Lab5: Conditional Sequence-to-Sequence VAE

Hsiang-Chun Yang
Institute of Multimedia Engineering
National Yang Ming Chiao Tung University
yanghc.cs09g@nctu.edu.tw

1 Introduction

In this lab, we will implement a conditional seq2seq VAE for English tense conversion and generation. Details about the implementation are as follows.

- The encoder and decoder must be implemented by LSTM
- Implement reparameterization
- Adopt teacher forcing and KL loss annealing the in training process
- Visualize the cross entropy loss, KL loss, BLEU-4 score, and Gaussian score during training

2 Derivation of CVAE

First,

$$\log p(X|c;\theta) = \log p(X,Z|c;\theta) - \log p(Z|X,c;\theta)$$

Second, introduce an arbitrary distribution $q(Z|c)$ on both sides and integrate over Z.

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

Second, introduce a distribution $q(Z|X,c;\phi)$ which is modeled by a neural network with parameter ϕ.

$$\begin{aligned}\mathcal{L}(X,c,q,\theta) &= \int q(Z|c)\log p(X,Z|c;\theta)dZ - \int q(Z|c)\log q(Z|c)dZ \\ &= \int q(Z|c)\log p(X|Z,c;\theta)dZ + \int q(Z|c)\log p(Z|c)dZ - \int q(Z|c)\log q(Z|c)dZ \\ &= E_{Z\sim q(Z|X,c;\phi)}\log p(X|Z,c;\theta) + E_{Z\sim q(Z|X,c;\phi)}\log p(Z|c) - E_{Z\sim q(Z|X,c;\phi)}\log q(Z|X,c;\theta) \\ &= E_{Z\sim q(Z|X,c;\phi)}\log p(X|Z,c;\theta) - KL(q(Z|X,c;\theta)||p(Z|c))\end{aligned}$$

3 Implementation details

3.1 Data

We use one-hot encoding to quantify an English word into a series of numbers. Alphabets `a` $\sim$ `z` are mapped to $0 \sim 25$. We also add a start-of-sequence (`SOS`) token and end-of-sequence (`EOS`) token before and after the encoded word. The implementation of the one-hot encoder is as follows.

```
class OneHotEncoder():
    def __init__(self):
        self.char2token = {}
        self.token2char = {}
        for i in range(26):
            self.char2token[chr(ord('a')+i)] = i
            self.token2char[i] = chr(ord('a')+i)
        self.char2token['SOS'] = 26
        self.token2char[26] = 'SOS'
        self.char2token['EOS'] = 27
        self.token2char[27] = 'EOS'
        self.num_token = 26 + 2

    def tokenize(self, word):
        chars = ['SOS'] + list(word) + ['EOS']
        return [self.char2token[char] for char in chars]

    def inv_tokenize(self, vec, show_token=False, check_end=True):
        word = ''
        for v in vec:
            char = self.token2char[v.item()]
            if len(char) > 1:
                ch = f'<{char}>' if show_token else ''
            else:
                ch = char
            word += ch
            if check_end and char == 'EOS':
                break
        return word
```

Regarding the dataset, for a given index i^{th}, we return the i^{th} word in the plain text file and its English tense. To digitize the English tense, we map each tense into a number. The mapping relations are as follows.

- simple present: 0
- third person: 1
- present progressive: 2
- simple past: 3

The implementation of our custom dataset is as follows.

```
class WordDataset(Dataset):
    def __init__(self, txt_dir, mode='train'):
        txt_path = os.path.join(txt_dir, f'{mode}.txt')
```

```
        self.data = np.loadtxt(txt_path, dtype=str)
        self.tense = ['sp', 'tp', 'pg', 'p']
        self.mode = mode

        if mode == 'train':
            self.data = self.data.reshape(-1)

        if mode == 'test':
            self.conversion = np.array([
                [0, 3],
                [0, 2],
                [0, 1],
                [0, 1],
                [3, 1],
                [0, 2],
                [3, 0],
                [2, 0],
                [2, 3],
                [2, 1]
            ])

    def __getitem__(self, idx):
        if self.mode == 'train':
            return self.data[idx], idx%len(self.tense)
        else:
            return self.data[idx,0],\
                    self.conversion[idx,0],\
                    self.data[idx,1],\
                    self.conversion[idx, 1]

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

3.2 Encoder

Figure 1 is the architecture of the encoder. $c1$ is the condition of input, which is English tense in this task. $hidden$ and $cell$ are two inputs of LSTM, which are initialized with zeros and concatenate with condition $c1$. Two outputs from LSTM will be sent into four different fully connected layers to produce mean and log variance. We estimate log variance instead of variance because variance is always greater than or equal to zero. However, the outputs of the neural network may be negative values. For the convenience of implementation, we combine the reparameterization trick with the encoder. To adopt the reparameterization trick, we sample a noise ϵ from $\mathcal{N}(0, I)$ with `torch.normal` and compute the latent vector z by an equation $e^{\log var./2} + mean$. The implementation of the encoder is as follows.

