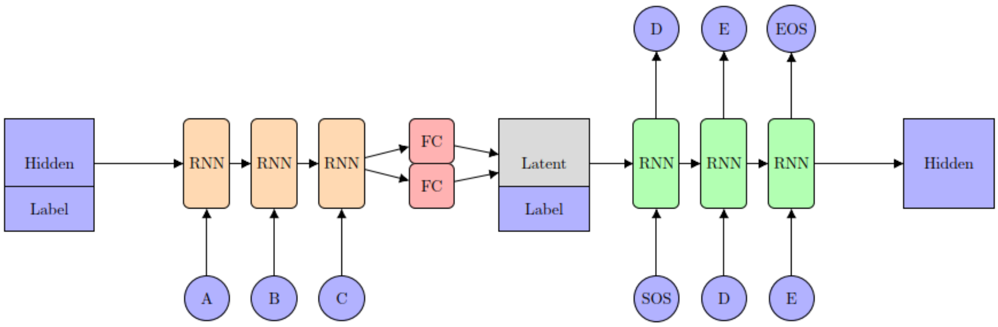
Lab 5 Conditional Sequence to Sequense VAE

1. Introduction

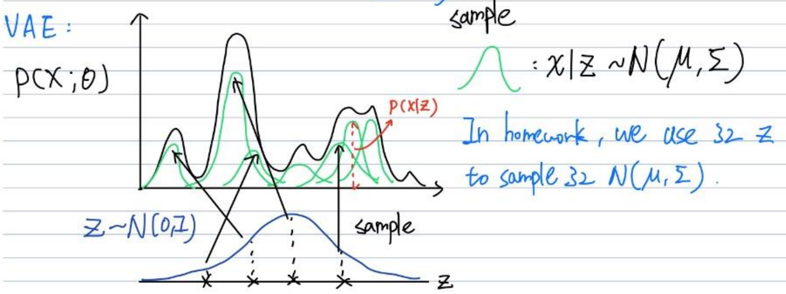
   Using CVAE to change the tense of the verbs , ex. abandon -> abandoned , input “abandon” to encoder with its tense(sp) , and use output and the target tense(p) as decoder input , and expect the verb “abandoned” as output .



2. Derivation of CVAE

Derivation of CVAE (VAE)

VAE:



$$\Rightarrow P(X;\theta) = \int_z P(z)\,P(X|z)\,dz$$, we expect to maximum $P(X;\theta)$

$\Rightarrow$ Maximum $\log P(X;\theta)$

latent vector

$$\log P(x;\theta) = \int_z q(z) \log P(x;\theta)\,dz$$

$$= \int_z q(z|x;\theta) \log\left(\frac{P(z,x;\theta)}{p(z|x;\theta)}\right) dz$$

$$= \int_z q(z|x;\theta) \log\left(\frac{P(z,x;\theta)}{q(z|x;\theta)}\,\frac{q(z|x;\theta)}{p(z|x;\theta)}\right) dz$$

$$= \int_z q(z|x;\theta) \log\left(\frac{P(z,x;\theta)}{q(z|x;\theta)}\right) dz + \int_z q(z|x;\theta) \log\left(\frac{q(z|x;\theta)}{P(z|x;\theta)}\right) dz$$

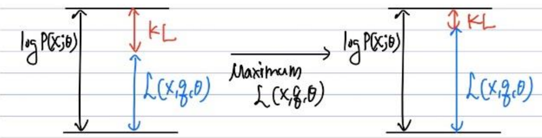
$$= \int_z q(z|x;\theta) \log\left(\frac{P(z,x;\theta)}{q(z|x;\theta)}\right) dz + KL\left(q(z|x;\theta) \,\|\, p(z|x;\theta)\right)$$

$$= \mathcal{L}(x,q,\theta) + \underbrace{KL\left(q(z) \,\|\, P(z|x;\theta)\right)}_{\geq 0}$$

$(q(z|x;\theta) = q(z))$ decoder output

$$\Rightarrow \log P(x;\theta) \geq \mathcal{L}(x, q, \theta) = \int_z q(z) \log\left(\frac{P(x|z;\theta)P(z)}{q(z)}\right) dz$$

$\Rightarrow$ Find $\overset{\text{decoder}}{P(x|z)}$ and $\overset{\text{encoder}}{q(z)}$ to maximum $\log P(x;\theta)$



