

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		B-ARG2	I-ARG2	I-ARG2	I-ARG2	O	O	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	case	today	.
B-ARG1	I-ARG1	I-ARG1		I-ARG1	I-ARG1	I-ARG1	I-ARG1	O	B-V		B-ARG2	I-ARG2	I-ARG2	B-ARGM-TMP
remove.01														

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	O	O	O	O	O	O
try.02														

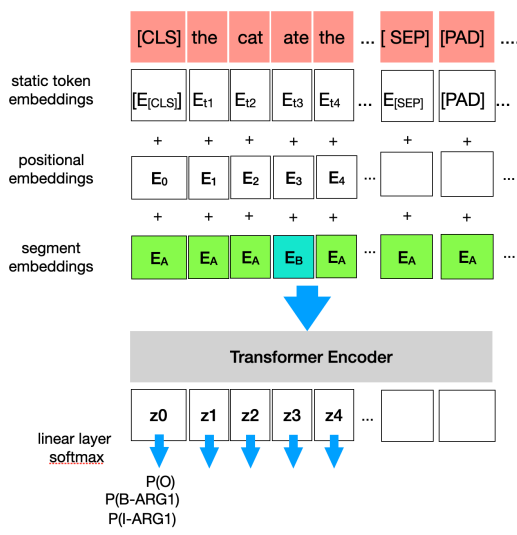
The SRL system will be implemented in [PyTorch](#). We will use BERT (in the implementation provided by the [Huggingface transformers](#) library) to compute contextualized token representations and a custom classification head to predict semantic roles. We will fine-tune the pretrained BERT model on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

sub-word tokenization in Masked Language Model to get good tokenizations. [CLS] stands for classification, [MASK] is the masked token, and [SEP] is the separator.

Subword tokenization: Byte-pair encoding

- goal: learn a good sub-word tokenization strategy
- start with a vocabulary of individual characters (+ special symbols)
- tokenize the training corpus, then merge the most frequently occurring adjacent pair of tokens, creating a new token
- merges only allowed inside of words (determined by white spaces)
- repeat until k new tokens have been created

✓ 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
```

```
--2024-12-11 23:14:37-- https://storage.googleapis.com/4705-bert-srl-data/ontonotes_srl.zip
Resolving storage.googleapis.com (storage.googleapis.com)... 142.251.2.207, 74.125.137.207, 142.250.101.207, ...
Connecting to storage.googleapis.com (storage.googleapis.com)|142.251.2.207|: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 24.1MB/s in 0.5s
```

```
2024-12-11 23:14:38 (24.1 MB/s) - 'ontonotes_srl.zip' saved [12369688/12369688]
```

```
! 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 0      0      0      0      0      0      0
```

- 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 propbank 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())
```

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.")
```

🔗 /usr/local/lib/python3.10/dist-packages/huggingface_hub/utils/_auth.py:94: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (<https://huggingface.co/settings/tokens>), set it as secret. You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or datasets.

```
warnings.warn(
tokenizer_config.json: 100% 48.0/48.0 [00:00<00:00, 1.61kB/s]
vocab.txt: 100% 232k/232k [00:00<00:00, 1.74MB/s]
tokenizer.json: 100% 466k/466k [00:00<00:00, 6.45MB/s]
config.json: 100% 570/570 [00:00<00:00, 33.9kB/s]
['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.

```
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):
        word_tokens = tokenizer.tokenize(word)
        if not word_tokens:
            continue
        tokenized_sentence.append(word_tokens[0])
        if label.startswith("B-"):
            continuation_label = "I-" + label[2:]
        else:
            continuation_label = label
        labels.append(label)

        for token in word_tokens[1:]:
            tokenized_sentence.append(token)
            labels.append(continuation_label)
    return tokenized_sentence, labels

#tokenized_sentence = tokenized_sentence + tokenizer.tokenize(word)# complete this loop
#labels = labels + [label] * len(tokenizer.tokenize(word)) # complete this loop

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']])
```

