Deep Learning and Practice Lab 5

309552007

袁鈺勛

A. Introduction

這次的作業是要實作 CVAE，並且 RNN 是使用 LSTM，所以和以往 CVAE 的示意圖不同，LSTM 會有兩個 hidden state 要學習。藉由這個 CVAE 來學英文單字的時態，並且做 sampling 餵給 decoder 來產生目標單字，便可以運用 BLEU-4 score 和 Gaussian score 來分析 CVAE 的學習狀況。下圖是程式中提供的 arguments，可以藉此控制各個 hyperparameter，並且可以藉由'-l'來 load 之前儲存的 model 和紀錄，用'-s'可以單純顯示之前的紀錄。

```
parser = ArgumentParser(description='VAE & CVAE')
parser.add_argument('-hs', '--hidden_size', default=256, type=check_hidden_size_type, help='RNN hidden size')
parser.add_argument('-ls', '--latent_size', default=32, type=int, help='Latent size')
parser.add_argument('-c', '--condition_embedding_size', default=8, type=int, help='Condition embedding size')
parser.add_argument('-k', '--kl_weight', default=0.0, type=check_float_type, help='KL weight')
parser.add_argument('-kt', '--kl_weight_type', default='monotonic', type=check_kl_type,
                    help='Fixed, monotonic or cyclical KL weight')
parser.add_argument('-t', '--teacher_forcing_ratio', default=0.5, type=check_float_type,
                    help='Teacher forcing ratio')
parser.add_argument('-tt', '--teacher_forcing_type', default='decreasing', type=check_teacher_type,
                    help='Fixed or decreasing teacher forcing ratio')
parser.add_argument('-lr', '--learning_rate', default=0.007, type=float, help='Learning rate')
parser.add_argument('-e', '--epochs', default=100, type=int, help='Number of epochs')
parser.add_argument('-l', '--load', default=0, type=check_load_type,
                    help='Whether load the stored model and accuracies')
parser.add_argument('-s', '--show_only', default=0, type=check_show_type, help='Whether only show the results')
parser.add_argument('-v', '--verbosity', default=0, type=check_verbosity_type, help='Verbosity level')
```

B. Derivation of CVAE

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

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

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

C. Implementation details

1. Dataloader

```python
class CharDict:
    def __init__(self):
        self.word_to_index = {}
        self.index_to_word = {}
        self.num_of_words = 0

        for word in ['SOS', 'EOS']:
            self.add_word(word)

        for num in range(26):
            self.add_word(chr(ord('a') + num))

    def add_word(self, word: str) -> None:
        """
        Add a word to the dictionary
        :param word: word to be added
        :return: None
        """
```

```python
        if word not in self.word_to_index:
            self.word_to_index[word] = self.num_of_words
            self.index_to_word[self.num_of_words] = word
            self.num_of_words += 1

    def string_to_long_tensor(self, input_string: str) -> LongTensor:
        """
        Convert string to Long Tensor represented by indices
        :param input_string: input string
        :return: Long Tensor represented by indices
        """
        sequence = ['SOS'] + list(input_string) + ['EOS']
        return LongTensor([self.word_to_index[char] for char in sequence])

    def long_tensor_to_string(self, long_tensor: LongTensor) -> str:
        """
        Convert Long Tensor to original string
        :param long_tensor: input Long Tensor
        :return: original string
        """
```

```python
        original_string = ''
        for obj in long_tensor:
            char = self.index_to_word[obj.item()]
            if len(char) < 2:
                original_string += char
            elif char == 'EOS':
                break
        return original_string
```

