DL HW5

1. Introduction

In this assignment, we should convert the tense of word by Conditional Sequence-to-sequence VAE. Auto encoder can encode the input to a hidden latent and decode to an output that approximates to the input. The difference between auto encoder and VAE(Variational Autoencoder) is the distribution of hidden latent in VAE should approximate Gaussian distribution. KL loss is a regularization term to reach this goal. KLD weight determines the power of KL loss. With this limitation, we can use arbitrary Gaussian noise as hidden latent. Feeding the hidden latent to the decoder, it can generate new words. Conditional VAE adds the additional condition to the encoder and decoder. The condition would concat with the input. The decoder should output the words depends on the condition.

2. Derivation of CVAE

I derive VAE at first, CVAE is easy to solve with VAE

$P(x,z) = p(x)p(z|x)$

$\Rightarrow \log P(x;\theta) = \log p(x,z;\theta) - \log p(z|x;\theta)$

integrate over z with a $q_\phi(z)$, $\begin{cases} q_\phi(z) \geq 0 \ \forall z \\ \int q_\phi(x)dz = 1 \end{cases}$

$\int q_\phi(z)\log p(x;\theta)dz$

$= \int q_\phi(z)\log p(x,z;\theta)dz - \int q_\phi(z)\log p(z|x;\theta)dz$

$= \int q_\phi(z)\log p(x,z;\theta)dz - \int q_\phi(z)\log q_\phi(z)dz + \int q_\phi(z)\log q_\phi(z)dz - \int q_\phi(z)\log p(z|x;\theta)dz$

$\Rightarrow \log p(x;\theta) = L(x,q_\phi,\theta) + KL(q_\phi(z)\|p(z|x;\theta))$ KL divergence

$L(x,q_\phi,\theta) = \int q_\phi(z)\log p(x,z;\theta)dz - \int q_\phi(z)\log q_\phi(z)dz$

$KL(q_\phi(z)\|p(z|x;\theta)) = \int q_\phi(z)\log \frac{q_\phi(z)}{p(z|x;\theta)}dz$

change $q_\phi(z|x;\theta')$

$\Rightarrow L(x,q_\phi,\theta) = E_{z\sim q_\phi(z|x;\theta')}\log p(x|z;\theta) + E_{z\sim q_\phi(z|x;\theta')}\log p(z)$

$- E_{z\sim q_\phi(z|x;\theta')}\log q_\phi(z|x;\theta')$

$= E_{z\sim q_\phi(z|x;\theta')}\log p(x|z;\theta) - KL(q_\phi(z|x;\theta')\|p(z))$

Both are assumed to Gaussian

change to CVAE

$\Rightarrow L(x,q_\phi,\theta|c) = E_{z\sim q_\phi(z|x,c;\theta')}\log p(x|z,c;\theta) - KL(q_\phi(z|x,c;\theta')\|p(z|c))$ #

3. Implementation details

A. Describe how you implement your model

- Dataloader

How dataloader converts words to tensor is similar to the lab4. I assign each alphabet different id. I combine different tense of the word. There are 12 combinations for one word. Each pair has tensor of word and tensor of tense. I set the four tense *sp*, *tp*, *pg*, *p* to 1, 2, 3, 4 respectively. For example, a pair is like (‘conduct’ ,‘conducts’, 0, 1) but the format are all convert to tensor. Word also convert to the number list.

In test.txt, there is no specific tense for each words, so I add the corresponding tense numbers to the file. (corresponding tense refers readme.txt)

```
abandon abandoned 0 3
abet abetting 0 2
begin begins 0 1
expend expends 0 1
sent sends 3 1
split splitting 0 2
flared flare 3 0
functioning function 2 0
functioning functioned 2 3
healing heals 2 1
```

```python
def dataloader(mode):
    if mode == 'train':
        data_word = []
        index = np.delete(np.array(np.meshgrid([0, 1, 2, 3], [0, 1, 2, 3])).T.reshape(-1,2), [0, 5, 10, 15], axis=0)

        with open('./dataset/train.txt') as f:
            for line in f.readlines():
                word_list = line.strip().split(' ')
                tense_list = []
                for word in word_list:
                    word_input = torch.from_numpy(np.append(str_to_int(word), EOS_token)).view(-1, 1)
                    # word_input = word
                    tense_list.append(word_input)

                data_word = build_pairs(data_word, tense_list, index)

        return data_word
    else:
        with open('./dataset/test.txt') as f:
            data_word = []
            origin = []
            for line in f.readlines():
                word_list = line.strip().split(' ')
                word1 = word_list[0]
                word2 = word_list[1]
                tense1 = int(word_list[2])
                tense2 = int(word_list[3])
                origin.append((word1, word2, tense1, tense2))
                word1_tenser = torch.from_numpy(np.append(str_to_int(word1), EOS_token)).view(-1, 1)
                word2_tenser = torch.from_numpy(np.append(str_to_int(word2), EOS_token)).view(-1, 1)
                data_word.append((word1_tenser, word2_tenser, torch.tensor(tense1), torch.tensor(tense2)))

        return data_word, origin
```

