AI6126 DEEP NEURAL NETWORK FOR NATURAL LANGUAGE PROCESSING

ASSIGNMENT 2: SEQ2SEQ MODEL FOR MACHINE TRANSLATION

School of Computer Science and Engineering

Programme: MSAI

Date: 23 April 2024

Authored By: Tan Jie Heng Alfred (G2304193L)

Contents

1. Introduction

1.1 Machine Translation

Machine translation refers to the task of translating a text (or speech) from one language to another. Usually, a machine translation model takes a sequence of input from a source language and output a sequence in a target language, such that both sequences are semantically equivalent. As such, these models are usually a sequence-to-sequence (seq2seq) model. In this project, we experimented with the use of various seq2seq models, such as Gated Recurrent Unit (GRU), Long Short-Term Memory (LSTM) network, bi-directional LSTM (biLSTM) and Transformer, as encoders and decoders for a machine translation task of translating from French sentences to English sentences. The encoder processes the input French sentence, while the decoder processes the encoded information from the encoder and output a corresponding English sentence.

1.2 Data Preparation and Evaluation

In this section, we briefly outline the data preparation steps we took, as well the evaluation metric that we used for our models. The dataset[1] consists of French and English tab-delimited sentence-pairs. We first split the files into lines, using new line (i.e., `\n`) as the delimiter. Then, we change the data from Unicode to the ASCII format, before normalizing the words from both languages using regular expressions, such as lower-casing the words, and removing any characters that are not alphabets, punctuations or spaces. Next, we pair them up, before selecting the pairs that are both shorter than 15 words (including the punctuations), and of certain form (e.g., English sentences that start with "I am" or "I'm"). This cuts the sentences that would be in our dataset from 232,736 in the initial dataset, to 22,907, where the French vocabulary (i.e., unique words) has 7,019 (normalized) words, while the English vocabulary has 4,638. Finally, we performed a train-test split, such that the test set is 0.1 times of this trimmed dataset.

For this machine translation task, we will feed the French sentence sequentially (i.e., word-by-word) into the encoder, before passing the encoded hidden state(s) to the decoder for English word generation, essentially translating the French sentence to an English one. To evaluate the performance of our seq2seq model, we will use the Recall-Oriented Understudy for Gisting Evaluation (ROUGE) score. In particular, we will be evaluating on ROUGE-1 and ROUGE-2 scores, which work on unigrams and bigrams respectively. For example, ROUGE-1 will look at overlapping unigrams between the generated English sentence and the ground-truth English sentence. Then, with the number of overlapped unigrams, it will calculate the recall, precision and F-measure. Mathematically, using ROUGE-1, we have

$$\text{Recall} = \frac{n(\text{overlapping unigram})}{n(\text{unigram in ground truth})}$$

$$\text{Precision} = \frac{n(\text{overlapping unigram})}{n(\text{unigram in generated sentence})}$$

$$\text{F measure} = \frac{2(\text{precision} \times \text{recall})}{\text{precision} + \text{recall}}$$

where $n(\cdot)$ returns the count. ROUGE-2 similarly has recall, precision and F-measure, but we count the overlapping bigrams. Therefore, the higher the ROUGE scores, the better. Also, from the equations, we can deduce the ROUGE-2 scores of a trained seq2seq model will never be higher than its ROUGE-1 counterpart.

2. Experiments Without Attention

In experiment 1, we used the GRU architecture for both our encoder and decoder (GRU-GRU). GRU is similar to a LSTM, where it has multiple gating mechanisms, but mainly differs from LSTM by the number of gates (two in GRU and three in LSTM). As an encoder, the GRU first initializes a hidden state (set to be tensor of zeroes in our case). Then, we sequentially input our embedded tokens, obtained from a randomly initialized embedding layer. At each step (i.e., input token), we obtain a hidden state and an output vector. We use the final hidden state of the encoder, obtained at the final step with the last token, as the initial input hidden state of our GRU decoder. Unlike the encoder, we the tokens which we sequentially input into the decoder may be the ground-truth tokens (i.e., teacher forcing), or the predicted token from the previous step (i.e., without teacher forcing). The predicted token can be obtained from the output vector at each step, by taking the word with the

[1] Obtained from: http://www.manythings.org/anki/fra-eng.zip

highest probability value (i.e., `topk` for $k = 1$). We trained this seq2seq model, and subsequent ones, for 5 epochs, with stochastic gradient descent (SGD) as our optimizer, setting the learning rate to be 0.01 for both our encoder and decoder. Our objective here is for the decoder output to be close to the ground truth token, as such we use a negative log-likelihood function, since both tensors are valid probability distributions (i.e., softmax applied). As for the dimensions of the hidden state, we use a value of 512 for every of our encoder and decoders, except for LSTM (see below).

