MSAI 337 : Natural Language Processing

Homework 3

Task 1 : Classification Method

Model Used : bert_base_uncased

Dataset Structuring : **Step-wise Approach on how files are read and how each entry in a training set is provided to data loader**

In order to create trainable pairs of prompt vs label for this questions, we used the following approach.

1. First we read the jsonl files for each split of dataset in this process, we have each item in the prepared list representing a single question
2. Preparing the actual training example : A. For each question in the original dataset, we extract the question_text, fact, answer_key and 4 * Choices B. We combine the different parts of the question as following :

   ```
   [CLS] <fact> [SEP] <stem> [A] <text of choice #1> [END]
   [CLS] <fact> [SEP] <stem> [B] <text of choice #2> [END]
   [CLS] <fact> [SEP] <stem> [C] <text of choice #3> [END]
   [CLS] <fact> [SEP] <stem> [D] <text of choice #4> [END]
   ```

   C. Plus we use the `ord(answer_key) - ord('A')` to generate the label D. So for each question, we are passing the entire question + choices prompt and expecting the model to generate 4 logits representing the right choice.

Model Architecture : **How pre-trained model is used and explaination behind the intuition of architecture**

The model architecture is based on the pre-trained BERT model (bert-based-uncased) with the custiom layer added on top.

We have added a **freeze_bert** flag to switch between fine-tuning and zero-shot approaches.

```
class Model(nn.Module):
    def __init__(self, model_name, num_classes, batch_size=8, freeze_bert=False):
        super(Model, self).__init__()
        self.pre_trained = BertModel.from_pretrained(model_name)

        if(freeze_bert):
            for param in self.pre_trained.parameters():
                param.requires_grad = False


        self.fc = nn.Linear(self.pre_trained.config.hidden_size , 4)
```

```
    def forward(self, input_ids, attention_mask):
        #print(input_ids.shape, attention_mask.shape)
        output = self.pre_trained(input_ids = input_ids, attention_mask =
attention_mask)
        output = output.pooler_output
        output = self.fc(output)
        return output
```

Model Architecture

- A pre-trained BERT model is used to leverage rich contextual embeddings.
- A fully connected layer maps the BERT output to logits for each choice.
- Freezing the BERT model is optional and can be used to reduce the computational load and overfitting.

Task-Specific Adjustments

- Special Tokens [CLS] and [SEP] : The input text preparation involves combining the fact, question, and choices into a single string with special tokens to structure the input. This structured input is then tokenized and converted into tensors that BERT can process. The use of special tokens [CLS], [SEP], and [END] helps BERT understand the boundaries and relationships between different parts of the input, enabling it to make accurate predictions for the multiple-choice question answering task.

```
input_text = ""
        for choice, choice_val in zip(choices, ["A ","B ","C ", "D "]) :
            formatted_text = f"[CLS] {fact} [SEP] {question} [SEP] {choice} [SEP]
{choice['text']} [END]"
            input_text += formatted_text

        inputs = self.tokenizer.encode_plus(
                input_text,
                add_special_tokens=True,
                max_length=self.max_length,
                padding='max_length',
                truncation=True,
                return_tensors='pt'
            )
```

- Fully Connected Layer: After processing the input through BERT, we extract the representation of the [CLS] token, which captures the combined context of the question and a particular answer choice. A fully connected layer is added on top of BERT to map the [CLS] token's representation to a logit for each answer choice. This logit indicates the model's confidence that the choice is the correct answer.

```
self.fc = nn.Linear(self.pre_trained.config.hidden_size , 4)
```

- Handling Multiple Choices: Each question and its corresponding answer choices are concatenated and processed separately. This way, the model considers each choice in the context of the question, generating a separate logit for each choice. During inference, the choice with the highest logit is selected as the model's prediction for the correct answer.

Hyperparameter :

```
model_name = 'bert-base-uncased'
num_classes = 4
epochs = 3
learning_rate = 0.0001
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
max_length = 512
batch_size = 2
```

Results : **Zero-Shot vs Pre-trained**

We decided to test this implementation using 2 approaches. One is zero-shot where we freeze the layers of the BERT and essentially train the final linear layer only. Another approach is the fine-tuning where we do not freeze the BERT layers and train the entire network in tandem to learn the QA ability.

Results For Zero-Shot

Train Accuracies : 49.13%, 50.4%, 52.42% Validation Accuracy : 50.4%, 50.6%, 44.6% **Test set Accuracies : 52.8%**

Results for Fine-tuned Model

Training Accuracies : 51.9%, 52.3%, 51.6% Validation Accuracies : 50.2%, 50.5%, 50.6% **Test set Accuracies : 55.2%**

Short Comings of Approach

- **Context Limit** : Because we are using the full text of the question + choice in the inputs, we might face information loss in case the content of question is greater than 512 tokens. This can be resolved by using a choice level classification approach where we could categorize each choice as either correct or false based on question + choice text. Obviously this would lead to a complicated accuracy measurement function where we need to aggregate the results from the individual choices and determine if the answers were correct or resort to a Recall metric.
- **Complicated Learning Task** : Here since we are providing a large number of information to the model at one go, the learning task for the model is comparatively harder because it has to understand the attention across the entire text of question + choices rather than focusing on the individual choices.

