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	case	today	.
B-ARG1	I-ARG1	B-V schedule.01	B-ARG2	I-ARG2	I-ARG2	I-ARG2	I-ARG2	O	O	O	O	O	O	O

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

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

and not all predicates need to be verbs

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

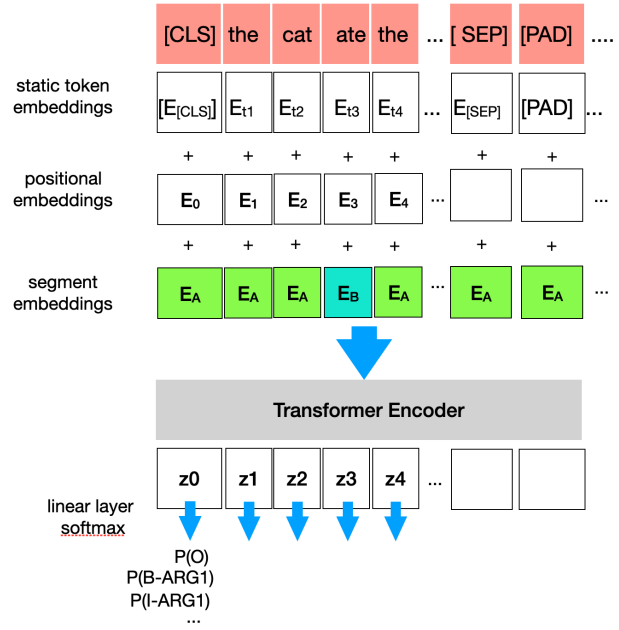
The SRL system will be implemented in PyTorch. We will use BERT (in the implementation provided by the Huggingface transformers library) to compute contextualized token representations and a custom classification head to predict semantic roles. We will fine-tune the pretrained BERT model on the SRL task.

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.

```
import torch
torch.cuda.is_available()
```

```
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:

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

```
--2024-12-10 23:03:05--  https://storage.googleapis.com/4705-bert-srl-data/ontonotes_srl.zip
Resolving storage.googleapis.com (storage.googleapis.com)... 172.217.218.207, 142.251.31.207, 142.251.18
Connecting to storage.googleapis.com (storage.googleapis.com)|172.217.218.207|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 12369688 (12M) [application/zip]
Saving to: 'ontonotes_srl.zip'

ontonotes_srl.zip   100%[===================>]  11.80M  10.6MB/s    in 1.1s

2024-12-10 23:03:06 (10.6 MB/s) - 'ontonotes_srl.zip' saved [12369688/12369688]
```

```
! unzip ontonotes_srl.zip
```

```
! unzip ontonotes_srl.zip
```

```
Archive:  ontonotes_srl.zip
  inflating: propbank_dev.tsv
  inflating: propbank_test.tsv
  inflating: propbank_train.tsv
  inflating: role_list.txt
```

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.

```
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())
```

```
len(role_to_id)
```

```
53
```

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

Double-click (or enter) to edit

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.

```
from transformers import BertTokenizerFast
tokenizer = BertTokenizerFast.from_pretrained('bert-base-uncased', do_lower_case=True)
tokenizer.tokenize("This is an unbelievably boring test sentence.")
```

