DL HW4

1. Introduction

In this assignment, we should correct the typo word by seq2seq encoder-decoder network with recurrent units . Seq2seq network can deal with the input and output which are different length. We use LSTM to preserve long term memory. Teacher forcing technique leads the model to correct direction and make it converge faster.

2. Derivation of BPTT

3. Experimental details

A. Describe how you implement your model

- Dataloader

I split the word to independent alphabets and convert the alphabet to corresponding numbers. Because SOS_token = 0 and EOS_token = 1 by default, I set the other number from 2. If word dropout is used, I set <UNK> term '28'. Also, I add EOS_token to the tail of every training word. For example, 'apple' will be '2' '17' '17' '13' '6' '1'.

```python
def str_to_int(x_str):
    return np.array([ord(x)-ord('a')+2 for x in x_str])

def int_to_str(x_int):
    return "".join(np.array([chr(x + ord('a')-2) for x in x_int]))


def dataloader(mode):
    if mode == 'train':
        # data = open('./train.json', 'r')
        with open('./train.json') as f:
            data_input = json.load(f)
        data = []
        for i in data_input:
            word_tar = torch.from_numpy(np.append(str_to_int(i['target']), EOS_token)).view(-1, 1)
            for j in i['input']:
                word_input = torch.from_numpy(str_to_int(j)).view(-1, 1)
                data.append((word_input, word_tar))

        return data
    else:
        with open('./test.json') as f:
            data_input = json.load(f)
        data = []
        origin = []
        for i in data_input:
            word_tar = torch.from_numpy(str_to_int(i['target'])).view(-1, 1)
            word_input = torch.from_numpy(str_to_int(i['input'][0])).view(-1, 1)
            origin.append((i['input'][0], i['target']))
            data.append((word_input, word_tar))

        return data, origin
```

- Encoder

I modify two parts of EncoderRNN in sample code. GRU is replaced with LSTM. LSTM needs additional cell state to remember long term memory.

```python
#Encoder
class EncoderRNN(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(EncoderRNN, self).__init__()
        self.hidden_size = hidden_size

        self.embedding = nn.Embedding(input_size, hidden_size)
        self.lstm = nn.LSTM(hidden_size, hidden_size)

    def forward(self, input, hidden, c_state):
        embedded = self.embedding(input).view(1, 1, -1)
        output = embedded

        output, (hidden, c_state) = self.lstm(output, (hidden, c_state))
        return output, hidden, c_state

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

I take one alphabet as the input to the encoder sequentially. Previous hidden layer and c_state from encoder will be the next states to the encoder. Finally, the last output is the latent of the whole input word.

```python
def train(input_tensor, target_tensor, encoder, decoder, encoder_optimizer, decoder_optimizer, criterion, max_length=MAX_LENGTH):
    encoder_hidden = encoder.initHidden()
    encoder_c_state = encoder.initHidden()

    encoder_optimizer.zero_grad()
    decoder_optimizer.zero_grad()

    input_length = input_tensor.size(0)
    target_length = target_tensor.size(0)

    loss = 0

    #----------sequence to sequence part for encoder----------#
    for ei in range(input_length):
        encoder_output, encoder_hidden, encoder_c_state = encoder(input_tensor[ei].cuda(GPUID[0]), encoder_hidden.cuda(GPUID[0]), \
                                                                  encoder_c_state.cuda(GPUID[0]))
```

- Decoder

Modification is similar to the encoder.

```python
#Decoder
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.lstm = nn.LSTM(hidden_size, hidden_size)
        self.out = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden, c_state):
        output = self.embedding(input).view(1, 1, -1)
        output = F.relu(output)

        output, (hidden, c_state) = self.lstm(output, (hidden, c_state))
        output = self.out(output[0])
        return output, hidden, c_state


