DLP Homework 5

電控碩 0860077 王國倫

1. Introduction

In this lab, we need to implement a conditional seq2seq VAE (Fig.1) for English tense conversion and generation. This English tense are four type, including simple present, third person, present progressive, simple past, these conditions will concatenate hidden and cell before input the encoder and decoder. In addition, we can also manually generate a Gaussian noise vector and feed it with different tenses to the decoder and generate a word those tenses. The encoder and decoder must be implemented by nn.LSTM, cannot use attention me-chanism in this lab, create dataloader for this lab, and use teacher forcing ratio and KL annealing schedule(monotonic and cycle) in the training, output training loss, test BLEU score and result of generated by Gaussian noise with 4 tenses.

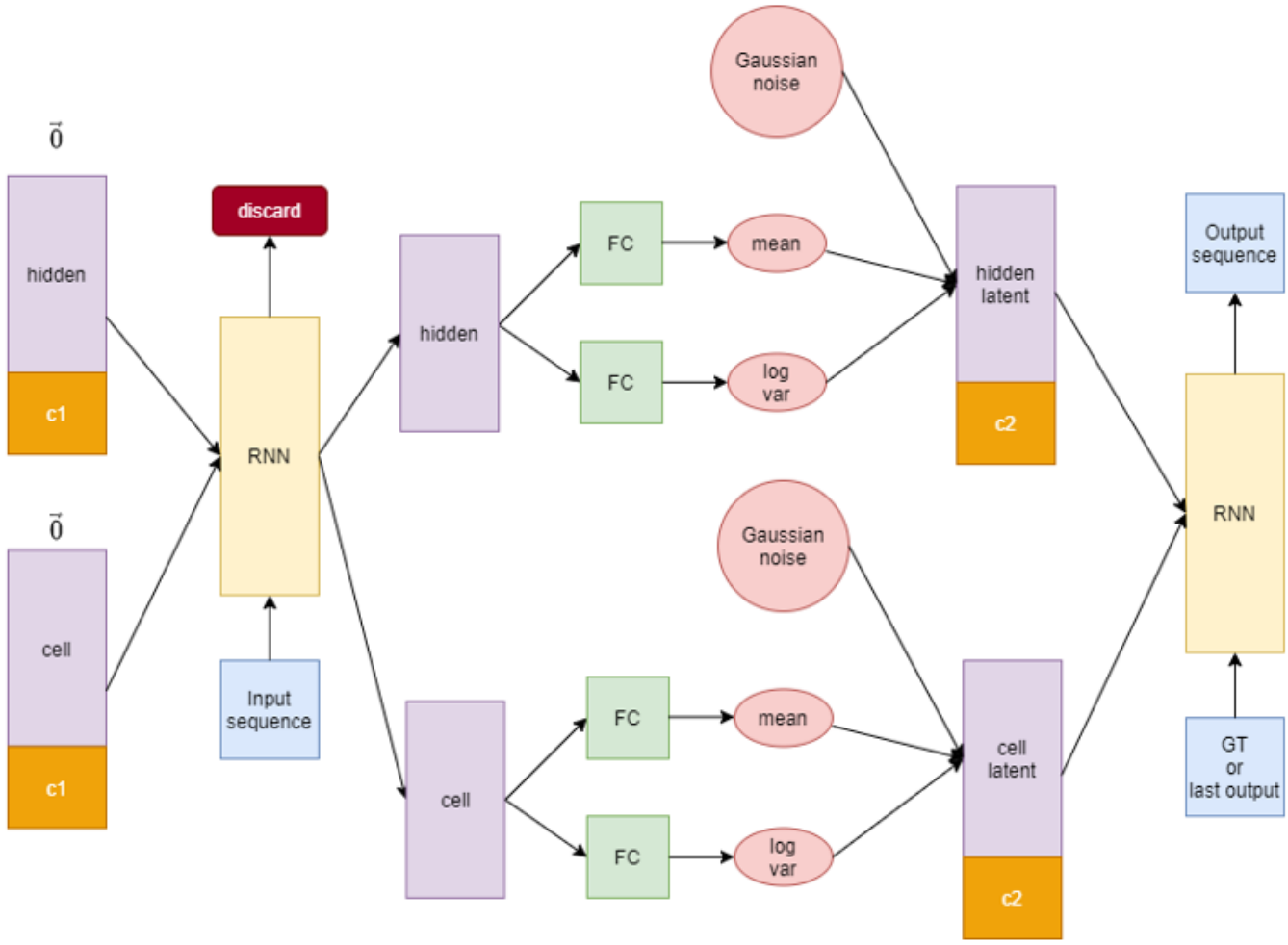


Fig.1 The overall of sequence-to-sequence CVAE architecture

2. Derivation of CVAE

The object of CVAE is maximum data likelihood:

