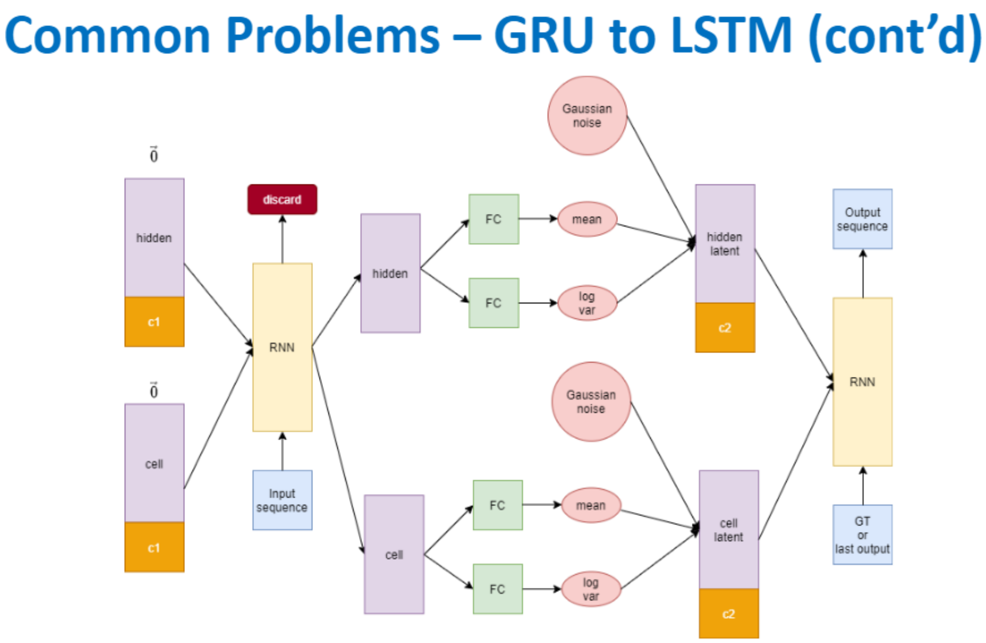
DLP Lab5
309553012 黃建洲

1. Introduction:

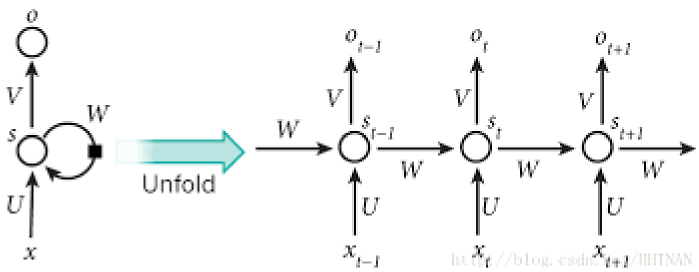
這次作業讓我們實作sequence-to-sequence的CVAE，並且所使用的RNN架構為LSTM。目標是訓練一組encoder和decoder，encoder會根據輸入的字串與condition來產生latent，latent送入decoder再逐字產生各字母的機率，並softmax回字串進行比對。最後利用bleu score和gaussian score來進行model的檢驗。

2. Derivation of CVAE

CVAE主要由兩個部分組成: Encoder和Decoder



參照SPEC所給的架構圖進行，其中RNN的所使用的為LSTM，架構圖如下



根據input, input condition和初始化的兩個sample，Encoder會分別產生兩組mean和logvar，並與一組隨機產生的高斯雜訊混合成兩組latent code。這兩組latent code分別再與input condition結合進入Decoder，得到output的熵值。

公式推導部分:

首先input x經過encoder產生機率分布p(z|x;c)作為latent code，基於Lstm的特性產生出兩個hidden，這兩個hidden分別利用一層FC layer取出mean和log variance, 與隨機取樣的gaussian分布合併計算如下

$$q_\phi\left(\mathbf{z}|\mathbf{x}^{(i)}\right) = \mathcal{N}\left(\mathbf{z}; \boldsymbol{\mu}^{(i)}, \boldsymbol{\sigma}^{2(i)}\mathbf{I}\right)$$

