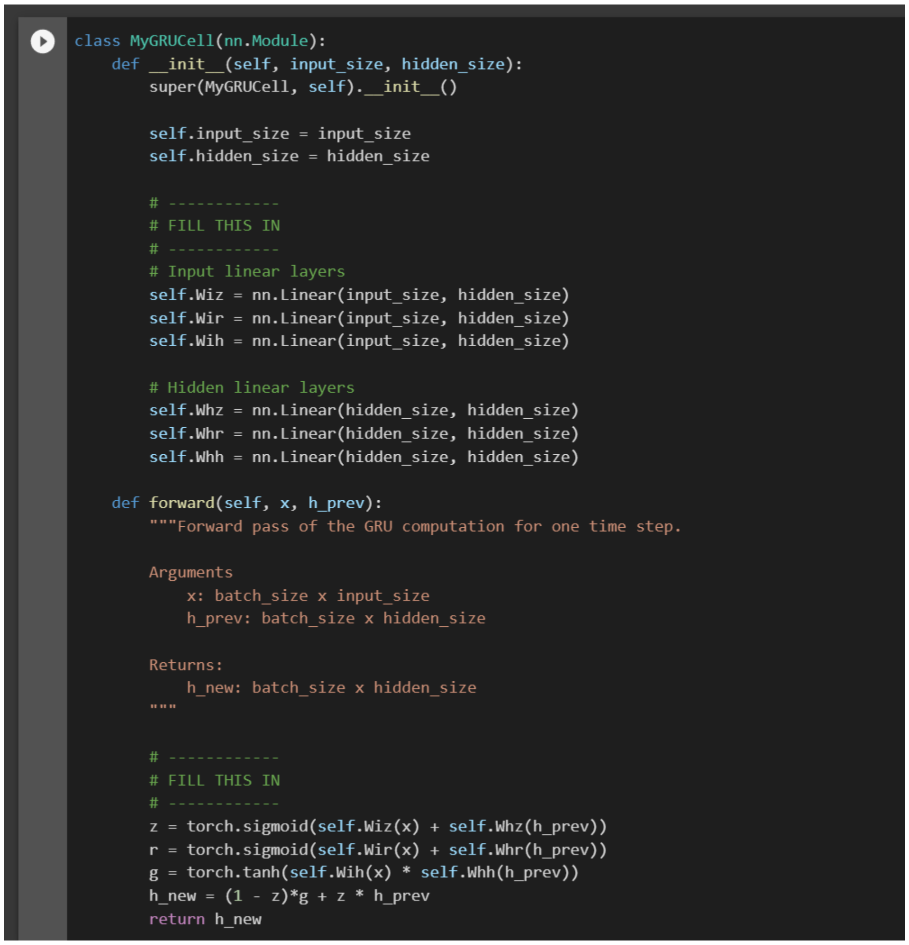
Programming Assignment 3: Natural Language Processing and Multimodel Learning

Name: Ruyi Qu
Student Number: 1004849569

Part 1 : Neural machine translation (NMT)

Q1 - Code a GRU Cell

The shown picture below is the screenshot of the code including both **init** and **forward** method.

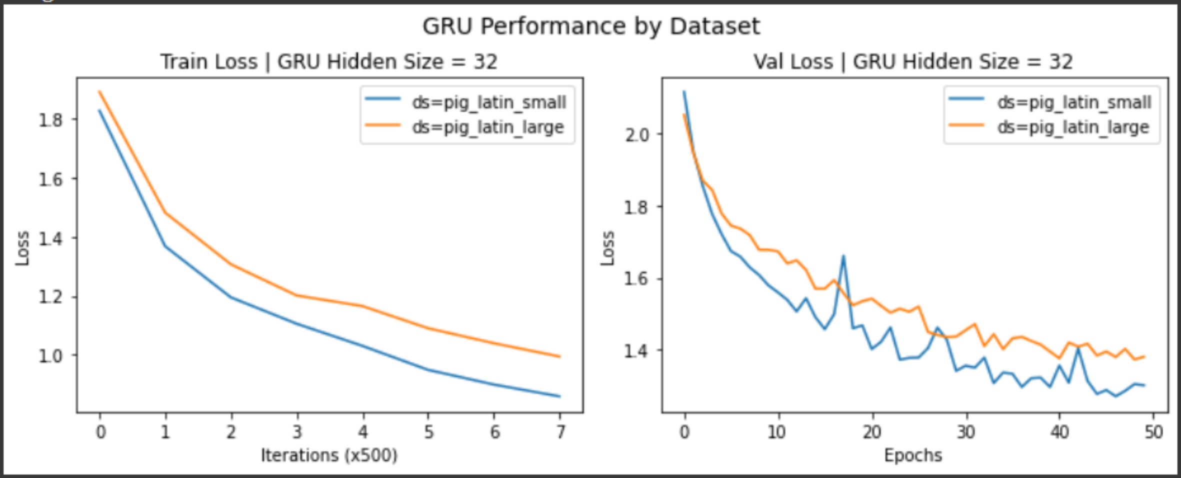


```python
class MyGRUCell(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(MyGRUCell, self).__init__()

        self.input_size = input_size
        self.hidden_size = hidden_size

        # ------------
        # 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(h_prev))
        h_new = (1 - z)*g + z * h_prev
        return h_new
```

After the models have been trained on both datasets, **pig_latin_small** and **pig_latin_large**, we got the picture (shown below) which compares the loss curves of the two models by running the **save_loss_comparison_gru** method.

```
[ ] save_loss_comparison_gru(rnn_losses_s, rnn_losses_l, rnn_args_s, rnn_args_l, "gru")
```

```
Plot saved to: /content/content/csc421/a3/loss_plot_gru.pdf
<Figure size 432x288 with 0 Axes>
```



We can see from this comparison graph that validation loss is decreasing for both datasets, with the smaller dataset performing sognificantly better (i.e. a ,lower val loss that what's of large datasets). This might because that the larger dataset casuses the model to overfit and thus lead to a higher loss.

Q2 - Identify failure modes

```
[ ] best_encoder = rnn_encode_s  # Replace with rnn_encode_s or rnn_encode_l
    best_decoder = rnn_decoder_s  # Replace with rnn_decoder_s or rnn_decoder_l
    best_args = rnn_args_s     # Replace with rnn_args_s or rnn_args_l

    TEST_SENTENCE = "i am planning to go shopping tomorrow"
    translated = translate_sentence(
        TEST_SENTENCE, best_encoder, best_decoder, None, best_args
    )
    print("source:\t\t{} \ntranslated:\t{}".format(TEST_SENTENCE, translated))
```

```
    source:         i am planning to go shopping tomorrow
    translated:     iway amway aninay-agsday ophtay onay ophosingshay omprorcay
```

The correct translation is "iway amway anningplay otay ogay oppingshay omorrowtay". We can see that longer words are harder for our model to translate. For example, "i" and "am" are translated correctly but long words like "tomorrow" failed.

Q3 - Comparing complexity

- LSTM encoder: $4 \times (H + D) \times H$ parameters
- GRU encoder: $3 \times (H + D) \times H$ parameters