Maximum $\sum_x log(p_\theta(x|c))$

$$
\begin{aligned}
log(p_\theta(x|c)) &= \int_z q_\varphi(z|x,c) log(p_\theta(x|c)) dz \\
&= \int_z q_\varphi(z|x,c) log(\frac{p_\theta(x,z|c)}{p_\theta(z|x,c)}) dz \\
&= \int_z q_\varphi(z|x,c) log(\frac{p_\theta(x,z|c) q_\varphi(z|x,c)}{p_\theta(z|x,c) q_\varphi(z|x,c)}) dz \\
&= \int_z q_\varphi(z|x,c) log(\frac{p_\theta(x,z|c)}{q_\varphi(z|x,c)}) dz + \int_z q_\varphi(z|x,c) log(\frac{q_\varphi(z|x,c)}{p_\theta(z|x,c)}) dz \\
&= \int_z q_\varphi(z|x,c) log(\frac{p_\theta(x,z|c)}{q_\varphi(z|x,c)}) dz + KL(q_\varphi(z|x,c)||q_\varphi(z|x,c))
\end{aligned}
$$

$\because KL(q_\varphi(z|x,c)||q_\varphi(z|x,c)) \geq 0$

$\therefore log(p_\theta(x|c)) \geq \int_z q_\varphi(z|x,c) log(\frac{p_\theta(x,z|c)}{q_\varphi(z|x,c)}) dz = ELBO$ (Evidence Lower Bound)

$\because$ Maximum $log(p_\theta(x|c)) \rightarrow$ Maximum **ELBO**

$$
\begin{aligned}
ELBO &= \int_z q_\varphi(z|x,c) log(\frac{p_\theta(x,z|c)}{q_\varphi(z|x,c)}) dz \\
&= \int_z q_\varphi(z|x,c) log(\frac{p_\theta(x|z,c) p_\theta(z|c)}{q_\varphi(z|x,c)}) dz \\
&= \int_z q_\varphi(z|x,c) log(p_\theta(x|z,c)) dz + \int_z q_\varphi(z|x,c) log(\frac{p_\theta(z|c)}{q_\varphi(z|x,c)}) dz \\
&= E_{z \sim q_\varphi(z|x,c)}[log(p_\theta(x|z,c))] - KL(q_\varphi(z|x,c)||p_\theta(z|c))
\end{aligned}
$$

where p_θ is decoder, q_φ is encoder, $p_\theta(z|c)$ is normal distribution($N(0,I)$).

The first term is reconstruction result, calculate cross entropy loss, and the second term is requlariza-tion result, calculate KL divergence loss between latent and normal distrib-ution.

$$
\begin{aligned}
&KL\Big(N(\mu,\sigma^2)||N(0,1)\Big) \\
&= \int \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(x-\mu)^2/2\sigma^2} \left(\ln \frac{e^{-(x-\mu)^2/2\sigma^2}/\sqrt{2\pi\sigma^2}}{e^{-x^2/2}/\sqrt{2\pi}} \right) dx \\
&= \int \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(x-\mu)^2/2\sigma^2} \ln \left[\frac{1}{\sqrt{\sigma^2}} \exp\Big(\frac{1}{2}\left[x^2 - (x-\mu)^2/\sigma^2\right]\Big) \right] dx \\
&= \frac{1}{2} \int \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(x-\mu)^2/2\sigma^2} \Big[-\ln\sigma^2 + x^2 - (x-\mu)^2/\sigma^2 \Big] dx \\
&= \frac{1}{2}\Big(\mu^2 + \sigma^2 - \ln\sigma^2 - 1\Big)
\end{aligned}
$$

Fig.2 KL divergence derivation

In practice, however, it's better to model σ^{2} as $ln(\sigma^{2})$, as it is more numerically stable to take exponent compared to computing log. Hence, the final KL divergence term is:

$$KL(N(\mu,\sigma^2),N(0,1)) = 0.5(\mu^2 + exp(\sigma^2) - \sigma^2 - 1)$$

3. Implementation details

A. Dataloader

In order to transform each alphabet to number, I create dict structure transform each other, and transform word to tensor and tensor to word.

```
#!/usr/bin/env python3

import torch

class Converter():
    def __init__(self):

        self.te2nb = {"sp":0, "tp":1, "pg":2, "p":3}
        self.nb2te = {0:"sp", 1:"tp", 2:"pg", 3:"p"}
        self.__setup_al2nb()
        self.__setup_nb2al()

    def __setup_al2nb(self):
        self.al2nb = {chr(i+97):i+3 for i in range(26)}
        self.al2nb.update({"SOS":0, "EOS":1, "PAD":2})
```

```
    def __setup_nb2al(self):
        self.nb2al = {i+3:chr(i+97) for i in range(26)}
        self.nb2al.update({0:"SOS", 1:"EOS", 2:"PAD"})

    def word2tensor(self, word):
        ten = []
        for i in word:
            ten.append(self.al2nb[i])
        ten.append(self.al2nb["EOS"])
        return torch.tensor(ten)

    def tensor2word(self, tensor):
        word  = ""
        for i in range(tensor.shape[0]):
            word += self.nb2al[tensor[i].item()] if \
            tensor[i].item() >= 3 else ""
        return word
```

The dataloader show below, the train dataset will return one word with its tense, and the test dataset will return input word with its tense and target word with its tense.

The get_max_length function will find the maximal length of word, and collate_fn function handle padding of each batch, make each batch is same seg_len.

```
#!/usr/bin/env python3
from torch.utils.data.dataset import Dataset
from .converter import Converter
import torch
from torch.nn.utils.rnn import pad_sequence

class WordDataset(Dataset):
    def __init__(self, mode):
        self.converter = Converter()
        self.__readdata(mode)
        self.max_length = self.get_max_length()
        self.mode = mode

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

    def __getitem__(self, index):

        if(self.mode == "train"):
            return self.converter.word2tensor(self.words[index]),
            self.tenses[index]
        else:
            return self.converter.word2tensor(self.words[index][0]), \
                   torch.tensor(self.tenses[index][0]), \
                   self.converter.word2tensor(self.words[index][1]), \
                   torch.tensor(self.tenses[index][1])

    def __readdata(self, mode):
```

```
        path = "./dataset/train.txt" if mode == "train" \
        else "./dataset/test.txt"

        self.words = []
        self.tenses = []

        with open(path, "r") as file:

            if(mode == "train"):
                for line in file:
                    self.words.extend(line.strip('\n').split(' '))
                    self.tenses.extend([i for i in range(4)])
            else:
                for line in file:
                    self.words.append(line.strip('\n').split(' '))

                tenses = [["sp","p"], ["sp","pg"],
                ["sp","tp"], ["sp","tp"], ["p","tp"],
                ["sp","pg"], ["p","sp"], ["pg","sp"],
                ["pg","p"], ["pg","tp"]]
                for te in tenses:
                    self.tenses.append([self.converter.te2nb[t] for t in te])

    def get_max_length(self):

        max_length = 0
        for word in self.words:
            max_length = max(max_length, len(word))
        return max_length

    def collate_fn(self, batch):

        pad_data = pad_sequence([word for word, _ in batch],
        batch_first=True, padding_value=2)
        # print(batch)
        _, tense = zip(*batch)

        return pad_data, torch.tensor(tense)
```

B. CVAE

The CVAE architecture composed of encoder and decoder, the input will embedding by nn.Embedding, because use 2 to padding the word, nn.Embedding will ignore 2, then the hidden and cell will concatenate embedded condition, input data, hidden and cell to lstm, output next hidden and cell, the decoder will output predict distribution.

The CVAE need input word, its tense and teacher forcing ratio, embedding word and tense input encoder to generate latent space, use fully connected layer output 32 dimension mean and log variance, use log variance make sure the value is positive, then reparameterization trick sampling same dimension hidden and cell as decoder input, fed decoder to reconstruct the word.

This lab, I use batch to train, so the termination condition is each generated word output EOS, then finish mini batch training.

