CSC413 PA3

Yinjun Zheng

Part 1

Question 1.

The code is shown as follows:

```
class MyLSTMCell(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(MyLSTMCell, self).__init__()

        self.input_size = input_size
        self.hidden_size = hidden_size

        # ------------
        # FILL THIS IN
        # ------------
        self.Wii = nn.Linear(input_size, hidden_size)
        self.Whi = nn.Linear(hidden_size, hidden_size)

        self.Wif = nn.Linear(input_size, hidden_size)
        self.Whf = nn.Linear(hidden_size, hidden_size)

        self.Wig = nn.Linear(input_size, hidden_size)
        self.Whg = nn.Linear(hidden_size, hidden_size)

        self.Wio = nn.Linear(input_size, hidden_size)
        self.Who = nn.Linear(hidden_size, hidden_size)
```

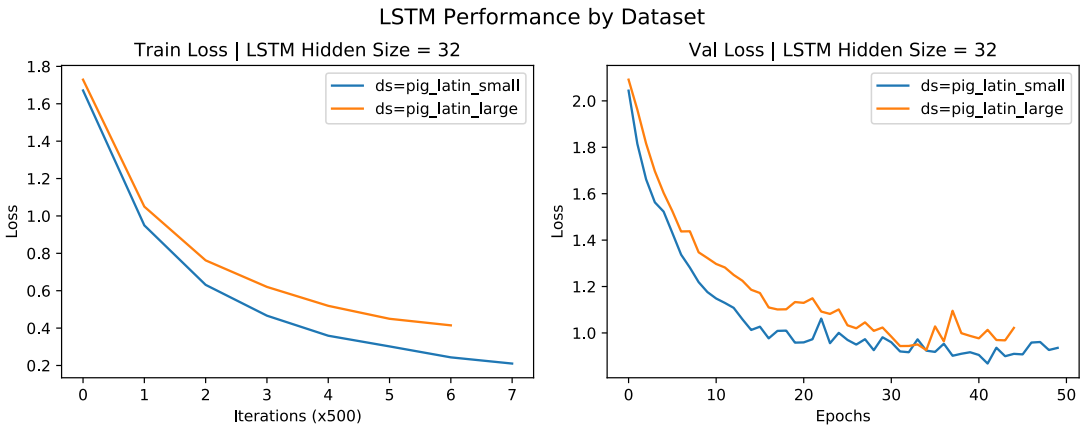
```
  def forward(self, x, h_prev, c_prev):
      """Forward pass of the LSTM computation for one time step.

      Arguments
          x: batch_size x input_size
          h_prev: batch_size x hidden_size
          c_prev: batch_size x hidden_size

      Returns:
          h_new: batch_size x hidden_size
          c_new: batch_size x hidden_size
      """

      # ------------
      # FILL THIS IN
      # ------------
      i = torch.sigmoid(self.Wii(x) + self.Whi(h_prev))
      f = torch.sigmoid(self.Wif(x) + self.Whf(h_prev))
      g = torch.tanh(self.Wig(x) + self.Who(h_prev))
      o = torch.sigmoid(self.Wio(x) + self.Who(h_prev))
      c_new = f * c_prev + i * g
      h_new = o * torch.tanh(c_new)
      return h_new, c_new
```

The loss plot is shown as follows:



Analysis:

The model performs better on the Pig_latin_small dataset. This is probably because the larger dataset contains more words, so there might be more extreme cases which make the loss larger than the smaller set.

Question 2.

The result is not good when I try longer words such as "program". It also fails when I try words that end with "ing". The reason might be that the model does have a good generalization on long words.

For example, the following example "I am trying test the program", it fails at the word "trying" and "program".

```
best_encoder = rnn_encode_l # Replace with rnn_losses_s or rnn_losses l
best_decoder = rnn_decoder_l # etc.
best_args = rnn_args_l

TEST_SENTENCE = 'i am trying test the program'
translated = translate_sentence(TEST_SENTENCE, best_encoder, best_decoder, None, best_args)
print("source:\t\t{} \ntranslated:\t{}".format(TEST_SENTENCE, translated))
```

```
source:         i am trying test the program
translated:     iway amway yingray esttay ethay opragedcay
```

Question 3.

(1) Number of unit: K

Because if we regard one LSTM as a unit, and for each word there is a LSTM unit, so there are K units in total.

(2) Number of connections: $4K * (DH + H^2)$

Because for each LSTM, there are four gates which have the same number of connections. Four each gate, connections between hidden states are H^2 and connections between input and hidden is DH. Also since there are K words, so the total connections are $4K * (DH + H^2)$.