```
class EncoderRNN(nn.Module):
    def __init__(
            self, input_size, hidden_input_size,
            cond_size, hidden_cond_size, latent_size):
        super(EncoderRNN, self).__init__()
```

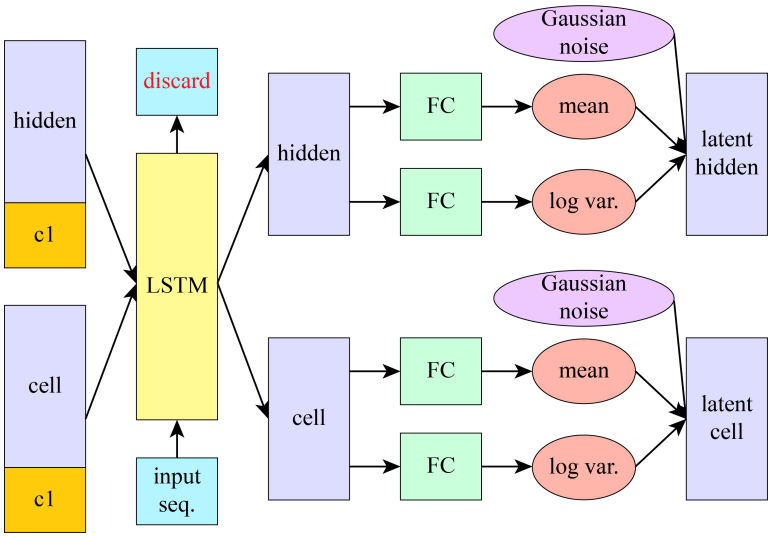

Figure 1: Encoder architecture

```
        self.hidden_input_size = hidden_input_size
        self.hidden_cond_size = hidden_cond_size
        self.latent_size = latent_size

        self.embed_word = nn.Embedding(input_size, hidden_input_size)
        self.embed_cond = nn.Embedding(cond_size, hidden_cond_size)
        self.lstm = nn.LSTM(hidden_input_size, hidden_input_size)
        self.mean_hidden= nn.Linear(hidden_input_size, latent_size)
        self.logvar_hidden = nn.Linear(hidden_input_size, latent_size)
        self.mean_cell = nn.Linear(hidden_input_size, latent_size)
        self.logvar_cell = nn.Linear(hidden_input_size, latent_size)

    def forward(self, inputs, hidden, cell, cond):
        cond_embd = self.embed_cond(cond).reshape(1, 1, -1)
        hidden = torch.cat((hidden, cond_embd), dim=2)
        cell = torch.cat((cell, cond_embd), dim=2)

        inputs_embd = self.embed_word(inputs)
        inputs_embd = inputs_embd.reshape(-1, 1, self.hidden_input_size)
        outputs, (hidden, cell) = self.lstm(inputs_embd, (hidden, cell))

        m_hidden = self.mean_hidden(hidden)
        lv_hidden = self.logvar_hidden(hidden)
        epsilon_hidden = torch.normal(
                            torch.zeros(self.latent_size),
                            torch.ones(self.latent_size)).to(device)
        z_hidden = torch.exp(lv_hidden/2)*epsilon_hidden + m_hidden

        m_cell = self.mean_cell(cell)
        lv_cell = self.logvar_cell(cell)
        epsilon_cell = torch.normal(
```