The CVAE class also have inference and generate function, inference need input word with its tense and target word with its tense, calculate test dataset BLEU score. Generate need input Gaussian noise for hidden and cell and four tenses to generate corresponding word.

```
#!/usr/bin/env python3

import torch.nn as nn
import torch
import torch.nn.functional as F

class CVAE(nn.Module):
    def __init__(self, hidden_size, use_cuda, input_size=29, latent_size=32,
    tense=4, condition_size=8):
        super(CVAE, self).__init__()
        self.condition_size = condition_size
        self.hidden_size = hidden_size
        self.te_em = nn.Embedding(tense, condition_size)
        self.encoder = Encoder(input_size, hidden_size, use_cuda)
        self.decoder = Decoder(input_size, hidden_size, use_cuda)
        self.hid2mean = nn.Linear(hidden_size, latent_size)
        self.hid2logvar = nn.Linear(hidden_size, latent_size)
        self.cell2mean = nn.Linear(hidden_size, latent_size)
        self.cell2logvar = nn.Linear(hidden_size, latent_size)
        self.lat2hid = nn.Linear(latent_size, hidden_size - condition_size)
        self.lat2cell =  nn.Linear(latent_size, hidden_size - condition_size)
        self.device = torch.device("cuda" if use_cuda else "cpu")

    def forward(self, input, condition, use_teacher_forcing):

        # get batch_size and length
        batch_size, length = input.shape

        # init encoder hidden and cell state (1 * batch_size * (hidden_size - conditio
        en_hidden = self.encoder.inithidden(batch_size,
        self.hidden_size - self.condition_size)
        en_cell_state = self.encoder.initcell(batch_size,
        self.hidden_size - self.condition_size)

        # tense embedding (batch_size * 1 * condition_size) -> (1 * batch_size  * cond
        tense_embedding = self.te_em(condition.view(-1, 1))
        tense_embedding = tense_embedding.permute(1, 0, 2)

        # cat condition into hidden and cell state
        en_hidden = torch.cat((en_hidden, tense_embedding), dim=2)
        en_cell_state = torch.cat((en_cell_state, tense_embedding), dim=2)
        # en_cell_state = torch.zeros_like(en_hidden, device=self.device)

        # encoder lstm
        for i in range(length):
            _, en_hidden, en_cell_state = self.encoder(input[:,i].view(-1,1),
```

```
            en_hidden, en_cell_state)

        # fully connection extract mean and logvar (1 * batch_size * latent_size)
        hid_mean = self.hid2mean(en_hidden)
        hid_logvar = self.hid2logvar(en_hidden)
        cell_mean = self.cell2mean(en_cell_state)
        cell_logvar = self.cell2logvar(en_cell_state)

        # reparameterize (1 * batch_size * latent_size)
        hid_latent = self.reparameterize(hid_mean, hid_logvar)
        cell_latent = self.reparameterize(cell_mean, cell_logvar)

        # decoder hidden and cell state (1 * batch_size * hidden_size)
        de_hidden = self.lat2hid(hid_latent)
        de_cell_state = self.lat2cell(cell_latent)
        de_hidden = torch.cat((de_hidden, tense_embedding), dim=2)
        de_cell_state = torch.cat((de_cell_state, tense_embedding), dim=2)
        # de_cell_state = torch.zeros_like(de_hidden, device=self.device)

        # decoder input(SOS = 0)
        de_input = torch.zeros(batch_size, 1, dtype=torch.long,
        device=self.device)

        # PAD tensor
        pad = torch.tensor([2 for i in range(batch_size)],
        device=self.device).view(-1,1)

        # decoder lstm
        for i in range(length):
            output, de_hidden, de_cell_state = self.decoder(de_input,
            de_hidden, de_cell_state)

            # reconstruction
            number = torch.max(output, dim=2)[1]
            predict = torch.cat((predict, number), dim=1) if i!=0 else number

            # predict distribution (batch_size * 1ength * input_size)
            distribution = torch.cat((distribution,
            output), dim=1) if i!=0 else output

            # teacher forcing
            if(use_teacher_forcing):
                de_input = input[:,i].view(-1,1)
            else:
                de_input = number

            # transform eos to pad
            de_input = torch.where(de_input!=1, de_input, pad)

            # record eos
            eos = torch.logical_or(eos, torch.eq(de_input, pad)) \
            if i!=0 else torch.eq(de_input, pad)

            if(torch.equal(eos, torch.tensor([True for i in range(batch_size)],
            device=self.device).view(-1,1))):
```

```
                break

        return predict, distribution, hid_mean, hid_logvar,
        cell_mean, cell_logvar

    def reparameterize(self, mean, logvar):

        std = torch.exp(0.5 * logvar)
        eps = torch.randn_like(std)
        latent = mean + eps * std
        return latent

class Encoder(nn.Module):
    def __init__(self, input_size, hidden_size, use_cuda):
        super(Encoder, self).__init__()

        self.embedding = nn.Embedding(input_size, hidden_size, padding_idx=2)
        self.lstm = nn.LSTM(hidden_size, hidden_size, batch_first=True)
        self.device = torch.device("cuda" if use_cuda else "cpu")

    def forward(self, input, hidden, cell_state):
        embedded = self.embedding(input)
        output, (hidden, cell_state) = self.lstm(embedded, (hidden, cell_state))
        return output, hidden, cell_state

    def inithidden(self, batch_size, size):
        return torch.zeros(1, batch_size, size, device=self.device)

    def initcell(self, batch_size, size):
        return torch.zeros(1, batch_size, size, device=self.device)

class Decoder(nn.Module):
    def __init__(self, input_size, hidden_size, use_cuda):
        super(Decoder, self).__init__()
        self.hidden_size = hidden_size

        self.embedding = nn.Embedding(input_size, hidden_size, padding_idx=2)
        self.lstm = nn.LSTM(hidden_size, hidden_size, batch_first=True)
        self.out = nn.Linear(hidden_size, input_size)
        self.softmax = nn.LogSoftmax(dim=2)
        self.device = torch.device("cuda" if use_cuda else "cpu")

    def forward(self, input, hidden, cell_state):
        output = self.embedding(input)
        output = F.relu(output)
        output, (hidden, cell_state) = self.lstm(output, (hidden, cell_state))
        output = self.softmax(self.out(output))
        return output, hidden, cell_state

    def inithidden(self, batch_size, size):
        return torch.zeros(1, batch_size, size, device=self.device)

    def initcell(self, batch_size, size):
        return torch.zeros(1, batch_size, size, device=self.device)
```

