Deep Learning and Practice Lab5
Conditional Sequence-to-sequence VAE

0756079 陳冠聞

May 8, 2019

1 Introduction

Variational Autoencoders(VAEs) is a generative model that combine the concept of latent factor model and autoencoder. VAEs allow us to formalize this problem in the framework of probabilistic graphical models where we are maximizing a lower bound on the log likelihood of the data.

In this lab, I will implement the conditional sequence-to-sequence VAE for the task of English tense conversion using the *PyTorch* package. Also, I train the model and evaluate it using the BLUE-score as well as the random generation test. Furthermore, I compare the performance between the model trained with/wihout KL cost anneling in the dicussion.

2 Seq2Seq CVAE

2.1 VAE

VAE is a probabilistic model with latent variable this is built on top of end-to-end trainable neural network. In VAEs, we want to choose the model parameter θ to maximize the marginal distribution $p_\theta(x) = \int p_\theta(x|z)p(z)dz$. However, this marginal distribution is intractable while we use a neural network to model the conditional distribution $p_\theta(x|z)$. So we maximize the variational lower bound

$$L(x, q, \theta) = logp_\theta(x) - KL(q(z)\|p_\theta(z|x)) \tag{1}$$

instead, where $q(z)$ is an arbitrary distribution.

And we can use another neural network $q_\theta(z|x)$ to model the distribution $q(z)$. Then the variational lower bound can be formulated as following (detail derivation please refer to the appendix)

$$L(x, q, \theta) = \mathbb{E}_{z \sim q_\theta(z|x)}[p_\theta(x|z)] - KL(q_\theta(z|x)\|p(z)) \tag{2}$$

The first term of right hand side in Equation 2 is the **reconstruction term**, which represent the expected likelihood to generate x. The second term is the **regularization term**, which is the distance between the latent distribution and the prior distribution $p(z)$.

Unfortunately, gradient can not be computed in the sample operation like the red rectangle in Figure 1. So we need to use the **Reparameterization Trick**, which draw sample from $N(0, I)$ then do differentible operation (multiplication and addition) instead of directly draw sample from $N(u, \Sigma)$. Then we can use back-propagation algorithm to train the model.

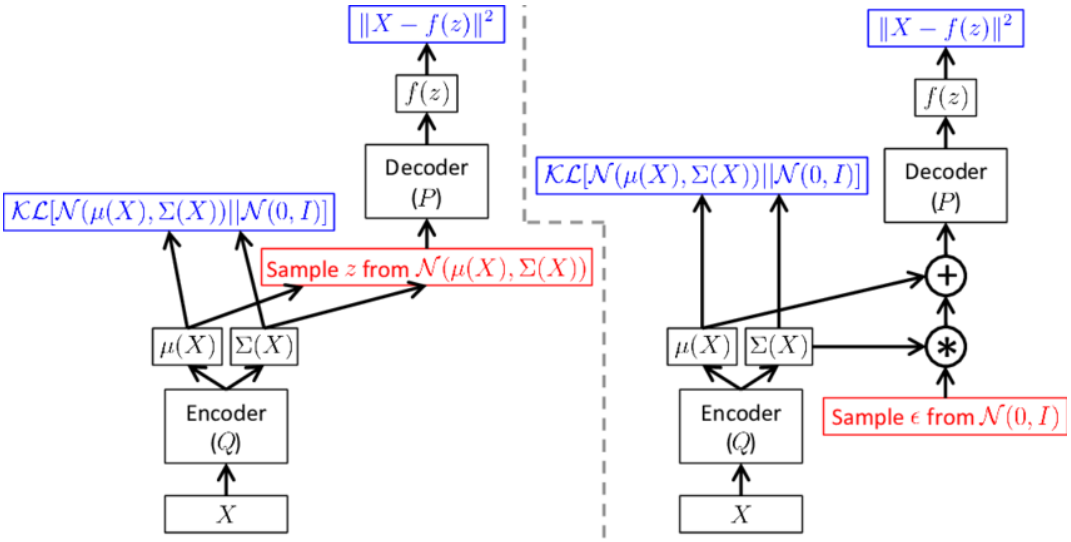


Figure 1: Reparameterization Trick

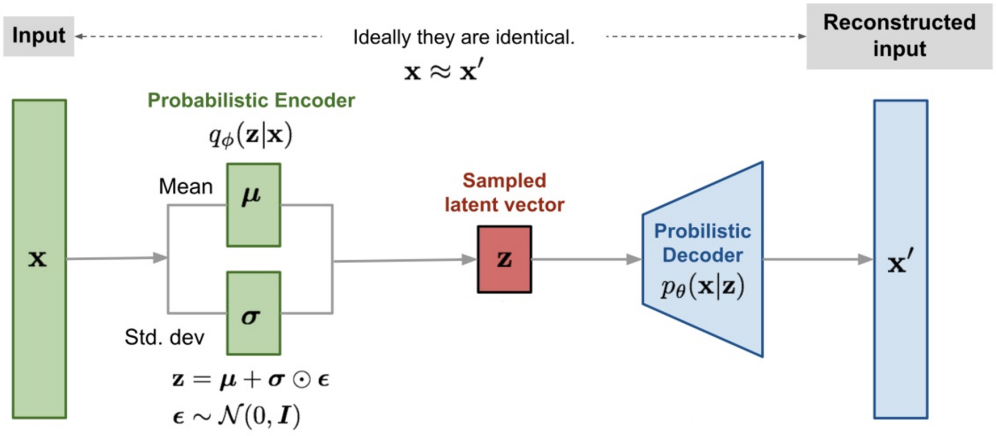


Figure 2: VAE architecture

2.2 Conditional VAE

In **Conditional VAE**, we want to maximize the conditional likelihood $p_\theta(x|c)$ while others remain the same. So the variational lower bound become