利用上面的 CharDict class 來將 word 轉換成數字，並且進行 word 和

LongTensor 之間的轉換，轉換會將 word 的每個 character 抽出並且在頭尾加上 SOS 和 EOS，最後將他們變成一個 list 來組成 LongTensor。

```
class TenseLoader(Dataset):
    def __init__(self, mode: str):
        if mode == 'train':
            file = './data/train.txt'
        else:
            file = './data/test.txt'

        self.data = np.loadtxt(file, dtype=np.str)

        if mode == 'train':
            self.data = self.data.reshape(-1)
        else:
            # sp, tp, pg, p
            # sp → p
            # sp → pg
            # sp → tp
            # sp → tp
            # p  → tp
            # sp → pg
            # p  → sp
            # pg → sp
            # pg → p
            # pg → tp
            self.targets = np.array([
                [0, 3],
                [0, 2],
                [0, 1],
                [0, 1],
                [3, 1],
                [0, 2],
                [3, 0],
                [2, 0],
                [2, 3],
                [2, 1],
            ])

        self.char_dict = CharDict()
        self.mode = mode
        self.tenses = [
            'simple-present',
            'third-person',
            'present-progressive',
            'simple-past'
        ]

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

    def __getitem__(self, index: int):
        if self.mode == 'train':
            condition = index % 4
            return self.char_dict.string_to_long_tensor(self.data[index]), condition
        else:
            input_long_tensor = self.char_dict.string_to_long_tensor(self.data[index, 0])
            input_condition = self.targets[index, 0]
            output_long_tensor = self.char_dict.string_to_long_tensor(self.data[index, 1])
            output_condition = self.targets[index, 1]
            return input_long_tensor, input_condition, output_long_tensor, output_condition
```

用上圖的 TenseLoader class 可以讀取 training data 或是 testing data，並

且在取用每一個資料的時候都會利用上面的 CharDict class 來將 word 轉換成 LongTensor 來做後面的訓練。

2. Encoder

```
class EncoderRNN(nn.Module):
    def __init__(self, input_size: int, condition_size: int, hidden_size: int, latent_size: int,
                 condition_embedding_size: int, train_device: device):
        super(EncoderRNN, self).__init__()

        self.input_size = input_size
        self.condition_size = condition_size
        self.hidden_size = hidden_size
        self.latent_size = latent_size
        self.condition_embedding_size = condition_embedding_size
        self.train_device = train_device

        # Embedding
        self.condition_embedding = nn.Embedding(num_embeddings=condition_size,
                                                embedding_dim=condition_embedding_size)
        self.input_embedding = nn.Embedding(num_embeddings=input_size,
                                            embedding_dim=hidden_size)

        # RNN
        self.lstm = nn.LSTM(input_size=hidden_size,
```

```
                            hidden_size=hidden_size)

        # Linear layers for hidden state
        self.hidden_mean = nn.Linear(in_features=hidden_size,
                                     out_features=latent_size)
        self.hidden_log_var = nn.Linear(in_features=hidden_size,
                                        out_features=latent_size)

        # Linear layers for cell state
        self.cell_mean = nn.Linear(in_features=hidden_size,
                                   out_features=latent_size)
        self.cell_log_var = nn.Linear(in_features=hidden_size,
                                      out_features=latent_size)

    def forward(self, inputs: LongTensor, prev_hidden: Tensor, prev_cell: Tensor,
                input_condition: int) -> Tuple[Tuple[Tensor, ...], Tuple[Tensor, ...]]:
        """
        Forward propagation
        :param inputs: inputs
        :param prev_hidden: previous hidden state
```

```
        :param prev_cell: previous cell state
        :param input_condition: input conditions
        :return: (hidden mean, hidden log variance, hidden latent), (cell mean, cell log variance, cell latent)
        """
        # Embed condition
        embedded_condition = self.embed_condition(input_condition)

        # Concatenate previous hidden state with embedded condition to get current hidden state
        hidden_state = cat((prev_hidden, embedded_condition), dim=2)

        # Concatenate previous cell state with embedded condition to get current cell state
        cell_state = cat((prev_cell, embedded_condition), dim=2)

        # Embed inputs
        embedded_inputs = self.input_embedding(inputs).view(-1, 1, self.hidden_size)

        # Get RNN outputs
        _, next_states = self.lstm(embedded_inputs, (hidden_state, cell_state))
        next_hidden, next_cell = next_states
```

```
        # Get hidden mean, log variance, and sampled values
        hidden_mean = self.hidden_mean(next_hidden)
        hidden_log_var = self.hidden_log_var(next_hidden)
        hidden_latent = randn(self.latent_size).to(self.train_device) * exp(0.5 * hidden_log_var).to(
            self.train_device) + hidden_mean

        # Get cell mean and log variance, and sampled values
        cell_mean = self.cell_mean(next_cell)
        cell_log_var = self.cell_log_var(next_cell)
        cell_latent = randn(self.latent_size).to(self.train_device) * exp(0.5 * cell_log_var).to(
            self.train_device) + cell_mean

        return (hidden_mean, hidden_log_var, hidden_latent), (cell_mean, cell_log_var, cell_latent)

    def init_hidden_or_cell(self) -> Tensor:
        """
        Return initial hidden or cell state
        :return: Tensor with zeros
        """
        return torch.zeros(1, 1, self.hidden_size - self.condition_embedding_size, device=self.train_device)
```

```
    def embed_condition(self, condition: int) -> Tensor:
        """
        Embed condition
        :param condition: original condition
        :return: embedded condition
        """
        condition_tensor = LongTensor([condition]).to(self.train_device)
        return self.condition_embedding(condition_tensor).view(1, 1, -1)
```

