Lab4 Report

0856091 蘇邱弘

August 2020

1 Introduction

A Variational AutoEncoder(VAE) is a generative model whose encodings distribution is regularised during the training in order to ensure that its latent space has good properties allowing us to generate some new data. In this lab, I implement a conditional seq2seq VAE for English tense conversion and generation.

In this report, I will first introduce the implementation details of my CVAE model, the dataloader, teacher forcing ratio, and the KL cost annealing. Next, I will show my testing results of English tense conversion and generation. Finally, I will do some discussions.

2 Implementaion Details

2.1 CVAE

In this lab, I implement the CVAE model using Pytorch, and the implementation details are as follows.

First of all, I implement an encoder using LSTM. For the encoder, the initial hidden state is a zero vector concatenated with the embedded condition. After putting the input and the hidden state into the encoder, we can get the last hidden state from it, and then take it to do the following steps. The code snippet of the LSTM encoder is shown in Figure 1.

After the encoder generate the last hidden state, I use two additional fully connected layers to generate the mean vector and the log variance vector respectively. And after getting these two vectors, I put them into reparameterize function to generate the latent vector. The code snippet of the reparameterize function is shown in Figure 2.

Then, I implement a decoder using LSTM. For the decoder, the initial hidden state is generated by latent_cond2hidden function, which embeds the latent vector and the condition to the same dimension as the decoder's hidden layer by using a fully connected layer. Then we can generate the output according to the teacher forcing ratio (described in 2.3), and use a fully connected layer to perform the classification. The code snippet of latent_cond2hidden function is shown in Figure 3, and the code snippet of the LSTM decoder is shown in Figure 4.

```python
class EncoderRNN(nn.Module):
    def __init__(self, hidden_size, latent_size, cond_embedding_size):
        super(EncoderRNN, self).__init__()

        self.input_size = 28
        self.hidden_size = hidden_size
        self.cond_embedding_size = cond_embedding_size

        self.input_embedding = nn.Embedding(self.input_size, hidden_size)
        self.cond_embedding = nn.Embedding(4, cond_embedding_size)

        self.lstm = nn.LSTM(input_size=hidden_size,
                            hidden_size=hidden_size, batch_first=True)

    def forward(self, input, condition):
        batch_size = input.size(0)

        embedded_cond = self.cond_embedding(condition)
        h_0 = torch.zeros(batch_size, self.hidden_size -
                          self.cond_embedding_size).to(device)
        h_0 = torch.cat((h_0, embedded_cond), dim=1)
        h_0 = h_0.view(1, batch_size, self.hidden_size)
        c_0 = torch.zeros(1, batch_size, self.hidden_size).to(device)

        input = self.input_embedding(input)

        _, (h_n, _) = self.lstm(input, (h_0, c_0))
        return h_n
```

Figure 1: Code snippet of the LSTM encoder

```python
def reparameterize(self, mean, logvar):
    std = torch.exp(0.5 * logvar)
    eps = torch.randn_like(std)
    latent = mean + eps * std
    return latent
```

Figure 2: Code snippet of reparameterize function in the CVAE class

```python
def latent_cond2hidden(self, latent, condition):
    batch_size = condition.size(0)

    embedded_cond = self.cond_embedding(condition)
    embedded_cond = embedded_cond.view(1, batch_size, -1)
    latent_cond = torch.cat((latent, embedded_cond), dim=2)
    hidden = self.latent_cond_embedding(latent_cond)

    return hidden
```

Figure 3: Code snippet of latent_cond2hidden function in the CVAE class

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

        self.output_size = 28
        self.hidden_size = hidden_size

        self.input_embedding = nn.Embedding(self.output_size, hidden_size)

        self.lstm = nn.LSTM(input_size=hidden_size,
                            hidden_size=hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, self.output_size)

    def forward(self, input, h_0, c_0):
        input = self.input_embedding(input)
        output, (h_n, c_n) = self.lstm(input, (h_0, c_0))
        output = self.fc(output)
        return output, h_n, c_n
```

Figure 4: Code snippet of the LSTM decoder

Finally, I put all together to a CVAE class. For training, I use forward function to train the encoder and the decoder simultaneously with same condition. For word prediction, I use inference function to generate a word from the input and the target condition. For word generation, I use generation function to generate a new word from a given condition. The code snippet of the CVAE class is shown in Figure 5, 6, and 7.