1.2 Loading the Dataset

Next, we are creating a PyTorch [Dataset](#) class. This class acts as a container 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 = []
        file = open(filename)
        content = file.readlines()
        for i in range(0, len(content), 4):
            tokenized_sentence, labels = tokenize_with_labels(content[i+1].strip().split("\t"), content[i+3].strip().split("\t"), self.tokenizer)
            print((tokenized_sentence, labels))
            self.items.append((tokenized_sentence, labels))

        # complete this method

    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]

        tokens.insert(0, "[CLS]")
        tags.insert(0, "[CLS]")
        tokens.append("[SEP]")
        tags.append("[SEP]")
        tokens.extend(["[PAD]"] * (self.max_len - len(tokens)))
```

[illegible]

```
data.__getitem__(0)
```

7/19

✓ 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 itself.
        # 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 introduction.
        # 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_indicator)
        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 tag.

        # 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 account.
        # It essentially computes the softmax first and then computes the negative log-likelihood for the target classes.
        return logits
```

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

```
model.safetensors: 0% | 0.00/440M [00:00<?, ?B/s]
```

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 = item['ids'].to('cuda', dtype = torch.long)
mask = item['mask'].to('cuda', dtype = torch.long)
pred = item['pred'].to('cuda', dtype = torch.long)
outputs = model(ids, mask, pred)
outputs.shape
```

```
torch.Size([1, 128, 53])
```

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

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

$$-\ln\left(\frac{1}{\text{num_labels}}\right).$$

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)
targets = item['targets'].to('cuda', dtype = torch.long)
loss = loss_function(outputs.transpose(2,1), targets)
loss.item() #this should be approximately the score from the previous cell
```

4.0632524490356445

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,  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',
```

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.

```
results = torch.zeros(128, dtype=torch.long)
resultsTokens = []
```

```
outputs = outputs.squeeze(0)
argmaxPredictions = outputs.argmax(dim =1)
print(argmaxPredictions.shape)
print(argmaxPredictions)
for i in range(len(argmaxPredictions)):
    tokenId = argmaxPredictions[i].item()
    if tokenId not in id_to_role:
        resultsTokens.append("[UNK]")
    else:
        resultsTokens.append(id_to_role[tokenId])
print(resultsTokens)
```

```
torch.Size([128])
tensor([34, 34,  6, 29, 29, 36, 29, 44,  4, 12,  9, 30, 18,  9, 18, 14, 52, 18,
        29, 18, 18, 18, 29, 29, 29, 29, 29, 29, 18, 18, 18, 18, 29, 29, 29, 29,
        18, 29, 29, 18, 18, 29, 18, 18, 18, 18, 29, 18, 18, 29, 29, 29, 29, 18,
        29, 29, 18, 18, 18, 18, 18, 29, 29, 18, 29, 29, 18, 29, 18, 18, 18,
        18, 29, 29, 18, 29, 18, 29, 29, 29, 29, 29, 29, 18, 18, 29, 29, 18,
        14, 18, 29, 18, 18, 14, 29, 29, 29, 29, 29, 18, 18, 18, 18, 18, 14, 18,
        18, 18, 34, 14, 14, 18, 18, 52, 52, 34, 52, 52, 18, 52, 28, 18, 18, 18,
        29, 14], device='cuda:0')
['I-ARG4', 'I-ARG4', 'B-ARG1', 'I-ARG0', 'I-ARG0', 'I-ARGM-ADV', 'I-ARG0', 'I-ARGM-MOD', 'O', 'B-ARGM-ADV', 'B-ARG3', 'I-ARG1', 'B-ARGM-
```

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)
        ids = ids[0, :, :]
        mask = mask[0, :, :]
        targets = targets[0, :, :]
        pred_mask = pred_mask[0, :, :]

        print(ids.shape, mask.shape, targets.shape, pred_mask.shape)

        # 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
    matching = torch.sum(torch.argmax(logits,dim=2) == targets)
    predictions = torch.sum(torch.where(targets!=-100,0,1))
    accuracy = matching / predictions
    print(f"Average Accuracy: {accuracy}")

    # 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}")
```