Part 2 Additive Attention

Question 1.

$$\hat{a}_i^{(t)} = f(Q_t, K_i) = W_2\big(ReLU\big(W_1([Q_t|\, K_i] + \, b_1)\big) + b_2\big)$$

$$a_i^{(t)} = softmax\left(\hat{a}_i^{(t)}\right)$$

$$c_t = \sum_{i=1}^{len(seq)} a_i^{(t)}\, K_i$$

Question 2.

The attention model is performing better than decoder without attention. The validation loss is 0.23, and the translation result is right. However, for decoder without attention, the loss is 0.9 and the translation for "conditioning" is not right.

Question 3.

The decoder with attention is training slower. It is probably because it needs to consider the weight from all the previous words, which takes longer time.

Question 4.

(1) Number of units: K

Because for each word there is one unit, same reason as in part 1 question 3.

(2) Number of connections: $K * (DHK + VH + 3H^2K + HK)$

For the rnn part, we have $(D + H)H$ connections。 Since there are K iterations, so the connection is $(D + H)HK$

Additionally, to get the attention for each word, we have connection (2H * H + H) * K, which is because there are two linear layer and one ReLU layer in the attention. The first has 2H * H, and the second has H. Since there are K iterations, so the total connection is (2H * H + H) * K. The final output layer has connection V*H.

Also since the sequence length is K, so the total connection is:

$K * ((D + H)HK + VH + 2H^2K + HK) \; = K * (DHK + VH + 3H^2K + HK)$.

Part 3 Scaled Dot Production Attention

Question 1.

The code is shown as follows:

```
def forward(self, queries, keys, values):
    """The forward pass of the scaled dot attention mechanism.

    Arguments:
        queries: The current decoder hidden state, 2D or 3D tensor. (batch_size x (k) x hidden_size)
        keys: The encoder hidden states for each step of the input sequence. (batch_size x seq_len x hidden_size)
        values: The encoder hidden states for each step of the input sequence. (batch_size x seq_len x hidden_size)

    Returns:
        context: weighted average of the values (batch_size x k x hidden_size)
        attention_weights: Normalized attention weights for each encoder hidden state. (batch_size x seq_len x 1)

        The output must be a softmax weighting over the seq_len annotations.
    """

    # ------------
    # FILL THIS IN
    # ------------
    batch_size = queries.shape[0]
    q = self.Q(queries)
    k = self.K(keys)
    v = self.V(values)
    # print(q.transpose(1,2).shape)
    unnormalized_attention = torch.bmm(k, q.transpose(1,2)) / self.scaling_factor
    attention_weights = self.softmax(unnormalized_attention)
    context = torch.bmm(attention_weights.transpose(1,2), v)
    return context, attention_weights
```

Question 2.

The code is shown as follows:

```
def forward(self, queries, keys, values):
    """The forward pass of the scaled dot attention mechanism.

    Arguments:
        queries: The current decoder hidden state, 2D or 3D tensor. (batch_size x (k) x hidden_size)
        keys: The encoder hidden states for each step of the input sequence. (batch_size x seq_len x hidden_size)
        values: The encoder hidden states for each step of the input sequence. (batch_size x seq_len x hidden_size)

    Returns:
        context: weighted average of the values (batch_size x k x hidden_size)
        attention_weights: Normalized attention weights for each encoder hidden state. (batch_size x seq_len x 1)

        The output must be a softmax weighting over the seq_len annotations.
    """

    # ------------
    # FILL THIS IN
    # ------------
    batch_size = queries.shape[0]
    q = self.Q(queries)
    k = self.K(keys)
    v = self.V(values)
    unnormalized_attention = torch.bmm(k, q.transpose(1,2)) / self.scaling_factor
    mask = torch.tril(unnormalized_attention)
    mask[mask == 0] = self.neg_inf
    attention_weights = self.softmax(mask)
    context = torch.bmm(attention_weights.transpose(1,2), v)
    return context, attention_weights
```

Question 3.

The reason we use positional encoding is that: in attention-based model as Transformer, there is no built-in notion of the sequence of tokens. Positional encoding is a way for us to encode the order of the sequence into our model.

The advantage of using positional encoding method is that the model can learn the sequence order by learning the position, so that it can generalize better for longer sentences. Also, other method such as one hot requires higher dimension, but encoding by sin and cos

function costs less.