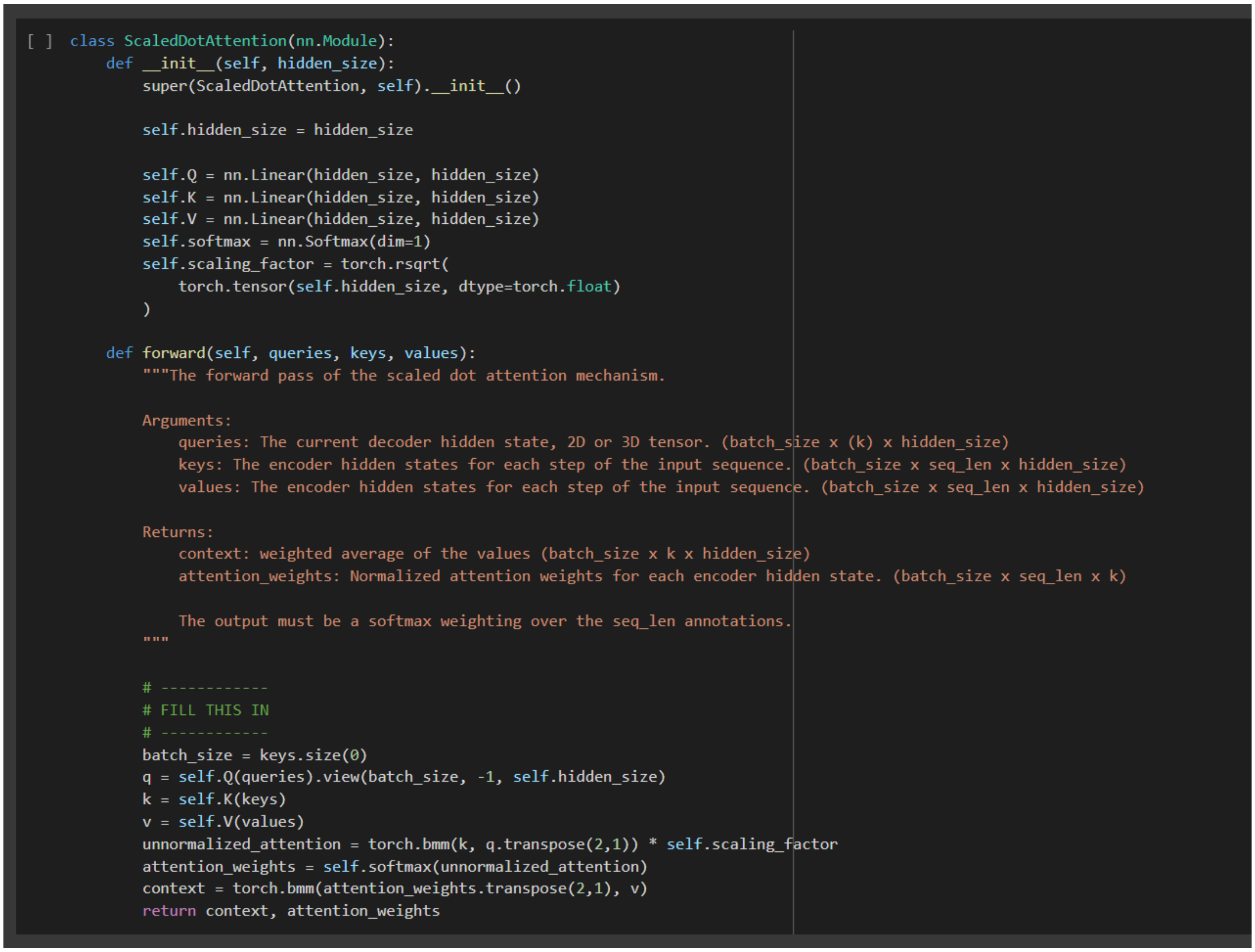
Part 2.1 : Additive Attention

Q3

It took 2m58s to run the RNN model with additive attention but only 2m13s for RNN model without additive attention. This happens because it adds more weight parameters to the model and therefore makes it a method with higher training time.

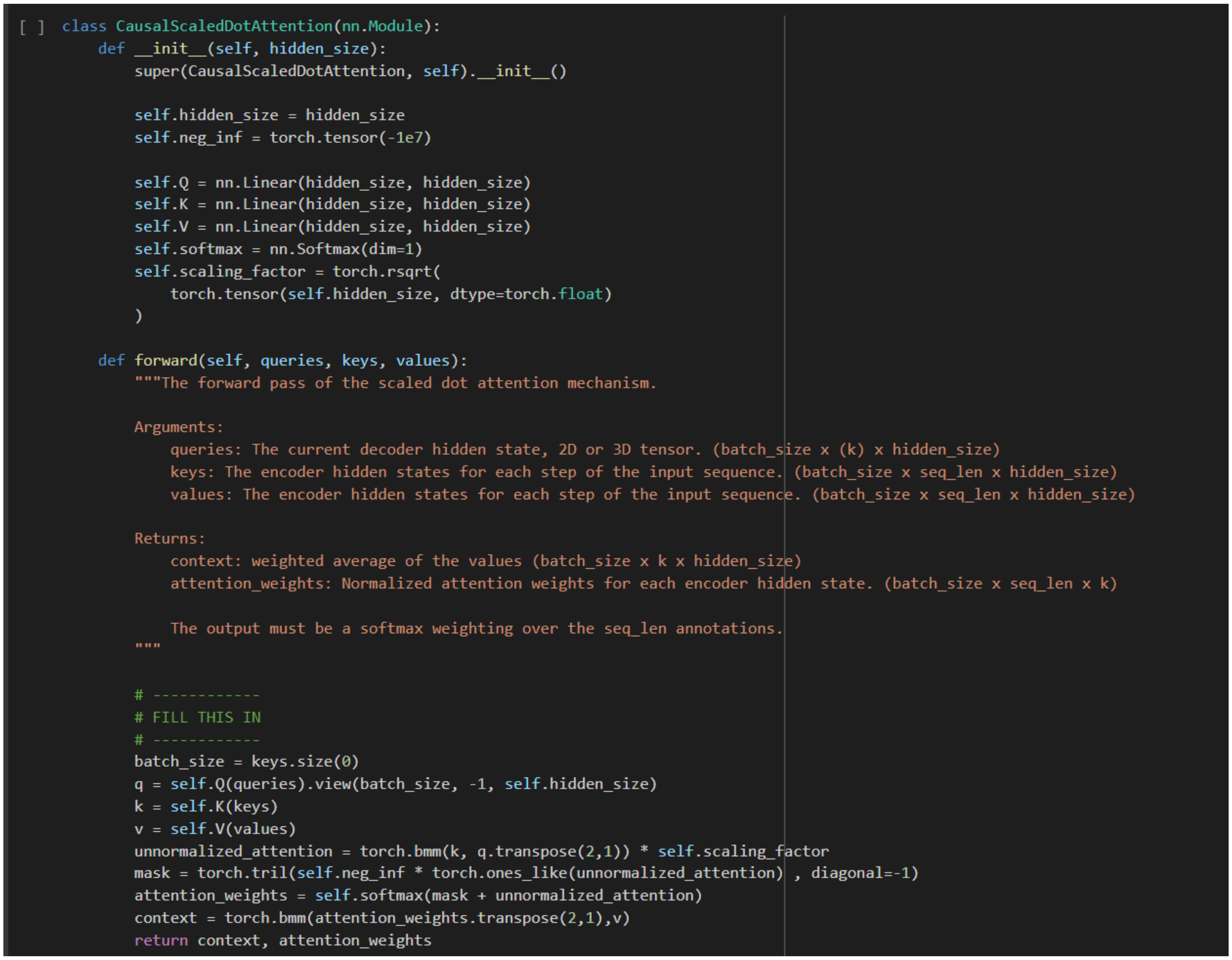
Part 2.2 : Scaled Dot Product Attention

Q1 - Implement the scaled dot-product attention mechanism



```
[ ] 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 = 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
```

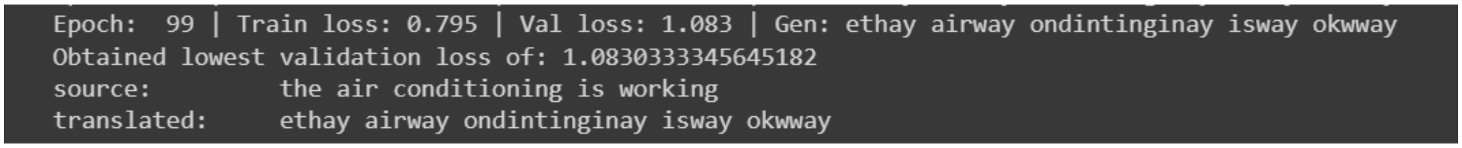
Q2 - Implement the causal scaled dot-product attention mechanism



```
[ ] 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.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(self.neg_inf * torch.ones_like(unnormalized_attention) , diagonal=-1)
            attention_weights = self.softmax(mask + unnormalized_attention)
            context = torch.bmm(attention_weights.transpose(2,1),v)
            return context, attention_weights
```

