DLP Lab4: Conditional Sequence-to-Sequence VAE

Zhi-Yi Chin
joycenerd.cs09@nycu.edu.tw

August 17, 2021

1 Introduction

In this lab, we are implementing conditional seq2seq VAE (Figure 1) for English tense conversion and generation. The following are the detailed tasks we need to do:

- Implementing encoder and decoder of CVAE using LSTM.
- Using reparameterization trick to generate the latent vector z.
- Adopt teacher forcing ratio and KL loss throughout the training process.
- Apply to KL annealing method (monotonic and cyclical).
- Visualize the results.

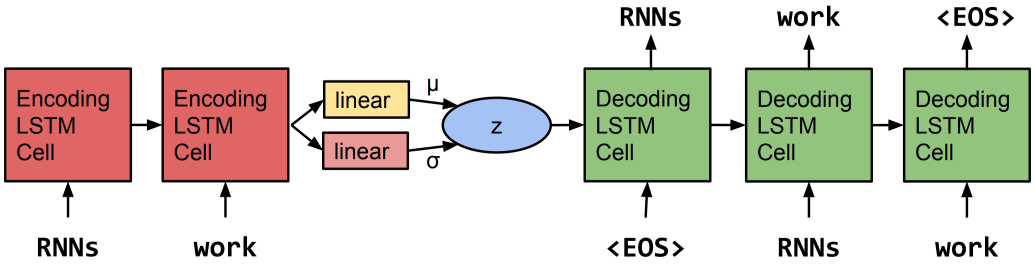


Figure 1: Conditional seq2seq VAE.

2 Derivation of CVAE

To determine θ, we would hope to maximize the marginal distribution $p(x;\theta)$ shown in Eq. 1

$$p(x;\theta) = \int p(x|z;\theta)p(z)dz \tag{1}$$

However, this becomes difficult as the integration over z is generally intractable when $p(x|z;\theta)$ is modeled by a neural network. To circumvent this difficulty, we recall Eq.3.

$$\log p(X;\theta) = \mathcal{L}(X,q,\theta) + \mathrm{KL}(q(z)||p(Z|X,\theta)) \tag{2}$$

where

$$\begin{aligned}\mathcal{L}(X,q,\theta) &= \int q(Z)\log p(X,Z;\theta)dZ - \int q(Z)\log q(Z)dZ \\ \mathrm{KL}(q(Z)||p(Z|X;\theta)) &= \int q(Z)\log\frac{q(Z)}{p(Z|Z;\theta)}dZ\end{aligned} \tag{3}$$

As the equality holds for any choice of $q(Z)$, we introduce a distribution $q(Z|X;\theta')$ modeled by another neural network with parameter θ'. We can replace every $q(Z)$ to $q(Z|X)$ in Eq.3 and organize into Eq.4.

$$\begin{aligned}\mathcal{L}(X,q,\theta) &= \int q(Z|X)\log p(X,Z)dZ - \int q(Z|X)\log q(Z|X)dZ \\ &= \int q(Z|X)\log p(X|Z)p(Z)dZ - \int q(Z|X)\log q(Z|X)dZ \\ &= \mathbb{E}_{Z\sim q(Z|X;\theta')}\log p(X|Z;\theta) + \mathbb{E}_{Z\sim q(Z|X;\theta')}\log p(Z) - \mathbb{E}_{Z\sim q(Z|X;\theta')}\log q(Z|X;\theta') \\ &= \mathbb{E}_{Z\sim q(Z|X;\theta')}\log p(X|Z;\theta) - \mathrm{KL}(q(Z|X;\theta')||p(Z))\end{aligned} \tag{4}$$

From Eq.4, we can easily see that $\mathbb{E}_{Z\sim q(Z|X;\theta')}\log p(X|Z;\theta)$ is the reconstruction term and $\mathrm{KL}(q(Z|X;\theta')||p(Z))$ is the regularization term.

