CSC413 Programming Assignment 3

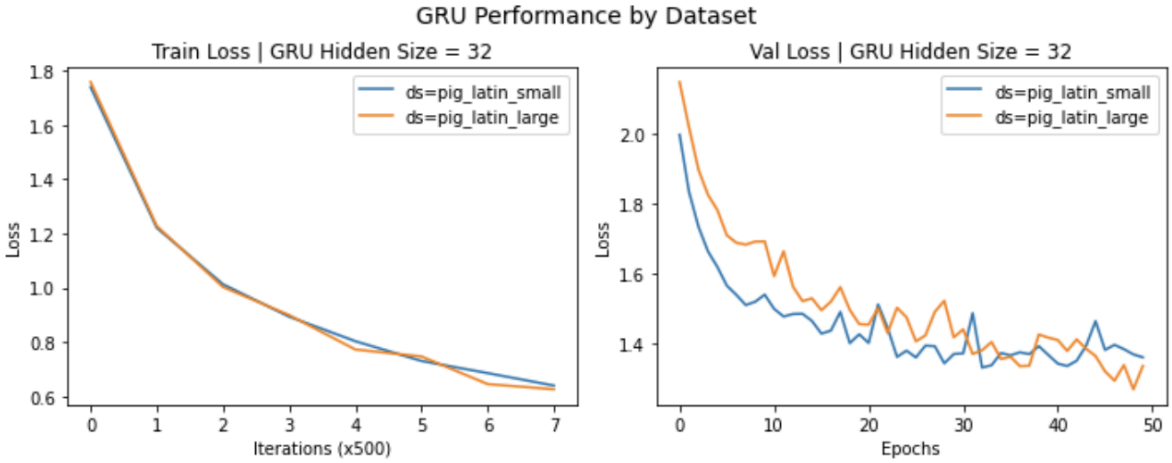
First Name: Kwan Kiu
Last Name: Choy
Student no: 1005005879

Part 1 Neural Machine Translation

Part 1 Neural Machine Translation

```
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
```



Question 1

When the model is trained on a larger dataset, it achieves better results. We can see from the graph that at the final epoch, the model that was trained with a larger dataset achieved a lower validation loss than the model trained with small dataset, the lowest validation loss in the model trained with a bigger dataset is also lower. This is because it learns more words and thus generalizes and performs better.

Question 2

It seems that words that contain consonant pairs e.g: "sh", "th" , "wh" and "ch". Apart from the TEST_SENTENCE provided, I have also tried "**shot the short chipmunk with thick bullet**", which got translated into "**otssay etcay ortsway impostroupsway ithway icktay ulberay-eantway**". This might be caused by not having enough examples in the training data even in the large dataset.

Question 3

LSTM Encoder: **4K * (D*H + H^2)** ; GRU Encoder: **3K * (D* H + H^2)**

Part 2: Addictive Attention

Question 3

The training speed of the Addictive attention model is slightly longer (6 seconds longer) than the original one (in above). The reason for that is although there is an early stop, more time would be needed when more inputs were being fed at each step.

Part 2.2 Scaled Dot Attention

```
class ScaledDotAttention(nn.Module):
   def __init__(self, hidden_size):
       super(ScaledDotAttention, self).__init__()89
```

```
        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 = keys.shape[0]
        q = self.Q(queries).view(batch_size, -1, self.hidden_size)
        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
```

```
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 = keys.shape[0]
       q = self.Q(queries).view(batch_size, -1, self.hidden_size)
       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(unnormalized_attention.shape),
```

```
diagonal=-1).to(device ='cuda') * self.neg_inf
        unnormalized_attention = unnormalized_attention + mask
        attention_weights = self.softmax(unnormalized_attention)
        context = torch.bmm(attention_weights.transpose(2, 1), v)
        return context, attention_weights
```

Question 3

The single block attention model performs worse than the RNNAttention model. We can see that from the much bigger lowest validation loss here (RNNAttention: 0.2, single block model: 1.2). One reason for that is because more time is needed to process the keys, queries, and values pairs, implying an increase in the number of parameters, and hence need training steps would increase.

Question 4

Same word in different positions can have different meanings. The positional embeddings are useful because it indicates the order of the tokens while the model that we are training does not keep track of the orders. Using the sin and cos for position embedding is better than one hot encoding because sin(x+k) and cos(x+k) where k is the offset, can be expressed as a linear combination of sin and cos. Moreover, sin and cos would be within the range of [-1, 1], meaning that these embeddings will be continuous and can thus give us more combinations with the same embedding dimension compared to one hot.

Question 5

Comparisons with RNNAttention

Lowest validation loss from transformers: 0.7849285908043384

Lowest validation loss from RNNAttention: 0.20935316750994667

As you can see, from the above, the transformer model perform worse than the RNNAttention model, since the lowest validation loss obtained from transformers is higher than that of the RNNAttention model. In terms of training time however, the transformer model's training time is less than that of the RNN Attention model.