$$L(x, q, \theta, c) = logp_\theta(x|c) - KL(q(z|c)\|p_\theta(z|x, c)) \tag{3}$$

Again we use the neural network $q_\theta(z|x, c)$ to model the distribution $q(z|c)$, then the variational lower bound can be formulated as

$$L(x, q, \theta, c) = \mathbb{E}_{z\sim q_\theta(z|x,c)}[p_\theta(x|z, c)] - KL(q_\theta(z|x, c)\|p(z|c)) \tag{4}$$

In practice, we often use $\widetilde{x}$ which is the concatenation of origin input x and condition c as the input of encoder Q. And use $\widetilde{z}$ which is the concatenation of origin latent z and condition c as the input of decoder P.

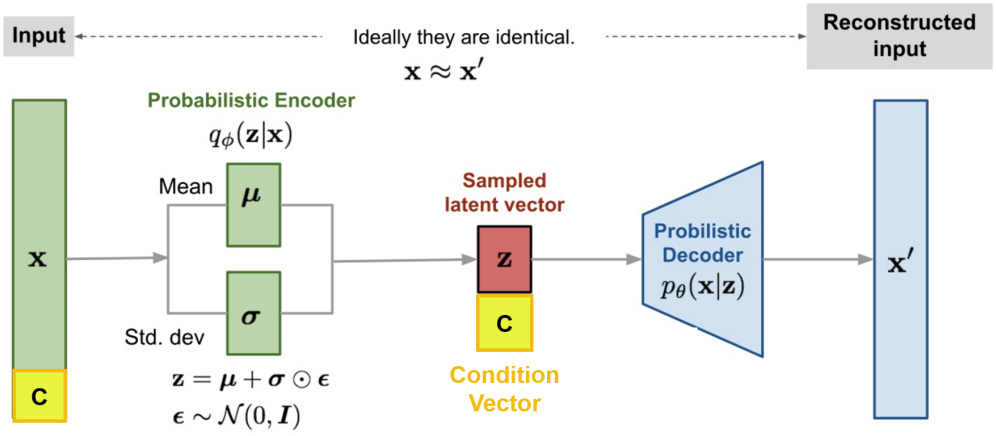


Figure 3: Conditional VAE architecture

2.3 Sequence-to-Sequence Encoder and Decoder

The task of this lab is the English tense conversion. Since the input word will have different length, I use the **Sequence-to-Sequence** model as our encoder and decoder in VAE. For the encoder, I use t-th character in word as input of Sequence-to-Sequence model at timestep t. For the decoder, I use the SOS (Start-Of-the-String) token as input at timestep t=1, and use the output of timestep t-1 as the input at timestep t to generate the sequence until the EOS (End-Of-the-String) token is generated.

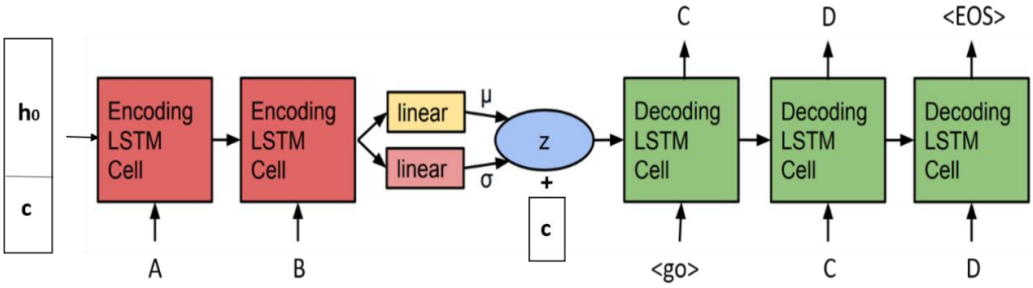


Figure 4: Conditional Seq2Seq VAE architecture

3 Implementaion

3.1 Seq2Seq CVAE

To construct the **Sequence-to-Sequence Conditional Variational Autoencoder**, I first construct the encoder and decoder. Both encoder and decoder use word embedding to project the input at each timestep to the dimensions of hidden size, then use **gated recurrent unit** to process hidden vector and embedded input at different timesteps, and generate the output and new hidden vector. The difference is that the decoder must output another sequence, so the output of hidden unit will pass through a fully connected layer to generate the output in the same dimension as input dimension. In addition, since the input will have condition, I will first embedding the condition vector to the R^8, then set the last 8 element of initial hidden unit to be this embedded condition vector.

```
class EncoderRNN(nn.Module):
    def __init__(self, input_size, hidden_size, cond_size):
        super(EncoderRNN, self).__init__()
        self.hidden_size = hidden_size
        self.cond_size = cond_size
        self.embedding = nn.Embedding(input_size, hidden_size)
        self.cond_embedding = nn.Embedding(cond_size, 8)
        self.gru = nn.GRU(hidden_size, hidden_size, batch_first=True)

    def forward(self, input, input_cond, hidden=None):
        if hidden is None:
            hidden = self.initHidden(input_cond)
        batch_size, seq_len = input.size(0), input.size(1)
        embedded = self.embedding(input).view(batch_size, seq_len, -1)
        output = embedded
        output, hidden = self.gru(output, hidden)
        return output, hidden

    def initHidden(self, cond_tensor):
        batch_size = cond_tensor.size(0)
        h0 = torch.zeros(batch_size, self.hidden_size-8).to(device)
        cond_embedding = self.cond_embedding(cond_tensor).view(batch_size, 8).to(device)
        hidden = torch.cat((h0, cond_embedding), dim=1).view(1, batch_size, self.hidden_size).to(device)
        return hidden
```

