Natural Language Processing CA#3

Student Name:
Pouya Haji Mohammadi Gohari

SID:810102113

Date of deadline
Saturday 11th May, 2024

Dept. of Computer Engineering

University of Tehran

Contents

Dataset At first we will investigate through train, valid and test sets. Unfortunately the test set has no ”srl_frames”. Using ”json” library in Python, we load each dataset.The implemented code below shows the procedure:

```
    with open('data/train.json', 'r') as my_file:
        train_set = json.load(my_file)
```

At next step we will get a sample from train set in order to be familiar with our dataset.

```
    keys_list = [key for key in train_set.keys()]
    for key in keys_list:
        print(f'The second {key} in traingin set is: {train_set[key][1]}')
```

The second sample for training set is as follow:

- Text: [’The’, ’progress’, ’of’, ’this’, ’coordinated’, ’offensive’, ’was’, ’already’, ’very’, ’entrenched’, ’by’, ’then’, ’.’]
- SRL[1] Frames: [’O’, ’O’, ’O’, ’O’, ’O’, ’B-ARG1’, ’O’, ’O’, ’O’, ’O’, ’O’, ’O’, ’O’]
- Verb Index: 4
- Word Indices: [13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]

Our datasets is a dictionary consist of a 2D list, where each list represent as a list of words. Another double list ”srl_frames” will provide us annotation of each word in corresponding sentence. (e.g., word ”offensive” is ’B-ARG1’ which is Beginning of Argument 1).
As example above shows, verb of sentence is indexed by 4 which is correspond to ’coordinated’ in that sentence.(The word indices is just showing the word index in the whole sentences.)

1 Part 1

As an illustrative example in previous section, dataset is now can be accessed and managed.
We will use a dictionary structure to map the SRL frames to the numerical values.The following code can be used to accomplish this process:(Note that this function will pad first and then mapped them to the numerical values)

```
    mapper_dict = {
    'O': 0,
    'B-ARG0': 1,
    'I-ARG0': 2,
    'B-ARG1': 3,
    'I-ARG1': 4,
    'B-ARG2': 5,
    'I-ARG2': 6,
    'B-ARGM-LOC': 7,
    'I-ARGM': 8,
    'B-ARGM-TMP':9,
    'I-ARGM-TMP':10
```

[1]Semantic Rule Labels

```
    }
    def mapping_dictionary(mapper:dict, srl_labels:list[list[str]]) -> list[list[
        ↪ int]]:
        """
            First padding to fix the output length
            Then simply maps the srl_frames to the tagset
        """
        max_length = max((len(labels) for labels in srl_labels), default=0)
        labels = [[0 for _ in tags] for tags in srl_labels]
        for index, tag_list in enumerate(srl_labels):
            if len(tag_list) < max_length:
                for j in range(max_length-len(tag_list)):
                    labels[index].append(0)
            for i, tag in enumerate(tag_list):
                labels[index][i] = mapper[tag]
        return labels
```

Attention: We write an document for each function in order to have more readability and simplify the procedure of using them.
In order to pad the sentences, we will calculate the maximum length and pad all sentences to fix the length of each sentences as follow:

```
def Padding(given_set:dict, max_length:int, Pad_token='<PAD>') -> dict:
        """
            Adding pad token to the list in order to fix the sizes the sentences
        """
        for index, token_list in enumerate(given_set['text']):
            if len(token_list) < max_length:
                for j in range(max_length - len(token_list)):
                    given_set['text'][index].append(Pad_token)
        return given_set
```

Now it is time to create a class for Vocab:
Adding constructor for vocab class:

```
    class Vocab(object):
        def __init__(self, word2id:dict) -> None:
            """
                If word2id is None then it would be initialized by a dictionary
                    ↪ otherwise will be initialized by word2id's value
            """
            self.word2id = {'<PAD>': 0,'<Start>': 1,'<End>': 2,'<Unknown>': 3} if
                ↪ word2id is None else word2id
            self.id2word = {value:key for key, value in self.word2id.items()}
```

If word2id is None then vocab will initialize a dictionary with following special tokens:

- '<PAD>': The PAD token will be indexed as 0.

- '<Start>', '<End>': These tokens will be used in the last question of this CA. They stand as Start of sentence and End of sentence.
- '<Unknown>': If any word that is not in vocab then it would be treated as unknown word.

Getting item from Vocab:

```
    def __getitem__(self, word) -> int:
        """
            return the word's key in word2id. If the word does not exist return id
               ↪  of Uknown word.
        """
        if word in self.word2id:
            return self.word2id[word]
        return self.word2id['<Unknown>']
```

If word is not in vocab then it would return word's id otherwise it would return '<Unknown>'.
Len method:(magic method)

```
    def __len__(self) -> int:
        """
            Return the vocab size
        """
        return len(self.word2id)
```

Add method:

```
    def add(self, word : str) -> int:
        """
            Add a word if it is not already in vocab and return the index of added
               ↪  word
        """
        if word not in self.word2id:
            length = len(self.word2id)
            self.word2id[word] = length
            self.id2word[length] = word
            return length
```

Word2indices method:

```
    def word2indices(self, sents:list[list[str]]) -> list[list[int]]:
        """
            Get list of sentences and returns the corresponding indices in vocab
               ↪ dictionary
        """
        indices = [[0 for _ in sentence] for sentence in sents]
        for i,tokens in enumerate(sents):
            for j,token in enumerate(tokens):
                indices[i][j] = self[token]
        return indices
```