上圖是 Encoder class，他的 RNN 是 LSTM，input 和 condition 我都是用 embedding 來將他們轉到高維，而不是採用 one-hot 來將 condition 轉成 one-hot vector。LSTM 輸出的 hidden state 和 cell state 都會經過各自的 linear layer 來產生各自的 mean 和 log variance，而這些 mean 和 logvariance 會再用來經過 sampling 取得 decoder 要使用的 hidden latent 以及 cell latent 來輸出。

3. Decoder

```
class DecoderRNN(nn.Module):
    def __init__(self, input_size: int, condition_size: int, hidden_size: int, latent_size: int,
                 condition_embedding_size: int, train_device: device):
        super(DecoderRNN, self).__init__()

        self.input_size = input_size
        self.condition_size = condition_size
        self.hidden_size = hidden_size
        self.latent_size = latent_size
        self.condition_embedding_size = condition_embedding_size
        self.train_device = train_device

        # Embedding
        self.condition_embedding = nn.Embedding(num_embeddings=condition_size,
                                                embedding_dim=condition_embedding_size)
        self.input_embedding = nn.Embedding(num_embeddings=input_size,
                                            embedding_dim=hidden_size)

        # Latent to hidden/cell
```

```
        self.hidden_latent_to_hidden_state = nn.Linear(in_features=latent_size + condition_embedding_size,
                                                       out_features=hidden_size)
        self.cell_latent_to_cell_state = nn.Linear(in_features=latent_size + condition_embedding_size,
                                                   out_features=hidden_size)

        # RNN
        self.lstm = nn.LSTM(input_size=hidden_size,
                            hidden_size=hidden_size)

        self.out = nn.Linear(hidden_size, input_size)

    def forward(self, inputs: LongTensor, hidden_latent: Tensor, cell_latent: Tensor) -> Tuple[Tensor, ...]:
        """
        Forward propagation
        :param inputs: inputs
        :param hidden_latent: hidden latent
        :param cell_latent: cell latent
        :return: output, next hidden latent, next cell hidden latent
        """
        # Embed inputs
```

```
        embedded_inputs = self.input_embedding(inputs).view(1, 1, self.hidden_size)

        # Get RNN outputs
        output, next_latents = self.lstm(embedded_inputs, (hidden_latent, cell_latent))
        next_hidden_latent, next_cell_latent = next_latents

        # Get decoded output
        output = self.out(output).view(-1, self.input_size)

        return output, next_hidden_latent, next_cell_latent

    def init_hidden_and_cell(self, hidden_latent: Tensor, cell_latent: Tensor,
                             input_condition: int) -> Tuple[Tensor, ...]:
        """
        Concatenate latent and condition, then convert latent to state and return
        :param hidden_latent: hidden latent
        :param cell_latent: cell latent
        :param input_condition: input condition
        :return: hidden state and cell state
        """
```

```
        # Embed condition
        embedded_condition = self.embed_condition(input_condition)

        concatenated_hidden_latent = cat((hidden_latent, embedded_condition), dim=2)
        concatenated_cell_latent = cat((cell_latent, embedded_condition), dim=2)
        return self.hidden_latent_to_hidden_state(concatenated_hidden_latent), self.cell_latent_to_cell_state(
            concatenated_cell_latent)

    def embed_condition(self, condition: int) -> Tensor:
        """
        Embed condition
        :param condition: original condition
        :return: embedded condition
        """
        condition_tensor = LongTensor([condition]).to(self.train_device)
        return self.condition_embedding(condition_tensor).view(1, 1, -1)
```