Figure 5: Code Snippet for EncoderRNN

```python
class DecoderRNN(nn.Module):
    def __init__(self, hidden_size, output_size):
        super(DecoderRNN, self).__init__()
        self.hidden_size = hidden_size
        self.embedding = nn.Embedding(output_size, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size, batch_first=True)
        self.out = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden):
        batch_size, seq_len = input.size(0), input.size(1)
        embedded = self.embedding(input).view(batch_size, seq_len, -1)
        output = F.relu(embedded)
        output, hidden = self.gru(output, hidden)
        output = self.out(output)
        return output, hidden

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

Figure 6: Code Snippet for DecoderRNN

Then I construct the VAE based the encoder and decoder above. When input pass through the encoder, I use **reparameterization trick** for the hidden vector at the last timestep to generate the latenet vector. After that, I concatenate the latent vector with embedded target condition using the same embedding method in encoder. Finally, I use fully connected layer to project the latent vector back to the dimension of hidden size and feed it to the decoder to generate the output sequence.

```python
class VAE(nn.Module):
    def __init__(self, input_size, hidden_size, latent_size, cond_size, output_size):
        super(VAE, self).__init__()

        # Encoder & Decoder
        self.encoder = EncoderRNN(input_size, hidden_size, cond_size)
        self.decoder = DecoderRNN(hidden_size, output_size)

        # Hidden to Latent (REPARAMETERIZATION)
        self.hidden2mean = nn.Linear(hidden_size, latent_size)
        self.hidden2logvar = nn.Linear(hidden_size, latent_size)

        # Latent to Hidden
        self.latent2hidden = nn.Linear(latent_size+8, hidden_size)
        self.cond_embedding = nn.Embedding(cond_size, 8)

        self.input_size = input_size
        self.hidden_size = hidden_size
        self.latent_size = latent_size
        self.cond_size = cond_size
        self.output_size = output_size

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

Figure 7: Code Snippet for VAE(part1)

```python
def forward(self, input, condition, use_teacher_forcing=True):
    #----------sequence to sequence part for encoder----------#
    batch_size, seq_length = input.size(0), input.size(1)
    encoder_output, encoder_hidden = self.encoder(input, condition, None)

    #---------- sequence to sequence part for VAE----------#
    mean = self.hidden2mean(encoder_hidden)
    logvar = self.hidden2logvar(encoder_hidden)
    latent = self.reparameterize(mean, logvar)
    hidden = self.latent_cond2hidden(latent, condition)

    #----------sequence to sequence part for decoder----------#
    outputs = Variable(torch.zeros(batch_size, seq_length, self.output_size)).to(device)
    decoder_input = torch.tensor([[SOS_token] * batch_size], device=device).view(batch_size, 1, 1)
    decoder_hidden = hidden
    if use_teacher_forcing:
        # Teacher forcing: Feed the target as the next input
        for di in range(seq_length):
            decoder_output, decoder_hidden = self.decoder(decoder_input, decoder_hidden)
            outputs[:, di, :] = decoder_output.view(batch_size, self.output_size)
            # next input
            decoder_input = input[:, di].view(batch_size, 1, 1)
    else:
        # Without teacher forcing: use its own predictions as the next input
        for di in range(seq_length):
            decoder_output, decoder_hidden = self.decoder(decoder_input, decoder_hidden)
            outputs[:, di, :] = decoder_output.view(batch_size, self.output_size)
            # next input
            topv, topi = decoder_output.topk(1)
            decoder_input = topi.squeeze().detach().view(batch_size, 1, 1)  # detach from history as input
    return outputs, mean, logvar
```

Figure 8: Code Snippet for VAE(part2)

3.2 Loss Function

The loss function of VAE is consist of reconstruction loss and regularization loss. The reconstruction loss is to make the output of model to be closer to the target as possible. In this lab, I use the **cross entropy** as the reconstruction loss function. For the regularization loss, it's mean to the force latent distribution to be closer to the prior distribution. In this lab we assume the prior is $N(0, I)$, so the loss is the KL divergence between latent distribution and standard normal distribution, and I adopt the formula in the https://arxiv.org/abs/1312.6114. To make the cross entropy losses and KL divergence losses properly scaled against each other, I average out the losses in a batch both cross entropy losses and KL divergence losses.

```python
def loss_fn(output, target, mean, logvar):
    num_classes = vocab_size
    # Cross Entropy Loss
    loss_fn_ce = nn.CrossEntropyLoss(reduction='mean')
    output = output.view(-1, num_classes)
    target = target.view(-1)
    CE_loss = loss_fn_ce(output, target)
    # KL Divergence
    KL_loss = -0.5 * torch.mean(1 + logvar - mean.pow(2) - logvar.exp())
    return CE_loss, KL_loss
```

Figure 9: Code Snippet for Loss Function

3.3 Teacher-Forcing ratio

Teacher forcing is a method for quickly and efficiently training recurrent neural network models that use the ground truth from a prior time step as input. Without Teacher forcing, it's hard for sequence model to learn the correct output since the error in early will propagate to the later in the sequence output, especially at the beginning of training, where the most of the outputs of model are wrong. However, the model trained with Teacher forcing will have trouble to produce correct sequence at test time since there are no correct answer to refer in the test time.