```
                            torch.zeros(self.latent_size),
                            torch.ones(self.latent_size)).to(device)
        z_cell = torch.exp(lv_cell/2)*epsilon_cell + m_cell

        return m_hidden, lv_hidden, z_hidden, m_cell, lv_cell, z_cell

    def init_hidden(self):
        s = self.hidden_input_size-self.hidden_cond_size
        return torch.zeros(1, 1, s, device=device)

    def init_cell(self):
        s = self.hidden_input_size-self.hidden_cond_size
        return torch.zeros(1, 1, s, device=device)
```

3.3 Decoder

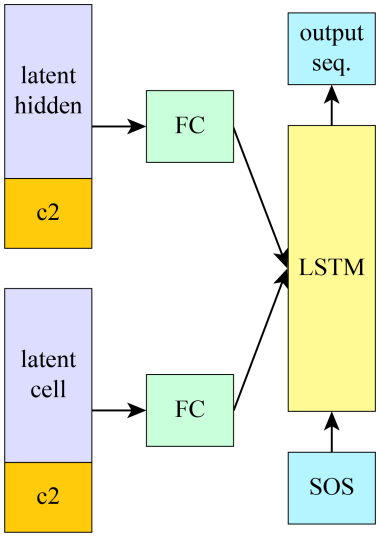

Figure 2: Decoder architecture

Figure 2 is the architecture of the decoder, which is much simpler than the encoder. First, concatenate latent vectors getting from encoder with condition $c2$, which is the target condition. Second, use a fully connected layer to transform the latent vectors from latent space to hidden space. Then, we use another LSTM to produce the result sequence with encoded input and start-of-sequence token. The implementation of the decoder is as follows.

```
class DecoderRNN(nn.Module):
    def __init__(
            self, input_size, hidden_input_size,
            hidden_cond_size, latent_size):
        super(DecoderRNN, self).__init__()
        self.input_size = input_size
        self.hidden_input_size = hidden_input_size
        self.hidden_cond_size = hidden_cond_size
        self.latent_size = latent_size
```

```
        s = latent_size+hidden_cond_size
        self.latent_hidden_to_hidden = nn.Linear(s, hidden_input_size)
        self.latent_cell_to_hidden = nn.Linear(s, hidden_input_size)
        self.embed_word = nn.Embedding(input_size, hidden_input_size)
        self.lstm = nn.LSTM(hidden_input_size, hidden_input_size)
        self.out = nn.Linear(hidden_input_size, input_size)

    def forward(self, inputs, hidden, cell):
        inputs_embd = self.embed_word(inputs)
        inputs_embd = inputs_embd.reshape(-1, 1, self.hidden_input_size)
        inputs_embd = F.relu(inputs_embd)
        output, (hidden, cell) = self.lstm(inputs_embd, (hidden, cell))
        output = self.out(output).reshape(-1, self.input_size)
        return output, hidden, cell

    def init_hidden(self, z_hidden, cond_embd):
        latent_hidden = torch.cat((z_hidden, cond_embd), dim=2)
        return self.latent_hidden_to_hidden(latent_hidden)

    def init_cell(self, z_cell, cond_embd):
        latent_cell = torch.cat((z_cell, cond_embd), dim=2)
        return self.latent_cell_to_hidden(latent_cell)
```

3.4 Text generation

For text generation, we use `torch.normal` to sample 100 Gaussian noise from $\mathcal{N}(0, I)$. The implementation of text generation is as follows. The `tokenizer` is an one-hot encoder. The `inference` inside `generate_words` is used to generate the output sequence with the sampled latent vectors. For the details of `inference`, please refer to `train.py`.

```
def generate_words(encoder, decoder, latent_size, tokenizer):
    decoder.eval()
    sos_token = tokenizer.char2token['SOS']
    eos_token = tokenizer.char2token['EOS']
    words = []
    with torch.no_grad():
        for i in range(100):
            z_h = torch.normal(
                        torch.zeros(1, 1, latent_size),
                        torch.ones(1, 1, latent_size)).to(device)
            z_c = torch.normal(
                        torch.zeros(1, 1, latent_size),
                        torch.ones(1, 1, latent_size)).to(device)
            words.append([])
            for cond in range(4):
                cond = to_long([cond]).to(device)
                cond_embd = encoder.embed_cond(cond)
                cond_embd = cond_embd.reshape(1, 1, -1).to(device)
                output = inference(
                            decoder, z_h, z_c, cond_embd,
```