After 5 epochs, we obtained a training loss of 0.6817. Then, we evaluate our model on a holdout test set of roughly 2300 sentence-pairs. We show the precision, recall, and F-measure using ROUGE1 and ROUGE2, on the training and test set, in Table 1 and Table 2 below respectively.

	Precision	Recall	F-measure
ROUGE-1	0.754	0.866	0.798
ROUGE-2	0.624	0.758	0.681

Table 1: Evaluation of GRU encoder and GRU decoder (experiment 1) on training set

	Precision	Recall	F-measure
ROUGE-1	0.621	0.715	0.660
ROUGE-2	0.451	0.542	0.488

Table 2: Evaluation of GRU encoder and GRU decoder (experiment 1) on test set

We can see that, for both the training and test set, the recall is higher than the precision across both the training and test sets. This suggest that the translated sentence includes a lot of the relevant words (i.e., many n-gram overlap with the ground-truth), but the translated sentence may be unnecessarily long. This may also suggest that our model has produced many repeated words. For example, one example sentence-pair has the French sentence as “elle deviant de plus en plus belle” and ground-truth English translation as “she is getting prettier”. However, our model translated to “she s prettier prettier”. Suppose the model translates to “she s prettier” instead, then the number of overlapping n-grams remain constant, hence the recall would not change, but the precision would increase, as we take reference to the length of our generated sentence.

In experiment 2 (LSTM-LSTM), we replaced both the GRU encoder and decoder with LSTM, while in experiment 3 (biLSTM-GRU), we replaced the GRU encoder with biLSTM one and keep the GRU decoder. The implementation remains largely the same as the one outlined in section 2.1, however because of the bidirectionality, the final hidden state from the biLSTM encoder will have $2 \times$`hidden_size`$= 2 \times 512 = 1024$ dimensions. Hence, our GRU decoder in experiment 3 will have hidden state dimension of 1024 instead of 512, in order to take in the hidden state from the encoder. In Tables 3 and 4 below, we show the different evaluation metrics on the test and training set respectively, with the trained models from experiments 2 and 3.

	Experiment 2: LSTM-LSTM			Experiment 3: biLSTM-GRU		
	Precision	Recall	F-measure	Precision	Recall	F-measure
ROUGE-1	0.744	**0.870**	**0.799**	0.744	0.864	0.796
ROUGE-2	**0.626**	**0.764**	**0.686**	0.615	0.747	0.670

Table 3: Evaluation of LSTM-LSTM (experiment 2) and biLSTM-GRU (experiment 3) on training set

	Experiment 2: LSTM-LSTM			Experiment 3: biLSTM-GRU		
	Precision	Recall	F-measure	Precision	Recall	F-measure
ROUGE-1	**0.625**	**0.723**	**0.667**	0.624	0.719	0.664
ROUGE-2	**0.455**	**0.551**	**0.495**	0.444	0.537	0.482

Table 4: Evaluation of LSTM-LSTM (experiment 2) and biLSTM-GRU (experiment 3) on test set

Firstly, note that we have the same issue of higher recall as compared to precision for both LSTM-LSTM and biLSTM-GRU models. Next, comparing all three experiments, we see that their test-set ROUGE1 F-measures are all close, ranging from 0.660 to 0.667, with LSTM-LSTM taking the lead, followed by biLSTM-GRU, then GRU-GRU. This is expected, since LSTM has a larger number of parameters by virtue of having more gates than GRU, thereby having more capacity to capture the nuances in the data. The better performance in LSTM-LSTM as compared to biLSTM-GRU also signals that bidirectionality does not necessarily translate to better encoding of the source sentences for the decoder to perform a better translation. This could be influenced by how we selected our sentence-pairs – since we enforced that the English sentences must contain certain prefixes, the structures of all the sentences in our training and test sets may be more homogeneous, as compared to real-world sentences. Since bidirectionality attempts to address the fluidity of natural language sentences, it is rendered less useful in our homogeneous dataset. Despite twice the capacity of the hidden state (in terms of dimensionality) to capture information in the source sentence, the additional information is mostly redundant.

Now, notice that all the ROUGE-1 metrics of biLSTM-GRU are higher than that of GRU-GRU, but its ROUGE-2 metrics all performed poorer than GRU-GRU. This suggests that biLSTM-GRU struggles slightly more than GRU-GRU when it comes to ordering of the English words to produce a translated sentence. Regardless, these models have negligible performance differences, in terms of ROUGE metrics. As such, we may want to consider using GRU-GRU due to its less complex gating mechanism and fewer parameters, hence a much faster run-time as well. Next, we will look at models that implement the attention mechanism.

