0516003 李智嘉 lab5 report

1. Introduction

這次lab將用CVAE實做動詞型態轉換，也就是sequence-to-sequence CVAE，我們的encoder與decoder都是利用GRU unit。

在encoder我們會加入input的動詞型態，而decoder則是我們想要轉換成的型態。輸入動詞時，我們會先將每個字母分開先做embedding，而後一一丟入GRU裡面進行運算。在decoder方面，我們會將前一個unit的output，當作是此次unit的input，而在training階段會有一定機率將此input變成ground truth，使他不至於一直吃到錯誤的東西而學不起來。

2. Implementation details

a. model and hyperparameters

首先是dataloader，分別是TrainLoader class, TestLoader class以及處理data的function: get_data

get_data在training時，會將train.txt裡面的文字都讀出來，並包成16種可能的轉換組合，格式是(idx1, word[idx1]), (idx2, word[idx2])（第21行），以供使用，在testing時則是很單純的依照空白切割（第26行）。

而TrainLoader很直接的回傳data length以及要什麼data（63行即66行），TestLoader則是會把動詞和他的時態標注都放好後再回傳（48, 49行就是在放時態進去）

```
5  def get_data(mode, alphabet):
6      data = []
7      if mode == 'train':
8          with open('data/train.txt', 'r') as fo:
9              w = fo.readlines()
10             for line in w:
11                 word_list = line[:-1].split(' ')
12                 alpha_list = []
13                 for word in word_list:
14                     temp = []
15                     for alpha in word:
16                         temp.append(alphabet[alpha])
17                     alpha_list.append(np.array(temp))
18
19                 for i in range(4):
20                     for j in range(4):
21                         data.append([(i ,alpha_list[i]), (j, alpha_list[j])])
22     elif mode == 'test':
23         with open('data/test.txt', 'r') as fo:
24             w = fo.readlines()
25             for line in w:
26                 data.append(line[:-1].split(' '))
27     return data
```

```
53 class TrainLoader(Dataset):
54     def __init__(self, mode):
55         self.mode = mode
56         self.alphabet = dict()
57         for idx, alpha in enumerate(string.ascii_lowercase):
58             self.alphabet[alpha] = idx + 2
59         self.data = get_data(mode, self.alphabet)
60         self.data_len = len(self.data)
61
62     def __len__(self):
63         return self.data_len
64
65     def __getitem__(self, idx):
66         return self.data[idx]
```

```
class TestLoader(Dataset):
    def __init__(self, mode):
        self.mode = mode
        self.alphabet = dict()
        for idx, alpha in enumerate(string.ascii_lowercase):
            self.alphabet[alpha] = idx + 2
        self.data = get_data(mode, self.alphabet)
        self.data_len = len(self.data)

    def __len__(self):
        return self.data_len

    def __getitem__(self, idx):
        data = []
        for word in self.data[idx][:2]:
            temp = []
            for alpha in word:
                temp.append(self.alphabet[alpha])
            data.append(np.array(temp))
        data.append(self.data[idx][2])
        data.append(self.data[idx][3])
        data = np.array(data)
        return data
```

再來說明model，這邊我分成EncoderRNN, EncodeLast, DecoderRNN，整體架構是：先將input 經過EncoderRNN後，產生出一個高維的hidden vector，再將這個hidden vector丟進EncodeLast，使其透過fully connect的方式產生出mean vector以及log_variance vector，再去sample一個mean=0, variance=1的Gaussian一些值，用這些值與方才的兩個輸出做運算，最後丟進fully connect產生要丟進DecoderRNN的hidden vector然後產出結果，詳細程式碼如下，可以看到在Encoder與Decoder分別都有nn.Embedding來embed字母，embed完後再丟進RNN運算。

```python
class EncoderRNN(nn.Module):
    def __init__(self, input_size, hidden_size, device):
        super(EncoderRNN, self).__init__()
        self.hidden_size = hidden_size

        self.embedding = nn.Embedding(input_size, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size)
        self.device = device

    def forward(self, input, hidden):
        embedded = self.embedding(input).view(1, 1, -1)
        output, hidden = self.gru(embedded, hidden)
        return output, hidden

    def initHidden(self):
        return torch.zeros(1, 1, self.hidden_size-4, device=self.device, dtype=torch.float32)
```

```python
class EncodeLast(nn.Module):
    def __init__(self, size, output_size, device):
        super().__init__()
        self.fc_mu = nn.Linear(in_features=size, out_features=output_size)
        self.fc_logvr = nn.Linear(in_features=size, out_features=output_size)
        self.fc_emb = nn.Linear(in_features=output_size, out_features=size-4)
        self.device = device
    def forward(self, embedding):
        mu = self.fc_mu(embedding)
        log_var = self.fc_logvr(embedding)
        std = torch.exp(log_var/2)

        tmp = torch.randn_like(std)
        embedding = mu + tmp*std
        embedding = self.fc_emb(embedding)

        return embedding, mu, log_var
```

