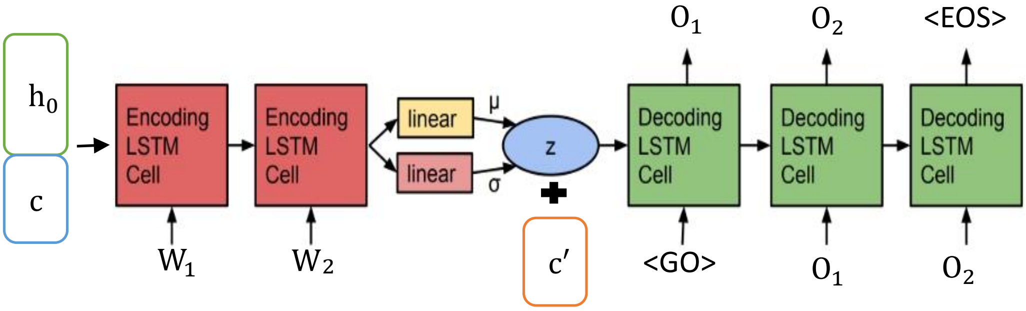
Lab 3: Conditional Sequence-to-sequence VAE

1. Introduction

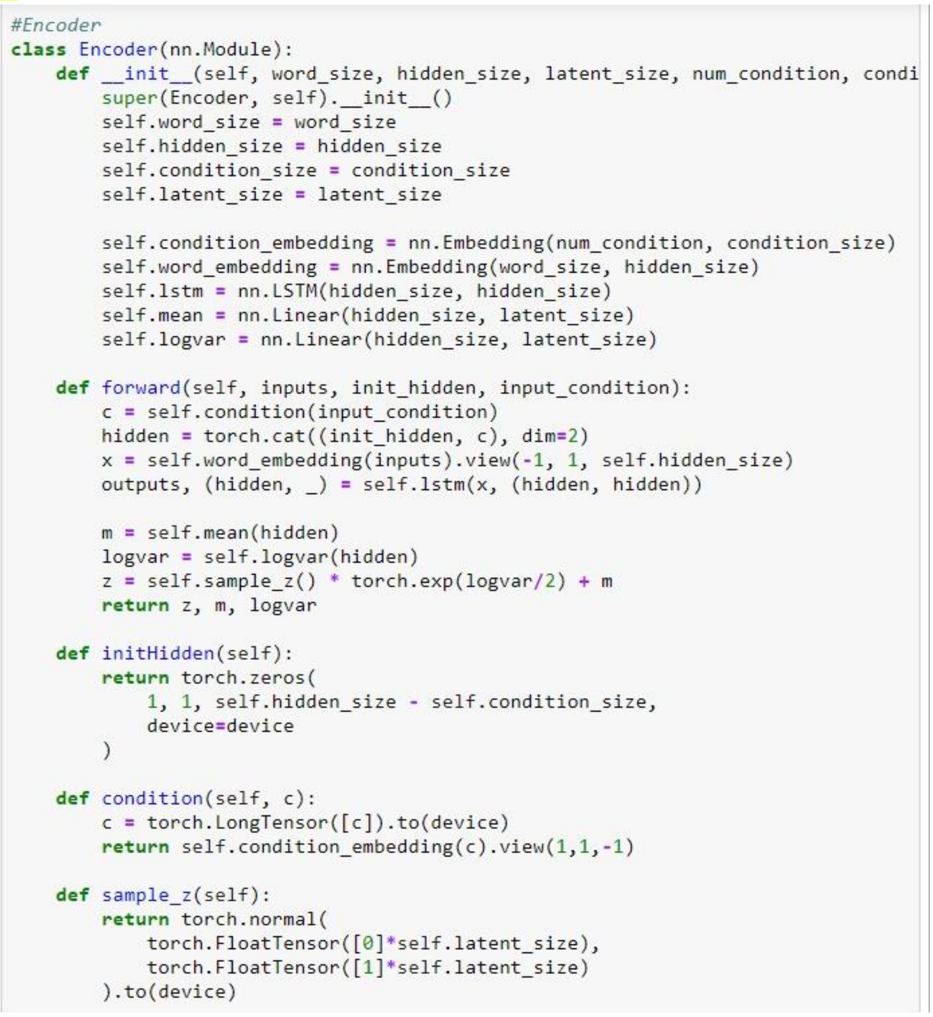


For CVAE, we will input c (tense) and vocabulary to RNN model and given another c’ (target tense) to get the vocabulary in target tense.