Question 4.

The model performance of scaled dot production model with hidden size 32 on smaller dataset is worse than the models in previous sections.

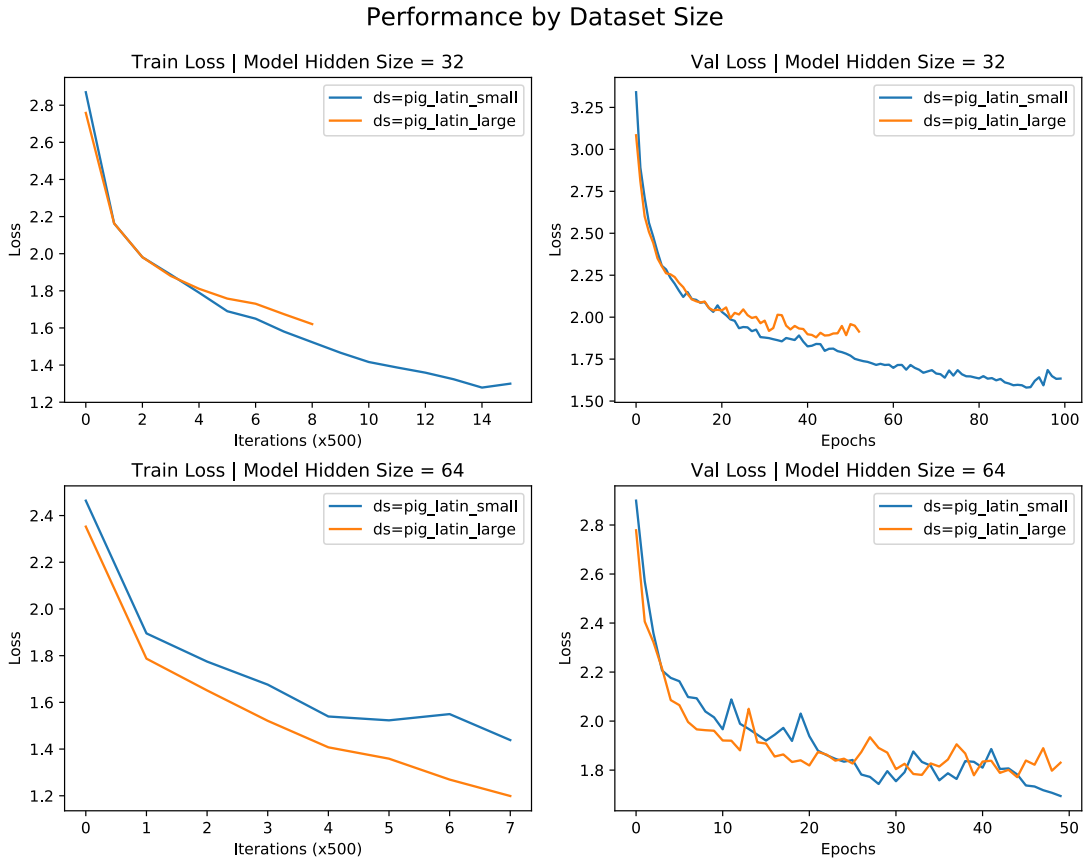
As we can see, from the result, the lowest validation loss is 1.5803557430825583. However, the validation loss in part 2 with additive attention is only 0.2814960081084836. The validation loss in part 1 is only 0.8680315683255109.

