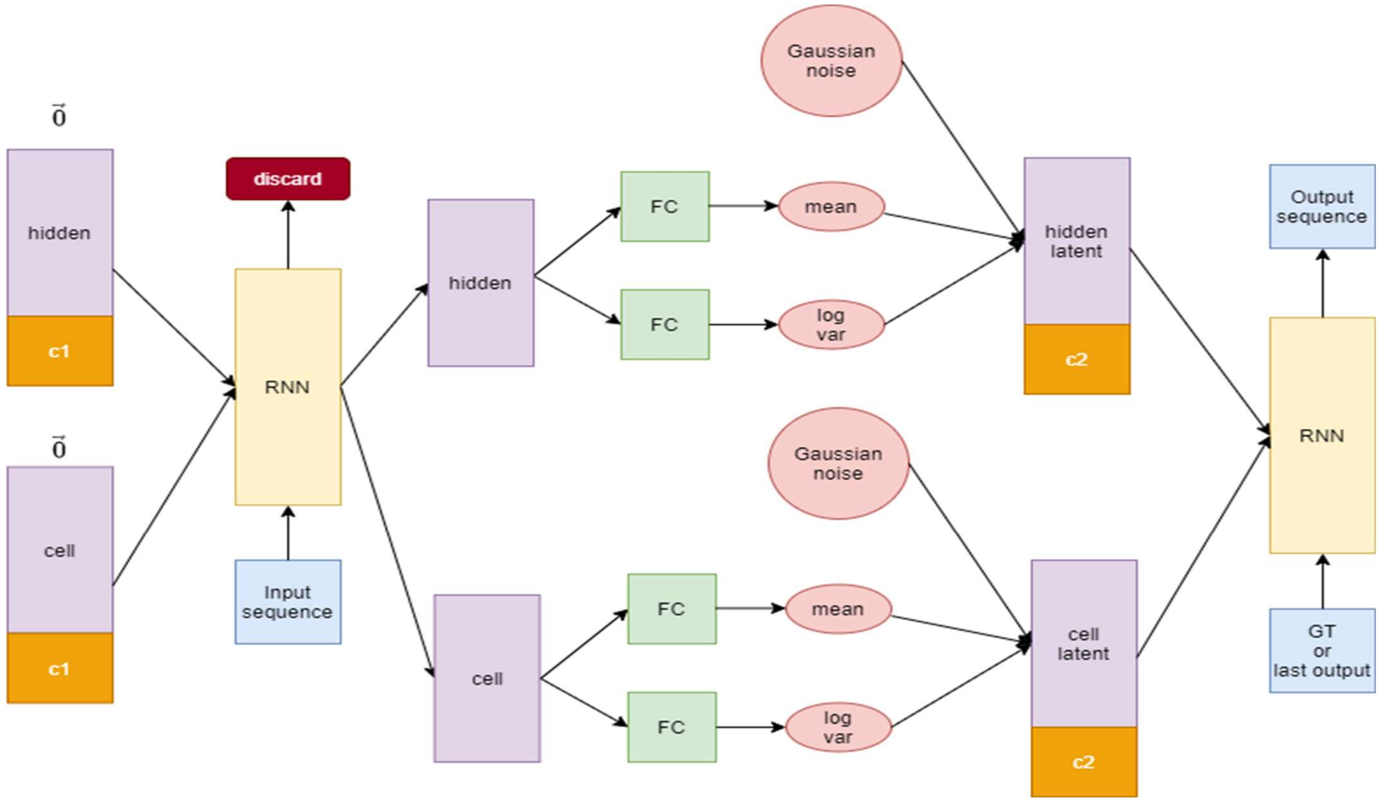
Report Spec

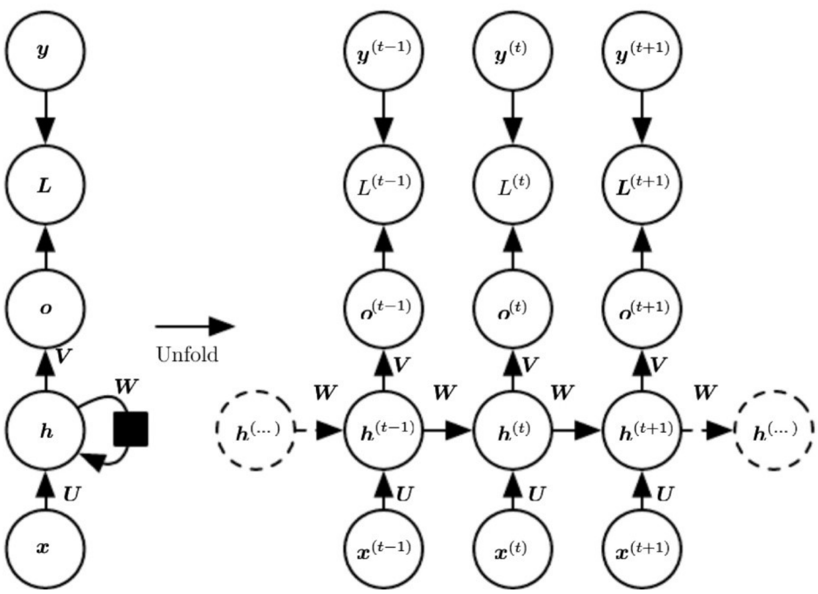
1. Introduction

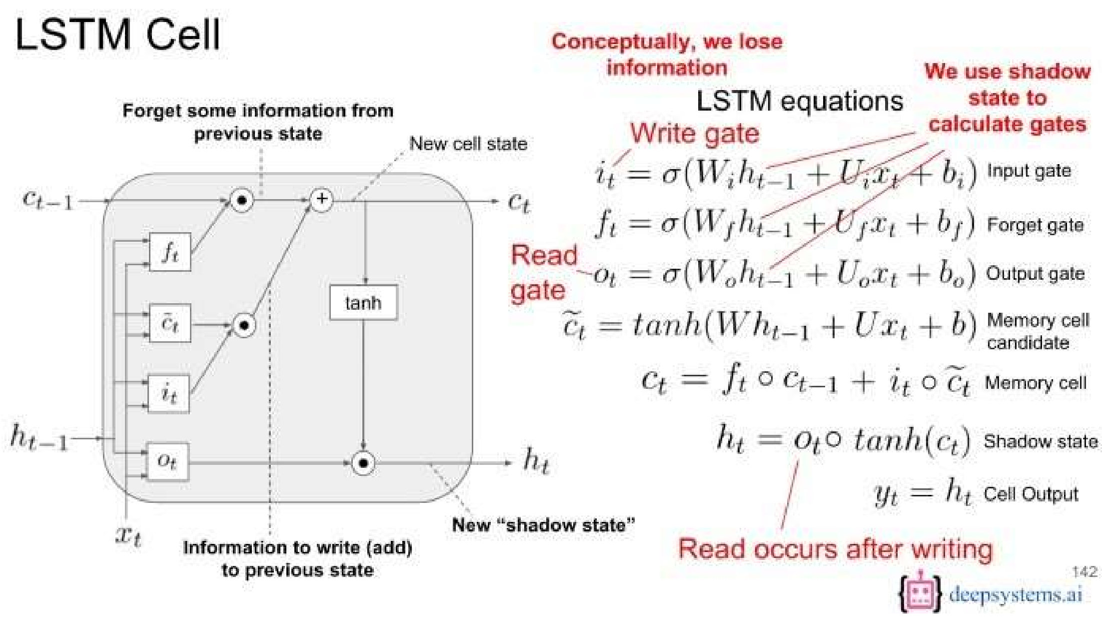
利用 CVAE 處理英文 word 時態轉換問題，其中本次注重學習的部分就是在 encoder 與 decoder 的中間部分利用 mean, var 與 Gaussian 取得 KLloss，相較於普通的 Autoencoder，其資料會更有關聯性一些(高斯分布)，而不是四散於整個 one hot vector。



2. Derivation of CVAE

其實主要就是 LSTM 與 Back-PropagationTHroughTime 的推導





3. Derivation of KL Divergence loss

The encoder distribution is $q(z|x) = \mathcal{N}(z|\mu(x), \Sigma(x))$ where $\Sigma = \text{diag}(\sigma_1^2, \ldots, \sigma_n^2)$.

The latent prior is given by $p(z) = \mathcal{N}(0, I)$.

Both are multivariate Gaussians of dimension n, for which in general the KL divergence is:

$$\mathfrak{D}_{\text{KL}}[p_1 \,||\, p_2] = \frac{1}{2}\left[\log\frac{|\Sigma_2|}{|\Sigma_1|} - n + \text{tr}\{\Sigma_2^{-1}\Sigma_1\} + (\mu_2 - \mu_1)^T\Sigma_2^{-1}(\mu_2 - \mu_1)\right]$$

where $p_1 = \mathcal{N}(\mu_1, \Sigma_1)$ and $p_2 = \mathcal{N}(\mu_2, \Sigma_2)$.