2. Implementation detail

A. Describe how you implement your model (encoder, decoder, reparameterization trick, dataloader, etc.).

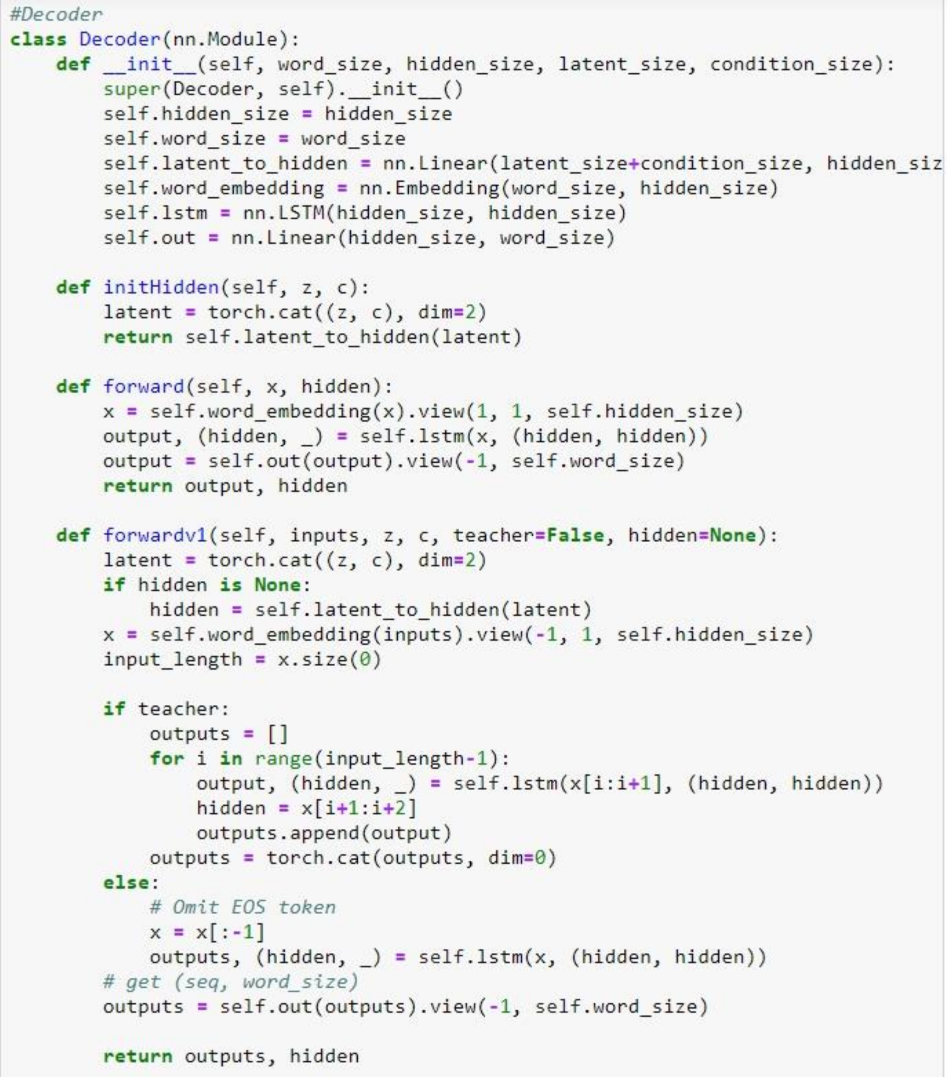
Encoder



```
#Encoder
class Encoder(nn.Module):
    def __init__(self, word_size, hidden_size, latent_size, num_condition, condi
        super(Encoder, self).__init__()
        self.word_size = word_size
        self.hidden_size = hidden_size
        self.condition_size = condition_size
        self.latent_size = latent_size

        self.condition_embedding = nn.Embedding(num_condition, condition_size)
        self.word_embedding = nn.Embedding(word_size, hidden_size)
        self.lstm = nn.LSTM(hidden_size, hidden_size)
        self.mean = nn.Linear(hidden_size, latent_size)
        self.logvar = nn.Linear(hidden_size, latent_size)

    def forward(self, inputs, init_hidden, input_condition):
        c = self.condition(input_condition)
        hidden = torch.cat((init_hidden, c), dim=2)
        x = self.word_embedding(inputs).view(-1, 1, self.hidden_size)
        outputs, (hidden, _) = self.lstm(x, (hidden, hidden))

        m = self.mean(hidden)
        logvar = self.logvar(hidden)
        z = self.sample_z() * torch.exp(logvar/2) + m
        return z, m, logvar

    def initHidden(self):
        return torch.zeros(
            1, 1, self.hidden_size - self.condition_size,
            device=device
        )

    def condition(self, c):
        c = torch.LongTensor([c]).to(device)
        return self.condition_embedding(c).view(1,1,-1)

    def sample_z(self):
        return torch.normal(
            torch.FloatTensor([0]*self.latent_size),
            torch.FloatTensor([1]*self.latent_size)
        ).to(device)
```