```python
class CVAE(nn.Module):
    def __init__(self, hidden_size, latent_size, cond_embedding_size):
        super(CVAE, self).__init__()

        self.hidden_size = hidden_size
        self.output_size = 28

        self.encoder = EncoderRNN(hidden_size=hidden_size, latent_size=latent_size,
                                  cond_embedding_size=cond_embedding_size)
        self.decoder = DecoderRNN(hidden_size=hidden_size)

        self.cond_embedding = nn.Embedding(4, cond_embedding_size)
        self.latent_cond_embedding = nn.Linear(
            latent_size + cond_embedding_size, hidden_size)

        self.hidden2mean = nn.Linear(hidden_size, latent_size)
        self.hidden2logvar = nn.Linear(hidden_size, latent_size)

    def reparameterize(self, mean, logvar):
        std = torch.exp(0.5 * logvar)
        eps = torch.randn_like(std)
        latent = mean + eps * std
        return latent

    def latent_cond2hidden(self, latent, condition):
        batch_size = condition.size(0)

        embedded_cond = self.cond_embedding(condition)
        embedded_cond = embedded_cond.view(1, batch_size, -1)
        latent_cond = torch.cat((latent, embedded_cond), dim=2)
        hidden = self.latent_cond_embedding(latent_cond)

        return hidden
```

Figure 5: Code snippet of the CVAE (part1)

```
def forward(self, input, condition, use_teacher_forcing):
    return self.__forward(input, condition, condition, use_teacher_forcing, 'train')

def inference(self, input, input_condition, target_condition):
    outputs, _, _ = self.__forward(
        input, input_condition, target_condition, False, 'inference')
    return outputs

def generation(self, latent, condition):
    batch_size = 1
    seq_length = MAX_LENGTH

    return self.__generation(latent, condition, batch_size, seq_length, False, None)

def __forward(self, input, input_condition, target_condition, use_teacher_forcing, mode):
    batch_size = input.size(0)
    seq_length = MAX_LENGTH
    if mode == 'train':
        seq_length = input.size(1)

    hidden = self.encoder(input, input_condition)
    mean = self.hidden2mean(hidden)
    logvar = self.hidden2logvar(hidden)      Ubuntu, a day ago • epoch gru
    latent = self.reparameterize(mean, logvar)

    outputs = self.__generation(
        latent, target_condition, batch_size, seq_length, use_teacher_forcing, input)

    return outputs, mean, logvar
```

Figure 6: Code snippet of the CVAE (part2)

```
def __generation(self, latent, target_condition, batch_size, seq_length, use_teacher_forcing, input):
    decoder_input = torch.tensor(
        [[SOS_token] * batch_size], device=device).view(batch_size, 1)

    decoder_hidden = self.latent_cond2hidden(latent, target_condition)
    decoder_cell = torch.zeros(1, batch_size, self.hidden_size).to(device)

    outputs = torch.zeros(batch_size, seq_length,
                          self.output_size).to(device)

    for di in range(seq_length):
        decoder_output, decoder_hidden, decoder_cell = self.decoder(
            decoder_input, decoder_hidden, decoder_cell)
        outputs[:, di, :] = decoder_output.view(
            batch_size, self.output_size)

        if use_teacher_forcing:
            # Teacher forcing: Feed the target as the next input
            decoder_input = input[:, di].view(batch_size, 1)
        else:
            # Without teacher forcing: use its own predictions as the next input
            topv, topi = decoder_output.topk(1)
            decoder_input = topi.squeeze().detach().view(
                batch_size, 1)  # detach from history as input

    return outputs
```

Figure 7: Code snippet of the CVAE (part3)

2.2 Dataloader

In this lab, I implement a dataloader which provide the word data as an one hot vector.

First of all, I use getData function to extract words from train.txt. For each word pair, it is stored as a list in length 4, so the whole training data is stored as an np array in shape (1227, 4) after the word extraction.

Next, in __getitem__ function, I randomly select a word from the specified word pair (according to the index), and return the word as an one hot vector, which has an EOS token (which is 1 in my settings) appended in the end, and the corresponding label (tense).

Finally, in order to perform batch training, I append EOS tokens to the one hot vectors with shorter length to insure every one hot vector have the same length in the same batch. The corresponding code snippets are shown in Figure 8 and 9.