In the VAE case, $p_1 = q(z|x)$ and $p_2 = p(z)$, so $\mu_1 = \mu$, $\Sigma_1 = \Sigma$, $\mu_2 = \vec{0}$, $\Sigma_2 = I$. Thus:

$$\begin{aligned}
\mathfrak{D}_{\text{KL}}[q(z|x) \,||\, p(z)] &= \frac{1}{2}\left[\log\frac{|\Sigma_2|}{|\Sigma_1|} - n + \text{tr}\{\Sigma_2^{-1}\Sigma_1\} + (\mu_2 - \mu_1)^T\Sigma_2^{-1}(\mu_2 - \mu_1)\right] \\
&= \frac{1}{2}\left[\log\frac{|I|}{|\Sigma|} - n + \text{tr}\{I^{-1}\Sigma\} + (\vec{0} - \mu)^T I^{-1}(\vec{0} - \mu)\right] \\
&= \frac{1}{2}\left[-\log|\Sigma| - n + \text{tr}\{\Sigma\} + \mu^T\mu\right] \\
&= \frac{1}{2}\left[-\log\prod_i \sigma_i^2 - n + \sum_i \sigma_i^2 + \sum_i \mu_i^2\right] \\
&= \frac{1}{2}\left[-\sum_i \log\sigma_i^2 - n + \sum_i \sigma_i^2 + \sum_i \mu_i^2\right] \\
&= \frac{1}{2}\left[-\sum_i (\log\sigma_i^2 + 1) + \sum_i \sigma_i^2 + \sum_i \mu_i^2\right]
\end{aligned}$$

覺得寫得很清楚的一篇文章。

4. Implementation details

a. Dataloader

`def creat_char2idx_dict():`
`    s = {'SOS':0,'EOS':1}`
`    for i in range(26):`
`        s.setdefault(chr(i+97),i+2)`
`    return s`
 	
`def creat_idx2char_dict():`
`    s = {0:'SOS',1:'EOS'}`
`    for i in range(26):`
`        s.setdefault(i+2,chr(i+97))`
`    return s`
	1. 根據
	`{'SOS':0,'EOS':1,'a':2,'b':3 ... 'z':27}` 建立對應的 dictionary
	
	2. 根據
	`{0:'SOS',1:'EOS',2:'a',3:'b' ... 27:'z'}` 建立對應的 dictionary

`def word2idx(word, eos = True):`
`    s = []`
`    for i in word:`
`        s.append(char2idx_dict[i])`
`    if eos:`
`        s.append(char2idx_dict['EOS'])`
`    return torch.tensor(s).view(-1,1)`
 	
`def idx2word(idx):`
`    word = ""`
`    for i in idx:`
`        if i == 1: break`
`        char = idx2char_dict[i.item()]`
`        word += char`
`    return word`
	3. word 根據 dictionary 轉換成 tensor encoder
	
	4. tensor 根據 dictionary 轉回文字 主要用於 decoder

`train_list = getdatafromtxt(path,'train')` `test_list = comptestlist(getdatafromtxt(path,'test'))`	5. 將 txt 檔轉成 list 型態

`training_pair = tensorsFromPair(`
`    random.randint(0, len(train_list)-1), train_list)`
`|`
`input_tensor = training_pair[0]`
`target_tensor = training_pair[1]`
	6. prepare data 中最重要的部分
	這邊使用 idx 隨機的方法提取每次的 pair
	提取的資料像是這樣
	[[‘work’ , 0],[‘working’ , 2]]
	當然也有可能取到顛倒

B. encoder

<pre>class EncoderRNN(nn.Module):
    def __init__(self, input_size, hidden_size, condition_size):
        super(VAE.EncoderRNN, self).__init__()
 	
        self.hidden_size = hidden_size
        self.condition_size = condition_size
 	
        self.embedding = nn.Embedding(input_size, hidden_size)
        self.lstm = nn.LSTM(hidden_size, hidden_size)
 	
    def forward(self, input, hidden, cell):
        embed = self.embedding(input).view(1, 1, -1)
        output, (hidden, cell) = self.lstm(embed, (hidden, cell))
 	
        return output, hidden, cell
 	
    def initHidden(self):
        return torch.zeros(1, 1, self.hidden_size - self.condition_size, device = device)
 	
    def initCell(self):
        return torch.zeros(1, 1, self.hidden_size - self.condition_size, device = device)</pre>
	其實 encoder 部分沒有很複雜單純使用個 embedding 轉成 one hot vector 後，接下來就是 RNN 的工作了，RNN 的部分使用的是 LSTM 其中 hidden 部分更較容易受新 input 影像，cell 則相反。