```
['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
(['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.

```
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):
        tokens = tokenizer.tokenize(word)
        tokenized_sentence.extend(tokens)
        labels.extend([label])
        # if not label.startswith("O") and len(tokens) > 1:
        #     new_label = "I" + label[1:]
        #     labels.extend([new_label] * (len(tokens) - 1))
        if label != "O":
            labels.extend(["I-" + "-".join(label.split("-")[1:])] * (len(tokens) - 1))
        else:
            labels.extend(["O"] * (len(tokens) - 1))
    return tokenized_sentence, labels


tokenize_with_labels("the fancyful penguin devoured yummy fish .".split(), "B-ARG0 I-ARG0 I-ARG0 B-V B-ARG1
```

```
(['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)

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

class SrlData(Dataset):

    def __init__(self, filename):

        super(SrlData, self).__init__()

        self.max_len = 128 # the max number of tokens inputted to the transformer.

        self.tokenizer = BertTokenizerFast.from_pretrained('bert-base-uncased', do_lower_case=True)

        self.items = []
        # complete this method
        line_idx = 0
        sentence, tags = [], []
        for line in open(filename, "r", encoding="utf-8"):
            line_idx += 1
            res = line_idx % 4
            if res == 2:
                sentence = line.strip().split()
            elif res == 0:
                tags = line.strip().split()
                self.items.append(tuple(tokenize_with_labels(sentence, tags, self.tokenizer)))


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

    def __getitem__(self, k):
        tokens, tags = self.items[k]
        if len(tokens)  > self.max_len - 2:
            tokens = tokens[:self.max_len - 2]
            tags = tags[:self.max_len - 2]
        attn_size = len(tokens) + 2
        attn_mask = torch.cat((torch.ones(attn_size), torch.zeros(self.max_len - attn_size)))
        def preprocess(words, max_len):
            words = ["[CLS]"] + words + ["[SEP]"]
            words += ["[PAD]"] * (max_len - len(words))
            return words
        tokens_tensor = self.tokenizer.convert_tokens_to_ids(preprocess(tokens, self.max_len))
        default_id = role_to_id["O"]
        tags_tensor = [role_to_id[tag] if tag in role_to_id.keys() else default_id for tag in preprocess(tag
        pred = torch.zeros(self.max_len)
        try:
            pred_idx = tags_tensor.index(role_to_id["B-V"])
            pred[pred_idx] = 1
        except ValueError:
            pass

        #complete this method
```

```
        return {'ids': torch.tensor(tokens_tensor, dtype = torch.long), #token_tensor,
                'mask': attn_mask.to(torch.long), #attn_mask,
                'targets': torch.tensor(tags_tensor, dtype = torch.long), #label_tensor,
                'pred': pred.to(torch.long), #pred_tensor
               }
```

```
# Reading the training data takes a while for the entire data because we preprocess all data offline
data = SrlData("propbank_train.tsv")
```

```
data[0]
```

```
{'ids': tensor([  101,  2057, 26438,  2135, 13260,  2017,  2000,  3422,  1037,  2569,
          3179,  1997,  2408,  2859,  1012,   102,     0,     0,     0,     0,
             0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
             0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
             0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
             0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
             0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
             0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
             0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
             0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
             0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
             0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
             0,     0,     0,     0,     0,     0,     0,     0]),
 'mask': tensor([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0]),
 'targets': tensor([   1,    5,   19,   43,   28,    6,    8,   32,   32,   32,   32,   32,
           32,   32,    4,    2, -100, -100, -100, -100, -100, -100, -100, -100,
         -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -100,
         -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -100,
         -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -100,
         -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -100,
         -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -100,
         -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -100,
         -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -100, -100,
         -100, -100, -100, -100, -100, -100, -100, -100]),
 'pred': tensor([0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0])}
```

2. Model Definition

```
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.

```
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 us to train the classifier by i
        # 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 encoding explained in the introd
        # In our segment encoding, 1 indicates the predicate, and 0 indicates everything else.
        bert_output =  self.encoder(input_ids=input_ids, attention_mask=attn_mask, token_type_ids=pred_indic

        enc_tokens = bert_output[0] # the result of encoding the input with BERT
        logits = self.classifier(enc_tokens) #feed into the classification layer to produce scores for each

        # Note that we are only interested in the argmax for each token, so we do not have to normalize
        # to a probability distribution using softmax. The CrossEntropyLoss loss function takes this into ac
        # It essentially computes the softmax first and then computes the negative log-likelihood for the ta
        return logits

model = SrlModel().to("cuda") # create new model and store weights in GPU memory
```

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:

```
# pick an item from the dataset. Then run
item = data[0]
ids, mask, targets, pred = [t.unsqueeze(0).to("cuda") for t in item.values()]
outputs = model(ids, mask, pred)
outputs.shape
```

```
C:\Users\PEI\anaconda3\envs\aigc\Lib\site-packages\transformers\models\bert\modeling_bert.py:439: UserWa
  attn_output = torch.nn.functional.scaled_dot_product_attention(
torch.Size([1, 128, 53])
```

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.