Final Overview of the results

- Zero-shot model is generally worse than the fine-tuned model. This very much makes sense because without the model learning the actual context of the questions, it will rely on its existing training dataset to generate the representations without having access to the specific question context.

Task 2 : Generative Method

What we are doing in this approach :

- We are trying to generate the final token that represents the answer-key for the question.
- We are passing the prompt consisting of everything provided in the question text + choices except the final answer. We are ending the prompt with the [ANSWER] token and we are expecting the model to generate the token after [ANSWER] token. We will then be caculating the loss based on the last generated token of the model + known label encoding.
- We will be using the **GPT2** from huggingface as our base model.

Model Architecture

For this approach we are using the simple BERT base uncased model as the sole generative model

```
class GPT2ForMCQA(nn.Module):
    def __init__(self, text_model, take_last_token=True):
        super(GPT2ForMCQA, self).__init__()
        self.gpt = text_model
    def forward(self, input_ids, attention_mask, labels=None):
        outputs = self.gpt(input_ids=input_ids, attention_mask=attention_mask,
labels=labels)
        return outputs
```

Since we are not using any classification based approach for this model, we are using the GPT-2 as the only layer in the model.

Dataset Processing

In order to create the datset we are doing the following :

1. Read the data from the jsonl files and create a list containing each individual question.
2. We are constructing the prompt for the question as follows `prompt = f'[START] {fact} [SEP] {question} [SEP] [A] {choices[0]["text"]} [SEP] [B] {choices[1]["text"]} [SEP] [C] {choices[2]["text"]} [SEP] [D] {choices[3]["text"]} [ANSWER]'`
3. For label token we are using the answer_key value
4. For both the prompt as well as the label, we are then passing them to the bert tokenizer and generating their tokens + attention_masks

Hyperparameters

- Model: For this task choosing the pre-trained model can also be considered a hypermetr. In multiple-choice question answering, understanding the context of the question and each choice is crucial. GPT-2's ability to capture context over long sequences helps it understand the relationship between the fact, the question, and the answer choices. Also, GPT-2 can handle various prompt formats, making it versatile for designing inputs that include the fact, question, and multiple choices. The model can be prompted in a way that it generates the correct answer label based on the provided context.

```
prompt = f'[START] {fact} [SEP] {question} [SEP] [A] {choices[0]["text"]} [SEP]
[B] {choices[1]["text"]} [SEP] [C] {choices[2]["text"]} [SEP] [D] {choices[3]
["text"]} [ANSWER]'
```

- epochs: 10: Gives us the number of times the entire training dataset will be passed through the model during training. More epochs generally mean better learning, but there is a risk of overfitting if the model is trained for too many epochs. 10 epochs is a reasonable starting point for fine-tuning.
- learning_rate: 3e-5 is the step size at each iteration while moving toward a minimum of the loss function. A smaller learning rate means the model learns more slowly, but it can lead to more precise adjustments to the model parameters. 3e-5 is a commonly used learning rate for fine-tuning pre-trained language models.
- take_last_token -> True: Determines whether to use the last token's logits for loss calculation. The input sequence is structured such that the answer label (A, B, C, or D) is expected immediately after the [ANSWER] token. By focusing on the last token's logits, the model explicitly learns to generate the correct answer label in this position, making it easier to train the model to produce the desired output. Additionaly, instead of considering the logits for all tokens in the sequence, focusing on the last token simplifies the loss calculation and, by concentrating on the last token's logits, the model can better learn the task-specific requirement of generating the correct answer label, leading to more effective fine-tuning.
- max_length:The maximum length of the input sequence to the model set to 128. Sequences longer than this will be truncated, and shorter ones will be padded. 128 tokens is often sufficient for many tasks while keeping computational costs manageable.
- batch_size: The number of samples processed before the model's internal parameters are updated, set to 8.A smaller batch size allows for more frequent updates to the model parameters, potentially leading to faster convergence, but it also means noisier gradient estimates. 8 is a reasonable batch size considering memory constraints.
- tokenizer.pad_token -> tokenizer.eos_token: Sets the padding token to be the same as the end-of-sequence token. Ensures that padding is handled correctly by the model, which can be important for maintaining consistent input lengths.

Limitations

- Dependency on Tokenization and Special Tokens: The approach heavily relies on the correct tokenization and the use of special tokens like [START], [SEP], and [ANSWER]. Any inconsistency in how these tokens are used or understood by the model can lead to incorrect predictions.
- Model Size and Computational Resources: GPT-2 is a large model and require significant computational resources for fine-tuning and inference. This can be a barrier for those with limited access to high-performance hardware. Cloud based solution is suitable for this task.
- The generative approach might struggle with questions that have ambiguous contexts or where the correct answer depends on nuanced understanding. The model may generate plausible but incorrect answers if it misinterprets the context.
- Using exact match accuracy as the sole evaluation metric may not fully capture the model's performance, especially if the model generates answers that are semantically correct but not exact matches.

Results

Zero-shot We get 0% percent accuracy when using just the evaluation loop on the test set without prior training on the QA datset.

Fine-tuned

Validation Accuracies : 27.6%, 56.8%, 61.0%, 59.0%, 59.3%, 59.63%, 57.59%, 58.59%, 59.8%, 57.4% Test Accuracies : 61.6%