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

Latent from the encoder is the initial hidden state of decoder. Last hidden layer of encoder is the initial cell state of the decoder. If using teacher forcing, the input to the decoder will be ground truth. If using dropout, the input to the decoder will be '28' (<UNK> term).

```python
decoder_input = torch.tensor([[SOS_token]])

decoder_hidden = encoder_output
decoder_c_state = encoder_hidden

use_teacher_forcing = True if random.random() < teacher_forcing_ratio else False


#----------sequence to sequence part for decoder----------#
if use_teacher_forcing:
    # Teacher forcing: Feed the target as the next input
    for di in range(target_length):
        use_dropout = True if random.random() < dropout_ratio else False
        decoder_output, decoder_hidden, decoder_c_state = decoder(
            decoder_input.cuda(GPUID[0]), decoder_hidden.cuda(GPUID[0]), decoder_c_state.cuda(GPUID[0]))
        loss += criterion(decoder_output, target_tensor[di])
        decoder_input = target_tensor[di]  # Teacher forcing
        # print(decoder_input)
        if use_dropout:
            decoder_input = torch.tensor(28).cuda(GPUID[0])
```

However, if not using teacher forcing, the input to the decoder will be the previous output of decoder.

```
else:
    # Without teacher forcing: use its own predictions as the next input
    for di in range(target_length):
        use_dropout = True if random.random() < dropout_ratio else False
        decoder_output, decoder_hidden, decoder_c_state = decoder(
            decoder_input.cuda(GPUID[0]), decoder_hidden.cuda(GPUID[0]), decoder_c_state.cuda(GPUID[0]))
        topv, topi = decoder_output.topk(1)
        decoder_input = topi.squeeze().detach()  # detach from history as input

        loss += criterion(decoder_output, target_tensor[di])
        # print(decoder_input)
        if decoder_input.item() == EOS_token:
            break
        if use_dropout:
            decoder_input = torch.tensor(28).cuda(GPUID[0])
```

B. Code of evaluation part screenshot

In the evaluation part, the length of output word is unknown. I set the length to max_length by default. If output of decoder is EOS_token, it means the tail of the word and the decode process ends. The inputs to the decoder are all the previous output of the decoder except the first one (SOS_token). I don't use any target tensor in this part.

```
def test(input_tensor, encoder, decoder, max_length=MAX_LENGTH):
    encoder_hidden = encoder.initHidden()
    encoder_c_state = encoder.initHidden()

    input_length = input_tensor.size(0)

    loss = 0

    #----------sequence to sequence part for encoder----------#
    for ei in range(input_length):
        encoder_output, encoder_hidden, encoder_c_state = encoder(input_tensor[ei].cuda(GPUID[0]), encoder_hidden.cuda(GPUID[0]),\
                                                                  encoder_c_state.cuda(GPUID[0]))

    decoder_input = torch.tensor([[SOS_token]])

    decoder_hidden = encoder_output
    decoder_c_state = encoder_hidden


    #----------sequence to sequence part for decoder----------#

    # Without teacher forcing: use its own predictions as the next input
    pred = []
    for di in range(max_length):

        decoder_output, decoder_hidden, decoder_c_state = decoder(
            decoder_input.cuda(GPUID[0]), decoder_hidden.cuda(GPUID[0]), decoder_c_state.cuda(GPUID[0]))
        topv, topi = decoder_output.topk(1)
        decoder_input = topi.squeeze().detach()  # detach from history as input


        if decoder_input.item() == EOS_token:
            break

        pred.append(decoder_input.cpu().numpy())

    return int_to_str(pred)
```