```
                                None, None, sos_token, eos_token, MAX_LEN)
                output_token = torch.max(torch.softmax(output, dim=1), 1)
                output_word = tokenizer.inv_tokenize(output_token[1])
                words[-1].append(output_word)
    return words
```

3.5 Hyper-parameters

- Training epochs: 150
- Learning rate: 0.05 (fixed)
- Optimizer: SGD
- Loss function: cross entropy + KL divergence
- LSTM hidden size: 256
- Condition embedded size: 8
- Latent size: 32

3.6 Teacher forcing/KL loss annealing

For teacher forcing, the ratios in the first 20% of iterations are set to one and linearly decrease from one to the given final ratio in the remaining 80% of iterations.

For KL annealing, we use one-cycle (monotonic) and two-cycle. In each cycle, weights in the first 20% of iterations are set to zero and linearly increase from zero to the given final weight in the remaining 80% of iterations. The implementation of teacher forcing and KL loss annealing are as follows.

```
def kld_weight_annealing(epoch, num_epoch, final_weight, cycle_num):
    num_epoch //= cycle_num
    epoch %= num_epoch
    thres = int(num_epoch*0.2)
    if epoch < thres:
        w = 0
    else:
        w = (epoch-thres) * final_weight / (num_epoch-thres)
    w = max(0, w)
    w = min(1, w)
    return w

def teacher_forcing_ratio(epoch, num_epoch, final_ratio):
    thres = int(num_epoch*0.2)
    if epoch < thres:
        r = 1
    else:
        r = final_ratio
            + (num_epoch-epoch)*(1-final_ratio)/(num_epoch-thres)
    r = max(0, r)
    r = min(1, r)
    return r
```

For the experiments, we use two different final ratios for teacher forcing, 0.8 and 0.4. In each case, we also try one-cycle KL annealing and two-cycle KL annealing. Thus, there will be four experiment trials in total.

Figure 3 is the visualization of teacher forcing ratio and KL annealing weight with different final ratio and number of annealing cycles.

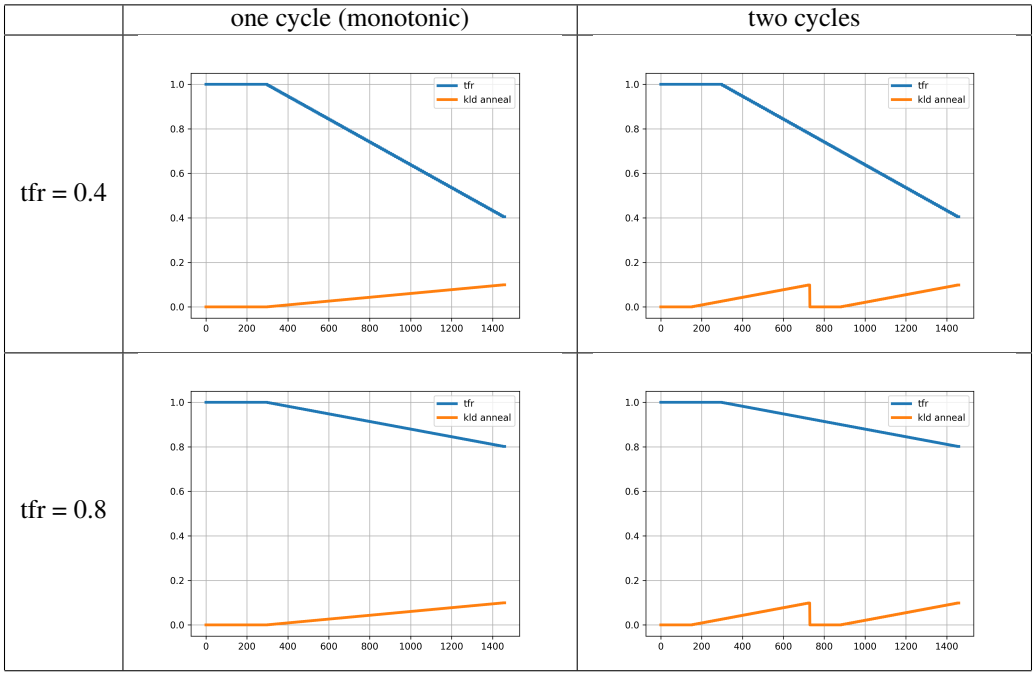

Figure 3: Different final teacher forcing ratios (tfr) and number of annealing cycles

4 Results

During the training process, we save the weights of the model each 500 iterations. To determine the best weights, we use the scoring criteria TA provided and choose one with the highest score.

4.1 Tense conversion

- final teacher forcing ratio = 0.4, one cycle