Object function : $\mathcal{L}(x, q, \theta)$:

$$\mathcal{L}(x, q, \theta) = \int_z q(z) \log\left(\frac{P(x|z;\theta)P(z)}{q(z)}\right) dz$$

$$= \int_z q(z) \log(P(x|z;\theta))\, dz + \int_z q(z) \log\left(\frac{P(z)}{q(z)}\right) dz$$

$$= \int_z q(z|x;\theta) \log(P(x|z;\theta))\, dz - KL(q(z|x;\theta) \| P(z))$$

$$= \underbrace{E_{z\sim q(z|x;\theta)} \log P(x|z;\theta)}_{\text{maximum}} - \underbrace{KL(q(z|x;\theta) \| P(z))}_{\text{minimum}}$$

Change to CVAE, the object function is :

$$E_{z\sim q(z|x,c;\theta)} \log P(x|z,c;\theta) - KL(q(z|x,c;\theta) \| P(z|c))$$

3. Implementation details

Dataloader (dataloader.py):

```python
def Dataloader(path,mode='train'):
    file = open(path, "rt")
    line = file.readlines()

    tense2num = {'sp': 0, 'tp': 1, 'pg': 2, 'p': 3}
    condition_list = list(tense2num.values())

    alphabet2num = {'SOS': 0, 'EOS': 1}
    alphabet2num.update([(chr(i+97),i+2) for i in range(0,26)])

    input_batch = []
    cond_batch = []

    if mode == 'test':
      for i in range(4):
          cond_batch.append(condition_list[i])

    for s in line:
      # read word each line
      word = s.split('\n')[0].split(' ')
      if mode == 'train':
          tmp = [word[0], word[1], word[2], word[3]]
      else:
          tmp = [word[0]]
```

```python
        # encode word to numbers
        for n in range(len(tmp)):
            encode = []
            for id in tmp[n]:
              encode.append(alphabet2num[id])

            encode.append(1)
            input_batch.append(encode)

            if mode == 'train':
                cond_batch.append(condition_list[n])

    return input_batch, cond_batch
```

I encode the “SOS” , “EOS” and 26 alphabet to 0~27 (input size is 28) and tense encoded “sp” , “tp” , “pg” , “p” to 0~3 . My dataloader will return two object , “input_batch” , which contains 4908 verbs for training or 10 for testing ; “cond_batch” , 4908 tense for training or 10 for testing .

CVAE (model.py):

Encoder :

```python
class CVAE(nn.Module):
    class EncoderRNN(nn.Module):
        def __init__(self, input_size, input_cond_size, hidden_size):
            super(CVAE.EncoderRNN, self).__init__()
            self.hidden_size = hidden_size
            self.embedding_N = nn.Embedding(input_size, hidden_size)
            self.rnn_N = nn.LSTM(hidden_size, hidden_size)
            for weight in self.rnn_N.parameters():
                if len(weight.size()) > 1:
                    init.orthogonal_(weight.data)

        def initial(self):
            h = torch.zeros(1,1,self.hidden_size-8, device=device)
            c = torch.zeros(1,1,self.hidden_size, device=device)

            return h ,c

        def forward(self, input, hidden, cell):
            embedded = self.embedding_N(input).view(1, 1, -1)
            embedded = embedded.permute(1,0,2)

            output, (h, c) = self.rnn_N(embedded, (hidden, cell))

            return output, h, c
```

Includes one embedding layer and one LSTM layer , and I found that initial an orthogonal weight will get the better result .

Decoder :

```python
    class DecoderRNN(nn.Module):
        def __init__(self, input_size, input_cond_size, hidden_size):
            super(CVAE.DecoderRNN, self).__init__()
            self.hidden_size = hidden_size
            self.embedding_D = nn.Embedding(input_size, hidden_size)
            self.rnn_D = nn.LSTM(hidden_size, hidden_size)
            for weight in self.rnn_D.parameters():
                if len(weight.size()) > 1:
                    init.orthogonal_(weight.data)

        def initial(self):
            h = torch.zeros(1,1,self.hidden_size, device=device)
            c = torch.zeros(1,1,self.hidden_size, device=device)

            return h, c

        def forward(self,input, hidden, cell):
            embedded = self.embedding_D(input).view(1, 1, -1)
            embedded = embedded.permute(1,0,2)

            output, (h, c) = self.rnn_D(embedded, (hidden, cell))

            return output, (h, c)
```

Includes one embedding layer and one LSTM layer , and I found that initial an orthogonal weight will get the better result .