```
import math
-math.log(1 / len(role_to_id), math.e)
```

```
3.970291913552122
```

```
loss_function = CrossEntropyLoss(ignore_index = -100, reduction='mean')

# complete this. Note that you still have to provide a (batch_size, input_pos)
# tensor for each parameter, where batch_size =1
outputs = model(ids, mask, pred).permute(0, 2, 1)
loss = loss_function(outputs, targets)
loss.item()   #this should be approximately the score from the previous cell
```

```
4.002588272094727
```

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
```

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.

```
pred_tags_id = torch.argmax(outputs, dim = 1)
pred_tags_id
```

```
tensor([[48, 51, 13, 37, 13, 36,  7, 33, 22, 35, 36, 47, 23, 36,  2,  2, 13, 13,
         13, 17, 17, 13, 13, 13, 13, 51, 13, 13, 13, 13, 13, 13, 13, 22, 34, 22,
         23, 23, 23, 22, 22, 22, 13, 11, 17, 13, 13, 13, 13, 13, 13,  8, 13, 13,
         13, 13, 13, 13, 13,  8,  8, 28, 34, 22, 23, 28, 16, 22, 22, 15, 51, 17,
         13, 51, 15, 13, 13, 13, 13, 15, 13, 13, 13,  6, 15, 13, 13, 13, 13,  8,
         28, 34, 22, 23, 23, 28, 22, 22, 11, 17, 13, 13, 31, 31, 15, 13, 13, 13,
         15, 51, 28, 13, 13, 13, 13, 13, 13, 13, 22, 27, 22, 22, 22, 23,  5, 22,
         51, 13]], device='cuda:0')
```

```
pred_tags = [id_to_role[x] if x > 0 else 'O' for x in pred_tags_id.squeeze().cpu().numpy()]
pred_tags
```

```
 'B-ARGM-CAU',
 'B-ARG1',
 'B-ARGM-DIS',
 'B-ARGM-CAU',
 'B-ARGM-CAU',
 'B-ARGM-CAU',
 'B-ARGM-CAU',
 'B-ARG2',
 'B-V',
 'I-ARG4',
 'B-ARGM-PRD',
 'B-ARGM-PRP',
 'B-ARGM-PRP',
 'B-V',
 'B-ARGM-PRD',
 'B-ARGM-PRD',
 'B-ARGM-ADJ',
 'B-ARGM-GOL',
 'B-ARGM-CAU',
 'B-ARGM-CAU',
 'I-ARG1-DSP',
 'I-ARG1-DSP',
 'B-ARGM-DIS',
 'B-ARGM-CAU',
 'B-ARGM-CAU',
 'B-ARGM-CAU',
 'B-ARGM-DIS',
 'I-ARGM-LVB',
 'B-V',
 'B-ARGM-CAU',
 'B-ARGM-CAU',
 'B-ARGM-CAU',
 'B-ARGM-CAU',
 'B-ARGM-CAU',
 'B-ARGM-CAU',
 'B-ARGM-CAU',
 'B-ARGM-PRD',
 'B-ARGM-LVB',
 'B-ARGM-PRD',
 'B-ARGM-PRD',
 'B-ARGM-PRD',
 'B-ARGM-PRP',
 'B-ARG0',
 'B-ARGM-PRD',
 'I-ARGM-LVB',
 'B-ARGM-CAU']
```

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.

```
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 below so 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.