The common approach to solve this issue is to scheduling teacher-forcing ratio. That is, whether to use teacher-forcing is determined by the probability. The probability to use teacher-forcing is high at the beginning of training to help model to learn quickly, and the probability to use teacher-forcing is low at the end of the training to make the model adapt to the test environment. In this lab, I set the teacher-forcing ratio to 1.0 at the beginning of training, and gradually decrease it follow the function which is $1 - sigmoid$.

3.4 KL annealing

In the VAE, the total loss is combined with the KL loss and the reconstruction loss. Interestingly, this two loss are against each other at some aspect. Minimize reconstruction loss can make the output of model similar to the target output. On the other hand, minimize KL loss make the latent space similar to the prior distribution, and it usually achieved by reducing the information carried by the latent variable, which make the task of reconstruction become harder. This characteristic of loss can be problematic, because it's easier for the model to ignore the input than learn the target output, and it's common that the model will ignore the input to yield zero KL loss.

One solution to solve this problem is called **KL annealing**. In KL annealing, at the start of training, we set the weight of KL loss to zero, so that the model learns to encode as much information in z as it can. Then, as training progresses, we gradually increase this weight, forcing the model to smooth out its encodings and pack them into the prior. In this lab, I annealing the weight of KL term using *sigmoid* function in a period. That is, I let the weight of KL loss term to be 0 at the beginning of a period, and raise the weight follow the *sigmoid* function in that peiord.

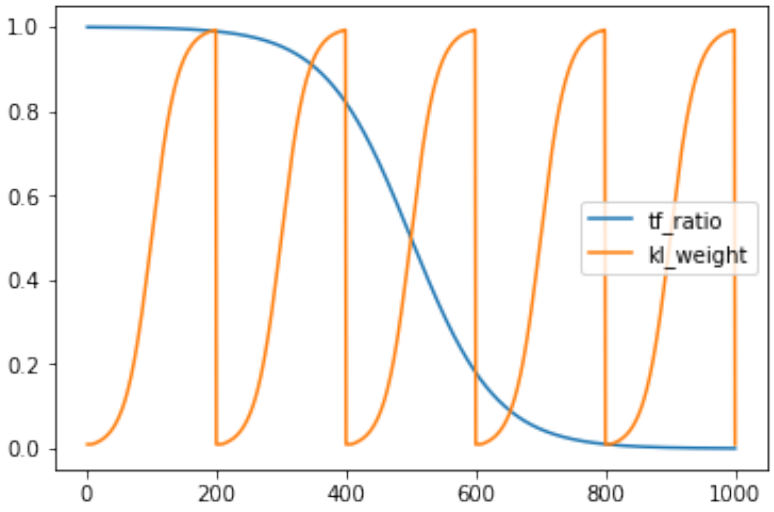


Figure 10: Teacher-Forcing ratio and KL weight

3.5 Dataloader

In this lab, I generate data by first draw a word as well as the tense of it, and represent it using one hot vector of R^{28}, where the first element and second element represent whether it is Start-Of-the-String (SOS) and End-Of-the-String (EOS), and the following elements represent the English alphabet from a to z respectively. In training, I feed the word as well as the condition to the encoder, and use the same word as well as tense to the decoder.

In order to adopt batch training, I use *torch.utils.data.DataLoader* class to load data. Since every word length might be different, I use the *rnn.pad_sequence* to pad EOS tokens at the end of shorter words, and make all the input words in one batch have the same length.

```
class MyData(data.Dataset):
    def __init__(self):
        words, tenses = prepare_data()
        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]
        input_cond  = self.tenses[idx][tense_index_input]
        input_tensor = tensorsFromWord(input)
        input_cond_tensor = tensorFromTense(input_cond)
        return input_tensor, input_cond_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_utils.pad_sequence(
        input_tensor, batch_first=True, padding_value=EOS_token))
    return input_tensor, input_cond_tensor
```

Figure 11: Code Snippet for Data Loader

4 Result

In this lab, I use the the conditional seq2seq VAE above to train on 1,227 pair of English word and tense. I use the hidden size 256 as well as latent size 32. Adam optimizer with initial learning rate 0.01 and batch size 32 is used in training, and I train 1,000 epochs in total.

4.1 Training Trend

From Figure 12 and Figure 11, at the beginning of the a period, since the KL weight is very small, the KL loss increases while the cross entropy loss decreases, which means that the model learn to reconstruct well, but the latent distribution is become different from the prior distribution. Afterward, when KL weight raise, the KL loss start to decrease and the cross entropy start to increase, which means that the latent distribution is closer to the prior distribution, but the reconstruction become worse.

Furthermore, as shown in Figure 13, the **BLEU** score decrease while the KL weight raise. The reason might be that when the KL weight raise, the latent distribution is forced to be closer to the unit Gaussian distribution, which could make the latent vector contain less information, lead to the task of reconstruction sequence from latent vector becomes harder.