Other layers :

```python
    def __init__(self, input_size, input_cond_size, hidden_size):
        super(CVAE,self).__init__()
        self.input_size = input_size
        self.encoder = self.EncoderRNN(input_size, input_cond_size, hidden_size).cuda()
        self.decoder = self.DecoderRNN(input_size, input_cond_size, hidden_size).cuda()
        self.embedding_cond = nn.Embedding(input_cond_size, 8)
        self.fc_meam_h = nn.Linear(hidden_size, 32)
        self.fc_logvar_h = nn.Linear(hidden_size, 32)
        self.fc_st_D = nn.Linear(40, hidden_size)
        self.fc_out = nn.Linear(hidden_size, input_size)
```

self.embedding_cond : Embeds the conditional to size 8 .

self.mean_h : A fully connection layer to get size 32 mean from final hidden unit of encoder .

self.logvar_h : A fully connection layer to get size 32 logvar from final hidden unit of encoder .

self.fc_st_D : A fully connection layer to change the size of hidden unit put into the decoder at initial state from 32+8 to 256 .

self.fc_out : A fully connection layer to change the size of the output of decoder from 256 to 28(input size) .

Reparameterize :

```python
    def reparameterize(self, mean, var):
        std = torch.exp(0.5*var)
        eps = torch.randn_like(std)
        z = eps.mul(std).add_(mean)

        return z
```

Forward :

```python
    def forward(self, input_word, cond, teach_ratio, KLD_ratio, criterion, optimizer):
        loss = 0
        optimizer.zero_grad()

        # concatenate conditional to hidden unit 0
        Cond = self.embedding_cond(cond)
        hidden_n, cell_n = self.encoder.initial()
        hidden_n = torch.cat((hidden_n, Cond.view(1, 1, -1)),2)

        # Encoder
        for alphabet in input_word:
            _, hidden_n, cell_n = self.encoder(alphabet, hidden_n, cell_n)

        # get mean and variance
        h_t = torch.squeeze(hidden_n)
        mean_h = self.fc_meam_h(h_t)
        logvar_h = self.fc_logvar_h(h_t)

        KLD_loss = -0.5 * torch.sum(1 + logvar_h - mean_h.pow(2) - logvar_h.exp())

        # sample
        latent_h = self.reparameterize(mean_h, logvar_h)

        pred = None
        out_fc = None

        _, cell_d = self.decoder.initial()
        decoder_input = torch.tensor([[[0]]]).cuda()
```

```python
        # concatenate conditional to latent vector
        hidden_d = latent_h.view(1, -1)
        hidden_d = torch.cat((hidden_d, Cond),1)
        hidden_d = self.fc_st_D(hidden_d).view(1, 1, -1)

        # Decoder
        for idx in range(input_word.shape[0]):
            use_teacher_forcing = True if random.random() < teach_ratio else False
            if (use_teacher_forcing) and (idx !=0):
                output_d, (hidden_d, cell_d) = self.decoder(input_word[idx - 1], hidden_d, cell_d)
            else:
                output_d, (hidden_d, cell_d) = self.decoder(decoder_input, hidden_d, cell_d)


            out_fc = self.fc_out(output_d) # classification
            out_alphabet = torch.argmax(out_fc).item() # return to alphabet code
            decoder_input = torch.tensor([[[out_alphabet]]]).cuda()

            if idx == 0:
                pred = out_fc.view(1,self.input_size)
            else:
                pred = torch.cat((pred, out_fc.view(1,self.input_size)),0)


        loss = criterion(pred, input_word.long())
        loss += KLD_loss*KLD_ratio
        loss.backward()
        optimizer.step()

        return loss.item(), KLD_loss.item()
```

Evaluation (for bleu testing):

```python
    def Eval(self, input_word, cond1, cond2):
        with torch.no_grad():
            hidden_n, cell_n = self.encoder.initial()
            Cond1 = self.embedding_cond(cond1).view(1, 1, -1)
            hidden_n = torch.cat((hidden_n, Cond1),2)

            for alphabet in input_word:
                _, hidden_n, cell_n = self.encoder(alphabet, hidden_n, cell_n)

            h_t = torch.squeeze(hidden_n)
            mean_h = self.fc_meam_h(h_t)
            logvar_h = self.fc_logvar_h(h_t)

            latent_h = self.reparameterize(mean_h, logvar_h)

            pred_word = []
            out_fc = None
```

```python
            _, cell_d = self.decoder.initial()
            decoder_input = torch.tensor([[[0]]]).cuda()
            Cond2 = self.embedding_cond(cond2).view(1, 1, -1)
            hidden_d = latent_h.view(1, 1, -1)
            hidden_d = torch.cat((hidden_d, Cond2),2)
            hidden_d = self.fc_st_D(hidden_d).view(1, 1, -1)

            for i in range(input_word.shape[0]):
                output_d, (hidden_d, cell_d) = self.decoder(decoder_input, hidden_d, cell_d)
                out_fc =  self.fc_out(output_d)
                out_alphabet = torch.argmax(out_fc).item()
                if out_alphabet == 1:
                    break
                decoder_input = torch.tensor([[[out_alphabet]]]).cuda()
                pred_word.append(out_alphabet)

        return pred_word
```

