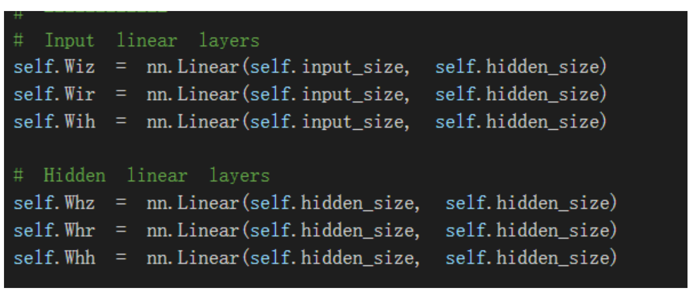
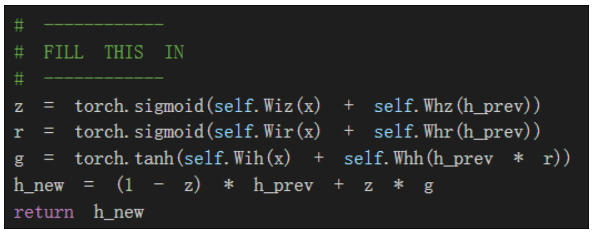
Part 1: Neural Machine Translation(NMT)

Question 1:
code left is __init__, right is forward()



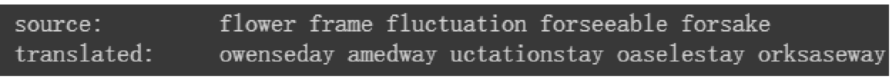
```
# ------------
# Input linear layers
self.Wiz = nn.Linear(self.input_size, self.hidden_size)
self.Wir = nn.Linear(self.input_size, self.hidden_size)
self.Wih = nn.Linear(self.input_size, self.hidden_size)

# Hidden linear layers
self.Whz = nn.Linear(self.hidden_size, self.hidden_size)
self.Whr = nn.Linear(self.hidden_size, self.hidden_size)
self.Whh = nn.Linear(self.hidden_size, self.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 * r))
h_new = (1 - z) * h_prev + z * g
return h_new
```

After running the model on both small and large datasets, the model trained with large dataset performs better than the other with a significant difference of 0.2 validation loss.
Probably because large datasets covers more possible corner cases for translation, which makes training more stable.

Question 2:



```
source:         flower frame fluctuation forseeable forsake
translated:     owenseday amedway uctationstay oaselestay orksaseway
```

All words starting with letter “f” won’t be translated correctly, as “f” won’t appear in final translation.

Question 3:

Total # of parameters for LSTM encoder: $4*(H^2+HD)$

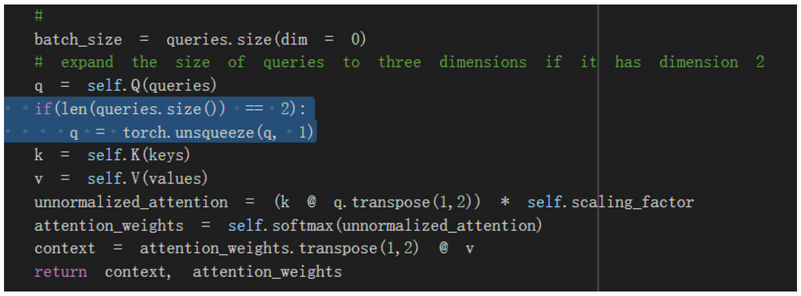
Total # of parameters for GRU encoder: $3*(H^2+HD)$

Additive attention:

Question 3:
Model with attention runs slower. That’s perhaps due to the parameters for calculating attention also requires time to train.