```python
def getData():
    words = []
    tenses = []

    with open('train.txt') as f:
        for line in f:
            words.append(line.split('\n')[0].split(' '))
            tenses.append([t for t in TENSES])

    return np.array(words), np.array(tenses)
```

Figure 8: Code snippet of getData function

```python
class MyData(Dataset):
    def __init__(self):
        words, tenses = getData()
        self.words = words
        self.tenses = tenses

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

    def __getitem__(self, idx):
        tense_index_input = random.randint(0, 4-1)
        input = self.words[idx][tense_index_input]
        condition = self.tenses[idx][tense_index_input]
        input_tensor = word2tensor(input)
        condition_tensor = tense2tensor(condition)
        return input_tensor, condition_tensor


def collate_fn(data):
    batch_size = len(data)
    input_tensor = [data[i][0] for i in range(batch_size)]
    input_cond_tensor = torch.LongTensor([data[i][1] for i in range(batch_size)])
    input_tensor = torch.LongTensor(rnn.pad_sequence(input_tensor, batch_first=True, padding_value=1))
    return input_tensor, input_cond_tensor
```

Figure 9: Code snippet of the dataloader

2.3 Teacher Forcing Ratio

In this lab, I use a scheduled teacher forcing ratio to do the training. The schedule function I use is a linear decay function, and each epoch use the same ratio. The code snippet of the schedule function is shown in Figure 10, and the plot of teacher forcing ratio respect to each epoch is shown in Figure 11.

```
def teacher_forcing_ratio_schedule(epoch, n_epochs):
    epoch -= 1
    teacher_forcing_ratio = 1 - (epoch / n_epochs)
    return teacher_forcing_ratio
```

Figure 10: Code snippet of teacher_forcing_ratio_schedule function

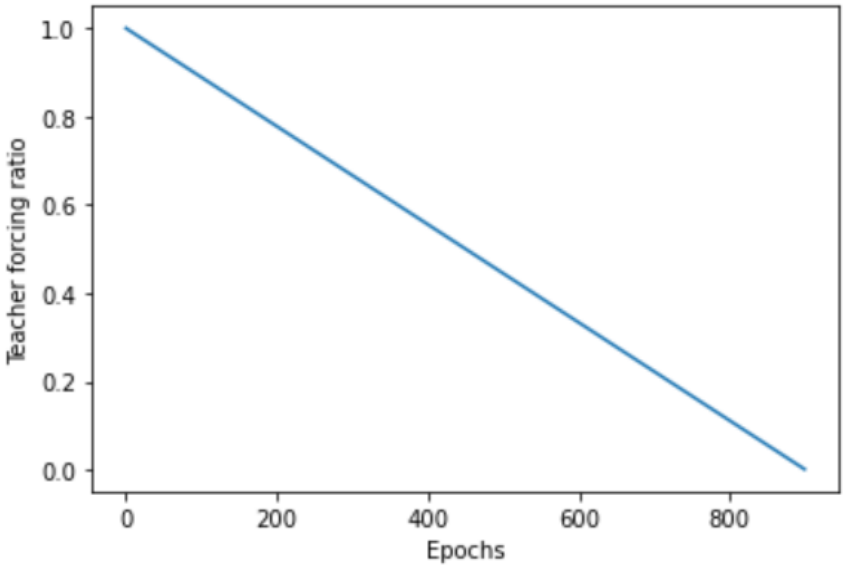


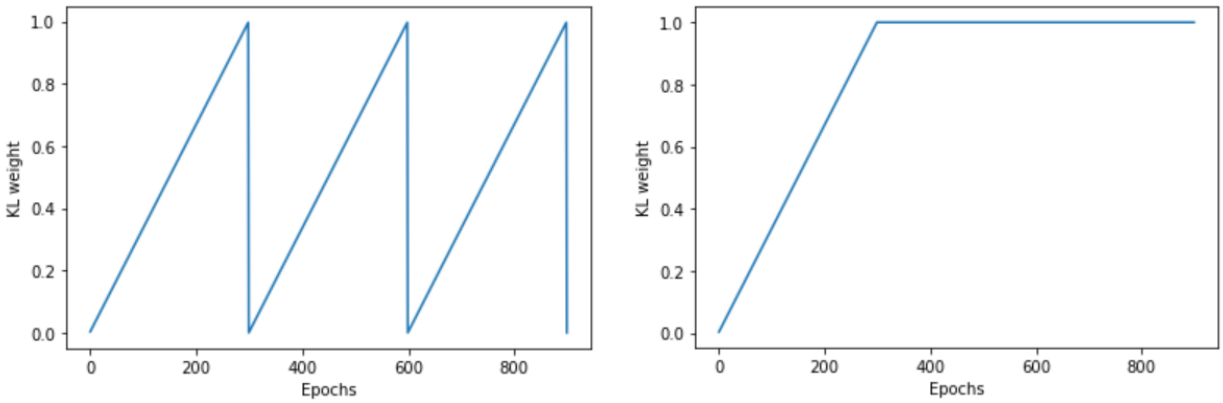
Figure 11: Teacher forcing ratio

2.4 KL Cost Annealing

In this lab, I use a scheduled KL weight to do the training. I use two different kinds of KL cost annealing methods: cyclical and monotonic. For cyclical method, the KL weight will raise from 0 to 1, and then drop back to 0 periodically. For the monotonic method, the KL weight will raise from 0 to 1, and then remain in 1. The code snippet of the schedule function is shown in Figure 12, and the plot of KL weight respect to each epoch is shown in Figure 13.