得到兩個latent，分別為先驗機率與後驗機率。以KL loss為優化目標就是為了讓這兩個機率分布盡量靠近，使encoder生成盡量相似的latent code。

兩個latent code進入decoder作為input z，用encoder得到p(z|x;c)，最後用生成的x與原本input的x計算中間的CrossEntropy Loss和KL divergence Loss用以做back propagation與update。

3. Implement detail:

Encoder:

```
class Encoder(nn.Module):
    def __init__(self, word_size=28, RNN_hidden_size=256, latent_size=32, num_condition=4, condition_embedding_size=8):
        print("Encoder Construction")
        super(Encoder, self).__init__()
        self.word_size = word_size
        self.RNN_hidden_size = RNN_hidden_size
        self.latent_size= latent_size
        self.num_condition = num_condition
        self.condition_embedding_size = condition_embedding_size

        self.condition_embedding = nn.Embedding(num_condition, condition_embedding_size)
        self.input_embedding = nn.Embedding(word_size, RNN_hidden_size)
        self.LSTM = nn.LSTM(RNN_hidden_size, RNN_hidden_size)
        self.mean_hidden = nn.Linear(RNN_hidden_size, latent_size)
        self.logvar_hidden = nn.Linear(RNN_hidden_size, latent_size)
        self.mean_cell = nn.Linear(RNN_hidden_size, latent_size)
        self.logvar_cell = nn.Linear(RNN_hidden_size, latent_size)

    def forward(self, input_x, input_c, init_hidden, init_cell):
        #c = torch.LongTensor([input_c]).to(device)
        #c = self.condition_embedding(c).view(1,1,-1)
        c = input_c
        hidden = torch.cat((init_hidden, c), dim=2)
        cell = torch.cat((init_cell, c), dim=2)

        x = self.input_embedding(input_x).view(-1,1,self.RNN_hidden_size)

        outputs, (output_hidden, output_cell) = self.LSTM(x, (hidden, cell))

        mean_hidden = self.mean_hidden(output_hidden)
        logvar_hidden = self.logvar_hidden(output_hidden)
        hidden_latent = self.sampling() * torch.exp(logvar_hidden / 2) + mean_hidden


        mean_cell = self.mean_cell(output_cell)
        logvar_cell = self.logvar_cell(output_cell)
        cell_latent = self.sampling() * torch.exp(logvar_cell / 2) + mean_cell

        return mean_hidden, logvar_hidden, hidden_latent, mean_cell, logvar_cell, cell_latent

    def initHidden(self):
        return torch.zeros(1,1, self.RNN_hidden_size - self.condition_embedding_size, device = device)

    def initCell(self):
        return torch.zeros(1,1, self.RNN_hidden_size - self.condition_embedding_size, device = device)

    def sampling(self):
        return torch.randn(self.latent_size).to(device)
```

將input word和input condition都利用指定步驟處理成所需求的tensor後，丟進LSTM產出output, hidden和cell，對hidden和cell分別利用一個fc layer產出mean和logvar後，分別與用sampling function產出的gaussian noise合併成latent code後回傳。