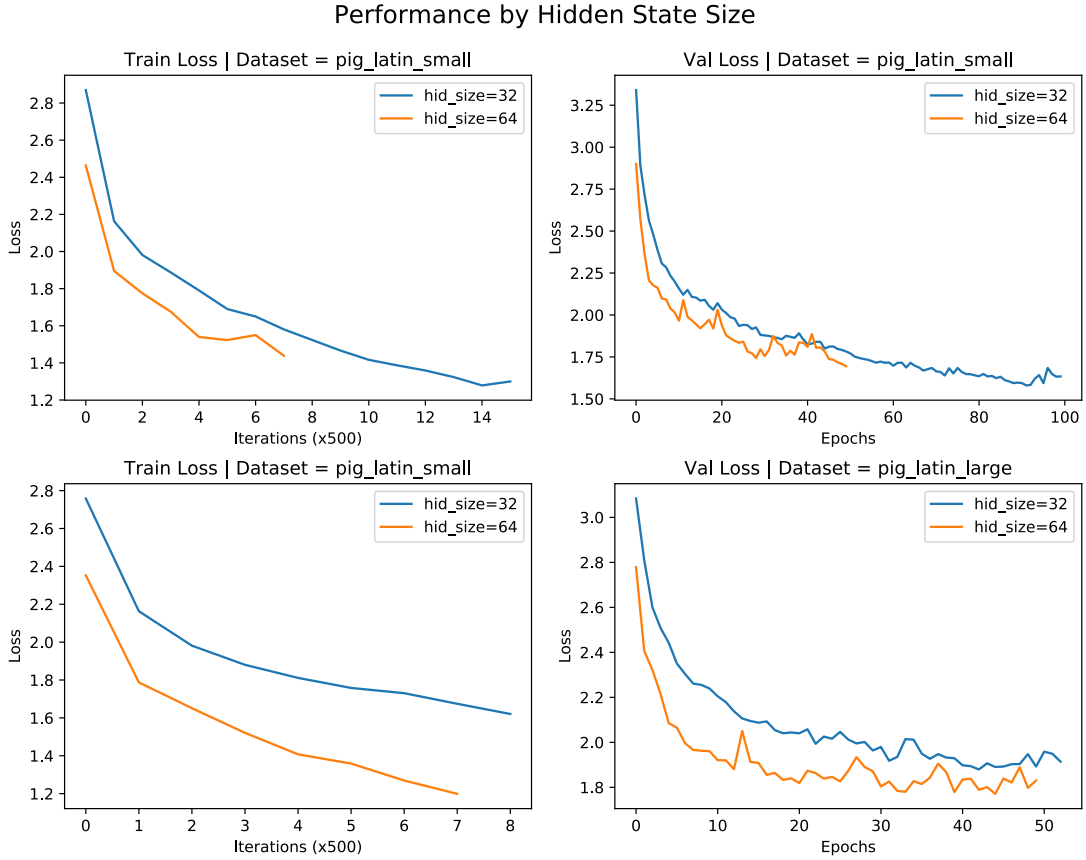
Also, we can see the translation result directly. In this model, the translation result is "ay aray oonaooayaayy isssay onghinginggyyEOSooy", which is obviously not a good translation. However, with additive attention in part 3, the result is "ethay airway onditioningcay isway orkingway", which is the right translation we want.

Therefore, the model performance is worse than the previous models.

Question 5.

The result is shown as follows:





The lowest validation loss is shown as follows:
The lowest validation loss for small dataset with hidden size 32 is 1.5803557430825583.
The lowest validation loss for large dataset with hidden size 32 is 1.8795731365680695.
The lowest validation loss for small dataset with hidden size 64 is 1.6943262263042171.
The lowest validation loss for large dataset with hidden size 32 is 1.7711772580559437.

Analysis:
The training loss decreases and converge in fewer iterations, while it takes more iterations for the validation loss to converge. Both training loss and validation loss are lower when we use smaller dataset. And the loss is also lower when we user a larger hidden size 64 than a hidden size of 32.
It is exactly what we expect, because a larger hidden size improves the model capacity, thus the performance is expected to be better. Also, based on the observation of in part 1, the loss of the smaller dataset tends to be smaller.

Part 4 Fine-tuning for arithmetic sentiment analysis

Question 1.

The code is shown as follows:

```
[ ]  from transformers import OpenAIGPTForSequenceClassification
     class GPTCSC413(OpenAIGPTForSequenceClassification):
         def __init__(self, config):
             super(GPTCSC413, self).__init__(config)
             # Your own classifier goes here
             self.classifier = nn.Sequential(
                 nn.Linear(config.hidden_size, config.hidden_size),
                 nn.ReLU(),
                 nn.Linear(config.hidden_size, self.config.num_labels)
                 )
```

Question 2.

The model freeze_bert has accuracy 0.74 and remains unchanged. The model finetune_bert has accuracy 0.96. The finetune bert has a better performance.

The result is shown as follows:

The model is performing poorly when the two input number are the same.

Question 3

The model finetune_gpt has accuracy 0.95.

The result is shown as follows:

An interesting fact is that two models give different answers to the questions including two minus.

For example, for the question "three minus two minus two", finetune_gpt does not give the right answer, but finetune_bert givers the right answer.

Question 4.

GPT is preferred when finding NLP solutions with limited data. The reason for that is GPT uses a few-shot learning process on the input token to predict the output result, while BERT needs a fine-tuning process when training the model. Also, since BERT is trained on latent relationship challenges between the text of different contexts, GPT is preferred when insufficient data is provided.