3 Derivation of KL Divergence Loss

First of all, why do we need KL loss? If the loss function only cares about output accuracy, it cannot prove the latent vector z is Gaussian normal distribution $\mathcal{N}(0,1)$. So we derive KL loss as Eq.5.

$$\begin{aligned}\mathrm{KL}(\mathcal{N}(\mu|\sigma^2)||\mathcal{N}(0,1)) &= \int \frac{1}{\sqrt{2\pi\sigma^2}}\exp(-\frac{(x-\mu)^2}{2\sigma^2})\left(\log\frac{\exp(-\frac{(x-\mu)^2}{2\sigma^2})/\sqrt{2\pi\sigma^2}}{\exp(\frac{-x^2}{2})/\sqrt{2\pi}}\right)dx \\ &= \int \frac{1}{\sqrt{2\pi\sigma^2}}\exp(-\frac{(x-\mu)^2}{2\sigma^2})\log\left(\frac{1}{\sqrt{\sigma^2}}\exp(\frac{1}{2}(x^2-\frac{(x-\mu)^2}{\sigma^2}))\right)dx \\ &= \int \frac{1}{\sqrt{2\pi\sigma^2}}\exp(-\frac{(x-\mu)^2}{2\sigma^2})\frac{1}{2}\left(-\log\sigma^2 + x^2 - \frac{(x-\mu)^2}{\sigma^2}\right)dx \\ &= \frac{1}{2}\left(\mathbb{E}_{x\sim\exp(-\frac{(x-\mu)^2}{\sigma^2})}(-\log\sigma^2) + \mathbb{E}_{x\sim\exp(-\frac{(x-\mu)^2}{\sigma^2})}(x^2) - \mathbb{E}_{x\sim\exp(-\frac{(x-\mu)^2}{\sigma^2})}\left(\frac{(x-\mu)^2}{\sigma^2}\right)\right) \\ &= \frac{1}{2}\left(-\log\sigma^2 + (\mu^2+\sigma^2) - \frac{1}{\sigma^2}\int\frac{1}{\sqrt{2\pi\sigma^2}}\exp(-\frac{(x-\mu)^2}{\sigma^2})(x-\mu)^2dx\right) \\ &= \frac{1}{2}(-\log\sigma^2 + \mu^2 + \sigma^2 - 1)\end{aligned} \tag{5}$$

The code implementation of KL Loss is as follow:

```
class KLLoss(nn.Module):
    def __init__(self):
        super(KLLoss,self).__init__()

    def forward(self,mean,log_var):
        return torch.sum(0.5 * (-log_var + (mean**2) + torch.exp(log_var) - 1))
```

4 Implementation Details

4.1 Data pre-processing

For **training** data, we flatten the data to 1 vector since for training data, there are four tenses in one line. The condition of the training data is the index of the flatten data modulo 4. For **testing** data, we have the input word, and target word and the tenses conversion have been mentioned in `readme.txt`, we code it into integer numbers.

```
class TextDataset(Dataset):
    def __init__(self,root_dir,mode):
        self.root_dir=root_dir
        self.mode=mode

        file=Path(self.root_dir).joinpath(f"{self.mode}.txt")
        self.data=np.loadtxt(file,dtype=str)

        if self.mode=='train':
            self.data=self.data.reshape(-1)
        else:
            self.tense=[[0,3],[0,2],[0,1],[0,1],[3,1],[0,2],[3,0],[2,0],[2,3],
                        [2,1]]
            self.tense=np.asarray(self.tense,dtype=int)

    def __len__(self):
        return self.data.shape[0]

    def __getitem__(self, index):
        if self.mode=='train':
            data=self.data[index]
            c=index%4
            return data,c
        else:
            data=self.data[index,0]
            target=self.data[index,1]
            c1=self.tense[index,0]
            c2=self.tense[index,1]
            return data,target,c1,c2
```

Listing 1: Dataset class.

Deep learning models only accept numbers, so we transform all the words into one hot encoder according to the alphabet. Since we only have lower case alphabets, so a...z mapped to 2...27. Moreover, we also add `SOS` token at the beginning of each word which is of value 0, and `EOS` token at the end of each word which is of value 1.

```python
class OneHotEncoder():
    def __init__(self):
        self.sos=0
        self.eos=1

    def tokenize(self,word):
        chars=['SOS']+list(word)+['EOS']
        token=[]
        for ch in chars:
            if ch=='SOS':
                token.append(self.sos)
            elif ch=='EOS':
                token.append(self.eos)
            else:
                token.append(ord(ch)-ord('a')+2)
        token=torch.from_numpy(np.asarray(token))
        return token

    def inv_tokenize(self,token):
        word=''
        for val in token:
            if val==1:
                break
            word+=chr(val-2+ord('a'))
        return word
```

Listing 2: Onehot encoder.