Decoder:(單一字母)

```python
class Decoder(nn.Module):
    def __init__(self, word_size = 28, RNN_hidden_size = 256, latent_size = 32, num_condition = 4, condition_embedding_size = 8):
        print("Decoder Construction")
        super(Decoder, self).__init__()
        self.word_size = word_size
        self.RNN_hidden_size = RNN_hidden_size
        self.latent_size = latent_size
        self.condition_embedding_size = condition_embedding_size

        self.condition_embedding = nn.Embedding(num_condition, condition_embedding_size)

        self.latentHiddenConvert = nn.Linear(latent_size + condition_embedding_size, RNN_hidden_size)
        self.latentCellConvert = nn.Linear(latent_size + condition_embedding_size, RNN_hidden_size)
        self.input_embedding = nn.Embedding(word_size, RNN_hidden_size)
        self.LSTM = nn.LSTM(RNN_hidden_size, RNN_hidden_size)
        self.fc = nn.Linear(RNN_hidden_size, word_size)

    def forward(self, input_x, input_c, latent_hidden, latent_cell, use_teacher_forcing = False):
        #c = torch.LongTensor([input_c]).to(device)
        #c = self.condition_embedding(c).view(1,1,-1)
        c = input_c
        latent_hidden = latent_hidden.view(1,1,-1)
        latent_cell = latent_cell.view(1,1,-1)

        hidden = self.hiddenProcessor(latent_hidden, c)
        cell = self.cellProcessor(latent_cell, c)

        x = self.input_embedding(input_x).view(1,1,self.RNN_hidden_size)

        outputs, (output_hidden, output_cell) = self.LSTM(x ,(hidden,cell))



        outputs = self.fc(outputs).view(-1, self.word_size)

        return outputs, output_hidden, output_cell


    def hiddenProcessor(self, latent_hidden, c):
        output = torch.cat((latent_hidden, c), dim=2)
        return self.latentHiddenConvert(output)

    def cellProcessor(self, latent_cell, c):
        output = torch.cat((latent_cell, c), dim=2)
        return self.latentCellConvert(output)
```

這邊的input x一次只代表一個字母(包含SOS和EOS)
從encoder接收兩個latent code後以及input condition和input word，首先將latent code與condition串接成Lstm的input，再利用LSTM取出output並回傳。這邊的output不會進行softmax，以利後面CrossEntropy計算方便。

整個單字的Decode過程:

```python
def decoder_process(train_data, decoder, input_x, input_c, latent_hidden, latent_cell, targetlength, use_teacher_forcing = False):
    sos_token = train_data.chardict.word2index['SOS']
    eos_token = train_data.chardict.word2index['EOS']
    outputs = []
    x = torch.LongTensor([sos_token]).to(device)

    for i in range(targetlength):
        x = x.detach()
        output, output_hidden, output_cell = decoder(x, input_c, latent_hidden, latent_cell)
        outputs.append(output)
        output_class = torch.max(torch.softmax(output,dim=1),1)[1]

        if output_class.item() == eos_token and use_teacher_forcing == False:
            break

        if use_teacher_forcing == True:
            x = input_x[i+1:i+2]
        else:
            x = output_class
    if len(outputs) != 0:
        outputs = torch.cat(outputs,dim=0)
    else:
        outputs = torch.FloatTensor([]).view(0,word_size).to(device)
    return outputs
```

一開始的輸入字母為SOS，丟進Decoder後會得到該字母的預測值，接著根據是否使用teacher forcing模式來決定下一個輸入字母的出處。若使用teacher forcing模式，輸入字母直接從dataset的對應位置來拿(ground truth)。若不使用，則輸入字母使用上一個迴圈的decoder output進行。
最後將所有output整併成一個陣列回傳。

Training:

```python
def runModel(encoder, decoder, train_data, test_data, epochs = 300, lr=0.007, KLweight = 0, teacher_forcing_ratio = 1):
    print("Start Training")
    encoder_optim = optim.SGD(encoder.parameters(), lr=lr)
    decoder_optim = optim.SGD(decoder.parameters(), lr=lr)
    criterion = nn.CrossEntropyLoss()
    curve_kl_loss = 0
    curve_loss = 0
    return_score = []
    return_kl_loss = []
    return_loss = []
    return_tfr = []
    return_klw = []
    for epoch in range(epochs):
        print(epoch)
        encoder.train()
        decoder.train()
        if epoch > 50:
          KLweight = 0.001 * ((epoch - 50))
        if KLweight > 1:
            KLweight = 1
        if epoch > 100:
          teacher_forcing_ratio = 1 - 0.001 * ((epoch - 100))
        if teacher_forcing_ratio < 0:
            teacher_forcing_ratio = 0
        for i in range(len(train_data)):
            data = train_data[i]
            inputs, c = data
            encoder_optim.zero_grad()
            decoder_optim.zero_grad()
            input_c1 = torch.LongTensor([c]).to(device)
            input_c1 = encoder.condition_embedding(input_c1).view(1,1,-1)
            input_c2 = torch.LongTensor([c]).to(device)
            input_c2 = encoder.condition_embedding(input_c2).view(1,1,-1)
            mean_hidden, logvar_hidden, latent_hidden, mean_cell, logvar_cell, latent_cell = encoder(inputs[1:].to(device), input_c1, encoder.initHidden(), encoder.initCell())



            use_teacher_forcing = True if random.random() < teacher_forcing_ratio else False

            input_length = inputs[1:].size(0)

            outputs = decoder_process(train_data, decoder, inputs.to(device), input_c2, latent_hidden, latent_cell, input_length, use_teacher_forcing = use_teacher_forcing)
            output_length = outputs.size(0)
            target = inputs[1:1 + output_length].to(device)
            loss = criterion(outputs, target)
```