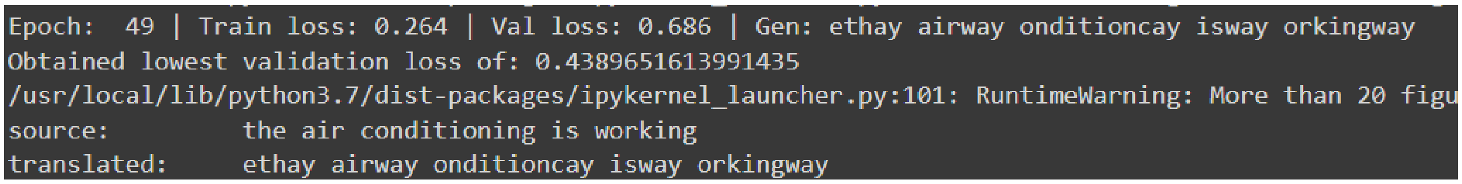
Q3

As shown in the picture below, a model using ScaledDotAttention mechanism as an encoder and decoder (i.e. single block dot-product attention model) has a lowest estimated val loss of 1.083.



```
  Epoch:  99 | Train loss: 0.795 | Val loss: 1.083 | Gen: ethay airway ondintinginay isway okwway
  Obtained lowest validation loss of: 1.0830333345645182
  source:         the air conditioning is working
  translated:     ethay airway ondintinginay isway okwway
```

Meanwhile, RNNAttention model as presented in the picture below has a lowest estimated val loss of 0.439.



```
Epoch:  49 | Train loss: 0.264 | Val loss: 0.686 | Gen: ethay airway onditioncay isway orkingway
Obtained lowest validation loss of: 0.4389651613991435
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:101: RuntimeWarning: More than 20 figu
source:         the air conditioning is working
translated:     ethay airway onditioncay isway orkingway
```

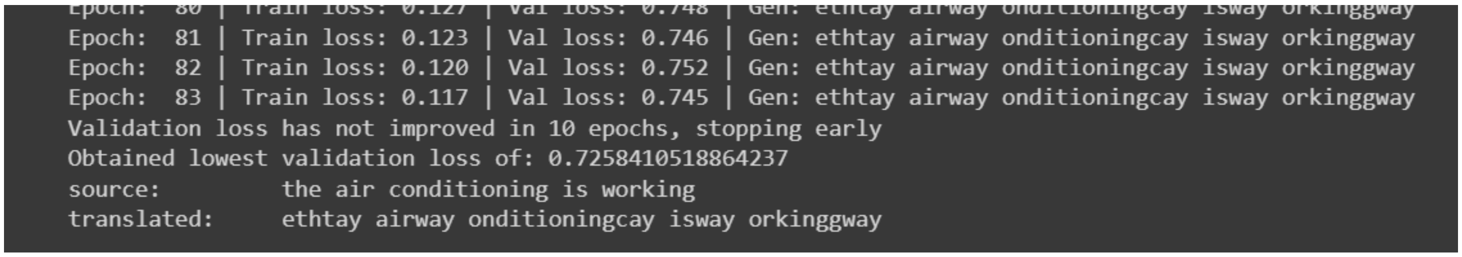
Clearly, we know that RNNAttention model performes better than the single block dot-product attention model as it has a lower val loss. This might because that RNNAttention has more parameters, which in turn, leads to a better estimation (i.e. reduced residuals) though it might result in a much slower training rate.

Q4

- Since this model is working on sequences, there must be a way where we would be able to "describe" the sequence and tokens within the sequence and its relative or absolute position. By doing so, we will be able to represent any token within any given sequence if we need.
- One-hot encoding is extremely inefficient when dealing with large volcabularies as it is designed in such way that each of its token will be a vector of a size equal to the volcabulary size. However, positional encoding has the same dimension as the embeddings, which are by definition relatively low-dimensional and therefore is more efficient (when facing large volcabularies).

Q5

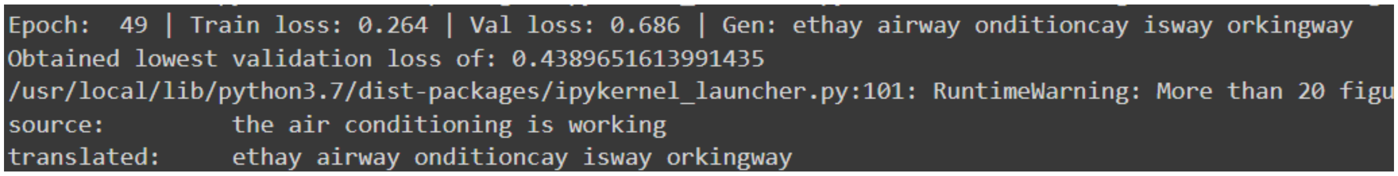
The below picture is the result I obtained from transformer:



```
Epoch:  80 | Train loss: 0.127 | Val loss: 0.748 | Gen: ethtay airway onditioningcay isway orkinggway
Epoch:  81 | Train loss: 0.123 | Val loss: 0.746 | Gen: ethtay airway onditioningcay isway orkinggway
Epoch:  82 | Train loss: 0.120 | Val loss: 0.752 | Gen: ethtay airway onditioningcay isway orkinggway
Epoch:  83 | Train loss: 0.117 | Val loss: 0.745 | Gen: ethtay airway onditioningcay isway orkinggway
Validation loss has not improved in 10 epochs, stopping early
Obtained lowest validation loss of: 0.7258410518864237
source:         the air conditioning is working
translated:     ethtay airway onditioningcay isway orkinggway
```

The correct translation should be: ethay airway onditioningcay isway orkingway

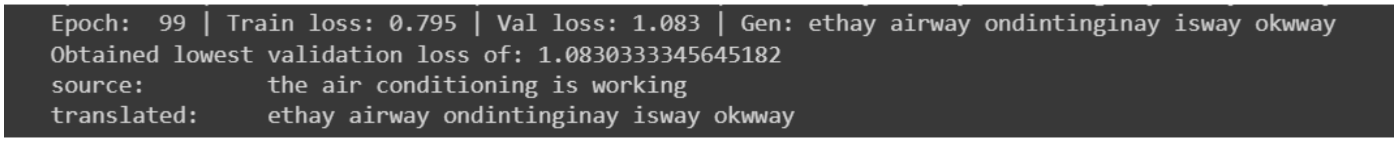
- **Compare to RNNAttention decoder:**
  (The below picture is the result I obtained from RNNAttention)



```
Epoch:  49 | Train loss: 0.264 | Val loss: 0.686 | Gen: ethay airway onditioncay isway orkingway
Obtained lowest validation loss of: 0.4389651613991435
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:101: RuntimeWarning: More than 20 figu
source:         the air conditioning is working
translated:     ethay airway onditioncay isway orkingway
```

  Notice that both RNNAttention decoder and transformer made minor mistake while trying to translate this sentence. However, RNNAttention decoder generates a lower loss than transformer.

- **Compare to single-block attention decoder:**
  (The below picture is the result I obtained from single-block attention decoder)

