# 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<br>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<br>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<br>try.02 | O | O | O | O | O | O | O |

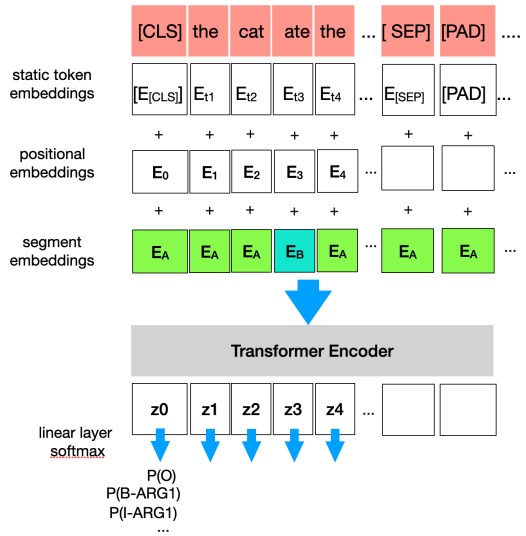
The SRL system will be implemented in PyTorch. We will use BERT (in the implementation provided by the Huggingface transformers library) to compute contextualized token representations and a custom classification head to predict semantic roles. We will fine-tune the pretrained BERT model on the SRL task.

### 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
```

```
--2024-12-12 03:45:01--  https://storage.googleapis.com/4705-bert-srl-data/ontonotes_srl.zip
Resolving storage.googleapis.com (storage.googleapis.com)... 173.194.193.207, 173.194.194.207, 142.250.152.207, ...
Connecting to storage.googleapis.com (storage.googleapis.com)|173.194.193.207|: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  --.-KB/s    in 0.04s