3. Experiments With Attention

In this section, we experimented with two models: a GRU-GRU model with attention mechanism (experiment 4), and a Transformer-GRU model (experiment 5). For the GRU-GRU model, the attention mechanism is implemented between a decoder's hidden state at step t, denoting it as $\mathbf{q}_t$, and all the encoder hidden states, denoting them as $K = [\mathbf{k}_1, \dots, \mathbf{k}_n]$. Then, the attention with respect to $\mathbf{q}_t$, denoted as $\boldsymbol{\alpha}_t$, is calculated as,

$$\boldsymbol{\alpha}_t = \mathrm{softmax}(K^T \mathbf{q}_t) K = \mathrm{softmax}\left(\begin{bmatrix} \mathbf{k}_1^T \mathbf{q}_t \\ \vdots \\ \mathbf{k}_n^T \mathbf{q}_t \end{bmatrix}\right) K$$

Since $\boldsymbol{\alpha}_t$ is a linear combination of the encoder hidden states, we can append this information to the initial decoder's hidden state, $\mathbf{q}_t$, forming $\begin{bmatrix} \mathbf{q}_t \\ \boldsymbol{\alpha}_t \end{bmatrix}$, and use it to generate the next token, instead of using the output from the GRU directly. This allows us to access relevant information from the encoder directly.

For the Transformer-GRU model, the (self) attention mechanism entirely lies within the Transformer encoder. In the GRU-GRU model with attention, our query at every time step is the decoder's hidden state, while the keys and values are from the encoder's hidden states. In the Transformer, however, the keys, queries, and values are all from the (projected) source word embeddings. To learn a more robust representation, we implemented multi-head self-attention, with eight attention heads. The output of each attention head A_i can be computed as,

$$A_i = \frac{\mathrm{softmax}(K_i Q_i^T)}{\sqrt{d}} V_i,$$

where $K_i = W_i^K X, Q_i = W_i^Q X, V_i = W_i^V X$, with $K_i, Q_i, V_i \in \mathbb{R}^{d \times d}$. Here, X are all the word embeddings in the sentence. In other words, the attention mechanism encodes the pairwise relevance between each of the source word embeddings. In our implementation, we also included a sinusoidal positional encoding, with 6 layers in the Transformer encoder. Since the Transformer would output encodings with same length as the input sequence, we take the mean of the final output encoding and pass it as the initial hidden state for our GRU-decoder.

In Tables 5 and 6 below, we tabulated the different evaluation metrics on the test and training set respectively, comparing the GRU-GRU and Transformer-GRU models.

	Experiment 4: GRU-GRU			Experiment 5: Transformer-GRU		
	Precision	Recall	F-measure	Precision	Recall	F-measure
ROUGE-1	**0.655**	**0.777**	**0.707**	0.501	0.552	0.520
ROUGE-2	**0.473**	**0.594**	**0.522**	0.311	0.359	0.328

Table 5: Evaluation of GRU-GRU (experiment 4) and Transformer-GRU (experiment 5) on training set

	Experiment 4: GRU-GRU			Experiment 5: Transformer-GRU		
	Precision	Recall	F-measure	Precision	Recall	F-measure
ROUGE-1	**0.561**	**0.665**	**0.604**	0.485	0.536	0.503
ROUGE-2	**0.366**	**0.459**	**0.404**	0.300	0.346	0.316

Table 6: Evaluation of GRU-GRU (experiment 4) and Transformer-GRU (experiment 5) on test set

Although both models use attention, the performance of the GRU-GRU model (experiment 4) is demonstrably better, with a ROUGE-1 F-measure significantly higher than that of the Transformer-GRU. This is because the attention mechanism in the GRU-GRU model serves to pass relevant information directly to the decoder for the downstream translation task, whereas the attention mechanism in the Transformer-GRU model serves to encode the information in the source sentence –relevant information in the encoder were not directly passed to the decoder for the translation task, but the information was instead compressed (i.e., taking the mean). This lets the decoder more effectively use the encoded information from the source sentence for the translation task.

However, neither of the models were able to match up with the other models that did not use attention. For a better point of comparison, we shall focus on the GRU-GRU (experiment 1) model without any attention. Comparing between this and the GRU-GRU model with attention (experiment 4), the latter performed appreciably worse, despite using identical architecture. This implies that the attention mechanism has directly led to a worse performance, given the training done. One reason for this could be the lack of quality training, causing the attention mechanism to produce more noise than information for the decoder to predict the next token. For instance, for a particular French sentence with an English ground-truth sentence of "i m sorry that i ve made you so unhappy", the model generated "i m sorry that you you you unhappy unhappy". In the ground truth sentence, we can see that the subject "I" has strong dependencies to "made" (i.e., the verb) and "you" (i.e., the object). As such, the attention calculated for the hidden state after the "that" token was generated might be split between the French counterpart of "made" and "you", such that their softmax probabilities are both similar, but neither are close to 1. Hence, the attention mechanism here could have introduced noise to the decoder's generation of the next token, as the decoder is confident in neither of the predictions. This could also be a symptom of a small number of epochs and lack of diverse data, as the model may not have sufficient quality training to identify the correct dependencies to attend to, given the context of the sentence.