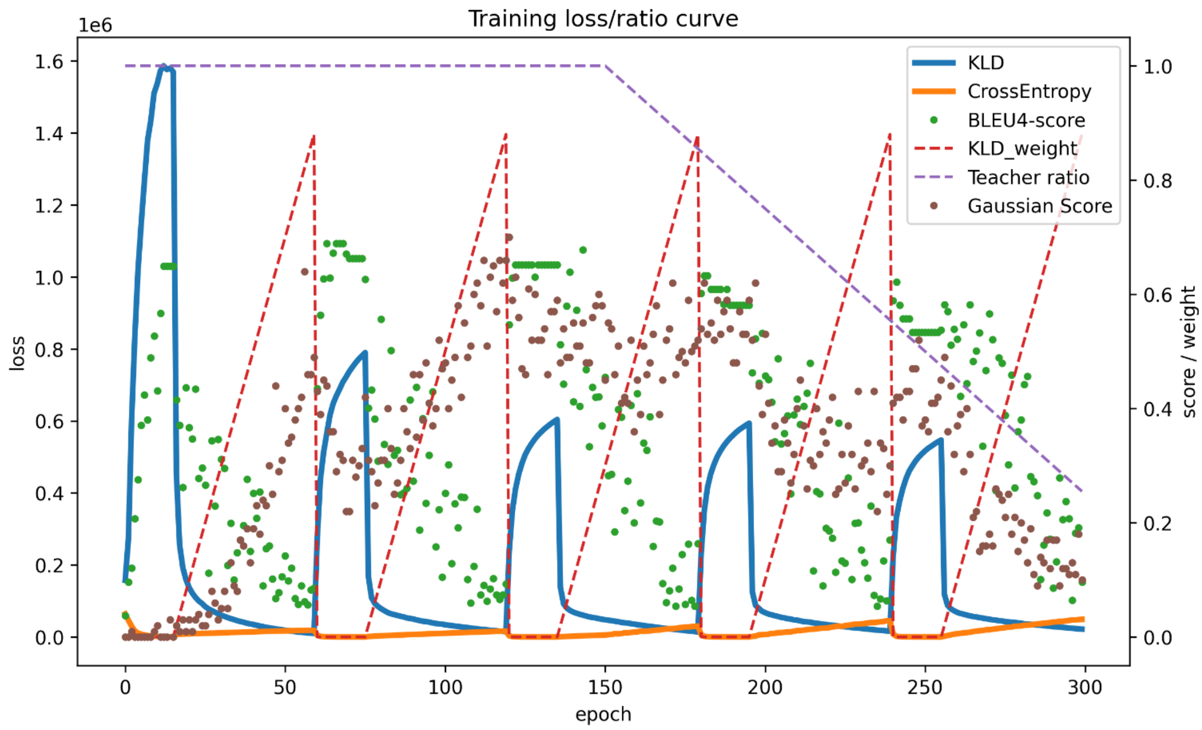
==

```python
            klLoss = Gaussian_KL_loss(mean_hidden, logvar_hidden, mean_cell, logvar_cell)
            total_loss = loss - (KLweight * klLoss)
            #loss.backward()
            total_loss.backward()
            encoder_optim.step()
            decoder_optim.step()

            curve_loss += loss.item()
            curve_kl_loss += klLoss.item()

            output_class = torch.max(torch.softmax(outputs, dim=1),1)[1]
            input_str = train_data.chardict.stringFromLongtensor(inputs, show_token=True)
            output_str = train_data.chardict.stringFromLongtensor(output_class, show_token=True)
            if i == 0:
                print(f"input: {input_str}, output: {output_str}, Loss: {loss}, KL_Loss: {klLoss}")
                #print(f"input: {input_str}, output: {output_str}, Loss: {loss}, KL_Loss: {klLoss}, mean_hidden: {mean_hidden}, lovgar_hidden: {logvar_hidden}, mean_cell: {mean_cel

        score = 0
        eval_score = testModel(encoder, decoder, test_data, train_data)
        score += sum(eval_score) / len(eval_score)

        print(f"bleu_score: {score}, total_loss: {curve_loss}, total_kl_loss: {curve_kl_loss}")
        save_model_by_score({'encoder':encoder, 'decoder':decoder}, score, os.path.join('.', "test"))

        return_score.append(score)
        return_kl_loss.append(curve_kl_loss)
        return_loss.append(curve_loss)
        return_tfr.append(teacher_forcing_ratio)
        return_klw.append(KLweight)

        curve_loss = 0
        curve_kl_loss = 0
    return return score, return kl loss, return loss, return tfr, return klw
```