C. decoder

<pre>class DecoderRNN(nn.Module):
    def __init__(self, input_size, hidden_size, condition_size):
        super(VAE.DecoderRNN, self).__init__()
        self.hidden_size = hidden_size
        self.embedding = nn.Embedding(input_size, hidden_size)
        self.lstm = nn.LSTM(hidden_size, hidden_size)
        self.out = nn.Linear(hidden_size, input_size)
        self.softmax = nn.LogSoftmax(dim=1)
 	
    def forward(self, input, hidden, cell):
        output = self.embedding(input).view(1, 1, -1)
        output = F.relu(output)
        output, (decoder_hidden, decoder_cell) = self.lstm(output, (hidden, cell))
        output = self.out(output[0])
        output = self.softmax(output)
        return output, decoder_hidden, decoder_cell
 	
    def initHidden(self):
        return torch.zeros(1, 1, self.hidden_size, device=device)</pre>
	decoder 部分其實與 GUR 整體差不多，改的只是 LSTM

D. Reparameterization_Trick

<pre>def Reparameterization_Trick(self, mean, logvar):
    std = torch.exp(logvar/2)
    eps = torch.randn_like(std)
    return mean + eps * std</pre>
	這部分其實也蠻好懂得，就整個資料從
	N(mean, var)中 sample 一個點
	Kl 訓練這部分主要是讓整個分布更有關聯性一些

E. Gaussian_generation

```
def gaussian_gen(self,maxlen):
    wordssss = []
    tense = torch.tensor([[0],[1],[2],[3]]).to(device)
    for n in range(100):
        word = []
        latent_h = torch.randn_like(torch.zeros(1, 1, 32)).to(device)
        latent_c = torch.randn_like(torch.zeros(1, 1, 32)).to(device)

        for tensor in tense:
            decoder_hidden = self.latent2decoder_h(torch.cat((latent_h, self.embedding_la_h(tensor).view(1, 1, -1)), dim = -1))
            decoder_cell = self.latent2decoder_c(torch.cat((latent_c, self.embedding_la_c(tensor).view(1, 1, -1)), dim = -1))
            decoder_input = torch.tensor([[SOS_token]], device=device)
            pred_idx = torch.tensor([]).to(device)

            for d in range(maxlen):
                decoder_output, decoder_hidden, decoder_cell = self.decoder(decoder_input, decoder_hidden, decoder_cell)
                topv, topi = decoder_output.topk(1)
                decoder_input = topi.squeeze().detach()  # detach from history as input
                pred_idx = torch.cat((pred_idx, decoder_input.view(1, -1)), 0)

                if decoder_input.item() == EOS_token:
                    break
            word.append(idx2word(pred_idx))
#              print(idx2word(pred_idx))
        wordssss.append(word)

    return wordssss
```

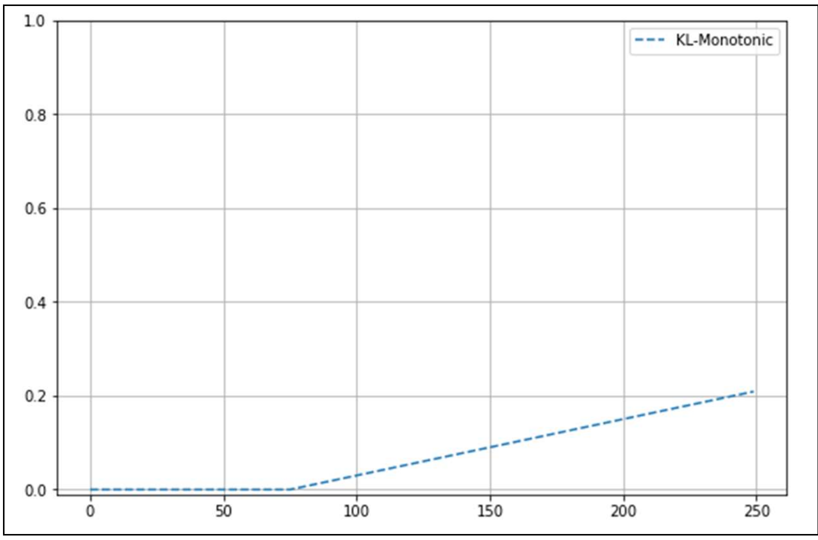
算是要理解整個 latent 才寫得出來的東西，首先有三個迴圈

1. 總共 100 次的迴圈
2. tensor 4 個型態 input 的迴圈
3. 最後再根據 decode model 的部分 依照字母 rnn 的訓練

其中原本 decoder_hidden 與 cell 的部分，原本是用 encoder 得到的 mean var 當作高斯函數的 smaple，現在直接取 random 高斯函數的隨機值