4.2 Encoder

We initialize the hidden state and cell state to 0. Then we embed condition c and concatenate c with hidden state and cell state, respectively. After that, we perform a fully connected layer to make the hidden state and cell state size to hidden layer size. Then we embed the input sequence x and passed the input sequence, hidden state, and cell state into LSTM and output the new hidden state and the new cell state. Then we perform a full connection to get mean and log variance for both hidden state and cell state. We get the latent z by $\exp(\text{logvar}/2) * \epsilon + \mu$, ϵ is sample from $\mathcal{N}(0, 1)$, this is known as the **reparameterization trick**.

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

```
        self.device=device

        self.embedding = nn.Embedding(input_size, hidden_size)
        self.c_embedding=nn.Embedding(c_size,c_hidden_size)
        self.lstm=nn.LSTM(self.hidden_size,self.hidden_size)
        self.hidden_fc=nn.Linear(self.hidden_size+self.c_hidden_size,
                                 self.hidden_size)
        self.cell_fc=nn.Linear(self.hidden_size+self.c_hidden_size,
                               self.hidden_size)

        self.hidden_mean=nn.Linear(hidden_size,z_size)
        self.hidden_log_var=nn.Linear(hidden_size,z_size)
        self.cell_mean=nn.Linear(hidden_size,z_size)
        self.cell_log_var=nn.Linear(hidden_size,z_size)

    def forward(self,x,hidden_state,cell_state,c):
        # hidden state
        c=self.c_embedding(c).reshape(1,1,-1)
        hidden_state=torch.cat((hidden_state,c),dim=2)
        hidden_state=self.hidden_fc(hidden_state)

        # cell state
        cell_state=torch.cat((cell_state,c),dim=2)
        cell_state=self.cell_fc(cell_state)

        x=self.embedding(x).reshape(-1,1,self.hidden_size)
        out,(hidden_state,cell_state)=self.lstm(x,(hidden_state,cell_state))

        # mean and variance
        hidden_mu=self.hidden_mean(hidden_state)
        hidden_log_sigma=self.hidden_log_var(hidden_state)
        hidden_eps=torch.normal(torch.zeros(self.z_size),
                                torch.ones(self.z_size)).to(self.device)
        # reparameterization trick
        hidden_z=torch.exp(hidden_log_sigma/2)*hidden_eps+hidden_mu

        cell_mu=self.cell_mean(cell_state)
        cell_log_sigma=self.cell_log_var(cell_state)
        cell_eps=torch.normal(torch.zeros(self.z_size),
                              torch.ones(self.z_size)).to(self.device)
        # reparameterization trick
        cell_z=torch.exp(cell_log_sigma/2)*self.std_normal+cell_mu

        return hidden_mu,hidden_log_sigma,hidden_z,cell_mu,cell_log_sigma,cell_z

    def init_hidden_and_cell(self):
        hidden_state=torch.zeros(1,1,self.hidden_size,device=self.device)
        cell_state=torch.zeros(1,1,self.hidden_size,device=self.device)
```

```
58          return hidden_state,cell_state
```

Listing 3: Encoder of CVAE

4.3 Decoder

To initialize the decoder, we first embed the condition. Then we concatenate hidden state latent code z and c and perform a full connection to make it of size hidden layer size, which is the decoder's initial hidden state. For the initial cell state, we do the same thing. For the tense conversion task, the latent code z is from the encoder, and in the word generation task, the latent code z is a sample from $\mathcal{N}(0, 1)$.

In the decoder, we take one token at a time and perform embedding and activate by ReLU. Then we use LSTM with the token, hidden state, cell state as inputs and output the output token, new hidden state, and new cell state.

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

        self.c_embedding=nn.Embedding(c_size,c_hidden_size)
        self.hidden_fc=nn.Linear(z_size+c_hidden_size,hidden_size)
        self.cell_fc=nn.Linear(z_size+c_hidden_size,hidden_size)
        self.embedding = nn.Embedding(output_size, hidden_size)
        self.lstm = nn.LSTM(hidden_size,hidden_size)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self,x,hidden_state,cell_state):
        x=self.embedding(x).reshape(-1,1,self.hidden_size)
        x=F.relu(x)
        out,(hidden_state,cell_state)=self.lstm(x,(hidden_state,cell_state))
        out=self.fc(out).reshape(-1,self.output_size)
        return out,hidden_state,cell_state

    def init_hidden_and_cell(self,c,hid_z,cell_z):
        c=self.c_embedding(c).reshape(1,1,-1)
        hid_state=torch.cat((hid_z,c),dim=2)
        hid_state=self.hidden_fc(hid_state)

        cell_state=torch.cat((cell_z,c),dim=2)
        cell_state=self.cell_fc(cell_state)