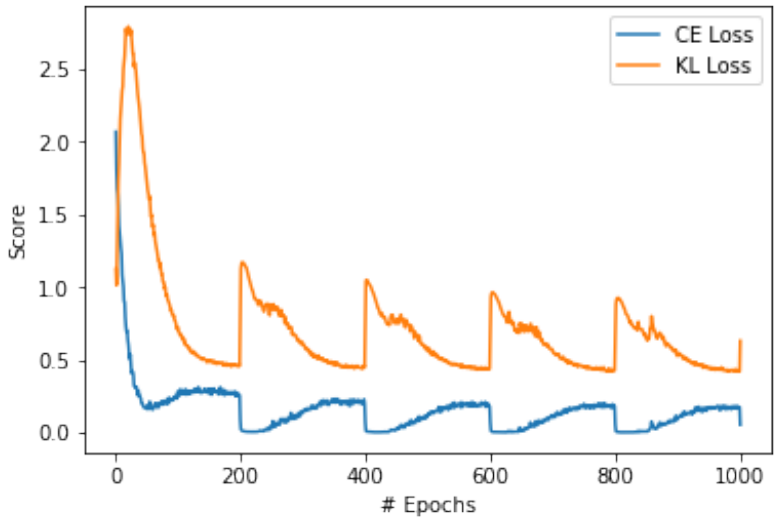


Figure 12: Cross Entropy and KL Loss Curve

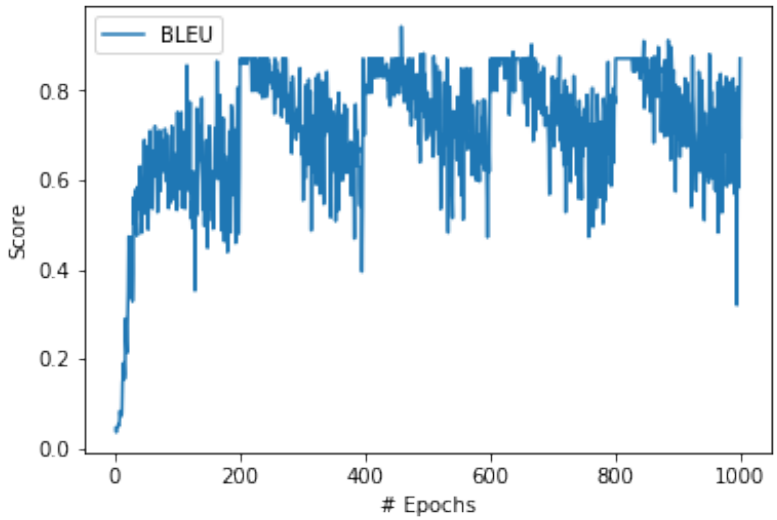


Figure 13: BLUE score curve

4.2 Model Evaluation

After training, I evaluate the model performance using the BLEU score on the tense conversion task. The resulting BLEU score is 0.86, and from the Figure 14 we can see that the model can successfully convert most of the words. Furthermore, to verify the latent distribution is similar to unit Gaussian distribution, I draw random noise from the unit Gaussian as latent vector to generate the output sequence, and the result in Figure 15 shows that the model can generate real words with correct tense from Gaussian noise.

```
 BLEU   | Prediction |   Target
------------------------------
1.0000 | abandoned  | abandoned
0.5946 | abettngg   | abetting
1.0000 | begins     | begins
1.0000 | expends    | expends
0.2857 | senss      | sends
0.7421 | spliting   | splitting
1.0000 | flare      | flare
1.0000 | function   | function
1.0000 | functioned | functioned
1.0000 | heals      | heals
------------------------------
Avg: 0.8622436082327134
```

Figure 14: BLEU scores

```
# Generation
z = torch.randn(1, 1, latent_size)
for tense in TENSES:
    x = generation(vae, z, tense=tense)
    print(x, tense)
```

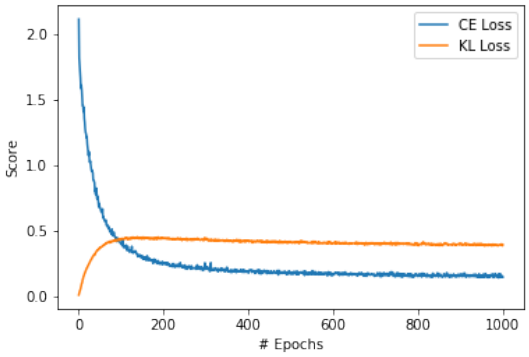
```
bend sp
bends tp
bending pg
bended p
```

Figure 15: Random Generation

5 Discussion

5.1 KL weights annealing

In this lab, I use the KL cost annealing technique, and I observe that both cross entropy loss and KL divergence loss are highly influenced by the weight of the KL loss term. To see the effect of using the KL weight annealing, I train another model without KL weight annealing. That is, the KL weight is fixed to 1 from the beginning of training to the end. As the Figure 16 shows, the loss curve of the model without KL weight annealing do not oscillate, but the resulting BLEU score can is pretty bad compared to the model with KL weight annealing.



(a) Loss Curve

BLEU	Prediction	Target
0.3303	aumlnoned	abandoned
0.1820	abhing	abetting
1.0000	begins	begins
1.0000	expends	expends
0.0744	sass	sends
0.1597	sprining	splitting
0.1574	fear	flare
0.8409	functioe	function
0.1561	fonctooed	functioned
1.0000	heals	heals

Avg: 0.49007851632868515

(b) BLEU Score

Figure 16: Result Without KL annealing

6 Appendix

Belows are the mathematical derivation for the **Variational Autoencoders** (VAEs).

