PA3

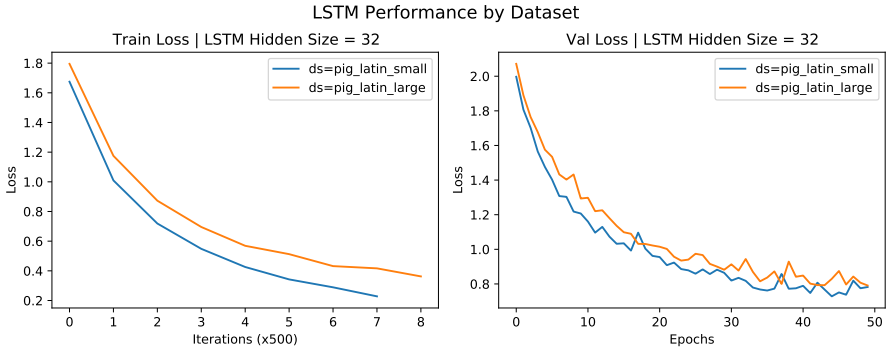
Polo Li

March 2021

1 Part I LSTM

1.1

```python
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
        # ------------
        #Linear: R^{dim_in} -> R^{dim_out}
        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
        # ------------
        sigmoid = nn.Sigmoid()
        tanh = nn.Tanh()
        i = sigmoid(self.Wii(x) + self.Whi(h_prev))
        f = sigmoid(self.Wif(x) + self.Whf(h_prev))
        g = tanh(self.Wig(x) + self.Whg(h_prev))
        o = sigmoid(self.Wio(x) + self.Who(h_prev))
        c_new = f * c_prev + i * g
        h_new = o * tanh(c_new)
        return h_new, c_new
```



Both larger and smaller models perform approximately well. Because piglatin is an easy language to translate and the smaller model is complex enough to learn the core set of rules in it.

1.2

"i want to get an a plus in this fascinating course" gets translated into "iiway antway otay etgay anway away usplay inway isthay actrimentay-ay orusecay". This is a failure because "fascinating" is translated into "actrimentay-ay", but should be "ascinatingfay" instead. The model doesn't work well on translating long words.

1.3

Number of LSTM Units: As defined in piazza, we need to count the number of hidden units in the whole LSTM which are computed using a unique set of weights. Since weights are shared between different LSTM cells, we only count the number of hidden cells in one LSTM cell, so number of LSTM units = H.
Number of connections: We ignore bias and element-wise connections. Thus number of connections is equal to number of weights, so number of connections = $4K(DH + H^2)$.

2 Part II Additive Attention

2.1

$$\hat{\alpha}_i^{(t)} = f(Q_t, K_i) = W_2(ReLU(W_1([Q_t; K_i]^T) + b_1)) + b_2 \tag{1}$$

$$\alpha_i^{(t)} = softmax(\hat{\alpha}_i^{(t)})_i \tag{2}$$

$$c_t = \sum_i \alpha_i^{(t)} K_i \tag{3}$$

2.2

The sentence "the air conditioning is working" gets translated into "ethay ariway ondintaciondway isway orkway-inceday" without attention, and "ethay airway onditionicgnay isway orkingway" with attention. As can be seen, "conditioning" and "working" has a much better result.

2.3

The training speed with and without attention are both around 200 seconds. This is possibly because decoder part still outputs tokens one by one. Parallelism has not been fully achieved yet.It can also be because the dataset is small so no significant difference can be observed.

2.4

Number of LSTM Units: As defined in piazza, we need to count the number of hidden units in the whole LSTM which are computed using a unique set of weights. Since weights are shared between different LSTM cells, we only count the number of hidden cells in one LSTM cell, so number of LSTM units = H.
Number of connections: We ignore bias, element-wise/activation connections and embedding process. We also assume batch size is 1.
In *AdditiveAttention*, for each time step, there are $(2H^2 + H)K$ connections in the calculation of $\hat{\alpha}^{(t)}$, KH connections in the calculation of c_t. So in total $K^2(2H^2 + 2H)$ connections.
For LSTM cells, we can apply the same calculation methodology as in Q1.3. The only difference is that cell state is now $(H + D)$-dimensional instead of D-dimensional. So in total $4K((D + H)H + H^2)$ connections.
The output layer consists of HVK connections.
All together, number of connections in decoder $= K^2(2H^2 + 2H) + 4K((D + H)H + H^2) + HVK$.

