CSC413 Programming Assignment 3 March 20, 2021

Part 1: Long Short-Term Memory

1. The code for this question is shown below.

```
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.Whg(h_prev))
        o = torch.sigmoid(self.Wio(x) + self.Who(h_prev))
        c_new = torch.mul(f, c_prev) + torch.mul(i, g)
        h_new = torch.mul(o, torch.tanh(c_new))
        return h_new, c_new

```

According to Figure 1, there is no large difference between models using the small dataset and large dataset. The LSTM model using small dataset performs better during the training, while the validation loss of the large dataset model is lower, which is reasonable. Since more training examples could reduce the impact of overfitting, it might be the reason for the higher training loss and lower validation loss for the smaller dataset model.

2. I tested five sentences and results are shown below.

```
source:    his son is an eight-year-old student
translated: ishay onsay isway anway ewtigay-ouncehsay udentday
correct:   ishay onsay isway anway eightway-earyay-oldway udentstay

```

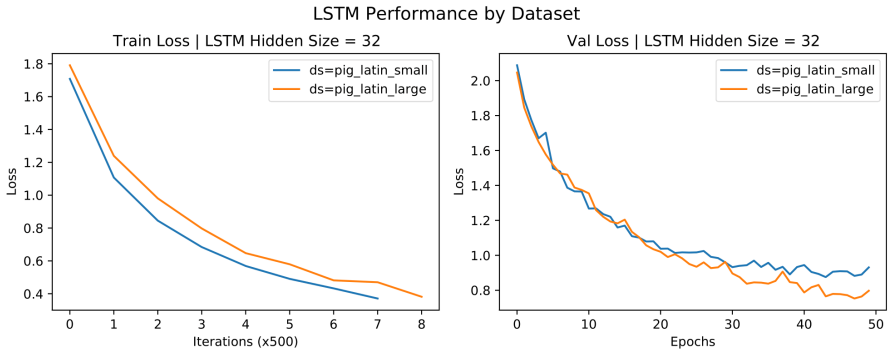


Figure 1: Train and validation loss comparison by dataset.

```
source:   the model with smaller dataset performs significantly better
translated: ethay odedmay ithway alerlsmay atesstay erstoursway intigslaingsay
    ettetyway
correct:  ethay odelmay ithway allersmay atasetday erformsay ignificantlysay
    etterbay

source:   people are facing a severe situation of resources shortage and
    environmental pollution and degradation of ecosystems
translated: eoplephay areway acingfay away eerkenay ituationsday ofway esurechtray
     orpaleday andway indenverelngway olluptionway andway egdgatinesway ofway
    eccospestlay
correct:  eoplepay areway acingfay away everesay ituationsay ofay esourcesray
    ortageshay andway environmentalway ollutionpay andway egradationday ofway
    ecosystemsway

source:   turning off the lights when leaving is not common
translated: untringway offay ethay ylustlay engay ealundway isway otnay ommouncay
correct:  urningtay offway ethay ightslay enwhay eavinglay isway otnay ommoncay

source:   there are many other ways to achieve a low-carbon lifestyle
translated: eterhay areway anmay otherway ackay otay achemhway away owlungay-ag-
    odhay ifulestleway
correct:  erethay areway anymay otherway aysway otay achieveay away owlay-arboncay
     ifestylelay
```

One significant failure is that the model performs badly when dealing with the word including the dash symbol. The dash symbol is either missing or too many, which may count two words as one, resulting in the incorrect combination of transformed word. For example, "eight-year-old" becomes "ewtigay-ouncehsay". "low-carbon" is translated to "owlungay-ag-odhay".

3. For each character, we trained the input, output, forget, and tanh gate separately. Since the output includes both next hidden units and next cell units, there are $8H$ training units in total. For a $K$ length word, the number of LSTM units should still be $8H$.
   For the number of connections, since the dimension of each gate is $H$, the number of connections is $4(H+D)\times H$ for training four gates. Then, to generate the new cell state, it requires the input gate, the forget gate, the tanh gate and the previous cell state. Thus, for this process, the number of connections should be $(2H+2H)\times H$. Similarly, for processing the new hidden state, we just take the product of output gate and new cell state, which gives the number of connections should be $2H$. Thus, the total number of connections is

$$\begin{aligned}&(4H^2+4DH+4H*H+2H)K\\=&(8H^2+4DH+2H)K\\=&8H^2K+4DHK+2HK\end{aligned}$$

Part 2: Additive Attention

1. Based on the given information, we can compute $\tilde{\alpha}_i^{(t)}$, $\alpha_i$, $c_t$ as follow.

$$\begin{aligned}
\tilde{\alpha}_i^{(t)} &= f(Q_t, K_i) \\
&= W_2 \text{ReLU}(W_1 \text{concat}(Q_t, K_i) + b_1) + b_2 \\
&= W_2 \max(0, W_1[Q_t, K_i] + b_1) + b_2 \\
\alpha_i^{(t)} &= \text{softmax}(\tilde{\alpha}_i^{(t)}) \\
&= \frac{\exp(\tilde{\alpha}_i^{(t)})}{\sum_t \exp(\tilde{\alpha}_i^{(t)})} \\
c_t &= \sum_i \alpha_i^{(t)} K_i
\end{aligned}$$

