NLP HW2 Retrieval-based QA Report

姓名： 薛竣祐　　學號： 111526009

一. 資料前處理

因為原始資料給了51篇文章，這對於pre-trained model的512字上限來說太多了，因此使用bm25選擇與問題相關程度最高的5篇文章，並把這5篇文章各自視為不同筆的資料來訓練。相當於一個問題，有5篇文章及各自的target。若answer在文章內找不到一樣的詞，則將答案設為CLS的位置，若有答案，甚至在一篇文章中有複數個答案都將他們的index記錄下來。

同時為了避免此部分佔用太多時間，將運算玩的結果儲存為json，這樣就能在下此執行時省去這部分的時間及運算。

將上述資料使用DataLoader batch過後，送進pre-trained tokenizer來獲得token-word-char之間的轉換關係，並把token作為model的輸入。

```python
def read_dataset(path="./NLP-2-Dataset/train", have_ans=True, top_k=5, df_saved=True):
    def calc_bm25_score(row): # row: [articles, question]
        seqs = re.findall(r"<s>\W*(.*?)\W*<\/s>",row[0]) # split articles by <s> and </s> tags
        tokenized_article = [seq.split(" ") for seq in seqs]
        tokenized_question = row[1].split(" ")
        score = BM25Okapi(tokenized_article).get_top_n(tokenized_question, seqs, n=top_k)
        return score

    def find_index(row): # row: [article, answer]
        all_pos = [[m.start(), m.end()] for m in re.finditer(re.escape(row[1]),row[0])] or [[0,0]]
        return all_pos

    if df_saved and exists(path+".json"):
        return pd.read_json(path+".json")

    df = pd.read_csv(path+".txt", header=None, sep=r" \|\|\| ", names=["article", "question", "answer"], engine="python", encoding="utf-8")
    df = df.dropna(axis=1)
    tqdm.pandas(desc="Calc bm25 score", mininterval=1)
    df["article"] = df[["article","question"]].progress_apply(calc_bm25_score, axis=1) # select top 5 articles
    df = df.explode("article").reset_index(drop=True) # flatten list of articles
    if have_ans:
        tqdm.pandas(desc="Calc answer index", mininterval=1)
        df["answer_index"]= df[["article", "answer"]].progress_apply(find_index, axis=1) # find answer index in each article
    else:
        df.drop("answer", axis=1, inplace=True)
    if df_saved:
        df.to_json(path+".json", indent=4)
    return df
```

二. 模型架構

---

1. Pre-train Model(Freeze)

   使用hugging face上的deepset/roberta-base-squad2模型作為這次使用的pre-train model。考量訓練的時間成本，不使用large版本，也不額外增加hidden size，使用預設的(batch_size*seq_len*768)。

---

2. Linear

   在Pre-train Model後加上一層linear層，將最後的768降為2，將各自代表start_index及end_index，結束後的結果為(batch_size*seq_len*2)。

3. Softmax

此為out layer，使用softmax將seq_len那維視為機率，機率最高的就是目標index，輸出的index是token index，因此在模型之後還要進行額外處理。

```python
class QAModel(nn.Module):
    def __init__(self, model_name="deepset/roberta-base-squad2"):
        super().__init__()
        self.config = AutoConfig.from_pretrained(model_name)
        self.tokenizer = AutoTokenizer.from_pretrained(model_name)
        self.model = AutoModel.from_pretrained(model_name,config=self.config).to(device)
        for param in self.model.parameters():
            param.requires_grad = False
        self.linear = nn.Linear(self.model.config.hidden_size, 2)

    def forward(self, x):
        out = self.model(**x) # (batch_size, token_size(max512), 768)
        out = self.linear(out.last_hidden_state)# (batch_size, token_size, 2)
        out[:,:,0] = out[:,:,0].softmax(dim=-1) # (batch_size, token_size, 2) start
        out[:,:,1] = out[:,:,1].softmax(dim=-1) # (batch_size, token_size, 2) end
        return out
```

三. 模型訓練

此任務的資料處理較為複雜，在訓練過程經常需要轉換各種資料格式，因此我定義了一個QALoss class，內部的資料轉換都是為了計算Loss所需的。

大致上是將answer的char index轉換為token index，之後與預測出來的token index likelihood使用CrossEntroy計算Loss。

