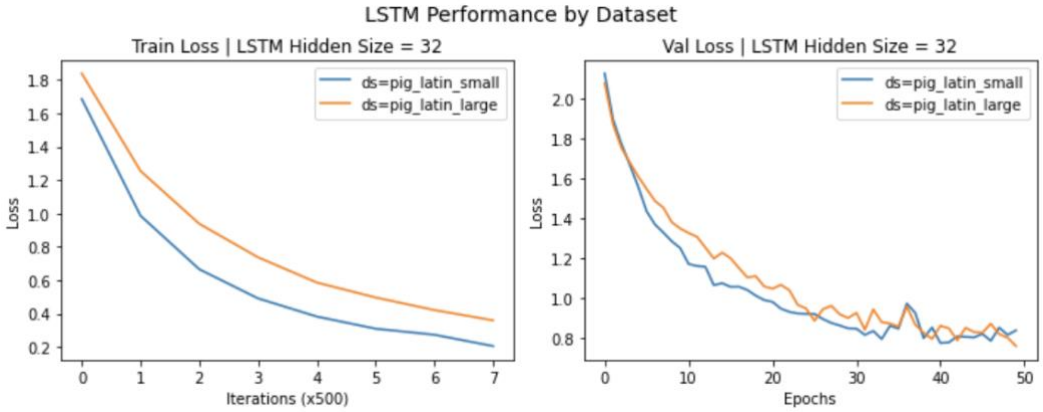
## Part 1: Long Short-Term Memory (LSTM)

1.
   - Screenshot of MyLSTMCell

```
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
```

- Loss Plots Output




No model performs significantly better than the other one. For RNN, the decoder only takes the final hidden state from the encoder as the input, but this compressed vector may lose much information for long sequences, therefore, for small or large dataset, the performance will still be very similar.

2. Failure Cases:

```
source:         the air conditioning is working
translated:     ethay ariway oncotingthay isway orguckway

source:         experienced an usual evening
translated:     expericenedway anway uusalway eveningway

source:         quick frozen fox jumps over the lazy dog
translated:     icuktay onfefray oxfay umpsjay overway ethay azyway ogday

source:         there are many things out there
translated:     eterhay areway anymay ingplyway outway eterhay

source:         as he crossed toward the pharmacy at the corner he involuntarily turned his head because of a burst of light that had ricocheted from his temple
translated:     asway ehay ossedmay owardtay ethay armachesay atway ethay ornercay ehay inviontrallyuay untrdeway ishay eadhay ecationway ofway away urstbay ofway ightlay atthay adhay iccoterdebay omf
```

By looking at the above failure cases, there are three cases that RNN may fail to translate:

i. When leading letters are all vowels: such as ‘experienced’, ‘usual’ are some failures in the second line.
ii. When leading letters are consonant pairs such as ‘fr’ and ‘th’ in words ‘frozen’, ‘there’, ‘things’ and ‘there’ in the third and fourth line
iii. For long sentences, the accuracy of translation has dramatically declined such as the last line.

3.

- Number of LSTM units: $H$
- Number of Connections in encoder: $(D \times H + H^2) \times 4K$

(I am not sure about this question, but I hope you can give me some partial marks if I explain my reasons here, really appreciate it☺ LSTM is using weight sharing in the hidden units, I think there will be no extra cost for each state, so the hidden size will simply be H. For number of connections, I counted the connections for 4 different gates including input-hidden and hidden-hidden connections)

## **Part 2: Additive Attention**

1.

$$\tilde{\alpha}_i^{(t)} = f(Q_t, K_i) = W_2\big(ReLU(W_1[Q_t; K_i] + b_1)\big) + b_2$$
$$\alpha_i^{(t)} = softmax(\tilde{\alpha}^{(t)})_i$$
$$c_t = \sum_{i=1}^{T} \alpha_i^{(t)} K_i$$

4.

- Number of LSTM units: $H$
- Number of Connections in attention network: $(H + 2H^2) \times KVD$

(Just like above, I explain my reasons here: number of units has the same reason as before. For number of connections in attention network: first linear layer has 2DHxH connections, second linear layer has DHx1 connections, since we need to take output sequence size the time step into consideration due to updating context vector: multiply it by KV in the end.)

## Part 3: Scaled Dot Product Attention

1. ScaledDotProduct

```
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))   #reciprocal of sqrt

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

2. CausalScaledProduct

```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.
        """