- Encoder

The structure is similar to the lab4. I replace the GRU with LSTM. LSTM needs additional cell state to remember long term memory. I embeds the alphabet to size 256 as input to encoder. I also embeds the tense to size 128. Then concat embedded tense and size 128 zeros as the initial hidden state to encoder.

```
def initHidden(self, tense):
    cond = self.embedding_tense(tense).view(1, 1, -1)

    return torch.cat((torch.zeros(1, 1, int(self.hidden_size/2)).cuda(GPUID[0]), cond), 2)
```

```
class EncoderRNN(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(EncoderRNN, self).__init__()
        self.hidden_size = hidden_size

        self.embedding_word = nn.Embedding(input_size, hidden_size)
        self.embedding_tense = nn.Embedding(input_size, int(hidden_size/2))

        self.lstm = nn.LSTM(hidden_size, hidden_size)
        self.fc = nn.Linear(hidden_size,int(hidden_size/2))
        self.fcVar = nn.Linear(hidden_size,int(hidden_size/2))
```

In the last of the encoder, I add fully connected layer to produce mean and variance that represents the distribution of the latent. The size of mean and variance is 128.

```
def forward(self, input, hidden, c_state, last=False):
    if last:
        embedded = self.embedding_word(input).view(1, 1, -1)
        output = embedded
        output, (hidden, c_state) = self.lstm(output, (hidden, c_state))
        mu = self.fc(hidden)
        logvar = self.fcVar(hidden)
        return mu, logvar

    embedded = self.embedding_word(input).view(1, 1, -1)
    output = embedded
    output, (hidden, c_state) = self.lstm(output, (hidden, c_state))
    return output, hidden, c_state
```

- Decoder

The structure is similar to lab4. I replace GRU with LSTM. I concat the noise z with the embedded tense as the initial hidden state to decoder.

```python
#Decoder
class DecoderRNN(nn.Module):
    def __init__(self, hidden_size, output_size):
        super(DecoderRNN, self).__init__()
        self.hidden_size = hidden_size

        self.embedding_word = nn.Embedding(output_size, hidden_size)
        self.embedding_tense = nn.Embedding(4, int(hidden_size/2))

        self.lstm = nn.LSTM(hidden_size, hidden_size)
        self.out = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden, c_state):
        output = self.embedding_word(input).view(1, 1, -1)
        output = F.relu(output)
        output, (hidden, c_state) = self.lstm(output, (hidden, c_state))
        output = self.out(output[0])
        return output, hidden, c_state

    def initHidden(self, z, tense):
        cond = self.embedding_tense(tense).view(1, 1, -1)

        return torch.cat((z.cuda(GPUID[0]), cond), 2)
```

- Training process

  First, I initialize the *encoder_hidden* and *encoder_c_state* by *initHidden* function of encoder above. I take one alphabet as the input to the encoder sequentially. Previous hidden layer and c_state from encoder will be the next states to the encoder. Finally, *mu* and *logvar* represents the latent of the whole input word.

```python
def train(input_tensor, target_tensor, input_tense, target_tense, encoder, decoder, encoder_optimizer, decoder_optimizer, criterion, KLD_weight, max_length=MAX_LENGTH):
    encoder_hidden = encoder.initHidden(input_tense.cuda(GPUID[0]))
    encoder_c_state = encoder_hidden

    encoder_optimizer.zero_grad()
    decoder_optimizer.zero_grad()

    input_length = input_tensor.size(0)
    target_length = target_tensor.size(0)

    cross_entropy_loss = 0

    #----------sequence to sequence part for encoder----------#
    for ei in range(input_length):
        if input_tensor[ei] == 1:
            mu, logvar = encoder(input_tensor[ei].cuda(GPUID[0]), encoder_hidden.cuda(GPUID[0]), encoder_c_state.cuda(GPUID[0]), True)
        else:
            encoder_output, encoder_hidden, encoder_c_state = encoder(input_tensor[ei].cuda(GPUID[0]), encoder_hidden.cuda(GPUID[0]), encoder_c_state.cuda(GPUID[0]))
```

I produce Gaussian noise randomly. Then scale the noise by *mu* and *logvar* from encoder.

$$z = N(0,I) \times \Sigma(x) + \mu(x)$$

```python
def get_z_encode(mu, logvar):
    std = logvar.mul(0.5).exp_()
    eps = torch.randn(std.size(0), std.size(1))
    z = eps.mul(std).add_(mu)
    return z
```