3 Part III Scaled Dot Product Attention

3.1

```python
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.
        """
        batch_size = queries.shape[0]
        q = self.Q(queries).view(batch_size, -1, self.hidden_size) #B x k x H
        k = self.K(keys) #B x S x H
        v = self.V(values) #B x S x H
        unnormalized_attention = k.bmm(q.transpose(1,2)) * self.scaling_factor # B x S x H & B x H x k -> B x S x k
        attention_weights = self.softmax(unnormalized_attention) #B x S x k
        context = attention_weights.transpose(1,2).bmm(v) #B x k x S & B x S x H -> B x k x H
        return context, attention_weights
```

3.2

```python
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 k)

            The output must be a softmax weighting over the seq_len annotations.
        """
        batch_size = queries.shape[0]
        q = self.Q(queries).view(batch_size, -1, self.hidden_size) #B x k x H
        k = self.K(keys) #B x S x H
        v = self.V(values) #B x S x H
        unnormalized_attention = k.bmm(q.transpose(1,2)) * self.scaling_factor
        unnormalized_attention = unnormalized_attention + self.neg_inf.expand_as(unnormalized_attention).tril(diagonal=-1).to(device ='cuda') #B x S x k
        attention_weights = self.softmax(unnormalized_attention) #B x S x k
        context = attention_weights.transpose(1,2).bmm(v) #B x k x S & B x S x H -> B x k x H
        return context, attention_weights
```

3.3

We want to have positional embeddings because the order of words in languages greatly affects the semantic meanings.The advantage of this sine and cosine encoding is that
1) it it kept within a a reasonable range, $[-1, 1]$. Encoding by adding a vector of increasing integers representing the position of the input will give very large position values for later words in a long sequence, diminishing the effect of word embeddings.
2) Making positional encoding dependent on i generates different frequencies for different dimensions, so that elements in $PE(p)$ and $PE(p + k)$ wouldn't have the same difference, i.e. each position vector is guaranteed to be very distinct.
3) One-hot encoding requires the hidden dimension to be larger than the sequence length. For sine and cosine functions, there's no such requirement on the hidden dimension.

