Programming Assignment 3

1 Part 1: Long Short-Term Memory (LSTM)

1.1

The implementations for init and forward method for MyLSTMCell are given in Figure 1 and Figure 2 respectively.

The loss plots produced are shown in Figure 3. Looking at loss plots produced, we can see that model trained on the smaller 'pig latin' dataset is performing better as it has a lower training and validation loss. The larger dataset size accounts for more words for the model to learn and can possibly lead to formation of longer sequences. This generation of potentially longer sequences (resulting from the larger dataset) can be a possible cause of weaker performance of the LSTM trained on larger dataset.

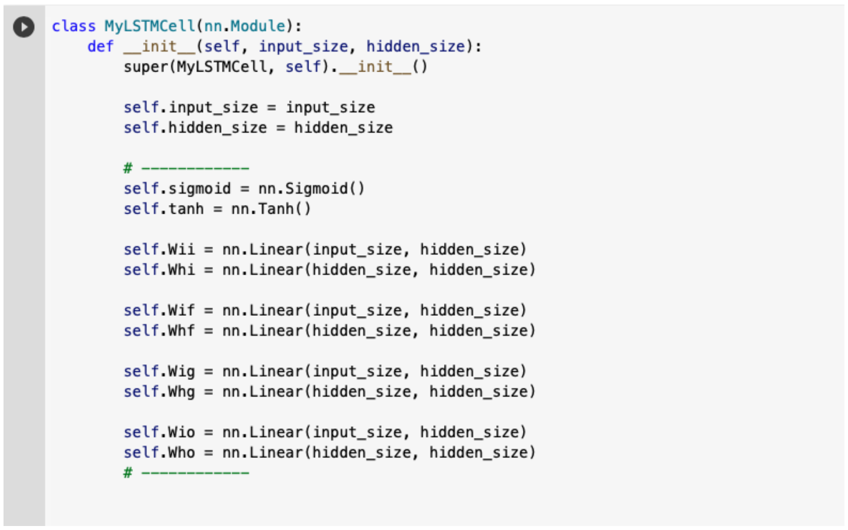
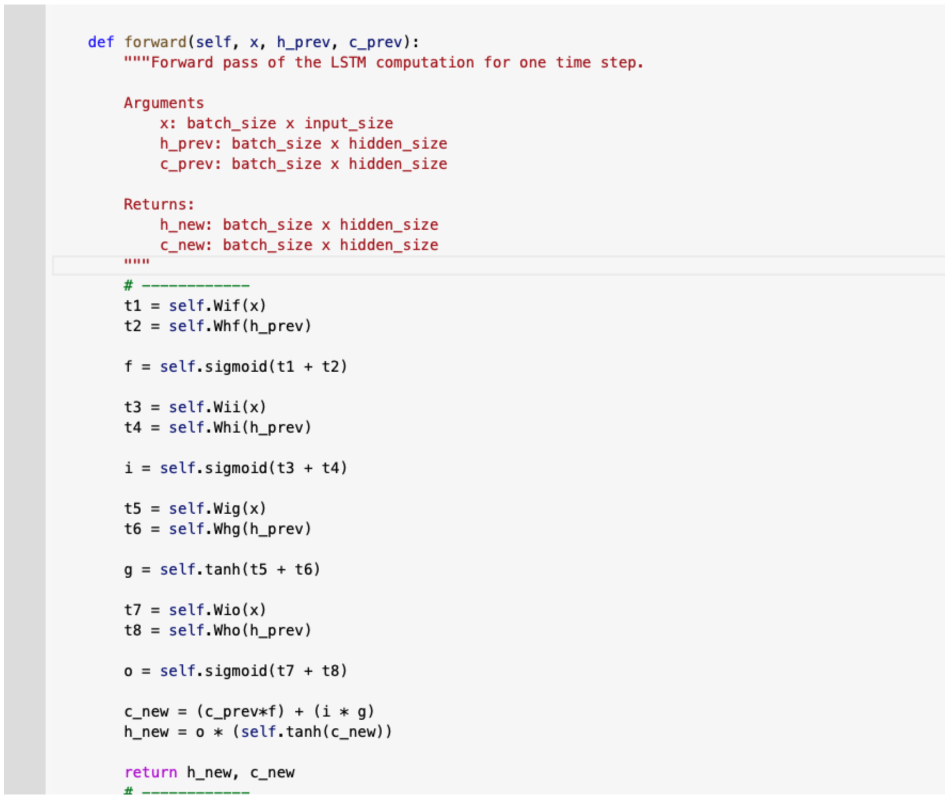

```
class MyLSTMCell(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(MyLSTMCell, self).__init__()

        self.input_size = input_size
        self.hidden_size = hidden_size

        # ------------
        self.sigmoid = nn.Sigmoid()
        self.tanh = nn.Tanh()

        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)
        # ------------
```