Transform into tensor:

```
    def to_input_tensor(self, sents:list[list[str]]) -> list[list[tensor]]:
        """
            First padding the sentences then\\
            convert sentences to indexes and turn them into tensors
        """
        temp_sents = copy.deepcopy(sents)
        max_length = max((len(sentence) for sentence in sents), default=0)
        for i,sentence in enumerate(temp_sents):
            if(len(sentence) < max_length):
                for _ in range(max_length - len(sentence)):
                    temp_sents[i].append('<PAD>')
        return tensor(self.word2indices(temp_sents), dtype=torch.long)
```

From corpus function:

```
    def from_corpus(corpus: list[list[str]], size:int, remove_frac:float,
       ↪ freq_cutoff:int) -> Vocab:
        """
            Getting an instance of Vocab class and add words by add method
        """
        vocab_model = Vocab(None)
        word_freq = {}
        for token_list in corpus:
            for token in token_list:
                if token in word_freq:
                    word_freq[token] += 1
                else:
                    word_freq[token] = 1
        sorted_word_freq = sorted(word_freq.items(), key=lambda item: item[1])
        num_items = len(sorted_word_freq)
        num_to_remove = int(num_items * remove_frac)
        remaingin_items = sorted_word_freq[num_to_remove:]
        remaingin_items = dict(remaingin_items)
        for word,count in remaingin_items.items():
            if len(vocab_model) >= size:
                break
            if count >= freq_cutoff:
                vocab_model.add(word)
        return vocab_model
```

This function first collect all words and their frequencies and then remove a fraction of this words with low frequencies and remaining words will add to the vocab if their frequencies is higher than 'frequency_cutoff'. First we are going to get an instance of this Object and add training words for future purposes.

2 Part 2: LSTM Encoder Model

2.1 Implementing

With the help of 'Pytorch' library, we will create a class of LSTM_Based and inherenct from 'nn.Module' to create our LTSM model as follow:

```
class LSTM_Based(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim) -> None:
        super(LSTM_Based, self).__init__()
        self.word_embeddings = nn.Embedding(vocab_size, embedding_dim)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, batch_first=True,
            ↪ num_layers=1)
        self.fc = nn.Linear(hidden_dim*2, output_dim)
        self.output_dim = output_dim

    def forward(self, sentences, predicate_indices):
        embedded = self.word_embeddings(sentences)
        lstm_out, (hidden, cell) = self.lstm(embedded)
        predicate_hidden = lstm_out[torch.arange(len(predicate_indices)),
            ↪ predicate_indices]
        predicate_hidden_expanded = predicate_hidden.unsqueeze(1).expand(-1,
            ↪ lstm_out.size(1), -1)
        enhanced_hidden = torch.cat((lstm_out, predicate_hidden_expanded), dim=2)
        output = self.fc(enhanced_hidden)
        return output
```

After specify the hyper parameters like 'vocab size' and 'embedding dim' and 'output dim' and get an instance of this class. Forward method will simply get sentences and verb indices correspond to that sentence. The shape of sentences would be $[batch\_size, seq\_len]$ and for verb indices would be $[batch\_size]$. After passing the sentences and verb indices to the embedding for each word.After passing sentences through embedding layers the lstm out shape would be $[batch\_size, seq\_len, embedding\_dim]$.
We will get the verb of lstm out and concatenate with all the outputs of lstm. Finally the lstm's outputs will be passed through linear layer to get the label for each word.Thus the shape of the linear output would be $[batch\_size, seq\_len, 11]$. Before jumping in training phase, we must create a dataset that data loader can interpret with.
Custom Dataset:

```
    class Custom_Dataset(Dataset):
        def __init__(self, input_set:dict) -> None:
            """
                Specify the dataset as a dictionary
            """
            self.txt = input_set['text']
            self.verb = input_set['verb_index']
            self.labels = input_set['srl_frames']

        def __len__(self):
            return len(self.txt)
```

```
        def __getitem__(self, index):
            return self.txt[index], self.verb[index], self.labels[index]
```

We will inherent from Dataset class in 'Pytorch' library. For Data loader we will use following implemented code:

```
train_dataloader = DataLoader(
    train_dataset,
    batch_size=64,
    shuffle=True,
    collate_fn=lambda batch : collate_batch(batch, vocab=vocab, mapping_dict=
       ↪ mapper_dict)
)
```

The collate function for each batch getting from dataset would be:

```
def collate_batch(batch, vocab:Vocab, mapping_dict:dict):
    text = []
    verb_index = []
    labels = []
    for b in batch:
        text.append(b[0])
        verb_index.extend([b[1]])
        labels.append(b[2])
    txt_tensor = vocab.to_input_tensor(text)
    verb_index_tensor = tensor(verb_index, dtype=torch.long)
    labels_tensor = tensor(mapping_dictionary(mapping_dict, labels), dtype=torch.
       ↪ long)
    return txt_tensor, verb_index_tensor, labels_tensor
```