        return hid_state,cell_state
```

Listing 4: CVAE decoder

```python
def decode(decoder,tokenizer,device,token,hid_z,cell_z,c,is_tf):
    in_token=torch.from_numpy(np.asarray(tokenizer.sos))
    in_token=in_token.to(device,dtype=torch.long)
    out_distribution=[]
    if token==None:
        max_len=30
    else:
        max_len=token.shape[0]-1

    hidden_state,cell_state=decoder.init_hidden_and_cell(c,hid_z,cell_z)
    for i in range(max_len):
        output,hidden_state,cell_state=decoder(in_token,hidden_state,cell_state)
        out_distribution.append(output)
        out_token=torch.max(torch.softmax(output,dim=1),1)[1]
        if out_token.item()==tokenizer.eos:
            break
        if is_tf==True:
            in_token=token[i+1]
        else:
            in_token=out_token
    out_distribution=torch.cat(out_distribution,dim=0).to(device)
    return out_distribution
```

Listing 5: The decoding process.

4.4 KL weight annealing

For monotonic annealing, in the first 20% of total epochs, the KL weight is set to 0, and after that, we increase linearly to the final KL weight, which we set as **0.2**. For cyclical annealing, we do the same thing as the monotonic method for each cycle.

```python
def klw_sched(anneal_method,cur_epoch,epochs,final_klw,anneal_cyc):
    if anneal_method=="monotonic":
        thres=0.2*epochs
        if cur_epoch<=thres:
            kl_w=0
        else:
            kl_w=final_klw*(cur_epoch-thres)/(epochs-thres)
    elif anneal_method=="cyclic":
        T=int(epochs/anneal_cyc)
        thres=int(T*0.2)
        cur_epoch%=T

        if cur_epoch<=thres:
            kl_w=0
        else:
            kl_w=final_klw*(cur_epoch-thres)/(T-thres)

    return kl_w
```

Listing 6: KL weight scheduler.

4.5 Teacher forcing ratio

In the first 20% of the total epochs, we set the teacher forcing ratio to 1. Furthermore, after that, we linearly decrease to final teacher forcing ratio, which we set to **0.8**.

```python
def tf_sched(cur_epoch,epochs,final_tf_ratio):
    thres=int(0.2*epochs)
    if cur_epoch<thres:
        tf_ratio=1
    else:
        tf_ratio=final_tf_ratio+
                 (epochs-cur_epoch)*(1-final_tf_ratio)/(epochs-thres)
    if tf_ratio>1:
        tf_ratio=1
    elif tf_ratio<final_tf_ratio:
        tf_ratio=final_tf_ratio
    return tf_ratio
```

Listing 7: Teacher forcing ratio scheduler.

4.6 Word generator

In word generation task, we random sample the latent z from $\mathcal{N}(0, 1)$. And for the four tenses, the condition c is range from 0...3. For each condition c, we give the decoder the latent z and the condition, then the decoder will generate a word with proper tense. We randomly generate 100 words, each with 4 tenses.

```
def gen_word(decoder,z_size,device,tokenizer):
    words_list=[]
    for i in range(100):
        hid_z,cell_z=gen_gauss_noise(z_size)
        hid_z=hid_z.to(device,dtype=torch.float)
        cell_z=cell_z.to(device,dtype=torch.float)
        words=[]
        for c in range(4):
            c=torch.from_numpy(np.asarray(c))
            c=c.to(device,dtype=torch.long)
            outputs=decode(decoder,tokenizer,device,None,hid_z,cell_z,c,
                           is_tf=False)
            out_token=torch.max(torch.softmax(outputs,dim=1),1)[1]
            out_word=tokenizer.inv_tokenize(out_token)
            words.append(out_word)
        words_list.append(words)
    return words_list
```

Listing 8: Word generation.

```
def gen_gauss_noise(z_size):
    hid_z=torch.normal(torch.zeros(1,1,z_size),torch.ones(1,1,z_size))
    cell_z=torch.normal(torch.zeros(1,1,z_size),torch.ones(1,1,z_size))
    return hid_z,cell_z
```

Listing 9: Sample from Gaussian.

4.7 Hyperparameters

- Final KL weight: 0.1.
- Learning rate: 0.05.
- Final teacher forcing ratio: 0.8.
- Epochs: 150.

5 Experimental Results

Some of the hyperparameters have been mentioned in Section 4. We provide the full lists here:

- Input size: 28.
- Hidden layer size: 256.
- Input condition size: 4.
- Condition hidden layer size: 8.
- Latent size: 32.

- Final KL weight: 0.1.
- Learning rate: 0.05.
- Final teacher forcing ratio: 0.8.
- Epochs: 150.

KL annealing	BLEU score	Gaussian score
monotonic	0.8312	**0.472**
cyclical	**0.9527**	0.452

Table 1: Best results of the two KL annealing method

5.1 Tense conversion