I try to embed both the condition(tense) and each word into 32 dim and 256 dim, then concatenate condition_embedding and initial hidden input. Using nn.LSTM to RNN model and feed vocabulary in order to get last hidden output and latent vector z. The latent code should distribute from normal distribution, so I convert last hidden output into mean and log variance by nn.Linear() . To get a latent code z, I get a random noise from normal distribution and times exp(log variance) and then add mean.

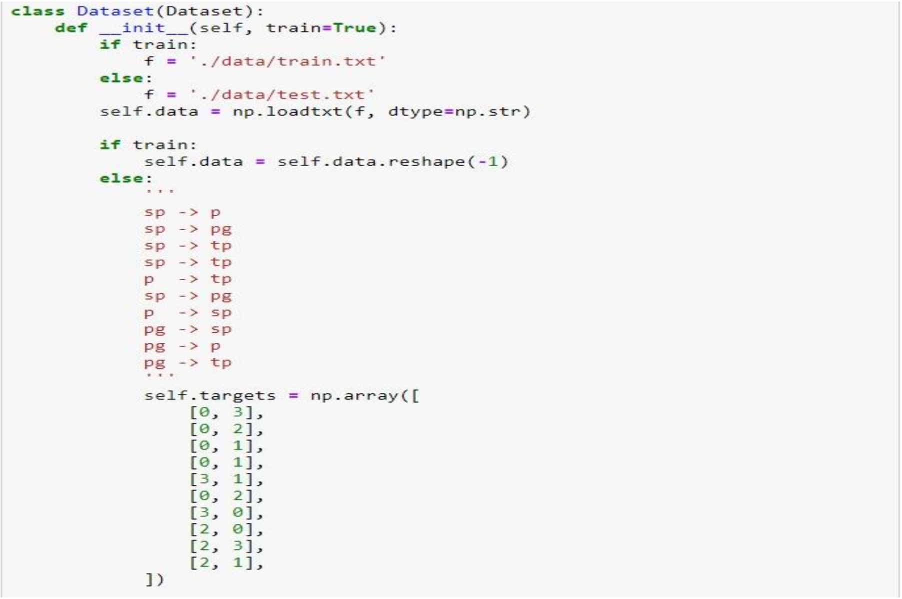
Decoder



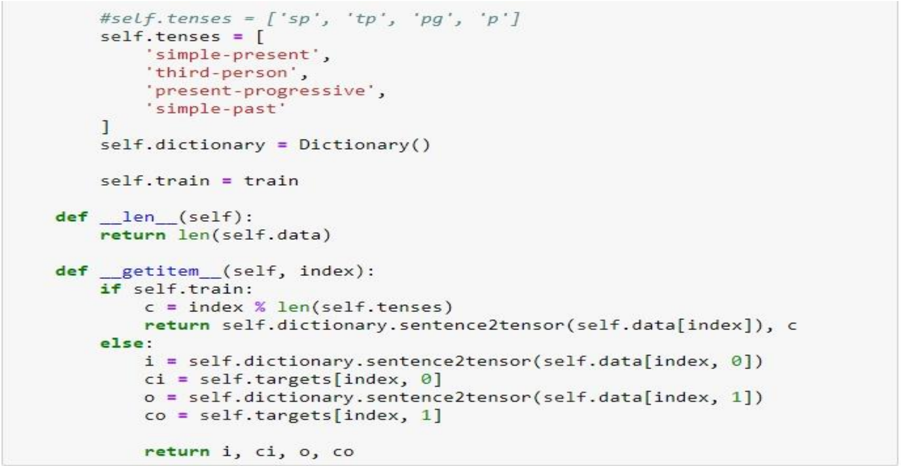
```
#Decoder
class Decoder(nn.Module):
    def __init__(self, word_size, hidden_size, latent_size, condition_size):
        super(Decoder, self).__init__()
        self.hidden_size = hidden_size
        self.word_size = word_size
        self.latent_to_hidden = nn.Linear(latent_size+condition_size, hidden_siz
        self.word_embedding = nn.Embedding(word_size, hidden_size)
        self.lstm = nn.LSTM(hidden_size, hidden_size)
        self.out = nn.Linear(hidden_size, word_size)

    def initHidden(self, z, c):
        latent = torch.cat((z, c), dim=2)
        return self.latent_to_hidden(latent)

    def forward(self, x, hidden):
        x = self.word_embedding(x).view(1, 1, self.hidden_size)
        output, (hidden, _) = self.lstm(x, (hidden, hidden))
        output = self.out(output).view(-1, self.word_size)
        return output, hidden

    def forwardv1(self, inputs, z, c, teacher=False, hidden=None):
        latent = torch.cat((z, c), dim=2)
        if hidden is None:
            hidden = self.latent_to_hidden(latent)
        x = self.word_embedding(inputs).view(-1, 1, self.hidden_size)
        input_length = x.size(0)

        if teacher:
            outputs = []
            for i in range(input_length-1):
                output, (hidden, _) = self.lstm(x[i:i+1], (hidden, hidden))
                hidden = x[i+1:i+2]
                outputs.append(output)
            outputs = torch.cat(outputs, dim=0)
        else:
            # Omit EOS token
            x = x[:-1]
            outputs, (hidden, _) = self.lstm(x, (hidden, hidden))
        # get (seq, word_size)
        outputs = self.out(outputs).view(-1, self.word_size)

        return outputs, hidden
```

First of all, we need to convert latent vector z and target tense c’ into hidden size through nn.Linear(). After that, we will input word step by step. We can given input the ground truth (teacher forcing) in order to avoid previous output is totally wrong or the RNN model output.