2. (Optional)
3. (Optional)
4. For each character, we trained the hidden state and attention weight separately. For the learning parameters from the concatenation, since it concatenates each query and corresponding keys by sequences, there are $2H$ training units for one sequence. After activation, the model would train them again to get the attention weight, which also requires $2H$ training units for one sequence. In addition, for RNN step, since we used the model in Part one, there are $8H$ LSTM units for training. Thus, for a $K$ length word, the number of LSTM units should be $12H$.
   For the number of connections, let's calculate it step by step. Since it calculates each attention by sequences, there are $K$ steps. For each step, it firstly get the attention based on the query and the key. After the concatenation, it trained the attention by two fully connected layer with one activation function. It firstly translates from $2H$ to $H$, and then $H$ to 1. After we get the normalized attention, the model also get the context by multiplication of the attention weight and values. Thus, the number of connections should be $2H^2K^2 + HK^2 + HK^2$. Then, it goes to RNN layer to generation the prediction based on the hidden unit and the cell unit. It requires $8H^2 + 4DH + 2H$ connections from part one. Thus, the total number of connections is

$$\begin{aligned}
&(2H^2K^2 + HK^2 + HK^2 + 8H^2 + 4DH + 2H)K \\
=&(2H^2K^2 + 2HK^2 + 8H^2 + 4DH + 2H)K \\
=&2H^2K^3 + 2HK^3 + 8H^2K + 4DHK + 2HK
\end{aligned}$$

   At the end, it connects to the fully connected layer to output the probabilities for each vocabulary. Since the size of the output layer is $KV$ and the previous concatenation has a shape of $HK$, the number of connections should be $HK^2V$. Thus, the total number of connections is $2H^2K^3 + 2HK^3 + 8H^2K + 4DHK + 2HK + HK^2V$.

Part 3: Scaled Dot Product Attention

1. The code for this question is shown below.

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

        self.hidden_size = hidden_size

        self.Q = nn.Linear(hidden_size, hidden_size)
        self.K = nn.Linear(hidden_size, hidden_size)
        self.V = nn.Linear(hidden_size, hidden_size)
        self.softmax = nn.Softmax(dim=1)
        self.scaling_factor = torch.rsqrt(torch.tensor(self.hidden_size, dtype=
    torch.float))

    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.size()[0]
        # 2D to 3D
        if len(queries.size()) == 2:
          queries = queries.view(batch_size, 1, self.hidden_size)
        q = self.Q(queries)
        k = self.K(keys)
        v = self.V(values)
        unnormalized_attention = torch.bmm(k, torch.transpose(q, 1, 2)) * self.scaling_factor
        attention_weights = self.softmax(unnormalized_attention)
        context = torch.bmm(torch.transpose(attention_weights, 1, 2), v)
        return context, attention_weights

```

2. The code for this question is shown below.

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

        self.hidden_size = hidden_size
        self.neg_inf = torch.tensor(-1e7)

        self.Q = nn.Linear(hidden_size, hidden_size)
        self.K = nn.Linear(hidden_size, hidden_size)
        self.V = nn.Linear(hidden_size, hidden_size)
        self.softmax = nn.Softmax(dim=1)
        self.scaling_factor = torch.rsqrt(torch.tensor(self.hidden_size, dtype=torch.float))

    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.size()[0]
        # 2D to 3D
        if len(queries.size()) == 2:
            queries = queries.view(batch_size, 1, self.hidden_size)
        q = self.Q(queries)
        k = self.K(keys)
        v = self.V(values)
        unnormalized_attention = torch.bmm(k, torch.transpose(q, 1, 2)) * self.scaling_factor
        mask = torch.tril(torch.ones_like(unnormalized_attention) * self.neg_inf, diagonal=-1)
        # print(mask)
        attention_weights = self.softmax(torch.triu(unnormalized_attention) + mask)
        context = torch.bmm(torch.transpose(attention_weights, 1, 2), v)
        return context, attention_weights

```

3. The positional encoding is required to process the information about the order and the position of each sequence in a context since the previous steps do not give any positional information to the model while the word is context-related. It is better since it not only gives the information about the absolute position in a context, but also tells the relative position within sequences.

4. The validation loss of this model is 0.7783684799447655, which is the highest loss compared with the previous two, where the additive attention model gets 0.2094228501940961 validation loss and the best validation loss of LSTMs model is 0.7528372138308791. The transformers failed to translate "air" correctly, and the LSTMs model performed badly when facing the word "conditioning", while additive attention model translate the text sentence correctly.