Figure 1:

1.2

The failure case identified:
Input: "greedy seedy and needy street"
Translation (Model output) : "eedigbay eedeway andway eedingway eetechay:
Expected translation (correct Pig Latin Translation) : "eedygray eedysay andway eedynay eetstray"

We can see our output produces bad results for the sentence given as input above. Sentences that contain a major stress on the vowel 'e' can be characterized as failure modes. When the model sees so many terms with sort of a 'repetitive' structure, it is not able to perform and generalize well. Words like "bees", "knees", "trees", etc. when put together in one sentence can make the model not perform well and can be characterized as a failure mode.



```
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
    """
    # ------------
    t1 = self.Wif(x)
    t2 = self.Whf(h_prev)

    f = self.sigmoid(t1 + t2)

    t3 = self.Wii(x)
    t4 = self.Whi(h_prev)

    i = self.sigmoid(t3 + t4)

    t5 = self.Wig(x)
    t6 = self.Whg(h_prev)

    g = self.tanh(t5 + t6)

    t7 = self.Wio(x)
    t8 = self.Who(h_prev)

    o = self.sigmoid(t7 + t8)

    c_new = (c_prev*f) + (i * g)
    h_new = o * (self.tanh(c_new))

    return h_new, c_new
    # ------------
```

Figure 2:

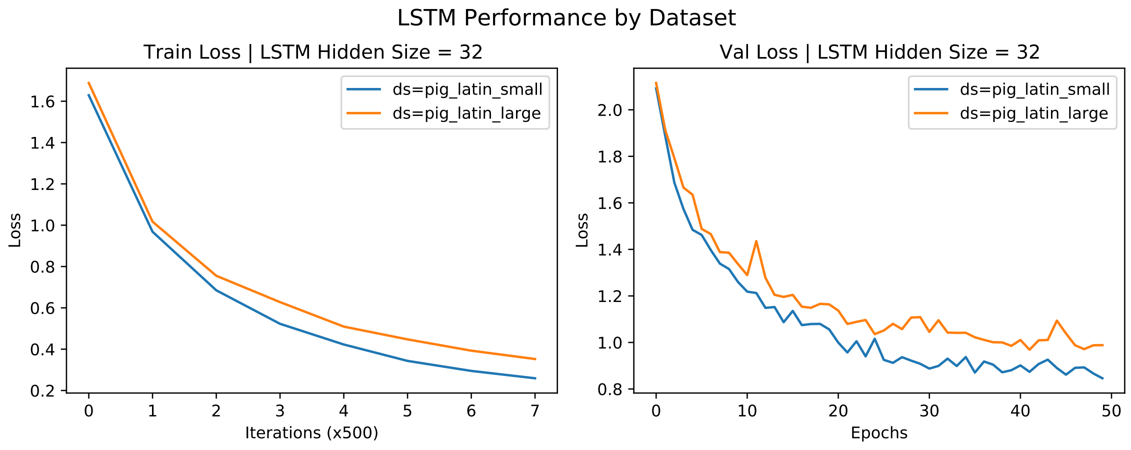


Figure 3:

1.3

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

2 Part 2: Additive Attention

2.1

$\tilde{\alpha}_i^{(t)} = f(Q_t, K_i) = W_1(ReLU(W_2([Q_t; K_i]) + b_2)) + b_1$
where $[Q_t; K_i]$ is just concatenation of Q_t and K_i

$\alpha_i^{(t)} = softmax(\tilde{\alpha}^{(t)})_i$
$c_t = \sum_{i=1}^{T} \alpha_i^{(t)} V_i$