C. Traning and evaluate

In the training, calculate CE loss, KLD loss, total loss, train BLEU score and test BLEU score, save the best test BLEU score, and use wandb record these information.

In the evaluate, calculate test dataset BLEU score and output generate word.

```
# training
    def training(self):

        best_model_weight = None
        best_bleu = 0.0

        # wandb.watch(self.model)

        for e in range(1, self.args.epochs + 1):

            total_loss = 0.0
            CE_loss = 0.0
            KLD_loss = 0.0
            BLEU_score = 0.0

            tbar = tqdm(self.trainloader)
            self.model.train()
            for i, (word, condition) in enumerate(tbar):

                # word (batch_size * length), condition (batch_size)
                word, condition = Variable(word),Variable(condition)

                # using cuda
                if self.args.cuda:
                    word, condition = word.cuda(), condition.cuda()

                # teacher forcing and kld wieght
                use_teacher_forcing = True if random.random() < \
                teacher_forcing_ratio(e, self.args.epochs) else False
                kld_weight = get_kld_weight(self.args.kld_loss_type,
                self.args.epochs, e, self.args.threshold)

                # predict
                prediction, distribution, hid_mean, hid_logvar,
                cell_mean, cell_logvar = self.model(word, condition,
                use_teacher_forcing)

                # calculate loss
                CE, KLD = self.loss_function(self.criterion, distribution, word,
                hid_mean, hid_logvar, cell_mean, cell_logvar)
                loss = CE + kld_weight * KLD
                CE_loss += CE.item()
                KLD_loss += KLD.item()
                for k in range(prediction.shape[0]):
```

```
                    predict, target = self.converter.tensor2word(prediction[k]),
                    self.converter.tensor2word(word[k])
                    BLEU_score += self.compute_bleu(predict, target)

                # update
                self.optimizer.zero_grad()
                loss.backward()
                self.optimizer.step()
                total_loss += loss.item()

                # show loss
                tbar.set_description('Total loss: {0:.4f}, CE loss: {1:.4f},\
                KLD loss: {2:.4f}, BLEU score: {3:.4f}' \
                    .format(total_loss / (i + 1), CE_loss / (i + 1),
                    KLD_loss / (i + 1),
                    BLEU_score / (prediction.shape[0] * (i + 1))))

            # evaluate
            test_predict, bleu_score = self.evaluate()

            # record
            wandb.log({"teacher forcing ratio": teacher_forcing_ratio(e,
            self.args.epochs)})
            wandb.log({"kld weight": kld_weight})
            wandb.log({"total loss": total_loss / (i + 1)})
            wandb.log({"CE loss": CE_loss / (i +1)})
            wandb.log({"KLD loss": KLD_loss / (i + 1)})
            wandb.log({"Train BLEU score": BLEU_score / (prediction.shape[0] * \
            (i + 1))})
            wandb.log({"Test BLEU score": bleu_score})

            # store best model
            if(bleu_score > best_bleu):
                best_bleu = bleu_score
                best_model_weight = copy.deepcopy(self.model.state_dict())

        # save the best model
        torch.save(best_model_weight,
        os.path.join(self.weight_path, self.file_name + '.pkl'))

        artifact = wandb.Artifact('model', type='model')
        artifact.add_file(os.path.join(self.weight_path,
        self.file_name + '.pkl'))
        self.run.log_artifact(artifact)
        self.run.join()

    # evaluation
    def evaluate(self):

        tbar = tqdm(self.testloader)
        self.model.eval()
        BLEU_score = 0.0
        predict_list = []

        for i, (input_word, input_tense, target_word,
```

```
        target_tense) in enumerate(tbar):
            input_word, input_tense = Variable(input_word),Variable(input_tense)
            target_word, target_tense = Variable(target_word),
            Variable(target_tense)

            # using cuda
            if self.args.cuda:
                input_word, input_tense = input_word.cuda(), input_tense.cuda()
                target_word, target_tense = target_word.cuda(),
                target_tense.cuda()

            with torch.no_grad():

                prediction = self.model.inference(input_word, input_tense,
                target_tense, self.max_length)
                predict, target = self.converter.tensor2word(prediction[0]),
                self.converter.tensor2word(target_word[0])
                BLEU_score += self.compute_bleu(predict, target)
                predict_list.append([predict, target,
                self.converter.tensor2word(input_word[0])])

            tbar.set_description('Test BLEU score: {0:.4f} ' \
            .format(BLEU_score/ (i + 1)))

        print(predict_list)

        return predict_list, BLEU_score / (i + 1)
```