We will get text, verb index, labels from each iteration in batch. From the method '*to_input_tensor*' in Vocab class we will pad them and turn into tensor. Also labels would be padded alongside with text by mapping dictionary implemented in previous section. (Verb indices will be tensor)
Training phase:

```
def train(model, optimizer, loss_fn, train_loader, val_loader, epochs, device="
   ↪ cuda"):
    accuracy_per_epoch = []
    loss_per_epoch = []
    val_loss_per_epoch = []
    val_accuracy_per_epoch = []
    reals, all_preds = [], []
    model.to(device)
    for epoch in range(epochs):
        training_loss = 0
        whole_predictions = []
        whole_labals = []
        model.train()
        for batch in tqdm(train_loader):
            optimizer.zero_grad()
```

```
            sentences = batch[0].to(device)
            verb_indices = batch[1].to(device)
            srl_labels = batch[2].to(device)
            preds = model(sentences, verb_indices)
            loss = loss_fn(preds.view(-1, model.output_dim), srl_labels.view(-1))
            loss.backward()
            training_loss += loss.item()
            optimizer.step()
            whole_predictions.extend(preds.view(-1, model.output_dim).argmax(axis
               ↪ =1).cpu().numpy().tolist())
            whole_labals.extend(srl_labels.view(-1).cpu().numpy().tolist())
        training_loss = training_loss/len(train_loader)
        train_accuracy = accuracy_score(whole_predictions, whole_labals)
        loss_per_epoch.append(training_loss)
        accuracy_per_epoch.append(train_accuracy)
        print(f'Epoch: {epoch}, training_loss: {training_loss:.2f}, train accuracy
           ↪  {train_accuracy:.2f}')
        val_loss, val_accuracy, reals, all_preds = validation(model, val_loader,
           ↪ loss_fn, device)
        val_loss_per_epoch.append(val_loss)
        val_accuracy_per_epoch.append(val_accuracy)
        print(f'validation loss: {val_loss:.2f}, validation accuracy: {
           ↪ val_accuracy:.2f}')
    print(classification_report(reals, all_preds))
    return loss_per_epoch, accuracy_per_epoch, val_loss_per_epoch,
       ↪ val_accuracy_per_epoch
```

For each epoch, we will append the real outputs to the list of real list.Predicted via model will be appended to predictions list. After seeing all the batches in training data loader, the accuracy will be calculated with accuracy score in 'sklearn' library.
At the end of each epoch, model will validate on the validation data loader. Finally at the end, the classification report will be printed. Validation function will be:

```
def validation(model, val_loader, loss_fn, device="cuda"):
    val_loss = 0
    model.eval()
    reals = []
    all_preds = []
    for batch in val_loader:
        sentence = batch[0].to(device)
        verb_indices = batch[1].to(device)
        labels = batch[2].to(device)
        with torch.no_grad():
            preds = model(sentence, verb_indices)
        loss = loss_fn(preds.view(-1, model.output_dim), labels.view(-1))
        val_loss += loss.item()
        all_preds.extend(preds.view(-1, model.output_dim).argmax(axis=1).cpu().
           ↪ numpy().tolist())
```

```
            reals.extend(labels.view(-1).cpu().numpy().tolist())
        val_loss = val_loss / len(val_loader)
        accuracy = accuracy_score(reals, all_preds)
        return val_loss, accuracy, reals, all_preds
```

The model's hyper parameters are set as defined in the description but the learning rate is set to $1e-2$.
The loss per epoch for training and validation set alongside with their accuracy is illustrated in figure 1:

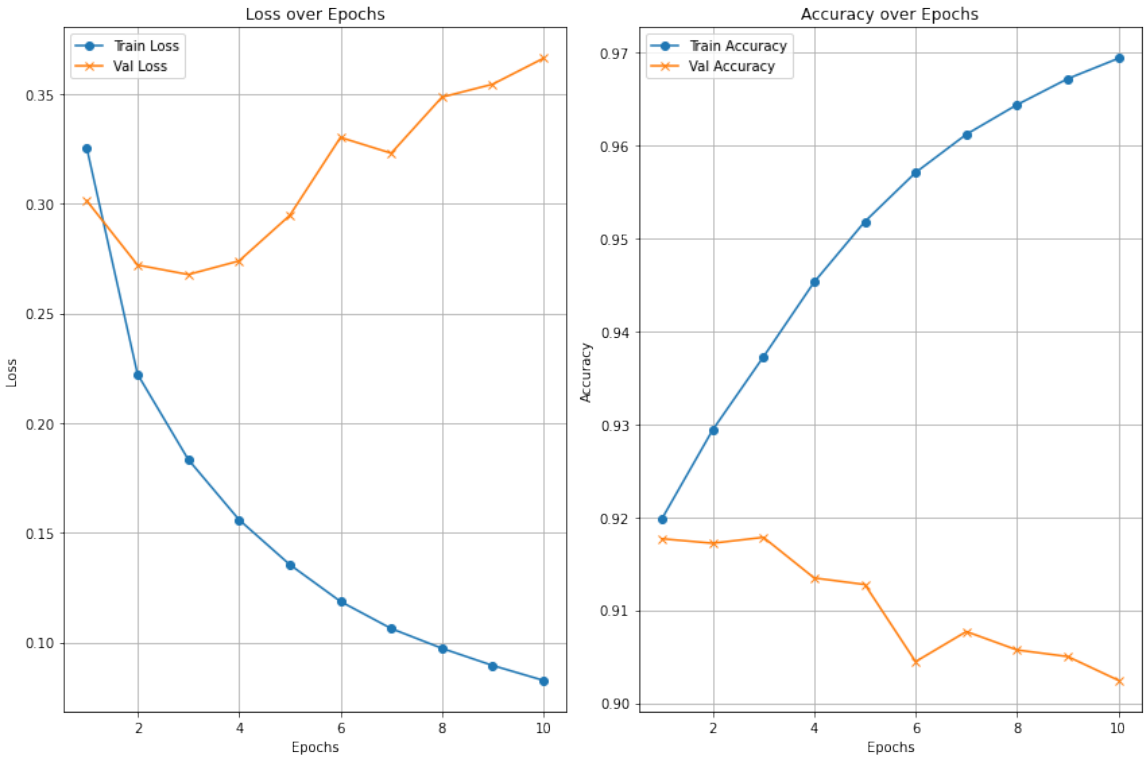

Figure 1: Accuracy and loss for validation and

Interpretation of the loss:
The training loss decreases over the epochs, which indicates that the model is effectively learning from the training data.On the other hand, validation loss shows a different pattern. It initially decreases, indicating that the model is generalizing well from the training data to unseen validation data. But from around epoch 5, the validation loss begins to increase, suggesting the beginning of overfitting.
Accuracy: There is a clear increase in training accuracy, which improves from 0.9 to nearly 0.97 over the epochs.The validation accuracy initially increases along with training accuracy, which is a good indicator of the model's ability to generalize.However, similar to the validation loss pattern, the validation accuracy will drop around epoch 5.
Wrapping this up, Both the training and validation metrics improve, which is typical and expected during the early stages of training. This phase shows that the model is learning useful patterns from the training data and can apply these patterns to validate data. But in later epochs, indicating the overfitting phase.

2.2 F1-score

The f1-score for validation is shown in table 1: Lower macro average for f1-score is indicating that classes are

f1-score	micro	macro
	0.90	0.41

Table 1: F1-score for validation set

imbalance. Classification report shows that the model is likely learning to predict 'o' instead of learning other this will be caused by imbalance classes since the 'o' labels are 34129 out of 37120 labels.That might be due to padding!
In contrast the micro average of 0.9 is average f1-score for all classes.

3 GRU Encoder Model

3.1 Implementation

The procedure for this model is same as previous but instead of using LSTM layer, we will use GRU as follow:(Note that the training and validating is same as LSTM, also dataset and dataloader is same as before).

```
class GRU_Baesd(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim):
        super(GRU_Baesd, self).__init__()
        self.word_embedding = nn.Embedding(vocab_size, embedding_dim)
        self.gru = nn.GRU(embedding_dim, hidden_dim, num_layers=1, batch_first=
            ↪ True)
        self.fc = nn.Linear(hidden_dim*2, output_dim)
        self.output_dim = output_dim

    def forward(self, sentences, predicate_indices):
        embeded = self.word_embedding(sentences)
        gru_out, _ = self.gru(embeded)
        predicate_hidden = gru_out[torch.arange(len(predicate_indices)),
            ↪ predicate_indices]
        predicate_hidden_expanded = predicate_hidden.unsqueeze(1).expand(-1,
            ↪ gru_out.size(1), -1)
        enhanced_hidden = torch.cat((gru_out, predicate_hidden_expanded), dim=2)
        output = self.fc(enhanced_hidden)
        return output
```

As illustrated in figure 2, the loss and accuracy for epoch is represented in two category training and validation dataset.

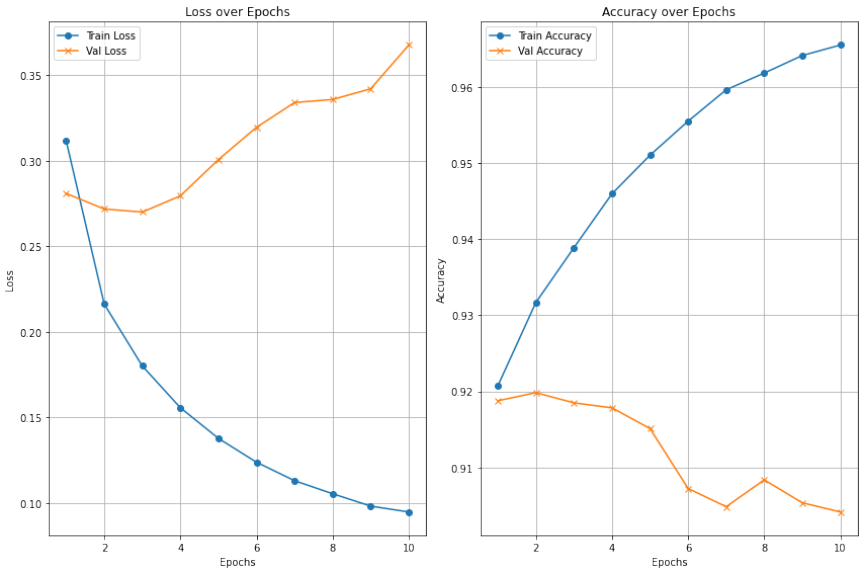

Figure 2: Loss and accuracy for training and validating

Table 2 shows the f1-score metrics for GRU Encoder model. If we compare these two models based on the

f1-score	micro	macro
	0.90	0.40

Table 2: Micro and Macro average of F1-score with Gru

micro and macro average of f1-score metrics interpretation of the results are:
Both models show the same macro F1-score of 0.9. This high score suggests that both models are generally effective across the different label classes when averaged without considering the class imbalance.(Note that macro average treats all classes equally, regardless of how many instances of each class appear in dataset.)
In contrast micro average will treat each class with respect to the classes frequencies. There is slightly difference between their micro F1-score.The slight decrease from LSTM to GRU suggests that the LSTM might be slightly better at correctly identifying true labels across all instances.
Model comparison and interpretation: Both model perform similarly well in terms of their ability ti generalize across different SRL labels, as indicated in macro f1-scores.The better micro F1-score for the LSTM model suggests it might be slightly more effective in dealing with class imbalance within dataset.This could be due to the LSTM's more complex gating architecture, which may capture dependencies in the data that the simpler GRU structure slightly misses.
Attention: Personally if i want to chose one, i will prefer the GRU model with respect to computationally efficient that LSTMs due to having fewer parameters.

