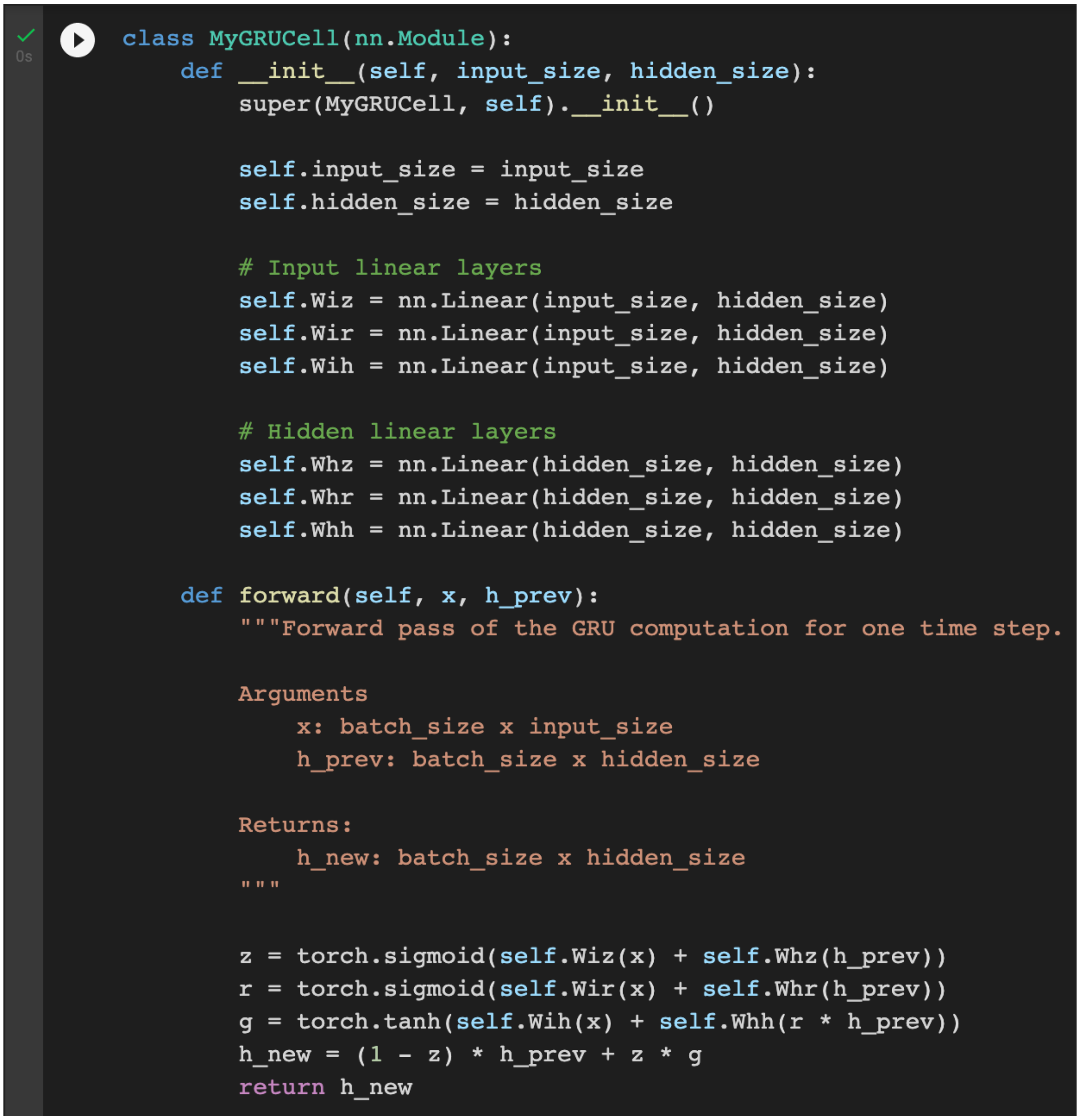
CSC413 Programming Assignment 3

Part 1: Neural machine translation (NMT)

Q1. GRU implementation.



```python
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

        # 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
        """

        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
```

The model trained using the larger dataset performs significantly better, since it has lower validation loss based on the loss curves. The reason might be that the more training data, the more information the trained model could obtain, and then the better performance.

Q2. Identify failure modes.

source:	implement scaled dot-product attention
translated:	impererlay-etetay alesestay othtay-eatormay atetingway
source:	there are significant differences
translated:	eertway areway igingsay-ondentway isefteredtay-oodway

As the above two test sentences and corresponding translated results show, the most common failure mode is the poor performance on longer words, such as “implement” and “significant”. Other failure modes might be poor performance on consonant pairs such as “sc” and “th”, and the words not in the training set.