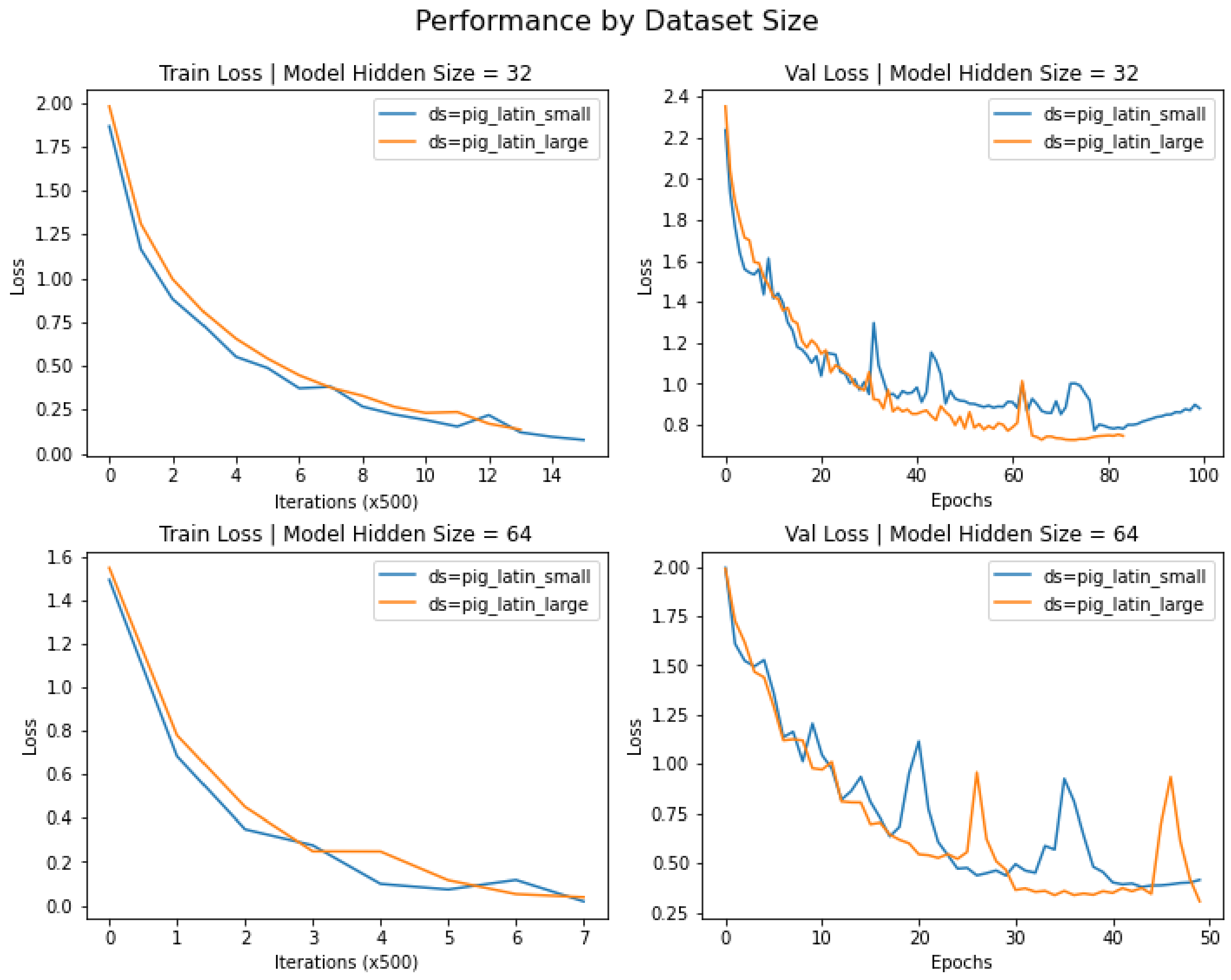


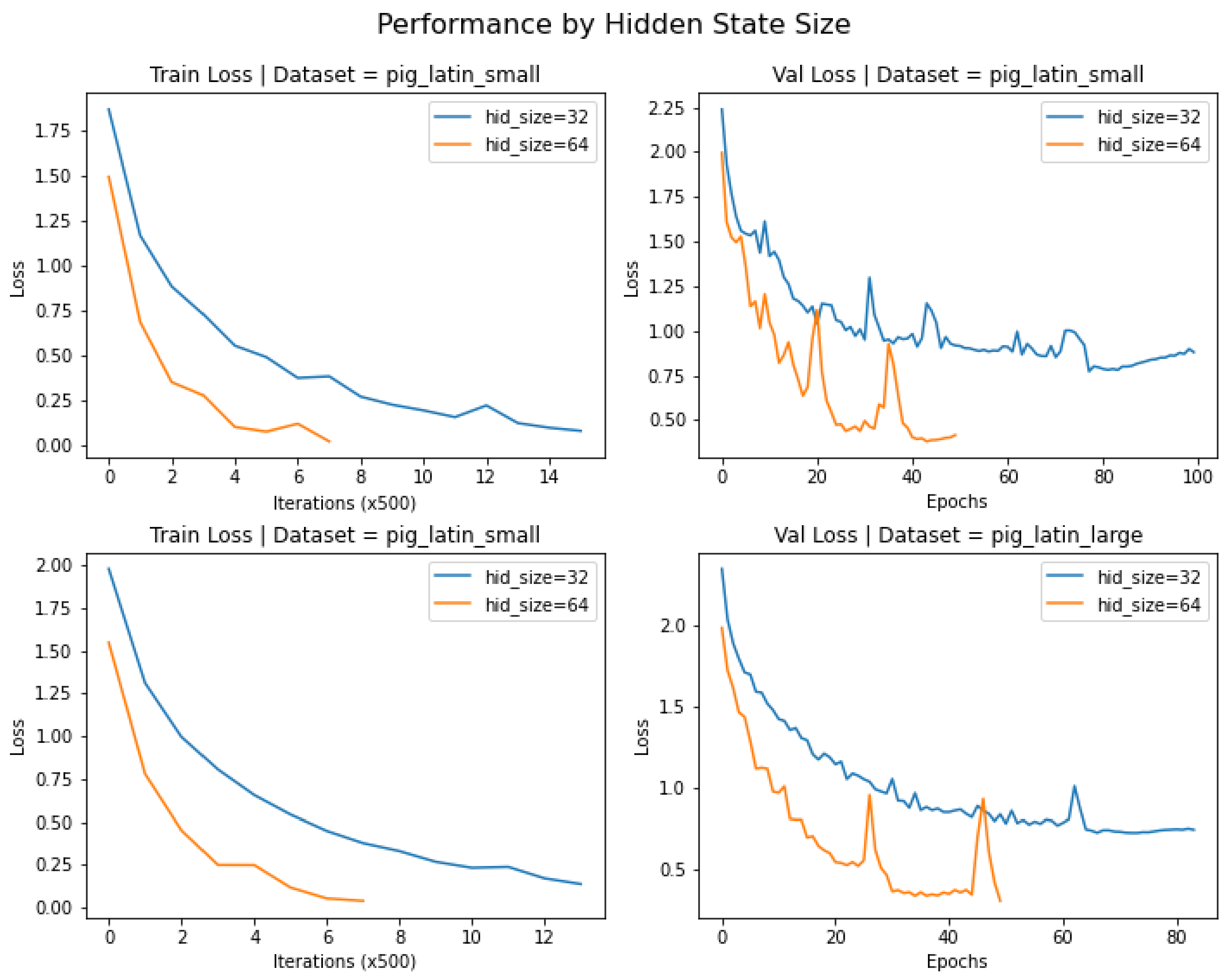
```
Epoch:  99 | Train loss: 0.795 | Val loss: 1.083 | Gen: ethay airway ondintinginay isway okwway
Obtained lowest validation loss of: 1.0830333345645182
source:         the air conditioning is working
translated:     ethay airway ondintinginay isway okwway
```

  Notice that both single-block attention decoder and transformer made mistakes while trying to translate this sentence, though single-block attention decoder made larger mistakes (should be *orkingway* instead of *okwway* and *onditioningcay* instead of *ondintinginay*) than the minor ones

RNN attention made (should be *orkingway* instead of *orkinggway*). Single-block attention decoder also has a larger loss than whats of transformer model.

Q6





We run the Transformer model using hidden size 32 vs. 64, and using the small vs. large dataset (in total, 4 runs). Below table illustrates what we got from each run.

	pig_latin_small	pig_latin_large
hidden size = 32	0.9501068491670417	0.7258410518864237
hidden size = 64	0.3802134682160072	0.30701605956918293