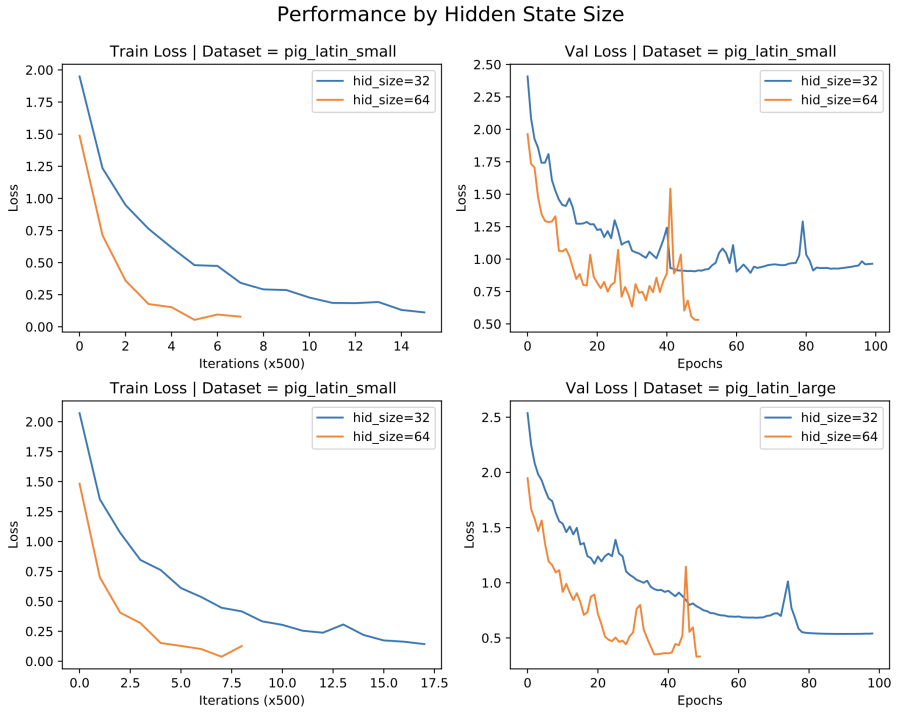
3.4

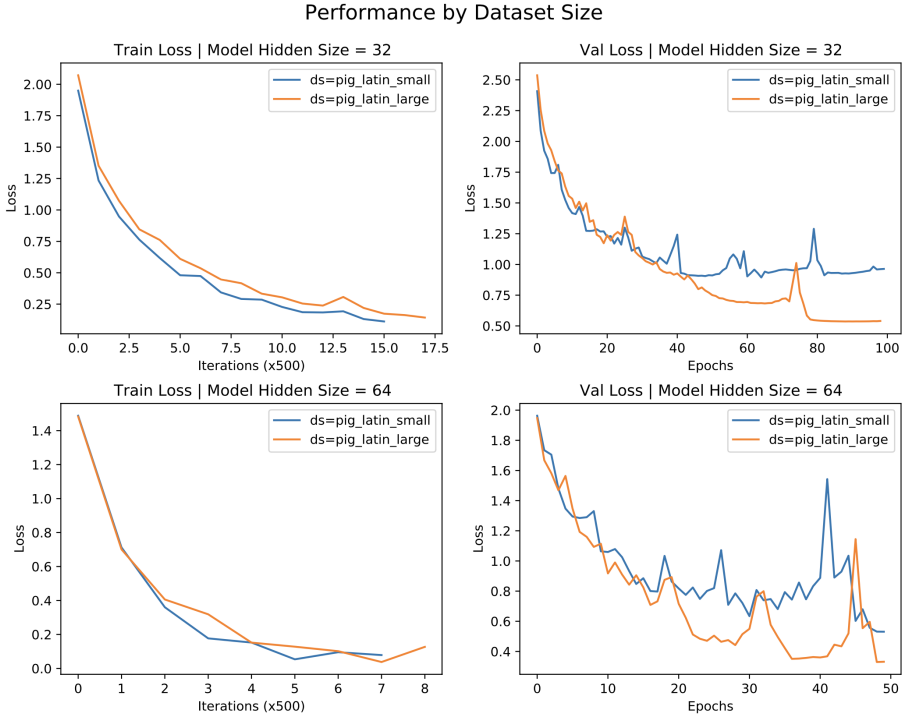
We compare the three models using the same configuration. As can be seen in table1(second row), Scaled Dot Product Attention performs roughly the same as RNN without attention(there is randomness in training), and performs worse than RNN with additive attention.

Table 1: Validation Loss across three models

Validation Loss	RNN Without Attention(Part 1)	Additive Attention(Part 2)	Scaled Dot Product Attention(Part 3)
Hidden Size: 32, Data size: Small	0.79	0.477	0.89
Hidden Size: 64, Data size: Small	0.55	0.154	0.54
Hidden Size: 32, Data size: Large	0.72	0.237	0.53
Hidden Size: 64, Data size: Large	0.442	0.137	0.33

3.5


Performance by Hidden State Size
Train Loss | Dataset = pig_latin_small
Val Loss | Dataset = pig_latin_small
Train Loss | Dataset = pig_latin_small
Val Loss | Dataset = pig_latin_large
hid_size=32
hid_size=64
Loss
Iterations (x500)
Epochs