5. The plots and the loss results are shown below.

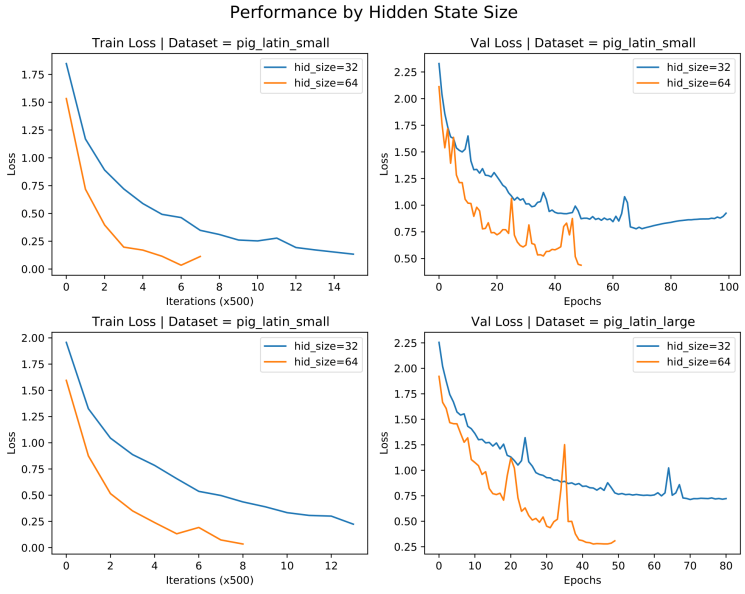


Figure 2: Train and validation loss comparison by hidden size.

According to Figure 2, we can see that the hidden state size affect the model performance a lot, where larger hidden state size would get lower train and validation loss. According to Figure 3,

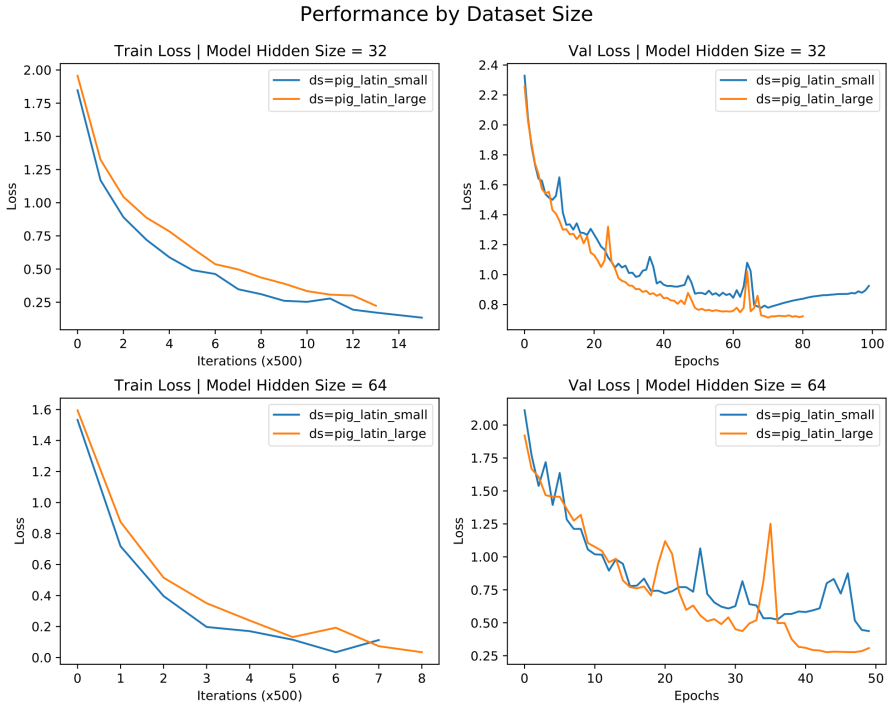


Figure 3: Train and validation loss comparison by dataset size.

| Hidden size \ Dataset size | small | large |
|---|---|---|
| 32 | 0.7783684799447655 | 0.7129870925206118 |
| 64 | 0.4370648428797722 | 0.276622651008298 |

Table 1: The best validation loss of four models with different hidden size and dataset size.

we found that the model using larger dataset would perform relatively better, but not significant. As we increased the number of iterations, the training loss would generally decrease, while the validation loss follows a decreasing trend with some oscillation. These results follow what I expected, since more hidden state size gain more trainable parameters and larger dataset provide more examples.

6. (Optional)
7. (Optional)

Part 4: Fine-tuning for arithmetic sentiment analysis

1. Similar to the BERT model, I simply add a fully connected layer as the classifier where it takes previous predictions and outputs the probabilities of each label. The code for this question is shown below.

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

```

2. (Optional)

3. (Optional)
4. GPT architecture is more perferred than BERT in the language generation task such as left-to-right language generation. GPT model is the autoregressive language model, which generate predictions based on the previous several inputs, while BERT model is a self-encoding language model, which does not consider the correlation between the prediction. Since BERT model has a different pre-training process and the generation process, it leads a poorer performance on the language generation task comparing with the GPT model.
5. (Optional)