考量一篇文章可能有多個答案index，因此Loss在同篇文章將會取最小的Loss，並將整個batch作平均，取得最終Loss。

```python
class QALoss(nn.Module):
    def __init__(self, batch_size):
        super().__init__()
        self.loss_fn = nn.CrossEntropyLoss()
        self.batch_size = batch_size

    def forward(self, batch_pred_pos, batch_match_pos, c2t_fn):
        batch_match_token_pos = self.get_char_to_token(batch_match_pos, c2t_fn)
        batch_loss = []
        for match_pos, pred_pos in zip(batch_match_token_pos, batch_pred_pos): # loop in batch
            loss = torch.stack([ self.start_and_end_loss(pos, pred_pos) for pos in match_pos ]) # multiple answer
            batch_loss.append(loss.min()) # choose mini loss in multiple answer
        final_loss = torch.stack(batch_loss).mean() # mean the batch loss
        return final_loss

    def start_and_end_loss(self,target_pos, pred_pos):
        # loss = start loss + end loss
        return self.loss_fn(pred_pos[:,0], target_pos[0]) + self.loss_fn(pred_pos[:,1], target_pos[1])

    def get_char_to_token(self, batch_match_pos, c2t_fn):
        batch_match_token_pos = []
        for i, match_pos in enumerate(batch_match_pos):
            if match_pos == [[0,0]]:
                batch_match_token_pos.append(torch.tensor(match_pos).to(device))
            else:
                batch_match_token_pos.append(torch.tensor([[c2t_fn(i, pos[0], 1), c2t_fn(i, pos[1]-1, 1)] for pos in match_pos]).to(device))
        return batch_match_token_pos
```

在訓練部分則定義了一個Trainer，內部定義了training、validate、testing時不同的行為。

Training：

tokenize->model->loss->backward->(save checkpoint、validate)

```python
def train(self):
    self.model.train()
    self.init_dataloader("train")
    p_bar = tqdm(self.train_dataloader, mininterval=1, desc="Training Batch, loss=0.0000", leave=False)

    for i, (batch_articles, batch_questions, _, batch_match_pos) in enumerate(p_bar):
        token = self.model.tokenizer(batch_questions, batch_articles, return_tensors="pt", padding=True, truncation=True).to(device)
        batch_pred_pos = self.model(token) # out put token index
        loss = self.qa_loss(batch_pred_pos, batch_match_pos, token.char_to_token)
        p_bar.set_description(f"Training, loss={loss:.4f}")
        loss.backward()
        self.optimizer.step()
        self.optimizer.zero_grad()
        if i+1 % self.save_interval == 0:
            self.save_checkpoint()
        if i+1 % self.val_interval == 0:
            self.validate()
    self.save_checkpoint()
    self.validate()
```

Validate：

tokenize->model->loss->f1

```python
def validate(self):
    self.model.eval()
    self.init_dataloader("val")
    f1_scores = []
    with torch.no_grad():
        p_bar = tqdm(self.val_dataloader, mininterval=1, desc="Validation Batch, loss=0.0000", leave=True)
        for batch_articles, batch_questions, batch_answer, batch_match_pos in p_bar:
            token = self.model.tokenizer(batch_questions, batch_articles, return_tensors="pt", padding=True, truncation=True).to(device)
            batch_pred_pos = self.model(token)
            loss = self.qa_loss(batch_pred_pos, batch_match_pos, token.char_to_token)
            p_bar.set_description("Testing, loss={:.4f}".format(loss.item()))
            batch_pred = self.pred_to_seq_index(batch_pred_pos, token)
            for target, article, pred in zip(batch_answer,batch_articles, batch_pred):
                f1_scores.append(compute_f1(target,article[pred[0]:pred[1]]))
    print(sum(f1_scores)/len(f1_scores))
    self.model.train()
```

Testing：

tokenize->model->token to char->choose best answer

```python
def predict(self, batch_size): # one batch = one question
    self.model.eval()
    self.init_dataloader("test", batch_size)
    with torch.no_grad():
        with open("test-submit-a.txt", "w") as f:
            p_bar = tqdm(self.test_dataloader, mininterval=1, desc="Testing Batch", leave=True)
            for batch_articles, batch_questions in p_bar:
                token = self.model.tokenizer(batch_questions, batch_articles, return_tensors="pt", padding=True, truncation=True).to(device)
                batch_pred_pos = self.model(token)
                batch_index = self.pred_to_seq_index(batch_pred_pos, token)
                answers = [article[index[0]:index[1]] for article, index in zip(batch_articles, batch_index)]
                most_common_ans = Counter(answers).most_common()[0][0]
                f.write(f"{batch_questions[0]} ||| {most_common_ans}\n")
    self.model.train()
```

四. 實驗成果分析與改進

本次實驗比較了兩種訓練方式，第一種為freeze pre-trained model，第二種則為不freeze。

執行時間(Colab)：freeze約為20分鐘，不freeze約為40分鐘。
Training Loss：兩者的皆在7~9之間徘徊，無明顯的下降趨勢。
Validate Loss：兩者的皆在8~9之間徘徊，無明顯的下降趨勢。
F1 Score：兩者皆小於0.1

上述描述可見，本次的訓練結果效果相當不彰，不僅Loss沒有再下降，F1 Score的成績也很詭異。
目前猜想有幾個問題可能有待改進：

1. F1 Score的計算是由token轉word再轉char，過程複雜，中間可能有計算錯誤。
2. Loss的計算方式也許可以簡單一點，可能不需要在單篇文章中尋找多個解答位置，另外Loss是將ans轉為token計算，可能可以跟F1 Score一樣，統一轉成char來計算。
3. 可以多嘗試不同optimizer與learning rate等超參數。
4. 花太多時間在解決與釐清執行效能與OOM問題，儘管整體架構看似完整，但實際效果不好，可以嘗試將心力多花在設計模型訓練步驟上。