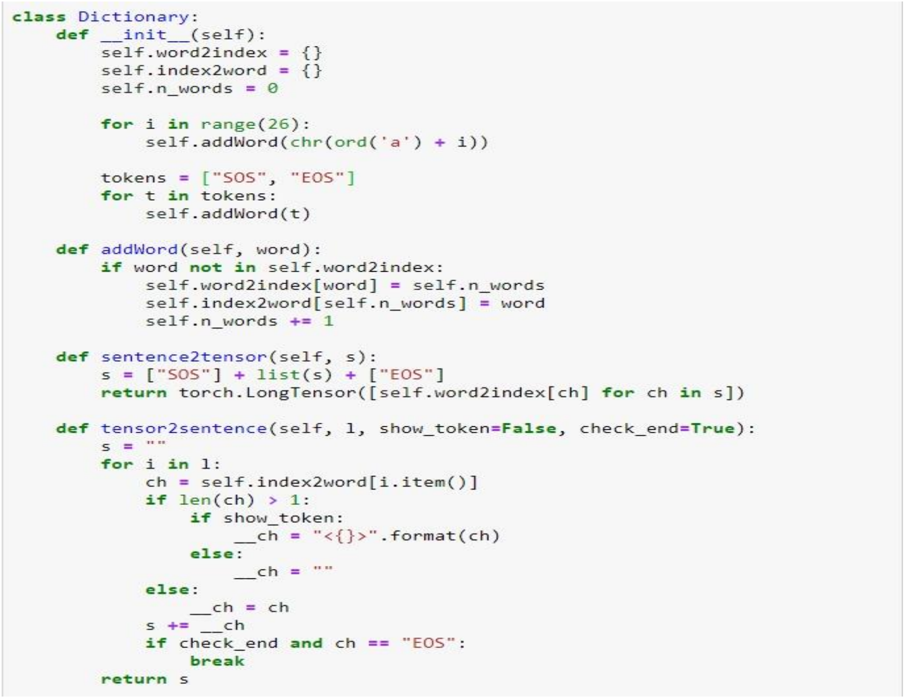
DataLoader



```
class Dataset(Dataset):
    def __init__(self, train=True):
        if train:
            f = './data/train.txt'
        else:
            f = './data/test.txt'
        self.data = np.loadtxt(f, dtype=np.str)

        if train:
            self.data = self.data.reshape(-1)
        else:
            '''
            sp -> p
            sp -> pg
            sp -> tp
            sp -> tp
            p  -> tp
            sp -> pg
            p  -> sp
            pg -> sp
            pg -> p
            pg -> tp
            '''
            self.targets = np.array([
                [0, 3],
                [0, 2],
                [0, 1],
                [0, 1],
                [3, 1],
                [0, 2],
                [3, 0],
                [2, 0],
                [2, 3],
                [2, 1],
            ])
```



```
        #self.tenses = ['sp', 'tp', 'pg', 'p']
        self.tenses = [
            'simple-present',
            'third-person',
            'present-progressive',
            'simple-past'
        ]
        self.dictionary = Dictionary()

        self.train = train

    def __len__(self):
        return len(self.data)

    def __getitem__(self, index):
        if self.train:
            c = index % len(self.tenses)
            return self.dictionary.sentence2tensor(self.data[index]), c
        else:
            i = self.dictionary.sentence2tensor(self.data[index, 0])
            ci = self.targets[index, 0]
            o = self.dictionary.sentence2tensor(self.data[index, 1])
            co = self.targets[index, 1]

            return i, ci, o, co
```



```
class Dictionary:
    def __init__(self):
        self.word2index = {}
        self.index2word = {}
        self.n_words = 0

        for i in range(26):
            self.addWord(chr(ord('a') + i))

        tokens = ["SOS", "EOS"]
        for t in tokens:
            self.addWord(t)

    def addWord(self, word):
        if word not in self.word2index:
            self.word2index[word] = self.n_words
            self.index2word[self.n_words] = word
            self.n_words += 1

    def sentence2tensor(self, s):
        s = ["SOS"] + list(s) + ["EOS"]
        return torch.LongTensor([self.word2index[ch] for ch in s])

    def tensor2sentence(self, l, show_token=False, check_end=True):
        s = ""
        for i in l:
            ch = self.index2word[i.item()]
            if len(ch) > 1:
                if show_token:
                    __ch = "<{}>".format(ch)
                else:
                    __ch = ""
            else:
                __ch = ch
            s += __ch
            if check_end and ch == "EOS":
                break
        return s
```

We create a dictionary to record word and its index, thus we can convert each easily. The total words in dictionary is 28 (26 character and <SOS>, <EOS>). We use dataloader to load the data not only record the word in dictionary but also convert it into LongTensor. We also set condition(sp, p, pg, tp) to different numbers (0, 1, 2, 3).

Reparameterization trick

To implement encoder and decoder as a neural network, we need to backpropagate through random sampling and that is the problem because backpropagation cannot flow through random node. To overcome this obstacle, we use reparameterization trick :

$$z \sim N(\mu, \sigma)$$

is equivalent to sampling $\varepsilon \sim N(0, 1)$ and setting $z = \mu + \sigma * \varepsilon$, we sampled while keeping our sampling operation differentiable.

