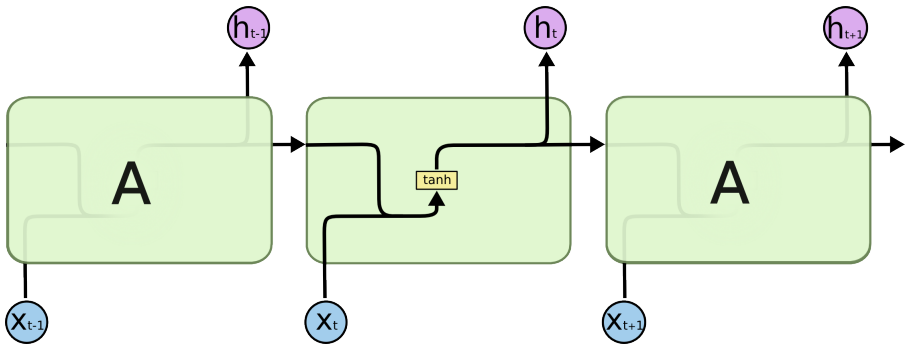
1. Introduction (5%)

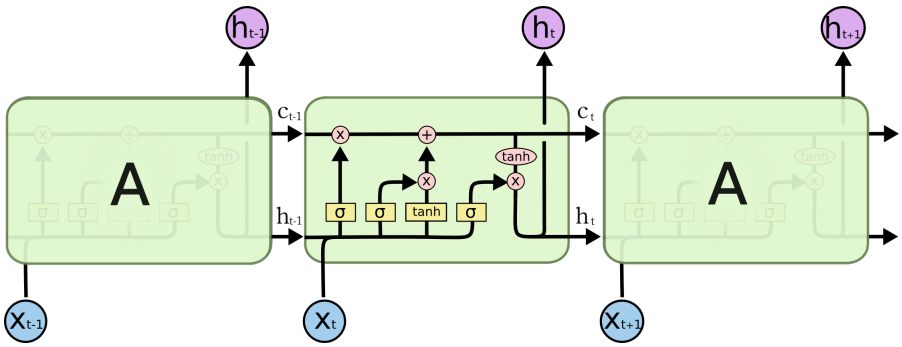
> 本次實驗希望透過 Conditional variational auto-encoder 搭配 LSTM 模型來做到時態轉換 (tense conversion) 與隨機四種時態的字彙生成 (text generation)。
>
> 在 training 的時候，會給 encoder 與 decoder 同樣的 tense condition （亦即 input condition = target condition）。首先 encoder 會透過 char by char 的方式來吃一個 word（因為 LSTM 這類模型具有記憶機制），再將輸出的 hidden 與 cell 透過 reparametrization trick 分別轉換成 latent code 餵給 decoder，而 decoder 的第一個 input 為 [SOS]，之後則視情況做 teacher forcing 來看要餵正確的 input character 還是 decoder 上一次的 output 給 decoder，直到 decoder output 出 [EOS]，同時也會調整 KL weight 來調整看當下要專注於調降 reconstruction loss 還是 regularization loss 上。
>
> 在做 testing 的時態轉換時，會給 encoder 與 decoder 不同的 tense condition （亦即 input condition ≠ target condition）。例如說輸入的 input condition 為 simple present、input word 為 access、target condition 為 present progressive，那麼將會希望 decoder 可以 output 的 word 為 accessing。這邊的轉換方式跟 training 很像，差別就在於這次多了 target tense condition 且不需要做 teacher forcing。
>
> 在做 testing 的字彙生成時，只需要用到 decoder。首先會從 Gaussian distribution N(0, I) (因為我們假設 latent code ~ N(0,I)) 抽出一個 latent code，接著再將這個 latent code 餵給 decoder，target condition 分別為 simple present、third person、present progressive、simple past，希望 decoder 可以根據該 latent code 隨機生成某個 word 的四種時態。
>
> 另外稍微介紹一下 RNN、GRU、LSTM 模型，因為實作 encoder 與 decoder 的本體時將會需要使用到他們。
>
> - RNN
>
> > $$h_t = \tanh\left(W_{ih}x_t + b_{ih} + W_{hh}h_{(t-1)} + b_{hh}\right)$$
> >
> > 注意這邊的 $W_{ih}, b_{ih}, W_{hh}, b_{hh}$ 與 t 無關，每個時間點都共用同一個參數。
> >
> > 
> >
>
> - LSTM

$$
\begin{aligned}
\text{Forget gate } f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\
\text{Input gate } i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\
\text{Output gate } o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\
\text{New memory cell content } \tilde{c}_t &= \tanh\left(W_{ic}x_t + b_{ic} + W_{hc}h_{t-1} + b_{hc}\right) \\
\text{Current memory cell state } c_t &= f_t \otimes c_{t-1} + i_t \otimes \tilde{c}_t \\
\text{Current hidden state } h_t &= o_t \otimes \tanh\left(c_t\right)
\end{aligned}
$$

Forget gate f_t 決定要留下多少比例的 previous memory cell c_{t-1} 資訊給 current memory cell state c_t，

Input gate i_t 決定有多少比例的 new memory cell content $\tilde{c}_t$ 要寫在 current memory cell state c_t，

Output gate o_i 決定要讀多少比例的 current memory cell state c_t 資訊來算 current hidden state h_t。

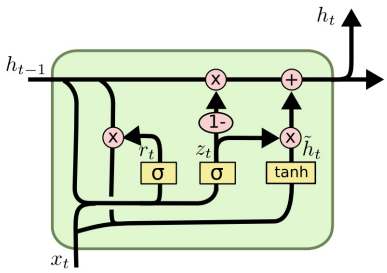


- GRU

$$
\begin{aligned}
\text{Update gate } u_t &= \sigma(W_{iu}x_t + b_{iu} + W_{hu}h_{t-1} + b_{hu}) \\
\text{Reset gate } r_t &= \sigma(W_{ir}x_t + b_{ir} + W_{hr}h_{t-1} + b_{hr}) \\
\text{New hidden state content } \tilde{h}_t &= \tanh\left(W_{ih}x_t + b_{ih} + r_t \otimes (W_{hh}h_{t-1} + b_{hh})\right) \\
\text{Current hidden state } h_t &= (1 - u_t) \otimes h_{t-1} + u_t \otimes \tilde{h}_t
\end{aligned}
$$

Update gate u_t 決定要分配多少比例的 previous hidden h_{t-1} 與 new hidden state content $\tilde{h}_t$ 來產出 current hidden state h_t

Reset gate r_t 決定要保留多少比例的 previous hidden h_{t-1} 資訊來計算 new hidden state content $\tilde{h}_t$



reference

[RNN 進化了 - LSTM 與 GRU]

> [Natural Language Processing with Deep Learning CS224N/Ling28 Lecture 7: Vanishing Gradients and Fancy RNNs]
>
> [Understanding LSTM Networks]
>
> [torch.nn.RNN]
>
> [torch.nn.LSTM]
>
> [torch.nn.GRU]