Q3. Comparing complexity.

total number of parameters of the LSTM encoder: 4(D + H) * H
total number of parameters of the GRU encoder: 3(D + H) * H

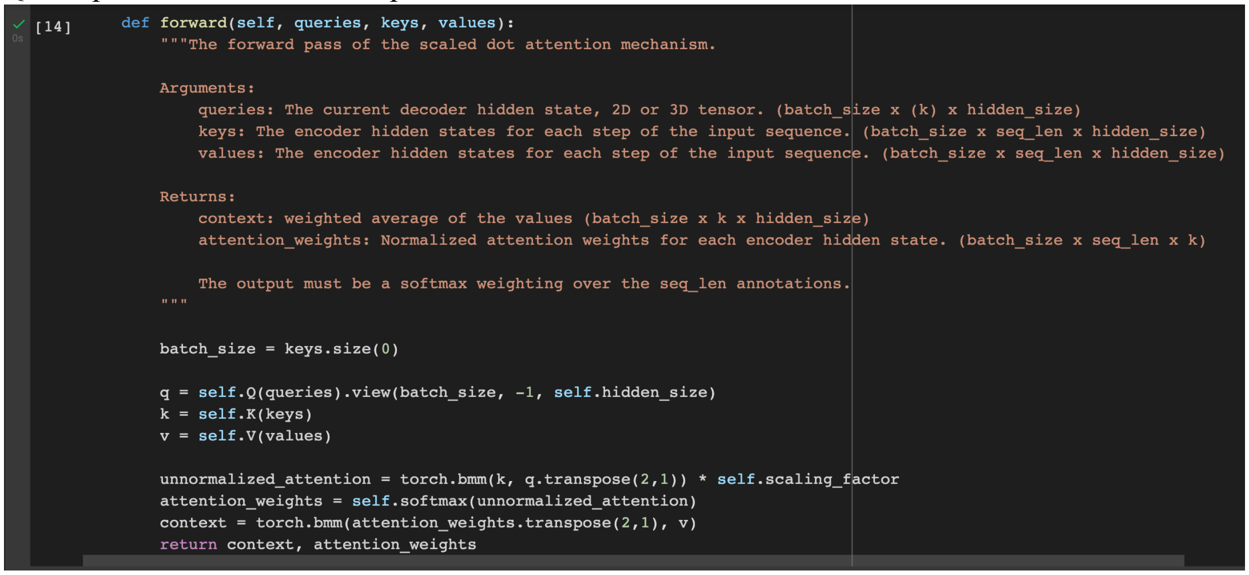
Part 2.1: Additive Attention

Q3. How does the training speed compare? Why?

The training speed of model with additive attention is slower than the previous model without attention, which might be due to extra computation in additive attention, that is, more parameters.

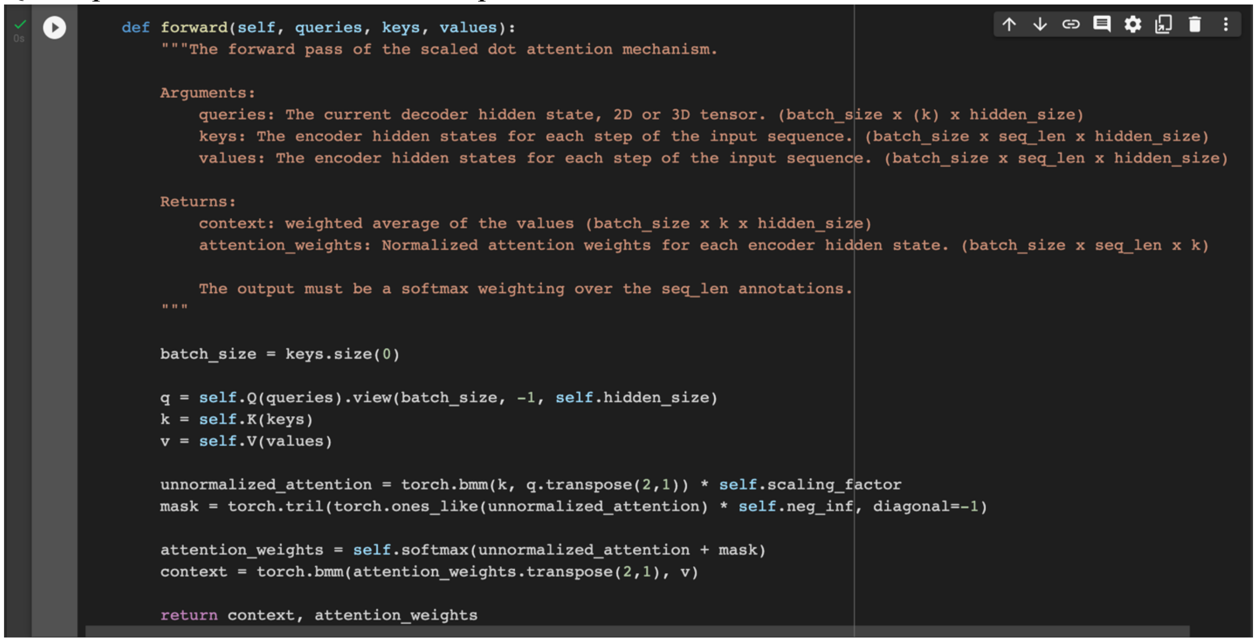
Part 2.2: Scaled Dot Product Attention

Q1. Implement the scaled dot-product attention mechanism.



```
[14]    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.
            """

            batch_size = keys.size(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
```

