HW2 Report

姓名：黃浩恩

學號：f05921120

Coworker：賴棹沅、謝忱、嚴聲揚

Q1 Describe your ELMo model

Q1.1 Training corpus processing. (tokenization, vocabulary building) (0.4%)

先透過助教給的 token.txt 做 split()得到 tokenization，再透過 Counter()彙整出字典，並以該單字出現 3 次以上之字，按照出現次數高到低排列，做為新的字典內容，Character 字數出現 1000 次也將其作成 Character 的字典。目前產生的 Word 字典約為 127000 字左右，Character 的字典約 100 左右。

Q1.2 Model architecture and implementation details. (0.4%)

先透過助教提供的 char_embedding 產生出 emb，然後丟進自己手刻之 Elmo 所建立的雙向 LSTM 做訓練，最後把輸出結果擴展至字典的大小後 softmax，與下一個字的 index 做 CrossEntropyLoss()，進而更新模型參數。

```
class ElmoNet(nn.Module):
    def __init__(self, projection_size, dim, dictionary_size):
        super().__init__()

        # forward net
        self.lstm_forward_1 = torch.nn.LSTM(input_size=projection_size, hidden_size=dim)
        self.linear_forward_1 = torch.nn.Linear(dim, int(dim / 2))

        self.lstm_forward_2 = torch.nn.LSTM(input_size=int(dim / 2), hidden_size=dim)
        self.linear_forward_2 = torch.nn.Linear(dim, dictionary_size)

        # backward net
        self.lstm_backward_1 = torch.nn.LSTM(input_size=projection_size, hidden_size=dim)
        self.linear_backward_1 = torch.nn.Linear(dim, int(dim / 2))

        self.lstm_backward_2 = torch.nn.LSTM(input_size=int(dim / 2), hidden_size=dim)
        self.linear_backward_2 = torch.nn.Linear(dim, dictionary_size)
        self.softmax = torch.nn.Softmax()

    def forward(self, _charEmbedding_forward, _charEmbedding_backward):
        forward, (_, _) = self.lstm_forward_1(_charEmbedding_forward)
        forward = self.linear_forward_1(forward)
        forward, (_, _) = self.lstm_forward_2(forward)
        forward = self.linear_forward_2(forward)
        forward = self.softmax(forward)

        backward, (_, _) = self.lstm_forward_1(_charEmbedding_backward)
        backward = self.linear_forward_1(backward)
        backward, (_, _) = self.lstm_forward_2(backward)
        backward = self.linear_forward_2(backward)
        backward = self.softmax(backward)

        return forward, backward
```

Q1.3 Hyperparameters of your ELMo model. (number of layers, hidden dimension, output dimension, optimization algorithm, learning rate and batch size) (0.4%)

Layers：LSTM 的 layers 都設定為 1。

hidden dimension：設定為 512。

output dimension：因應字典長度(len(word_dictionaries))，故選擇。

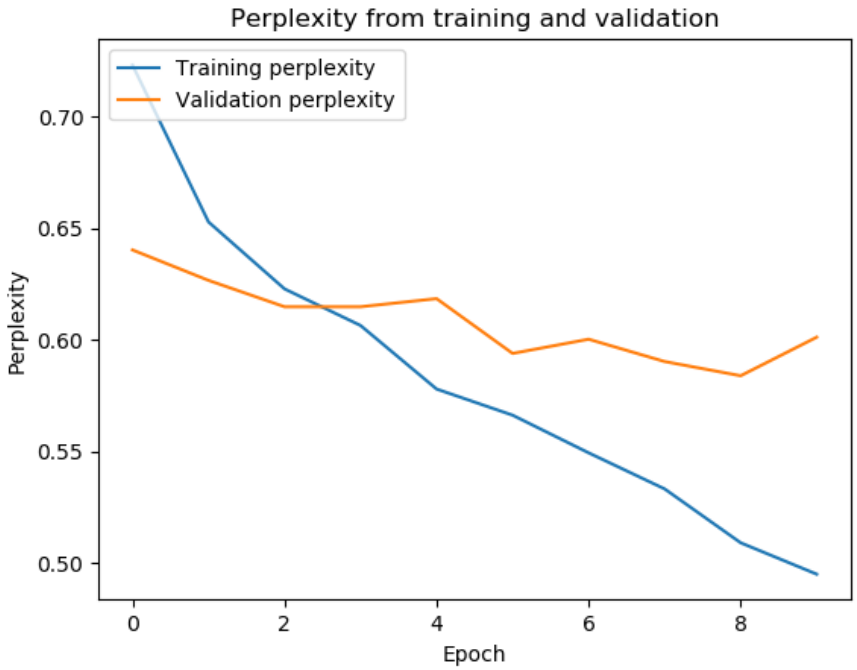
optimization algorithm：使用 Adam。

learning rate：使用 1e-3。

batch size：因 GPU 記憶體不足，故採用 14。

Q1.4 Plot the perplexity score on train/dev set while training. (0.4%)