3.2 Answering the questions

3.2.1 Advantages of LSTM over RNN

The main advantage of LSTM over standard RNN stem from their ability to handle long-term dependencies and avoid porblems like vanishing and exploding gradients.

RNN typically struggle to maintain information for long sequences this is due to vanishing gradient problem, where gradients of the loss function become so small that early layers in the sequence stop learning effectively. However LSTM consist gates that regulate the flow of information. These gates (input, output, and forget) can selectively remember of forget information, which allows LSTMs to carry information across many timesteps. This makes them exceptionally good at learning dependencies between events that occur with long gaps in the input data sequences.

3.2.2 Differences between LSTM and GRU

Both are types of RNN architectures used for processing sequential models. Although both are designed to address the problem of vanishing gradients in basic RNNs, they have different structures and operation mechanisms. Aspects of differences are as follow:

1. Architecture: LSTMs have a more complex architecture.They have three gates:

   - Input gate decides which values to update.
   - Output gate controls what the next hidden state should be.
   - Forget gate determines what information should be discarded from the cell state.

   Moreover, LSTMs maintain two state vectors;the cell state and the hidden state, which help it to capture long-term dependecies.

2. GRUs simplify the structure seen in LSTMs by using only two gates:

   - Update gate acts similarly to the LSTM's input and forget gates combined;it determines both how much the past information(from previous steps) needs to be passed along to the future.
   - Reset gate decides how much of the past information to forget.

   GRUs combine the cell and hidden state into a single vector, making them less complex often faster ti train than LSTMs.

3. Parameters: LSTMs generally have more parameters due to more complex gate mechanisms and state maintenance.However, GRU have fewer parameters due to simplified gate mechanisms and unified state, where can lead to faster training times and require less data to generalize well.

4. Dataset: GRUs need less data to be trained since they have fewer parameters.In contrast LSTMs needs more data to be trained.

3.2.3 Why concatenate the verb with words?

: Intuitively verb has very crucial role in Semantic Role labeling as it is the pivot around which other words in the sentence define their roles.Moreover, each word in the sentence relates to the verb in a way, such as subject(who is performing action), the object(who is receiving action) or other elements like time and location.
Another reason is by concatenating the verb state and output state allows each words representation to be directly informed by the verb's contextual information.This process will ensure verb will influence on each words, leading to more accurate role assignments.

3.2.4 Vanishing Gradients

:
They are several techniques can be obtained to address this issue.Some of these techniques are:

1. Gradient Clipping: One of the simplest yet effective ways to address the both exploding and vanishing gradients is gradient clipping. This method simply involves clipping the gradients during back propagation to ensure the do not exceed a defined threshold.

2. Using Proper Activation Functions: Choosing good activation function can significantly impact the behaviour of gradients. Relu or its variants can be helpful for preventing vanishing gradients instead of using sigmoid and tanh.

3. Using dropout can helpful in RNNs.

4. Instead of padding the sequence we can truncate them in order time steps not getting more.

4 Encoder-Decoder Model

4.1 Preprocessing Data for QA

For this purpose we implement a QA class to make questions and answers in easy way:

```
class QuestionAnsewring(object):
    def __init__(self, dataset:dict, map:dict) -> None:
        """
            Pass the Dataset and the vocab model.
        """
        self.txt = copy.deepcopy(dataset['text'])
        self.labels = copy.deepcopy(dataset['srl_frames'])
        self.verb_indexes = copy.deepcopy(dataset['verb_index'])
        self.srl_mapping = mapper_dict
        self.questions = []
        self.answering = []
        self.map = map
        self.label_maps = ['ARG0', 'ARG1', 'ARG2', 'ARGM-LOC', 'ARGM-TMP']

    def mapping_labels(self) -> None:
        """
            Map the labels to simplicity
        """
        for i,label_list in enumerate(self.labels):
            for j,label in enumerate(label_list):
                self.labels[i][j] = self.map[label]

    def make_quesiton_answering(self, index) -> None:
        sentence = self.txt[index]
        verb_index = self.verb_indexes[index]
        labels = self.labels[index]
```

```
        for tag in self.label_maps:
            question = [sentence[verb_index], '[SEPT]']
            question.extend(sentence)
            answering = []
            question.extend([tag])
            if tag in labels:
                answering.extend(['<Start>'])
                for i, label in enumerate(labels):
                    if label == tag:
                        answering.extend([sentence[i]])
                answering.extend(['End'])
            else:
                answering.extend(['<Start>', '<End>'])
            self.questions.append(question)
            self.answering.append(answering)

    def get_dataset(self):
        """
            Creating 5 question for each tag with text and verb indexes
        """
        for index in tqdm(range(len(self.txt))):
            self.make_quesiton_answering(index)
        return self.questions, self.answering
```

Mapping labels method is a simply get the true SRL labels and mapped them with following dictionary:

```
mapping = {
    'O': 'O',
    'B-ARG0': 'ARG0',
    'I-ARG0': 'ARG0',
    'B-ARG1': 'ARG1',
    'I-ARG1': 'ARG1',
    'B-ARG2': 'ARG2',
    'I-ARG2': 'ARG2',
    'B-ARGM-LOC': 'ARGM-LOC',
    'I-ARGM': 'ARGM-LOC',
    'B-ARGM-TMP':'ARGM-TMP',
    'I-ARGM-TMP': 'ARGM-TMP'
}
```