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

```
train()
```



Streaming output truncated to the last 5000 lines.

```
Batch loss: 2.6176912784576416
Average Accuracy: 0.3199999928474426
torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128])
Batch loss: 1.3308223485946655
Average Accuracy: 0.6896551847457886
torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128])
Batch loss: 0.40997278690338135
Average Accuracy: 1.0
torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128])
Batch loss: 1.615568995475769
Average Accuracy: 0.40909090638160706
torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128])
Batch loss: 0.5462942719459534
Average Accuracy: 0.9833333492279053
torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128])
Batch loss: 3.4061200618743896
Average Accuracy: 0.3529411852359772
torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128])
Batch loss: 2.3347690105438232
Average Accuracy: 0.28333333134651184
torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128])
Batch loss: 0.8319981694221497
Average Accuracy: 0.8095238208770752
torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128])
Batch loss: 1.1882147789001465
Average Accuracy: 0.6716417670249939
torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128])
Batch loss: 2.0461342334747314
Average Accuracy: 0.190476194024086
torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128])
Batch loss: 0.4794996976852417
Average Accuracy: 1.0
torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128])
Batch loss: 1.1223000288009644
Average Accuracy: 0.7441860437393188
torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128])
Batch loss: 0.4767235219478607
Average Accuracy: 1.0
torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128])
Batch loss: 0.9365212917327881
Average Accuracy: 0.84375
torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128])
Batch loss: 2.383810520172119
Average Accuracy: 0.3777777850627899
torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128])
Batch loss: 0.5237948894500732
Average Accuracy: 0.9444444179534912
torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128])
Batch loss: 0.7062304615974426
Average Accuracy: 0.9090909361839294
torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128])
Batch loss: 0.4307894706726074
Average Accuracy: 0.9615384340286255
torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128])
Batch loss: 0.7351340055465698
Average Accuracy: 0.6153846383094788
torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128])
```

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

```
train()
```



```

Batch loss: 0.23708420100212097
Average Accuracy: 1.0
torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128])
Batch loss: 0.906179666519165
Average Accuracy: 0.8181818127632141
torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128])
Batch loss: 0.7329984307289124
Average Accuracy: 0.8888888955116272
torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128])
Batch loss: 1.2420952320098877
Average Accuracy: 0.5333333611488342
torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128])
Batch loss: 0.497203528881073
Average Accuracy: 1.0
torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128])
Batch loss: 0.6410763263702393
Average Accuracy: 0.8799999952316284
torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128])
Batch loss: 0.8712614178657532
Average Accuracy: 0.8367347121238708
torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128])
Batch loss: 1.2525110244750977
Average Accuracy: 0.84375
torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128])
Batch loss: 0.5001534223556519
Average Accuracy: 0.875
torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128])
Batch loss: 1.801880121231079
Average Accuracy: 0.29032257199287415
torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128])
Batch loss: 0.4544088542461395
Average Accuracy: 0.9444444179534912
torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128])
Batch loss: 1.4727084636688232
Average Accuracy: 0.6129032373428345
torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128])
Batch loss: 0.45367056131362915
Average Accuracy: 0.9921875
torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128])
Batch loss: 0.4922294616699219
Average Accuracy: 0.9615384340286255
torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128])
Batch loss: 0.5991157293319702
Average Accuracy: 0.914893627166748
torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128]) torch.Size([1, 128])
Batch loss: 0.8161030411720276
Average Accuracy: 0.6842105388641357
Training loss epoch: 1.0902709057521105