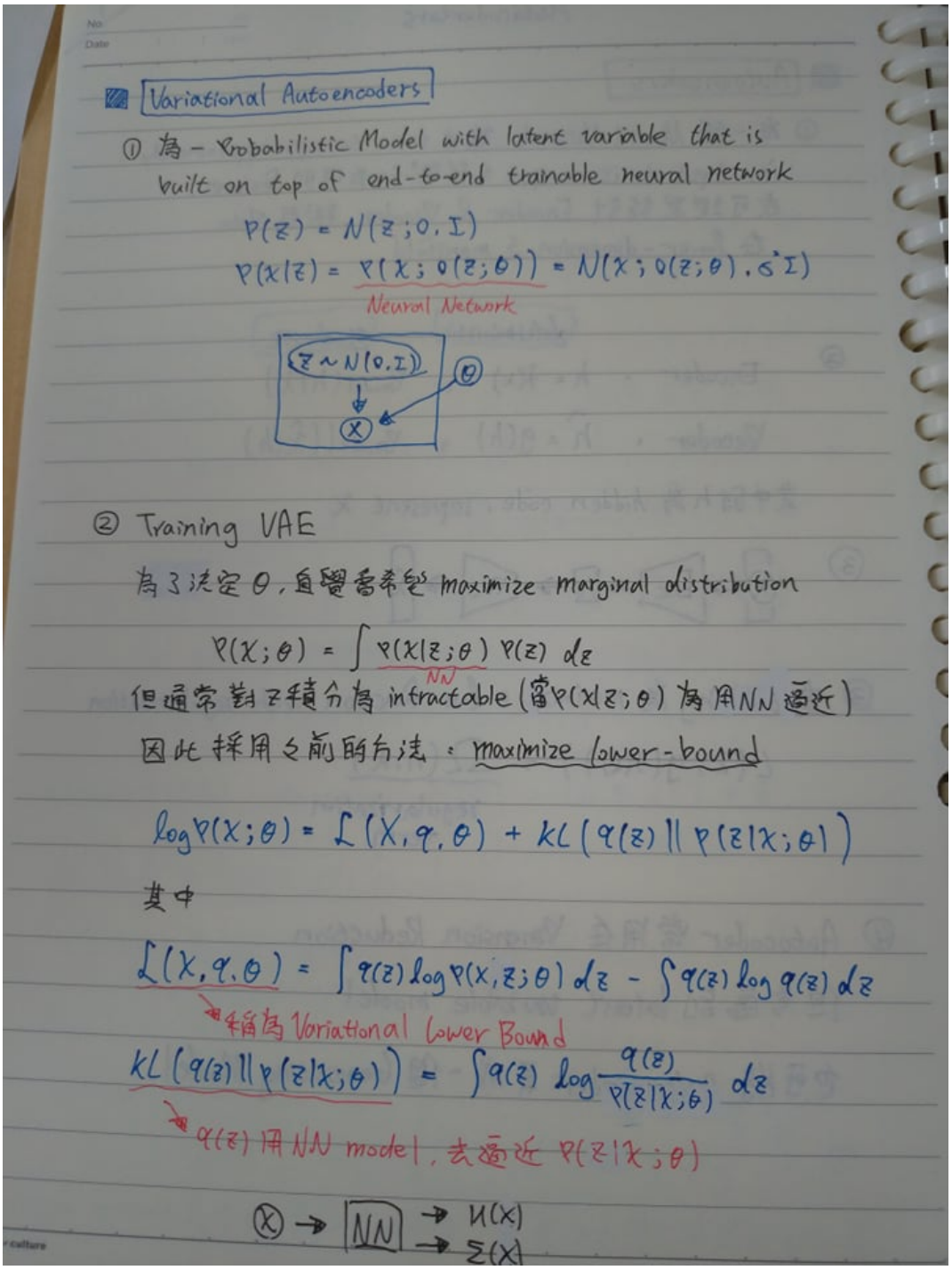


Figure 17: VAE Derivation (part1)