上圖是 Decoder class，RNN 一樣是 LSTM，同樣的會將 input 和 condition 用 embedding 轉到高維，並且會將 encoder 給的 hidden latent 和 cell latent 經過 linear layer 轉成要餵給 LSTM 的 hidden state 和 cell state，最後會將 LSTM 的輸出經過一個 linear layer 來轉成 decode 出來的目標 word 機率 Tensor。

4. KL loss

```python
def kl_loss(hidden_mean: Tensor, hidden_log_variance: Tensor, cell_mean: Tensor, cell_log_variance: Tensor) -> Ten
    """
    Compute KL divergence loss
    :param hidden_mean: mean of hidden state
    :param hidden_log_variance: log variance of hidden state
    :param cell_mean: mean of cell state
    :param cell_log_variance: log variance of cell state
    :return: loss
    """
    return torch.sum(0.5 * (hidden_mean ** 2 + torch.exp(hidden_log_variance) - hidden_log_variance - 1
                            + cell_mean ** 2 + torch.exp(cell_log_variance) - cell_log_variance - 1))
```

上圖是計算 KL loss 的 function，因為 LSTM 有 hidden 和 cell，所以會希望這兩者在 condition 底下的機率分布會是 normal gaussian，因此就是將兩者和 normal gaussian 各自的 KL loss 和在一起。

5. Compute Gaussian score

```python
def compute_gaussian(decoder: DecoderRNN,
                     input_size: int,
                     latent_size: int,
                     test_dataset: TenseLoader,
                     train_device: device) -> float:
    """
    Compute Gaussian score for testing
    :param decoder: decoder
    :param input_size: input size (word)
    :param latent_size: latent size
    :param test_dataset: testing dataset
    :param train_device: training device
    :return: Gaussian score
    """
    words = []
    for _ in range(100):
        hidden_noises, cell_noises = randn(latent_size).to(train_device), randn(latent_size).to(train_device)
        group = []
        for condition in range(len(test_dataset.tenses)):
            group.append(generate_word(decoder=decoder,
                                       input_size=input_size,
                                       hidden_noises=hidden_noises,
                                       cell_noises=cell_noises,
                                       condition=condition,
                                       target_length=28,
                                       test_dataset=test_dataset,
                                       train_device=train_device))
        words.append(group)
    return gaussian_score(words)
```

在計算 Gaussian score 的時候不像 training 以及 testing BLEU-4 score 的時候一樣都會先將 data 餵給 encoder 拿到 latent 再餵給 decoder 得到 output，他反而是 sample noise 直接餵給 decoder，因此我在計算 Gaussian score 時是以上圖的 function 計算，他會從 normal gaussian 取樣拿到 hidden noise 和 cell noise，並且給想要的 condition 來產生 words。

6. Hyperparameters

Hyperparameters 可以利用在 introduction 貼的 arguments 來設定。

a. Teacher forcing ratio

```python
def get_current_teacher_forcing_ratio(epoch: int) -> float:
    """
    Get current teacher forcing ratio based on current epoch
    :param epoch: current epoch
    :return: teacher forcing ratio
    """
    if epoch < 150:
        return 1.0

    ratio = 1.0 - 0.005 * (epoch - 150)
    if ratio <= 0.0:
        return ratio
    return ratio
```

當 argument '--teacher_forcing_type'是 decreasing 時會使用上面的 function 來得到當下的 teacher forcing ratio，如果'--teacher_forcing_type'是 fixed 就會採用 argument 設定的 teacher forcing ratio。

b. KLD loss weight annealing

```python
def monotonic_kl_annealing(epoch: int) -> float:
    """
    Get monotonic KL cost annealing based on current epoch
    :param epoch: current epoch
    :return: KL weight
    """
    if epoch < 50:
        return 0.0

    weight = 0.0016 * (epoch - 50)
    if weight >= 1.0:
        return 1.0
    return weight
```