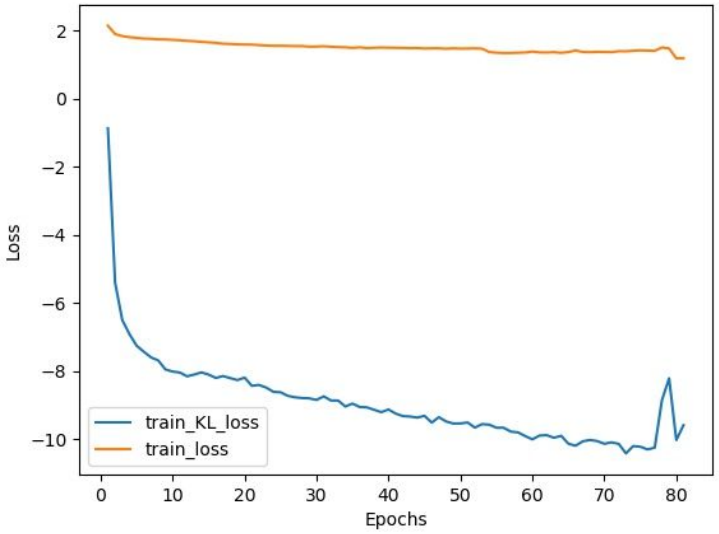
```python
class DecoderRNN(nn.Module):
    def __init__(self, hidden_size, output_size, device):
        super(DecoderRNN, self).__init__()
        self.hidden_size = hidden_size

        self.embedding = nn.Embedding(output_size, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size)
        self.out = nn.Linear(hidden_size, output_size)
        self.device = device

    def forward(self, input, hidden):
        output = self.embedding(input).view(1, 1, -1)
        output = F.relu(output)
        output, hidden = self.gru(output, hidden)
        output = self.out(output[0])
        return output, hidden

    def initHidden(self):
        return torch.zeros(1, 1, self.hidden_size-4, device=self.device, dtype=torch.float32)
```

最後是整體訓練流程，先介紹我的各個參數：
condition embedding = 4（這邊我直接用one-hot vector，後來想想應該要先embed後效果應該會比較好，可是距離作業deadline已經來不及就沒改了）
Learning Rate = 0.05
latent hidden size = 32
hidden_size = 256 + 4(condition code) = 260
而整份作業精華的點也就是在怎麼訓練這個sequence-to-sequece CVAE
其實我試過很多方法，一開始直接將KLD_weight在每筆data後逐漸增加，慢慢到1就停止，這樣的問題是會讓model直接學不起來。
後來我改成手動規劃upper bound，也就是這次不要直接到1了，而是到那個upper bound就停止，但訓練的速度很難預測，所以成效也不是很好。
再來我又嘗試將KLD weight的增加搬到Epoch為一單位，也就是同一個Epoch都會是同樣的KLD weight，但這樣的問題就是，一開始在訓練時，KLD weight會非常低，低到model會無限制的不管mean和log_variance是多少，導致後面要算log_variance的expoential時，會直接超過數值，讓CUDA crash，所以也不行
最後我是利用前一次的bleu來規劃下次的upper bound，例如如果這次的bleu值介於0.5~0.6之間的話，下次的teacher forcing ratio就是0.4，KLD upper bound就是0.5，之類，效果還不錯，但還是沒有成功訓練上去，
後來我用了一個還不錯的weight再去用方才提到的同樣Epoch同樣KLD weight的方式去train，可以train到很好的bleu，甚至到1.0，但KL loss就沒有很好，以至於在用Gaussian產生文字時表現很糟糕，但最後還是有得到一個折中的model，下面會再用圖片說明。（目前最折中的model合計共train了81 Epoch）