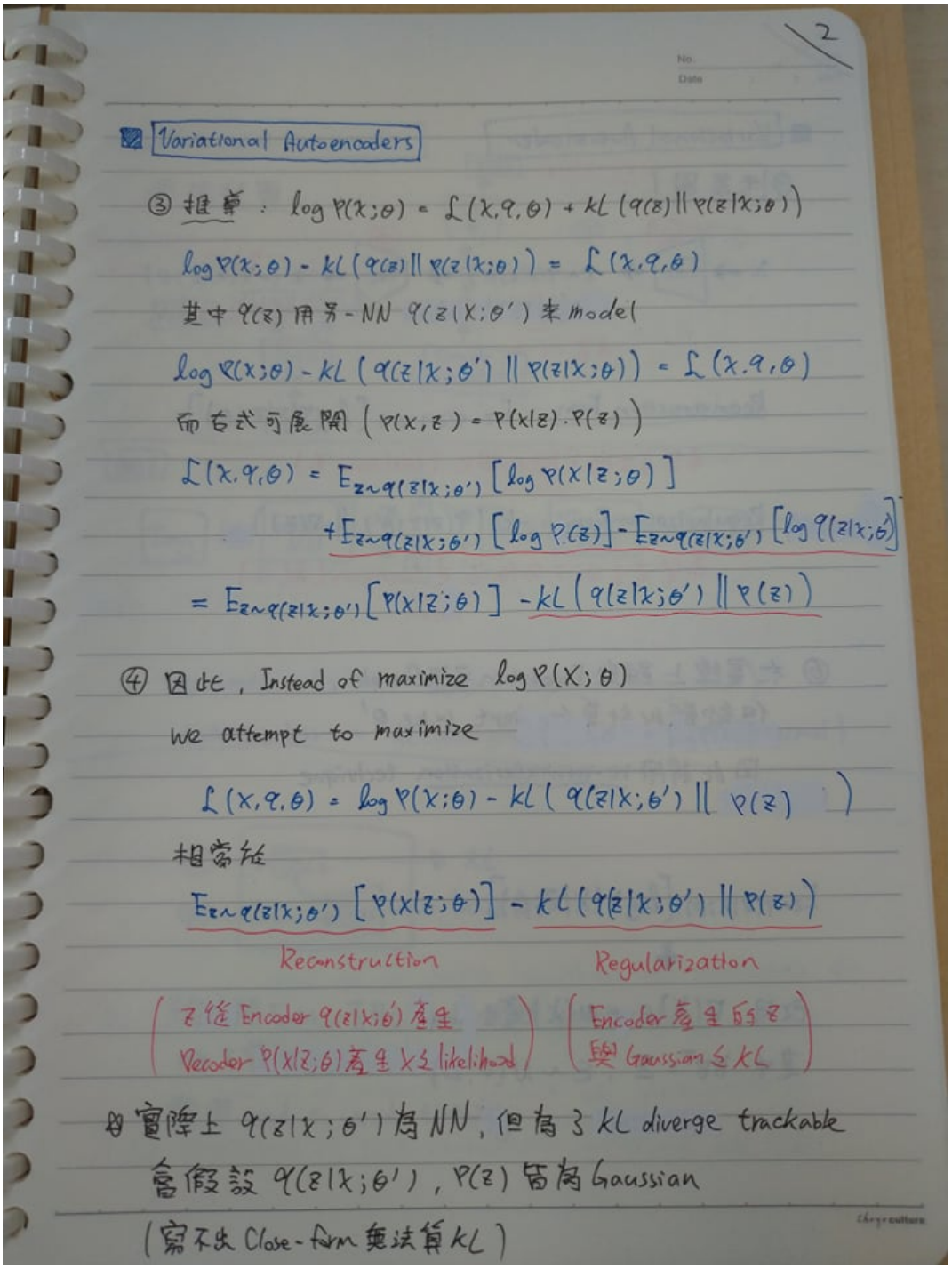


Figure 18: VAE Derivation (part2)

