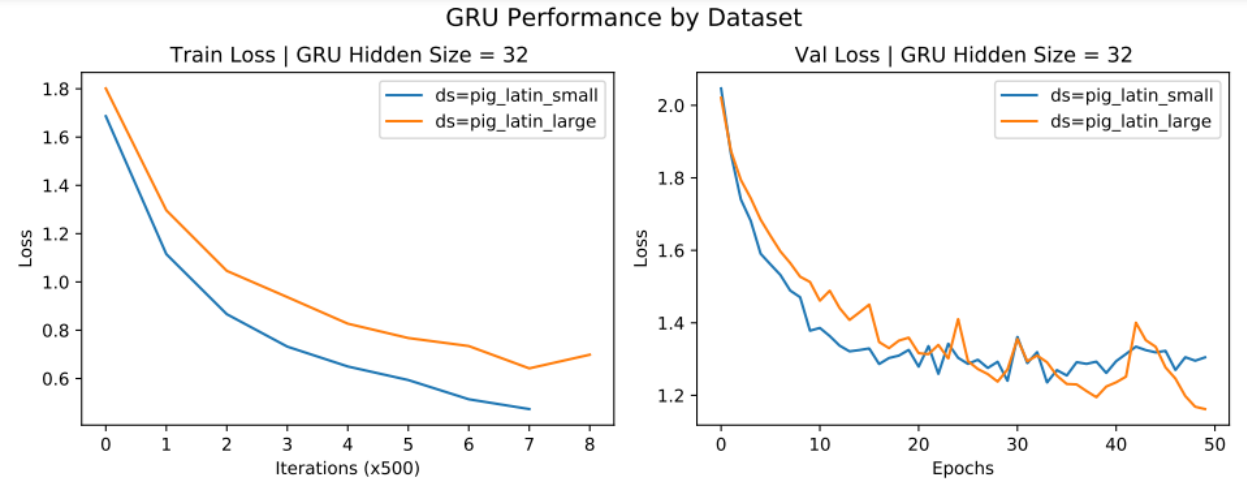
PA 3 Write Up

Part 1: Neural machine translation (NMT)

Answer to question 1:

```
[ ]  class MyGRUCell(nn.Module):
         def __init__(self, input_size, hidden_size):
             super(MyGRUCell, self).__init__()

             self.input_size = input_size
             self.hidden_size = hidden_size

             # ------------
             # FILL THIS IN
             # ------------
             # Input linear layers
             self.Wiz = nn.Linear(input_size, hidden_size)
             self.Wir = nn.Linear(input_size, hidden_size)
             self.Wih = nn.Linear(input_size, hidden_size)

             # Hidden linear layers
             self.Whz = nn.Linear(hidden_size, hidden_size)
             self.Whr = nn.Linear(hidden_size, hidden_size)
             self.Whh = nn.Linear(hidden_size, hidden_size)

         def forward(self, x, h_prev):
             """Forward pass of the GRU computation for one time step.

             Arguments
                 x: batch_size x input_size
                 h_prev: batch_size x hidden_size

             Returns:
                 h_new: batch_size x hidden_size
             """

             # ------------
             # FILL THIS IN
             # ------------
             z = torch.sigmoid(self.Wiz(x) + self.Whz(h_prev))
             r = torch.sigmoid(self.Wir(x) + self.Whr(h_prev))
             g = torch.tanh(self.Wih(x) + (self.Whh(r * h_prev)))
             h_new = (1-z) * h_prev + z * g
             return h_new
```



There does not seem to be a significant difference in terms of performance for the two models, but the model trained on the larger dataset has a slightly lower validation loss of 1.162 compared to the validation loss for the smaller data size of 1.235. This may be because the model is not complex enough to learn the rules of the language regardless of the dataset size (although larger dataset helps a bit more as it prevents the model from overfitting).

Answer to question 2:

Using the model trained on the larger dataset, we see that the model fails to translate long words:

```
source:         computer science
translated:     oppersthay ienceway
```

```
source:         undergraduate students
translated:     underarway undestway
```

```
source:         wonderful watermelon
translated:     onfurepray aterementway
```

Answer to question 3:

The total number of parameters in the LSTM encoder is $4\mathrm{DHK} + 4\mathrm{H}^2\mathrm{K}$ and the total number of parameters in the GRU encoder is $3\mathrm{DHK} + 3\mathrm{H}^2\mathrm{K}$.

