Applied Deep Learning Homework 1 report

M10915045 施信宏

Q1 : Data processing

Ans1:

使用 github 上的 sample code，以及使用 GloVe 中 glove.840B.300d 作為 pre-trained embedding。

Intent classification 及 slog-Tagging 的前處理方法相同。

讀取[train,eval] json 檔內的內容後，分別用 Counter 及 Set 計算單字出現字數及類別名稱後，用 Set 的特性即可得到不重複的類別名，此時存成 tag2idx 及 intent2idx 檔案，再呼叫 build_vocab，將 Counter 內的單字存成 pickle 檔，供之後訓練及測試階段使用，然後開啟 glove.840B.300d 內容，找取 Counter 內單字對應的向量，並存入 Dict 中，找尋完全部單字後，製作 embedding matrix ， 若單字不存在於 pre-trained embe-dding 中，則使用 raddom() * 2 – 1 製作 300 維向量作為替代，處理完成後存成 embedding.pt 供之後 model 的 embedding 使用。在此 function build_vocab 完成，得到訓練集及測試集所有單字，及相對應的 embedding matrix。

Q2: Describe intent classification model

A : model

model 的設計如圖 1 所示，在此使用雙向 LSTM 架構進行分類，使用雙向將使輸出的 hidden_size 變為 2 倍，因前後都需走訪更新，並使用 batch_first 將輸入及輸出改變成(batch, seq_len, input_size)，進入 LSTM 訓練後的輸出為(batch, seq_len, 2 * hidden_size)。

Model 的 forward 設計如圖 2，先將 batch(由 dataloader 輸出的 context vector，皆 padding 至 max_len = 64，shape = [batch_size , max_len]) 丟入 embedding_layer 將 context vector 轉換成 embedding Vector(shape = [batch_size , max_len,embedding_dim])，此時便可將 embedding Vector 作為 input 放入 LSTM 做訓練。

LSTM 的輸出為[batch_size,seq_len,hidden_size * 2]，因此任務為多對一分類問題，只需取得最後一個時間點的 hidden state 放入另一個全連接層，輸出為 150 個分類，即可完成第一個任務。

若使用 LSTM 需將所有時間的 hidden state 取平均，在 Q5 討論。

```python
class SeqClassifier(torch.nn.Module):
    def __init__(
        self,
        embeddings: torch.tensor,
        hidden_size: int,
        num_layers: int,
        dropout: float,
        bidirectional: bool,
        num_class: int,
        pad_idx
    ) -> None:
        super(SeqClassifier, self).__init__()
        self.embeddings = Embedding.from_pretrained(embeddings, freeze=False , padding_idx= pad_idx)
        # TODO: model architecture
        self.embedding_dim = embeddings.size(1)
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.dropout = dropout
        self.output_dim = num_class
        self.rnn = nn.LSTM(embeddings.size(1), hidden_size, num_layers = num_layers,bidirectional = bidirectional ,
        batch_first= True)
        self.classifier = nn.Sequential( nn.Dropout(dropout),
                                         nn.Linear(self.hidden_size * 2, self.output_dim)
                                        )
```

圖 1

```python
def forward(self, batch):
    inputs = self.embeddings(batch)
    x, _  = self.rnn(inputs,None)
    # x  dimension (batch, seq_len, hidden_size)
    # get rnn final layer hidden state
    #x = x[:, -1, :]
    x = torch.mean(x, dim = 1)
    x = self.classifier(x)
    return x
```

圖 2

B： performance

在 kaggle public leader board 得到 0.86577

C : loss function

使用 crossentropyloss 作為 loss function，在多分類上具有良好的效果，pytorch 結合了 nn.LogSoftmax()和 nn.NLLLoss()兩個函數。

D : optimization algorithm

使用 Adam，為 RMSProp 與 Momentum 的結合，具有 RMSProp 的自適應調整 learning rate 的特性，亦具有 Momentum 可記錄之前更新的方向，在更新 gradient 上有良好的效果，Learning rate 初始值為 1e-3，batch_size 設定為 128。

Q3: Describe slot tagging model

A : model

model 的設計如圖 3 所示，在此使用雙向 LSTM 架構進行多對多分類，相關 shape 及變數內容解釋如 Q2 所述，故在此不重複講解。

在分類的類別上多出了一個類別 :<PAD> ，讓一個句子 padding 的部分對應到類別<PAD>，故為 10 分類問題。

Model 的 forward 設計如圖 4，先將 batch(由 dataloader 輸出的 context vector，皆 padding 至 max_len = 64，shape = [batch_size , max_len]) 丟入 embedding_layer 將 context vector 轉換成 embedding Vector(shape = [batch_size , max_len,embedding_dim])，此時便可將 embedding Vector 作為 input 放入 LSTM 做訓練。