This method was implemented so we can correctly asnwer each question with begining and inside of arguments. Next method called ’make question answering’ will get an index of text and for each sentence will make 5 question and 5 answers as description said.Finally with ’get dataset’ method we will get all questions and answers. Following example will illustrate that QA class will create questions and answers:
Question = [’inscribed’, ’[SEPT]’, ’A’, ’primary’, ’stele’, ’,’, ’three’, ’secondary’, ’steles’, ’,’, ’and’, ’two’, ’inscribed’, ’steles’, ’.’, ’ARG1’]
Answer: [’<Start>’, ’steles’, ’<End>’]
Now we can treat all questions and answers as list of sentences.Thus we can use vocab class implemented in first

section.

```
    all_questions_answer = copy.deepcopy(questions)
    for answer in answerings:
        all_questions_answer.append(answer)
    print(len(questions), len(answerings))
    vocab = from_corpus(all_questions_answer, size=20000, remove_frac=0.3,
       ↪ freq_cutoff=1)
```

Len of vocab will be: 13365 word.Now it is time to get the embedding of each word in vocab if that vocab is in the Glove vectors otherwise we will initialize it with rand. The code below simply doing this:(First loading a 300 dimensional vector as embedding and then check each word in vocab is in the embeddings)

```
    glove_embdedding = {}
    with open('Glove\glove.6B.300d.txt', 'r') as my_file:
        for line in my_file:
            values = line.split()
            glove_embdedding[values[0]] = np.asarray(values[1:], dtype=float)
    embds = tensor([[0 for _ in range(300)] for __ in range(len(vocab))], dtype=
       ↪ torch.float)
    for key,value in vocab.word2id.items():
        if key in glove_embdedding:
            embds[value] = torch.from_numpy(glove_embdedding[key]).long()
        else:
            embds[value] = torch.rand(embds.shape[1]).reshape((embds.shape[1]))
    embds[vocab['<PAD>']] = torch.zeros((1,embds.shape[1]))
    embds[vocab['<Unknown>']] = torch.mean(embds, dim=0, keepdim=True, dtype=float
       ↪ ).squeeze(0)
    embds.dtype
```

Attention:Embedding for PAD will be assign to zero.Also for Unknown words we will consider the mean of all embedding vectors.

4.2 Implementing Model

We will create 4 classes and inherent each from nn.Module.
Encoder with bidirectional LSTM:

```
class Encoder(nn.Module):
    def __init__(self, embedding_dim, encoder_hidden_dim, pretrained_embeddings):
        super(Encoder, self).__init__()
        self.embedding = nn.Embedding.from_pretrained(pretrained_embeddings,
           ↪ freeze=False)
        self.lstm = nn.LSTM(embedding_dim, encoder_hidden_dim, num_layers=1,
           ↪ batch_first=True, bidirectional=True)
        self.fc1 = nn.Linear(encoder_hidden_dim*2, encoder_hidden_dim)
        self.fc2 = nn.Linear(encoder_hidden_dim*2, encoder_hidden_dim)

    def forward(self, src):
```

```
        embedded = self.embedding(src)
        encoder_outputs, (hidden, cell) = self.lstm(embedded)
        hidden = torch.tanh(self.fc1(torch.cat((hidden[0, :, :],hidden[1, :, :]),
            ↪ dim=1)))
        cell = torch.tanh(self.fc2(torch.cat((cell[0,:,:], cell[1, :, :]), dim=1))
            ↪ )
        return encoder_outputs, hidden.unsqueeze(0), cell.unsqueeze(0)
```

Explain each layers:

- Embedding Layer: we will use the glove embedding vectors as initialize values for embedding vectors.Moreover, freezing argument will be set to 'False' in order that model can update the embedding vectors parameters.
- LSTM: The encoder will get an embedding vector and will return concatenated two vectors obtained by bidirectional lstm.
- Two Linear Layer: For bidirectional LSTMs the hidden and cell shape will be [2, batch, encoder hidden dimension] so we need these layers to transform them into [1, batch, encoder hidden dimension].
- Note that the encoder outputs from LSTM layers would be [batch, sequence length, 2*encoder hidden dimension]

Attention Class:

```
class Attention(nn.Module):
    def __init__(self, encoder_hidden_dim, decoder_hidden_dim):
        super(Attention, self).__init__()
        self.attn = nn.Linear(encoder_hidden_dim * 2 + decoder_hidden_dim,
            ↪ decoder_hidden_dim)
        self.v = nn.Parameter(torch.rand(decoder_hidden_dim))

    def forward(self, decoder_hidden, encoder_outputs):
        src_len = encoder_outputs.shape[1]
        decoder_hidden = decoder_hidden.unsqueeze(1).repeat(1, src_len, 1)

        energy = torch.tanh(self.attn(torch.cat((decoder_hidden, encoder_outputs),
            ↪  dim=2)))
        energy = energy.permute(0, 2, 1)

        attn_scores = torch.matmul(self.v, energy)
        return torch.softmax(attn_scores, dim=1)
```

Explain each layers:

- Linear layers: From previous decoder hidden we will have [batch, decoder hidden dimension] and from encoder outputs we will have [batch, sequence length, 2*encoder hidden dimension]. We will concatenate them and passed through this linear layer so we can transform the [batch, sequence length , 3*decoder hidden dimension] to [batch, sequence length, encoder hidden dimension]. (Note that we will unsqueeze dimension 1 for decoder hidden vector then repeat them due second dimension then concatenate it with encoder outputs)

- Parameters: These parameters will be matmul with energy(explained above) to derive which output of energy we must attend to.Further more, the output of the attention will be [batch, encoder hidden dimension]