D. loss function and BLEU score

The loss function will calculate cross entropy and KL divergence, especially the hidden and cell need to consider, because hidden and cell are decoder input from reparameterizated latent sample, these need calculate kL divergence together.

The BLEU score provided from sample.py, calculate generated word and target word.

```
    def loss_function(self, criterion, distribution, target, hid_mean, hid_logvar, cell_m

        """
        distribution : batch_size * length * input_size
        target : batch_size * length
        mean : 1 * batch_size * latent_size
        logvar : 1 * batch_size * latent_size
        """
        # cross entropy loss
        batch_size, length, _ = distribution.shape

        CE = criterion(distribution.permute(0, 2, 1), target[:,:length])

        # kl loss
```

```
        KLD_hid = -0.5 * torch.sum(1 + hid_logvar - \
        hid_mean.pow(2) - hid_logvar.exp())
        KLD_cell = -0.5 * torch.sum(1 + cell_logvar - \
        cell_mean.pow(2) - cell_logvar.exp())

        return CE, (KLD_hid + KLD_cell) / batch_size

    def compute_bleu(self, output, reference):
        cc = SmoothingFunction()
        if len(reference) == 3:
            weights = (0.33,0.33,0.33)
        else:
            weights = (0.25,0.25,0.25,0.25)
        return sentence_bleu([reference], output,
        weights=weights,smoothing_function=cc.method1)
```

E. Text generation

I use get_gaussian_score from sample.py to calculate Gaussian score, the Gaussian_score function use torch.randn generate latent and cell tensor, then input these and its tense to generate word, output 400 word and calculate its Gaussian score.

```
    def get_gaussian_score(self, words):
        words_list = []
        score = 0

        yourpath = os.path.join(self.args.save_folder, "dataset", "train.txt")
        with open(yourpath,'r') as fp:
            for line in fp:
                word = line.strip('\n').split(' ')
                words_list.extend(word)
            for i in set(words):
                if(i in words_list):
                    score += 1
        return score/len(words)

    def Gaussian_score(self):

        self.model.eval()
        predict_list = []
        tense = torch.tensor([i for i in range(4)]).to(self.device)

        with torch.no_grad():
            for i in range(100):
                latent = torch.randn(1, 1, 32).to(self.device).expand(1,4,32)
                cell = torch.randn(1, 1, 32).to(self.device).expand(1,4,32)
                predict_word = self.model.generate(tense, latent,
                cell, self.max_length)
                predict_list.extend([self.converter.tensor2word\
```

```
                (predict_word[i]) for i in range(4)])
        print(predict_list)

        print("Gaussian score : {0:.2f}".format(self.get_gaussian_score\
        (predict_list)))
```