```

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(), "sr1_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("sr1_model_fulltrain_2epoch_finetune_1e-05.pt"))
model = model.to('cuda')

```

 <ipython-input-28-56056f4e4de6>:4: FutureWarning: You are using `torch.load` with `weights_only=False` (the current default value), which is unsafe. To silence this warning, use `torch.load` with `weights_only=True` (the default in the future) to load only the weights and not the full state dictionary. To silence this warning, use `torch.load` with `weights_only=True` (the default in the future) to load only the weights and not the full state dictionary.

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 gunfire ".split()
```

```
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.
    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.
    """
    #pred_id = torch.argmax(logits, dim = 1)
    # check argmax == 0
    #return [id_to_role[x] if x > 0 else "0" for x in pred_id.squeeze().cpu().numpy()]
    print(logits.shape)
    outputs_argmax = torch.argmax(logits, dim=2)
    print(outputs_argmax.shape)
    outputs_tags = []
    for label_id in outputs_argmax[0]:
        if label_id.item() in id_to_role:
            outputs_tags.append(id_to_role[label_id.item()])
        else:
            outputs_tags.append("0")
    print(outputs_tags)
    return outputs_tags

def label_sentence(tokens, pred_idx):
    """
    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.
    """
    # 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_length = 128
    attn_length = len(tokens) + 2
    attn_mask = torch.cat((torch.ones(attn_length), torch.zeros(max_length - attn_length)))
    tokens = ["[CLS]"] + tokens + ["[SEP]"]
    tokens = tokens + ["[PAD]"] * (max_length - len(tokens))
    token_tensor = tokenizer.convert_tokens_to_ids(tokens)
    token_tensor = torch.tensor(token_tensor, dtype=torch.long).unsqueeze(0)
    attn_mask = attn_mask.unsqueeze(0).to(torch.long)
    pred_arr = torch.zeros(max_length)
    pred_arr[pred_idx] = 1
    pred_arr = pred_arr.unsqueeze(0).to(torch.long)
    logits = model(token_tensor.to(device), attn_mask.to(device), pred_arr.to(device))
    return decode_output(logits.squeeze(0))[1:]
    """
    token_ids = ["[CLS]"]
    attention = [0]*128
    attention[0] = 1
    pred = [0]*128
    pred[pred_idx + 1] = 1
    for i in range(min(len(tokens), 127)):
        token_ids.append(tokens[i])
        attention[i+1] = 1

    token_ids.append("[SEP]")
    attention[min(len(tokens) + 1, 127)] = 1
    while len(token_ids) < 128:
        token_ids.append("[PAD]")

    token_ids = tokenizer.convert_tokens_to_ids(token_ids)
    token_tensor = torch.tensor(token_ids, dtype=torch.long).unsqueeze(0).to('cuda')
    attention = torch.tensor(attention, dtype=torch.long).unsqueeze(0).to('cuda')
    pred = torch.tensor(pred, dtype=torch.long).unsqueeze(0).to('cuda')

    output = model(token_tensor, attention, pred)
    return decode_output(output)
```

Now you should be able to run

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

```
→ torch.Size([1, 128, 53])
torch.Size([1, 128])
['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '[CLS]', '0', '0', 'B-V', '0', '0', '0', '0', '0', '0', '[CLS]', '0', '0', '0',
['A', '0'],
('U.', '0'),
('N.', '0'),
('team', '0'),
('spent', '0'),
('an', '0'),
('hour', '0'),
('inside', '0'),
('the', '0'),
('hospital', '0'),
(',', '0'),
('where', '[CLS]'),
('it', '0'),
('found', '0'),
('evident', 'B-V'),
('signs', '0'),
('of', '0'),
('shelling', '0'),
('and', '0'),
('gunfire', '0'),
('.', '0')]
```

The expected output is somethign like this:

```
('A', '0'),
('U.', '0'),
('N.', '0'),
('team', '0'),
('spent', '0'),
('an', '0'),
('hour', '0'),
('inside', '0'),
('the', 'B-ARGM-LOC'),
('hospital', 'I-ARGM-LOC'),
(',', '0'),
('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'),
('.', '0'),
```