```
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
    tr_preds, tr_labels = [], []
    # put model in training mode
    model.train()

    for idx, batch in enumerate(loader):

        # Get the encoded data for this batch and push it to the GPU
        ids = batch['ids'].to(device, dtype = torch.long)
        mask = batch['mask'].to(device, dtype = torch.long)
        targets = batch['targets'].to(device, dtype = torch.long)
        pred_mask = batch['pred'].to(device, dtype = torch.long)

        # Run the forward pass of the model
        logits = model(input_ids=ids, attn_mask=mask, pred_indicator=pred_mask)
        loss = loss_function(logits.transpose(2,1), targets)
        tr_loss += loss.item()
        # print("Batch loss: ", loss.item()) # can comment out if too verbose.

        nb_tr_steps += 1
        nb_tr_examples += targets.size(0)

        if idx % 100==0:
            torch.cuda.empty_cache() # can help if you run into memory issues
            curr_avg_loss = tr_loss/nb_tr_steps
            print(f"Current average loss: {curr_avg_loss}")

        # Compute accuracy for this batch
        mask = (targets != -100)
        pred = torch.argmax(logits,dim=2)
        matching = torch.sum(pred[mask] == targets[mask])
        predictions = torch.sum(mask)
        print("Batch accuracy: ", matching.item() / predictions.item())
        tr_preds.append(matching.item())
        tr_labels.append(predictions.item())

        # Run the backward pass to update parameters
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    epoch_loss = tr_loss / nb_tr_steps
    print(f"Training loss epoch: {epoch_loss}")
    epoch_acc = sum(tr_preds) / sum(tr_labels)
    print(f"Training accuracy epoch: {epoch_acc}")
```

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

```
train()
```

```
Batch accuracy:  0.9653767820773931
Batch accuracy:  0.9646408839779006
Batch accuracy:  0.9252100840336135
Batch accuracy:  0.966282165039929
Batch accuracy:  0.9785575048732943
Batch accuracy:  0.9238095238095239
Batch accuracy:  0.9323583180987203
Batch accuracy:  0.9460112812248187
Batch accuracy:  0.8675373134328358
Batch accuracy:  0.9533932951757972
Batch accuracy:  0.9426559356136821
Batch accuracy:  0.917470525187567
Batch accuracy:  0.8959183673469387
Batch accuracy:  0.9176598049837487
Batch accuracy:  0.9533333333333334
Batch accuracy:  0.9040156709108716
Batch accuracy:  0.9415584415584416
Batch accuracy:  0.9672131147540983
Batch accuracy:  0.9219653179190751
Batch accuracy:  0.9522842639593908
Batch accuracy:  0.901174168297456
Batch accuracy:  0.8866886688668867
Batch accuracy:  0.9533468559837728
Batch accuracy:  0.9609895337773549
Batch accuracy:  0.9355140186915888
Batch accuracy:  0.9369565217391305
Batch accuracy:  0.92497320471597
Batch accuracy:  0.9177350427350427
Batch accuracy:  0.9091760299625468
Batch accuracy:  0.936542669584245
Batch accuracy:  0.923728813559322
Batch accuracy:  0.9492957746478873
Batch accuracy:  0.9126436781609195
Batch accuracy:  0.8905742145178764
Batch accuracy:  0.9328793774319066
Current average loss: 0.3832588611078857
Batch accuracy:  0.9796126401630989
Batch accuracy:  0.9632850241545894
Batch accuracy:  0.9373737373737374
Batch accuracy:  0.9120370370370371
Batch accuracy:  0.8106129917657823
Batch accuracy:  0.9066410009624639
Batch accuracy:  0.9261954261954262
Batch accuracy:  0.9424460431654677
Batch accuracy:  0.9631391200951248
Batch accuracy:  0.9323553382233088
Batch accuracy:  0.8683926645091694
Batch accuracy:  0.9151398264223722
Batch accuracy:  0.9116279069767442
Batch accuracy:  0.9293044469783353
Batch accuracy:  0.9685534591194969
Batch accuracy:  0.9161406672678089
Batch accuracy:  0.9317365269461078
Batch accuracy:  0.9494505494505494
Training loss epoch: 0.38301696275192226
Training accuracy epoch: 0.8931394703444948
```

```
# model.load_state_dict(torch.load("srl_model_fulltrain_1epoch_finetune_1e-05.pt"))
torch.save(model.state_dict(), "srl_model_fulltrain_1epoch_finetune_1e-05.pt")
```

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