Another trick is variance from nn.Linear() will return negative number, but variance can not be negative. We take it as log variance, we can convert variance into positive by exp(log var).

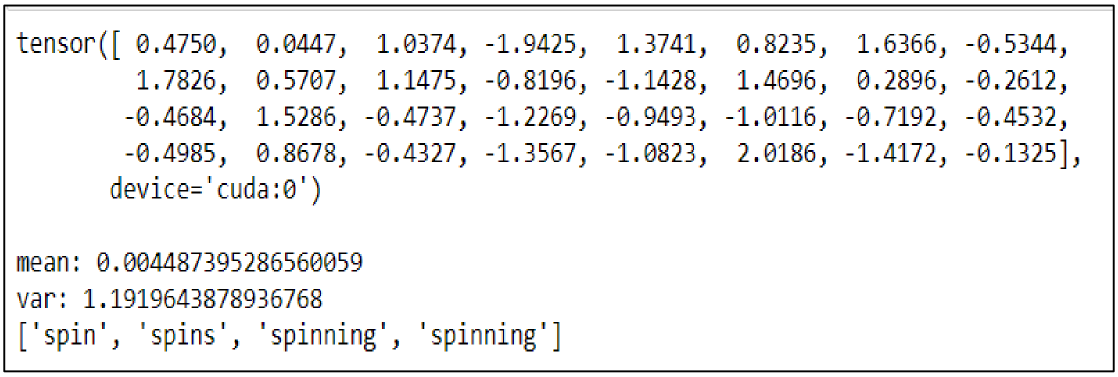
B. Specify the hyperparameters (KL weight, learning rate, teacher forcing ratio, epochs, etc.)

- KL weight: 0.0 with slope 0.001
- Learning rate: 0.01
- Teacher forcing ratio: 0.85
- Epochs: 150
- LSTM Hidden size: 256
- Latent size: 32
- Optimizer: SGD

3. Results and discussion

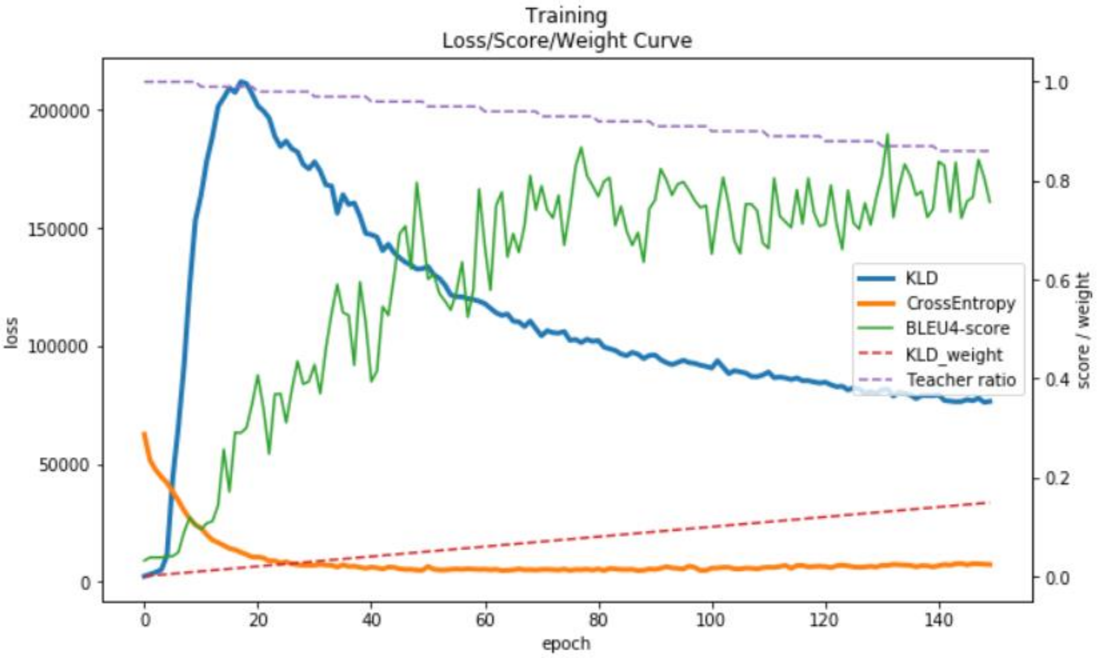
A. Show your results of tense conversion and generation