如圖，透過 10 個 epoch，訓練之結果如下，運算方式為(1-acc)。



於圖可知，acc 於 training 時有一直在上升，使困惑度下降，但是於 validation 之結果可以看出可能是 overfitting 了，但是目前還沒有特別實作 dropout。

Q1.5 Show the performance of the BCN model with and without ELMo on the public leaderboard. (0.4%)

BCN model without ELMo	BCN model with ELMo
0.42352	0.47330

Q2 Compare different settings for ELMo

Q2.1 Different number of training steps.

於 Validation acc 分數比較如下表：

Different number of training steps	Acc
5	0.3842
10	0.4256
15	0.4677

Q2.2 Different hyperparameters.

於 Validation acc 分數比較如下表：

Different hyperparameters	Acc @ epoch 10	Training Speed
Layers：1 hidden dimension：512 output dimension： len(word_dictionaries) optimization algorithm： Adam learning rate：2e-4 batch size：14	0.4256	19.19 it/s
Layers：2 hidden dimension：1024 output dimension： len(word_dictionaries) optimization algorithm： Adam learning rate：5e-4 batch size：14	0.4039	12.19 it/s

目前判斷可能因為 training data 可能太少，於第二個 situation 把 model 設得太複雜了，導致可能還沒訓練得很好，但是 training 的速度幾乎慢了快要一倍...。

Q3 Describe your model that passes strong baseline.

Q3.1 Input of your model. (word embedding? character embedding?)

於 Passes Strong Baseline 之 model 為 BERT，故直接將文字進行輸入，再透過內部 tokenization 進行 token 與 token2index，故 input 至 model 為 word embedding。

Q3.2 Model architecture.

使用 BERT 架構的 BertForSequenceClassification Class，於這個 model 中，將 BERT_tokenize 後的結果丟入 BERT model 中，就可以透過 Linear 來做 Classification，故可以做該作業的選擇題。

```
class BertForSequenceClassification(BertPreTrainedModel):
    def __init__(self, config, num_labels):
        super(BertForSequenceClassification, self).__init__(config)
        self.num_labels = num_labels
        self.bert = BertModel(config)
        self.dropout = nn.Dropout(config.hidden_dropout_prob)
        self.classifier = nn.Linear(config.hidden_size, num_labels)
        self.apply(self.init_bert_weights)

    def forward(self, input_ids, token_type_ids=None, attention_mask=None, labels=None):
        _, pooled_output = self.bert(input_ids, token_type_ids, attention_mask, output_all_encoded_layers=False)
        pooled_output = self.dropout(pooled_output)
        logits = self.classifier(pooled_output)

        if labels is not None:
            loss_fct = CrossEntropyLoss()
            loss = loss_fct(logits.view(-1, self.num_labels), labels.view(-1))
            return loss
        else:
            return logits
```

Q3.3 Hyperparameters of your model.

以下為使用 BERT 之 Hyperparameters：

bert_model：採用 BERT 基本之 model [bert-base-uncased]

seq_length：最長之字句長度為 128。

train_batch_size：因 GPU 記憶體之限制，選擇 16。

learning_rate：採用 2e-4 作為 LR。

num_train_epochs：20。

optimization algorithm：透過 BERT 內建之 class BertAdam(Optimizer)來當作 optimization。

Q4 Describe your best model.

Q4.1 Describe your best model

Q4.1.a Input to your model

於 Passes Strong Baseline 之 model 為 BERT，故直接將文字進行輸入，再透過內部 tokenization 進行 token 與 token2index，故 input 至 model 為 word embedding。

Q4.1.b Model architecture

使用 BERT 架構的 BertForSequenceClassification Class，於這個 model 中，將 BERT_tokenize 後的結果丟入 BERT model 中，就可以透過 Linear 來做 Classification，故可以做該作業的選擇題。

```
class BertForSequenceClassification(BertPreTrainedModel):
    def __init__(self, config, num_labels):
        super(BertForSequenceClassification, self).__init__(config)
        self.num_labels = num_labels
        self.bert = BertModel(config)
        self.dropout = nn.Dropout(config.hidden_dropout_prob)
        self.classifier = nn.Linear(config.hidden_size, num_labels)
        self.apply(self.init_bert_weights)

    def forward(self, input_ids, token_type_ids=None, attention_mask=None, labels=None):
        _, pooled_output = self.bert(input_ids, token_type_ids, attention_mask, output_all_encoded_layers=False)
        pooled_output = self.dropout(pooled_output)
        logits = self.classifier(pooled_output)

        if labels is not None:
            loss_fct = CrossEntropyLoss()
            loss = loss_fct(logits.view(-1, self.num_labels), labels.view(-1))
            return loss
        else:
            return logits
```

