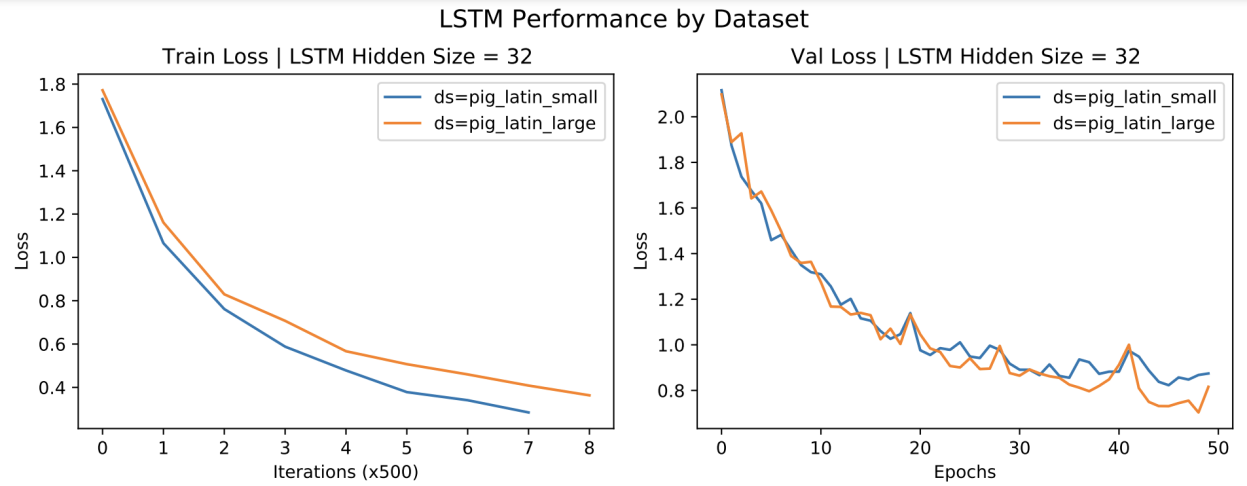
# Programming Assignment 3

## Part 1:LSTM

1.

```
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
```




The model trained on the smaller dataset performs better in terms of training loss, but for validation loss, the model trained on the larger dataset performs slightly better. This may be because with the small dataset there is overfitting so training on a larger dataset leads to better generalization.

2. **source:** this is phenomenal
**translated:** isthay isway enovablinay
A failure mode is translating more difficult or longer words (in this case 'phenomenal') that are most likely not present in the dataset. As you can see, there are some new letters incorporated while some letters are dropped.

3. Since the weights are shared, there is only one hidden state.
The number of connections in the encoder model is 4KH(H+D) because there are 4 gates each with H(H+D) connections over a length K.

## Part 2: Additive Attention

1.

$$\tilde{\alpha}_i^{(t)} = f(Q_t, K_i) = W_2[ReLU(W_1[Q_t; K_i] + b_1)] + b_2 = W_2[max(0, W_1[Q_t; K_i] + b_1)] + b_2$$

where $[Q_t; K_i]$ means concatenation.

$$\alpha_i^{(t)} = softmax(\tilde{\alpha}^{(t)})_i$$

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

4. As with Part 1 Q3, the number of units is 1. Based on the AdditiveAttention code provided, the attention network has K(2H*H+H) connections and the output layer has HV connections per timestep. Combining these connections with the number of connections of LSTMs calculated in Part 1 Q3, the total number of connections is $K^2(2H^2 + H) + KHV + 4KH(H + D)$.

## Part 3: Scaled Dot Product Attention

1.

```
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 = keys.size(0)
        q = self.Q(queries).view(batch_size, -1, self.hidden_size)
        k = self.K(keys).view(batch_size, -1, self.hidden_size)
        v = self.V(values).view(batch_size, -1, self.hidden_size)
        unnormalized_attention = torch.bmm(k, q.transpose(2,1)) * self.scaling_factor
        attention_weights = self.softmax(unnormalized_attention)
        context = torch.bmm(attention_weights.transpose(2,1), v)
        return context, attention_weights
```

