4705_fa24_hw4

November 28, 2024

1 COMS W4705 Spring 24

1.1 Homework 4 - Semantic Role Labelling with BERT

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

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

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

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

and not all predicates need to be verbs

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

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

1.1.1 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 (BERT pretraining objectives: MLM & NSP), 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).

1.2 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.

[1]:
```
import torch
if torch.cuda.is_available():
    DEVICE = 'cuda'
elif torch.backends.mps.is_available():
    DEVICE = 'mps'
else:
    DEVICE = 'cpu'
print(DEVICE)
```

```
mps
```

1.3 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
```

[ ]:
```
# ! unzip ontonotes_srl.zip
```

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
                schedule.01
B-ARG1  I-ARG1  B-V     B-ARG2  I-ARG2  I-ARG2  I-ARG2  I-ARG2  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.

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

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

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

1.4 Part 1 - Data Preparation

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

1.4.1 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.

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

[3]:
```
['this',
 'is',
 'an',
 'un',
 '##bel',
```

```
 '##ie',
 '##va',
 '##bly',
 'boring',
 'test',
 'sentence',
 '.']
```

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

```
>>> tokenize_with_labels("the fancyful penguin devoured yummy fish .".split(), "B-ARG0 I-ARG0 I
(['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.

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

    tokenized_sentence = []
    labels = []
```

```
    for word, label in zip(sentence, text_labels):
        bert_word = tokenizer.tokenize(word)
        tokenized_sentence += [*bert_word]
        labels.append(label)
        labels += [label.replace("B", "I") if "B-" in label else label] *␣
↪(len(bert_word) - 1)

    return tokenized_sentence, labels
```

[5]:
```
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)
```

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

1.4.2 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 = []

        with open("propbank_train.tsv", 'r') as datafile:
            while True:
                lines = [datafile.readline().strip() for _ in range(4)]
                if not all(lines):
                    break
                self.items.append(
                    tokenize_with_labels(
                        sentence=lines[1].split('\t'),
                        text_labels=lines[3].split('\t'),
```

```
                        tokenizer=self.tokenizer)
                    )

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

    def __getitem__(self, k):
        tokens, labels = self.items[k]
        tokens, labels = tokens[:self.max_len-2], labels[:self.max_len-2]
        tokens = ['[CLS]'] + tokens + ['[SEP]'] + ['[PAD]'] * (self.max_len -␣
↪len(tokens) - 2)
        tokens = torch.tensor(self.tokenizer.convert_tokens_to_ids(tokens),␣
↪dtype=torch.long)

        labels = torch.tensor(
            [role_to_id['[CLS]']] +
            [role_to_id.get(l, role_to_id['O']) for l in labels] +
            [role_to_id['[SEP]']] +
            [role_to_id['[PAD]']] * (self.max_len - len(labels) - 2),
            dtype=torch.long)

        #complete this method
        return {'ids': tokens,
                'mask': (tokens != 0).long(),
                'targets': labels,
                'pred': (labels == 28).long()
               }
```

[10]:
```
# Reading the training data takes a while for the entire data because we␣
↪preprocess all data offline
data = SrlData("propbank_train.tsv")
```

1.5 2. Model Definition

[39]:
```
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.

[40]:
```
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
```

[41]:
```
model = SrlModel().to(DEVICE) # create new model and store weights in GPU memory
```

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

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

[42]:
```
trial_item = data[85]
ids = trial_item['ids'].unsqueeze(0).to(DEVICE)
mask = trial_item['mask'].unsqueeze(0).to(DEVICE)
pred = trial_item['pred'].unsqueeze(0).to(DEVICE)

outputs = model(ids, mask, pred)
```

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.

[43]:
```
outputs.shape
```

[43]:
```
torch.Size([1, 128, 53])
```

[44]:
```
import math
-math.log(1 / len(role_to_id), math.e)
```

[44]:
```
3.970291913552122
```

[45]:
```
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)
target = trial_item['targets'].unsqueeze(0).to(DEVICE)
loss = loss_function(outputs.transpose(2,1), target) # loss_func expect output␣
 ↪in [N, num choices, ...]
loss.item()   #this should be approximately the score from the previous cell
```

[45]:
```
4.047802448272705
```

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
```

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.