I generate sample from normal distribution which is called by encoder.sample_z().



```
tensor([ 0.4750,  0.0447,  1.0374, -1.9425,  1.3741,  0.8235,  1.6366, -0.5344,
         1.7826,  0.5707,  1.1475, -0.8196, -1.1428,  1.4696,  0.2896, -0.2612,
        -0.4684,  1.5286, -0.4737, -1.2269, -0.9493, -1.0116, -0.7192, -0.4532,
        -0.4985,  0.8678, -0.4327, -1.3567, -1.0823,  2.0186, -1.4172, -0.1325],
       device='cuda:0')

mean: 0.004487395286560059
var: 1.1919643878936768
['spin', 'spins', 'spinning', 'spinning']
```

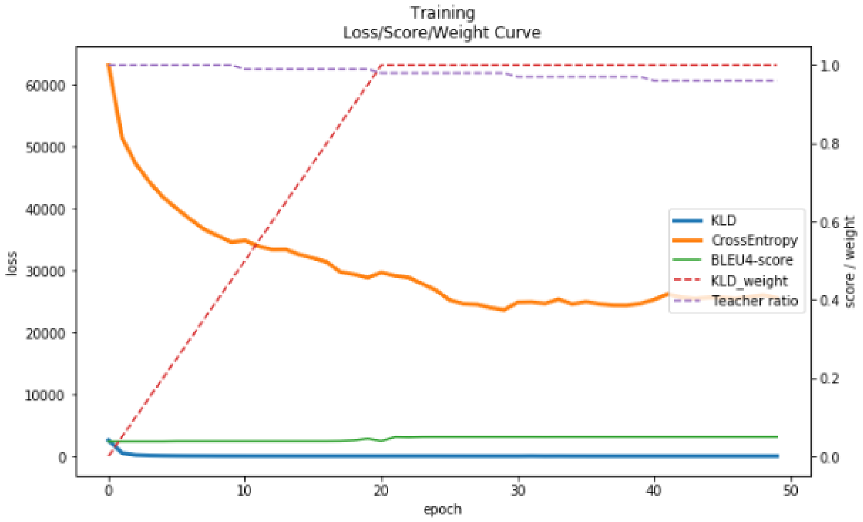
B. Plot the Crossentropy loss, KL loss and BLEU-4 score curves during training and discuss the results according to your setting of teacher forcing ratio, KL weight, and learning rate.



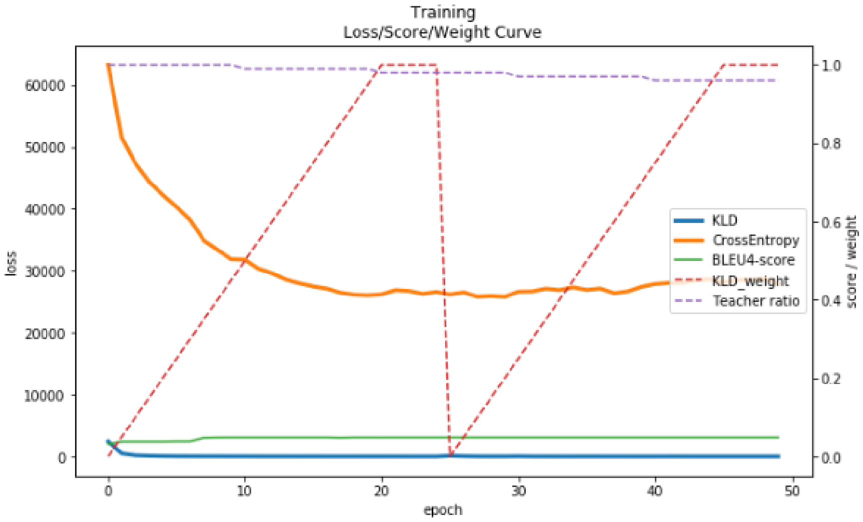
From the picture above, I set teacher forcing ratio 0.85, KL weight 0 and learning rate 0.01. I control KLD_weight in small ratio and the BLEU4-score got higher score and KLD loss became lower. We try to decrease the teacher forcing ratio to let the model become stronger. With higher KLD_weight, model can learn better and KLD Loss become lower.