2024-12-12 03:45:01 (285 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  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())
```

Note that we are also mapping the '[PAD]' token to the value -100. This allows

- List item
- List item

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 s
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, 3.83kB/s]
vocab.txt: 100%  232k/232k [00:00<00:00, 4.91MB/s]
tokenizer.json: 100%  466k/466k [00:00<00:00, 5.21MB/s]
config.json: 100%  570/570 [00:00<00:00, 40.4kB/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):
        # Tokenize the word into subwords
        subwords = tokenizer.tokenize(word)

        # Add the first subword with the original label
        if subwords:
            tokenized_sentence.append(subwords[0])
            labels.append(label)

            # For subsequent subwords, append the I- version of the label
            if len(subwords) > 1:
                labels.extend([f"I-{label.split('-')[1]}" for _ in subwords[1:]])
                tokenized_sentence.extend(subwords[1:])
        else:
            # In case there are no subwords (which is rare), handle gracefully
            tokenized_sentence.append(word)
            labels.append(label)



    return tokenized_sentence, labels


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 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

        # Read the dataset file (tsv format)
        self.load_data(filename)

    def load_data(self, filename):
        # Load the data from the TSV file
        with open(filename, 'r', encoding='utf-8') as file:
            current_item = []
            for line in file:
                line = line.strip()
                if line.startswith('ontonotes/') or line == '':
                    if current_item:
                        self.process_item(current_item)
                        current_item = []
                else:
                    current_item.append(line)
            if current_item:
                self.process_item(current_item)

    def process_item(self, item):
        sentence = item[0].split()
        predicate = item[1].strip()
        bio_tags = item[2].split()
        tokens_with_labels = list(zip(sentence, bio_tags))
        self.items.append((tokens_with_labels, predicate))




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

    def __getitem__(self, k):
        tokens_with_labels, predicate = self.items[k]
        tokens, labels = zip(*tokens_with_labels)

        # Add [CLS] and [SEP] tokens
        tokens = ['[CLS]'] + list(tokens) + ['[SEP]']
        labels = ['O'] + list(labels) + ['O']

        # Convert tokens to ids and pad
        token_ids = self.tokenizer.convert_tokens_to_ids(tokens)
        label_ids = [role_to_id.get(label, role_to_id['O']) for label in labels]

        # Padding
        padding_length = self.max_len - len(token_ids)
        if padding_length > 0:
            token_ids += [self.tokenizer.pad_token_id] * padding_length
            label_ids += [role_to_id['O']] * padding_length
        else:
```

```
            token_ids = token_ids[:self.max_len]
            label_ids = label_ids[:self.max_len]

        # Create attention mask
        attn_mask = [1 if token_id != self.tokenizer.pad_token_id else 0 for token_id in token_ids]

        # Create predicate indicator mask
        pred_tensor = [0] * self.max_len
        if 'B-V' in labels:
            pred_index = labels.index('B-V')
            if pred_index < self.max_len:
                pred_tensor[pred_index] = 1

        # Convert to tensors
        token_tensor = torch.tensor(token_ids, dtype=torch.long).unsqueeze(0)
        label_tensor = torch.tensor(label_ids, dtype=torch.long).unsqueeze(0)
        attn_mask_tensor = torch.tensor(attn_mask, dtype=torch.long).unsqueeze(0)
        pred_tensor = torch.tensor(pred_tensor, dtype=torch.long).unsqueeze(0)



        #complete this method
        return {
            'ids': token_tensor,
            'mask': attn_mask_tensor,
            'targets': label_tensor,
            'pred': pred_tensor
        }


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

## 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: 100%                                   440M/440M [00:02<00:00, 227MB/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

# outputs = model(ids, mask, pred)

# Assuming you have already defined and loaded your model
# Move model to GPU if available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)

dev_loader = DataLoader(data, batch_size=1, shuffle=False)

# Get a single item from the dev set
single_item = next(iter(dev_loader))

# Extract the components and move them to the correct device
ids = single_item['ids'].squeeze(1).to(device)  # Shape: [1, 128]
mask = single_item['mask'].squeeze(1).to(device)  # Shape: [1, 128]
pred = single_item['pred'].squeeze(1).to(device)  # Shape: [1, 128]
targets = single_item['targets'].squeeze(1).to(device)  # Shape: [1, 128]

# Print shapes to verify
print("Input IDs shape:", ids.shape)
print("Attention Mask shape:", mask.shape)
print("Predicate Indicator shape:", pred.shape)
print("Target Labels shape:", targets.shape)

# Run the model
with torch.no_grad():
    outputs = model(ids, mask, pred)

# Print output shape
print("Model Output shape:", outputs.shape)

# Process predictions if needed
predictions = torch.argmax(outputs, dim=-1)
print("Predictions shape:", predictions.shape)
print("First few predictions:", predictions[0][:10])
print("First few targets:", targets[0][:10])
```

```
Input IDs shape: torch.Size([1, 128])
Attention Mask shape: torch.Size([1, 128])
Predicate Indicator shape: torch.Size([1, 128])
Target Labels shape: torch.Size([1, 128])
Model Output shape: torch.Size([1, 128, 53])
Predictions shape: torch.Size([1, 128])
First few predictions: tensor([ 8, 14, 14, 49, 33, 52,  3, 52, 49, 36], device='cuda:0')
First few targets: tensor([ 4,  5, 19, 28,  6,  8, 32, 32, 32, 32], device='cuda:0')
```

**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')
num_labels = len(role_to_id)
outputs_reshaped = outputs.view(-1, num_labels)  # Shape: [batch_size * sequence_length, num_classes]
targets_reshaped = targets.view(-1)  # Shape: [batch_size * sequence_length]
```

```
loss = loss_function(outputs_reshaped, targets_reshaped)
print("Computed loss:", loss.item())

# Calculate and print the expected initial loss for comparison
initial_loss = -math.log(1 / num_labels, math.e)
print("Expected initial loss:", initial_loss)
# 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)
# loss = loss_function(...)
# loss.item()   #this should be approximately the score from the previous cell
# Compute the loss
predictions = torch.argmax(outputs, dim=-1)  # Shape: [batch_size, sequence_length]

# Print the predictions
print("Predictions:")
print(predictions)
# Decode predictions to actual labels
decoded_predictions = [[id_to_role[pred.item()] for pred in seq] for seq in predictions]