Decoder Class:

```
class Decoder(nn.Module):
    def __init__(self, embedding_dim, encoder_hidden_dim, decoder_hidden_dim,
        ↪ output_dim, pretrained_embedding) -> None:
        super(Decoder, self).__init__()
        self.embedding = nn.Embedding.from_pretrained(pretrained_embedding, freeze
            ↪ =False)
        self.attention = Attention(encoder_hidden_dim, decoder_hidden_dim)
        self.lstm = nn.LSTM(encoder_hidden_dim * 2 + embedding_dim,
            ↪ decoder_hidden_dim, batch_first=True)
        self.fc_out = nn.Linear(decoder_hidden_dim, output_dim)

    def forward(self, input, hidden, cell, encoder_outputs):
        input = input.unsqueeze(1)
        embedded = self.embedding(input)

        attn_weights = self.attention(hidden[-1], encoder_outputs)
        attn_applied = torch.matmul(attn_weights.unsqueeze(1), encoder_outputs)
        lstm_inptut = torch.cat((embedded, attn_applied), dim=2)
        lstm_output, (hidden, cell) = self.lstm(lstm_inptut, (hidden, cell))
        output = self.fc_out(lstm_output)
        return output, hidden, cell, attn_weights
```

Explain:

- After getting each word of previous decoder it would be concatenate with attn weights to feed through lstm and finally will passed through the linear function to predict the word in output.

Sequence 2 Sequence Class:

```
class Seq2Seq(nn.Module):
    def __init__(self, encoder, decoder, device) -> None:
        super(Seq2Seq, self).__init__()
        self.encoder = encoder
        self.decoder = decoder
        self.device = device

    def forward(self, src, trg, teacher_forcing_ratio=0.5):
        batch_size = src.shape[0]
        trg_len = trg.shape[1]
        trg_vocab_size = self.decoder.fc_out.out_features
        outputs = torch.zeros(batch_size, trg_len, trg_vocab_size).to(self.device)
        encoder_ouptus, hidden, cell = self.encoder(src)
        input = trg[:, 0]
        for t in range(1, trg_len):
```

```
            output, hidden, cell, _ = self.decoder(input, hidden, cell,
               ↪ encoder_ouptus)
            outputs[:, t, :] = output.squeeze(1)
            teacher_force = np.random.rand() < teacher_forcing_ratio
            input = trg[:, t] if teacher_force else output.squeeze(1).argmax(1)

        return outputs
```

Explain:

- First questions will be fed to encoder to get the encoder outputs and hidden and cell.
- After that first word of answer will be treated as input of decoder.
- Next step is feed decoder by the encoder outputs and hidden and cell from previous decoder or encode!(For the first word hidden and cell would be hidden and cell of encoder and for next iterations would be previous decoder's hidden and cell)
- Finally we will teacher force decoder or just simply get the output's decoder.

Before jumping through trining and validating we should create Dataset and Data loader for each sets.

```
class QAdataset(Dataset):
    def __init__(self, questions:list[list[str]], answers:list[list[str]]):
        """
            Construct with questions and aswers
        """
        self.quests = questions
        self.answs = answers
        print(len(questions))
        print(len(answers))

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

    def __getitem__(self, index):
        return self.quests[index], self.answs[index]
```

Collate function and creating dataset and data loader for train and validation sets:

```
        def collate_batch(batch, vocab):
    input_ids = []
    outptut_ids = []
    for b in batch:
        input_ids.append(copy.deepcopy(b[0]))
        outptut_ids.append(copy.deepcopy(b[1]))
    return vocab.to_input_tensor(input_ids), vocab.to_input_tensor(outptut_ids)

    train_qa_dataset = QAdataset(questions, answerings)
    train_dataloader = DataLoader(train_qa_dataset, batch_size=16, shuffle=True,
       ↪ drop_last=True, collate_fn=lambda batch: collate_batch(batch, vocab))
```

```
    valid_qa_dataset = QAdataset(valid_qustions, valid_answering)
    valid_dataloader = DataLoader(valid_qa_dataset, batch_size=16, shuffle=True,
       ↪ drop_last=True, collate_fn=lambda batch: collate_batch(batch, vocab))
```

After creating model we should train and validate(Note that the training and validating is just like workshop).After struggling with hyper parameters we choose to set the hyper parameters as in table 3:

batch	encoder dimension	decoder dimension	embedding dimension	epochs
128	128	128	300	20

Table 3: Hyper parameters

The loss and accuracy for each epoch for training and validating has been illustrated in figure 3:

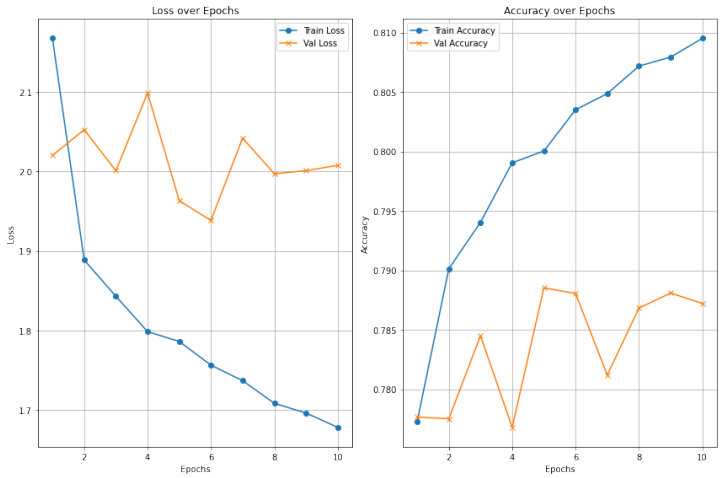

Figure 3: Loss and Accuracy per epoch for training and validation sets

Interpretation of the figure 3: While the model learns the training data effectively, its performance on the validation set raises concerns about its ability to generalize.(That can due to imbalance data especially with pad tokens)