KL cost annealing 的實驗問題:

Monotonic method:



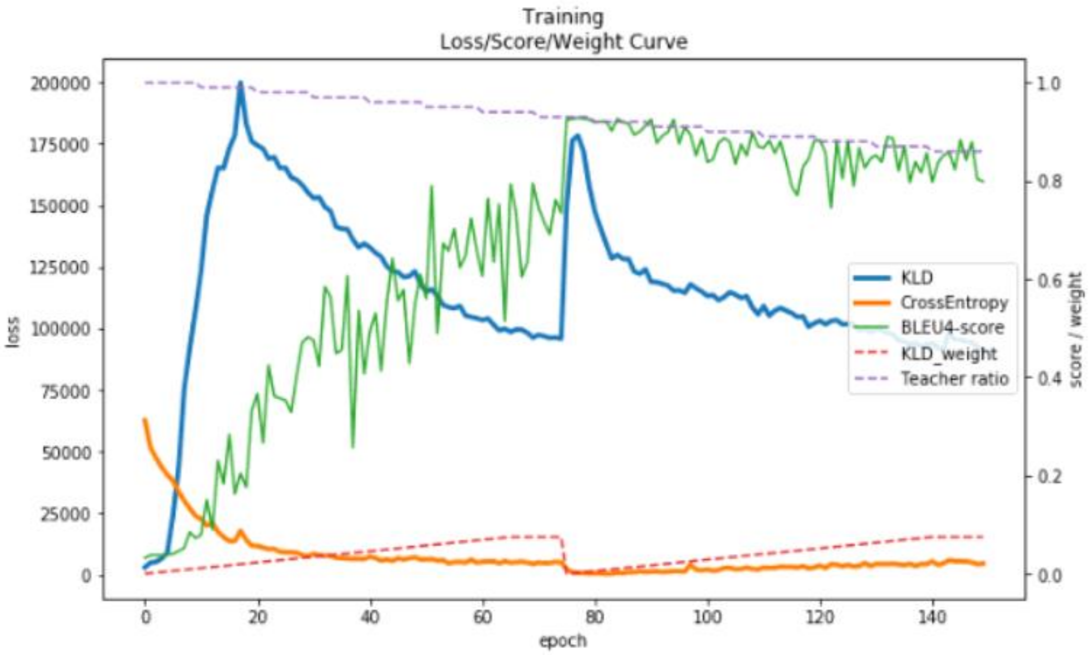
Cyclical method:



我在測試這 2 個方法時，發現如果 KLD_weight 上升太快時，會使 BLEU4-score 維持在一個很低的狀態，我嘗試增加 epoch 次數到 200 次但結果仍然跟 epoch 50 次是一樣的圖，我也嘗試調整 learning rate 從 0.01 -> 0.001 但 BLEU4-score 還是會卡在 0.04865 的地方，可是我如果將 KLD_weight 的上升率調慢的話 model 就會正常開始學習東西，因此我猜測我的 Annealing 方法結果不盡理想的原因是因為 KLD_weight 在學習上是需要非常慢的比率(約在 0.001 左右)才能讓機器穩定的做學習。

下面的圖示我對猜測的實驗結果，因為受限於 GPU 的關係，所以我做了一點調整來模擬 KL weight annealing。可以發現 cyclical method 有一個很明顯的高

峰值，且 KLD_weight 需要一個很緩慢的上升比率才行(KLD_weight=0.001)。


Training
Loss/Score/Weight Curve
loss
epoch
score / weight
KLD
CrossEntropy
BLEU4-score
KLD_weight
Teacher ratio