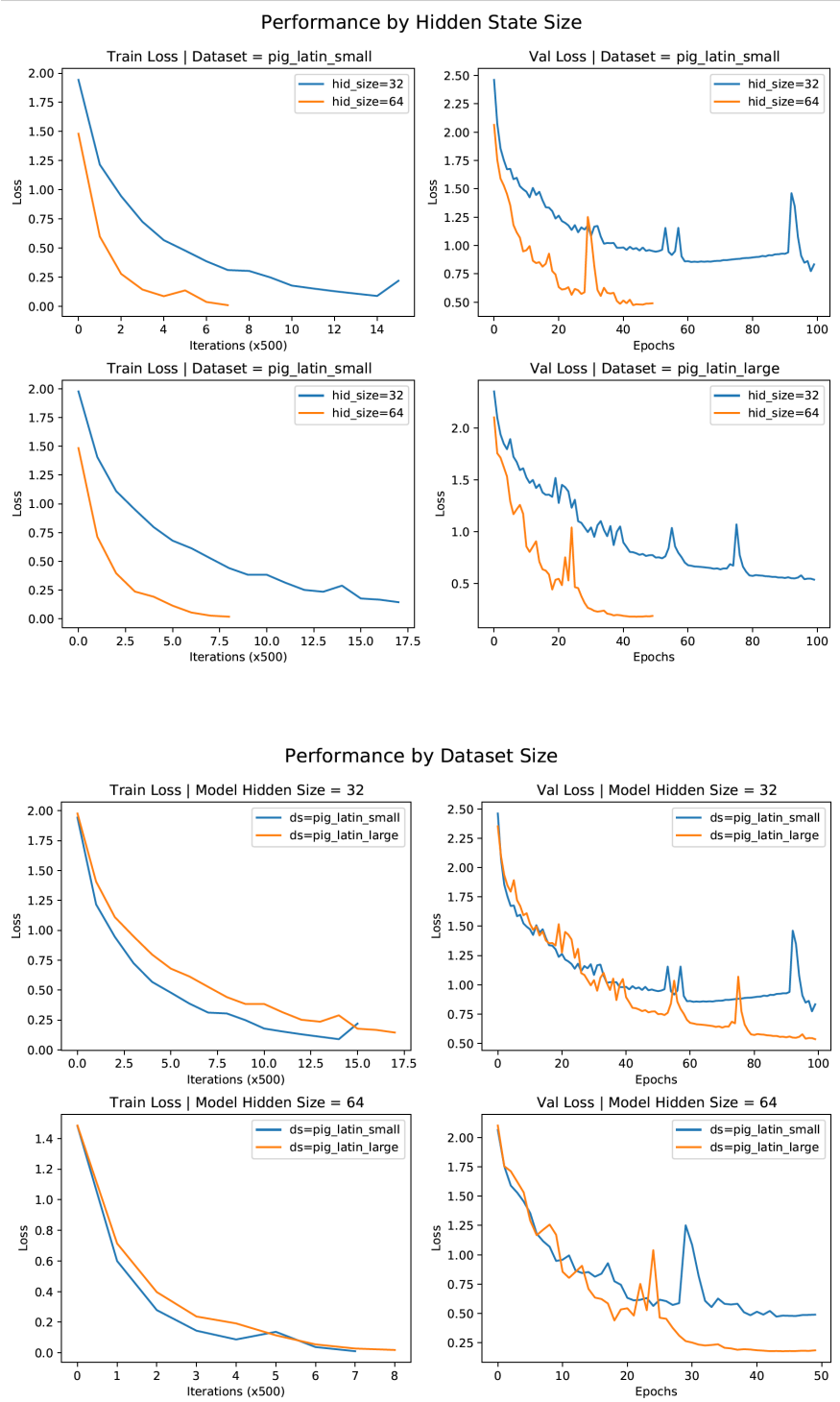
2.

```
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 = keys.size(0)
        q = self.Q(queries).view(batch_size, -1, self.hidden_size)
        k = self.K(keys).view(batch_size, -1, self.hidden_size)
        v = self.V(values).view(batch_size, -1, self.hidden_size)
        unnormalized_attention = torch.bmm(k, q.transpose(2,1)) * self.scaling_factor
        mask = torch.tril(self.neg_inf * torch.ones(unnormalized_attention.size()), diagonal=-1).cuda()
        attention_weights = self.softmax(unnormalized_attention + mask)
        context = torch.bmm(attention_weights.transpose(2,1), v)
        return context, attention_weights
```

3. Since transformer architecture uses multi-head attention, each word in a sentence is simultaneously inputted to the model so the model does not have a sense of the position or the order of the words. Thus, positional encoding is needed. The advantage of this particular method for positional encoding is that, unlike one hot encoding, it can be easily extended to represent position of varying lengths. This is important for transformers because, for example, when translating one language to another the length of the input might differ from the output.

4. The transformer model using hidden size 32 and the small dataset performed better than the LSTM trained on the small dataset but worse than the LSTM trained on the large dataset in terms of validation loss. This model also performed worse than the RNN model with additive attention. The translation of the sentence still has errors related to position of the letters ('ariway') and missing letters ('onditiongcay').

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

| Hidden Size | Dataset Size | Lowest Validation Loss |
|---|---|---|
| 32 | small | 0.7727 |
| 32 | large | 0.5358 |
| 64 | small | 0.4733 |
| 64 | large | 0.1779 |

Increasing hidden size decreases the number of iterations that is needed to be performed. This is expected since with larger hidden size, more information is captured about each input so the model can train with more information per iteration. In addition, with a larger dataset, the model is able to generalize better. This is also expected since with less data it is easy to overfit and so training on a smaller dataset typically leads to worse generalization.

## Part 4: Fine-tuning for arithmetic sentiment analysis

1. As with the linear layer for BERT, I implemented a linear layer for GPTCSC413.

```
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. Since GPT is unidirectional and auto-regressive hence only looks at past tokens, it may be preferred to use GPT for prediction tasks where there are no future tokens. An example of this might be to use GTP for generating search suggestions based on the set of words entered.