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
```



[('do', 'you', 'need', 'me', 'to', 'spell', 'out', 'what', 'has', 'been', 'repeated', 'in', 'this', 'thread', 'ten', 'times', 'and',
(['without', 'the', 'jefferson', '', 's', 'freedoms', 'spelled', 'out', '', 'whichever', 'political', 'party', 'is', 'dominant', '
(['to', 'the', 'historian', 'douglas', 'brink', '##ley', '', 'democratic', 'electoral', 'victories', 'in', 'november', '2006', 'spel
(['to', 'the', 'historian', 'douglas', 'brink', '##ley', '', 'democratic', 'electoral', 'victories', 'in', 'november', '2006', 'spel
(['to', 'the', 'historian', 'douglas', 'brink', '##ley', '', 'democratic', 'electoral', 'victories', 'in', 'november', '2006', 'spel
(['it', 'must', 'suck', 'to', 'be', 'hated', 'by', 'everyone', '.', '.', '.'], [0, 0, 0, 0, 0, B-V, 0, 0, 0, 0, 0, 0, 0, 0,
(['if', 'being', 'fat', 'did', 'not', 'suck', '', 'then', 'the', 'billboard', 'would', 'n', '', 't', 'upset', 'her', '.'], [0, 0, E
(['if', 'being', 'fat', 'did', 'not', 'suck', '', 'then', 'the', 'billboard', 'would', 'n', '', 't', 'upset', 'her', '.'], [0, 0, C
(['it', 'sucks', 'cu', '##z', 'then', 'all', 'you', 'can', 'add', 'is', 'yahoo', '!'], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
(['during', 'europe', '', 's', '1', '', '000', '-', 'year', '', 'dark', 'ages', '', 'which', 'began', 'after', 'once', '-'],
(['during', 'europe', '', 's', '1', '', '000', '-', 'year', '', 'dark', 'ages', '', 'which', 'began', 'after', 'once', '-'],
(['during', 'europe', '', 's', '1', '', '000', '-', 'year', '', 'dark', 'ages', '', 'which', 'began', 'after', 'once', '-'],
(['sydney', '-', 'australia', '', 's', 'centre', '-', 'left', 'opposition', 'leader', 'kevin', 'rudd', 'swept', 'into', 'power', 'ir
(['they', '', 're', 'warm', '', 'human', '', 'en', '##vel', '##hoping', 'in', 'a', 'slow', 'country', 'lang', '##uo', '##n', 'on',
(['they', '', 're', 'warm', '', 'human', '', 'en', '##vel', '##hoping', 'in', 'a', 'slow', 'country', 'lang', '##uo', '##n', 'on',
(['disastrous', 'fires', 'that', 'swept', 'through', 'southern', 'greece', 'last', 'month', 'destroyed', 'more', 'than', '97', ',',
(['how', 'many', 'thrash', '##ed', 'some', 'poor', 'native', 'gun', 'to', '##ter', 'to', 'death', '.'], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
(['theresa', 'may', 'is', 'probably', 'worth', 'five', 'or', 'six', 'celia', 'barlow', '##s', '', 'though', 'to', 'really', 'thrash
(['if', 'line', '2', 'meant', 'she', 'was', 'ugly', '', 'then', 'these', 'lines', 'would', 'have', 'to', 'mean', 'she', 'was', 'also
(['if', 'line', '2', 'meant', 'she', 'was', 'ugly', '', 'then', 'these', 'lines', 'would', 'have', 'to', 'mean', 'she', 'was', 'also
(['all', 'that', 'trouble', 'is', 'worth', 'it', '', 'though', '', 'for', 'the', 'second', 'i', 'touch', 'her', 'soft', 'furry', 'r
(['the', 'et', '##ique', '##tte', 'for', 'use', '##net', 'always', 'has', 'been', 'to', 'post', 'in', 'context', '', 'at', 'the', 't