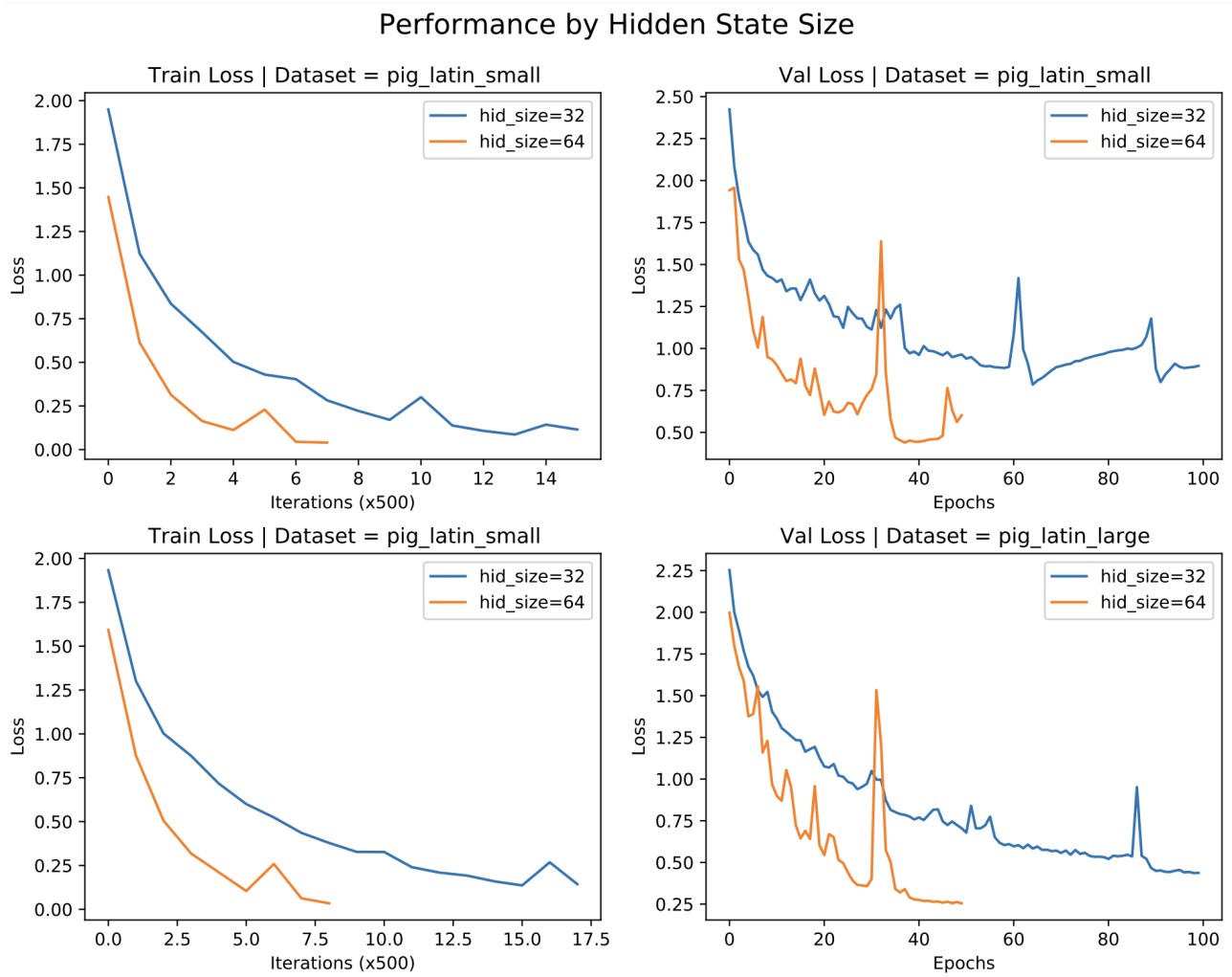
Comparisons with Single Block Attention

Lowest validation loss from transformers: 0.7849285908043384

Lowest validation loss from Single Block Attention: 1.1933680046827366

From the above, we can see that the transformers have a lower lowest validation loss than that of the single block attention. In terms of training speed, the transformer model is slower than the single block attention model.

6.