Gaussian generation (In line 185 : torch.randn_like(torch.zeros(1, 1, 32)) is noise):

```python
180     def gaussion(self, Cond):
181         Result = []
182         with torch.no_grad():
183             for i in range(100):
184                 print(i,end='\r')
185                 latent = torch.randn_like(torch.zeros(1, 1, 32)).cuda()
186                 pred_batch = []
187 
188                 for cond in Cond:
189                     word = []
190                     _, cell_d = self.decoder.initial()
191                     out_fc = None
192                     decoder_input = torch.tensor([[[0]]]).cuda()
193                     Cond_ = self.embedding_cond(cond.long()).view(1, 1, -1)
194 
195                     hidden_d = latent.view(1, 1, -1)
196                     hidden_d = torch.cat((hidden_d, Cond_),2)
197                     hidden_d = self.fc_st_D(hidden_d).view(1, 1, -1)
```

```python
199                     for i in range(100):
200                         output_d, (hidden_d, cell_d) = self.decoder(decoder_input, hidden_d, cell_d)
201                         out_fc =  self.fc_out(output_d)
202                         out_alphabet = torch.argmax(out_fc).item()
203                         if out_alphabet == 1:
204                             break
205                         decoder_input = torch.tensor([[[out_alphabet]]]).cuda()
206                         word.append(out_alphabet)
207                     pred_batch.append(word)
208 
209                 Result.append(pred_batch)
210 
211         return Result
```

Test (test.py) : Test tense switching and gaussian score .

blue.py : compute bleu score .