2. Derivation of CVAE (5%)

> $$
> \begin{aligned}
> & p(X, Z; \theta) = p(X; \theta) p(Z|X; \theta) \\
> \Rightarrow & \log p(X, Z; \theta) = \log p(X; \theta) + \log p(Z|X; \theta) \\
> \Rightarrow & \log p(X; \theta) = \log p(X, Z; \theta) - \log p(Z|X; \theta) \\
> \Rightarrow & \text{introduce an arbitrary distribution } q(Z) \text{ on both sizeds and integrate over } Z \\
> & \int q(Z) \log p(X; \theta) dZ = \int q(Z) \log p(X, Z; \theta) dZ - \int q(Z) \log p(Z|X; \theta) dZ \\
> & \qquad = \left(\int q(Z) \log p(X, Z; \theta) dZ - \int q(Z) \log q(Z) dZ \right) \\
> & \qquad + \left(\int q(Z) \log q(Z) dZ - \int q(Z) \log p(Z|X; \theta) dZ \right) \\
> & \qquad = L(X, q, \theta) + KL(q(Z) || p(Z|X; \theta)) \\
> & \text{note that the first term doesn't change,} \\
> & \int q(Z) \log p(X; \theta) dZ = \log p(X; \theta) \int q(Z) dZ = \log p(X; \theta) \because p(X; \theta) \text{ is irrelevant to} \\
> \Rightarrow & \log p(X; \theta) = L(X, q, \theta) + KL(q(Z) || p(Z|X; \theta)), \\
> & \text{where} \begin{cases} L(X, q, \theta) = \int q(Z) \log p(X, Z; \theta) dZ - \int q(Z) \log q(Z) dZ \\ \qquad = E_{Z \sim q(Z)} [\log p(X, Z; \theta) dZ] - E_{Z \sim q(Z)} \log q(Z) \\ \qquad = E_{Z \sim q(Z)} \log p(X|Z; \theta) + E_{Z \sim q(Z)} \log p(Z; \theta) - E_{Z \sim q(Z} \\ KL(q(Z) || p(Z|X; \theta)) = \int q(Z) \log \dfrac{q(Z)}{p(Z|X; \theta)} dZ \end{cases} \\
> & \text{note that KL divergence is non-negative, hence } \log p(X; \theta) \geq L(X, q, \theta), \\
> & \text{we can increase } \log p(X; \theta) \text{ by increasing } L(X, q, \theta) \\
> \Rightarrow & \text{note that the equality holds for any choice of } q(Z), \\
> & \text{we can introduce a distribution } q(Z|X; \theta') \\
> & \text{modeled by another neural network with parameter } \theta' \text{ to obtain,} \\
> & L(X, q, \theta) = E_{Z \sim q(Z|X; \theta')} \log p(X|Z; \theta) + E_{Z \sim q(Z|X; \theta')} \log p(Z) - E_{Z \sim q(Z|X; \theta')} \log q(Z|X; \\
> & \qquad = E_{Z \sim q(Z|X; \theta')} \log p(X|Z; \theta) - KL(q(Z|X; \theta') || p(Z))
> \end{aligned}
> $$

3. Derivation of KL Divergence loss (5%)

> KL divergence 為整個 variational auto-encoder 的 lower bound 的一部份，為了有宏觀的視野，以下將會先從 cross entropy 講起，接著才會講到 KL divergence。
>
> 根據官方 document 所寫 [torch.Tensor.backward computes the gradient of current tensor w.r.t. graph leaves.]，並會透過 optimizer.step() 根據算出的 gradient 使 object function 往可以變"更小"的方向去更新 $(\text{weight} = \text{weight} - \text{lr} * \Delta\text{weight})$，variational auto encoder 則希望能夠"最大"化 lower bound，即

$\max E_{Z\sim q(Z|X;\theta')}(\log p(X|Z;\theta) - \mathrm{KL}(q(Z|X;\theta')||p(Z)))$，而為了配合 backward 的特性，實作 求最佳 lower bound 時會將這個目標函數加上負號，變成 $\min -E_{Z\sim q(Z|X;\theta')}(\log p(X|Z;\theta) - \mathrm{KL}(q(Z|X;\theta')||p(Z)))$。

針對前項 reconstruction term 來說， $E_{Z\sim q(Z|X;\theta')} = -H(q(Z|X;\theta'), p(X|Z;\theta))$ ，可以透過 nn.CrossEntropyLoss 去實作出 $H(q(Z|X;\theta'), p(X|Z;\theta)$，這邊稍微比較一下 cross entropy：

general cross entropy vs. binary cross entropy

$$\text{general cross entropy } H(p,q) = -\sum_{x=1}^{n}[p(x)\log q(x)]$$

$$\text{binary cross entropy } H(p,q) = -\sum_{x=1}^{2}[p(x)\log q(x)] = [p(x)\log q(x) + (1-p(x))\log(1$$

torch.nn.CrossEntropyLoss vs. torch.nn.functional.binary_cross_entropy

$$\mathrm{nn.CrossEntropyLoss}()(\text{vector } x, \text{scalar label } c)$$
$$= -\log\frac{\exp x[c]}{\sum_j \exp(x[j])} = -x[c] + \log\left(\sum_j \exp(x[j])\right)$$

$$\mathrm{F.binary_cross_entropy}(\text{vector } x_1, \text{vector } x_2, \text{reduction} = \text{'mean'}))$$
$$= \text{mean}\left(\begin{bmatrix} l_1 \\ l_2 \\ \vdots \\ l_N \end{bmatrix}\right), \text{ where } l_i = -w_i[x_{2,i} * \log x_{1,i} + (1-x_{2,i}) * \log(1-x_{1,i})]$$

針對後項 regularization term 來說，由於假設 $p(Z) := N(0,I), q(Z|X;\theta') = N(\mu_{\theta'}(x), \Sigma_{\theta'}(x))$，為了簡化運算符號，以下將會用 μ, Σ 代表 $\mu_{\theta'}(x), \Sigma_{\theta'}(x)$，**將推出** $\mathrm{KL}(q(Z|X;\theta')||p(Z)))$ **可以寫成** μ, Σ **的函數**：

$$
\begin{aligned}
\mathrm{KL}(q(Z|X;\theta')||p(Z))) =& \mathrm{KL}(N(\mu,\sigma)||N(0,I)) \\
=& \int \frac{1}{\sqrt{(2\pi)^N|\Sigma|}} e^{-\frac{1}{2}(X-\mu)^T\Sigma^{-1}(X-\mu)} \left(\log \frac{1}{\sqrt{(2\pi)^N|\Sigma|}} e^{-\frac{1}{2}(X-\mu)^T\Sigma^{-} \right. \\
=& \int \frac{1}{\sqrt{(2\pi)^N|\Sigma|}} e^{-\frac{1}{2}(X-\mu)^T\Sigma^{-1}(X-\mu)} \log \left(\frac{\frac{1}{\sqrt{(2\pi)^N|\Sigma|}} e^{-\frac{1}{2}(X-\mu)^T\Sigma^{-1}(X}}{\frac{1}{\sqrt{(2\pi)^N}} e^{-\frac{1}{2}X^TX}} \right. \\
=& \int \frac{1}{\sqrt{(2\pi)^N|\Sigma|}} e^{-\frac{1}{2}(X-\mu)^T\Sigma^{-1}(X-\mu)} \left(\log \frac{1}{\sqrt{|\Sigma|}} + \log e^{-\frac{1}{2}(X-\mu)^T\Sigma^{-} \right. \\
=& E_{X\sim q(Z|X;\theta')} \left(-\frac{1}{2}\log|\Sigma| - \frac{1}{2}(X-\mu)^T\Sigma^{-1}(X-\mu) + \frac{1}{2}X^TX \right) \\
=& -\frac{1}{2}\log|\Sigma| + E_{X\sim q(Z|X;\theta')} \left(-\frac{1}{2}\mathrm{tr}\left(\Sigma^{-1}(X-\mu)(X-\mu)^T\right) + \frac{1}{2} \right. \\
=& -\frac{1}{2}\log|\Sigma| - \frac{1}{2}\mathrm{tr}\left(\Sigma^{-1}E_{X\sim q(Z|X;\theta')}[(X-\mu)(X-\mu)^T]\right) + \frac{1}{2}E_{X} \\
=& -\frac{1}{2}\log|\Sigma| - \frac{1}{2}\mathrm{tr}\left(\Sigma^{-1}\Sigma\right) + \frac{1}{2}(\mathrm{tr}\left(\Sigma\right) + \mu^T\mu) \\
=& -\frac{1}{2}\log|\Sigma| - \frac{1}{2}N + \frac{1}{2}(\mathrm{tr}\left(\Sigma\right) + \mu^T\mu) \\
=& \frac{1}{2}\left[-\log\prod_i^N \sigma_i^2 - N + \sum_i^N \sigma_i^2 + \sum_i^N \mu_i^2 \right] \\
=& \frac{1}{2}\left[-\sum_i^N \log\sigma_i^2 - N + \sum_i^N \sigma_i^2 + \sum_i^N \mu_i^2 \right] \\
=& \frac{1}{2}\sum_i^N \left[-\log\sigma_i^2 - 1 + \sigma_i^2 + \mu_i^2 \right]
\end{aligned}
$$