[128]:
```
predicted_label_ids = torch.argmax(outputs, axis=2)
print(predicted_label_ids)

print([id_to_role.get(i.item(), id_to_role[-100]) for i in␣
 ↪predicted_label_ids[0]])
```

```
tensor([[33, 33, 39,  5,  4, 47, 33,  7, 47, 47,  8,  5, 39, 39, 30, 33, 47, 47,
         23,  4, 33,  5, 12, 33, 33, 33, 47, 33, 33, 33, 33, 33, 33, 33, 33, 47,
         47, 47, 33, 47, 33, 33, 24, 24, 33, 33, 24, 33, 47, 33, 33, 47, 33, 33,
         33, 33, 33, 33,  7, 33, 33, 47, 33, 47, 47, 33, 33, 33, 33, 33, 24, 33,
         33, 33, 33, 33, 16, 47, 47, 51, 47, 47, 33, 33, 33, 33, 33, 33, 33, 33,
         47, 33, 33, 47, 51, 47, 33, 33, 33, 33, 33, 24, 33, 33, 27, 20, 27, 47,
         33, 47, 47, 33, 33, 33, 33, 24, 33, 33, 33, 33, 27, 33, 33, 33, 33, 33,
         24, 24]], device='mps:0')
['I-ARG3', 'I-ARG3', 'I-ARGM-DIS', 'B-ARG0', 'O', 'I-ARGM-PRP', 'I-ARG3',
'B-ARG1-DSP', 'I-ARGM-PRP', 'I-ARGM-PRP', 'B-ARG2', 'B-ARG0', 'I-ARGM-DIS',
'I-ARGM-DIS', 'I-ARG1', 'I-ARG3', 'I-ARGM-PRP', 'I-ARGM-PRP', 'B-ARGM-PRP', 'O',
'I-ARG3', 'B-ARG0', 'B-ARGM-ADV', 'I-ARG3', 'I-ARG3', 'I-ARG3', 'I-ARGM-PRP',
'I-ARG3', 'I-ARG3', 'I-ARG3', 'I-ARG3', 'I-ARG3', 'I-ARG3', 'I-ARG3', 'I-ARG3',
'I-ARGM-PRP', 'I-ARGM-PRP', 'I-ARGM-PRP', 'I-ARG3', 'I-ARGM-PRP', 'I-ARG3',
'I-ARG3', 'B-ARGM-PRR', 'B-ARGM-PRR', 'I-ARG3', 'I-ARG3', 'B-ARGM-PRR',
'I-ARG3', 'I-ARGM-PRP', 'I-ARG3', 'I-ARG3', 'I-ARGM-PRP', 'I-ARG3', 'I-ARG3',
'I-ARG3', 'I-ARG3', 'I-ARG3', 'I-ARG3', 'B-ARG1-DSP', 'I-ARG3', 'I-ARG3',
'I-ARGM-PRP', 'I-ARG3', 'I-ARGM-PRP', 'I-ARGM-PRP', 'I-ARG3', 'I-ARG3',
'I-ARG3', 'I-ARG3', 'I-ARG3', 'B-ARGM-PRR', 'I-ARG3', 'I-ARG3', 'I-ARG3',
'I-ARG3', 'I-ARG3', 'B-ARGM-EXT', 'I-ARGM-PRP', 'I-ARGM-PRP', 'I-ARGM-LVB',
'I-ARGM-PRP', 'I-ARGM-PRP', 'I-ARG3', 'I-ARG3', 'I-ARG3', 'I-ARG3', 'I-ARG3',
'I-ARG3', 'I-ARG3', 'I-ARG3', 'I-ARGM-PRP', 'I-ARG3', 'I-ARG3', 'I-ARGM-PRP',
'I-ARGM-LVB', 'I-ARGM-PRP', 'I-ARG3', 'I-ARG3', 'I-ARG3', 'I-ARG3', 'I-ARG3',
'B-ARGM-PRR', 'I-ARG3', 'I-ARG3', 'B-ARGM-LVB', 'B-ARGM-MOD', 'B-ARGM-LVB',
'I-ARGM-PRP', 'I-ARG3', 'I-ARGM-PRP', 'I-ARGM-PRP', 'I-ARG3', 'I-ARG3',
'I-ARG3', 'I-ARG3', 'B-ARGM-PRR', 'I-ARG3', 'I-ARG3', 'I-ARG3', 'I-ARG3',
'B-ARGM-LVB', 'I-ARG3', 'I-ARG3', 'I-ARG3', 'I-ARG3', 'I-ARG3', 'B-ARGM-PRR',
'B-ARGM-PRR']
```