(['the', 'et', '##ique', '##tte', 'for', 'use', '##net', 'always', 'has', 'been', 'to', 'post', 'in', 'context', '', 'at', 'the', 't
(['i', 'will', 'make', 'an', 'exception', 'sometimes', 'and', 'trim', 'down', 'for', 'high', 'speeds', '', 'since', 'that', 'is', 'r
(['you', 'were', 'probably', 'lucky', 'that', 'none', 'of', 'the', 'op', 'cases', 'tripped', 'the', 'bounds', '', 'however', 'some',
(['you', 'were', 'probably', 'lucky', 'that', 'none', 'of', 'the', 'op', 'cases', 'tripped', 'the', 'bounds', '', 'however', 'some',
(['we', 'have', 'literally', 'been', '', 'tripped', 'up', '', 'by', 'a', 'vest', '##ig', '##ial', 'organ', 'more', 'than', 'half',
(['we', 'have', 'literally', 'been', '', 'tripped', 'up', '', 'by', 'a', 'vest', '##ig', '##ial', 'organ', 'more', 'than', 'half',
(['it', 'boil', '##s', 'down', 'to', 'capitals', 'being', 'reserved', 'for', 'macro', '##s', '', 'and', 'that', 'is', 'inherited',
(['it', 'boil', '##s', 'down', 'to', 'capitals', 'being', 'reserved', 'for', 'macro', '##s', '', 'and', 'that', 'is', 'inherited',
(['the', 'author', 'of', 'the', 'controversial', '1994', 'book', '', 'the', 'bell', 'curve', '', 'and', 'champion', 'of', 'iq', 'te
(['the', 'author', 'of', 'the', 'controversial', '1994', 'book', '', 'the', 'bell', 'curve', '', 'and', 'champion', 'of', 'iq', 'te
(['the', 'author', 'of', 'the', 'controversial', '1994', 'book', '', 'the', 'bell', 'curve', '', 'and', 'champion', 'of', 'iq', 'te
(['essentially', '', 'my', 'choice', 'boil', '##s', 'down', 'to', 'ri', '##tz', 'for', 'best', 'quality', '', 'cost', '##co', 'for
(['lots', 'of', 'people', 'do', '-', 'chat', '##s', 'boiled', 'then', 'covered', 'in', 'butter', 'and', 'a', 'sp', '##rin', '##kle',
(['lots', 'of', 'people', 'do', '-', 'chat', '##s', 'boiled', 'then', 'covered', 'in', 'butter', 'and', 'a', 'sp', '##rin', '##kle',
(['lots', 'of', 'people', 'do', '-', 'chat', '##s', 'boiled', 'then', 'covered', 'in', 'butter', 'and', 'a', 'sp', '##rin', '##kle',
(['lots', 'of', 'people', 'do', '-', 'chat', '##s', 'boiled', 'then', 'covered', 'in', 'butter', 'and', 'a', 'sp', '##rin', '##kle',
(['the', 'answer', 'to', 'them', 'would', 'be', 'a', 'dl', 'good', 'enough', 'to', 'generate', 'some', 'pass', 'rush', 'on', 'its',
(['the', 'answer', 'to', 'them', 'would', 'be', 'a', 'dl', 'good', 'enough', 'to', 'generate', 'some', 'pass', 'rush', 'on', 'its',
(['the', 'answer', 'to', 'them', 'would', 'be', 'a', 'dl', 'good', 'enough', 'to', 'generate', 'some', 'pass', 'rush', 'on', 'its',
(['much', 'of', 'this', 'cash', 'was', 'money', 'i', 'saved', 'from', 'not', 'owning', 'cc', '##s', 'and', 'having', 'those', 'compar
(['so', 'easy', 'to', 'use', 'and', 'faster', 'than', 'ever', 'before', '', 'just', 'few', 'clicks', 'to', 'rip', 'a', 'dvd', '!'],
(['the', 'point', 'came', 'up', 'in', 'the', 'course', 'of', 'the', 'following', 'report', 'about', 'a', 'mini', '-', 'bus', 'bombing
(['the', 'point', 'came', 'up', 'in', 'the', 'course', 'of', 'the', 'following', 'report', 'about', 'a', 'mini', '-', 'bus', 'bombing