Q2. Implement the causal scaled dot-product attention mechanism.



```
    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.
        """

        batch_size = keys.size(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_like(unnormalized_attention) * self.neg_inf, diagonal=-1)

        attention_weights = self.softmax(unnormalized_attention + mask)
        context = torch.bmm(attention_weights.transpose(2,1), v)

        return context, attention_weights
```

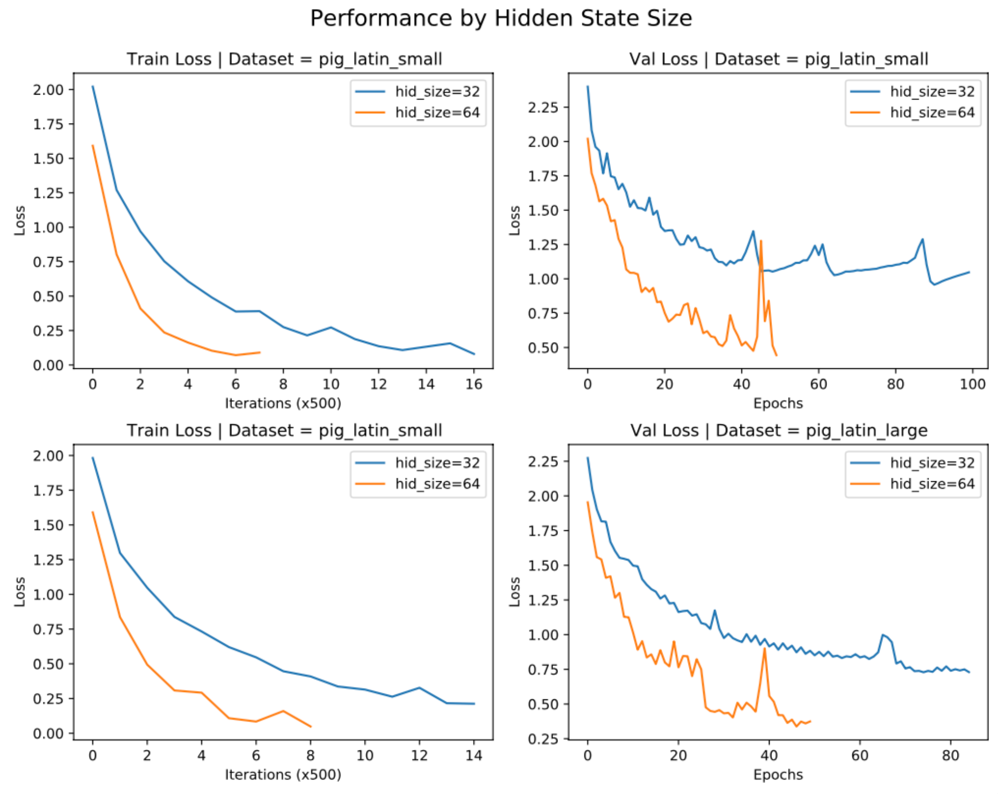
Q3.
validation loss of the model using RNNAttention: 0.3938221916090697
validation loss of the model using ScaledDotAttention: 1.1820401936769485
As the training output shows, the validation loss of the ScaledDotAttention model is greater, so it has a worse performance compared to the RNNAttention model.
This might because ScaledDotAttention does not work well for sequential encoding/decoding.