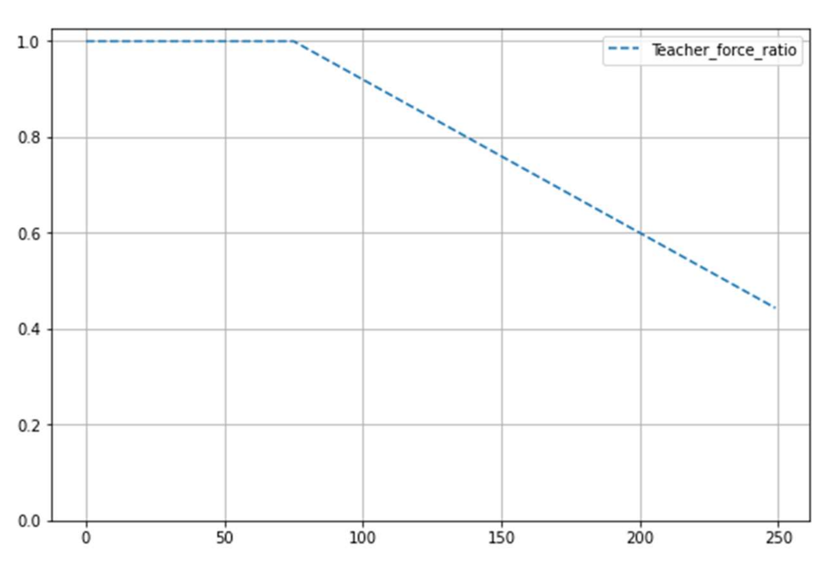
7. Results and discussion

KL weight 的部分 ：



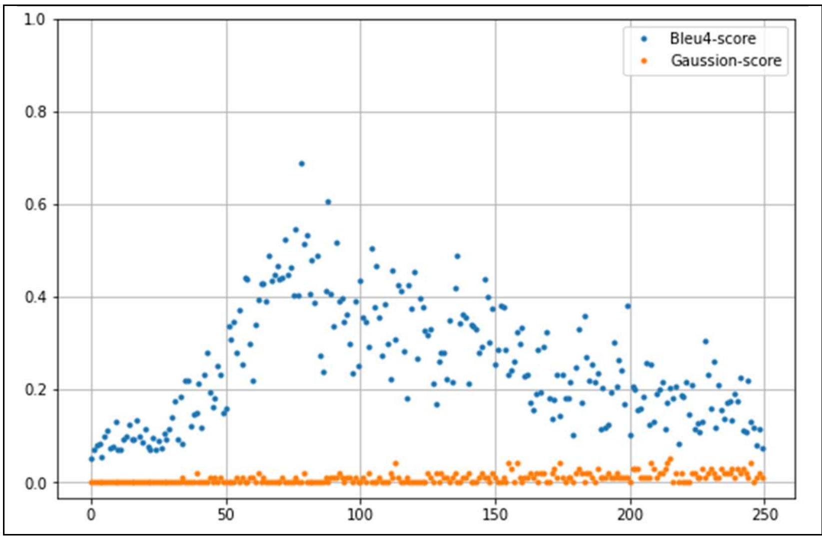
這邊我使用 KL monotonic 來當作最後的 KL weight 如右圖。在前面 30000 步，weight 為 0，目的是為了先著重訓練 celoss 的部分，若 weight 太高 KL 整個 Bleu score 會上不去

teacher Forcing ratio 的部分:



teacher forcing 的部分，在最後沒有降到最低而是降到 0.5 附近，是因為 KL weight 提高了，若 teacher forcing ratio，降太低的話，整體 Bleu 效果會非常不好，

Score 的部分



當 weight 給為 0 時可以看到對於 bleu score 的訓練成果是非常好的，但當開始給出 KL weight 後，整體開始下降，且 Gaussion 開始提升