KL weight和teacher forcing ratio我分別設定從50和100個epoch後會開始上升/下降，斜率為0.001
每一個epoch中，每一個迴圈都從訓練集內拿出一個單字，由於訓練集的資料是按照順序拜放的，condition則按照index % 4(種類的condition) 給予，包成tensor之後與單字一同丟進encoder產生latent code。將這個latent code與事先打包好的conditiontensor一同丟進去decoder，並由前述的過程產出整個單字的熵值。利用output算出CrossEntropy loss以及利用兩組mean和logvariance來算出KL Loss，再利用兩個loss值計算出total loss，接下來同其他神經網路的步驟進行back propagation和update。最後利用Nltk算出bleu score和作業提供的gaussian score，利用這兩項指標看是否要儲存model參數。

超參數的部分: KLweight和teacher forcing ratio前面已經提過，learning rate為0.007，epochs為300，其他都照spec進行設置。

4. Result and Discussion



我做了相當多的嘗試，前期的嘗試基本都沒有太好的結果，直到最後使用了週期較短的週期性KLW訓練才有不錯的結果。

就我觀察到的結果來說， 我認為在KL weight為0的時期將CrossEntropy Loss訓練到非常小是不利於整體的訓練的。以我前期的訓練而言，若我在開始時使用了50個epoch(每個epoch有4908個iterations)來以CrossEntropy為優化目標，那麼在加入KL Loss時會有兩個問題: 第一個問題是在該時間點的KL LOSS通常極大，改變的幅度相當可能影響到Model的良度。第二個問題則是KL Loss也難以在週期範圍內下降到讓gaussian score開始上升的幅度， 除非根據訓練狀況調整KL annealing的斜率，但那在訓練上就不太general了。

因此我將300個epoch分為5個週期，每個週期的前15個epoch純粹以CrossEntropy，而後以0.02為斜率將KL Loss加入，直到週期結束歸零。

最後由於load model會帶有一定的隨機性，不一定能呈現出最好的結果，因此附上觀測到最好的結果。其中Bleu score為0.67，並且Gaussian score為0.55

```
Epoch  144
tfr : 1, klw: 0.16
target_str: abandoned, outputs_str: abandoning
target_str: abetting, outputs_str: abet
target_str: begins, outputs_str: beging
target_str: expends, outputs_str: expend
target_str: sends, outputs_str: send
target_str: splitting, outputs_str: split
target_str: flare, outputs_str: flare
target_str: function, outputs_str: function
target_str: functioned, outputs_str: functions
target_str: heals, outputs_str: hearing
BLEU-4 score : 0.6777301563616674
Loss:  3402.155469345278
KL Loss:  60805.97105693817
Score:  0.6777301563616674
Gaussian Score: 0.55
```