```python
def KL_weight_schedule(epoch, n_epochs, KL_annealing_method):
    period = n_epochs // 3

    if KL_annealing_method == 'cyclical':
        epoch %= period

    KL_weight = epoch / period
    KL_weight = min(1, KL_weight)

    return KL_weight
```

Figure 12: Code snippet of KL_weight_schedule function



(a) cyclical

(b) monotonic

Figure 13: KL weight

3 Results

The followings are the hyper-parameters I used:

- batch size: 16
- epochs: 900
- learning rate: 0.001
- optimizer: Adam
- loss function: cross entropy
- latent size: 32

- condition embedding size: 8
- LSTM hidden layer size: 256

3.1 Results of Tense Conversion and Generation

Figure 14 and 15 are the screenshots of the BLEU score and gaussian score of my CVAE model. The results are evaluated by the model in the 303-th epoch. For gaussian score, I generate gaussian noises by torch.randn function to calculate the score of 100 generated words. The code snippet of gaussian score evaluation is shown in Figure 16.

```
target: begins
prediction: begins

input: expend
target: expends
prediction: expends

input: sent
target: sends
prediction: senses

input: split
target: splitting
prediction: spliting

input: flared
target: flare
prediction: flare

input: functioning
target: function
prediction: function

input: functioning
target: functioned
prediction: functioned

input: healing
target: heals
prediction: heals

Average BLEU-4 score: 0.8959207002666769
```

Figure 14: BLEU score

```
['terry', 'terries', 'terring', 'terried']
['beduc', 'beduces', 'beducing', 'beduc']
['multiply', 'multiplies', 'multiplying', 'multiplied']
['feer', 'feers', 'feering', 'negled']
['reason', 'peers', 'rearing', 'peered']
['distrust', 'distrusts', 'distrusting', 'distrusted']
['ejaculate', 'ejploys', 'ejaculating', 'ejaculated']
['spray', 'sprays', 'spraying', 'sprayed']
['speak', 'seemons', 'speaking', 'speaked']
['intervene', 'intervenes', 'intervening', 'intervened']
['authorize', 'affirms', 'affirming', 'affirmed']
['reseat', 'rewires', 'rewiring', 'rewired']
['pead', 'peels', 'peeling', 'peeled']
['welop', 'wellows', 'wrenching', 'wrenched']
['charge', 'changes', 'clipping', 'charged']
['recall', 'recalls', 'recalling', 'recalled']
['focuse', 'focuses', 'focuses', 'focused']
['betton', 'betakes', 'bettoing', 'betaked']
['coumble', 'coughinates', 'coumbling', 'coughinated']
['study', 'studies', 'studying', 'studied']
['participate', 'participates', 'participating', 'participated']
['inply', 'umps', 'inplying', 'emerged']
['receive', 'receives', 'receiving', 'received']
['testify', 'testifies', 'testifying', 'testifted']
['testify', 'testifies', 'testifying', 'testified']
['drown', 'drowns', 'drowning', 'drowned']
['invol', 'invoses', 'invowing', 'invowed']
['blinker', 'blinks', 'blinkening', 'blinked']
['gremar', 'grimaces', 'grimacing', 'grimaced']
['bleck', 'blects', 'blecting', 'blected']
Gaussian score: 0.33
```

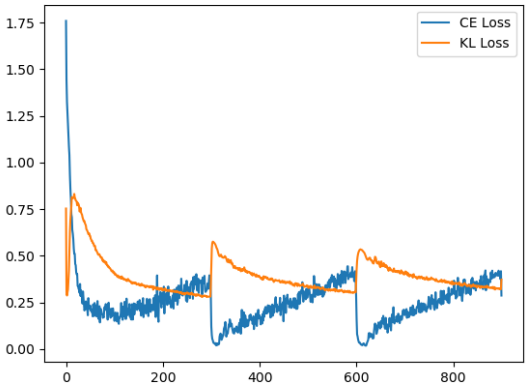
Figure 15: Gaussian score

```python
def eval_model_gaussian(cvae, verbose=False):
    latent_size = 32
    generate_words = []

    for i in range(100):
        z = torch.randn(1, 1, latent_size)
        generate_tenses = []

        for tense in TENSES:
            w = word_gen(cvae, z, tense=tense)
            generate_tenses.append(w)

        if verbose:
            print(generate_tenses)

        generate_words.append(generate_tenses)

    gaussian_score = Gaussian_score(generate_words)
    if verbose:
        print(f'Gaussian score: {gaussian_score}')