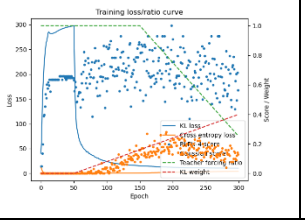
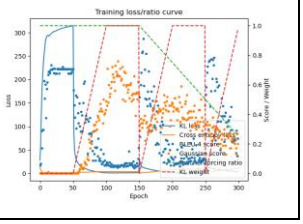
```python
def cyclical_kl_annealing(epoch: int) → float:
    """
    Get cyclical KL cost annealing based on current epoch
    :param epoch: current epoch
    :return: KL weight
    """
    if epoch < 50:
        return 0.0

    weight = 0.02 * ((epoch - 50) % 100)
    if weight ⩾ 1.0:
        return 1.0
    return weight
```

如果 argument '--kl_weight_type'是 monotonic 或 cyclical 就會使用上面的 function 來得到當下的 KLD weight，如果 argument '--kl_weight_type'是 fixed 就會採用 argument 設定的 KLD weight。

D. Results and discussion

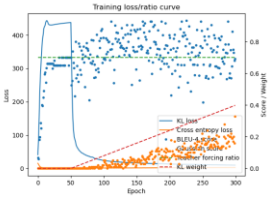
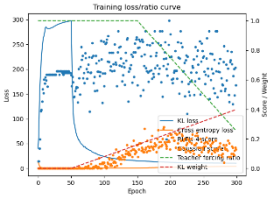
1. KLD weight

	Monotonic	Cyclical
Learning rate 0.007		

由上面的表格可以看出當 KLD weight 開始增加不是零的時候，BLEU-4 score 會下降，而 Gaussian score 會上升，這是因為 KLD loss 是在計算 hidden latent 和 normal Gaussian 之間的差距，所以當 KLD weight 增加的時候會使 CVAE 的 hidden latent 往 normal Gaussian 靠近，使得直接以 normal Gaussian 做 sample 後計算的 Gaussian score 會直接受影響而上升，但 BLEU-4 score 因為 encoder 還沒辦法 output 出很接近 normal gaussian 的 hidden latent，因此會下降。

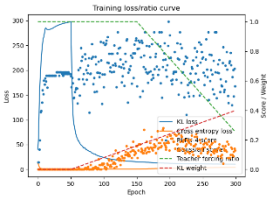
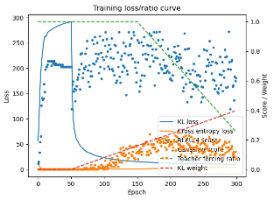
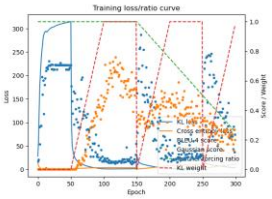
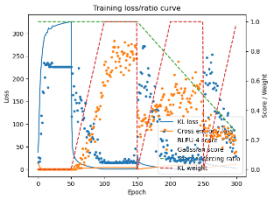
2. Teacher forcing ratio

	Fixed	Decreasing

Monotonic KLD weight		

由上面的表格可以看出如果以固定的 teacher forcing ratio 可以使 BLEU-4 score 較為固定在比較高的分數，而且 Gaussian score 也會緩慢上升，但如果是 decreasing 就會發現當 teacher forcing ratio 開始下降的時候，BLEU-4 score 和 Gaussian score 都會開始跟著下降，這是因為 teacher forcing 可以讓 decoder 在 decode 的時候拿到正確的 input 而不是自己上一個 state output 出來的錯誤 input，因此當 teacher forcing ratio 下降的時候，decoder 開始有較大的機率拿到錯誤 input，就會使得 decoder 內的 weight 開始往錯誤的方向學習。

3. Learning rate

	Learning rate 0.007	Learning rate 0.005
Monotonic KLD weight		
Cyclical KLD weight		

從上面的表格可以稍微看出當 learning rate 減少的時候 BLEU-4 score 分散的較集中，尤其在 monotonic 比較容易看出，這應該是因為 learning rate 較大的時候會造成震盪，但這邊因為 learning rate 的變化較小所以不太明顯。