```
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')
```

```
> ipython-input-35-c26dac4e8ccd:3: FutureWarning: You are using `torch.load` with `weights_only=False` (the current default value), which
model.load_state_dict(torch.load("src1 model fulltrain 2epoch finetune 1e-05.pt"))
```

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.
```

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

    ids = ids.squeeze(0) # remove extra dimension for batch size in input_ids
    mask = mask.squeeze(0) # remove extra dimension for batch size in attention_mask
    pred_mask = pred_mask.squeeze(0) # remove extra dimension for batch size in predicate indicator

    logits = model(input_ids=ids, attn_mask=mask, pred_indicator=pred_mask)
    predLogic = torch.where(targets ==100, 0, 1) & torch.where(targets == 1,0,1) & torch.where(targets == 2,0,1)
    matching = torch.sum((torch.argmax(logits,dim=2) == targets) & predLogic)
    predictions = torch.sum(predLogic)
    total_correct += matching
    total_predictions += predictions
    acc = total_correct / total_predictions
    print(f"Accuracy: {acc}")

    print(f"total_correct: {total_correct}")
    print(f"total_predictions: {total_predictions}")
    if idx % 100 == 0:
        print(f"idx:{idx}")

evaluate_token_accuracy(model, loader)
```




```
total_predictions: 1285850
Accuracy: 0.1397668421268463
total_correct: 179734
total_predictions: 1285956
```

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

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('cuda', dtype = torch.long)
        mask = batch['mask'].to('cuda', dtype = torch.long)
        targets = batch['targets'].to('cuda', dtype = torch.long)
        pred_mask = batch['pred'].to('cuda', dtype = torch.long)
```

```

# Reshape the input tensors to (batch_size, sequence_length)
ids = ids.squeeze(0) # remove extra dimension for batch size in input_ids
mask = mask.squeeze(0) # remove extra dimension for batch size in attention_mask
pred_mask = pred_mask.squeeze(0) # remove extra dimension for batch size in predicate indicator
#targets = targets.squeeze(0) # Remove extra dimension

logits = model(input_ids=ids, attn_mask=mask, pred_indicator=pred_mask)
predLabels = decode_output(logits)

#print(targets)

targetLabels = []
# Iterate through each element of the sequence
for label_id in range(len(targets)):
    if targets[label_id] in id_to_role:
        targetLabels.append(id_to_role[label_id.item()])
    else:
        targetLabels.append("0")

CLSIndex = 1
#CLSIndex = targetLabels.index("[CLS]")
SEPIndex = 14 #dummy value to at least make the program run
#SEPIndex = targetLabels.index("[SEP]")

NtargetLabels = targetLabels[CLSIndex+1:SEPIndex]
NpredLabels = predLabels[CLSIndex+1:SEPIndex]

predSpansDict = extract_spans(NpredLabels)
predSpansset = set(extract_spans(NpredLabels))
targetSpansDict = extract_spans(NtargetLabels)
targetSpansset = set(extract_spans(NtargetLabels))

print(predSpansDict)
print(targetSpansDict)

for span, label in predSpansDict.items():
    if span in targetSpansDict and targetSpansDict[span] == label:
        total_tp += 1
    elif span in targetSpansDict and targetSpansDict[span] != label:
        total_fp += 1

for span, label in targetSpansDict.items():
    if (span in predSpansDict and predSpansDict[span] != label) or span not in predSpansDict:
        total_fn += 1

print(f"curr_tp: {total_tp}")
print(f"curr_fp: {total_fp}")
print(f"curr_fn: {total_fn}")

total_p = total_tp / (total_tp + total_fp) if (total_tp + total_fp) != 0 else 0 # Avoid division by zero
total_r = total_tp / (total_tp + total_fn) if (total_tp + total_fn) != 0 else 0 # Avoid division by zero
total_f = (2 * total_p * total_r) / (total_p + total_r) if (total_p + total_r) != 0 else 0 # Avoid division by zero

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

evaluate_spans(model, loader)

```



```
curr_tp: 0
curr_fp: 0
curr_fn: 0
torch.Size([1, 128, 53])
torch.Size([1, 128])
['[CLS]', '[CLS]', '0', '0', 'I-V', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0']
{}
{}
curr_tp: 0
curr_fp: 0
curr_fn: 0
torch.Size([1, 128, 53])
torch.Size([1, 128])
['[CLS]', '[CLS]', '0', '0', 'I-V', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0']
{}
{}
curr_tp: 0
curr_fp: 0
curr_fn: 0
torch.Size([1, 128, 53])
torch.Size([1, 128])
['[CLS]', '[CLS]', '0', '0', '0', '0', '0', '0', '0', '0', 'B-V', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0']
{}
{}
curr_tp: 0
curr_fp: 0
curr_fn: 0
torch.Size([1, 128, 53])
torch.Size([1, 128])
['[CLS]', '[CLS]', 'B-ARGM-DIS', '0', '0', '0', '0', '0', '0', 'B-V', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0']
{({0, 1}: 'ARGM-DIS')}
{}
curr_tp: 0
curr_fp: 0
curr_fn: 0
torch.Size([1, 128, 53])
torch.Size([1, 128])
```

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