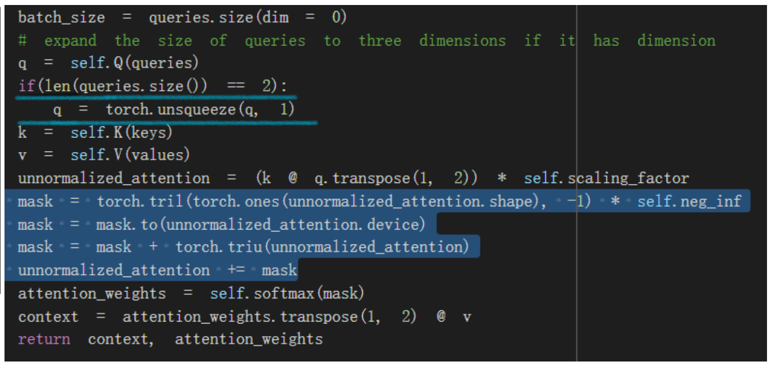
Scaled Dot Product Attention

Scaled-dot-product-implementation



```
#
batch_size  =  queries.size(dim  =  0)
#  expand  the  size  of  queries  to  three  dimensions  if  it  has  dimension  2
q  =  self.Q(queries)
if(len(queries.size())  ==  2):
    q  =  torch.unsqueeze(q,  1)
k  =  self.K(keys)
v  =  self.V(values)
unnormalized_attention  =  (k  @  q.transpose(1,2))  *  self.scaling_factor
attention_weights  =  self.softmax(unnormalized_attention)
context  =  attention_weights.transpose(1,2)  @  v
return  context,  attention_weights
```

Casual scaled-dot-product-implementation



```
batch_size  =  queries.size(dim  =  0)
#  expand  the  size  of  queries  to  three  dimensions  if  it  has  dimension
q  =  self.Q(queries)
if(len(queries.size())  ==  2):
    q  =  torch.unsqueeze(q,  1)
k  =  self.K(keys)
v  =  self.V(values)
unnormalized_attention  =  (k  @  q.transpose(1,  2))  *  self.scaling_factor
mask  =  torch.tril(torch.ones(unnormalized_attention.shape),  -1)  *  self.neg_inf
mask  =  mask.to(unnormalized_attention.device)
mask  =  mask  +  torch.triu(unnormalized_attention)
unnormalized_attention  +=  mask
attention_weights  =  self.softmax(mask)
context  =  attention_weights.transpose(1,  2)  @  v
return  context,  attention_weights
```

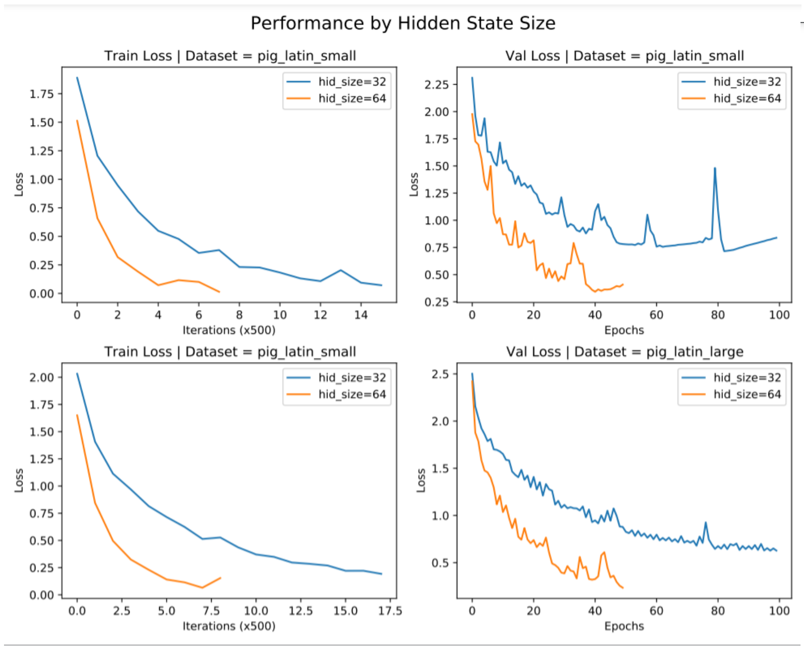
Question 3:
Compared with RNN + Attention model, single-attention model performs quite terrible. Possibly due to lacking of recurrence, which reduced the connection/association between previous predictions and future predictions.