I initialize the de*coder_hidden* and *decoder_c_state* by *initHidden* function of decoder above. Noise *Z* would concat with embedded tense. I take one alphabet as the input to the decoder sequentially. Previous hidden layer and c_state from decoder will be the next states to the decoder. If using teacher forcing, the input to the decoder will be ground truth. if not, the input to the decoder will be the previous output of decoder.

```python
z = get_z_encode(mu.cpu(), logvar.cpu())

decoder_input = torch.tensor([[SOS_token]])
decoder_hidden = decoder.initHidden(z.cuda(GPUID[0]), target_tense.cuda(GPUID[0]))
decoder_c_state = decoder_hidden

use_teacher_forcing = True if random.random() < teacher_forcing_ratio else False

#----------sequence to sequence part for decoder----------#
if use_teacher_forcing:
    # Teacher forcing: Feed the target as the next input
    for di in range(target_length):
        decoder_output, decoder_hidden, decoder_c_state = decoder(
            decoder_input.cuda(GPUID[0]), decoder_hidden.cuda(GPUID[0]), decoder_c_state.cuda(GPUID[0]))
        cross_entropy_loss += criterion(decoder_output, target_tensor[di])
        decoder_input = target_tensor[di]  # Teacher forcing

else:
    # Without teacher forcing: use its own predictions as the next input
    for di in range(target_length):
        decoder_output, decoder_hidden, decoder_c_state = decoder(
            decoder_input.cuda(GPUID[0]), decoder_hidden.cuda(GPUID[0]), decoder_c_state.cuda(GPUID[0]))
        topv, topi = decoder_output.topk(1)
        decoder_input = topi.squeeze().detach()  # detach from history as input

        cross_entropy_loss += criterion(decoder_output, target_tensor[di])
        if decoder_input.item() == EOS_token:
            break
```

Loss is z_kl_loss and cross_entropy loss. KLD weight determines the power of KL loss.

$$kl\_loss = -\frac{1}{2}[\sum_i (log\sigma_i^2 + 1) - \sum_i \mu^2 - \sum_i \sigma_i^2]$$

```python
z_kl_loss = -0.5 * torch.sum(1 + logvar.cuda(GPUID[0]) - mu.cuda(GPUID[0]).pow(2) - logvar.cuda(GPUID[0]).exp())*KLD_weight
ce_weight = 1
loss = z_kl_loss + cross_entropy_loss*ce_weight

loss.backward()

encoder_optimizer.step()
decoder_optimizer.step()

return cross_entropy_loss.item() / target_length, cross_entropy_loss*ce_weight, z_kl_loss, loss
```

- KLD weight

If cyclical mode, KLD_weight starts with 0, grow up to 1 within certain number of epoch/iterator. When it reach to 1, it remains for the same number of epoch/iterator and becomes 0 again. The cycle continues to the end.

If monotonic mode, KLD_weight starts with 0, grow up to 1 within certain number of epoch/iterator. When it reach to 1, it remains at 1 to the end. It won't change any more. It's the difference from the cyclical mode.

```
if kl_mode == 'c':
    KLD_decay = 2
    if kl_count < KLD_decay:
        KLD_weight += (1/KLD_decay)
        kl_count += 1
    elif kl_count == KLD_decay*2:
        KLD_weight = 0
        kl_count = 0
    else:
        KLD_weight = 1
        kl_count += 1
elif kl_mode == 'm':
    KLD_decay = 375000
    if iter*epoch < KLD_decay:
        KLD_weight += (1/KLD_decay)
else:
    print("wrong mode")
    exit(1)
```

- Evaluation - BLUE-4 score

In the evaluation part, I feeds *input_tenser* and *input_tense* to encoder. I produce z of *mu* and *logvar* from encoder. Then I feeds *Z* and *target_tenser* to decoder.

```
def test(input_tensor, input_tense, target_tense, encoder, decoder, max_length=MAX_LENGTH):
    encoder_hidden = encoder.initHidden(input_tense.cuda(GPUID[0]))
    encoder_c_state = encoder_hidden

    input_length = input_tensor.size(0)


    #----------sequence to sequence part for encoder----------#
    for ei in range(input_length):
        if input_tensor[ei] == 1:
            # print(input_tensor[ei])
            mu, logvar = encoder(input_tensor[ei].cuda(GPUID[0]), encoder_hidden.cuda(GPUID[0]), encoder_c_state.cuda(GPUID[0]), True)
        else:
            encoder_output, encoder_hidden, encoder_c_state = encoder(input_tensor[ei].cuda(GPUID[0]), encoder_hidden.cuda(GPUID[0]), \
                                                                      encoder_c_state.cuda(GPUID[0]))

    z = get_z_encode(mu.cpu(), logvar.cpu())

    decoder_input = torch.tensor([[SOS_token]])
    decoder_hidden = decoder.initHidden(z.cuda(GPUID[0]), target_tense.cuda(GPUID[0]))
    decoder_c_state = decoder_hidden
```