Part 2.1: Additive Attention

Answer to question 3:

The training speed for the additive attention model is slower than the training speed for the model without attention because it needs to re-compute context vectors at every time step.

Part 2.2: Scaled Dot Product Attention

```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 k)

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

        # ------------
        # FILL THIS IN
        # ------------
        batch_size = queries.size(0)
        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, 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
```

```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.
        """

        # ------------
        # FILL THIS IN
        # ------------
        batch_size = queries.size(0)
        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, q.transpose(2,1)) * self.scaling_factor
        mask = torch.tril(torch.ones_like(unnormalized_attention) * self.neg_inf, diagonal = -1) # tril
        unnormalized_attention += mask
        attention_weights = self.softmax(unnormalized_attention)
        context = torch.bmm(attention_weights.transpose(2,1), v)
        return context, attention_weights
```

Answer to question 3:

The scaled dot product attention model performs worse than the RNNAttention model previously. This new model has a validation loss of 1.375 compared to the previous RNNAttention validation loss of 0.353. This may be because the RNNAttention encoder/decoder model is more complex and its additive attention's function f is a two-layer fully connected network with non-linearity introduced by the ReLU activation. Also the RNNAttention encoder encodes positional information while the attention encoder used in this model does not.

Answer to question 4:

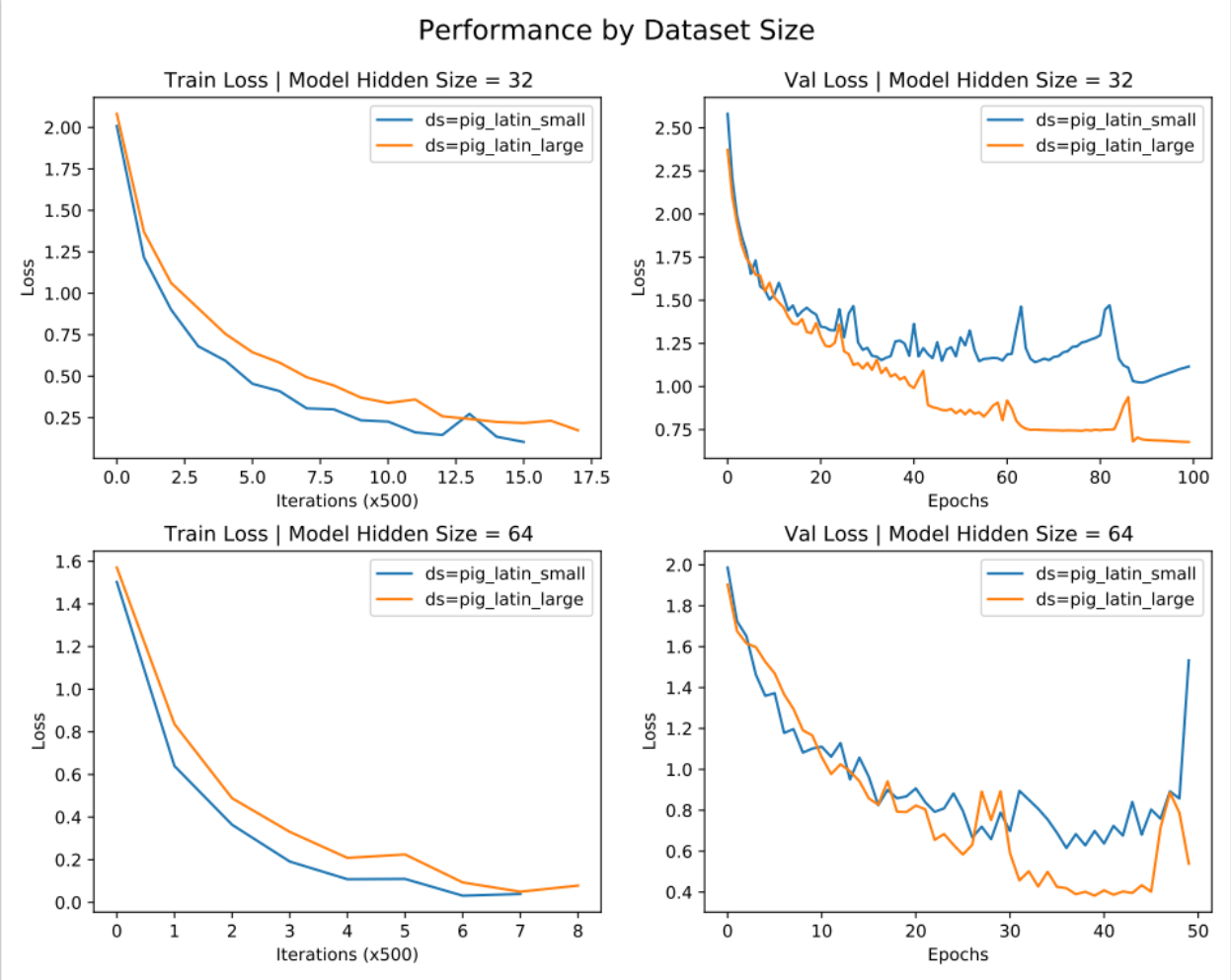
We need to encode the position of each word through positional encoding because the attention encoder outputs do not depend on the order of the inputs (unlike RNNs). The advantage of using this positional encoding method as opposed to one hot encodings is that it results in less sparse encodings