# Print decoded predictions
print("\nDecoded Predictions:")
for seq in decoded_predictions:
    print(seq)
```

```
Computed loss: 4.013655662536621
Expected initial loss: 3.970291913552122
Predictions:
tensor([[ 8, 14, 14, 49, 33, 52,  3, 52, 49, 36, 10, 41, 33, 10, 52, 11, 11, 11,
         24, 11, 11, 24, 11, 11, 11, 11, 11, 11, 24, 11, 11, 11, 11, 11, 11, 11,
         11, 11, 11, 11, 11, 11,  5, 11, 11, 11, 11, 24, 24, 11, 11, 11, 11, 24,
         11, 11, 11, 24, 24, 24, 24, 24, 11, 11, 11, 11, 11,  5, 11, 11, 11, 11,
         24, 11, 24, 11, 11, 24, 11, 11, 11, 11, 11, 11, 11, 24, 11, 11, 24, 24,
         11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11,
         11, 11, 11, 11, 11, 11, 11, 11, 11, 24, 11, 11, 11, 11, 11, 11, 11, 24,
         11, 24]], device='cuda:0')

Decoded Predictions:
['B-ARG2', 'B-ARGM-DIR', 'B-ARGM-DIR', 'I-ARGM-TMP', 'I-ARG3', 'I-V', '[prd]', 'I-V', 'I-ARGM-TMP', 'I-ARGM-ADV', 'B-ARG4', 'I-ARGM
```

**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',
```

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.

```
Start coding or generate with AI.
```

## 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 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.

```
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_correct = 0
    total_valid = 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).squeeze(1)  # Remove extra dimension
        mask = batch['mask'].to(device, dtype=torch.long).squeeze(1)  # Remove extra dimension
        pred_mask = batch['pred'].to(device, dtype=torch.long).squeeze(1)  # Remove extra dimension
        targets = batch['targets'].to(device, dtype=torch.long).squeeze(1)  # Remove extra dimension

        # # Verify shapes (optional)
        # print("ids shape:", ids.shape)
        # print("mask shape:", mask.shape)
        # print("pred_mask shape:", pred_mask.shape)
        # print("targets shape:", targets.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
        predictions = torch.argmax(logits, dim=2)  # Shape: [batch_size, seq_len]

        # Mask out [PAD] tokens (where target == -100)
        valid_mask = targets != -100  # Shape: [batch_size, seq_len]

        correct_preds = torch.sum((predictions == targets) & valid_mask)  # Count correct predictions ignoring [PAD]
        valid_tokens = torch.sum(valid_mask)  # Count valid tokens (excluding [PAD])

        total_correct += correct_preds.item()
        total_valid += valid_tokens.item()

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

    epoch_loss = tr_loss / nb_tr_steps
    epoch_accuracy = total_correct / total_valid

    print(f"Training loss epoch: {epoch_loss}")
    print(f"Training accuracy epoch: {epoch_accuracy}")
```

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

```
train()
```

```
Batch loss:  0.05705595413446264
Batch loss:  0.057836759835481644
Batch loss:  0.048255786299705505
Batch loss:  0.11533121764659882
Batch loss:  0.10019412636756897
Batch loss:  0.05304483696818352
Batch loss:  0.05480283126235008
Batch loss:  0.02678680419921875
Batch loss:  0.0739184319972992
Batch loss:  0.05495186895132065
Batch loss:  0.04402704909443855
Batch loss:  0.03803986310958862
Batch loss:  0.03926940634846687
Batch loss:  0.06669806689023972
Batch loss:  0.04561386629939079
Batch loss:  0.036706309765577316
Batch loss:  0.04966650903224945
Batch loss:  0.05039387196302414
Batch loss:  0.045958828181028366
Batch loss:  0.054396696388721466
Batch loss:  0.08447334170341492
Batch loss:  0.06545116752386093
Batch loss:  0.04149201139807701
Batch loss:  0.04007282480597496
Batch loss:  0.07069820910692215
Batch loss:  0.02034011110663414
Batch loss:  0.03658866882324219
Batch loss:  0.04610573872923851
Batch loss:  0.059232957661151886
Batch loss:  0.051261208951473236
Batch loss:  0.14087119698524475
Batch loss:  0.047457434237003326
Current average loss: 0.07484272063354694
Batch loss:  0.06886084377765656
Batch loss:  0.057105645537376404
Batch loss:  0.06020304560661316
Batch loss:  0.10792865604162216
Batch loss:  0.039761632680892944
Batch loss:  0.029866429045796394
Batch loss:  0.06910938024520874
Batch loss:  0.10132233053445816
Batch loss:  0.02379114180803299
Batch loss:  0.03893979266285896
Batch loss:  0.07531799376010895
Batch loss:  0.04789898172020912
Batch loss:  0.05672780051827431
Batch loss:  0.07469314336776733
Batch loss:  0.03531168773770332
Batch loss:  0.06588427722454071
Batch loss:  0.04385865852236748
Training loss epoch: 0.07481463499390761
Training accuracy epoch: 0.9780107163433687
```