Comparing between this and Transformer-GRU, we see that the Transformer-GRU model performed much worse. One possible reason is the lack of inductive bias within the architecture of a Transformer. This means that it requires a large amount of data to perform well and capture intricacies within the dataset; this is not the case for our project. Along the same vein, the homogeneity of our sentence-pairs plays to the advantage of a recurrent model, like the GRU, which has biases inherent in the architecture that models the structure of a sentence. Therefore, the GRU-GRU model could take into account the predictable structure of our input sentences. On the other hand, Transformer takes in all the tokens in the sentence in parallel, which does not leverage on the predictable structure of our sentence-pairs. This is further exacerbated by using a non-learnable positional encoder. Hence, it is not well-equipped to leverage on the predictable sentence structure.

As further experiment, we implemented a Transformer-GRU model, but with attention identical in experiment 4 between the decoder and the encoder's hidden states (which are the final output of the Transformer layer). The model achieved an even lower ROUGE1 F-measure of 0.470 and ROUGE2 F-measure of 0.280. This further exemplifies that the attention mechanism of experiment 4, in our current implementation, produces more noise than signal, especially in the face of homogeneous sentence structure, and lack of training and data. In general, the attention-based models seemed to severely underperform the non-attention-based ones. Apart from the reasons explained above, another crucial context of our dataset is the length of the sentences. Since we limited both English and French sentences to a maximum length of 15, the short length of each sequence does not pose much of an issue to the recurrent models like GRU and LSTM. This short sequence length also renders the ability to model long-range dependency of attention-based models less useful.

4. Conclusion

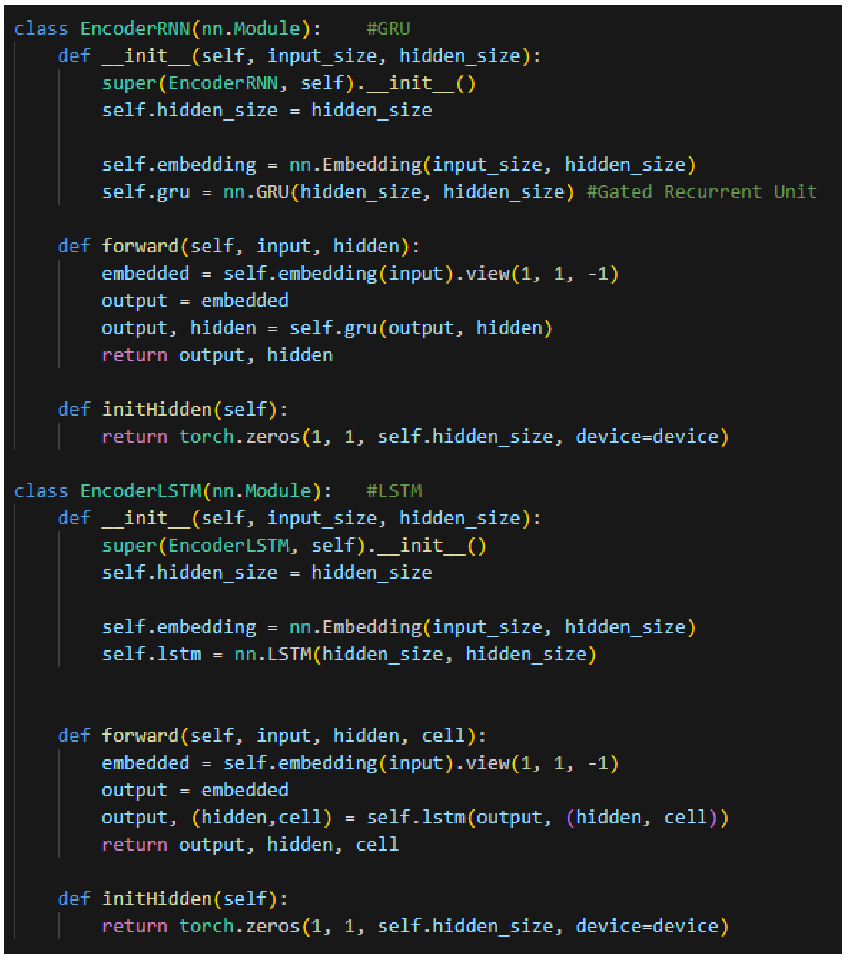
Through our experiments, we noticed that even though most, if not all, of the current state-of-the-art language models are Transformer-based, with extensive use of attention, the same does not apply to our case, given our implementation and resources. In fact, all the model with a recurrent architecture performed identically well, and much better than those with attention. Also, our evaluation was done based mostly on ROUGE scores, however, there are some limitations to that. Since ROUGE scores are calculated based on overlapping n-grams, it does not consider the general structure (especially for small n, like unigram) and the semantic of the sentence. A sentence could be judged to have a good ROUGE metric, across precision, recall and F-measure, but it may not be a semantically sound sentence due to its incoherent structure. In the future, other metrics – such as BERTscore or human evaluation – could be used in conjunction with ROUGE scores to ensure a more holistic evaluation of the models.