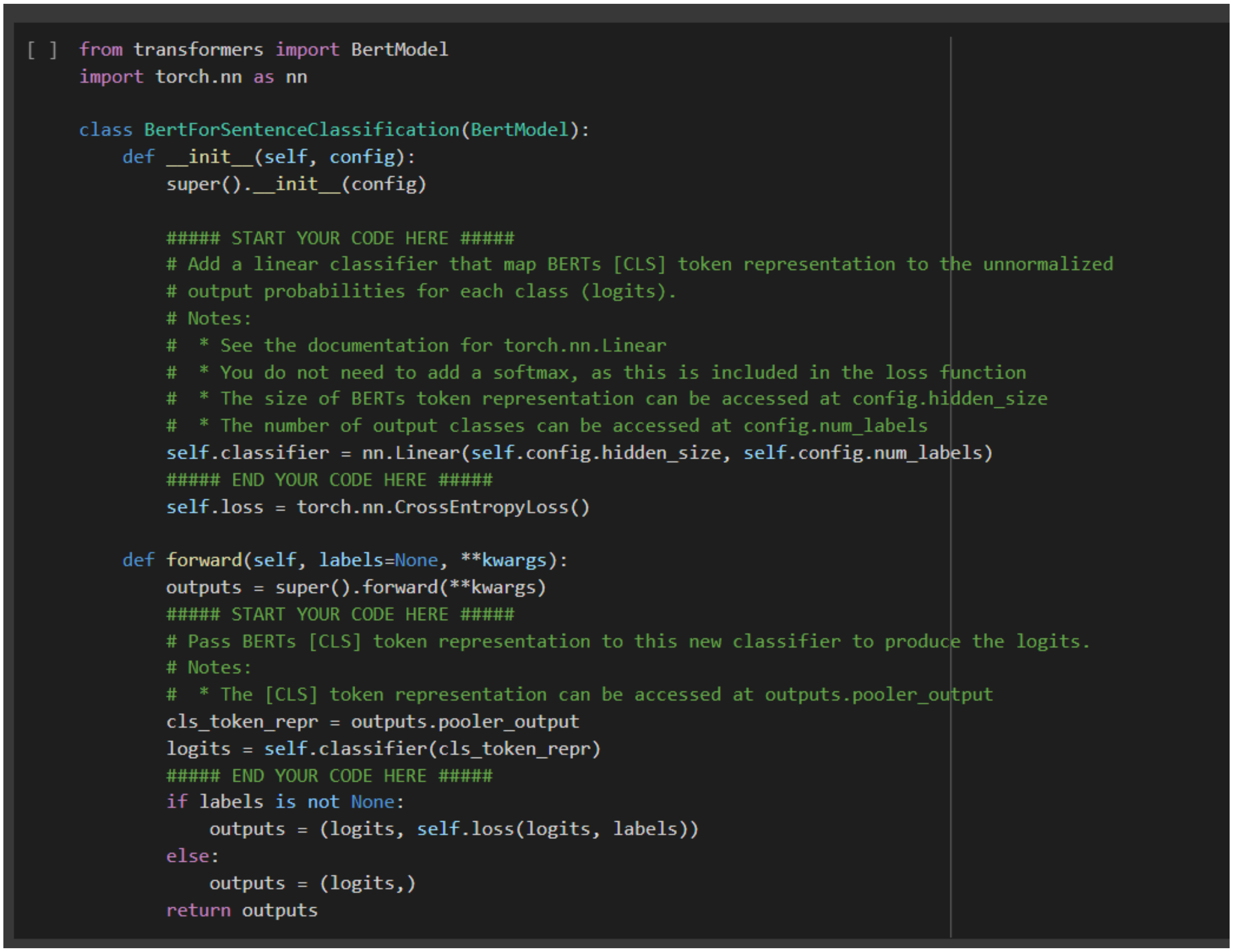
By this table, we know that when fixing the dataset, increase in hidden size will decrease the validation loss but will take longer, vice versa. And, when fixing the hidden size, increase in data size will decrease the validation loss (but the training takes a longer time) and thus lead to a better performance. vice versa.

These results meet my expectation perfectly. My reasoning is as follows:

- when fixing the dataset, increase in hidden size will increase number of parameters thus increase the training time. This increase in parameter also decreases the loss as it gives a better generalization. Vice versa.
- when fixing the hidden size, increase in data size will give the more data to learn insight from and therefore will help with generalization and decrease validation loss. This will, however, also increase the training time.