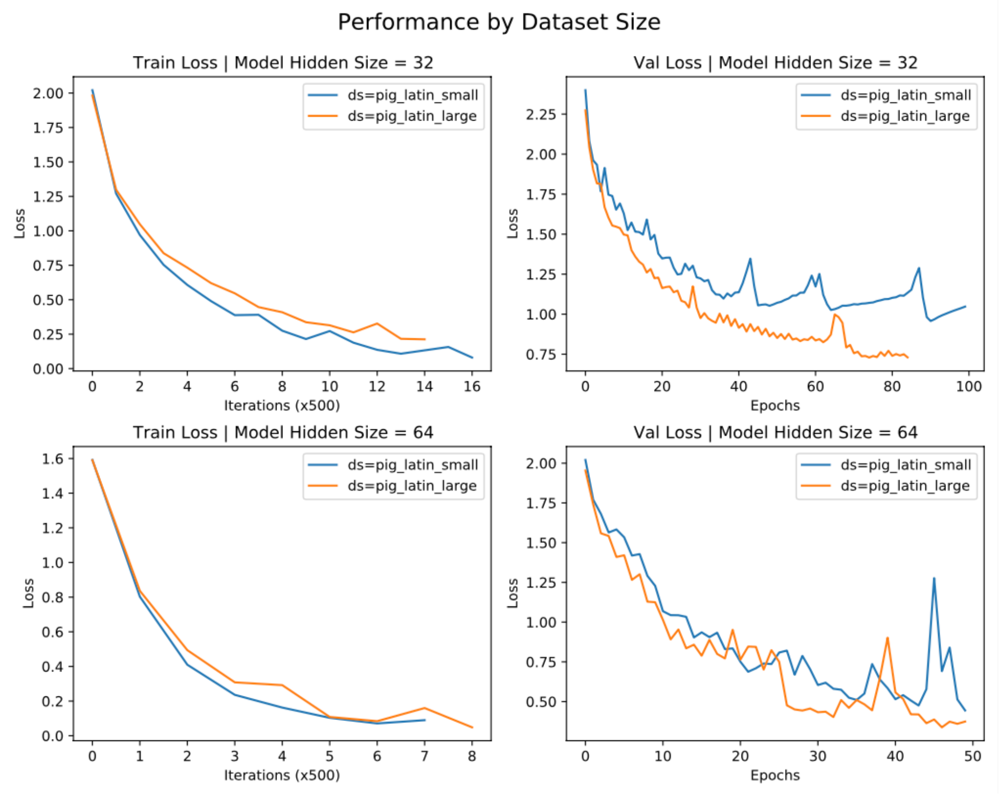
Q4.
The reason why we need to represent the position of each word through this positional encoding is that positional encodings could represent the relative position of the tokens within the sequence, which is specifically required for the transformer.
Compared to methods such as a one hot positional encoding, this "sine and cosine" method could handle longer sentences and have better generalization.

Q5.
Translation results for the model with transformer generally performed as well or worse than the RNNAttention and single-block Attention decoders. This might be due to the combination of the small model capacity and dataset size.

Q6.
The lowest attained validation loss for each run:
Hidden size = **32** with **small** dataset, obtained lowest validation loss of: 0.9571305304765702
Hidden size = **32** with **large** dataset, obtained lowest validation loss of: 0.7289705030464878
Hidden size = **64** with **small** dataset, obtained lowest validation loss of: 0.44411674374714494
Hidden size = **64** with **large** dataset, obtained lowest validation loss of: 0.3379830677634648

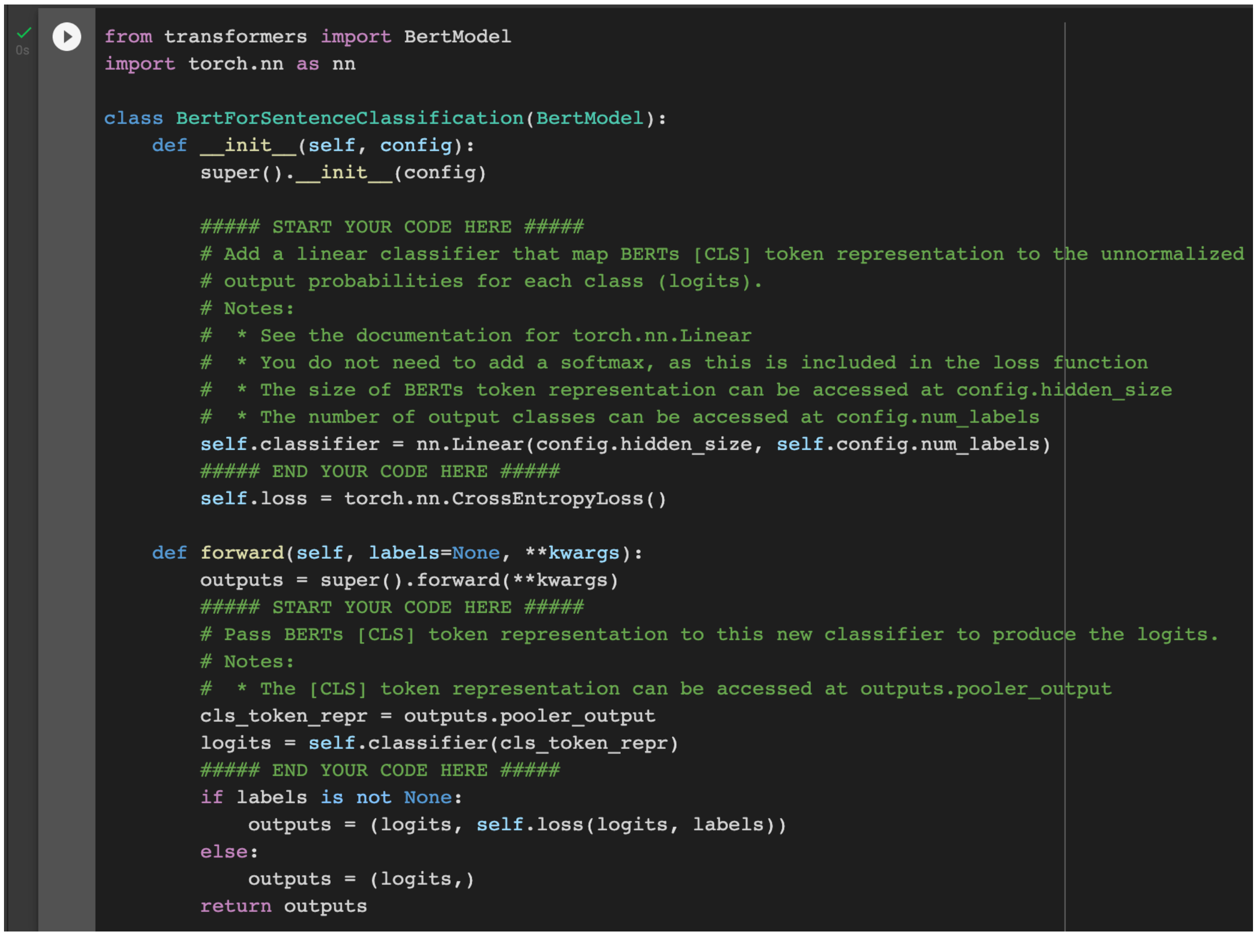
The intended result is that increasing model capacity via the hidden size and increasing dataset size both lower the validation loss, that is, increase the model generalization performance.