因此最終lower bound 在實作時可以直接如下算出

$$
\begin{aligned}
& \min -E_{Z\sim q(Z|X;\theta')}(\log p(X|Z;\theta) - \mathrm{KL}(q(Z|X;\theta')||p(Z))) \\
=& \min -\left(-H(q(Z|X;\theta'), p(X|Z;\theta)) - \frac{1}{2}\sum_i^N [-\log\sigma_i^2 - 1 + \sigma_i^2 + \mu_i^2] \right) \\
=& \min \left(H(q(Z|X;\theta'), p(X|Z;\theta)) + \frac{1}{2}\sum_i^N [-\log\sigma_i^2 - 1 + \sigma_i^2 + \mu_i^2] \right)
\end{aligned}
$$

```
# loss function = KL weight*KLD + BCE, where BCE =
nn.CrossEntropyLoss()
BCE = nn.CrossEntropyLoss(reduction='sum')(output_word_dist[1:],
input_word[1:]) #ignore 'SOS'
KLD_hidden = -0.5 * torch.sum(1 + logvar_hidden - mean_hidden.pow(2) -
logvar_hidden.exp())
KLD_cell = -0.5 * torch.sum(1 + logvar_cell - mean_cell.pow(2) -
logvar_cell.exp())
KLD = (KLD_hidden + KLD_cell)/2
L = BCE + KLD_weight*KLD
```

reference

> [Auto-Encoding Variational Bayes]

> [KL divergence between two univariate Gaussians]
>
> [KL divergence between two multivariate Gaussians]
>
> [Deriving the KL divergence loss for VAEs]
>
> [The Matrix Cookbook Section 8.2]
>
> [Derivations for Linear Algebra and Optimization p.13]

4. Implementation details (15%)
   - Describe how you implement your model (encoder, decoder, reparameterization trick, dataloader, etc.). Note that you must prove that your text generation is produced by Gaussian noise or you will get no point at this part. (paste/screenshot your code)

> - Data Loader
>
> 首先先定義一個 dictionary 決定該如何轉換 string 到 digit：
>
> ```
> class DataTransformDict():
> def __init__(self):
>  self.tense2num = {'sp': 0, 'tp': 1, 'pg': 2, 'p': 3}
>  self.alpha2num = {'SOS': 0, 'EOS': 1, 'a': 2, 'b': 3, 'c': 4,
> 'd': 5, 'e': 6, 'f': 7, 'g': 8, 'h': 9, 'i': 10, 'j': 11, 'k': 12,
> 'l': 13, 'm': 14, 'n': 15, 'o': 16, 'p': 17, 'q': 18, 'r': 19, 's':
> 20, 't': 21, 'u': 22, 'v': 23, 'w': 24, 'x': 25, 'y': 26, 'z': 27}
>  self.num2alpha = dict((v,k) for k,v in self.alpha2num.items())
> #exchange key and value
>  self.num2tense = dict((v,k) for k,v in self.tense2num.items())
> #exchange key and value
> ```
>
> 再來定義該如何處理 dataset，希望最終能將 string 透過 dictionary 的定義來轉換成 tensor，轉換步驟都寫在程式的註解上了，簡單來說就是將每個 word 轉換成 tensor 並加上在字首與字尾分別加上 [SOS] 與 [EOS] 的 token，最終 training dataset output 出來的東西將會是 (word tensor, condition tensor) 的 pair，testing dataset output 出來的東西將會是 ((input word tensor, input condition tensor), (target word tensor, target condition tensor)) 的兩個 pair。
>
> ```
> class TenseTransformDataset(Dataset):
> def __init__(self, path, mode):
>  self.path = path
>  self.mode = mode
>  self.words_list = []
>  self.tenses_list = []
>  if mode == 'train':
>  words_list = np.loadtxt(path, dtype=np.str)
>  self.tenses_list = ['sp', 'tp', 'pg', 'p']*len(words_list)
>  # simple present(sp), third person(tp), present progressive(pg),
> simple past(p); tenses_list: ['sp', 'tp', 'pg', 'p', 'sp', 'tp',
> 'pg', 'p', ...]
>  self.words_list = words_list.flatten().tolist() #words_list:
> ['abandon', 'abandons', 'abandoning', 'abandoned', 'abet', 'abets',
> 'abetting', 'abetted', ...
> ```

```
    elif mode == 'test':
        self.tenses_list = [['sp','p'], ['sp','pg'], ['sp','tp'], ['sp','tp'], ['p','tp'], ['sp','pg'], ['p','sp'], ['pg','sp'], ['pg','p'], ['pg','tp']]
        self.words_list = np.loadtxt(path, dtype=np.str).tolist()
def __len__(self): #return the size of dataset
    return len(self.words_list)
def __getitem__(self, index): #return processed index-th words and tenses
    typeTrans = DataTransformDict()
    alpha2num = typeTrans.alpha2num
    tense2num = typeTrans.tense2num
    if self.mode == 'train':
        word = self.words_list[index] #word: 'abandon'
        tense = self.tenses_list[index] #tense: 'sp'
        vector = [alpha2num[char] for char in word] #vector :[2, 3, 2, 15, 5, 16, 15]
        vector.insert(0, alpha2num['SOS'])
        vector.append(alpha2num['EOS']) #vector: [0, 2, 3, 2, 15, 5, 16, 15, 1]
        #vector = vector + [1]*(20 - len(vector)) #padding
        pair = (torch.tensor(vector, dtype=torch.long), torch.tensor(tense2num[tense], dtype=torch.long)) #pair: (tensor([ 0,  2,  3,  2, 15,  5, 16, 15,  1]), tensor(0))
        return pair
    elif self.mode == 'test':
        pair_list = []
        for word, tense in zip(self.words_list[index], self.tenses_list[index]): #word: 'abandon', tense: 'sp', self.words_list[index]: ['abandon', 'abandoned'], self.tenses_list[index]: ['sp', 'p']
            # convert word to vector
            vector = [alpha2num[char] for char in word] #vector :[2, 3, 2, 15, 5, 16, 15]
            vector.insert(0, alpha2num['SOS'])
            vector.append(alpha2num['EOS']) #vector: [0, 2, 3, 2, 15, 5, 16, 15, 1]

            # append vector version's (word, tense) pair to pair_list
            pair = ( torch.tensor(vector,dtype=torch.long), torch.tensor(tense2num[tense],dtype=torch.long) ) #pair: (tensor([ 0,  2,  3,  2, 15,  5, 16, 15,  1]), tensor(0))
            pair_list.append(pair)
        return tuple(pair_list) #tuple(pair_list): ((tensor([ 0,  2,  3,  2, 15,  5, 16, 15,  1]), tensor(0)), (tensor([ 0,  2,  3,  2, 15,  5, 16, 15,  6,  5,  1]), tensor(3)))
```

- Encoder

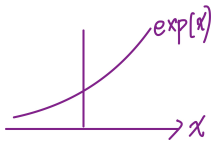
> condition_embedding layer 會將 input_condition 根據 4(condition_size) 個 condition 的資訊轉換成 8(condition_embedding_size) 維度的 tensor，char_embedding layer 則會將 input_word 根據 28(vocab_size) 個 character 的資訊轉換成 256(hidden_size) 維度的 tensor；而 forward function 會輸入 (1, 1, 256) 維度的 cell state 與 hidden state，和已轉換成 (1, 1, 256) 維度的 embedded input character，這邊的 cell state 與 hidden state 可能是由 (1, 1, 256-8) 維度的 init hidden state 與 init cell state 去 concate (1, 1, 8) 維度的 input condition 而來，也可能是直接由 encoder 輸出的維度為 (1, 1, 256) 的 cell state 或 hidden state 而來，接著透過 LSTM 輸出出新的 (1, 1, 256) 維度的 cell state 與 hidden state，和 (1, 1, 256) 維度的 output，不過這個 output 沒有任何用處，將會被丟掉 (discard)。
>
> ```
> class Encoder(nn.Module):
>  def __init__(self, vocab_size, condition_size, condition_embedding_size, hidden_size):
>  super(CVAE.Encoder, self).__init__()
>  self.condition_embedding = nn.Embedding(condition_size, condition_embedding_size)
>  self.char_embedding = nn.Embedding(vocab_size, hidden_size)
>  # parameters: the size of vocabulary, how many dimensions we want the char tensor to transfer to
>  self.lstm = nn.LSTM(hidden_size, hidden_size) #parameters: the number of expected features in the input x, the number of features in the hidden state h
>  def forward(self, input_char_embedded, hidden_state, cell_state):
>  _, (hidden_state, cell_state) = self.lstm(input_char_embedded, (hidden_state, cell_state)) #discard encoder's output
>  return _, hidden_state, cell_state
> ```