The length of output word is unknown. I set the length to max_length by default. If output of decoder is EOS_token, it means the tail of the word and the decode process ends. The inputs to the decoder are all the previous output of the decoder except the first one (SOS_token). I don't use any target tensor in this part.

```
    #----------sequence to sequence part for decoder----------#

    pred = []
    # Without teacher forcing: use its own predictions as the next input
    for di in range(max_length):
        decoder_output, decoder_hidden, decoder_c_state = decoder(
            decoder_input.cuda(GPUID[0]), decoder_hidden.cuda(GPUID[0]), decoder_c_state.cuda(GPUID[0]))
        topv, topi = decoder_output.topk(1)
        decoder_input = topi.squeeze().detach()  # detach from history as input

        if decoder_input.item() == EOS_token:
            break
        pred.append(decoder_input.cpu().numpy())

    return int_to_str(pred)
```

- Evaluation - Gaussian score

I produce a Gaussian noise randomly by *torch.randn,* and then concat Z and the embedded tense. I feeds these to decoder and get the word with tense.

```
def word_generation(encoder, decoder, max_length=MAX_LENGTH):
    z = torch.randn(1, 1, 128)
    pred_list = []
    for tense in range(4):
        target_tense = torch.tensor(tense)

        decoder_input = torch.tensor([[SOS_token]])
        decoder_hidden = decoder.initHidden(z.cuda(GPUID[0]), target_tense.cuda(GPUID[0]))
        decoder_c_state = decoder_hidden

        #----------sequence to sequence part for decoder----------#

        pred = []
        # Without teacher forcing: use its own predictions as the next input
        for di in range(max_length):
            decoder_output, decoder_hidden, decoder_c_state = decoder(
                decoder_input.cuda(GPUID[0]), decoder_hidden.cuda(GPUID[0]), decoder_c_state.cuda(GPUID[0]))
            topv, topi = decoder_output.topk(1)
            decoder_input = topi.squeeze().detach()  # detach from history as input

            if decoder_input.item() == EOS_token:
                break
            pred.append(decoder_input.cpu().numpy())
        pred_list.append(int_to_str(pred))

    return pred_list
```

B. Specify the hyperparameters

Each epoch: 75000 iterators

The number of epoch: 32

Teacher forcing ratio: 1

Learning rate: 0.01 with 10e-4 decay

KL weight:

cyclical -

reach to 1 within 5000 iterators, grow up linearly (1/5000 each iterator) and remain at 1 for 5000 iterators.

4. Results and discussion

A. Show your results and Plot the loss

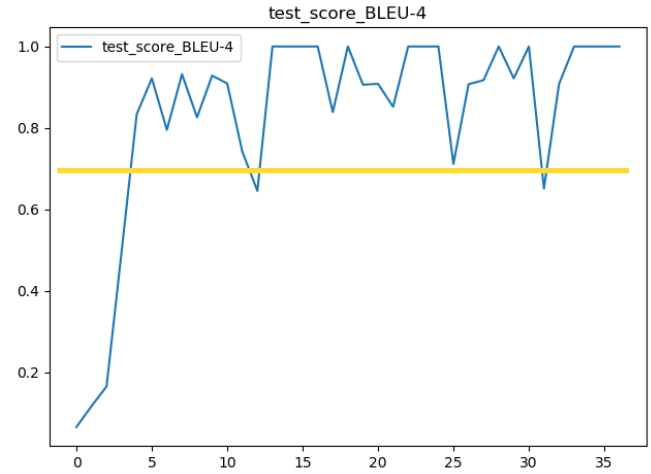
- tense conversion

```
==========================================
input:abandon
target:abandoned
pred:abandoned
==========================================
input:abet
target:abetting
pred:abetting
==========================================
input:begin
target:begins
pred:begins
==========================================
input:expend
target:expends
pred:expends
==========================================
input:sent
target:sends
pred:sends
==========================================
input:split
target:splitting
pred:splitting
==========================================
input:flared
target:flare
pred:flare
==========================================
input:functioning
target:function
pred:function
==========================================
input:functioning
target:functioned
pred:functioned
==========================================
input:healing
target:heals
pred:heals
BLEU-4 score:1.000000
```