```
input: begin
target: begins
prediction: behonds

input: split
target: splitting
prediction: splitting

...

input: abet
target: abetting
prediction: abetting

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

```
Average BLEU-4 score: 0.7478
```

- final teacher forcing ratio = 0.4, two cycles

```
input: abet
target: abetting
prediction: abetting

input: functioning
target: function
prediction: function

...

input: flared
target: flare
prediction: flare

input: expend
target: expends
prediction: expends

Average BLEU-4 score: 0.9286
```

- final teacher forcing ratio = 0.8, one cycle

```
input: functioning
target: function
prediction: function

input: begin
target: begins
prediction: bends

...

input: flared
target: flare
prediction: flare

input: functioning
target: functioned
prediction: fincited

Average BLEU-4 score: 0.6550
```

- final teacher forcing ratio = 0.8, two cycles

```
input: functioning
target: function
```

```
prediction: function

input: split
target: splitting
prediction: splitting

...

input: healing
target: heals
prediction: heals

input: abet
target: abetting
prediction: abetting

Average BLEU-4 score: 0.9286
```

4.2 Text generation

- final teacher forcing ratio = 0.4, one cycle

```
['gmeer', 'smarges', 'gmeering', 'gmeeded']
['condece', 'condeces', 'condeding', 'condeded']
['repail', 'resalls', 'repailing', 'repalled']
['prewed', 'prawls', 'prewing', 'prawled']
['sleaph', 'slaspens', 'slaspening', 'slaspened']
...
['neall', 'neals', 'nealling', 'nealled']
['glash', 'glashes', 'glashing', 'glashed']
['fash', 'fashs', 'fashing', 'fashed']
['fickle', 'fickles', 'flackling', 'flackered']
['aaroid', 'aaroies', 'aaroiding', 'aarosed']
Gaussian score: 0.0600
```

- final teacher forcing ratio = 0.4, two cycles

```
['revolve', 'revolves', 'retouching', 'revolved']
['creak', 'creaks', 'creaking', 'creaked']
['capitulate', 'capitules', 'characting', 'capituled']
['yearn', 'yearns', 'yearing', 'yearned']
['consent', 'consents', 'consenting', 'consented']
...
['continulate', 'continues', 'continuing', 'continued']
['arouse', 'arouses', 'arousing', 'aroused']
['swold', 'swols', 'swooping', 'swored']
['improve', 'improves', 'improving', 'improved']
['encounter', 'encounts', 'encounting', 'instracted']
Gaussian score: 0.3000
```

- final ratio = 0.8, one cycle

```
['commence', 'commences', 'commencing', 'commenced']
['admonish', 'admonishes', 'admonishing', 'admonted']
['indicate', 'indicates', 'indicating', 'indicated']
['spare', 'spares', 'arose', 'spared']
['sonserve', 'sonserves', 'hindering', 'softened']
...
['repeat', 'gulps', 'repeating', 'grew']
['escape', 'escapes', 'escaping', 'escaped']
['retrace', 'retraces', 'retracing', 'retraced']
['disappploy', 'disappploys', 'disappploying', 'disapplied']
['recore', 'recores', 'criewing', 'cried']
Gaussian score: 0.2800
```

- final ratio = 0.8, two cycles

```
['swammer', 'swamms', 'ascending', 'swammered']
['plunge', 'plunges', 'plunging', 'plunged']
['collapse', 'collapses', 'collapsing', 'collapsed']
['blink', 'blinks', 'blinking', 'bled']
['wrench', 'wrenches', 'wrenching', 'wrenched']
...
['crumble', 'writes', 'grimacing', 'glistened']
['hestinue', 'hestinues', 'hesting', 'hestinued']
['discontinue', 'discontinues', 'discontinuing', 'discontinued']
['remain', 'remains', 'remaining', 'remained']
['betch', 'betches', 'belching', 'betched']
Gaussian score: 0.3900
```

4.3 Visualization

Figure 4 is the visualization of cross entropy loss, KL loss, BLEU-4 score, and Gaussian score. The blue line is the KL loss while the orange line is thee cross entropy loss. And the green and brown points are BLEU-4 score and Gaussian score, respectively.