1.6 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.

[47]:
```
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.

[48]:
```
loss_function = CrossEntropyLoss(ignore_index = -100, reduction='mean')

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

def train():
    """
    Train the model for one epoch.
    """
    tr_loss = 0
    nb_tr_examples, nb_tr_steps = 0, 0
    tr_preds, tr_labels = [], []
    total_correct = 0
    total_predictions = 0
    # put model in training mode
    model.train()

    for idx, batch in enumerate(loader):

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

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

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

        if idx % 100==0:
            #torch.cuda.empty_cache() # can help if you run into memory issues
```

```
            curr_avg_loss = tr_loss/nb_tr_steps
            print(f"Current average loss: {curr_avg_loss}")

        # Compute accuracy for this batch
        matching = torch.sum(torch.argmax(logits, dim=2) == targets)
        predictions = torch.sum(torch.where(targets==-100, 0, 1))
        total_correct += matching
        total_predictions += predictions

        # 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_predictions if total_predictions !=␣
↪0 else 0  # Avoid division by zero
    print(f"Training loss epoch: {epoch_loss}")
    print(f"Average accuracy epoch: {epoch_accuracy:.2f}")
```

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

[49]:
```
train()
```

```
Current average loss: 3.8489789962768555
Current average loss: 2.032761191377545
Current average loss: 1.7417847810693048
Current average loss: 1.5897034058143134
Current average loss: 1.4797293839609236
Current average loss: 1.3818399067410452
Current average loss: 1.2951422192887736
Current average loss: 1.2201558276692062
Current average loss: 1.1540000996414046
Current average loss: 1.098895550832632
Current average loss: 1.0457295478878916
Current average loss: 1.0029691543013912
Current average loss: 0.9644977349127262
Current average loss: 0.928303958577527
Current average loss: 0.8974224679666617
Current average loss: 0.8676183697464147
Current average loss: 0.8417978709764886
Current average loss: 0.8187766099838143
Current average loss: 0.797694412404997
Current average loss: 0.7763528629455486
Current average loss: 0.7574201597087923
Current average loss: 0.7391122446379056
Current average loss: 0.7226728083449891
Current average loss: 0.7074825900863015
```

```
Current average loss: 0.693265890569947
Current average loss: 0.6804360862745376
Current average loss: 0.6685178043383078
Current average loss: 0.6565730643852778
Current average loss: 0.6453755298381361
Current average loss: 0.6354114093716002
Current average loss: 0.6257466681547381
Current average loss: 0.616416473276805
Current average loss: 0.6072201362180546
Current average loss: 0.5985777701249234
Current average loss: 0.5905661241854334
Current average loss: 0.5828460957009055
Current average loss: 0.5754788730131093
Current average loss: 0.5688016439625232
Current average loss: 0.5621508520662047
Current average loss: 0.5556425379980255
Current average loss: 0.5496060466582374
Current average loss: 0.5436328664963433
Current average loss: 0.5379204720691453
Current average loss: 0.5326803841374137
Current average loss: 0.5274397066149541
Current average loss: 0.522212878899213
Current average loss: 0.5173150913737644
Current average loss: 0.5123226226067142
Current average loss: 0.5081022285395204
Current average loss: 0.5035420597265433
Current average loss: 0.49929855615079366
Current average loss: 0.4946607351244676
Current average loss: 0.4904872649860506
Current average loss: 0.4865567096040128
Current average loss: 0.48241031610773705
Current average loss: 0.47871219792749165
Current average loss: 0.474799270999206
Current average loss: 0.4712148674931511
Current average loss: 0.46783204007186757
Current average loss: 0.46416926591290475
Current average loss: 0.46089876959386855
Current average loss: 0.45751991552215115
Current average loss: 0.45413016491485325
Current average loss: 0.45116939435632736
Current average loss: 0.4484095286902106
Current average loss: 0.4454064824293346
Current average loss: 0.44263150061694734
Current average loss: 0.4399188531290558
Current average loss: 0.43707046464990473
Current average loss: 0.43426019759704854
Current average loss: 0.43195765042684703
Current average loss: 0.42979088611454885
```