main.py (for training and testing):

```python
parser = argparse.ArgumentParser(description='Set up')
parser.add_argument('--lr', type=float, default = 0.005)
parser.add_argument('--epochs', type=int, default = 500)
parser.add_argument('--hidden_size', type=int, default = 256)
parser.add_argument('--feq', type=float, default = 2.0)
parser.add_argument('--Mode', type=str, default = 'train')
args = parser.parse_args()
print(args)

if args.Mode == 'train':
    wandb.init(project='CVAE')
    wandb.save('/home/austin/nctu_hw/DL/DL_hw5/model.py')
    config = wandb.config
    config.hidden_size = args.hidden_size
    config.epochs = args.epochs
    config.learning_rate = args.lr

    model = CVAE(28, 4, args.hidden_size)
    model = model.cuda()
    wandb.watch(model)
    input_batch, cond_batch = Dataloader('/home/austin/nctu_hw/DL/DL_hw5/lab5_dataset/train.txt','train')
    Batch = []

    for i in range(len(input_batch)):
        input_batch[i] = torch.LongTensor(input_batch[i]).cuda()
        cond_batch[i]= torch.LongTensor([cond_batch[i]]).cuda()
        Batch.append([input_batch[i], cond_batch[i]]) # size : 4908

    criterion = torch.nn.CrossEntropyLoss().cuda()
    optimizer = optim.SGD(model.parameters(), lr=args.lr, momentum=0.9)
    BLeU = Bleu()
    test = Test()
```

```python
def gen_teach_ratio(Epoch):
    return 1.0 - (Epoch/args.epochs)
def sigmoid(x):
    return 1.0 / (1.0 + np.exp(-x))
def gen_KLD_ratio(mode, Epoch):
    if mode == 'mon':
        return (Epoch/args.epochs)*0.25
    else:
        period = args.epochs//args.feq
        Epoch %= period
        ratio = sigmoid((Epoch - period // 2.0) / (period // 10)) / 2.0
        return ratio*0.5
```

```python
    for epoch in range(args.epochs):
        print('Epoch : ', epoch+1 ,'...',end='\r')
        random.shuffle(Batch)
        Loss = []
        KLD_avg = []

        teach_ratio = gen_teach_ratio(epoch)
        KL_ratio = gen_KLD_ratio('cyc', epoch)

        for sample in Batch:
            loss, KLD = model(sample[0], sample[1], teach_ratio, KL_ratio, criterion, optimizer)
            Loss.append(loss)
            KLD_avg.append(KLD)

        CE_loss = sum(Loss)/len(Loss) - (sum(KLD_avg)/len(KLD_avg))*KL_ratio
        model.eval()
        test.bleU_test(model)
        model.train()
        score = BLeU.get_score()

        wandb.log({"teach_ratio": teach_ratio})
        wandb.log({"BLeU": score})
        wandb.log({"KL_ratio": KL_ratio})
        wandb.log({"KLD": sum(KLD_avg)/len(KLD_avg)})
        wandb.log({"loss": sum(Loss)/len(Loss)})
        wandb.log({"CE_loss": CE_loss})
        file = open('/home/austin/nctu_hw/DL/DL_hw5/Record.txt', "a+")
        file.write(str(teach_ratio)+'/'+str(KL_ratio)+'/'+str(sum(Loss)/len(Loss))+'/'+str(CE_loss)+'/'+str(sum(KLD_avg)/len(KLD_avg))+'/'+str(score)+'\n')

        print('Epoch : ',epoch+1,' Loss : ',sum(Loss)/len(Loss),' CE_Loss :', CE_loss, 'KLD_loss : ',sum(KLD_avg)/len(KLD_avg), ' Bleu : ', score)
        if score >= 0.7:
            name = '/home/austin/nctu_hw/DL/DL_hw5/weight/CVAE_'+str(epoch+1)+'_'+str(sum(Loss)/len(Loss))+'_'+str(score)+'.pth'
            torch.save(model.state_dict(), name)
            wandb.save(name)
```

```python
else:
    model = CVAE(28, 4, args.hidden_size)
    model.load_state_dict(torch.load('/home/austin/nctu_hw/DL/DL_hw5/weight/CVAE_45_0.38194182919856_0.8323583241361134.pth'))
    model = model.cuda()
    model.eval()
    score = 0
    BLeU = Bleu()
    test = Test()
    test.gaussian_test(model)
    for i in range(1000):
        print(i,end='\r')
        test.bleU_test(model)
        score = BLeU.get_score()
        if score >= 0.8:
            print('Bleu score : ', score)
            break
```

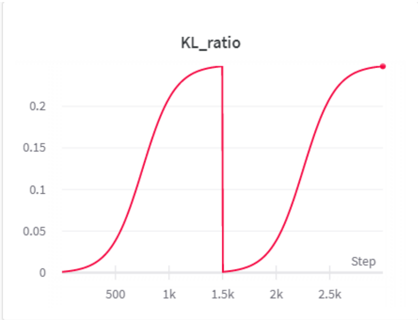
Hyperparameters :

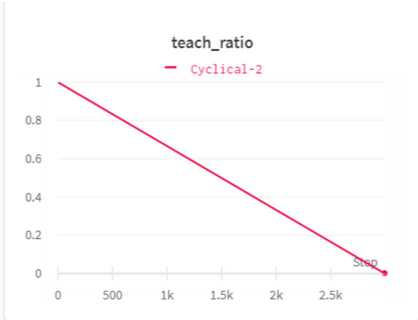
Learning rate : 0.005 , Epochs : 500

I shuffle the training data with each epoch .

KL weight : sigmoid with two cycles (max : 0.25)

Teach : from 1 down to 0 .