- Decoder

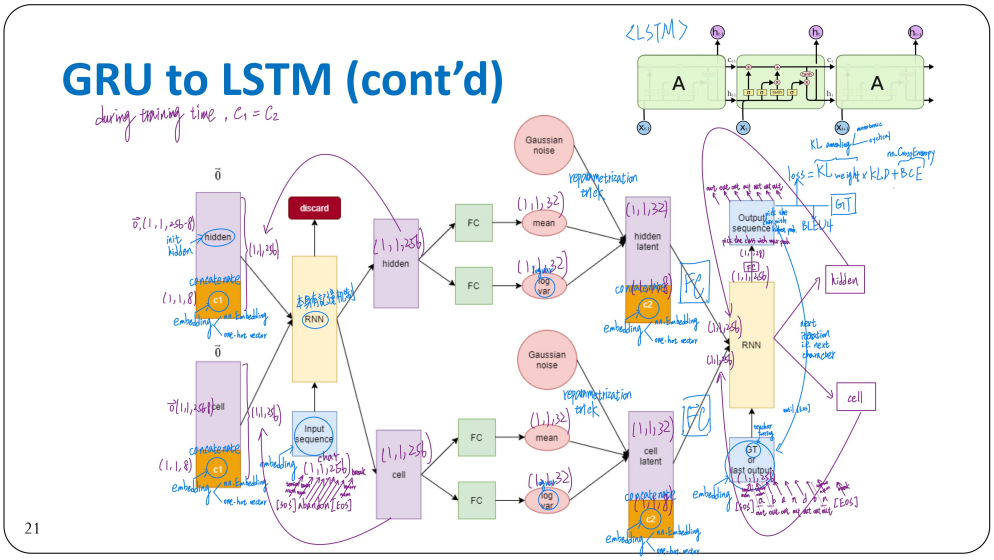
> condition_embedding layer 會將 input_condition 根據 4(condition_size) 個 condition 的資訊轉換成 8(condition_embedding_size) 維度的 tensor，char_embedding layer 則會將 input_word 根據 28(vocab_size) 個 character 的資訊轉換成 256(hidden_size) 維度的 tensor，兩個 LatentCon2hidden layer 會將 latent concat condition 得到的 40 維度的 tensor 轉換成 256 維度的 tensor，hidden2vocab layer 會將 256 維度的 tensor 轉換成 28 維度的 tensor；而 forward function 會輸入 (1, 1, 256) 維度的 cell state 與 hidden state，和已轉換成 (1, 1, 256) 維度的 embedded input character，這邊的 cell state 與 hidden state 是透過 (1, 1, 32) 維度的 latent code 去 concate (1, 1, 8) 維度的 target condition 再經過 fully connected layer 轉換成 (1, 1, 256) 維度而來，其中 (1, 1, 32) 維度的 latent code 則可能是由 encoder 的 output 轉換成 (μ, Σ) 再透過 reparametrization trick 模擬抽取自 $N(\mu, \Sigma)$ 而來，也可能是由 Gaussian distribution $N(0, I)$ 隨機抽取而來，而 (1, 1, 256) 維度的 embedded input character 則可能是從 teacher forcing 的正確 input character 而來，也可能是從上一次 decoder output 出來的 character 而來，最後透過 ReLU function、LSTM layer、hidden2vocab layer、softmax layer 輸出含有最有可能的 char 的資訊的 distribution。
>
> ```
> class Decoder(nn.Module):
>  def __init__(self, hidden_size, vocab_size):
> ```

```
        super(CVAE.Decoder,self).__init__()
        self.condition_embedding = nn.Embedding(condition_size,
condition_embedding_size)
        self.char_embedding = nn.Embedding(vocab_size, hidden_size)
        self.cellLatentCon2hidden =
nn.Linear(latent_size+condition_embedding_size, hidden_size)
        self.hiddenLatentCon2hidden =
nn.Linear(latent_size+condition_embedding_size, hidden_size)
        self.lstm = nn.LSTM(hidden_size, hidden_size)
        self.hidden2vocab = nn.Linear(hidden_size, vocab_size)
        self.softmax = nn.LogSoftmax(dim=-1)
    def forward(self, input_char_embedded, hidden_state,
cell_state):
        output = F.relu(input_char_embedded)
        output, (hidden_state, cell_state) = self.lstm(output,
(hidden_state, cell_state))
        output_char_dist = self.hidden2vocab(output)
        output_char_dist = self.softmax(output_char_dist)
        return output_char_dist, hidden_state, cell_state
```

- Reparametrization trick

> 之所以會需要 reparametrization trick 是因為原本對 variational auto-encoder 做 back-ward propogation 時，中間會卡一個 stochastic latent code $z \sim N(\mu, \Sigma)$，這會使 gradient 變得很難算，要 propagate graident 的對象會一直變動，所以乾脆把會造成 stochastic 的因素丟給新的變數 $\epsilon \sim N(0, I)$，透過 $z = \mu + \sqrt{\Sigma}N(0, I) = \mu + \sqrt{\Sigma}\epsilon$ 來做出 sample 至 $N(\mu, \Sigma)$ 的效果。
>
> 而在透過 reparametrization trick 實作從 $N(\mu, \Sigma)$ 抽出一個 latent 的過程時，會使用 $\log \text{var}$ 而非 var ，這是因為以下兩點原因：
>
> 1. 使用 σ^2 來當 normal distribution 的 parameter 是 tradition
> 2. $\sigma^2 \in \mathbb{R}^+$，若使用之來當 neural network 的 output，就須限制 neural network output $\in \mathbb{R}^+$，這樣還不如使用 $x = \log \text{var} \in \mathbb{R}$ 來當 neural network 的 output，之後再利用 $\exp(x) = \exp(\log \text{var}) = \sigma^2 > 0$ 轉回 σ^2，如此即可確保 $\sigma^2 > 0$ 的條件。
>
> 
>
>
> ```
> def reparameterize(self, mean, logvar):
>  std = torch.exp(0.5 * logvar)
>  eps = torch.randn_like(std)
>  return mean + eps * std
> ```
>
> reference
>
> > [Why we learn logσ2 in VAE reparameterization trick instead of standard deviation?]
>
> - CVAE

encoder、decoder、reparametrization 是透過 CVAE 包起來的，所以在此對他們的 function 的描述就以 pass 來帶過，詳情可以再往回參考他們個字的定義，而 generate function 將會再等等介紹，所以也以 pass 來帶過，這邊將會專注在介紹 forward function，整個流程可以由下張圖表示，紫色的字代表 tensor 的維度（各個維度數字的定義寫在 "Specify the hyperparameters (KL weight, learning rate, teacher forcing ratio, epochs, etc.) - other hyparameters" 裡）或者需要額外增加的連結或 neural network 層，FC 代表 fully connected layer，藍字代表在該範圍需要使用的技巧。



首先 input_condition 經過 encoder 的 condition_embedding layer 轉成 (1, 1, 8) 維度，再與 init_hidden function 產生出 (1, 1, 256-8) 維度的 zero tensor 進行 concat 得到最初的 (1, 1, 256) 維度的 hidden state 與 cell state，而 input_word 則一一把本身的 character 透過 encoder 的 char_embedding layer 轉成 (1, 1, 256) 維度，將轉換過的 input_word、cell_state、hidden state 輸入進 encoder 得到新的 cell_state、hidden_state，output 則被丟掉，接著再將下一個轉換過的 input_word 與新的 cell_state、hidden_state 丟進 encoder ，如此循環得到最終 encoder 輸出的 hidden_state 與 cell_state。

這個(1, 1, 256) 維度的 hidden_state 與 cell_state 將分別經過 CVAE 的 hidden2meanLatent layer、hidden2logvarLatent layer、cell2meanLatent layer、cell2logvarLatent layer 產生 (1, 1, 32) 維度的 mean_hidden、logvar_hidden、mean_cell、log_cell，經過 reparametrize trick 得到 (1, 1, 32) 維度的 latent_hidden 與 latent_cell。