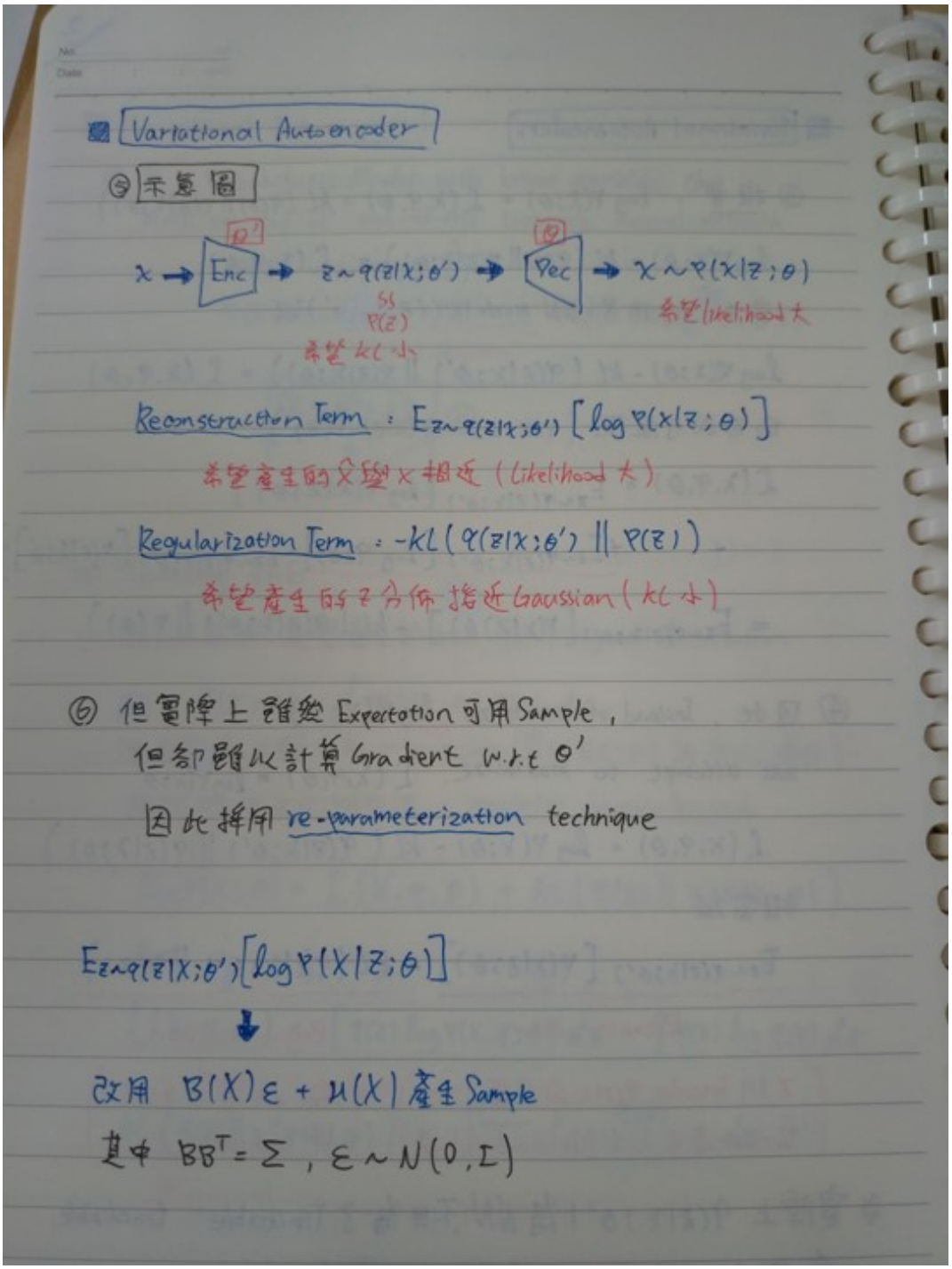


Figure 19: VAE Derivation (part3)

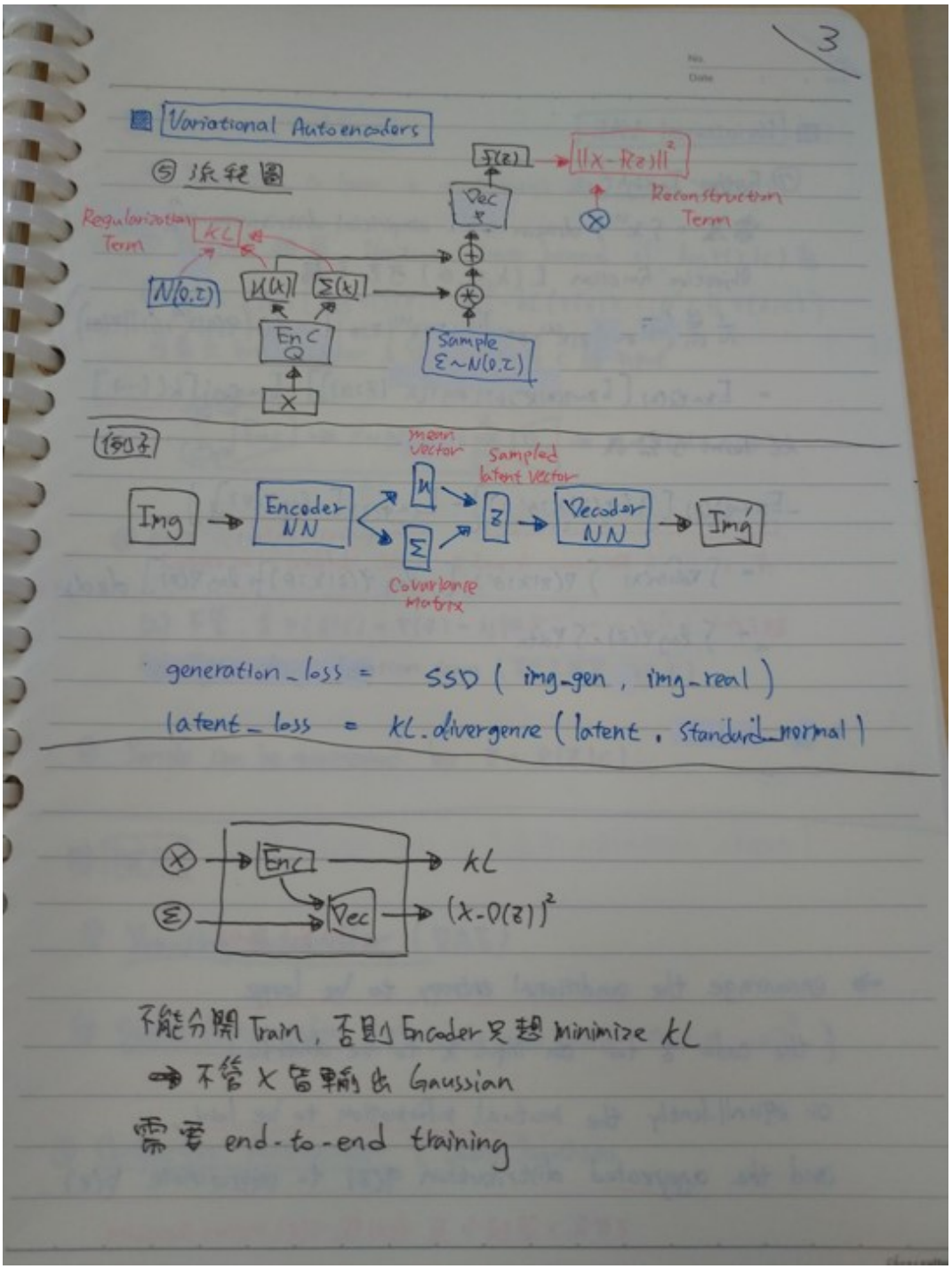


Figure 20: VAE Derivation (part4)