```
input: functioning
target: functioned
prediction: functioned

input: abet
target: abetting
prediction: abeting

input: functioning
target: function
prediction: function

input: split
target: splitting
prediction: splitting

input: abandon
target: abandoned
prediction: abandon

input: flared
target: flare
prediction: flan

input: begin
target: begins
prediction: begins

input: sent
target: sends
prediction: sents

Average BLEU-4 score: 0.8036
```

(a) Monotonic

```
input: sent
target: sends
prediction: sends

input: functioning
target: functioned
prediction: functioned

input: flared
target: flare
prediction: flare

input: abet
target: abetting
prediction: abetting

input: expend
target: expends
prediction: endeats

input: functioning
target: function
prediction: function

input: healing
target: heals
prediction: heals

input: split
target: splitting
prediction: splitting

Average BLEU-4 score: 0.9186
```

(b) Cyclical

Figure 2: Results of tense conversion.

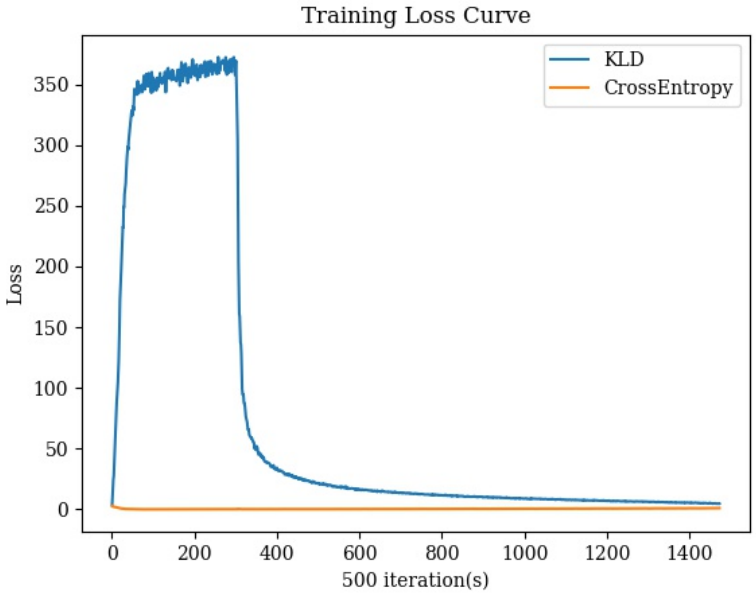
5.2 Word generation

(a) Monotonic

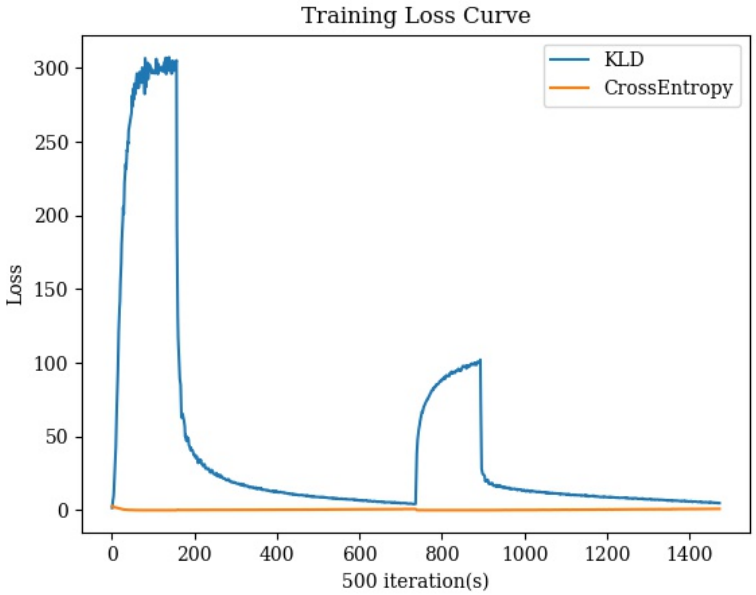
(b) Cyclical

Figure 3: Results of word generation.

5.3 Loss curve



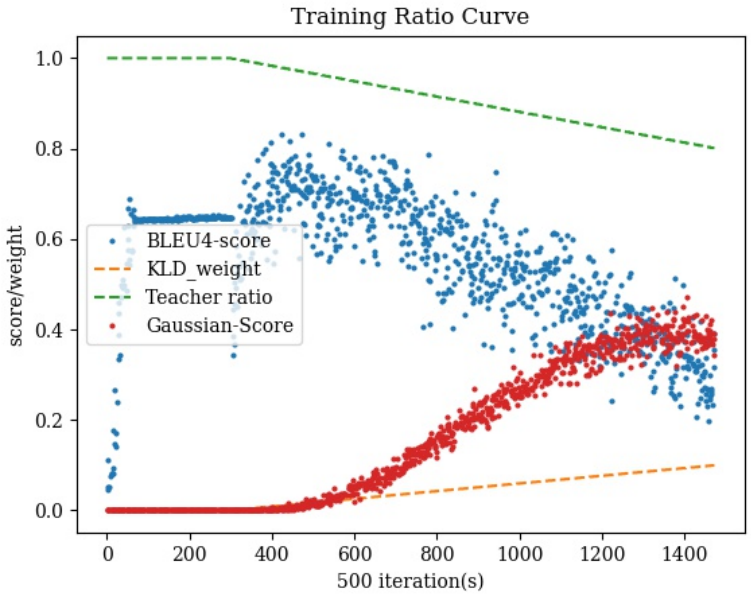
(a) Monotonic



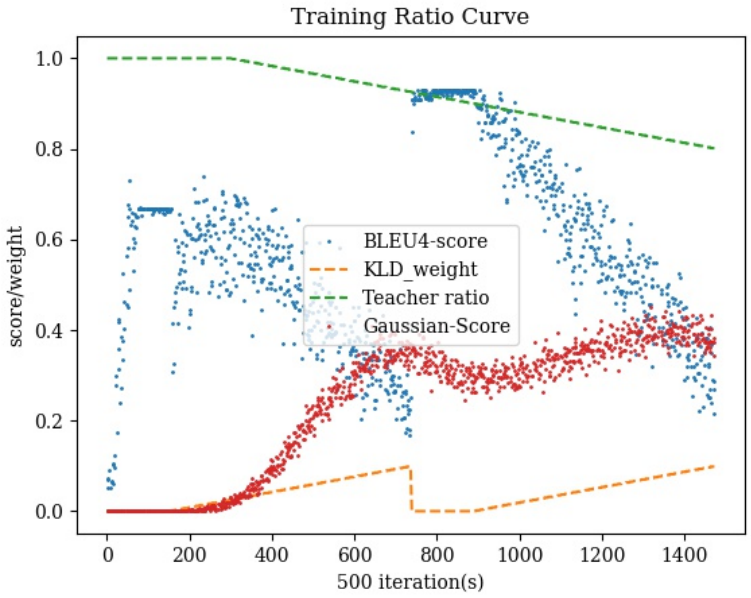
(b) Cyclical

Figure 4: Loss curve.

5.4 Ratio/score



(a) Monotonic



(b) Cyclical

Figure 5: Ratio and score curve.

6 Discussion