```
train()
```

```
Batch accuracy:  0.9037636903765091
Batch accuracy:  0.9734151329243353
Batch accuracy:  0.9567839195979899
Batch accuracy:  0.9031674208144796
Batch accuracy:  0.9376114081996435
Batch accuracy:  0.9682539682539683
Batch accuracy:  0.9157007376185459
Batch accuracy:  0.9393364928909953
Batch accuracy:  0.9348739495798319
Batch accuracy:  0.9295272078501338
Batch accuracy:  0.960730593607306
Batch accuracy:  0.9371196754563894
Batch accuracy:  0.9377551020408164
Batch accuracy:  0.9641760491299898
Batch accuracy:  0.9547368421052631
Batch accuracy:  0.9669603524229075
Batch accuracy:  0.9621513944223108
Batch accuracy:  0.9338235294117647
Batch accuracy:  0.911
Batch accuracy:  0.9330188679245283
Batch accuracy:  0.9584158415841584
Batch accuracy:  0.9549738219895288
Batch accuracy:  0.9591633466135459
Batch accuracy:  0.9110889110889111
Batch accuracy:  0.9758454106280193
Batch accuracy:  0.9096774193548387
Batch accuracy:  0.9019823788546255
Batch accuracy:  0.9401140684410646
Batch accuracy:  0.9102428722280888
Batch accuracy:  0.9455357142857143
Batch accuracy:  0.9471624266144814
Batch accuracy:  0.9298928919182083
Current average loss: 0.19342926452513154
Batch accuracy:  0.9635732870771899
Batch accuracy:  0.9395604395604396
Batch accuracy:  0.8990256864481843
Batch accuracy:  0.9427609427609428
Batch accuracy:  0.9902702702702703
Batch accuracy:  0.921911421911422
Batch accuracy:  0.9667049368541906
Batch accuracy:  0.970113085621971
Batch accuracy:  0.9780941949616648
Batch accuracy:  0.9257322175732218
Batch accuracy:  0.9539295392953929
Batch accuracy:  0.9447852760736196
Batch accuracy:  0.9437439379243453
Batch accuracy:  0.98635477582846
Batch accuracy:  0.9621421975992613
Batch accuracy:  0.9345018450184502
Batch accuracy:  0.9702564102564103
Batch accuracy:  0.9554367201426025
Training loss epoch: 0.19337006357082417
Training accuracy epoch: 0.940139334072977
```

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.

```
torch.save(model.state_dict(), "srl_model_fulltrain_2epoch_finetune_1e-05.pt")
```

4. Decoding

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

model = SrlModel().to('cuda')
```

```
model.load_state_dict(torch.load("srl_model_fulltrain_2epoch_finetune_1e-05.pt"))
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.

```
tokens = "A U. N. team spent an hour inside the hospital , where it found evident signs of shelling and gunf


def decode_output(logits): # it will be useful to have this in a separate function later on
    """
    Given the model output, return a list of string labels for each token.
    """
    pred_id = torch.argmax(logits, dim = 1)
    return [id_to_role[x] if x > 0 else 'O' for x in pred_id.squeeze().cpu().numpy()]


def label_sentence(tokens, pred_idx):

    # complete this function to prepare token_ids, attention mask, predicate mask, then call the model.
    # Decode the output to produce a list of labels.
    # max_len = data.max_len
    max_len = 128
    # token_ids = tokenizer.convert_tokens_to_ids(tokens)
    attn_size = len(tokens) + 2
    attn_mask = torch.cat((torch.ones(attn_size), torch.zeros(max_len - attn_size)))
    tokens = ["[CLS]"] + tokens + ["[SEP]"]
    tokens += ["[PAD]"] * (max_len - len(tokens))
    tokens_tensor = tokenizer.convert_tokens_to_ids(tokens)
    tokens_tensor = torch.tensor(tokens_tensor).unsqueeze(0).to(torch.long)
    attn_mask = attn_mask.unsqueeze(0).to(torch.long)
    pred = torch.zeros(max_len)
    pred[pred_idx] = 1
    pred = pred.unsqueeze(0).to(torch.long)
    logits = model(tokens_tensor.to(device), attn_mask.to(device), pred.to(device))
    return decode_output(logits.squeeze(0))[1:]


# Now you should be able to run
device = 'cuda'
label_test = label_sentence(tokens, 13) # Predicate is "found"
for x, y in zip(tokens, label_test):
    print(x, y)
```

```
A O
U. O
N. O
team O
spent O
an O
hour O
inside O
the O
hospital O
, O
where O
it B-V
found O
```

```
evident 0
signs 0
of 0
shelling 0
and 0
gunfire 0
. 0
```

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.