```
Current average loss: 0.42750759903067864
Current average loss: 0.425040180068435
Current average loss: 0.4227143317106656
Current average loss: 0.4203861307528095
Current average loss: 0.41812936356762903
Current average loss: 0.41597168955218095
Current average loss: 0.4139728515303039
Current average loss: 0.41184291777192394
Current average loss: 0.40985022417278144
Current average loss: 0.4076962210670244
Current average loss: 0.4056021862019091
Current average loss: 0.4035461369191483
Current average loss: 0.40149302339721626
Current average loss: 0.39968827069410434
Current average loss: 0.397847692317695
Current average loss: 0.39588281522707175
Current average loss: 0.39394636161658403
Current average loss: 0.39240297042323075
Current average loss: 0.3907377377149132
Current average loss: 0.38917159591532846
Current average loss: 0.38748129043449153
Current average loss: 0.3859656657440831
Current average loss: 0.38446368144403276
Current average loss: 0.38296150409078084
Current average loss: 0.38131166504359026
Current average loss: 0.3800030751761524
Current average loss: 0.37841073042314866
Training loss epoch: 0.37814274972031886
Average accuracy epoch: 0.89
```

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

[50]:
```
train()
```

```
Current average loss: 0.2523636519908905
Current average loss: 0.21674129005410883
Current average loss: 0.21593701657815953
Current average loss: 0.21603093879216928
Current average loss: 0.2130406890296728
Current average loss: 0.21129933088839412
Current average loss: 0.20698380991568582
Current average loss: 0.20879106441998957
Current average loss: 0.206842789298549
Current average loss: 0.206976697911466
Current average loss: 0.20505515576063932
Current average loss: 0.2058383920496426
Current average loss: 0.2052666527730142
Current average loss: 0.20465166703565224
Current average loss: 0.20455900400472826
```

```
Current average loss: 0.20363407796726554
Current average loss: 0.2036381594110734
Current average loss: 0.20311934641315474
Current average loss: 0.2021578585171554
Current average loss: 0.202376537624627
Current average loss: 0.20219343502273684
Current average loss: 0.20183473511260785
Current average loss: 0.20148962798076886
Current average loss: 0.20177626786271888
Current average loss: 0.20167620701157565
Current average loss: 0.2014901043429393
Current average loss: 0.20105446925081
Current average loss: 0.20105196273672232
Current average loss: 0.2007736087507731
Current average loss: 0.2006438331873057
Current average loss: 0.2004698565275083
Current average loss: 0.20044884106077634
Current average loss: 0.2002133777698514
Current average loss: 0.20003321813574634
Current average loss: 0.19983042763111802
Current average loss: 0.19967712221895412
Current average loss: 0.19939036167986324
Current average loss: 0.19916230296603862
Current average loss: 0.19903721653319006
Current average loss: 0.19877136584253868
Current average loss: 0.1990164024320074
Current average loss: 0.1992962956355857
Current average loss: 0.19922066211054876
Current average loss: 0.19934425965190572
Current average loss: 0.19921324087293737
Current average loss: 0.1989246223513364
Current average loss: 0.19878503537322484
Current average loss: 0.1987008948209526
Current average loss: 0.19835730555562942
Current average loss: 0.1981466545768415
Current average loss: 0.1981675983342355
Current average loss: 0.1981658254551247
Current average loss: 0.19805004216416872
Current average loss: 0.19767217069985146
Current average loss: 0.197340412471989
Current average loss: 0.1972943392335567
Current average loss: 0.1973751456608274
Current average loss: 0.19735981422706647
Current average loss: 0.19713614426760032
Current average loss: 0.19688706076881518
Current average loss: 0.19669374092603858
Current average loss: 0.1967208999661105
Current average loss: 0.19652254271088657
```