How does the teacher forcing ratio affect the results? We have tried out numerous teacher forcing ratios. In extreme case 0, the model performs poorly. When we set the teacher forcing ratio to 0, the model only learns from the output in the previous step, so if the output is incorrect, the results will always be wrong. In another extreme case 1, the bleu score rise to 78% then stopped ever since. Then we have tried 0.4, 0.6, and 0.8, and the bleu score increases when the teacher forcing ratio increases. The best bleu score we get is when the teacher forcing ratio is set to 0.8.

How does KLD weight affect the results? We found out that KLD weights have a significant effect on the Gaussian score. At the beginning of the training, we set the KLD weight to 0. In this phase, the Gaussian score never exceeds 0.02. Then after we start to raise the KLD weights, the Gaussian score rises as well. In KL cyclical annealing, KLD weight will drop back to 0 in every cycle. In this phase, the Gaussian score drops as well! KLD weight and bleu score work the opposite; when the KLD weight rises, the bleu score drops. We have tried 0.1,0.2,0.4 as our KLD weight, and we found out that 0.1 yields the best results. Since this is the regularization, we think 0.2 and 0.4 regularize too much, so the accuracy suffers.

Which KLD annealing method is better? No matter which method we choose, the Gaussian score can be around 0.4. However, the BLEU score is higher when using the cyclical annealing method, so we think the cyclical annealing method is better than the monotonic annealing method.