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

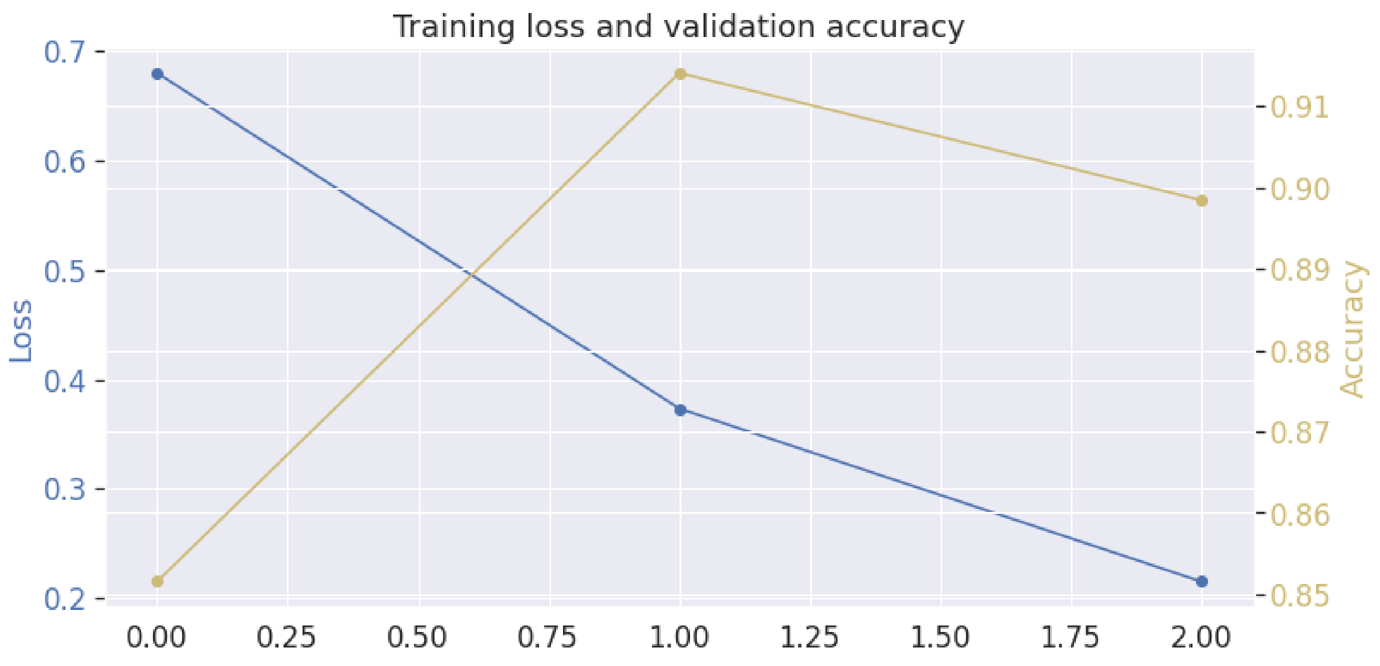
Q1 - Add a classifier to BERT

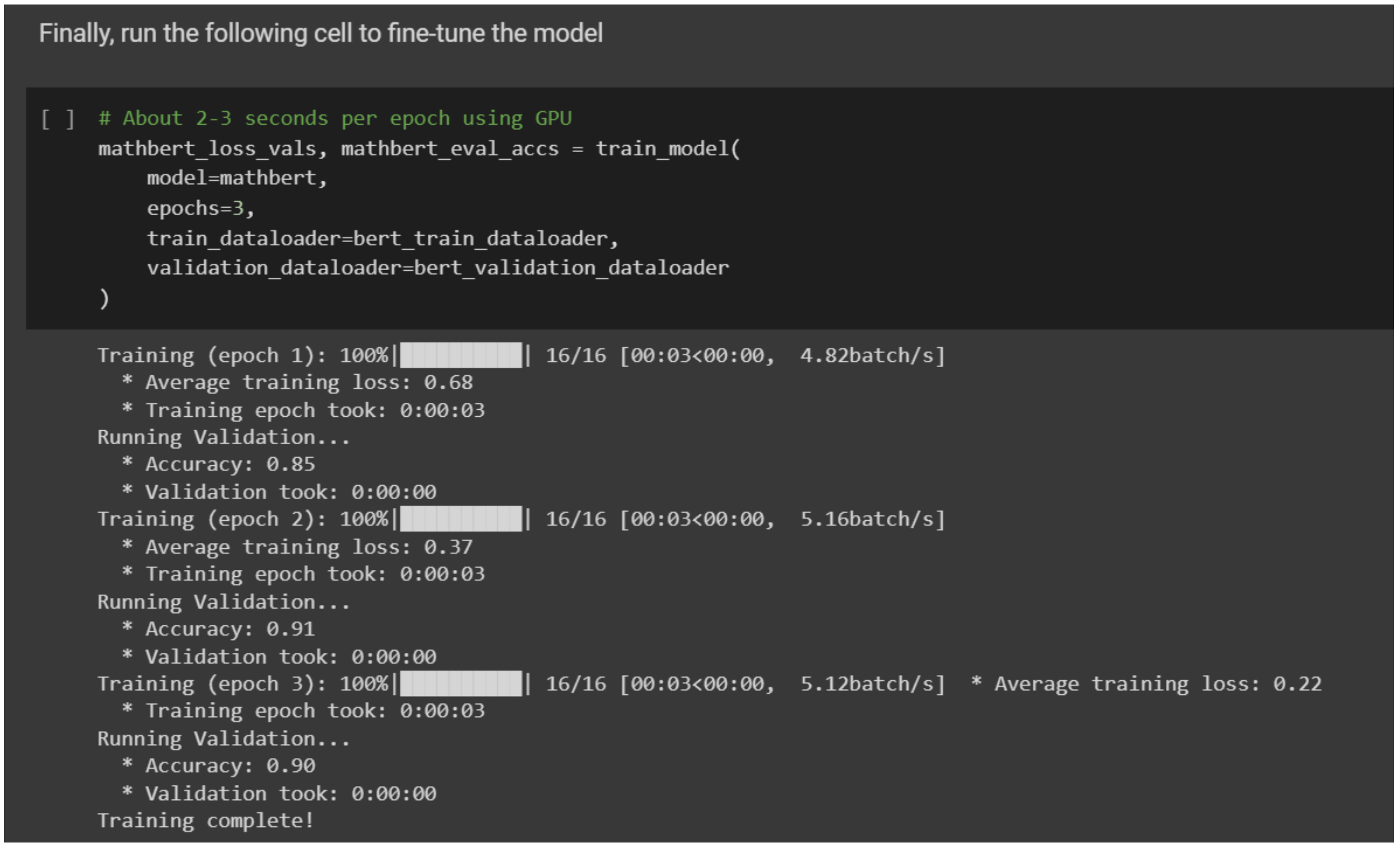


```
[ ]  from transformers import BertModel
     import torch.nn as nn

     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(self.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

- The results generated by fine-tuning

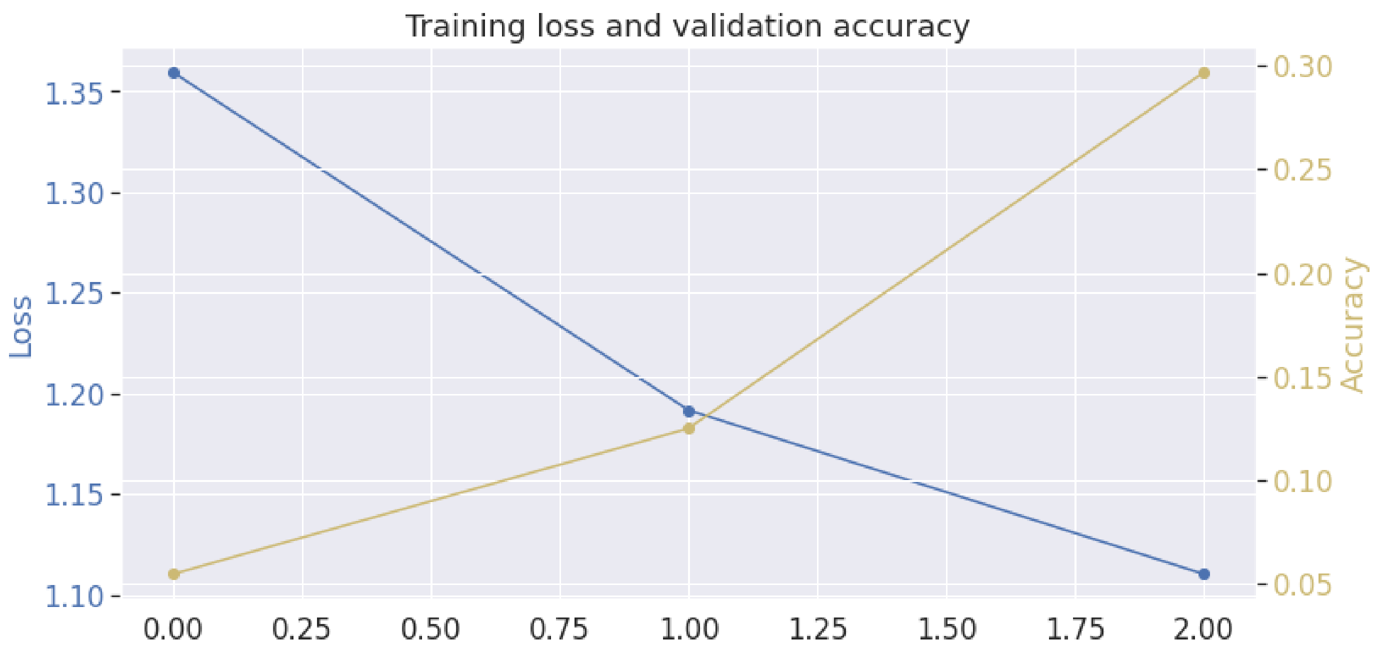



Finally, run the following cell to fine-tune the model

```
[ ]  # About 2-3 seconds per epoch using GPU
     mathbert_loss_vals, mathbert_eval_accs = train_model(
         model=mathbert,
         epochs=3,
         train_dataloader=bert_train_dataloader,
         validation_dataloader=bert_validation_dataloader
     )
```

```
Training (epoch 1): 100%|██████████| 16/16 [00:03<00:00,  4.82batch/s]
  * Average training loss: 0.68
  * Training epoch took: 0:00:03
Running Validation...
  * Accuracy: 0.85
  * Validation took: 0:00:00
Training (epoch 2): 100%|██████████| 16/16 [00:03<00:00,  5.16batch/s]
  * Average training loss: 0.37
  * Training epoch took: 0:00:03
Running Validation...
  * Accuracy: 0.91
  * Validation took: 0:00:00
Training (epoch 3): 100%|██████████| 16/16 [00:03<00:00,  5.12batch/s]  * Average training loss: 0.22
  * Training epoch took: 0:00:03
Running Validation...
  * Accuracy: 0.90
  * Validation took: 0:00:00
Training complete!
```

- The results generated by fine-tuning only the classifiers weights, leavng BERT's weight frozen:



```
[ ]  # About 1 second per epoch on GPU
     mathbert_frozen_loss_vals, mathbert_frozen_eval_accs = train_model(
         model=mathbert_frozen,
         epochs=3,
         train_dataloader=bert_train_dataloader,
         validation_dataloader=bert_validation_dataloader
     )
```

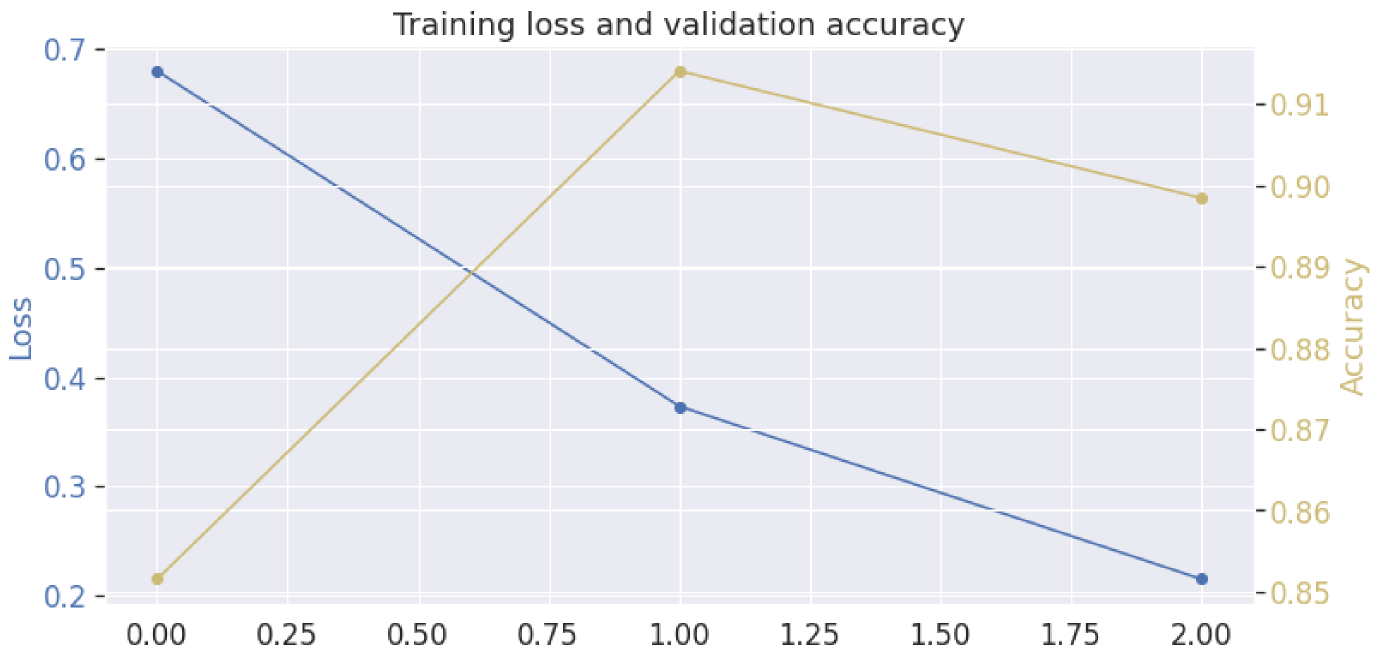
```
Training (epoch 1): 100%|██████████| 16/16 [00:00<00:00, 17.04batch/s]
  * Average training loss: 1.36
  * Training epoch took: 0:00:01
Running Validation...
  * Accuracy: 0.05
  * Validation took: 0:00:00
Training (epoch 2): 100%|██████████| 16/16 [00:00<00:00, 18.04batch/s]
  * Average training loss: 1.19
  * Training epoch took: 0:00:01
Running Validation...
  * Accuracy: 0.12
  * Validation took: 0:00:00
Training (epoch 3): 100%|██████████| 16/16 [00:00<00:00, 18.36batch/s]
  * Average training loss: 1.11
  * Training epoch took: 0:00:01
Running Validation...
  * Accuracy: 0.30
  * Validation took: 0:00:00
Training complete!
```

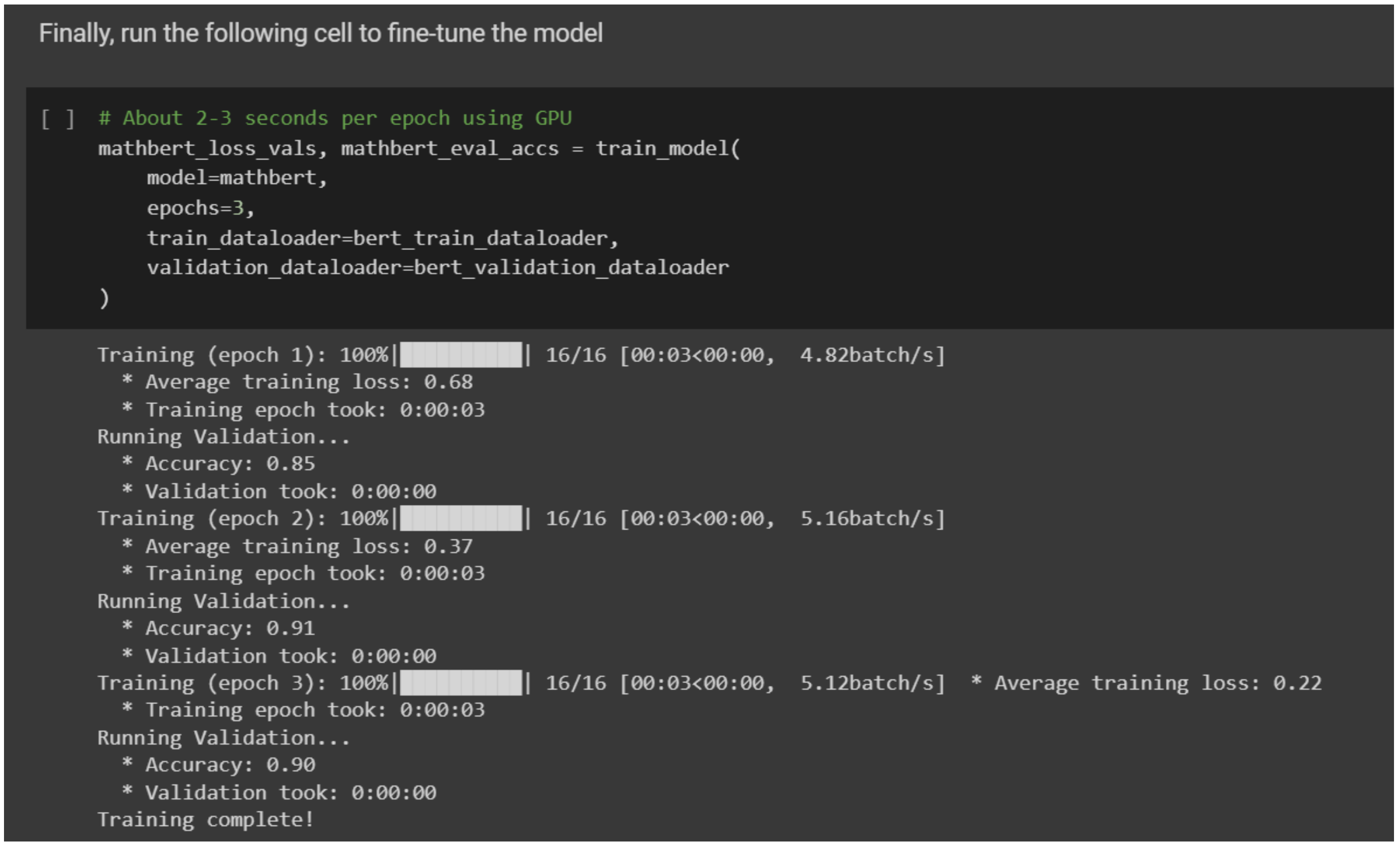
(1) Compared to fine-tuning, the **train time** is faster when BERTs weights are frozen. This might because when BERTs weights are frozen, we are only training the parameters of the classifiers rather than fine-tune both ther classifer and BERT model jointly. When number of parameters decreases, training time decreases as well.

(2) Compared to fine-tuning, the model **perform** worse when BERTs weights are frozen. This might also because of the mentioned reason. Fine-tuning has more parameters involved and therefore have a lower validation loss and thus performs better when BERTs weights are NOT frozen.

Q4 - Effect of pretraining data.

- The results generated by fine-tuning BERT with the pretrained weights from MathBERT

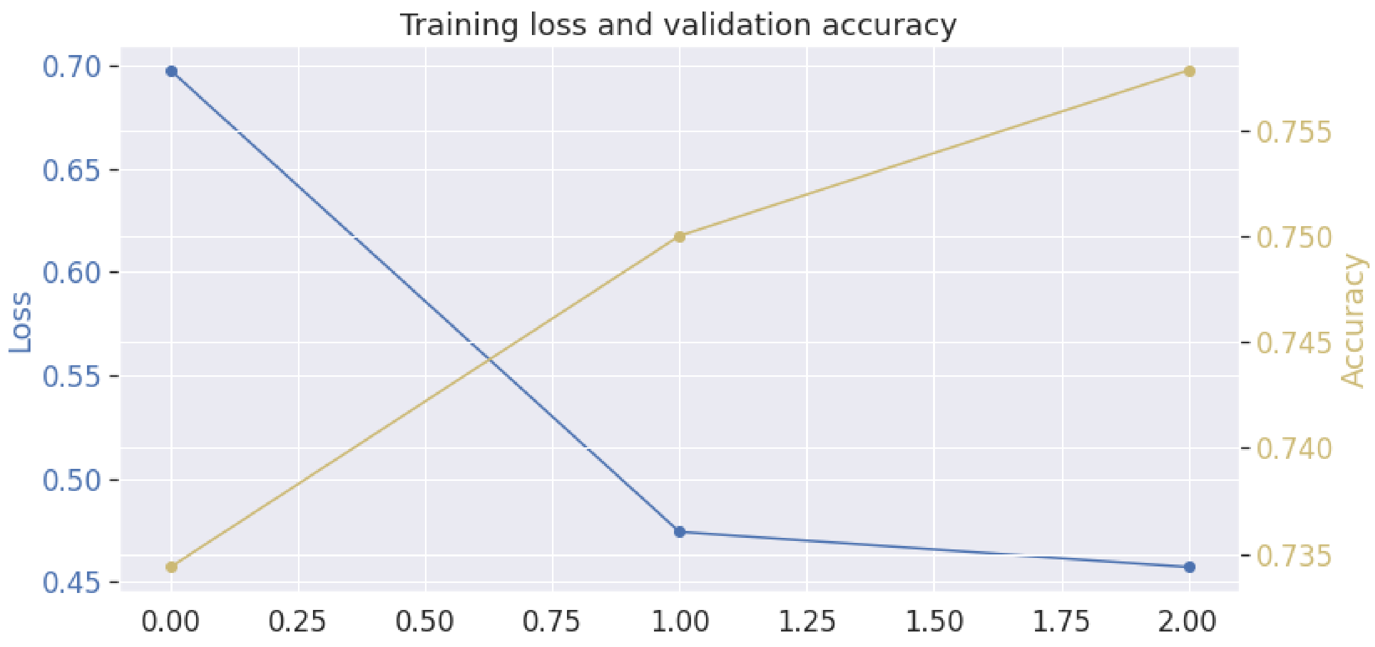


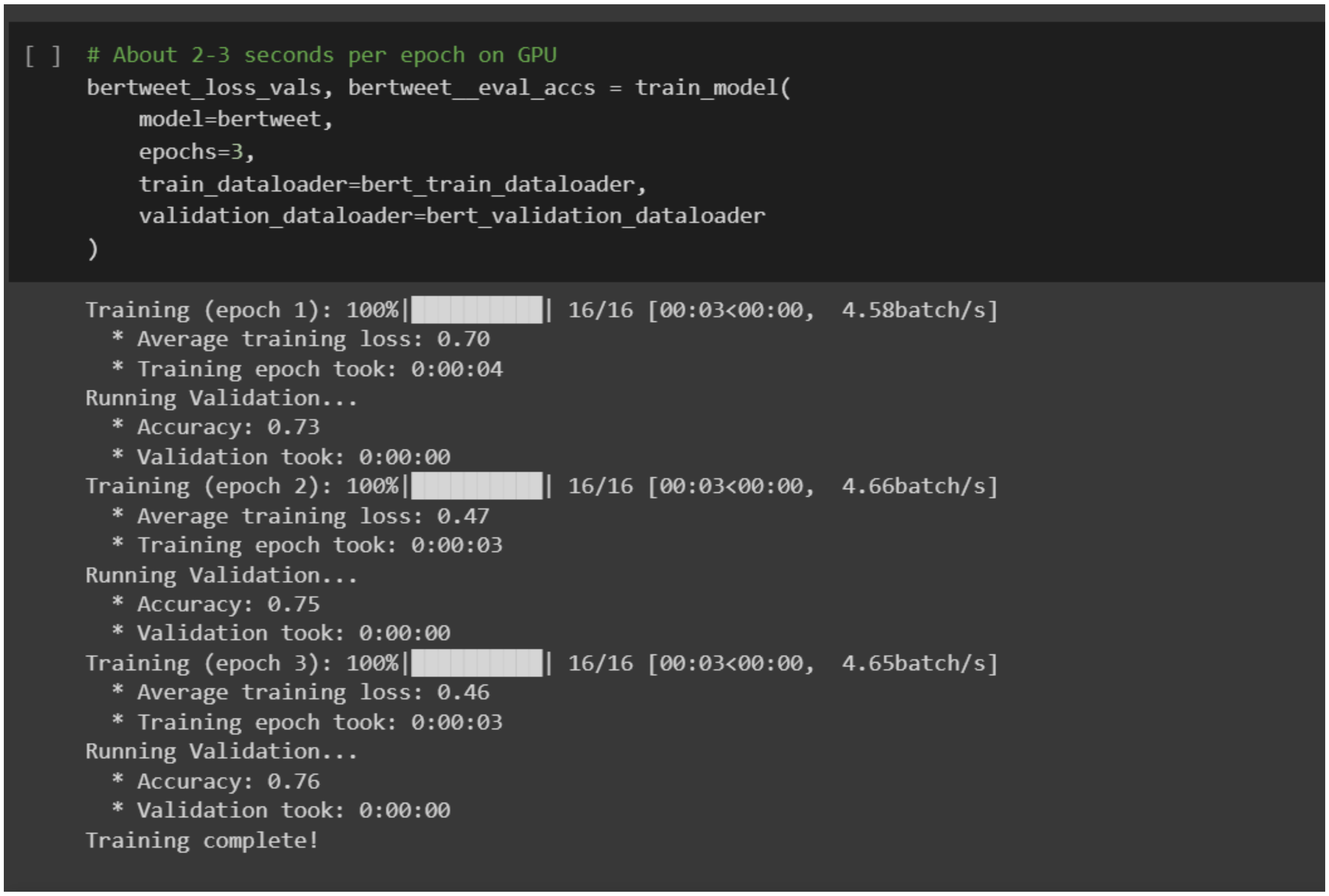


Finally, run the following cell to fine-tune the model