F. Teacher forcing ratio and kld weight

In the training, I also need teacher forcing ratio and kld weight, the teacher forcing ratio will decrease 1 to 0, depond on current epoch, kld weight tpye can divide two methods, monotonic and cycle, the picture show Fig.3.

```
#!/usr/bin/env python3

def teacher_forcing_ratio(epoch, epochs):
    """
    epochs : totoal epochs
    epoch : current epoch
    """

    # from 1.0 to 0.0
    teacher_forcing_ratio = 1. - (1. / (epochs - 1)) * (epoch - 1)
    return teacher_forcing_ratio

def get_kld_weight(kl_cost_type, epochs, epoch, threshold):
    """
    kl_cost_type : 'monotonic' or 'cycle'
    epochs : totoal epochs
    epoch : current epoch
    threshold :
        montonic -> threshold for 0.0 to 1.0
        cycle -> each threshold repeat
    """

    if(kl_cost_type == 'monotonic'):
        return (0.25 / (threshold - 1)) * (epoch - 1) if epoch < threshold \
        else 0.25

    elif(kl_cost_type == 'cycle'):
        epoch %= threshold
        return (0.25 / (threshold - 1)) * (epoch - 1) if epoch is not 0 \
        else 0.25

    else:
        raise("kl_cost_type not exist")
```

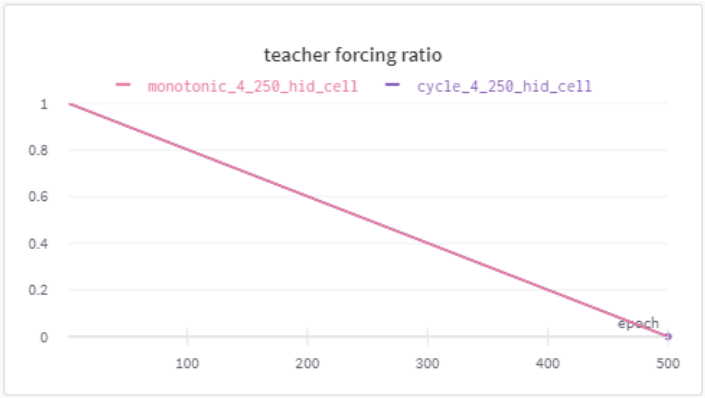


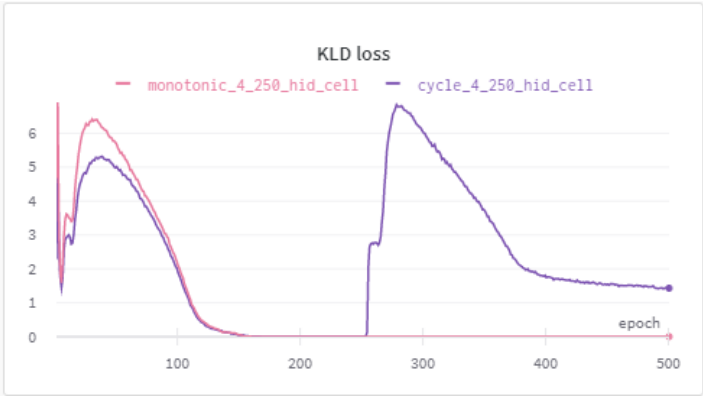
Fig.3 teacher focring ratio and kld weight

G. Hyperparameters

- learning rate : 0.05
- epochs : 500
- batch_size : 4 (shuffle)
- threshold : 250
- hidden size : 256
- kld weight type : cycle and monotonic