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

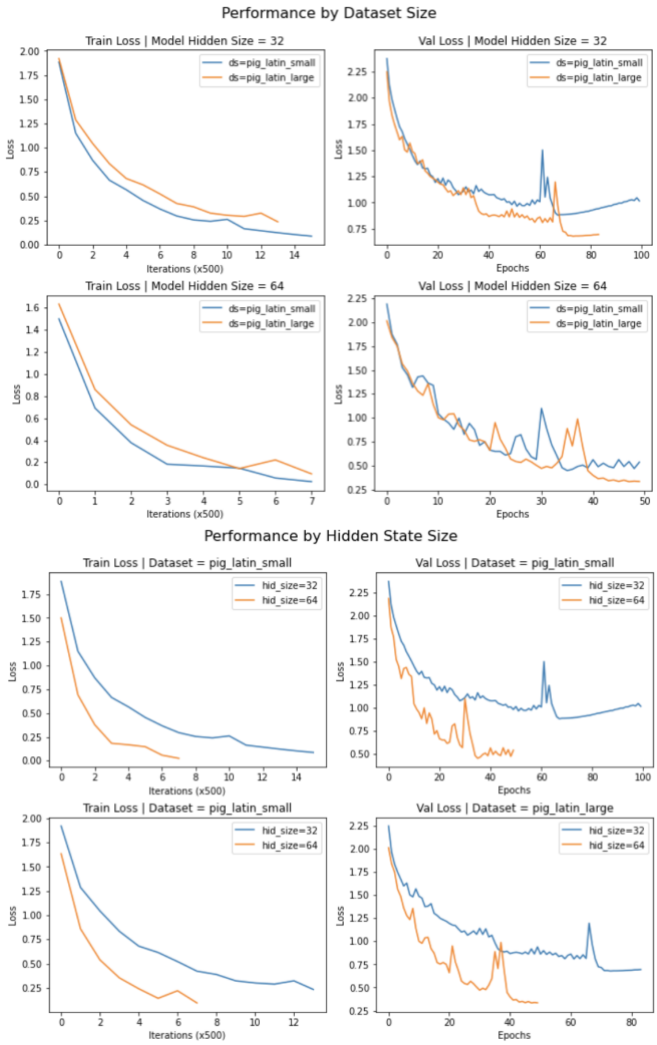
3. Unlike RNN which inherently take the order of words into account and parse the word by word sequentially, transformer increases its speed by introducing multi-head self-attention layers. However, it will lose the critical information of the order of the words to better understand the semantic, therefore, positional encoding is introduced here to provide this ‘useful information’. One hot encoding may make the model learn the embedding of early positions much more reliably than later positions, but sinusoid positional encoding allows the model to extrapolate to sequence length longer than ones in training process so that it can capture more pattern from encoding and can also be directly added to the input embedding.

4. At epoch 100, the training loss is 0.061 and obtained lowest validation loss is 0.731. By comparing the results with other two decoders, it clearly shows that the transformer model with hidden units 32 is better than the RNN decoder but worse than the RNN decoder with additive attention. However, the translation is very accurate as shown below.

```python
TEST_SENTENCE = 'the air conditioning is working'
translated = translate_sentence(TEST_SENTENCE, trans32_encoder_s, trans32_decoder_s, None, trans32_args_s)
print("source:\t\t{} \ntranslated:\t{}".format(TEST_SENTENCE, translated))
```

```
source:         the air conditioning is working
translated:     ethay airway onditiongcay isway orkingway
```

5.




**Effect of Dataset Size:** By looking at the first graph, for either hidden size of 32 or 64, small dataset has smaller training errors but higher validation errors than the large dataset. Although small dataset has slightly fewer iterations to converge to the lowest error in validation set, the trending of loss begins to increase at the end which means the model starts to overfit the data and generalize worse than the large dataset. This is expected since small dataset will always result in smaller error than larger dataset but will not generalize well to unseen data.

**Effect of hidden units:** By looking at the second graph, it clearly shows larger hidden size will have fewer iterations to converge for both training and validation process, the loss of large hidden size is also lower than the loss of small hidden size and no trending of increasing which means large hidden size will generalize better in the validation set. This is also expected since larger hidden size will capture more embedding information from more neurons.

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

1. GPTCSC413 class: I implement the same MLP layer as the BERT model in order to compare the two models later after training, the accuracy for both models is very similar.

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

```
finttune_gpt_loss_values, finttune_gpt_eval_accs = train_model(model_finetune_gpt, 4,
                                                               gpt_train_dataloader,
                                                               gpt_validation_dataloader) # about 5 minutes for 4 epochs using CPU
```

```
======== Epoch 1 / 4 ========
Training...

  Average training loss: 0.59
  Training epcoh took: 0:00:46
Running Validation...
  Accuracy: 0.91
  Validation took: 0:00:03

======== Epoch 2 / 4 ========
Training...

  Average training loss: 0.32
  Training epcoh took: 0:00:48
Running Validation...
  Accuracy: 0.95
  Validation took: 0:00:03

======== Epoch 3 / 4 ========
Training...

  Average training loss: 0.26
  Training epcoh took: 0:00:49
Running Validation...
  Accuracy: 0.95
  Validation took: 0:00:03

======== Epoch 4 / 4 ========
Training...

  Average training loss: 0.22
  Training epcoh took: 0:00:47
Running Validation...
  Accuracy: 0.95
  Validation took: 0:00:03

Training complete!
```

4. The key difference between GPT and BERT architectures is that they are uni-directional and bi-directional, respectively which mean GPT can only predict the future left-to-right context. This could be very useful in natural language generation areas where machines will help people to write documents or reports, the text generated will be prepared by human and GPT model can therefore predict the future texts autoregressively based the previous few words and save human's time.