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

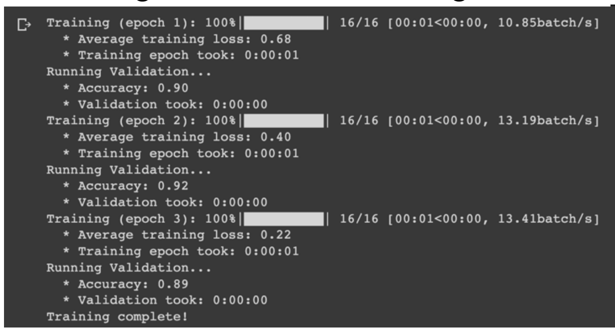
Q1. BertForSentenceClassification Implementation.



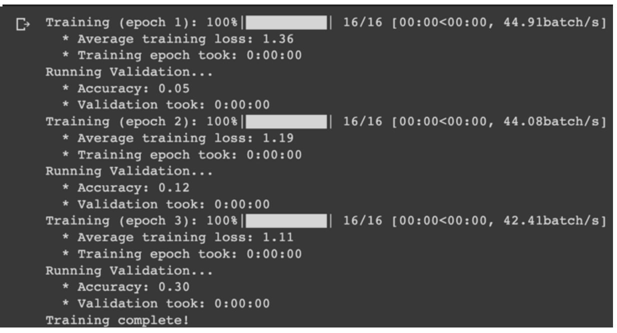
```python
from transformers import BertModel
import torch.nn as nn

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, self.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
```

Q3. Freezing the pretrained weights.

Fine-tuning VS frozen BERTS weights

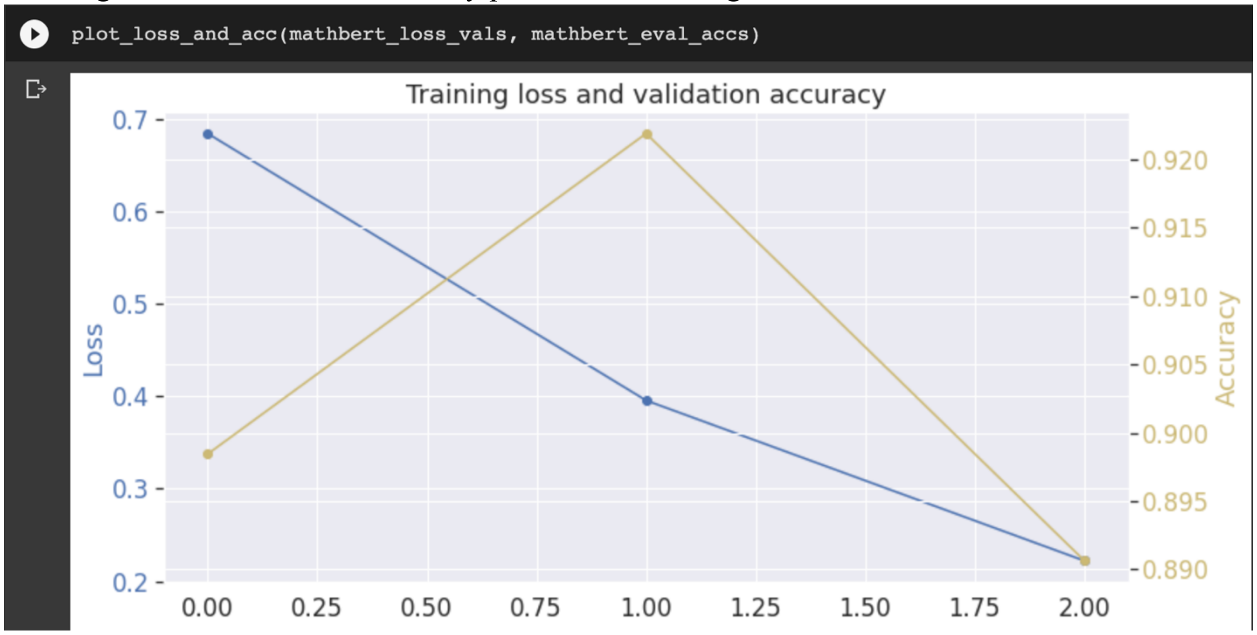


```
Training (epoch 1): 100%|██████████| 16/16 [00:01<00:00, 10.85batch/s]
  * Average training loss: 0.68
  * Training epoch took: 0:00:01
Running Validation...
  * Accuracy: 0.90
  * Validation took: 0:00:00
Training (epoch 2): 100%|██████████| 16/16 [00:01<00:00, 13.19batch/s]
  * Average training loss: 0.40
  * Training epoch took: 0:00:01
Running Validation...
  * Accuracy: 0.92
  * Validation took: 0:00:00
Training (epoch 3): 100%|██████████| 16/16 [00:01<00:00, 13.41batch/s]
  * Average training loss: 0.22
  * Training epoch took: 0:00:01
Running Validation...
  * Accuracy: 0.89
  * Validation took: 0:00:00
Training complete!
```