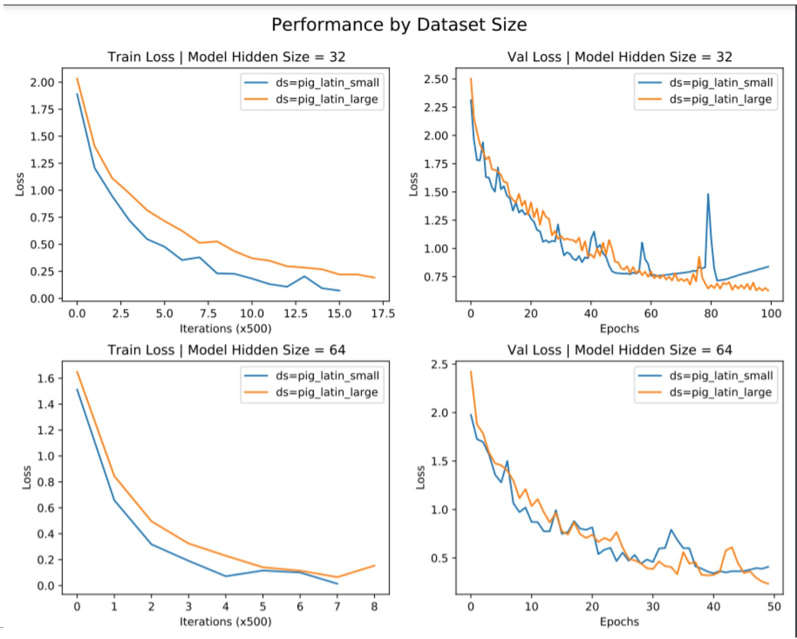
Question 4:
Positional encoding determines the overall meaning of sentence about to be translated. It's better than other encodings as sine and cosine function gives the relative position of what the model should attend to when predicting the next output.

Question 5:
RNNAttention:
Compared with RNNAttention model, transformer still performs poorly by looking at final validation loss. The final test-translation still have details missing; for example, my training shows letter "i" in "air" doesn't show up in final translation.

Single-layer Attention:
Transformer performs better than single-layer attention model, as the final validation loss is significantly lower. The translation also contains more details, with dord's structure retained.

Question 6:





Training result (final best validation loss):
Original setting: 0.715471369586885
Increased dataset size: 0.6271883726728206
Increased hidden size: 0.3420589083340019
Increased hidden size & increased dataset size: 0.2342876510466544

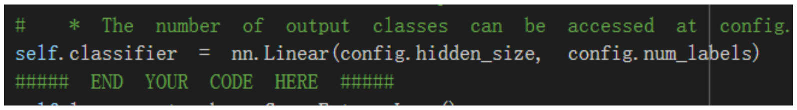
Based on above observations of training loss and final validation loss, both increasing dataset size and increasing hidden size can improve model performance.

By viewing loss as a function of gradient descent iterations, increased batch size leads to a significant reduction of both training loss and validation loss with a shorter time compared to original batch size.
On the other hand, larger datasets makes fluctuation of model training loss and validation loss less violent, and leads to a better generalization.

These results matches with effect of improving sample size and increasing hidden size.

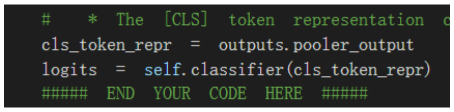
Part 3:

Code implementation inside __init__:



```
#   * The number of output classes can be accessed at config.
self.classifier = nn.Linear(config.hidden_size, config.num_labels)
##### END YOUR CODE HERE #####
```

code implementation inside forward():



```
#   * The [CLS] token representation c
cls_token_repr = outputs.pooler_output
logits = self.classifier(cls_token_repr)
##### END YOUR CODE HERE #####
```

Question 3:
The training time has dramatically reduced, as most parameters doesn't require tuning.

However leaving BERT weight untuned makes BERT model not adapted onto new scenarios, leading to significantly poor performance (accuracy dropped to 0.30 from 0.9)

Question 4:
The performance is still not as good as expected (0.7-0.8, compared to 0.9 by tuned mathBERT). The reason is due to daily-life tweets doesn't involve too much arithmetic scenarios, thus dataset of BERTweets is also affected. As BERTweets is not trained for arithmetic tasks specifically, decrease in prediction accuracy is expected.

Part 4:

Question 2:
Caption: butterfly above flower not red

My search process first found a bee on flower, then got stuck on a butterfly on side of a red flower until I put "not red" at end of sentence.