4.3 Validation and F1-score

The F1-score metric is illustrated in table 4:

F1-score	Micro	Macro
	0.00	0.72

Table 4: Macro and Micro f1-score fo sequence 2 sequence

4.4 Answer the Questions

4.4.1 Limitation of SRL to QA

Transforming SRL labels into QA systems using encoder-decoder models has several complexities and limitations. Reason one is for loss of structural information. For instance SLR represent as a structure that understand of sentence semantics, also categorizing different parts of the sentence according to their semantic roles relative to the main verb.When transforming this structural understanding into QA format, there is a risk of losing some of the semantic richness that SRL annotations provide.
This can be challenging to transform SRL labels to QA, since QA systems typically expect input in the form of a natural language question which may not directly map to the structured format used in SRL. Second reason might be effected by generating relevant and meaningful questions for SRL labels involves understanding the context and significant of each role within the broader narrative of the text.
Third reason is might due to higher parameters a encoder-decoder models has.Thus to triaining this models we must have significantly high data while we have 66550 pair of question answering.

4.4.2 Why Use BOS and EOS

In training encoder-decoder models the use of special tokens for the beginning of sequence(BOS in our vocab <Start>) and end of sequence(EOS in our vocab <End>) is crucial.
The BOS token is used to signal the beginning of a sequence. When the decoder part of an encoder-decoder model starts generating an output sequence, the BOS token acts as the initial input.The EOS token, on the other hand, is crucial for indicating when a sequence should end. The EOS token tells the model when to stop generating further tokens, thus defining the length and completeness of the output sequence.
BOS and EOS tokens help standardize sequence lengths by clearly defining starting and stopping points, which are essential for batching and model training where tensors of consistent shapes are required.
Another reason is that the encode-decoder models learns not to just generate content but also where content generation should begin and end.Further, in inference model will continue generation till the EOS token is produced by the decoder!

5 Analysis

5.1 Quantity

For LSTM based we obtain 0.4 micro f1-score.In contrast for sequence to sequence we obtained 0.00! To be certain, we can not compare them with accuracy since the output of encoder-decoder is vocab size that we softmax on them but in LSTM we have 11 labels.

5.2 Quality

We will get two examples from validation since the test set has no srl labels to see the true labels. First two samples are: [”In”, ”the”, ”summer”, ”of”, ”2005”, ”,”, ”a”, ”picture”, ”that”, ”people”, ”have”, ”long”, ”been”, ”looking”, ”forward”, ”to”, ”started”, ”emerging”, ”with”, ”frequency”, ”in”, ”various”, ”major”, ”Hong”, ”Kong”, ”media”, ”.”] [”The”, ”most”, ”important”, ”thing”, ”about”, ”Disney”, ”is”, ”that”, ”it”, ”is”, ”a”, ”global”, ”brand”, ”.”] Now we are going to predict them with the LSTM therefore the predictions are: [’O’, ’O’, ’O’, ’O’, ’O’, ’O’,

’O’, ’O’, ’O’, ’I-ARG0’, ’O’, ’O’, ’O’, ’O’, ’B-ARG1’, ’O’, ’O’, ’I-ARG1’, ’I-ARG1’, ’I-ARG1’, ’I-ARG1’, ’I-ARG1’, ’I-ARG1’, ’I-ARG1’, ’O’, ’O’, ’O’] [’O’, ’O’, ’O’, ’O’, ’O’, ’O’, ’O’, ’O’, ’O’, ’O’, ’B-ARG2’, ’I-ARG2’, ’I-ARG2’, ’O’, ’O’, ’O’, ’O’, ’O’, ’O’, ’O’, ’O’, ’O’, ’O’, ’O’, ’O’, ’O’, ’O’] We implemented beam search for model it generates words that we can not evaluate on them based on the predictions above.Overally the second model are not good for our task since the limitations that said before.In the end the imbalance data for LSTM(due to padding process) and imbalance generation of data due to the classification report the LSTM based model can be better in our task.

The predictions for seq 2 seq has no meaning since it would generate words based on their frequency over training. Predictions are: [[’<PAD>’, ’<End>’, ’<PAD>’, ’<PAD>’, ’<PAD>’], [’<PAD>’, ’to’, ’the’, ’the’, ’the’], [’<PAD>’, ’<End>’, ’<PAD>’, ’<PAD>’, ’<PAD>’], [’<PAD>’, ’<End>’, ’<PAD>’, ’<PAD>’, ’<PAD>’], [’<PAD>’, ’<End>’, ’<PAD>’, ’<PAD>’, ’<PAD>’], [’<PAD>’, ’<End>’, ’<PAD>’, ’<PAD>’, ’<PAD>’], [’<PAD>’, ’it’, ’<End>’, ’<PAD>’, ’<PAD>’], [’<PAD>’, ’a’, ’<End>’, ’<PAD>’, ’<PAD>’], [’<PAD>’, ’<End>’, ’<PAD>’, ’<PAD>’, ’<PAD>’], [’<PAD>’, ’<End>’, ’<PAD>’, ’<PAD>’, ’<PAD>’]].

So in summary, the seq2seq are not good in our task either in quality or quantity(metrci).

Attention: The beam search is not useful here since the model just generate ’The’, ’to’, ’PAD’, ’END’, etc. so even with beam search we can not perfrom well in seq2seq for srl classifications.