```
Current average loss: 0.19613007715019493
Current average loss: 0.19594622271738096
Current average loss: 0.19593416850734832
Current average loss: 0.19579362147060908
Current average loss: 0.1957824719574154
Current average loss: 0.195629967429996
Current average loss: 0.19544455969004143
Current average loss: 0.19540547638218447
Current average loss: 0.19520472842187986
Current average loss: 0.19503627627014022
Current average loss: 0.19491398219800357
Current average loss: 0.1947505617751582
Current average loss: 0.19443432114389622
Current average loss: 0.1943158468079871
Current average loss: 0.194110168198044
Current average loss: 0.19387565817459013
Current average loss: 0.19367235413189177
Current average loss: 0.19366963051279074
Current average loss: 0.19351790684536643
Current average loss: 0.19342767870956012
Current average loss: 0.1933471014999852
Current average loss: 0.1932228584888577
Current average loss: 0.193176248727732
Current average loss: 0.1931570711379002
Current average loss: 0.19311989230698806
Current average loss: 0.1930437765604101
Current average loss: 0.19289170722178417
Current average loss: 0.19269800254322464
Current average loss: 0.1925861085648917
Current average loss: 0.1923379896905238
Current average loss: 0.19217148089797223
Current average loss: 0.19206196460593522
Current average loss: 0.19190197184630137
Current average loss: 0.19194348430775712
Current average loss: 0.19192149372418668
Current average loss: 0.19180580335335212
Training loss epoch: 0.19176460378992083
Average accuracy epoch: 0.94
```

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.

[51]:
```
torch.save(model.state_dict(), "srl_model_fulltrain_2epoch_finetune_1e-05.pt")
```

1.7 4. Decoding

[114]:
```
# Optional step: If you stopped working after part 3, first load the trained␣
↪model

model = SrlModel().to(DEVICE)
model.load_state_dict(torch.load("srl_model_fulltrain_2epoch_finetune_1e-05.
↪pt"))
model = model.to(DEVICE)
```

```
/var/folders/63/09bwbr390mv1m8j9yh164qh80000gn/T/ipykernel_57998/550884001.py:4:
FutureWarning: You are using `torch.load` with `weights_only=False` (the current
default value), which uses the default pickle module implicitly. It is possible
to construct malicious pickle data which will execute arbitrary code during
unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for
more details). In a future release, the default value for `weights_only` will be
flipped to `True`. This limits the functions that could be executed during
unpickling. Arbitrary objects will no longer be allowed to be loaded via this
mode unless they are explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control of the
loaded file. Please open an issue on GitHub for any issues related to this
experimental feature.
  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.

[136]:
```
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.
    """
    predicted_label_ids = torch.argmax(logits.squeeze(0), axis=1)
```

```
    return [id_to_role.get(i.item(), id_to_role[-100]) for i in␣
↪predicted_label_ids]
```