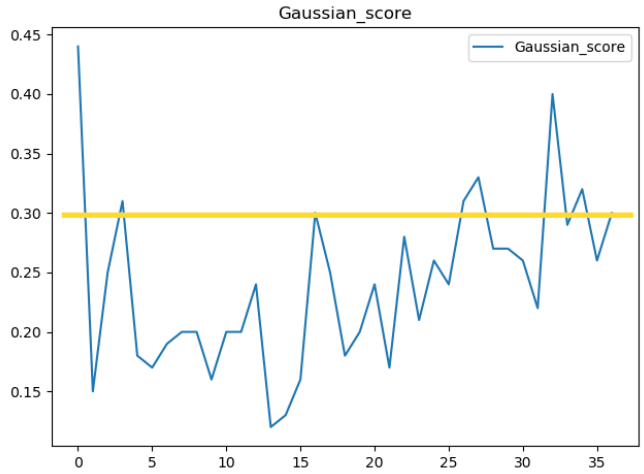
- tense generation

```
['pray', 'prays', 'prupting', 'prayed']
['beat', 'beats', 'becating', 'becated']
['deny', 'denies', 'dining', 'dined']
['arch', 'arches', 'arching', 'accepted']
['illumine', 'illumines', 'illumining', 'illumined']
['tat', 'tats', 'tating', 'tated']
['single', 'exists', 'instituting', 'existed']
['assomb', 'assombs', 'assominating', 'assombed']
['esteem', 'esteems', 'esteeming', 'esteemed']
['spur', 'spurs', 'spurring', 'spurred']
['retreat', 'retreats', 'retreating', 'retreated']
['obew', 'obews', 'obewing', 'obewed']
['spolute', 'spolls', 'spolling', 'spolted']
['hinder', 'hinders', 'hindering', 'hindered']
['wow', 'wows', 'wowing', 'wow']
['snatch', 'snatches', 'snatching', 'snatched']
['receme', 'recemes', 'receming', 'recrewed']
['abound', 'abounds', 'abounding', 'abounded']
['summon', 'summons', 'summoning', 'summoned']
['scramble', 'scrambles', 'scrambling', 'scrambled']
['toss', 'tosses', 'tost', 'tossed']
['direct', 'directs', 'directing', 'directed']
['llow', 'llots', 'llotting', 'llotted']
['tewer', 'tewers', 'tewering', 'thered']
['distract', 'distracts', 'distracting', 'distracted']
['get', 'gets', 'gettering', 'gestered']
['cqulve', 'calcuses', 'calculating', 'culculated']
['compromise', 'compromises', 'compromising', 'crowled']
['pray', 'prays', 'praying', 'prayed']
['blend', 'slins', 'blinging', 'blew']
['issent', 'issests', 'yielding', 'issented']
['inhibit', 'inhibits', 'inhibiting', 'inhibited']
['blow', 'blows', 'blowing', 'blent']
['trudge', 'trudges', 'trudging', 'travelled']
['pinish', 'pinies', 'pinying', 'punsed']
['leave', 'leaves', 'leaving', 'lefl']
['villeng', 'villes', 'villenging', 'villed']
['blink', 'blinks', 'blinking', 'blinked']
['recur', 'recurs', 'recurring', 'recurred']
['tate', 'tates', 'tating', 'took']
['cease', 'ceases', 'ceasing', 'ceased']
['stammer', 'stammers', 'stammering', 'stammered']
['strew', 'strewes', 'strewing', 'strewed']
['intervene', 'intervenes', 'intervening', 'intervened']
['cussip', 'cussipes', 'cussiping', 'cussiped']
Gaussian score: 0.32
```