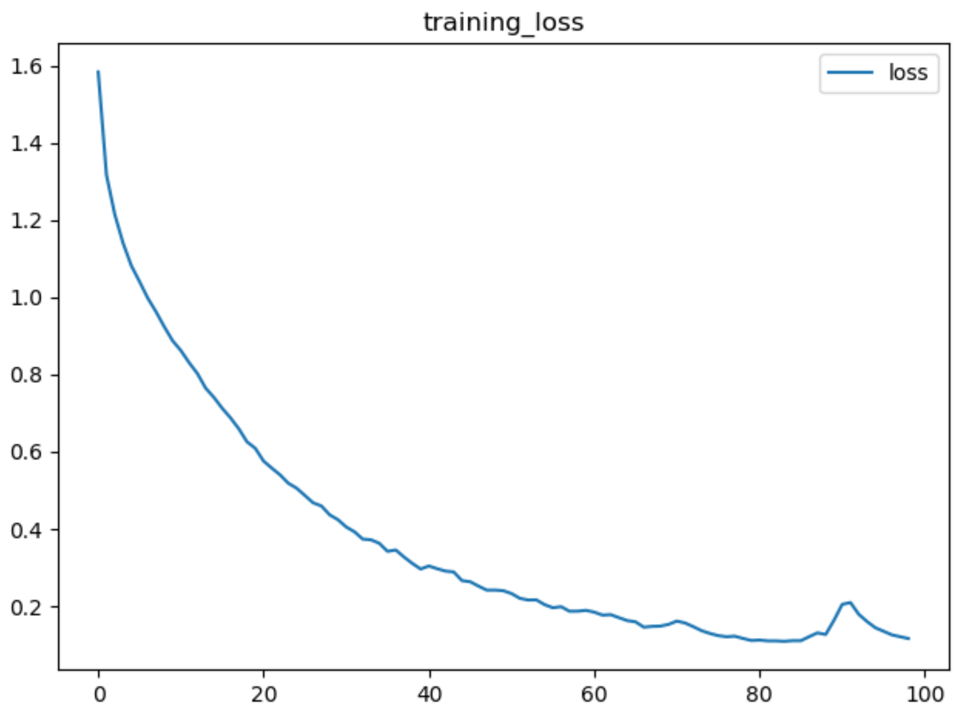
4. Results and discussion

A. Result

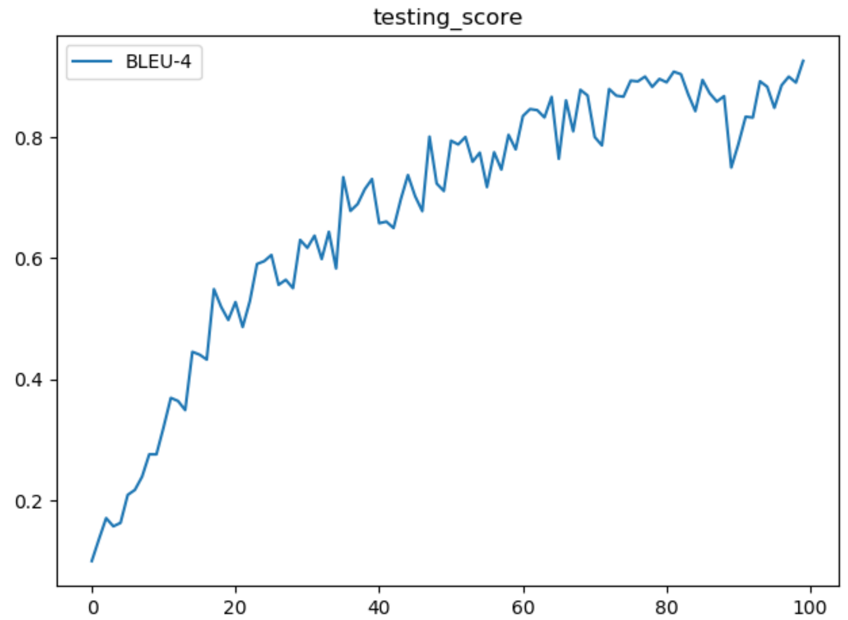
- Spelling correction

```
========================================
input:journel
target:journal
pred:journal
========================================
input:leason
target:lesson
pred:lesson
========================================
input:mantain
target:maintain
pred:maintain
========================================
input:miricle
target:miracle
pred:miracle
========================================
input:oportunity
target:opportunity
pred:opportunity
========================================
input:parenthasis
target:parenthesis
pred:parenthesis
========================================
input:recetion
target:recession
pred:recession
========================================
input:scadual
target:schedule
pred:schedule
BLEU-4 score:0.925754
```

- Loss curve



- BLUE-4 sore testing curve



B. Discussion

The result above is the best model among those I've tried. I set learning rate 0.01 with decay 10e-4 and teacher forcing 0.5. If setting teacher forcing to 1, training loss is much lower. But when testing, the score is also lower. The reason is that we don't have the ground truth when testing. When I set the value to 0.5, the score gets higher.

I also use dropout in the model. Dropout rate is 0.2. It would set some alphabet to <UNK> in decode time with the probability. Dropout makes the model more robust and improve the performance. The following figure shows the testing score of the model with and without dropout.