5. Appendix

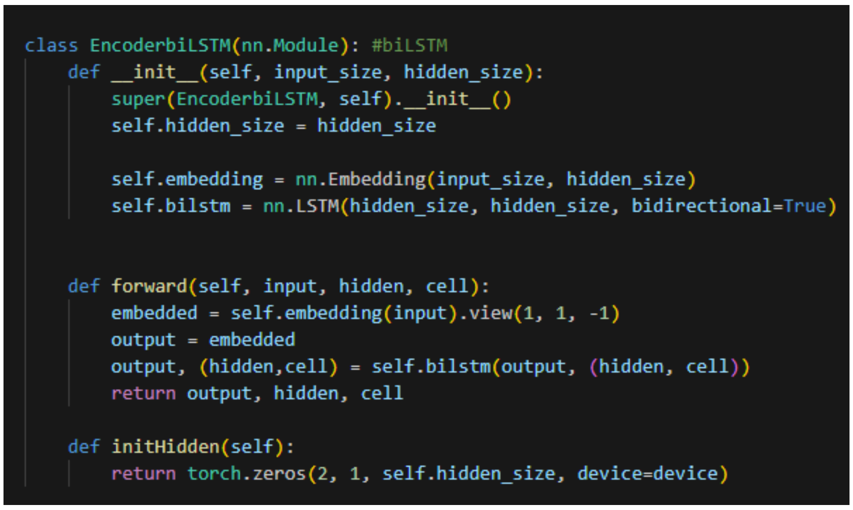
Code snippets.

In this appendix, we include only the definition of the different encoders and decoders, including their `forward` function.

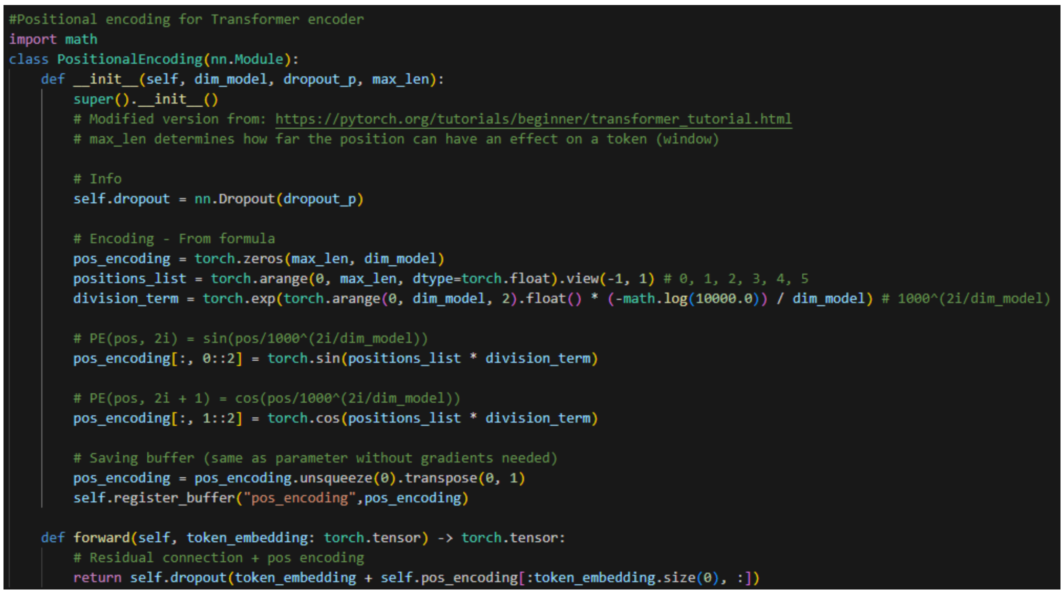
A. All the encoder network used.



