2epoch_finetune_1e-0
# model = mode.to('cuda')
```

TODO: Complete the evaluate_token_accuracy function below. The function should iterate through the items in the data loader (see training loop in part 3). Run the model on each sentence/predicate pair and extract the predictions.

For each sentence, count the correct predictions and the total predictions. Finally, compute the accuracy as #correct_predictions / #total_predictions

Careful: You need to filter out the padded positions ([PAD] target tokens), as well as [CLS] and [SEP]. It's okay to include [B-V] in the count though.

In [80]:
```
def evaluate_token_accuracy(model, loader):

    model.eval() # put model in evaluation mode

    # for the accuracy
    total_correct = 0 # number of correct token label predictions.
    total_predictions = 0 # number of total predictions = number of tokens i

    for batch in loader:

        input_ids = batch['ids'].to('cuda')
        attn_mask = batch['mask'].to('cuda')
        labels = batch['targets'].to('cuda')
        pred_indicator = batch['pred'].to('cuda')

        with torch.no_grad():
            logits = model(input_ids=input_ids, attn_mask=attn_mask, pred_in
            predictions = torch.argmax(logits, dim=-1)


        for i in range(input_ids.size(0)):
            #filter out tokens
            valid_indices = (labels[i] != -100)
            correct = (predictions[i][valid_indices] == labels[i][valid_indi
            total_correct += correct
            total_predictions += valid_indices.sum().item()


    acc = total_correct / total_predictions if total_predictions > 0 else 0
    print(f"Accuracy: {acc:.4f}")


evaluate_token_accuracy(model,loader)
```

```
Accuracy: 0.9663
```

6. Span-Based evaluation

While the accuracy score in part 5 is encouraging, an accuracy-based evaluation is problematic for two reasons. First, most of the target labels are actually O. Second, it only tells us that per-token prediction works, but does not directly evaluate the SRL performance.

Instead, SRL systems are typically evaluated on micro-averaged precision, recall, and F1-score for predicting labeled spans.

More specifically, for each sentence/predicate input, we run the model, decode the output, and extract a set of labeled spans (from the output and the target labels). These

spans are (i,j,label) tuples.

We then compute the true_positives, false_positives, and false_negatives based on these spans.

In the end, we can compute

- Precision: true_positive / (true_positives + false_positives) , that is the number of correct spans out of all predicted spans.
- Recall: true_positives / (true_positives + false_negatives) , that is the number of correct spans out of all target spans.
- F1-score: (2 *precision* recall) / (precision + recall)

For example, consider

	[CLS]	The	judge	scheduled	to	preside	over	his	trial	was	remov
	0	1	2	3	4	5	6	7	8	9	1O
target	[CLS]	B-ARG1	I-ARG1	B-V	B-ARG2	I-ARG2	I-ARG2	I-ARG2	I-ARG2	O	O
prediction	[CLS]	B-ARG1	I-ARG1	B-V	I-ARG2	I-ARG2	O	O	O	O	O

The target spans are (1,2,"ARG1"), and (4,8,"ARG2").

The predicted spans would be (1,2,"ARG1"), (14,14,"ARGM-TMP"). Note that in the prediction, there is no proper ARG2 span because we are missing the B-ARG2 token, so this span should not be created.

So for this sentence we woudl get: true_positives: 1 false_positives: 1 false_negatives: 1

TODO: Complete the function evaluate_spans that performs the span-based evaluation on the given model and data loader. You can use the provided extract_spans function, which returns the spans as a dictionary. For example {(1,2): "ARG1", (4,8):"ARG2"}

In [89]:
```
def extract_spans(labels):
    spans = {} # map (start,end) ids to label
    current_span_start = 0
    current_span_type = ""
    inside = False
    for i, label in enumerate(labels):
        if label.startswith("B"):
            if inside:
                if current_span_type != "V":
                    spans[(current_span_start,i)] = current_span_type
            current_span_start = i
            current_span_type = label[2:]
            inside = True
        elif inside and label.startswith("O"):
            if current_span_type != "V":
                spans[(current_span_start,i)] = current_span_type
            inside = False
        elif inside and label.startswith("I") and label[2:] != current_span_
            if current_span_type != "V":
                spans[(current_span_start,i)] = current_span_type
            inside = False
    return spans
```

In [94]:
```
def evaluate_spans(model, loader):

    total_tp = 0  # Total true positives
    total_fp = 0  # Total false positives
    total_fn = 0  # Total false negatives

    for idx, batch in enumerate(loader):
        ids, mask, targets, pred_mask = (
            batch['ids'].to('cuda'),
            batch['mask'].to('cuda'),
            batch['targets'].to('cuda'),
            batch['pred'].to('cuda')
        )

        with torch.no_grad():
            logits = model(input_ids=ids, attn_mask=mask, pred_indicator=pre

        for j in range(logits.size(0)):
            logit = logits[j]
            target = targets[j]

            # process non padded tokens
            non_padding_mask = target != -100
            predicted = torch.argmax(logit, dim=1)[non_padding_mask]
            true = target[non_padding_mask]

            # predictions to labels
            predicted_labels = [id_to_role.get(int(idx), 'O') for idx in pre
```

```
            true_labels = [id_to_role.get(int(idx), 'O') for idx in true]

            # extract spans from labels
            predicted_spans = extract_spans(predicted_labels)
            target_spans = extract_spans(true_labels)

            tp_set = set(predicted_spans.keys()) & set(target_spans.keys())
            total_tp += sum(1 for span in tp_set if predicted_spans[span] ==

            fp_set = set(predicted_spans.keys()) - set(target_spans.keys())
            total_fp += len(fp_set)

            fn_set = set(target_spans.keys()) - set(predicted_spans.keys())
            total_fn += len(fn_set)

    # precision, recall, and f1 score
    if total_tp + total_fp > 0 and total_tp + total_fn > 0:
        precision = total_tp / (total_tp + total_fp)
        recall = total_tp / (total_tp + total_fn)
        f1_score = 2 * precision * recall / (precision + recall)
    else:
        precision, recall, f1_score = 0, 0, 0

    print(f"Overall P: {precision:.4f}  Overall R: {recall:.4f}  Overall F1:


evaluate_spans(model, loader)
```

```
Overall P: 0.8924  Overall R: 0.8965  Overall F1: 0.8945
```

In my evaluation, I got an F score of 0.82 (which slightly below the state-of-the art in 2018)

OPTIONAL:

Repeat the span-based evaluation, but print out precision/recall/f1-score for each role separately.