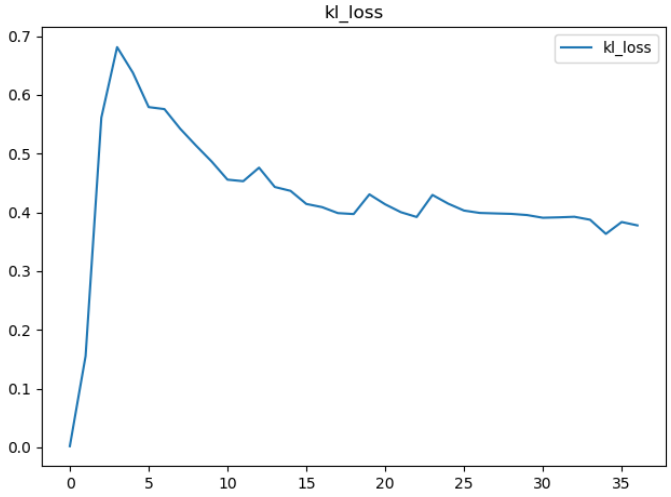
- Test score BLEU-4



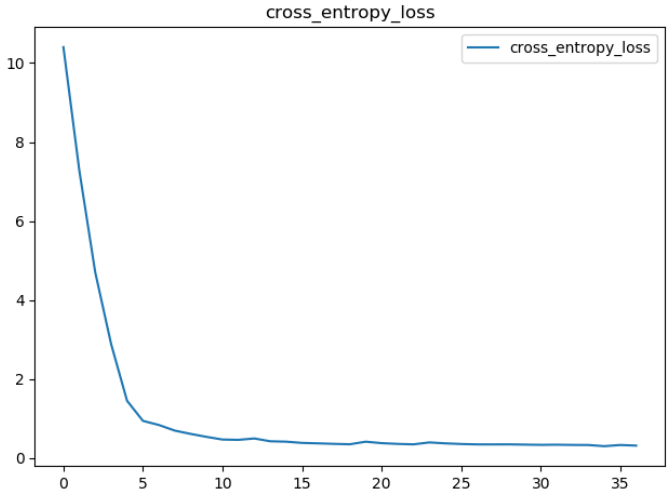
- Gaussian score



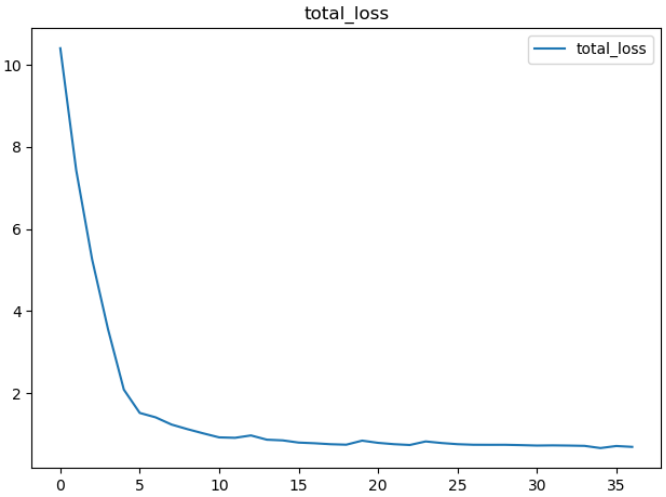
- KL loss



- Cross entropy loss



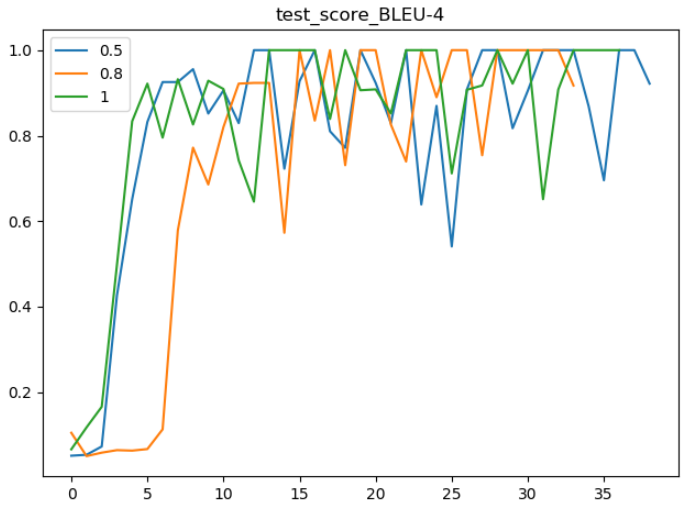
- Total loss

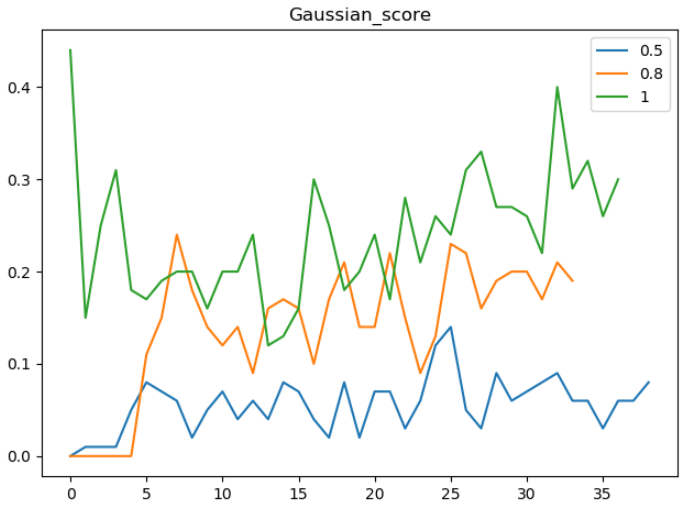


B. Discuss the results

- Teacher forcing ratio

With the experience of lab4, I set the teacher forcing to 0.5 at first. I think this is the best ratio for learning. Therefore, most of experiment are training with ratio 0.5. However, when I try to compare the result of different teacher forcing ratio, I find that ratio = 1 are better than 0.5. The following is the score comparison.