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

```
train()
```

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')
```

```
<ipython-input-22-56056f4e4de6>:4: FutureWarning: You are using `torch.load` with `weights_only=False` (the current default value),
  model.load_state_dict(torch.load("srl_model_fulltrain_2epoch_finetune_1e-05.pt"))
```

**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.

    """

    predictions = torch.argmax(logits, dim=-1).squeeze(0)  # Shape: [sequence_length]
    return [id_to_role[pred.item()] for pred in predictions]



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.
    tokenized_input = tokenizer(tokens,
                                is_split_into_words=True,
                                return_tensors="pt",
                                padding=True,
                                truncation=True,
                                max_length=128)

    # Prepare input tensors
    token_ids = tokenized_input['input_ids'].to(device)
    attention_mask = tokenized_input['attention_mask'].to(device)

    # Create predicate mask
    pred_mask = torch.zeros_like(token_ids)
    pred_mask[0, pred_idx + 1] = 1  # +1 for [CLS] token
    pred_mask = pred_mask.to(device)

    # Call the model
    with torch.no_grad():
        logits = model(input_ids=token_ids, attn_mask=attention_mask, pred_indicator=pred_mask)

    # Decode the output
    labels = decode_output(logits)

    # Align labels with original tokens (removing [CLS] and [SEP] tokens)
    aligned_labels = labels[1:len(tokens)+1]

    return aligned_labels



# Now you should be able to run

label_test = label_sentence(tokens, 13) # Predicate is "found"
zip(tokens, label_test)
for token, label in zip(tokens, label_test):
    print(f"{token}: {label}")
```

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

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 = 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.

```
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():  # Disable gradient computation
        for idx, batch in enumerate(loader):
            # Load batch data to device
            ids = batch['ids'].to(device).squeeze(1)
            mask = batch['mask'].to(device).squeeze(1)
            pred_mask = batch['pred'].to(device).squeeze(1)
            targets = batch['targets'].to(device).squeeze(1)

            # Forward pass to get predictions
            logits = model(input_ids=ids, attn_mask=mask, pred_indicator=pred_mask)
            predictions = torch.argmax(logits, dim=2)  # Shape: [batch_size, seq_len]

            # Mask out invalid positions: [PAD], [CLS], and [SEP]
            valid_mask = (targets != -100)  # Exclude padded positions
```

```
            correct_preds = torch.sum((predictions == targets) & valid_mask)
            valid_tokens = torch.sum(valid_mask)

            # Update counts
            total_correct += correct_preds.item()
            total_predictions += valid_tokens.item()

            if idx % 10 == 0:  # Print every 10 batches
                print(f"Batch {idx}:")
                print(f"  Running total accuracy: {total_correct / total_predictions:.4f}")


    acc = total_correct / total_predictions

    # acc = total_correct / total_predictions
    return acc


dev_data = SrlData("propbank_dev.tsv")
dev_loader = DataLoader(dev_data, batch_size=32, shuffle=False)

# Evaluate token accuracy
accuracy = evaluate_token_accuracy(model, dev_loader)
```

```
Batch 530:
  Running total accuracy: 0.9826
Batch 540:
  Running total accuracy: 0.9826
Batch 550:
  Running total accuracy: 0.9826
Batch 560:
  Running total accuracy: 0.9826
Batch 570:
  Running total accuracy: 0.9826
Batch 580:
  Running total accuracy: 0.9824
Batch 590:
  Running total accuracy: 0.9824
Batch 600:
  Running total accuracy: 0.9823
Batch 610:
  Running total accuracy: 0.9822
Batch 620:
  Running total accuracy: 0.9821
Batch 630:
  Running total accuracy: 0.9822
Batch 640:
  Running total accuracy: 0.9821
Batch 650:
  Running total accuracy: 0.9822
Batch 660:
  Running total accuracy: 0.9821
Batch 670:
  Running total accuracy: 0.9821
Batch 680:
  Running total accuracy: 0.9822
Batch 690:
  Running total accuracy: 0.9821
Batch 700:
  Running total accuracy: 0.9822
Batch 710:
  Running total accuracy: 0.9822
Batch 720:
  Running total accuracy: 0.9821
Batch 730:
  Running total accuracy: 0.9821
Batch 740:
  Running total accuracy: 0.9820
Batch 750:
  Running total accuracy: 0.9821
Batch 760:
  Running total accuracy: 0.9820
Batch 770:
  Running total accuracy: 0.9820
Batch 780:
  Running total accuracy: 0.9820
Batch 790:
  Running total accuracy: 0.9820
Batch 800:
  Running total accuracy: 0.9820
Batch 810:
  Running total accuracy: 0.9819
```