[183]:
```
import numpy as np
```

[198]:
```
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.
    token_count = np.zeros(len(tokens))
    predicate_mask = torch.zeros((128, ), dtype=torch.long)
    transformed_tokens = ['[CLS]']
    for i, t in enumerate(tokens):
        if i == pred_idx:
            predicate_mask[len(transformed_tokens)] = 1
        bert_word = tokenizer.tokenize(t)
        token_count[i] += len(bert_word)
        transformed_tokens += [*bert_word]

    transformed_tokens = transformed_tokens + ['[SEP]'] + ['[PAD]'] * (127 -␣
↪len(transformed_tokens))
    token_ids = torch.tensor(tokenizer.
↪convert_tokens_to_ids(transformed_tokens), dtype=torch.long)
    attention_mask = (token_ids != 0).long()

    logits = model(input_ids=token_ids.unsqueeze(0).to(DEVICE),
                   attn_mask=attention_mask.unsqueeze(0).to(DEVICE),
                   pred_indicator=predicate_mask.unsqueeze(0).to(DEVICE))

    final_predictions = []
    predicted_labels = decode_output(logits)[1:]
    idx = 0
    for count in token_count:
        final_predictions.append(predicted_labels[idx])
        idx += int(count)

    return final_predictions
```

[199]:
```
# Now you should be able to run
label_test = label_sentence(tokens, 13) # Predicate is "found"
for t, l in zip(tokens, label_test):
    print(t, l)
```

```
A O
U. O
N. O
team O
```

```
spent O
an O
hour O
inside O
the B-ARGM-LOC
hospital I-ARGM-LOC
, O
where B-ARGM-LOC
it B-ARG0
found B-V
evident B-ARG1
signs I-ARG1
of I-ARG1
shelling I-ARG1
and I-ARG1
gunfire I-ARG1
. O
```

The expected output is something 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'),
```

1.7.1 5. Evaluation 1: Token-Based Accuracy

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

[215]:
```
dev_data = SrlData("propbank_dev.tsv") # Takes a while because we preprocess␣
 ↪all data offline
```

[216]:
```
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(DEVICE)
```

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):
        # 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)
        pred_mask = batch['pred'].to(DEVICE, dtype = torch.long)
        targets = batch['targets'].to(DEVICE, dtype = torch.long)

        # Run the forward pass of the model
        # print(ids.shape, mask.shape, pred_mask.shape)
        logits = model(input_ids=ids, attn_mask=mask, pred_indicator=pred_mask).
↪squeeze(0)
        predictions = torch.argmax(logits, dim=1)

        bool_mask = mask.squeeze(0).clone().bool()
        valid_indices = torch.where(mask.squeeze(0) == 1)[0]
        bool_mask[valid_indices[0]] = False # filtered out [CLS]
        bool_mask[valid_indices[-1]] = False # filtered out [SEP]
        filtered_predictions = predictions[bool_mask]
        filtered_target = targets.squeeze(0)[bool_mask]
```

```
        total_predictions += len(filtered_predictions)
        total_correct += torch.sum(filtered_predictions == filtered_target)

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

[250]:
```
evaluate_token_accuracy(model, loader)
```

```
Accuracy: 0.9554257392883301
```

1.7.2 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

Token	[CLS]	The	judge	scheduled	to	preside	over	his	trial	was	removed	from	the	case	today	.
Index	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"}

[351]:
```
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
```

[364]:
```
def evaluate_spans(model, loader):

    total_tp = 0
    total_fp = 0
    total_fn = 0

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

        # Run the forward pass of the model
        logits = model(input_ids=ids, attn_mask=mask, pred_indicator=pred_mask)
        valid_length = torch.sum(mask) - 2
```

```
        predicted_labels = decode_output(logits)[1:valid_length]
        target_labels = [id_to_role[i.item()] for i in targets[0][1:
↪valid_length]]

        predicted_span = extract_spans(predicted_labels)
        target_span = extract_spans(target_labels)

        tp = 0
        for span, label in predicted_span.items():
            result = target_span.get(span)
            if result == label: # correct prediction
                tp += 1
            elif result == None: # span not in target
                total_fp += 1

        total_tp += tp
        total_fn += len(target_span) - tp

    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}")
```

[365]:
```
evaluate_spans(model, loader)
```

```
Overall P: 0.8828027613412229  Overall R: 0.8707174775356638  Overall F1:
0.8767184734735373
```

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

1.7.3 OPTIONAL:

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