3. Results and discussion

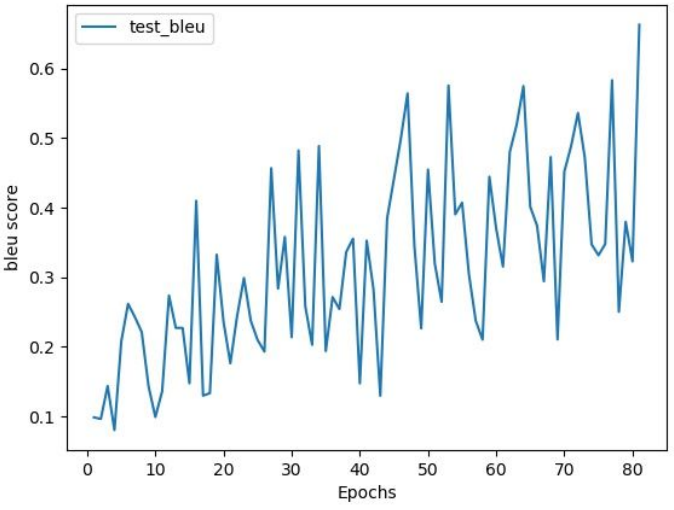
a. Loss and KL Loss

右圖的前77的Epoch是我用bleu來設定upper bound的結果，後面是用一整個Epoch都是同樣的KLD weight來train的
可以看到train_loss在很緩慢的下降，相比之下，KL_loss則是逐漸的增加，也算是慢慢放寬了對於encoder輸出的mean和variance要符合特定值的限制，使得model有比較高的自由去學著fit data
這某種程度上也算是符合當時設計upper bound的用意，就是不要讓KL loss急遽的上升，慢慢學，雖然沒有辦法完全水平或是變小，但也是不錯的了。



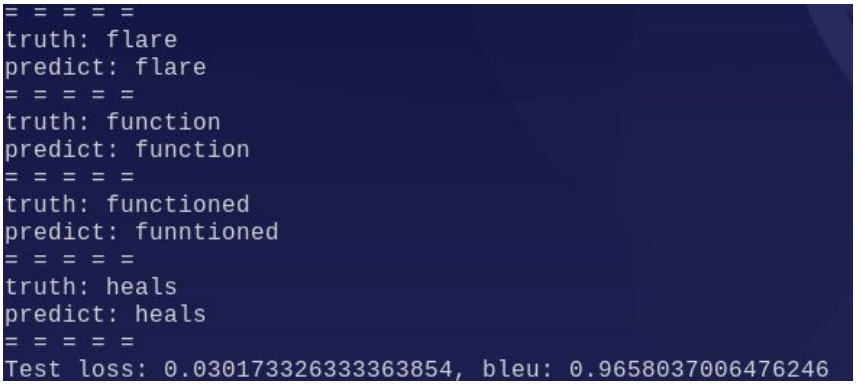
b. BLEU-4 score

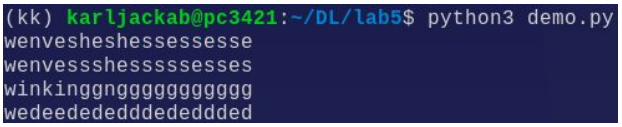
可以看到上升的非常曲折，且十分不穩定，如同筆者在作業deadline前的情緒一般，一波三折，潸然淚下
對於這個圖，我的解讀是model正在KLD loss以及decoder loss取得平衡，而由於逐步增加upper bound的方式，可以讓他雖然是很曲折，但還是有逐步上升的趨勢



c. Result Compare

右邊先看到的是test bleu有到1.0的model所產生的圖，可以看到雖然test的bleu十分優秀，但在生成字上面卻完全看不懂，而在這邊我也才發現，本來的model裡面，沒有特別去學End Sign的標注，都到ground truth的長度就cut掉，所以在這種test上面就會無限制的輸出，於是我再重新調整weight時也順便加入了這項訓練。





再來右邊這兩張，雖然test bleu沒有那麼好，但在生成詞上面卻也能有機會表現到這樣，由此可以看到KL loss的重要性，如果偏差太多的話是很難生成出像樣的詞出來的