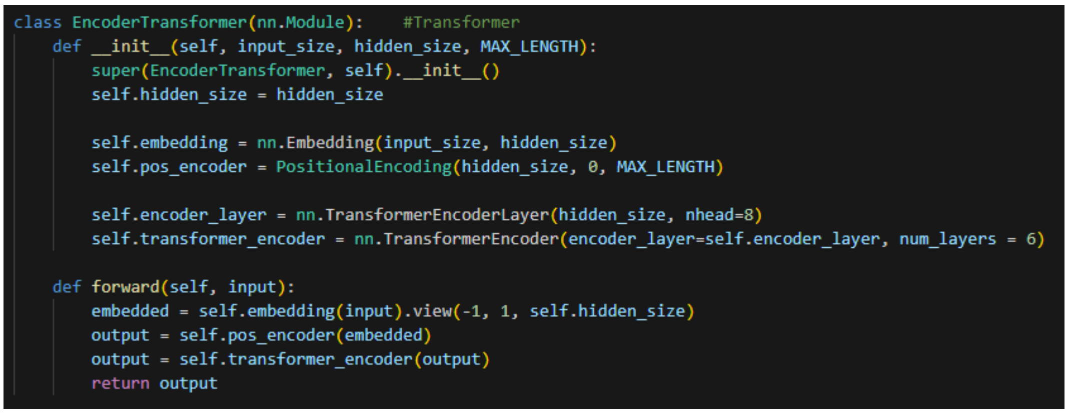
```
class EncoderRNN(nn.Module):    #GRU
    def __init__(self, input_size, hidden_size):
        super(EncoderRNN, self).__init__()
        self.hidden_size = hidden_size

        self.embedding = nn.Embedding(input_size, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size) #Gated Recurrent Unit

    def forward(self, input, hidden):
        embedded = self.embedding(input).view(1, 1, -1)
        output = embedded
        output, hidden = self.gru(output, hidden)
        return output, hidden

    def initHidden(self):
        return torch.zeros(1, 1, self.hidden_size, device=device)

class EncoderLSTM(nn.Module):   #LSTM
    def __init__(self, input_size, hidden_size):
        super(EncoderLSTM, self).__init__()
        self.hidden_size = hidden_size

        self.embedding = nn.Embedding(input_size, hidden_size)
        self.lstm = nn.LSTM(hidden_size, hidden_size)


    def forward(self, input, hidden, cell):
        embedded = self.embedding(input).view(1, 1, -1)
        output = embedded
        output, (hidden,cell) = self.lstm(output, (hidden, cell))
        return output, hidden, cell

    def initHidden(self):
        return torch.zeros(1, 1, self.hidden_size, device=device)
```



```
class EncoderbiLSTM(nn.Module): #biLSTM
    def __init__(self, input_size, hidden_size):
        super(EncoderbiLSTM, self).__init__()
        self.hidden_size = hidden_size

        self.embedding = nn.Embedding(input_size, hidden_size)
        self.bilstm = nn.LSTM(hidden_size, hidden_size, bidirectional=True)


    def forward(self, input, hidden, cell):
        embedded = self.embedding(input).view(1, 1, -1)
        output = embedded
        output, (hidden,cell) = self.bilstm(output, (hidden, cell))
        return output, hidden, cell

    def initHidden(self):
        return torch.zeros(2, 1, self.hidden_size, device=device)
```

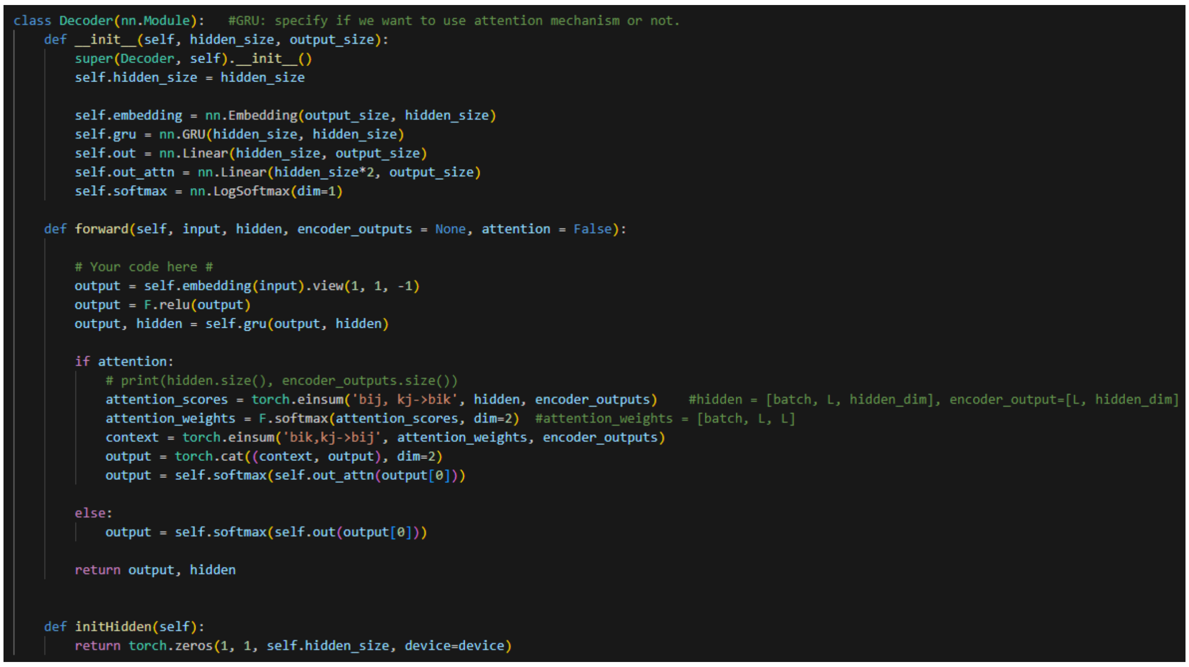


```python
#Positional encoding for Transformer encoder
import math
class PositionalEncoding(nn.Module):
    def __init__(self, dim_model, dropout_p, max_len):
        super().__init__()
        # Modified version from: https://pytorch.org/tutorials/beginner/transformer_tutorial.html
        # max_len determines how far the position can have an effect on a token (window)

        # Info
        self.dropout = nn.Dropout(dropout_p)

        # Encoding - From formula
        pos_encoding = torch.zeros(max_len, dim_model)
        positions_list = torch.arange(0, max_len, dtype=torch.float).view(-1, 1) # 0, 1, 2, 3, 4, 5
        division_term = torch.exp(torch.arange(0, dim_model, 2).float() * (-math.log(10000.0)) / dim_model) # 1000^(2i/dim_model)

        # PE(pos, 2i) = sin(pos/1000^(2i/dim_model))
        pos_encoding[:, 0::2] = torch.sin(positions_list * division_term)

        # PE(pos, 2i + 1) = cos(pos/1000^(2i/dim_model))
        pos_encoding[:, 1::2] = torch.cos(positions_list * division_term)

        # Saving buffer (same as parameter without gradients needed)
        pos_encoding = pos_encoding.unsqueeze(0).transpose(0, 1)
        self.register_buffer("pos_encoding",pos_encoding)

    def forward(self, token_embedding: torch.tensor) -> torch.tensor:
        # Residual connection + pos encoding
        return self.dropout(token_embedding + self.pos_encoding[:token_embedding.size(0), :])
```

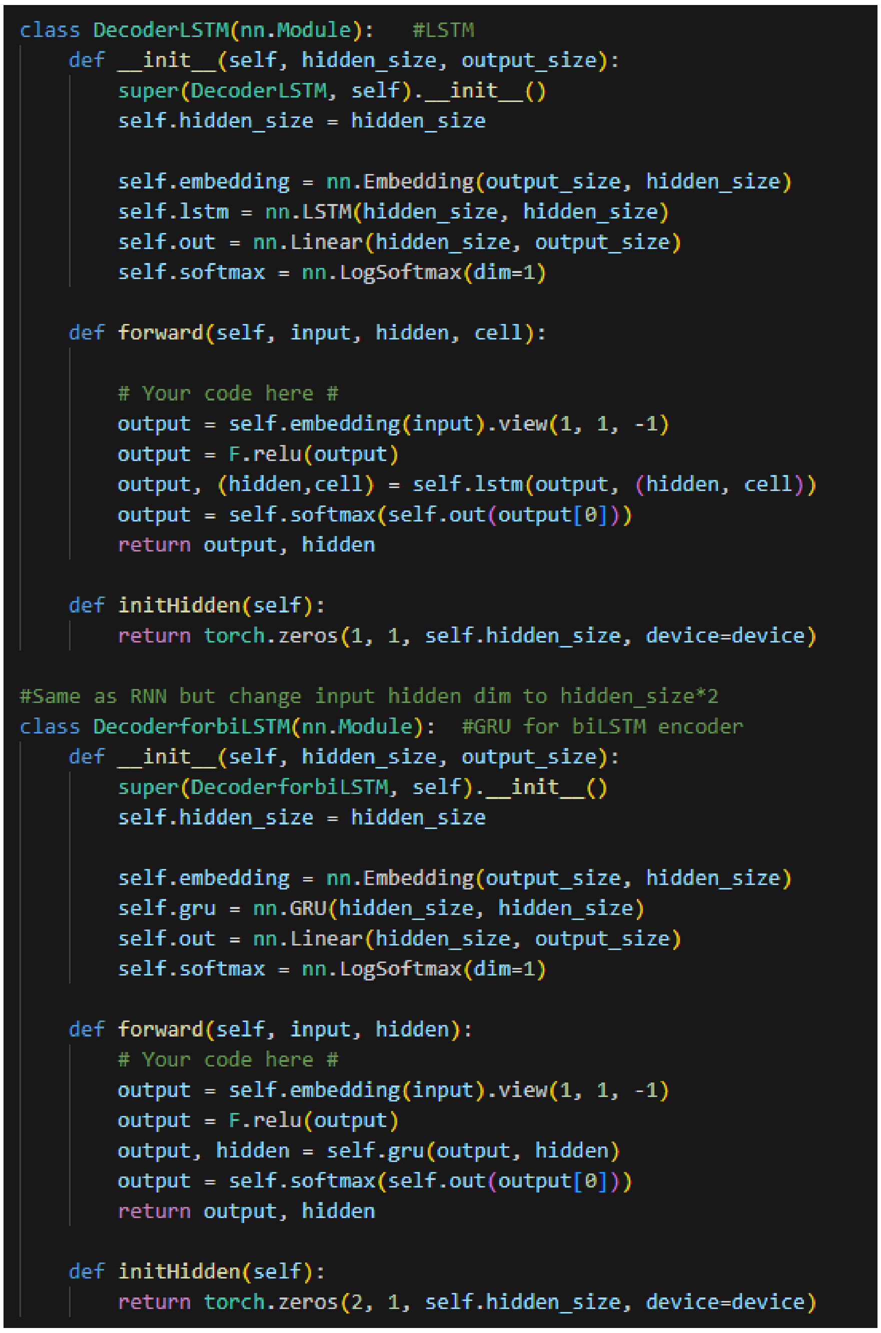


```python
class EncoderTransformer(nn.Module):     #Transformer
    def __init__(self, input_size, hidden_size, MAX_LENGTH):
        super(EncoderTransformer, self).__init__()
        self.hidden_size = hidden_size

        self.embedding = nn.Embedding(input_size, hidden_size)
        self.pos_encoder = PositionalEncoding(hidden_size, 0, MAX_LENGTH)

        self.encoder_layer = nn.TransformerEncoderLayer(hidden_size, nhead=8)
        self.transformer_encoder = nn.TransformerEncoder(encoder_layer=self.encoder_layer, num_layers = 6)

    def forward(self, input):
        embedded = self.embedding(input).view(-1, 1, self.hidden_size)
        output = self.pos_encoder(embedded)
        output = self.transformer_encoder(output)
        return output
```

B. All the decoder networks used.



```python
class Decoder(nn.Module):   #GRU: specify if we want to use attention mechanism or not.
    def __init__(self, hidden_size, output_size):
        super(Decoder, self).__init__()
        self.hidden_size = hidden_size

        self.embedding = nn.Embedding(output_size, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size)
        self.out = nn.Linear(hidden_size, output_size)
        self.out_attn = nn.Linear(hidden_size*2, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden, encoder_outputs = None, attention = False):

        # Your code here #
        output = self.embedding(input).view(1, 1, -1)
        output = F.relu(output)
        output, hidden = self.gru(output, hidden)

        if attention:
            # print(hidden.size(), encoder_outputs.size())
            attention_scores = torch.einsum('bij, kj->bik', hidden, encoder_outputs)    #hidden = [batch, L, hidden_dim], encoder_output=[L, hidden_dim]
            attention_weights = F.softmax(attention_scores, dim=2)  #attention_weights = [batch, L, L]
            context = torch.einsum('bik,kj->bij', attention_weights, encoder_outputs)
            output = torch.cat((context, output), dim=2)
            output = self.softmax(self.out_attn(output[0]))

        else:
            output = self.softmax(self.out(output[0]))

        return output, hidden


    def initHidden(self):
        return torch.zeros(1, 1, self.hidden_size, device=device)
```


```
class DecoderLSTM(nn.Module):    #LSTM
    def __init__(self, hidden_size, output_size):
        super(DecoderLSTM, self).__init__()
        self.hidden_size = hidden_size

        self.embedding = nn.Embedding(output_size, hidden_size)
        self.lstm = nn.LSTM(hidden_size, hidden_size)
        self.out = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden, cell):

        # Your code here #
        output = self.embedding(input).view(1, 1, -1)
        output = F.relu(output)
        output, (hidden,cell) = self.lstm(output, (hidden, cell))
        output = self.softmax(self.out(output[0]))
        return output, hidden

    def initHidden(self):
        return torch.zeros(1, 1, self.hidden_size, device=device)

#Same as RNN but change input hidden dim to hidden_size*2
class DecoderforbiLSTM(nn.Module):  #GRU for biLSTM encoder
    def __init__(self, hidden_size, output_size):
        super(DecoderforbiLSTM, self).__init__()
        self.hidden_size = hidden_size

        self.embedding = nn.Embedding(output_size, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size)
        self.out = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden):
        # Your code here #
        output = self.embedding(input).view(1, 1, -1)
        output = F.relu(output)
        output, hidden = self.gru(output, hidden)
        output = self.softmax(self.out(output[0]))
        return output, hidden

    def initHidden(self):
        return torch.zeros(2, 1, self.hidden_size, device=device)
```