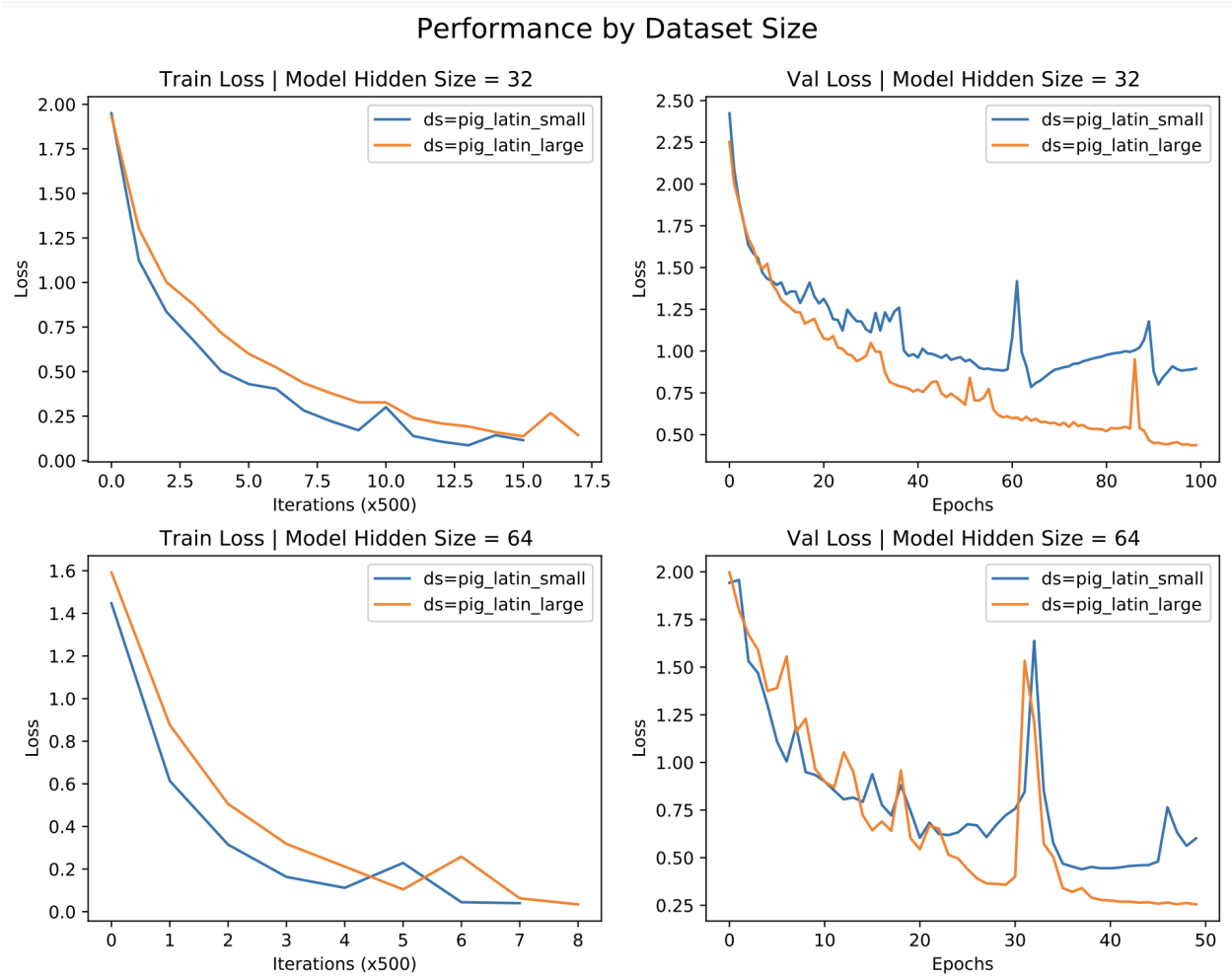


Validation Losses:

	Hidden_size =32	Hidden size = 64

Dataset small	0.7849285908043384	0.43915095590054987
Dataset big	0.43609637812098334	0.25479348254276024

From the above plots and figures, we can see that as the hidden state increases, the validation losses will increase. This is what we will expect, as hidden state size increases, the model can generalize better.



Validation Losses:

	Hidden_size =32	Hidden size = 64
Dataset small	0.7849285908043384	0.43915095590054987

Dataset big	0.43609637812098334	0.25479348254276024

From the above plots and figures, we can see that regardless of what hidden state size is, the model that is trained on a large dataset will usually have a lower validation loss than those that are trained with a small dataset. This make sense as with a larger dataset, the model learned more words, and hence would have a higher chance of performing better on an unseen sentence.

Part 2: BERT

```
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
```

Question 3

When BERT weights are frozen the training time of the model decreases from 0:00:04 to 0:00:01. The validation accuracy also dropped when compared to the second model from 0.9 to 0.3 for the final epoch. This is because the time needed to update the weights are saved, and as a result, there are no adjustments made to the weights, thus a drop in accuracy.

Question 4

The accuracy obtained using pre-trained weights from BERTweet is lower than that of the fine-tuned Bert model from weights obtained from MathBERT. This is because the MathBERT corpus is more similar to the dataset that we are given for this task, hence the weights trained from the MathBERT corpus give us better results for our task.

Part 3 CLIP

caption = "A yellow butterfly is on a purple flower in front of a green background"