2.2

2.3

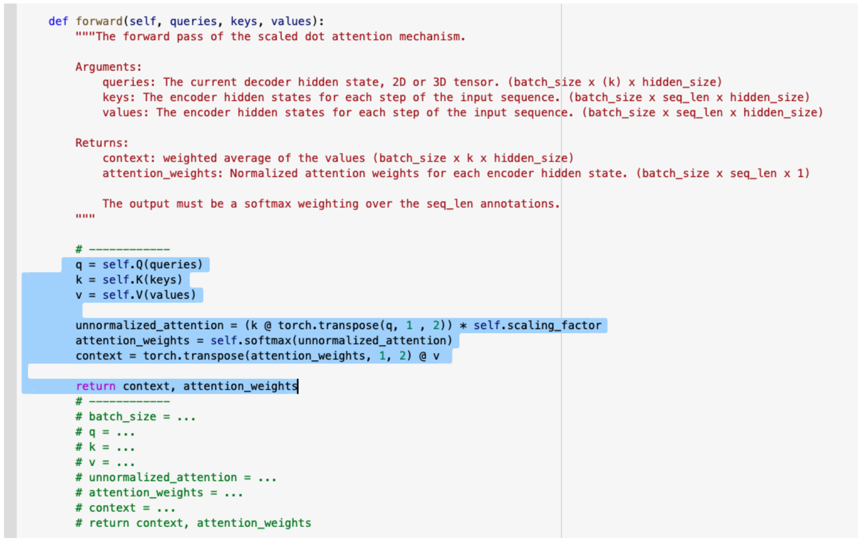
2.4

Number of units: H
Number of connections: $K(4(2H + H^2) + K(H^2 + H) + HV)$

3 Part 3: Scaled Dot Product Attention

3.1

Look at Figure 4 for Scaled dot product attention implementation.

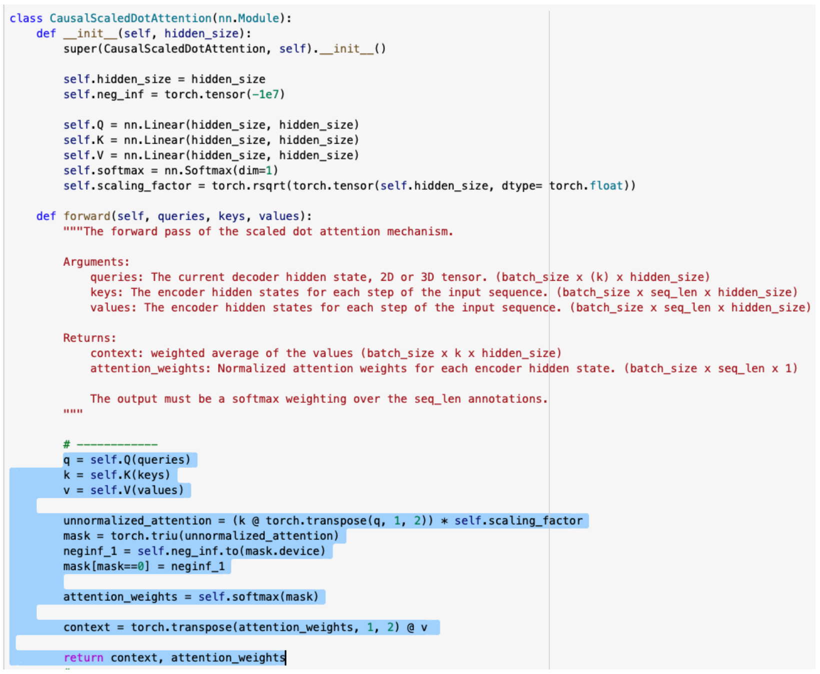


```
def forward(self, queries, keys, values):
    """The forward pass of the scaled dot attention mechanism.

    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.
    """

    # ------------
    q = self.Q(queries)
    k = self.K(keys)
    v = self.V(values)

    unnormalized_attention = (k @ torch.transpose(q, 1 , 2)) * self.scaling_factor
    attention_weights = self.softmax(unnormalized_attention)
    context = torch.transpose(attention_weights, 1, 2) @ v

    return context, attention_weights
    # ------------
    # batch_size = ...
    # q = ...
    # k = ...
    # v = ...
    # unnormalized_attention = ...
    # attention_weights = ...
    # context = ...
    # return context, attention_weights
```