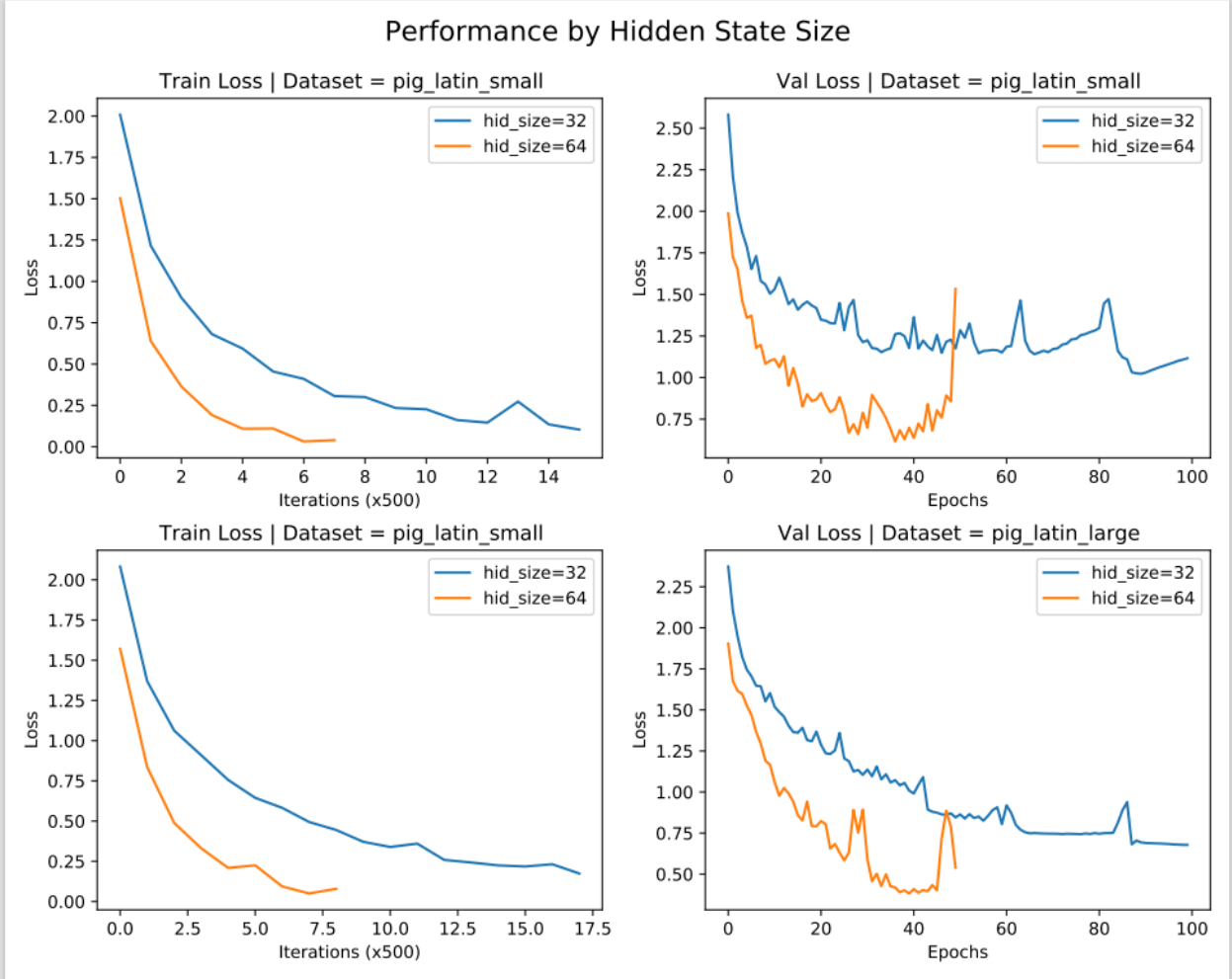
and it also allows the model to extrapolate to sequence lengths longer than the ones encountered during training.