接下來如果是在 training phase，會將 **input_condition(在 training 的階段, target condition = input condition)** 透過 decoder 的 condition_embedding layer 轉換成 (1, 1, 8) 維度的 tensor，將它分別與 encoder output 轉換而來的 (1, 1, 32) 維度的 latent_hidden 與 latent_cell 做 concat 得到 (1, 1, 40) 維度的 tensor，再分別經過 decoder 的 hiddenLatentCon2hidden layer 與 cellLatentCon2hidden layer 轉換成 (1, 1, 256) 維度的 tensor，亦即decoder 初始的 hidden_state 與 cell_state，將他們與經過 decoder 的 char_embedding layer 轉換過的 (1, 1, 256)維度的初始 input character [SOS] 一起輸進去 decoder，得到 (1, 1, 28) 維度 output_char_dist、(1, 1, 256) 維度的新 hidden_state 與 cell_state，output_char_dist 代表 decoder 認為下一個字可能是哪個 character 的機率，**將它存進 output_word_dist 裡面以便將來去計算 reconstruction loss**，**而下一個 input**

character 將會根據 teacher_forcing_ratio 決定看要取自 ground truth input word 的 character 還是要取自 decoder output 出來的 character，接著流程一樣，將轉換過的 (1, 1, 256) 維度的 input character 跟新的 (1, 1, 256) 維度的 hidden_state 與 cell_state 一起輸進去 decoder 產生出下一個 character 的預測機率分佈，如此循環直到遇到 [EOS] 停止，這樣就可以將 output_word_dist return 回去去計算 **reconstruction loss**、將 (mean_hidden, logvar_hidden), (mean_cell, logvar_cell) return 回去去計算 **regularization loss**。

而如果是在 testing phase，則會將 **target_condition(在 testing 的階段, target condition ≠ input condition)**，透過 decoder 的 condition_embedding layer 轉換成 (1, 1, 8) 維度的 tensor，將它分別與 encoder output 轉換而來的 (1, 1, 32) 維度的 latent_hidden 與 latent_cell 做 concat 得到 (1, 1, 40) 維度的 tensor，再分別經過 decoder 的 hiddenLatentCon2hidden layer 與 cellLatentCon2hidden layer 轉換成 (1, 1, 256) 維度的 tensor，亦即decoder 初始的 hidden_state 與 cell_state，將他們與經過 decoder 的 char_embedding layer 轉換過的 (1, 1, 256)維度的初始 input character [SOS] 一起輸進去 decoder，得到 (1, 1, 28) 維度 output_char_dist、(1, 1, 256) 維度的新 hidden_state 與 cell_state，output_char_dist 代表 decoder 認為下一個字可能是哪個 character 的機率，**將它取最大值存進 output_word 裡面以便將來去計算 BLEU4-score，而下一個 input character 取自 decoder output 出來的 character**，接著流程一樣，將轉換過的 (1, 1, 256) 維度的 input character 跟新的 (1, 1, 256) 維度的 hidden_state 與 cell_state 一起輸進去 decoder 產生出下一個 character 的預測機率分佈，如此循環直到遇到 output 出來最大機率的 character 為 [EOS] 才停止，這樣就可以將 output_word return 回去去計算 BLEU4-score 看 tense conversion 的效果。

```
class CVAE(nn.Module):
#conditional variation auto-encoder
class Encoder(nn.Module):
   pass # has been discussed

class Decoder(nn.Module):
   pass # has been discussed

def __init__(self, vocab_size, condition_size,
condition_embedding_size, hidden_size, latent_size, max_length,
device):
   super(CVAE, self).__init__()
   self.vocab_size, self.condition_size,
self.condition_embedding_size, self.hidden_size, self.latent_size,
self.max_length, self.device = vocab_size, condition_size,
condition_embedding_size, hidden_size, latent_size, max_length,
device
   self.encoder = self.Encoder(vocab_size, condition_size,
condition_embedding_size, hidden_size)
   self.hidden2meanLatent = nn.Linear(hidden_size, latent_size)
   self.hidden2logvarLatent = nn.Linear(hidden_size, latent_size)
   self.cell2meanLatent = nn.Linear(hidden_size, latent_size)
   self.cell2logvarLatent = nn.Linear(hidden_size, latent_size)
   self.decoder = self.Decoder(hidden_size, vocab_size)
```

```
def forward(self, input_word, input_condition, target_condition,
use_teacher_forcing):
   # ----------Encoder---------#
   # initialize hidden state and cell state and forward the encoder
char by char
   hidden_state = torch.cat((self.init_hidden(),
self.encoder.condition_embedding(input_condition))).view(1,1,-1)
   cell_state = torch.cat((self.init_hidden(),
self.encoder.condition_embedding(input_condition))).view(1,1,-1)
#hidden state and cell state share the same condition_embedding
layer
   # forward the encoder char by char
   for input_char in input_word:
          input_char_embedded =
self.encoder.char_embedding(input_char).view(1,1,-1) #embedding the
input character and transfer the size to (sequence length, batch
size, input vector's dimension)
          _, hidden_state, cell_state =
self.encoder.forward(input_char_embedded, hidden_state, cell_state)
   # forward from encoder to decoder
   mean_hidden = self.hidden2meanLatent(hidden_state)
   logvar_hidden = self.hidden2logvarLatent(hidden_state)
   latent_hidden = self.reparameterize(mean_hidden, logvar_hidden)
   mean_cell = self.cell2meanLatent(cell_state)
   logvar_cell = self.cell2logvarLatent(cell_state)
   latent_cell = self.reparameterize(mean_cell, logvar_cell)

   # ----------Decoder---------#
   # handle the hidden state & cell state and forward the encoder
char by char
   if target_condition == None: #training with teacher forcing,
condition = input condition
          hidden_state =
self.decoder.hiddenLatentCon2hidden(torch.cat((latent_hidden,
self.decoder.condition_embedding(input_condition).view(1, 1, -1)),
dim=-1))
          cell_state =
self.decoder.cellLatentCon2hidden(torch.cat((latent_cell,
self.decoder.condition_embedding(input_condition).view(1, 1, -1)),
dim=-1)) #hidden state and cell state share the same
condition_embedding layer
          output_word =
torch.tensor([],dtype=torch.long,device=self.device)
          output_word_dist = torch.zeros(len(input_word),
vocab_size, device=self.device)
          for idx_input_char, input_GT_char in
enumerate(input_word):
                 if input_GT_char ==
DataTransformDict().alpha2num['EOS']:
                     break
                 if input_GT_char ==
DataTransformDict().alpha2num['SOS']:
```

```
                    input_char = input_GT_char #decoder's first
input char must be 'SOS'
                elif use_teacher_forcing:
                    input_char = input_GT_char
                else:
                    input_char = output_char
                input_char_embedded =
self.decoder.char_embedding(input_char).view(1, 1, -1)
                output_char_dist, hidden_state, cell_state =
self.decoder.forward(input_char_embedded, hidden_state, cell_state)
                output_word_dist[idx_input_char+1] =
output_char_dist.squeeze() # record the predicted word to compute
cross entropy
                output_char = output_char_dist.argmax().view(-1) #
view(-1) is to make concate available
                output_word =
torch.cat((output_word.detach(),output_char.detach())) #record the
predicted word with highest probability to check whether input word
is same as output word
            return output_word, output_word_dist, (mean_hidden,
logvar_hidden), (mean_cell, logvar_cell)

    elif target_condition != None: #testing without teacher forcing,
condition = target condition
            hidden_state =
self.decoder.hiddenLatentCon2hidden(torch.cat((latent_hidden,
self.decoder.condition_embedding(target_condition).view(1, 1, -1)),
dim=-1))
            cell_state =
self.decoder.cellLatentCon2hidden(torch.cat((latent_cell,
self.decoder.condition_embedding(target_condition).view(1, 1, -1)),
dim=-1)) #hidden state and cell state share the same
condition_embedding layer
            input_char =
torch.tensor(DataTransformDict().alpha2num['SOS'],device =
self.device)
        output_word = torch.tensor([], dtype=torch.long,
device=self.device)
            for i in range(self.max_length):
                input_char_embedded =
self.decoder.char_embedding(input_char).view(1, 1, -1)
                output_char_dist, hidden_state, cell_state =
self.decoder.forward(input_char_embedded, hidden_state, cell_state)
                output_char = output_char_dist.argmax().view(-1)
                output_word =
torch.cat((output_word.detach(),output_char.detach()))  # record
the predicted word with highest probability to compute the BLEU-4
score
                if output_char ==
DataTransformDict().alpha2num['EOS']:
                    break
                input_char = output_char
            return output_word
```

```
def generate(self, latent_hidden, latent_cell, target_condition):
   pass # will be discussed later

def reparameterize(self, mean, logvar):
   pass # has been discussed

def init_hidden(self):
   return torch.zeros(self.hidden_size -
self.condition_embedding_size, device=self.device)
```