5 Discussion

5.1 KL loss annealing mechanism

In the early stage of the training process, the model only focuses on minimizing the cross entropy loss (reconstruction). We can see that the cross entropy loss decreases while the BLEU-4 score increases. Meanwhile, the distance between the latent distribution and $\mathcal{N}(0, I)$ becomes higher since it might be difficult to approximate the latent distribution with a simple Gaussian distribution in the beginning. Thus, the KL loss (regularization) grows up. When the weight of KL loss increases, the model starts to take the KL loss into consideration. In this time, the latent distribution is getting closer to $\mathcal{N}(0, I)$. But the cross entropy loss increases since the optimal latent distribution might be more complicated than a Gaussian. There is an antagonism between the cross entropy loss and the KL loss. To tackle this problem, we can use a more complex distribution to approximate latent distribution, increase the number of annealing cycles, or try to increase the teacher forcing ratio.

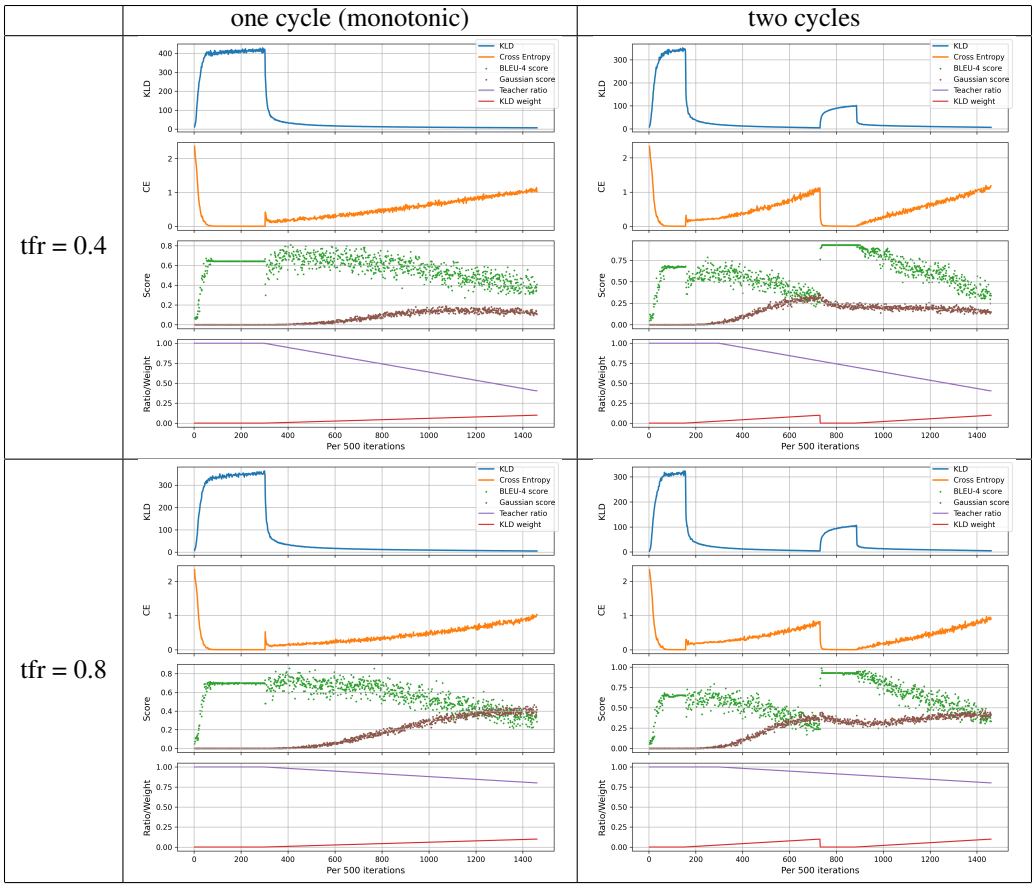

Figure 4: Visualizations

5.2 Different final teacher forcing ratios

When comparing the results between difference final teacher forcing ratios, we discover that a higher teacher forcing ratio may increase the Gaussian score. But we have not figured out the reason yet.

5.3 One annealing cycle v.s. two annealing cycles

It looks like that we are fine-tuning the model cycle-by-cycle. In each cycle, we minimize the cross entropy loss first and then minimize the KL loss. But when we try to lower the KL loss, the cross entropy loss increases. In the next cycle, we minimize the cross entropy loss and KL loss again based on the previous cycle. We can see that both BLEU-4 score and Gaussian score become higher cycle after cycle.