LSTM 的輸出為[batch_size,seq_len,hidden_size * 2]，因此任務為多對多分類問題，每一個時間點都需輸出一個類別，故將 LSTM 的輸出直接放入全連接層中，則在每個時間點都有 10 個類別的對應值，取最大的作為此時間點單字的 tagging，持續數個 epoch 後完成訓練。

```python
def __init__(
    self,
    embeddings: torch.tensor,
    hidden_size: int,
    num_layers: int,
    dropout: float,
    bidirectional: bool,
    num_class: int,
    pad_idx
) -> None:
    super(SlotClassifier, self).__init__()
    self.embeddings = Embedding.from_pretrained(embeddings, freeze=False , padding_idx = pad_idx)
    # TODO: model architecture
    self.embedding_dim = embeddings.size(1)
    self.hidden_size = hidden_size
    self.num_layers = num_layers
    self.output_dim = num_class
    self.rnn = nn.LSTM(embeddings.size(1), hidden_size, num_layers = num_layers,bidirectional = bidirectional ,
    batch_first= True)
    self.fc = nn.Linear(self.hidden_size * 2 if bidirectional else hidden_size, self.output_dim)
    self.dropout = nn.Dropout(dropout)
```

圖 3

```python
def forward(self, batch):
    inputs = self.dropout(self.embeddings(batch))
    x, _  = self.rnn(inputs,None)
    # x  dimension (batch, seq_len, hidden_size)
    # get GRU final layer hidden state
    predictions = self.fc(self.dropout(x))
    return predictions
```

圖 4

B : performance

在 kaggle public leader board 得到 0.78605

C : loss function

使用 crossentropyloss 作為 loss function，在多分類上具有良好的效果，pytorch 結合了 nn.LogSoftmax()和 nn.NLLLoss()兩個函數。

D : optimization algorithm

使用 Adam，為 RMSProp 與 Momentum 的結合，具有 RMSProp 的自適應調整 learning rate 的特性，亦具有 Momentum 可記錄之前更新的方向，在更新 gradient 上有良好的效果，Learning rate 初始值為 1e-3，batch_size 設定為 128。

Q4 : Sequence Tagging Evaluation

Ans:

Validation set 的 token accuracy , joint accuracy , classification report 在第 10 個 epoch 下的數值如圖 5。

```
[ Epoch10: 57/57 ] loss:0.007 acc:97.674
Train | Loss:0.00612 token accuracy: 98.155
Valid | Loss:0.01475
Valid | token accuracy: 96.214
Joint Accuracy : 77.8%
              precision    recall  f1-score   support

        date       0.71      0.70      0.70       206
  first_name       0.93      0.94      0.94       102
   last_name       0.89      0.69      0.78        78
      people       0.73      0.71      0.72       238
        time       0.85      0.83      0.84       218

   micro avg       0.79      0.76      0.78       842
   macro avg       0.82      0.77      0.79       842
weighted avg       0.79      0.76      0.78       842
```

圖 5

	Predicted Positives	Predicted Negatives
Positives	True Positives	False Negatives
Negatives	False Positives	True Negatives

Recall : True Positive / (True Positive + False Negative)
Precision : True Positive / (True Positive + False Positive)
F1-score = 2 * Precision * Recall / (Precision + Recall)

在此任務中，Precision 高可視為較謹慎的模型，雖然時常沒辦法抓出實際詞性，但只要有抓出幾乎都是正確的，而後者則是一個寬鬆的模型，雖然有時候會抓錯，但幾乎該抓的都有抓到(Recall 高)

F1-score 是「precision」和「recall」的調和平均數，可看作是該二指標的綜合指標，能較全面地評斷模型的表現。

在 token accuracy 下達到 96.2%的準確率，代表大多數的詞性預測正確，但在 joint accuracy 下卻只有 77.8%的準確率，此時可以依據 classification report 的報表做猜測。

可以發現在 first_name 這一部分的分數達到 0.94，代表訓練上在此部分成功，但在 date,people 上的分數卻差強人意，但也十分合理。日期的表示法有許多種，而人名的部分又有許多，若驗證集資料的單字未出現在訓練集資料上，便容易判斷錯誤導致 joint accuracy 下降，但 time 的分數有達 0.85，估計是資料上的格式近似，沒有難理解的表示方法，故在 F1-score 上仍有可接受的分數；由此推估，造成 joint accuracy 不高的情況，大多的情況可能為在 date,people,time 上的預測錯誤。

Q5: Compare with different configurations

Ans:

在 intent classification 有使用不同的架構做訓練，分別為 GRU 及 LSTM。

使用 GRU 時，能直接使用最後一層的 hidden state 放入輸出為 150 個分類的全連接層中，並能順利完成訓練。若使用 LSTM，經過多個 epoch，loss 仍無法下降，這問題有去請教助教，助教的建議是將所有時間點的 hidden state 取平均，再放入全連接層當中，改成這個寫法便能使 LSTM 訓練成功，但當中的原因我還未搞的清楚。

比較 GRU 及 LSTM 兩者的準確率，確實相差不遠，在 kaggle 的 public leader board 分別得到 0.85688，0.85422；甚至 LSTM 沒有比 GRU 表現來的優秀，參數部分則分別為 880 萬及 1100 萬，訓練上的收斂速度大約皆在第五個 epoch 上就完成了收斂，loss 來到了 0.01，但 LSTM 的消耗成本高，也能理解現今較多人使用 GRU 的原因。

確認模型訓練無誤後，將訓練集及測試集一起放入網路中做訓練，增加資料量，能使 kaggle public leader board 上的分數來到 0.86577，提升了約 1%的準確率。

至於較進階的方法未仔細閱讀就沒有套用在實作上，但 kaggle 上蠻多人的分數達到 0.9 以上，應也是套用了些許技巧，單單改變 rnn 架構應只能提高至 0.89 左右。