    return gaussian_score
```

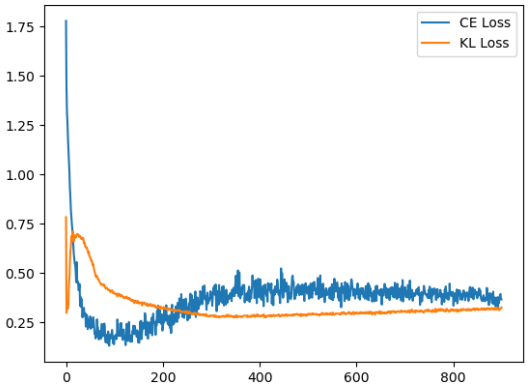
Figure 16: Code snippet of eval_model_gaussian

3.2 Training Trends

Below is the training trend with the CVAE model of two different KL cost annealing methods. Figure 17 shows the training trend of the KL loss and cross entropy loss, and Figure 18 shows the training trend of the BLEU score.

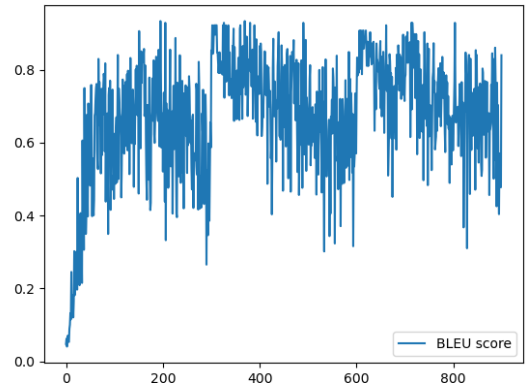


(a) cyclical

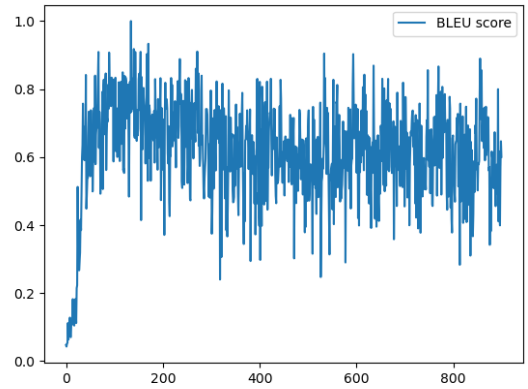


(b) monotonic

Figure 17: KL loss and cross entropy loss



(a) cyclical



(b) monotonic

Figure 18: BLEU score

4 Discussion

4.1 KL Weight

While training the VAE, the loss is combined with the reconstruction error (cross entropy loss), which indicates the similarity between the input and the output, and the regularization term (KL loss), which indicates the similarity between the latent space and the prior distribution.

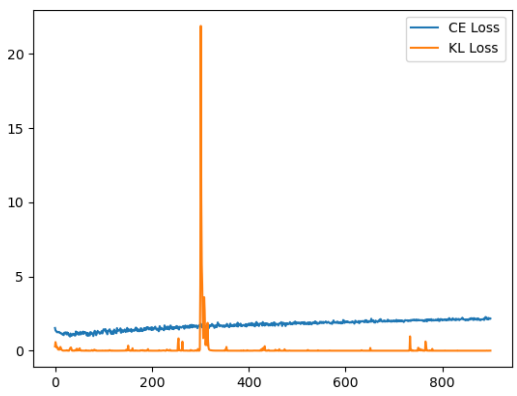
During training, the KL loss usually drop faster, because it is much easier for the model to ignore the input in order to make latent space closer to prior distribution than reconstruct the output from input. So if we don't apply any KL cost annealing technique, it may make the latent vector carry less information from the input, which make the reconstruction task harder.

Figures in 3.2 show the loss value after applying KL cost annealing technique (which is described in 2.4). As shown in the figures, the cross entropy loss will drop fast, while the KL loss increases in the beginning. This can make the model learn to reconstruct the target output (thus make BLEU score increase fast). Then, the KL weight will gradually increase, which make KL loss start to decrease and cross entropy loss start to increase. This means that the model starts to learn to make latent space closer to prior distribution, but this makes the reconstruction task harder (thus make BLEU score decrease).