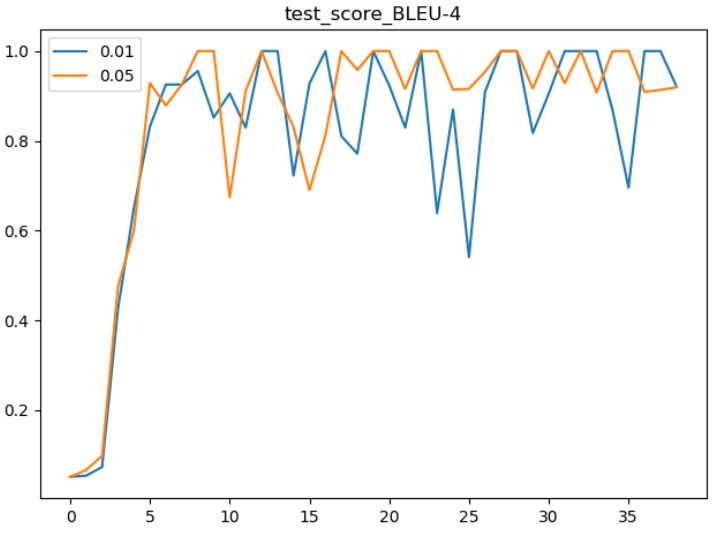


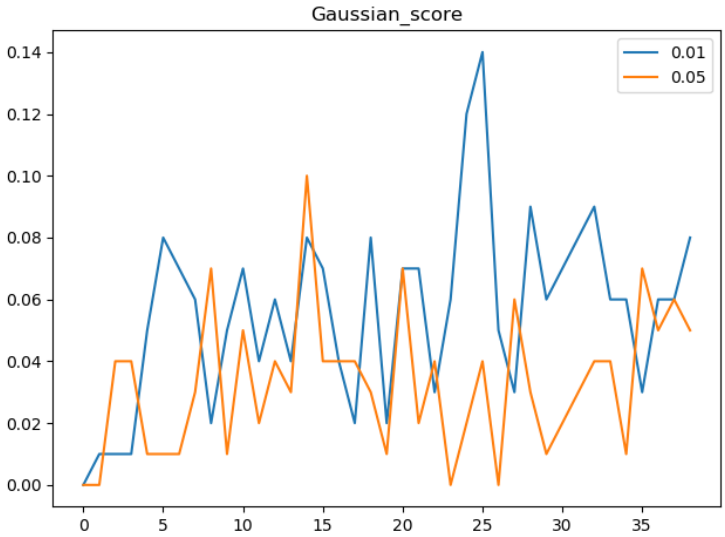
Generally, model with ratio 0.5 would have better adaptability in testing. But in test score, the trend of 1 and 0.5 is similar. I think the reason is the input word and output word is very similar in this task. Most case only change the tail of the word. Therefore, although we never take the previous output of decoder as next input. The model tends to produce the word just like the word of encoder.

In Gaussian score, it's reasonable that ratio 1 > 0.8 > 0.5. This score count how many generating words match words in training data. This means not only the model should produce correct four tense words, but also the word should appear in training data. For higher teacher forcing ratio, the model learns the training data more clearly.

- Learning rate

With the experience of lab4, learning rate 0.01 is better than default 0.05. Thus, I try two setting and observe the result. Both teacher forcing rate are 0.5. The following images is the score comparison.

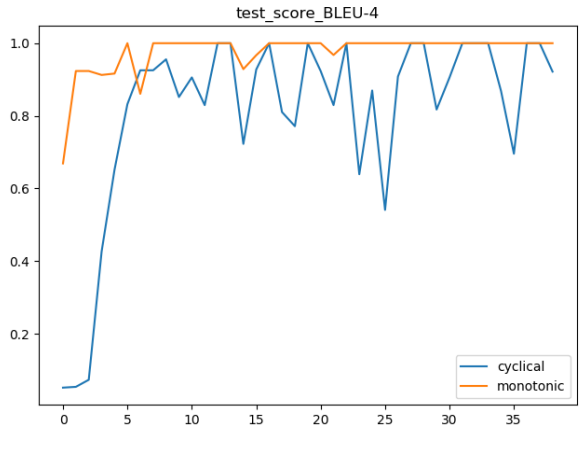


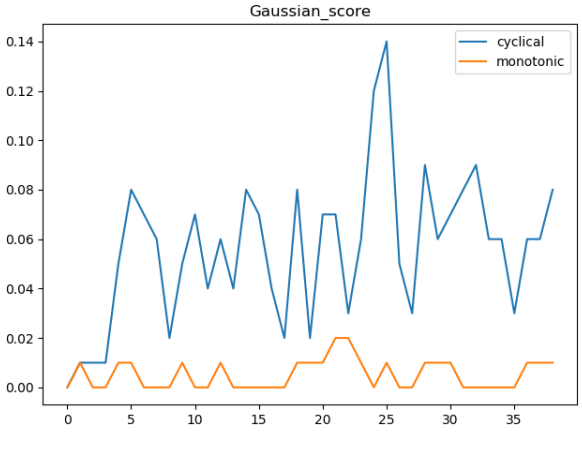


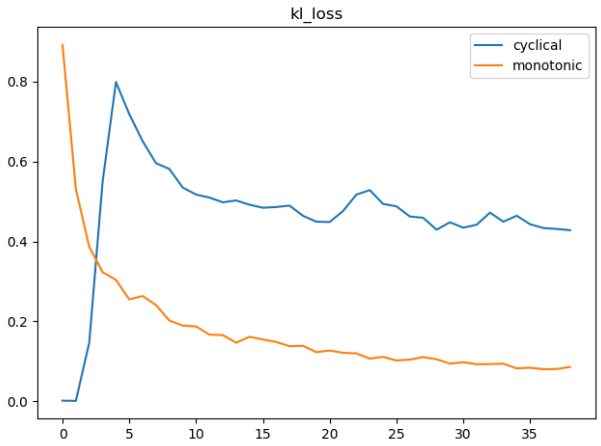
Roughly, the performance of two models is similar. If looking at more details, we can find the trends of 0.01 in Gaussian score climbs up slightly, and 0.05 just fluctuates around 0.02 to 0.08. I think that means 0.01 has more chance to find the better local minimum.