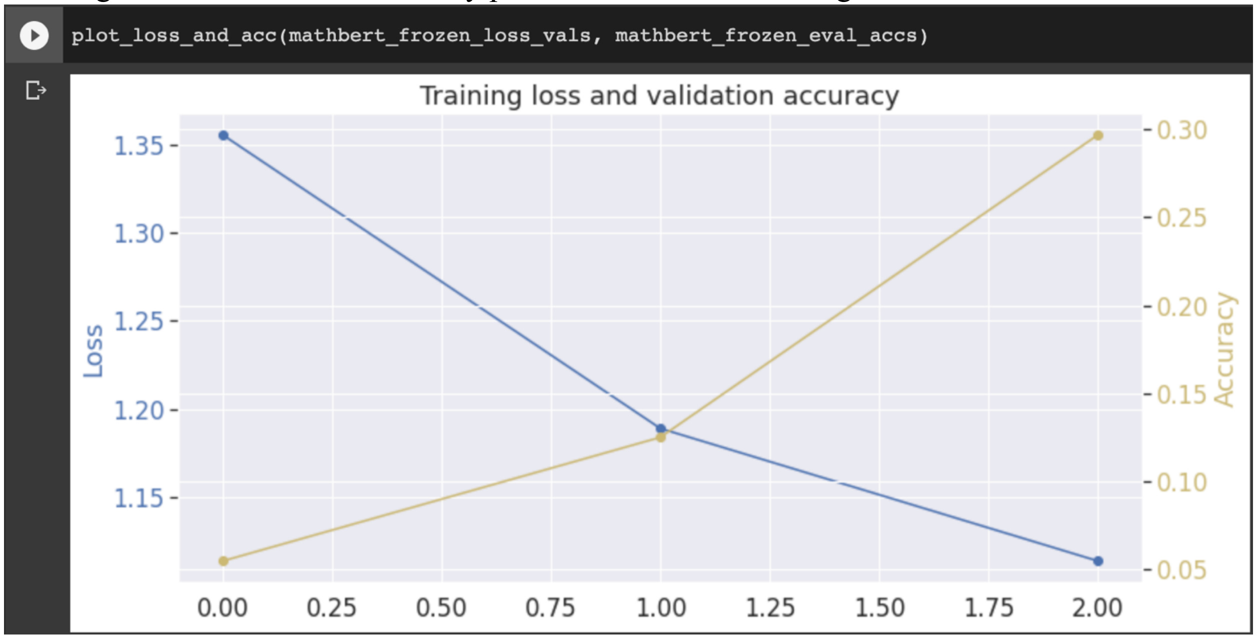
```
Training (epoch 1): 100%|██████████| 16/16 [00:00<00:00, 44.91batch/s]
  * Average training loss: 1.36
  * Training epoch took: 0:00:00
Running Validation...
  * Accuracy: 0.05
  * Validation took: 0:00:00
Training (epoch 2): 100%|██████████| 16/16 [00:00<00:00, 44.08batch/s]
  * Average training loss: 1.19
  * Training epoch took: 0:00:00
Running Validation...
  * Accuracy: 0.12
  * Validation took: 0:00:00
Training (epoch 3): 100%|██████████| 16/16 [00:00<00:00, 42.41batch/s]
  * Average training loss: 1.11
  * Training epoch took: 0:00:00
Running Validation...
  * Accuracy: 0.30
  * Validation took: 0:00:00
Training complete!
```

Training loss and validation accuracy plot for Fine-tuning



```
plot_loss_and_acc(mathbert_loss_vals, mathbert_eval_accs)
```