- Text generation

> 之所以可以做 text generation 是因為每個 input word 都有自己一個特別的從 Gaussian sample 出來的 latent code，所以隨機從 Gaussian sample 出一個 latent code 就相當於對應到隨機抽出某個 input word 的 latent code。
>
> 這是 CVAE 裡面定義的 generate function，它只會用到 decoder，過程基本上跟在 testing phase 做 tense conversion 的 decoder 做的事情一樣，只是這次輸給 decoder 的 latent_hidden 與 latent_cell 是隨機從 Gaussian distribution 抽取而來，而 return 的 output_word 是為了計算 gaussian-score 來看 text generation 的效果。

```
def generate(self, latent_hidden, latent_cell, target_condition):
#generating a word with specific latent code
# ----------Decoder---------#
hidden_state =
self.decoder.hiddenLatentCon2hidden(torch.cat((latent_hidden,
self.decoder.condition_embedding(target_condition).view(1, 1, -1)),
dim=-1))
cell_state =
self.decoder.cellLatentCon2hidden(torch.cat((latent_cell,
self.decoder.condition_embedding(target_condition).view(1, 1,
-1)),dim=-1))  # hidden state and cell state share the same
condition_embedding layer
input_char = torch.tensor(DataTransformDict().alpha2num['SOS'],
device=self.device)
output_word = torch.tensor([], dtype=torch.long,
device=self.device)
for i in range(self.max_length):
  input_char_embedded =
self.decoder.char_embedding(input_char).view(1, 1, -1)
  output_char_dist, hidden_state, cell_state =
self.decoder.forward(input_char_embedded, hidden_state, cell_state)
  output_char = output_char_dist.argmax().view(-1)
  output_word =
torch.cat((output_word.detach(),output_char.detach()))  # record
the predicted word with highest probability to compute the BLEU-4
score
  if output_char == DataTransformDict().alpha2num['EOS']:
        break
  input_char = output_char
return output_word
```

> 以下截圖裡的紅匡為 latent code 是隨機抽取自 Gaussian distribution $N(0, I)$ 的證明：
>
> ```
> model.eval()
> words_list = [] # record different words with different tenses
> for i in range(num_word): #generate num_word amount of words
>  words = [] # record a word with different tenses
>  # randomly generate latent_hidden and latent_cell from Gaussian distribution
>  N(0,I) which decides the word's shape
>  latent_hidden = torch.randn(latent_size, device=device).view(1, 1, -1)
>  latent_cell = torch.randn(latent_size, device=device).view(1, 1, -1)
>  # handle the specific word's tense
>  for target_condition in ['sp', 'tp', 'pg', 'p']:
>  target_condition =
>  torch.tensor(DataTransformDict().tense2num[target_condition],
>  dtype=torch.long, device=device) # convert tense from string to
>  tensor, and reshape it to fit the LSTM's input format
>  # generate word with different tenses
>  output_word = model.generate(latent_hidden, latent_cell,
>  target_condition) #generate a word according to the random latent
>  using decoder
>  output_word = ''.join([DataTransformDict().num2alpha[idx.item()] for idx
>  in output_word.detach()[:-1]])  # ignore EOS, convert word from
>  tensor to string
>  words.append(output_word)
>  words_list.append(words)
> gaussian_score = compute_Gaussian_score(words_list)
> ```

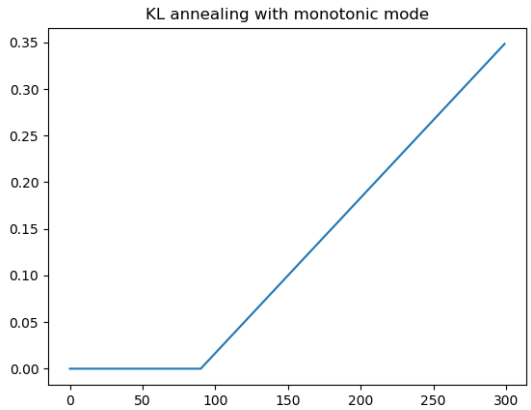
- Specify the hyperparameters (KL weight, learning rate, teacher forcing ratio, epochs, etc.)

  > - KL weight
  >
  > 將會透過 epoch、epoch_size、KL_fix_ratio、KL_monotonic_ratio、KL_cyclical_ratio、KL_upper_bound 來決定 KL_weight 該如何調整，其中 KL_fix_ratio 代表前期有多久要使 KL_weight 固定在 0 上，KL_monotonic_ratio 代表 KL_weight 開始上升之後，需要上升的區間長度有多少，KL_cyclical_ratio 決定半個週期的長度，KL_upper_bound 決定 KL_weight 最高只能到達多少。
  >
  > ```
  > def KL_weight_annealing(epoch, epoch_size, KL_annealing_mode,
  > weight_ratio_annealing_package):
  >  _, _, KL_fix_ratio, KL_monotonic_ratio, KL_cyclical_ratio,
  > KL_upper_bound = weight_ratio_annealing_package
  >  # KL_upper_bound determine how high the KLD_weight will be
  >  # KL_fix_ratio determine how long the KLD_weight will stick to 0
  >  if epoch / epoch_size < KL_fix_ratio:
  >  KLD_weight = 0
  >  else:
  >  moved_epoch = (epoch - epoch_size * KL_fix_ratio)
  >  moved_epoch_size = (epoch_size - epoch_size * KL_fix_ratio)
  >  if KL_annealing_mode == 'monotonic':
  >  # KL_monotonic_ratio control increasing interval's length
  >  if moved_epoch < KL_monotonic_ratio * moved_epoch_size:
  >  KLD_weight = moved_epoch * KL_upper_bound /
  > (moved_epoch_size * KL_monotonic_ratio)
  >  else:
  >  KLD_weight = KL_upper_bound
  >  elif KL_annealing_mode == 'cyclical':
  >  # KL_cyclical_ratio control increasing interval's or 1
  > interval's length
  > ```

```
        if (moved_epoch // (
                KL_cyclical_ratio * epoch_size)) % 2 == 0:  # if
the quotient is an odd number
            KLD_weight = (moved_epoch % (KL_cyclical_ratio *
epoch_size)) * KL_upper_bound / (
                        epoch_size * KL_cyclical_ratio)
        else:  # if the quotient is an even number
            KLD_weight = KL_upper_bound
 return KLD_weight
```

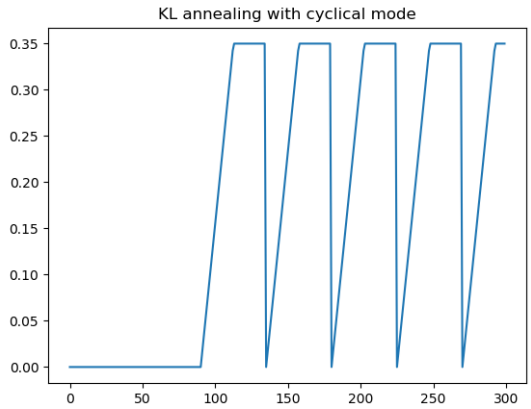
實驗中 monotonic KL_weight 所使用的參數為：

```
KL_fix_ratio=0.3, KL_upper_bound=0.35, KL_monotonic_ratio=1
```



實驗中 cyclical KL_weight 所使用的參數為：

```
KL_fix_ratio=0.3, KL_upper_bound=0.35, KL_monotonic_ratio=0.075
```