## 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 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"}

```
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):
        label_str = id_to_role[label]
        if label_str.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_str[2:]
            inside = True
        elif inside and label_str.startswith("O"):
            if current_span_type != "V":
                spans[(current_span_start,i)] = current_span_type
            inside = False
        elif inside and label_str.startswith("I") and label_str[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

    with torch.no_grad():  # Disable gradient computation during evaluation
        for idx, batch in enumerate(loader):
            # Load batch data to device
            ids = batch['ids'].to(device).squeeze(1)
            mask = batch['mask'].to(device).squeeze(1)
            pred_mask = batch['pred'].to(device).squeeze(1)
            targets = batch['targets'].to(device).squeeze(1)

            # Forward pass to get predictions
            logits = model(input_ids=ids, attn_mask=mask, pred_indicator=pred_mask)
            predictions = torch.argmax(logits, dim=2)  # Shape: [batch_size, seq_len]

            if idx % 1000 == 0:  # Print every 10 batches
                print(f"Batch {idx}:")
```

```
            # Decode the predicted and target spans
            for i in range(len(targets)):
                target_spans = extract_spans(targets[i].cpu().numpy())  # Extract target spans
                pred_spans = extract_spans(predictions[i].cpu().numpy())  # Extract predicted spans

                # Compare the predicted spans with the target spans
                for span, label in pred_spans.items():
                    if span in target_spans and target_spans[span] == label:
                        total_tp += 1  # True positive
                    else:
                        total_fp += 1  # False positive

                for span in target_spans:
                    if span not in pred_spans:
                        total_fn += 1  # False negative

            if (idx + 1) % 1000 == 0:
                precision = total_tp / (total_tp + total_fp) if (total_tp + total_fp) > 0 else 0
                recall = total_tp / (total_tp + total_fn) if (total_tp + total_fn) > 0 else 0
                f1 = (2 * precision * recall) / (precision + recall) if (precision + recall) > 0 else 0

                print(f"Running Precision: {precision:.4f}, Running Recall: {recall:.4f}, Running F1: {f1:.4f}")



    # Calculate precision, recall, and F1 score
    precision = total_tp / (total_tp + total_fp) if (total_tp + total_fp) > 0 else 0
    recall = total_tp / (total_tp + total_fn) if (total_tp + total_fn) > 0 else 0
    f1_score = (2 * precision * recall) / (precision + recall) if (precision + recall) > 0 else 0

    # Print out the overall precision, recall, and F1 score
    print(f"Overall Precision: {precision:.4f}")
    print(f"Overall Recall: {recall:.4f}")
    print(f"Overall F1 Score: {f1_score:.4f}")



    # for idx, batch in enumerate(loader):

    #     pass # compelte this


    # 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)
```

```
Batch 16000:
Running Precision: 0.7507, Running Recall: 0.8148, Running F1: 0.7814
Batch 17000:
Running Precision: 0.7518, Running Recall: 0.8151, Running F1: 0.7822
```