Training loss and validation accuracy plot for frozen BERTS weights

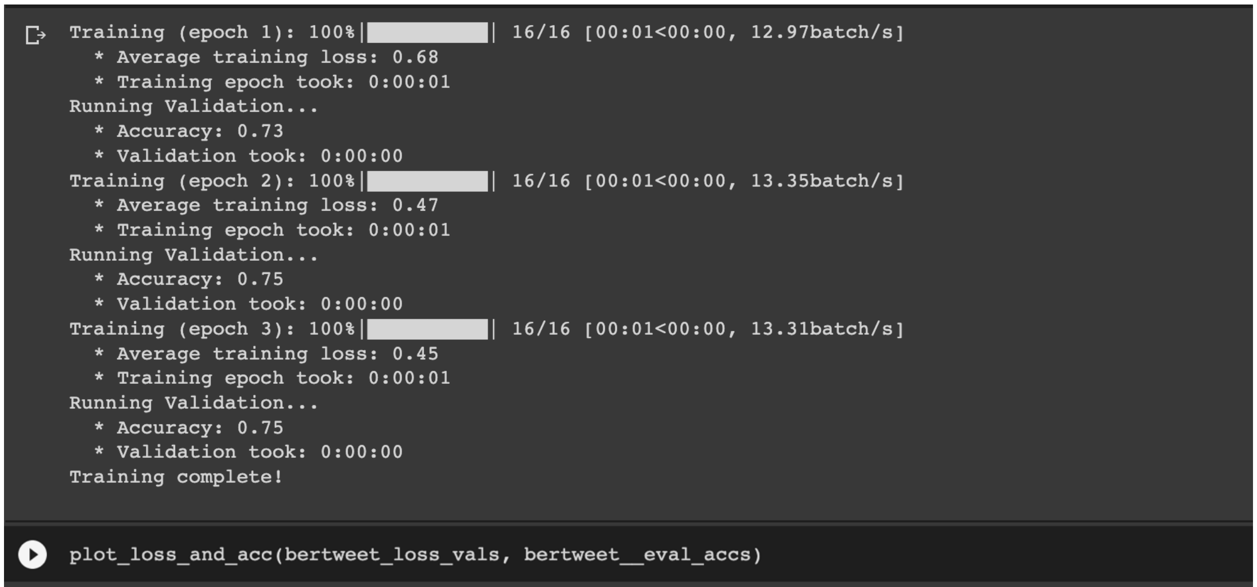


```
plot_loss_and_acc(mathbert_frozen_loss_vals, mathbert_frozen_eval_accs)
```

As the above output results show,

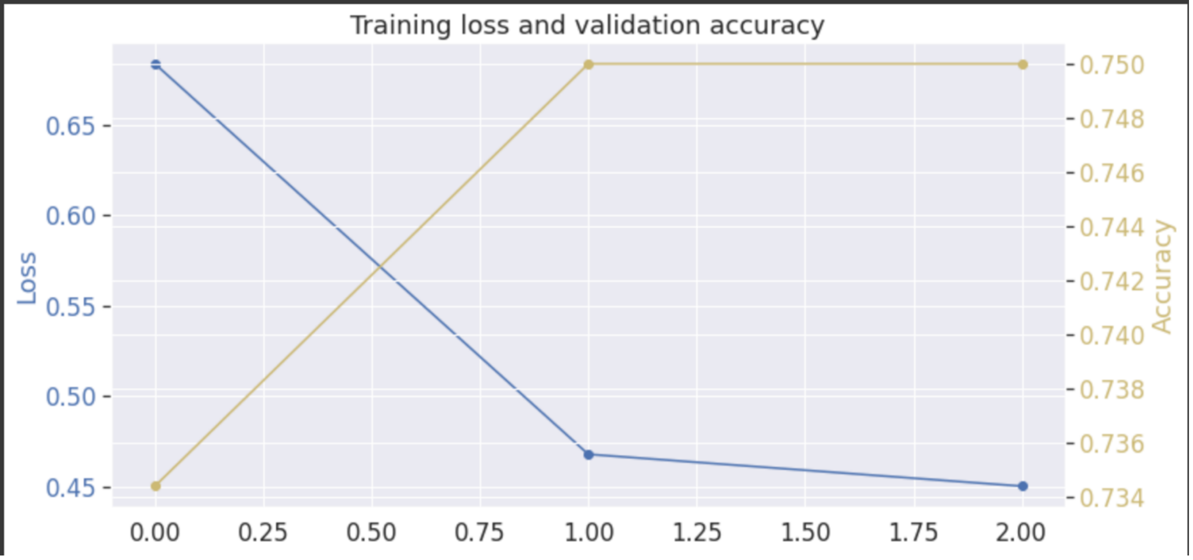
1. train time decreases when BERTs weights are frozen since it could skip gradient computation for the frozen layers.
2. Validation accuracy decreases from 0.89 to 0.3 when BERTs weights are frozen since it lost the information inside the frozen layers or the BERTS weights were not well-trained.