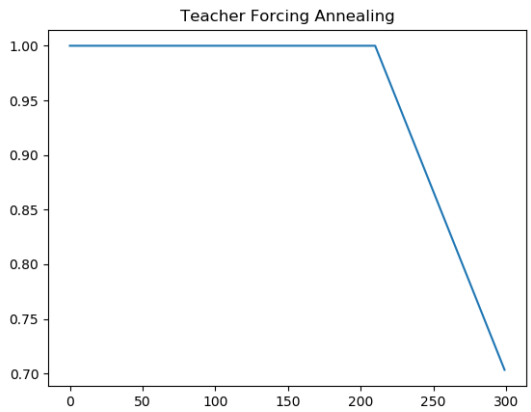
- teacher forcing ratio

> 將會透過 epoch、epoch_size、teacher_fix_ratio、teacher_lower_bound 來調整 teacher forcing ratio，其中 teacher_fix_ratio 代表 teacher forcing ratio 要固定在 1 的區間長度，teacher_lower_bound 代表 teacher forcing ratio 最低只能下降到多少值。

```
def teacher_forcing_ratio_annealing(epoch, epoch_size,
weight_ratio_annealing_package):
  teacher_fix_ratio, teacher_lower_bound, _, _, _, _ =
weight_ratio_annealing_package
  # teacher_lower_bound determine how low the teacher forcing ratio
will be
  # teacher_fix_ratio determine how long the teacher forcing ratio
will stick to 1
  if epoch / epoch_size < teacher_fix_ratio:
        teacher_forcing_ratio = 1
  else:
        moved_epoch = (epoch - epoch_size * teacher_fix_ratio)
        moved_epoch_size = (epoch_size - epoch_size *
teacher_fix_ratio)
        teacher_forcing_ratio = 1 - moved_epoch * (1 -
teacher_lower_bound) / moved_epoch_size
  return teacher_forcing_ratio
```

實驗中 teacher forcing ratio 所使用的參數為：

```
fix_ratio=0.7, lower_bound=0.7
```



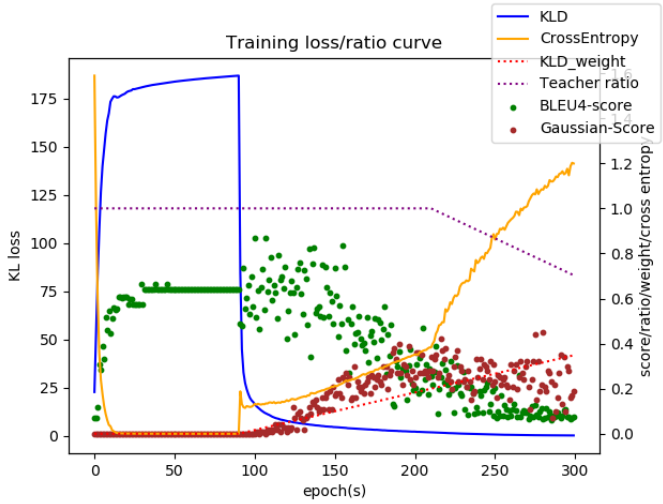
- other-hyperameters

```
epoch_size = 300
hidden_size = 256 # the dimension of hidden state or the embedded
input word
latent_size = 32 # the dimension of embedded latent code
condition_embedding_size = 8 # the dimension of embedded condition
condition_size = 4 # the number of conditions
vocab_size = 28  # the number of vocabulary
LR = 0.05  # learning rate
MAX_LENGTH = 20 # the max length of a word
num_word_generate = 100 #how many words should be generated during
text generation phase
```

5. Results and discussion (20%)
   - Show your results of tense conversion and generation and Plot the Crossentropy loss, KL loss and BLEU-4 score curves during training (5%)

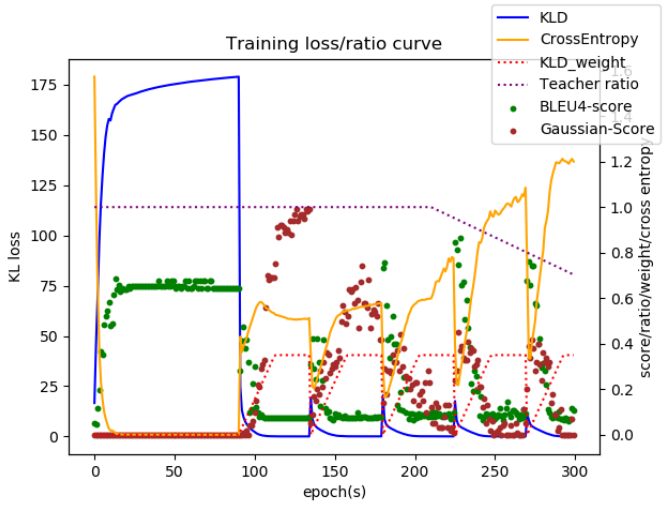
Monotonic KL weight

```
Input: abandon, Target:abandoned, Prediction:abandoned
Input: abet, Target:abetting, Prediction:exiting
Input: begin, Target:begins, Prediction:begins
Input: expend, Target:expends, Prediction:exherts
Input: sent, Target:sends, Prediction:sents
Input: split, Target:splitting, Prediction:splitting
Input: flared, Target:flare, Prediction:flare
Input: functioning, Target:function, Prediction:finction
Input: functioning, Target:functioned, Prediction:functioned
Input: healing, Target:heals, Prediction:heals
BLEU-4 score:0.74528
[['swing', 'swings', 'swinging', 'swing'],
 ['observe', 'observes', 'observing', 'observed'],
 ['madis', 'mades', 'madiating', 'radiated'],
 ['reseat', 'reseats', 'reseating', 'reseated'],
 ['fundo', 'fundies', 'funding', 'fundored'],
 ['festion', 'festions', 'foredining', 'festioned'],
 ['survey', 'speculates', 'prevailing', 'exacerbated'],
 ['exurnish', 'exuders', 'exudering', 'exudered'],
 ['drunt', 'drowns', 'drunting', 'drunt'],
 ['thrust', 'thrusts', 'thrugging', 'threw']]
Gaussian score:0.21
```



Cyclical KL weight

```
Input: abandon, Target:abandoned, Prediction:abandoned
Input: abet, Target:abetting, Prediction:abetting
Input: begin, Target:begins, Prediction:begins
Input: expend, Target:expends, Prediction:expends
Input: sent, Target:sends, Prediction:seizes
Input: split, Target:splitting, Prediction:splitting
Input: flared, Target:flare, Prediction:flaunt
Input: functioning, Target:function, Prediction:function
Input: functioning, Target:functioned, Prediction:functioned
Input: healing, Target:heals, Prediction:heals
BLEU-4 score:0.82976
[['send', 'sends', 'seeing', 'send'],
 ['repair', 'repairs', 'reassuring', 'repaired'],
 ['mention', 'mentions', 'mentioning', 'feinted'],
 ['cling', 'clings', 'clinging', 'clinged'],
 ['award', 'awards', 'finalizing', 'awarded'],
 ['switch', 'steers', 'steaming', 'steamed'],
 ['crow', 'crows', 'crowing', 'crowed'],
 ['withstand', 'withstands', 'withstanding', 'withstood'],
 ['tighten', 'tightens', 'tightening', 'tightened'],
 ['trouble', 'troubles', 'troubling', 'trod']]
Gaussian score:0.33
```



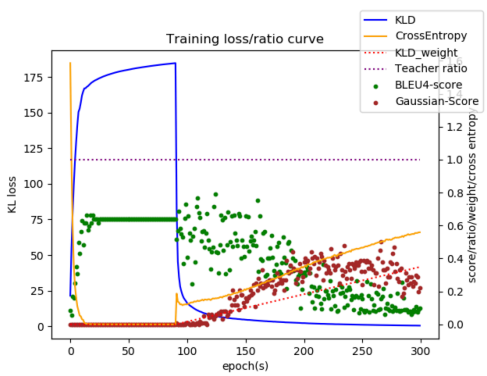
最高的 BLEU4-score 與 Gaussian-score 分別為 0.82976 與 0.33。

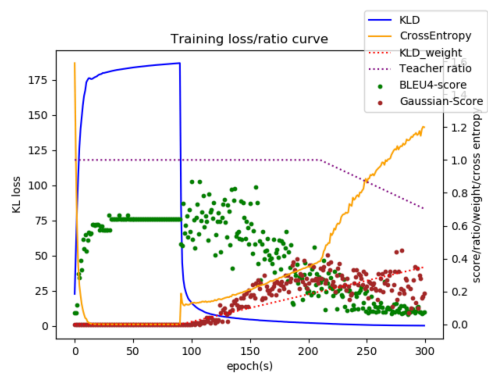
- Discuss the results according to your settings of teacher forcing ratio, KL weight, and learning rate. Note that this part mainly focuses on your discussion, if you simply just paste your results, you will get a low score. (15%)