Answer to question 5:

Comparing the Transformer encoder/decoder (with hidden size 32 and small dataset) with the RNNAttention model, the RNNAttention model still achieved a lower validation score (generalizes better) and had a more accurate translation on the test sentence. However, the Transformer model performed better than the Attention model (Transformer reached lower validation loss and so generalizes better) and gave a more accurate translation of the test sentence. This is expected as the Transformer is more complex than the Attention model. It has a greater number of layers and adds positional encoding.

Answer to question 6:





lowest attained validation loss:

hidden size 32, pig_latin_small: 1.0229302943599494

hidden size 32, pig_latin_large: 0.6785160089628055

hidden size 64, pig_latin_small: 0.6152846815741875

hidden size 64, pig_latin_large: 0.3825042962920494

It appears that increasing the model complexity via hidden size and increasing the dataset size both result in better generalization of the model (lower validation loss and better translations). We see from the first graph Performance by Dataset Size that regardless of the hidden size, increasing the dataset size increases the generalization abilities of the model. Furthermore, as shown in the second graph Performance by Hidden Size, regardless of the dataset size, increasing the hidden size results in faster decrease for both the validation and training loss as a function of gradient descent iterations. These results are as expected since a larger dataset provides the model more examples to learn the translation

rule from while increased complexity (increased hidden size) enables the model to learn more complex rules/relationships faster.

Part 3: Fine-tuning Pretrained Language Models (LMs)

Answer to question 1:

```
[ ]  from transformers import BertModel
     import torch.nn as nn

     class BertForSentenceClassification(BertModel):
         def __init__(self, config):
             super().__init__(config)

             ##### START YOUR CODE HERE #####
             # Add a linear classifier that map BERTs [CLS] token representation to the unnormalized
             # output probabilities for each class (logits).
             # Notes:
             #  * See the documentation for torch.nn.Linear
             #  * You do not need to add a softmax, as this is included in the loss function
             #  * The size of BERTs token representation can be accessed at config.hidden_size
             #  * The number of output classes can be accessed at config.num_labels
             self.classifier = nn.Linear(config.hidden_size, config.num_labels)
             ##### END YOUR CODE HERE #####
             self.loss = torch.nn.CrossEntropyLoss()

         def forward(self, labels=None, **kwargs):
             outputs = super().forward(**kwargs)
             ##### START YOUR CODE HERE #####
             # Pass BERTs [CLS] token representation to this new classifier to produce the logits.
             # Notes:
             #  * The [CLS] token representation can be accessed at outputs.pooler_output
             cls_token_repr = outputs.pooler_output
             logits = self.classifier(cls_token_repr)
             ##### END YOUR CODE HERE #####
             if labels is not None:
                 outputs = (logits, self.loss(logits, labels))
             else:
                 outputs = (logits,)
             return outputs
```

Answer to question 2:

Answer to question 3:

The training time when the BERTs weights are frozen is shorter than finetuning BERT in question 2. This is because there are less parameters to tune and update (less computations) when the BERTs weights are frozen. However, the effect is that compared to fine tuning in q2, the performance (validation accuracy) is much lower for the frozen BERTs weights model. This is because allowing for finetuning BERTs weight would enable the model to optimize over more parameters (learn more relevant features/encodings) to generalize better to the specific task at hand.

Answer to question 4:

Finetuning BERT with the pretrained weights from BERTweet results in lower validation accuracy than finetuning BERT with pretrained weights from MathBERT. This is probably because the dataset that we are using to finetune (training and validation dataset) is more similar to the mathematical corpus that MathBERT pretrained on than the corpus of tweets that BERTweet pretrained on, so the MathBERT model is able to apply its pre-learned knowledge better.

Part 4: Connecting Text and Images with CLIP

```
caption = "butterfly on purple flower"
```

I first used the caption “butterfly on flower” and got a butterfly on a pink flower, so I changed it to “butterfly on purple flower” to get the imaged as wanted.