Q4.1.c Hyperparameters (optimization algorithm, learning rate, batch size and other model-specific options)

bert_model：採用 BERT 基本之 model [bert-base-uncased]

seq_length：最長之字句長度為 128。

train_batch_size：因 GPU 記憶體之限制，選擇 16。

learning_rate：採用 2e-4 作為 LR。

num_train_epochs：20。

optimization algorithm：透過 BERT 內建之 class BertAdam(Optimizer)來當作 optimization。

Q4.2 Describe the reason you think why your best model performs better than other models.

BERT 現在用的 pre-training 是透過 4~16 個 Google TPU 上運算四天的結果，故光是 BERT 所使用的 pre-training 就是一個非強大的 pre-training。BERT 建立在預訓練上下文的基礎上，透過深度雙向網路與無監督的語言，故 BERT 可以了解大體上該 word 於上下文的關係。

Q5 Compare different input embeddings.

Character embedding 的好處其實有很多，比如字典數量小、輕便，使 OOV 的機率大大降低。但是需要透過很大的運算資源與記憶體去存放展開來後的結果(split 之運算量與許多 padding 的記憶體空間)，使跑程式上需要許多資源。

Word embedding 則是字典量非常大，且些許的字出現次數少，卻需要存放於字典中，記憶體消耗資源比 Character embedding 有時候來的大，另外亦需要準備極大的 embedding matrix 提供訓練時使用，故記憶體資源消耗會很大。

byte pair 原本作為壓縮算法被發明，目標是找出一種碼表，將最頻繁的 bype pair 壓縮替換為一個新的 byte。被應用於 embeddings 上時，以 character 為基礎，在上面找字詞中最常出現的 pair，並另存為新的字典，但是更可以掌握到單字之 character 之 pair 關係，字典又不會太大，使 model size 與 dictionary size 都有不錯的記憶體使用表現。

Q6 BERT

Q6.1 Please describe the tasks that BERT used for pre-training. What benefits do they have comparing to language model pretraining used in ELMo?

BERT 現在用的 pre-training 是透過 4~16 個 Google TPU 上運算四天的結果，故光是 BERT 所使用的 pre-training 就是一個非強大的 pre-training。BERT 建立在預訓練上下文的基礎上，透過深度雙向網路與無監督的語言，故 BERT 可以了解大體上該 word 於上下文的關係。

Q6.2 Please describe in detail how you would formulate the problem in HW1 and apply BERT on it.

BERT 原先的 example 有提供看上下文做 Binary 選擇的功能，故可能會透過該 example 做延伸。

將每個對話的後三個 context 做成 str，先行放在上文，再將每個 option 放在下文(可能採用 1 對 9 錯的比例)，再透過 label 告訴 BERT，哪種上下文的對應是正確的，進而進行 Binary 選擇。

Q7 Bonus

Q7.1 Compare more than 2 contextualized embedding on this task.

沒做。

Q7.2 Apply your ELMo embedding to other tasks.

沒做。

Q7.3 Apply any kind of contextualized embedding method on HW1 and report the performance.

這邊實作將 BERT 應用於 HW1 上。比較於 Q6.2，發現當初想的太理想，採用 1 對 9 錯的比例結果大悲劇，上下文的對應資料量瞬間增加 10 倍，故最後採用 1 對 4 錯的比例，而且僅切了 3 分之 1 的原始資料當作 training set(因為好像超過 Bert example 的訓練資料的上限，暫時沒有去更改)，設定參數如下：

Hyperparameters

Batch：16

Epoch：10

Learning rate：1e-4

Max seq length：128

train 了 6 小時候，透過 metrics 反映訓練結果

	metrices score without BERT	metrices score with attention without BERT	metrices score with BERT
Training	0.7027	0.7562	0.8242
Validation	0.5942	0.6355	0.7636

這裡提點一下，因為以前採用 1 對 9 錯，故 metrices 分數一開始從 0.1 開始爬，但是目前採用的 BERT@HW1 上面，是 1 對 4 錯，本身訓練的基準就以 0.2 開始，並且有很快的進步速度，這點可能於分數比較上沒有一個基準的開始，有點不客觀(之前 HW1 設定不同的對錯比數，metrics 其實不太準)，不過整體來說 BERT 表現挺出色的，若 HW1 早點用，就可能可以過 strong 了…。