```
dev_data = SrlData("propbank_dev.tsv") # Takes a while because we preprocess all data offline
```

```
from torch.utils.data import DataLoader
loader = DataLoader(dev_data, batch_size = 1, shuffle = False)
```

```
# 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-05.pt"))
model = model.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.

```
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 in the data.

    # iterate over the data here.
    with torch.no_grad():
        for idx, batch in enumerate(loader):
            ids = batch['ids'].to(device, dtype = torch.long)
            mask = batch['mask'].to(device, dtype = torch.long)
            targets = batch['targets'].to(device, dtype = torch.long)
            pred_mask = batch['pred'].to(device, dtype = torch.long)
            logits = model(input_ids=ids, attn_mask=mask, pred_indicator=pred_mask)
            pos_mask = (targets != -100)
            pred_ids = torch.argmax(logits,dim=2)
            matching = torch.sum(pred_ids[pos_mask] == targets[pos_mask])
            predictions = torch.sum(pos_mask)
            total_correct += matching
            total_predictions += predictions

    acc = total_correct / total_predictions
    print(f"Accuracy: {acc}")


evaluate_token_accuracy(model, loader)
```

```
Accuracy: 0.9371520280838013
```

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	removed	from	the	case	today	.
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
target	[CLS]	B-ARG1	I-ARG1	B-V	B-ARG2	I-ARG2	I-ARG2	I-ARG2	I-ARG2	O	O	O	O	O	O	O
prediction	[CLS]	B-ARG1	I-ARG1	B-V	I-ARG2	I-ARG2	O	O	O	O	O	O	O	O	B-ARGM-TMP	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 would 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"}

```
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_type:
            if current_span_type != "V":
                spans[(current_span_start,i)] = current_span_type
            inside = False
    return spans



def evaluate_spans(model, loader):
    total_tp = 0
    total_fp = 0
    total_fn = 0

    for idx, batch in enumerate(loader):

        ids = batch['ids'].to(device, dtype = torch.long)
        mask = batch['mask'].to(device, dtype = torch.long)
        targets = batch['targets'].to(device, dtype = torch.long)
        pred_mask = batch['pred'].to(device, dtype = torch.long)
        logits = model(input_ids=ids, attn_mask=mask, pred_indicator=pred_mask)
        # TODO: filter out the padded positions ([PAD] target tokens)
        pos_mask = (targets != -100)
        pred_ids = torch.argmax(logits,dim=2)[pos_mask]
        targets = targets[pos_mask]
        pred_labels = [id_to_role[x] if x > 0 else 'O' for x in pred_ids.squeeze().cpu().numpy()]
        target_labels = [id_to_role[x] if x > 0 else 'O' for x in targets.squeeze().cpu().numpy()]
        pred_spans = extract_spans(pred_labels)
        target_spans = extract_spans(target_labels)
        tp, fp = 0, 0
        for pred_rng, pred_label in pred_spans.items():
            if pred_rng in target_spans.keys() and target_spans[pred_rng] == pred_label:
                tp += 1
            else:
                fp += 1
        fn = len(target_spans) - tp
        total_tp += tp
        total_fp += fp
        total_fn += fn

    total_p = total_tp / (total_tp + total_fp)
    total_r = total_tp / (total_tp + total_fn)
    total_f = (2 * total_p *total_r) / (total_p + total_r)
```

```
    print(f"Overall P: {total_p}  Overall R: {total_r}  Overall F1: {total_f}")

evaluate_spans(model, loader)
```

```
Overall P: 0.8117638018289002  Overall R: 0.8319663370124099  Overall F1: 0.8217409181326192
```

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.