Performance by Dataset Size
Train Loss | Model Hidden Size = 32
Val Loss | Model Hidden Size = 32
Train Loss | Model Hidden Size = 64
Val Loss | Model Hidden Size = 64
ds=pig_latin_small
ds=pig_latin_large
Loss
Iterations (x500)
Epochs

The lowest validation loss can be found in the last column in Table 1 in section 3.4.
Effect on Loss function tendency: Increasing hidden state size makes the lowest training/validation loss much smaller, and also makes the loss function converge to that point much quicker. Increasing dataset size makes immaterial impact on the training loss; loss function with small dataset converges after a certain number of iterations, but loss function for big dataset keeps on decreasing afterwards.
Effect on model generalization: Both increasing hidden size and increasing dataset size makes the validation loss lower, thus better generalization.

These phenomena are expected as a 64 dimensional model is more flexible and powerful and shouldn't overfit; larger dataset enables the model to learn a more general representation and perform better on unseen tasks.

3.6

After replacing ScaledDotAttention with CausalScaledDotAttention inside decoder, the output sentence is consists of either a long sequence of repeated characters or just a blank sentence, which doesn't make any sense. This is because the output sentence is trained to pay attention to future tokens, which it cannot do.

3.7

Advantage of Dot-Product Attention: While the two are similar in theoretical complexity, dot-product attention is much faster and more space-efficient in practice, since it can be implemented using highly optimized matrix multiplication code.
Advantage of Additive Attention: While for small values of d_k the two mechanisms perform similarly, additive attention outperforms dot product attention without scaling for larger values of d_k.

4 Part IV Fine-tuning for Arithmetic Sentiment Analysis

4.1

I emulated the design of $BertCSC413\_MLP$ by adding a fully-connected layer with $ReLU$ activation and a linear layer on top of GPT. The model works in training session and gives 95% accuracy.

4.2

```
plot_loss_and_acc(freeze_bert_loss_vals, freeze_bert_eval_accs)
```


```
plot_loss_and_acc(finetune_bert_loss_vals, finetune_bert_eval_accs)
```


```
eval_testdata(model_freeze_bert, bert_test_dataloader, bert_tokenizer, show_all_predictions=False)
```

```
Predicting labels for 160 test sentences...
Number of expressions with negative result 47
 0  predicted correctly , accuracy  0.0

Number of expressions with 0 result 2
 0  predicted correctly , accuracy  0.0

Number of expressions with positive result 111
 111  predicted correctly , accuracy  1.0
```

```
eval_testdata(model_finetune_bert, bert_test_dataloader, bert_tokenizer, show_all_predictions=False)
```

```
Predicting labels for 160 test sentences...
Number of expressions with negative result 47
 47  predicted correctly , accuracy  1.0

Number of expressions with 0 result 2
 0  predicted correctly , accuracy  0.0

Number of expressions with positive result 111
 109  predicted correctly , accuracy  0.9819819819819819
```

In training, the frozen model will only update weights in the classifier layer, whereas the fine-tuned model will update all weights.
Fine-tuned BERT model gives smaller loss on both training and test dataset(better generalization).

As can be seen, in frozen BERT, non of the negative and zero results are predicted correctly, but all positive results are predicted correctly. This is likely because the original BERT adjusts each input sequence such that the top classifier layer always treats such information as "positive".

4.3

Table 1: Interesting result across three models

	Fine-tuned GPT	Fine-tuned BERT	Frozen BERT
"negative three minus two"	-	+	+
"three minus two"	+	-	+
"three plus two"	+	+	+
"three minus two minus two"	-	-	+
"three minus two minus zero"	+	-	+
expression = "two minus two"	-	-	+
"three minus four"	-	-	+

4.4

GPT-3 is more preferred when there's limited amount of data.
"On the architecture dimension, while BERT is trained on latent relationship challenges between the text of different contexts, GPT-3 training approach is relatively simple compared to BERT. Therefore, GPT-3 can be a preferred choice at tasks where sufficient data isn't available, with a broader range of application."This statement is cited from https://analyticsindiamag.com/gpt-3-vs-bert-for-nlp-tasks/: