Part 1: Neural machine translation (NMT)

1.
__init__

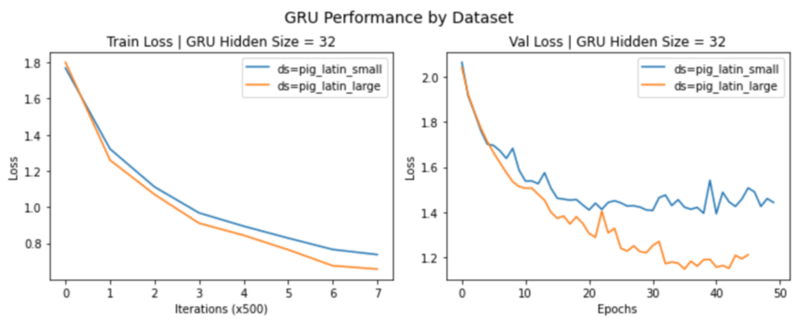
```
# ------------
# FILL THIS IN
# ------------
# Input linear layers
self.Wiz = nn.Linear(input_size, hidden_size)
self.Wir = nn.Linear(input_size, hidden_size)
self.Wih = nn.Linear(input_size, hidden_size)


# 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)
```

Forward based on handout v1.1

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

Compare the loss curves:



Based on validations loss plots, the model trained on big dataset has a lower validation loss and as a result better generalization. The big dataset contains more unique words, it gives model more opportunity to learn so it performs better, with less overfit.

2.
Example1:

```
source:        exceptional flavor
translated:    expcerpay-acheswаy avoway
```

The model is not working well on large words. It doesn't keep history well and gets lost at the middle.

Example2:

```
ource:          onomatopoeia is hard
translated:     omponationcay isway adway
```

rare words that are not seen by model. like "onomatopoeia" translates to a word that is not even close enough to original one.

3.
LSTM -> 4(H*DK) + 4(H*H)
GRU -> 3(H*DK) + 3(H*H)

Part 2.1: Additive Attention
3.
I printed train time and for model with additive attention (with small dataset) it's 223 sec, but for vanilla model (with small dataset) it's 185 sec. The training time increases because with attention decoder computes an attention weight for each encoder hidden state, which is extra step in comparison with previous vanilla model.

Part 2.2: Scaled Dot Product Attention
ScaledDotProduct

```python
# ------------
# FILL THIS IN
# ------------
batch_size = keys.size(0)

q = self.Q(queries).view(batch_size, -1, self.hidden_size)
k = self.K(keys).view(batch_size, -1, self.hidden_size)
v = self.V(values).view(batch_size, -1, self.hidden_size)

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
```

CasualScaledDotProduct

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

q = self.Q(queries).view(batch_size, -1, self.hidden_size)
k = self.K(keys).view(batch_size, -1, self.hidden_size)
v = self.V(values).view(batch_size, -1, self.hidden_size)

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
```

3.
RNNAttention model performed better than the encoder decoder model with ScaledDotAttention. Lowest validation low for first one is 0.34 but for the second one is 1.23. For this problem and dataset additive attention is more suitable than scaled dot attention.

4.
Positional encoding represents the relative position of tokens in sentences which affects the semantic meaning very much. In transformers it's essential since we don't have it by default there. Using this method (eq 11, 12) encodings gives better generalization and handles longer sequences better. One hot encoding is not effective since it requires large hidden dimension unlike eq 11, 12 method.

5.

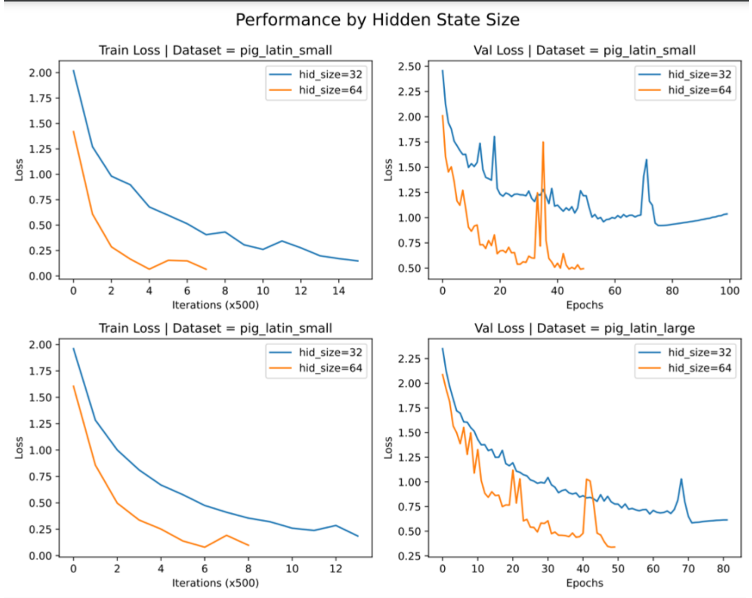
Encoder type: rnn Decoder type: rnn_attention (additive) (RNNAttentionDecoder)	`lowest validation loss of: 0.341950689686644`
Encoder type: attention (AttentionEncoder) Decoder type: attention (AttentionDecoder)	`lowest validation loss of: 1.2309618010753538`
Encoder type: transformer (TransformerEncoder) Decoder type: transformer (TransformerDecoder)	`lowest validation loss of: 0.9215109746267156`

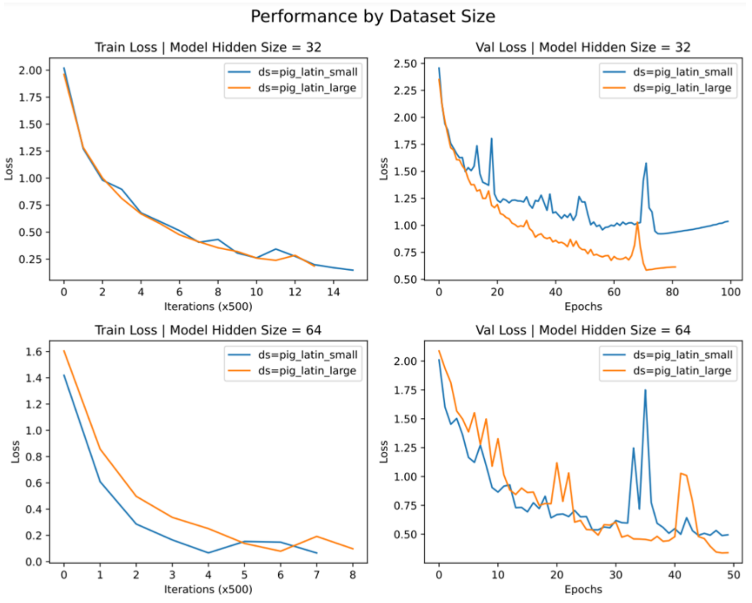
Based on the results stated in the table, the best decoder for this problem and dataset is RNNAttentionDecoder with additive attention (0.34). It performs better than scaled dot attention. TransformerDecoder is better than AttentionDecoder but still worst that RNNAttentionDecoder. My guess is that transformer needs more trainings to reach that loss, since based on my observation it was still improving at the end of training.

6.

	Small dataset	Large dataset
Hidden size 32	`lowest validation loss of: 0.9215109746267156`	`lowest validation loss of: 0.584720823421793270`
Hidden size 64	`lowest validation loss of: 0.4886014160768288`	`lowest validation loss of: 0.3430892709150629`

Based on the table and plots, we see that increasing the hidden size and the dataset size both help with improving validation loss and generalization. They both give model more opportunity to learn more complicated patterns.

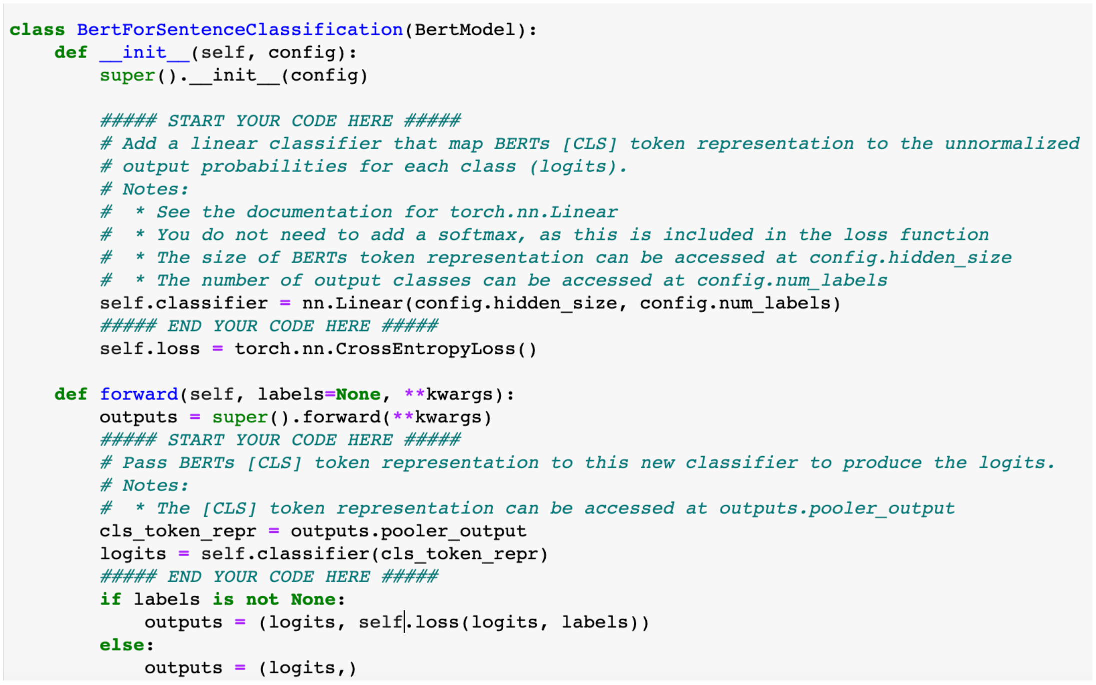




Part 3: Fine-Tuning Pretrained Language Models (LMs)

1.

BertForSentenceClassification



```
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,)
```

3.

MathBERT	Training epoch too k: 0:00:01	Average training l oss: 0.22	Validation Accura cy: 0.89
MathBERT frozen	Training epoch too k: 0:00:00	Average training l oss: 1.11	Validation Accura cy: 0.30
BERTweet	raining epoch took : 0:00:01	Average training l oss: 0.45	Validation Accura cy: 0.74

In frozen model we leave some weights frozen out of training process as a result the training time becomes shorter because the frozen weights don’t need to be updated during training. The table shows the same.

In frozen model the accuracy is lower since the frozen weight are not trained for our specific problem and we’re using the weights that are optimized for another dataset.

4.

The loss of BERTweet is more than mathBERT and the accuracy is lower (based on table). The reason is that Verbal Arithmetic Dataset is closer to mathBERT than BERTweet so the pretrained weights are more usable in this case.

Part 4: Connecting Text and Images with CLIP

My chosen caption: “a brown butterfly on purple flower”
It was an easy process and my first try. Just tried to describe objects with their color in the image.