Q4. Effect of pretraining data.



```
Training (epoch 1): 100%|██████████| 16/16 [00:01<00:00, 12.97batch/s]
  * Average training loss: 0.68
  * Training epoch took: 0:00:01
Running Validation...
  * Accuracy: 0.73
  * Validation took: 0:00:00
Training (epoch 2): 100%|██████████| 16/16 [00:01<00:00, 13.35batch/s]
  * Average training loss: 0.47
  * Training epoch took: 0:00:01
Running Validation...
  * Accuracy: 0.75
  * Validation took: 0:00:00
Training (epoch 3): 100%|██████████| 16/16 [00:01<00:00, 13.31batch/s]
  * Average training loss: 0.45
  * Training epoch took: 0:00:01
Running Validation...
  * Accuracy: 0.75
  * Validation took: 0:00:00
Training complete!
```

```
plot_loss_and_acc(bertweet_loss_vals, bertweet__eval_accs)
```

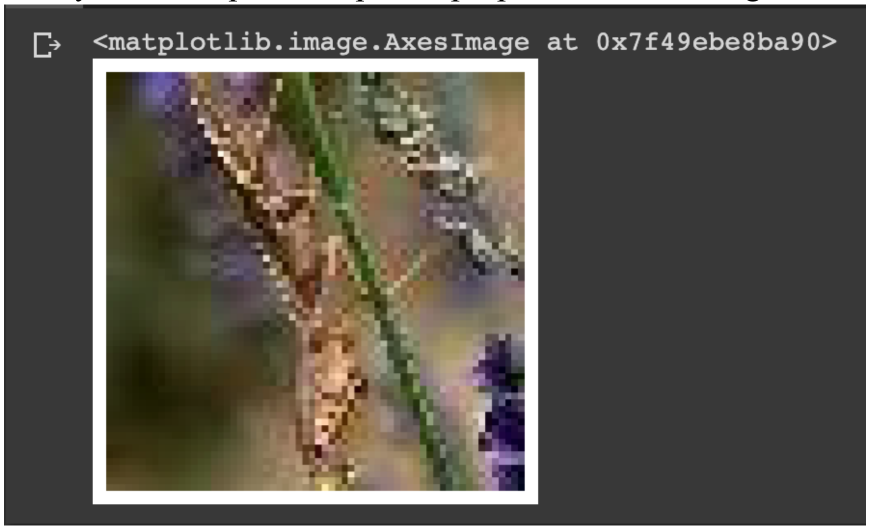


The highest validation accuracy of BERTweet model is 0.75, which is lower than 0.92 of the MathBERT model. This might because their pretrained datasets are different, specifically, the BERTweet model pretrained based on English tweets, while the MathBERT model pretrained based on a large mathematical corpus ranging from pre-kindergarten, to high-school, to college graduate level mathematical content.

Part 4: Connecting Text and Images with CLIP

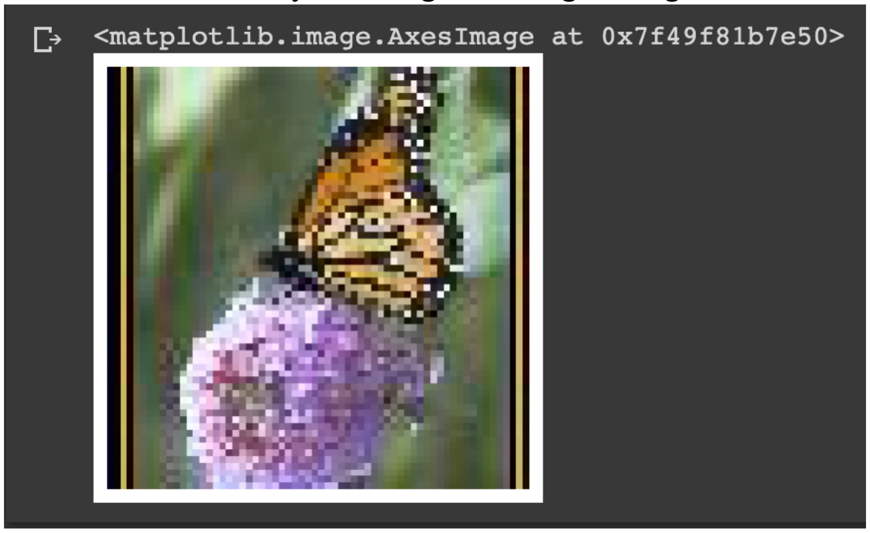
Q2. Prompting CLIP.

Firstly, I came up with caption "purple flower", but I got this.



```
<matplotlib.image.AxesImage at 0x7f49ebe8ba90>
```

Then I tried "butterfly", and I got the target image.



```
<matplotlib.image.AxesImage at 0x7f49f81b7e50>
```

I think the process is relatively easy, since the main features of the target image are "purple flower" and "butterfly".