- KLD weight

I find model is sensitive to KLD weight in monotonic than cyclical. KLD weight grows up linearly. If the target iterator that should reach to 1 is too big. It grows up with slow speed and KLD weight is close 0. The model doesn't care about the KL loss. The following images shows comparison.

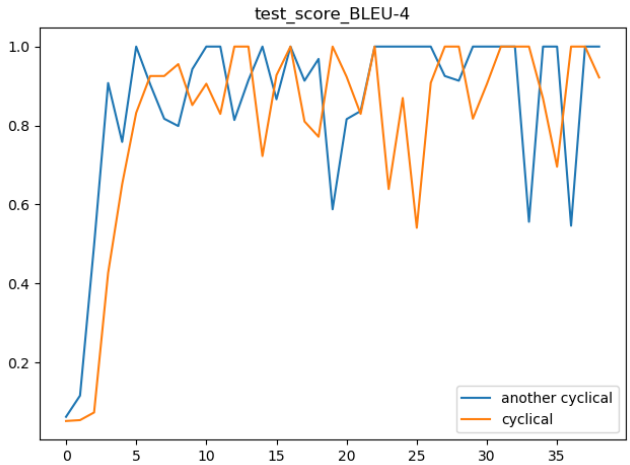


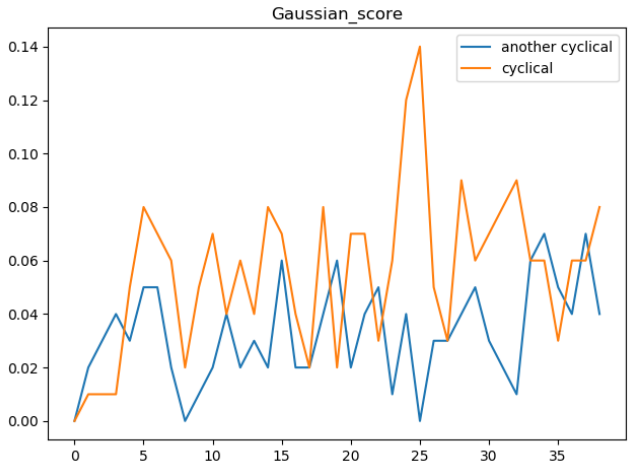


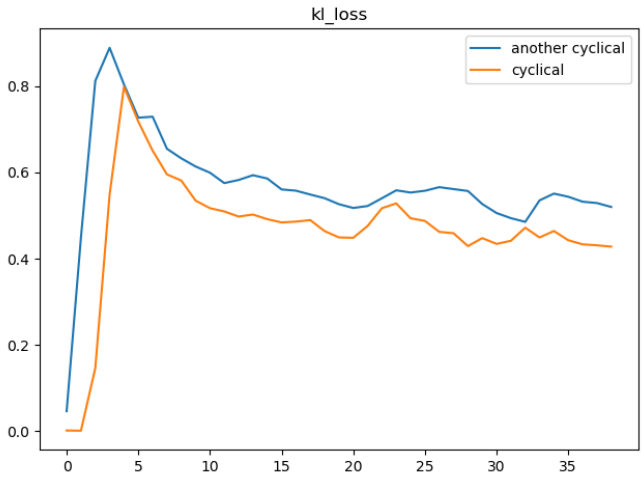


We can see kl loss decrease quickly and lower than cyclical. In teasing score, it also grows up faster. But in the Gaussian score, the performance is bad. This means the model really learns a good task but the regularization doesn't work. The latent doesn't limit to Gaussian distribution, so when we want to generate words by noise, it can't do a good job.

I also try different kinds of cyclical mode. The cyclical mentioned above is that the number of iterator is the same as those remains for. It means it reaches to 1 within 5000 iterators and remains at 1 for 5000 iterators. I have tried another kinds of cyclical. It reaches to 1 within 2000 iterators and remains for remaining iterators(73000). The following images shows comparison.







We can find the influence of kl loss of another cyclical really earlier than cyclical. But the performance is not better than cyclical. Thus I think original one is good enough to implement this task.

- Conclusion

In this lab, we can know how important the balance of reconstruction term and regularization term. If reconstruction term is too big, the model can still learn the task well. But the noise wouldn't approximate Gaussian noise properly. If regularization term is too big, the model is hard to learn the task. KL cost annealing helps to solve this problem. It controls KLD weight within 0 and 1 for different iterators. It determines the influence of lk loss. It's also important to set a proper number of iterator to reach to 1.