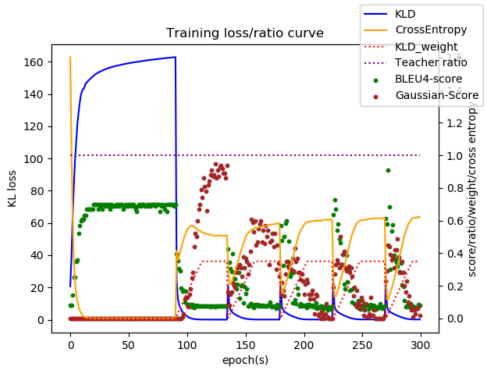
在實驗中並沒有改動 learning rate，它一直維持在預設值 0.05 上，有改動的值只有 teacher forcing ratio 與 KL weight。

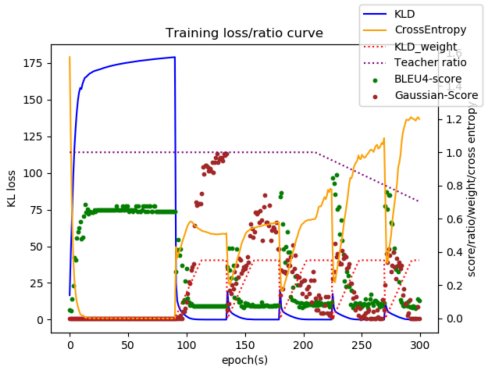
關於 teacher forcing ratio

比較左圖與右圖，將 teacher forcing ratio 固定在 1 之後，發現 BLEU-4 score 的震盪幅度不會收斂，同一個 model weights 可能會出現很高的 BLEU4-score (0.8x) 也可能會出現很低的 model weights (0.0x)，代表 teacher forcing ratio 對於訓練的收斂有很大的關係，或許是因為在做 testing 的時候並沒有使用 teacher forcing 的技巧，因此若沒有在 training 的時候適時放手拿掉 teacher forcing 的話，training 的動作與 testing 的動作就會有挺大的差異，造成 variance 過大，但上升 teacher forcing 的代價就是 reconstruction loss(cross entropy) 會爆升。



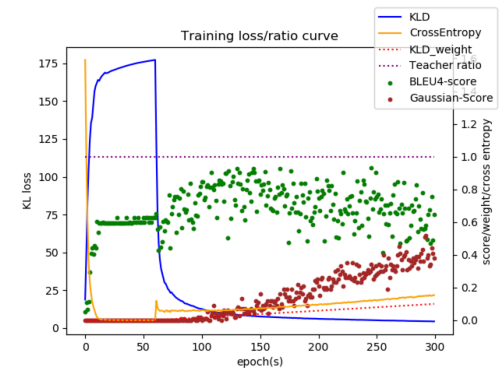


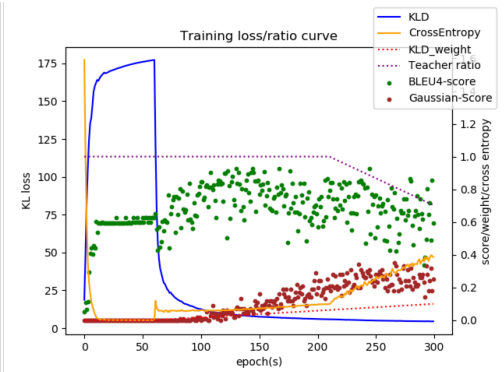


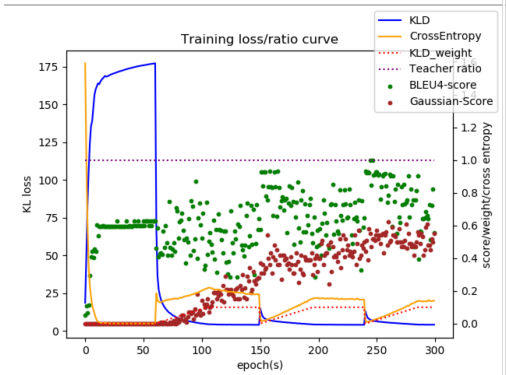


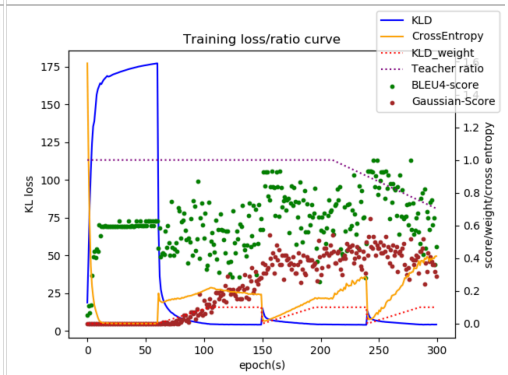
關於 KL weight

> 比較下面四張圖與 teacher forcing ratio 的上面那四張圖，若 KL weight 一下子升得太快，將會導致 BLEU-4 score 降低得很快，出現 BLEU4-score 與 gaussian-score 交叉的情況，而若 KL weight 升得太高，會導致 gaussian-score 異常高且預測的字都長差不多的情況，這次將 KL-weight 的最高上限設成 0.1、KL_cyclical_ratio 設成原先的兩倍(0.15) ，可以看到結果明顯比原先的實驗來得好很多，當 KL weight 上升時，BLEU4-score 與 gaussian-score 交叉的情況很輕微，BLEU-4 score 下降的速度很慢，gaussian-score 上升的速度很正常，tense conversion 與 text generation 都各自具備著具準確率與多樣性。









> 而使用這個參數在 monotonic KL weight之中最好的 model 可以達到這樣的準確率：

```
Input: abandon, Target:abandoned, Prediction:abandoned
Input: abet, Target:abetting, Prediction:abetting
Input: begin, Target:begins, Prediction:begins
Input: expend, Target:expends, Prediction:peels
Input: sent, Target:sends, Prediction:sents
Input: split, Target:splitting, Prediction:splitting
Input: flared, Target:flare, Prediction:flare
Input: functioning, Target:function, Prediction:symbolize
Input: functioning, Target:functioned, Prediction:functioned
Input: healing, Target:heals, Prediction:heals
BLEU-4 score:0.74048
[['entitle', 'entitles', 'entitling', 'entitled'],
 ['abstain', 'abstains', 'abstaining', 'abstained'],
 ['keep', 'keeps', 'keeping', 'keel'],
 ['supervise', 'supervises', 'supervising', 'supervised'],
 ['buzz', 'buzzes', 'buzzing', 'buzzed'],
 ['draw', 'drans', 'drawing', 'dran'],
 ['overcome', 'overcomes', 'overcoming', 'overcomed'],
 ['bankroll', 'bainsays', 'bayshing', 'bainsayed'],
 ['puct', 'publishes', 'publishing', 'publis'],
 ['gobble', 'gobbles', 'gobbling', 'gobbled']]
Gaussian score:0.39
```

在 cyclical KL weight之中最好的 model 則可以達到這樣的準確率：

```
Input: abandon, Target:abandoned, Prediction:abandoned
Input: abet, Target:abetting, Prediction:abetting
Input: begin, Target:begins, Prediction:begins
Input: expend, Target:expends, Prediction:expends
Input: sent, Target:sends, Prediction:sents
Input: split, Target:splitting, Prediction:splitting
Input: flared, Target:flare, Prediction:flare
Input: functioning, Target:function, Prediction:function
Input: functioning, Target:functioned, Prediction:functioned
Input: healing, Target:heals, Prediction:heals
BLEU-4 score:0.92857
[['subside', 'subsides', 'subsiding', 'subsided'],
 ['disclose', 'discloses', 'disclosing', 'disclosed'],
 ['forsw', 'forswores', 'forsworing', 'forswore'],
 ['federate', 'federates', 'featuring', 'featured'],
 ['condescend', 'condescends', 'winding', 'widened'],
 ['revert', 'reverts', 'reverting', 'revorted'],
 ['enforce', 'enforces', 'enforcing', 'enforced'],
 ['extend', 'extends', 'extending', 'extended'],
 ['misdeal', 'minimizes', 'misdealing', 'misdealed'],
 ['convene', 'convenes', 'convening', 'convened']]
Gaussian score:0.55
```

效果極好！