```
[ ] # About 2-3 seconds per epoch using GPU
    mathbert_loss_vals, mathbert_eval_accs = train_model(
        model=mathbert,
        epochs=3,
        train_dataloader=bert_train_dataloader,
        validation_dataloader=bert_validation_dataloader
    )

    Training (epoch 1): 100%|██████████| 16/16 [00:03<00:00,  4.82batch/s]
      * Average training loss: 0.68
      * Training epoch took: 0:00:03
    Running Validation...
      * Accuracy: 0.85
      * Validation took: 0:00:00
    Training (epoch 2): 100%|██████████| 16/16 [00:03<00:00,  5.16batch/s]
      * Average training loss: 0.37
      * Training epoch took: 0:00:03
    Running Validation...
      * Accuracy: 0.91
      * Validation took: 0:00:00
    Training (epoch 3): 100%|██████████| 16/16 [00:03<00:00,  5.12batch/s]  * Average training loss: 0.22
      * Training epoch took: 0:00:03
    Running Validation...
      * Accuracy: 0.90
      * Validation took: 0:00:00
    Training complete!
```

- The results generated by fine-tuning BERT with the pretrained weights from BERTweets





```
[ ] # About 2-3 seconds per epoch on GPU
    bertweet_loss_vals, bertweet__eval_accs = train_model(
        model=bertweet,
        epochs=3,
        train_dataloader=bert_train_dataloader,
        validation_dataloader=bert_validation_dataloader
    )
```

```
Training (epoch 1): 100%|██████████| 16/16 [00:03<00:00,  4.58batch/s]
  * Average training loss: 0.70
  * Training epoch took: 0:00:04
Running Validation...
  * Accuracy: 0.73
  * Validation took: 0:00:00
Training (epoch 2): 100%|██████████| 16/16 [00:03<00:00,  4.66batch/s]
  * Average training loss: 0.47
  * Training epoch took: 0:00:03
Running Validation...
  * Accuracy: 0.75
  * Validation took: 0:00:00
Training (epoch 3): 100%|██████████| 16/16 [00:03<00:00,  4.65batch/s]
  * Average training loss: 0.46
  * Training epoch took: 0:00:03
Running Validation...
  * Accuracy: 0.76
  * Validation took: 0:00:00
Training complete!
```

Compared to fine-tuning BERT with the pretrained weights from MathBERT, the model perform worse when we fine-tune BERT with the pretrained weights from BERTweets as $0.76 < 0.90$. This is because the dataset we are using is a verbal arithmetic dataset and the task the model is asked to do is to do math. Words like "minus" or "add" which are so simple and clear in MathBERT but since these words are not commonly used words in tweets, it might add noise to the training process. For example, MathBERT can easily know that add means arithmetic addition, wheras BERTweets might interpret it as short for "advertising", which will then impact the performance negatively.

Part 4 : Connecting Text and Images with CLIP

Q2 - Prompting CLIP

The caption I used is : "an orange butterfly on a purple flower". The process of finding the caption is easy. All we need to do is to give accurate descriptions to the objects' name, color and relative position.