4. Results and discussion

A. CE loss, KLD loss and BLEU score



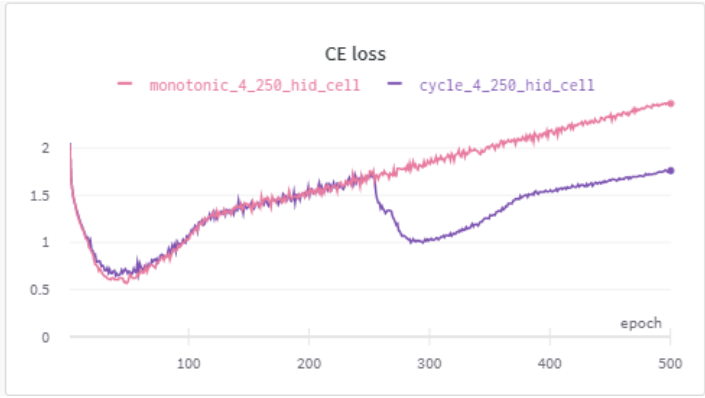


Fig.4 KLD loss and CE loss

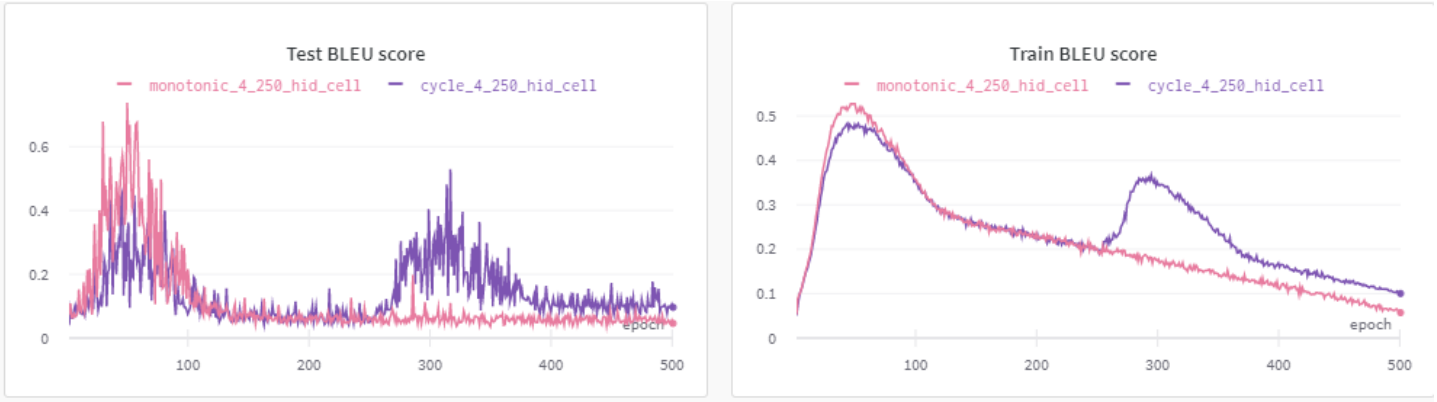


Fig.5 train and test BLEU score

B. Test BLEU and predicted word

Fig.6 test BLEU score and predicted word

The test BLEU score is 0.7320, and its predicted word show below.

[['abandoned', 'abandoned', 'abandon'], ['abetting', 'abetting', 'abet'], ['begins', 'begins', 'begin'], ['expends', 'expends', 'expend'], ['seems', 'sends', 'sent'], ['spliling', 'splitting', 'split'], ['flare', 'flare', 'flared'], ['functionate', 'function', 'functioning'], ['functioned', 'functioned', 'functioning'], ['launches', 'heals', 'healing']]

C. Gaussian score and generated word

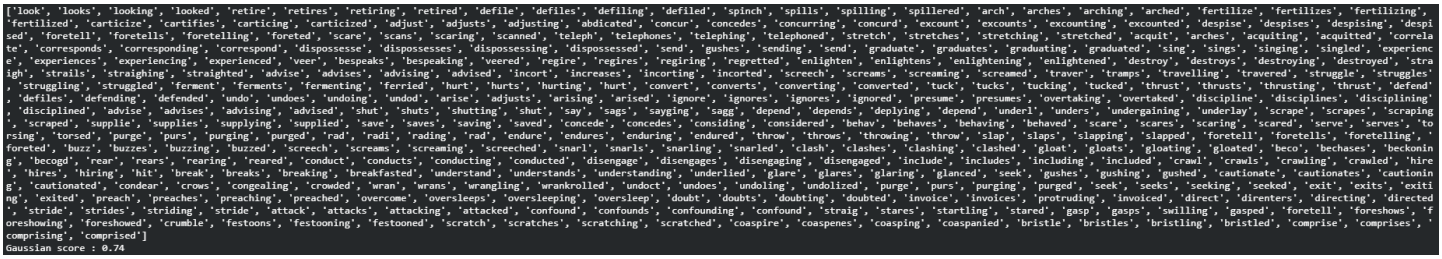


Fig.7 Gaussian score and generated word

The Gaussian score is 0.74, and its generated word show Fig.7, I list some generated word below.

['look', 'looks', 'looking', 'looked', 'retire', 'retires', 'retiring', 'retired', 'defile', 'defiles', 'defiling', 'defiled', 'spinch', 'spills', 'spilling', 'spillered', 'arch', 'arches', 'arching', 'arched', 'fertilize', 'fertilizes', 'fertilizing', 'fertilized', 'carticize', 'cartifies', 'carticing', 'carticized', 'adjust', 'adjusts', 'adjusting', 'abdicated']

D. Discussion

In the begining, the KLD and CE loss is high, also mean the model haven't learned yet, and its train and test BLEU is low. Along with CE loss decrease, the word start predicted well, the BLEU present a significant rise, at the same time, latent distribution and normal distribution are less and less like, so the KLD loss is more high. About 100 epoch, the kld weigth is more and more high, lead to KLD loss control whole loss, KLD loss start decrease, then the CE loss increase and BLEU score is decrease. Finally, the kld weight keep grow, then CE loss increase, KLD loss and BLEU score decrease.

tags: `DLP2021`