Figure 4:

3.2

Look at Figure 5 for Causal Scaled Dot product attention implementation.



```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 1)

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

        # ------------
        q = self.Q(queries)
        k = self.K(keys)
        v = self.V(values)

        unnormalized_attention = (k @ torch.transpose(q, 1, 2)) * self.scaling_factor
        mask = torch.triu(unnormalized_attention)
        neginf_1 = self.neg_inf.to(mask.device)
        mask[mask==0] = neginf_1

        attention_weights = self.softmax(mask)

        context = torch.transpose(attention_weights, 1, 2) @ v

        return context, attention_weights
```

Figure 5:

3.3

We need to represent the position of each word through positional encoding because it is an efficient way to indicate position/order of each word. We need to include this positional encoding as we are doing sequence to sequence prediction and therefore the position of tokens in said sequence are of importance to us and we would like to keep a track of that.

The advantage of using positional encoding method compared to other methods is based on the idea that we are not doing any 'learning' for positional information (unlike RNN). We include positional encoding directly in the vector representation of words which allows us to deal with long sequences and also doesn't make us suffer from problems such as vanishing gradients (which can arise when we rely on temporal information heavily in our learning process).

3.4

We can see that, the language model, when trained with hidden size = 32 and a small dataset, doesn't perform that well compared to the decoders before. The validation loss achieved in the model with additive attention was 0.233 which is much better compared to validation loss of 0.876 which is what we got for this model. Looking at the way the learning progresses and the desired losses, we can see that the decoder before this model, i.e. the decoder associated to the Additive attention weights performs much better and therefore, we can understand that the small dataset and the hidden size of 32 doesn't get us the best performance

3.5

For batch size = 64, Hidden size = 32, Lowest validation loss = 0.8760
For batch size = 512, Hidden size = 32, Lowest validation loss = 0.7720
For batch size = 64, Hidden size = 64, Lowest validation loss = 0.4947
For batch size = 512, Hidden size = 64, Lowest validation loss = 0.4134

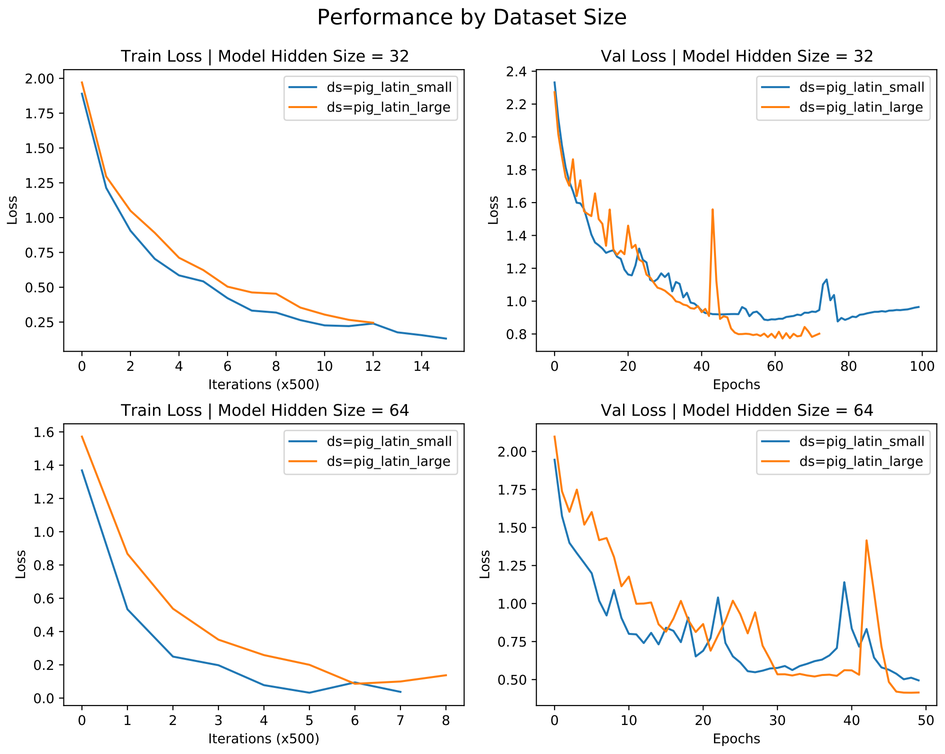


Figure 6:

Looking at the results in the plots shown in Figure 7, Figure 6, we can infer that regardless model with hidden size as 64 always performs better than the model with 32 hidden dimensions. This is evident by the validation loss values given above and even by the plots, we can see that the number of iterations for loss minimization is always less than the number of iterations taken for model with hidden size of 32 regardless of dataset size. Also, the number of epochs to train is also lower in model with hidden size 64 as compared to model with hidden size 32. Now as for the effect of dataset size of generalization/performance, we can see, by looking at the plots, that when hidden size is kept constant, then the model trained with larger data set would be better performing and indeed, would be more generalizable.

Therefore we can see that training the model with hidden size = 64 and batch size = 512 would give us a strong model such that the gradient descent algorithm will in a few iterations for the loss and the model will also be generalisable.

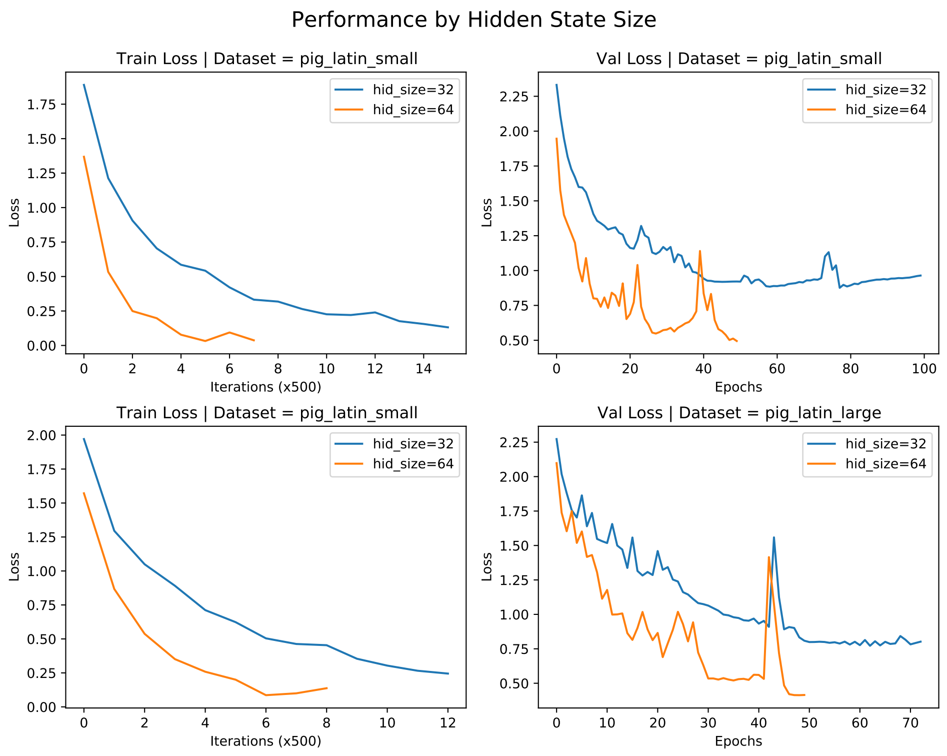


Figure 7:

4 Part 4: Fine-tuning for arithmetic sentiment analysis

4.1

I built the classifier by using an MLP which outputs a distribution over the possible class labels; I used the documentation of OpenAIGPTForSequenceClassification to parametrize my layer so that its in accordance with the OpenAIGPTForSequenceClassification class. The actual code/implementation can be found in the ipynb file.

4.2

4.3

4.4

A possible scenario in which GPT might be preferred to BERT would be where a sequence is standarly processed in left to right direction. If we want to predict the next unknown and previously unseen word in a sequence, then GPT might be a better choice as it solely relies on the precontext (context to the left of target word) to make predictions. For example, the word suggestion mechanism in our phones where our phone prompts us different words when we are writing a text, etc., can be probably implemented via GPT model and may give stronger performance than BERT applied to the same problem.