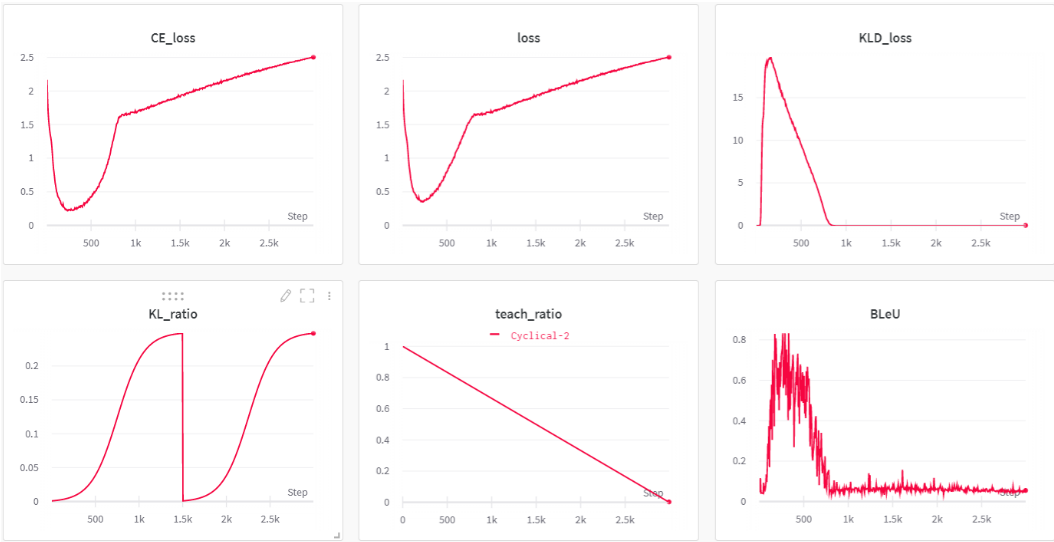




4. Result and discussion

(I use the model weight saved at 45 epochs .):

```
austin@austin: ~/nctu_hw/DL/DL_hw5 63x21
austin@austin:~/nctu_hw/DL/DL_hw5$ python3 main.py --Mode=test
Namespace(Mode='test', epochs=500, feq=2.0, hidden_size=256, lr
=0.005)
Gaussian score :  0.36
Bleu score :  0.8323583241361134
austin@austin:~/nctu_hw/DL/DL_hw5$
```

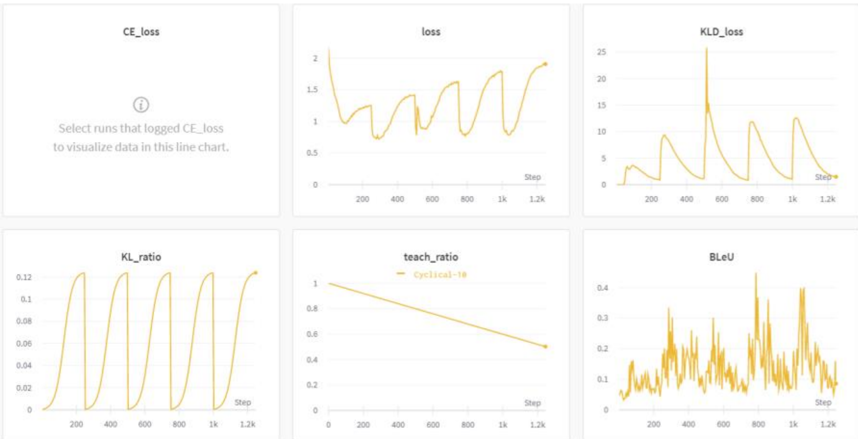


Bleu / Gaussian example :

```
1  abandone
2  abett
3  begins
4  expends
5  sense
6  splitt
7  flare
8  function
9  functioned
10 heals
```

```
1  indicate
2  indires
3  indicating
4  indicated
5  =============
6  thore
7  tolaches
8  belchong
9  thoodwed
10 =============
11 teaan
12 teaans
13 prattling
14 teaandoned
15 =============
16 oather
17 oathers
18 oathering
19 oathered
20 =============
```

I found that the bleu score is positively related to KLD loss , the two curves are very similar . In the early stages of training , KL weight was very low , and reconstruction loss was keep going down . When the KL weight up to 0.002 , reconstruction loss started to go up , and made the KLD loss and bleu down . I assume that when the KLD loss low enough , the reconstruction loss will keep down , so I set the max value of kl weight only 0.25 to ensure that the reconstruction loss drops enough to get high bleu score . For learning rate , because I update the model for every words in one epoch , It is better to use lower rate , and I find that 0.005 can make it almost drop vertically . For teach ratio , the higher ratio at the early stages can make loss drop steadily , so simply using monotonic mode . I found the issue that in the second cycle of kl weight , when it went down again , it doesn't make reconstruction loss down , and KLD loss doesn't go up again too . So I redo the experiment , this time I set 10 cycles of kl weight :



Although my computer was shutdown when epoch closed to 250 , only half of data , but it still can prove that my assumption is correct : When the KLD loss low enough , the reconstruction loss will keep down . And take a look at higher cycles , it makes the bleu score drops before it has time to rise enough , but the highest score in each cycle has an upward trend .

The curve of using monotonic KL weight (from 0 down to 0.25) :