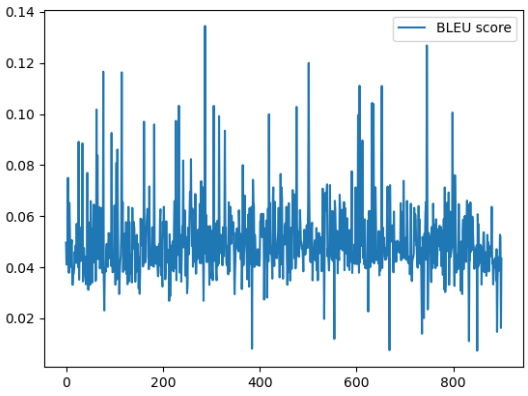
In this lab, I apply two KL cost annealing methods: cyclical and monotonic, and as shown in 3.2, with cyclical method, the loss curves and the BLEU score will oscillate. On the contrary, with monotonic method, the loss curves the BLEU score are more stable. And for these two methods, the performance doesn't have much different on the BLEU scores.

4.2 Learning Rate

In this lab, I use learning rate of 0.001 to train the model and get a good result. Actually, I have tried different learning rates, but I found that the training will fail and get a bad result. For example, I have tried learning rate 0.01, and the training trend is shown in Figure 19. As shown in the figure, I notice that with a larger learning rate, the KL loss will be minimize faster in the beginning, and thus make latent vector contain less information, which leads to the failure of the training.



(a) Loss value



(b) BLEU score

Figure 19: Training trend with learning rate = 0.01

4.3 Teacher Forcing Ratio

Teacher forcing is a technique for training RNN. With teacher forcing, the model can learn better by using the ground truth as the input of next timestamp. But if we keep using the ground truth, it will have problem when we applying testing data without label.

Thus, in this lab, I use scheduled teacher forcing ratio to train the LSTM decoder. The overall idea is: in the beginning, the model hasn't learn to output correctly, so we use higher teacher forcing ratio to make it learn better. But after the model becomes smarter, we can gradually reduce the teacher forcing ratio, thus it can still output the correct label without the ground truth. The scheduling function is described in 2.3, and as we can see in 3.2, the BLEU scores didn't drop significantly due to the decreasing teacher forcing ratio, which means that the model can still perform well without the teaching from the ground truth label.