Implementing RNN, LSTM, Sequence and Transformer

1. Implementing RNN

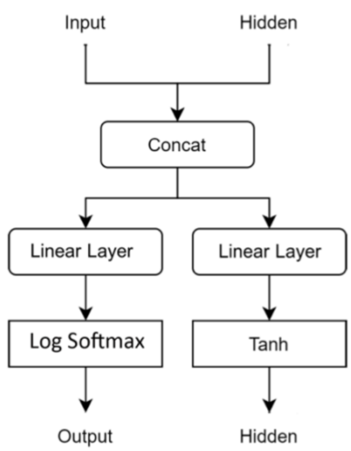
You should make sure you have the following packages installed

```
$ pip install torchtext==0.8.1
$ pip install torch==1.7.1
$ pip install spacy==2.3.5
$ pip install tqdm
$ pip install numpy
```

Additionally, you will need the Spacy tokenizers in English and German language, which can be downloaded as such:

```
$python -m spacy download en
$python -m spacy download de
```

You will be using PyTorch Linear layers and activations to implement a vanilla RNN unit. Please refer to the following structure and complete the code in `RNN.py`:



2. Implement LSTM

You will be using PyTorch nn.Parameter and activations to implement an LSTM unit. You can simply translate the following equations using nn.Parameter and PyTorch activation functions to build an LSTM from scratch:

$$
\begin{aligned}
i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\
f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\
g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\
o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\
h_t &= o_t \odot \tanh(c_t)
\end{aligned}
$$

Here's a great visualization of the above equation to help you understand LSTM unit.
https://colah.github.io/posts/2015-08-Understanding-LSTMs/

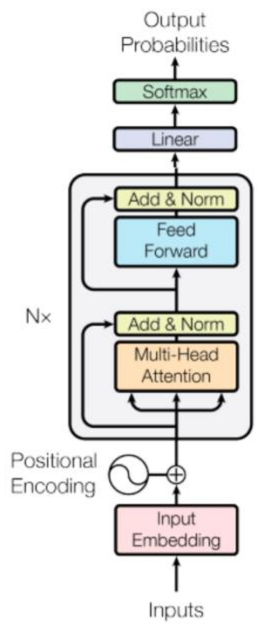
3. Seq2Seq Implementation

seq2seq.py you will see the files needed to complete this section. In these files you will complete the initialization and forward pass in __init__ and forward function. Encoder.py Decoder.py Seq2Seq.py.

Train seq2seq on the dataset with the default hyperparameters.

4. Transformers Implementation

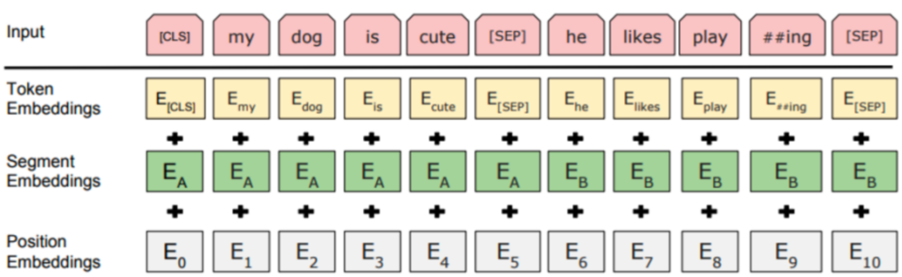
We will be implementing a one-layer Transformer encoder which, similar to an RNN, can encode a sequence of inputs and produce a final output of possibility of tokens in target language. The architecture can be seen below. You will see the file Transformer.py. You will implement the functions in the TransformerTranslator class.



You can refer to the original paper in the link below:
https://arxiv.org/pdf/1706.03762.pdf)%20for%20more%20details

Additional help for Transformer Implementation

We will format our input embeddings similarly to how they are constructed in [BERT (source of figure)](https://arxiv.org/pdf/1810.04805.pdf). Unlike a RNN, a Transformer does not include any positional information about the order in which the words in the sentence occur. Because of this, we need to append a positional encoding token at each position. (We will ignore the segment embeddings and [SEP] token here, since we are only encoding one sentence at a time). We have already appended the [CLS] token for you in the previous step.



Your first task is to implement the embedding lookup, including the addition of positional encodings. Complete the code section for Deliverable 1, which will include part of __init__ and embed.

Attention can be computed in matrix-form using the following formula:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

We want to have multiple self-attention operations, computed in parallel. Each of these is called a *head*. We concatenate the heads and multiply them with the matrix *attention_head_projection* to produce the output of this layer.
After every multi-head self-attention and feedforward layer, there is a residual connection + layer normalization. Make sure to implement this, using the following formula:

$$\text{LayerNorm}(x + \text{Sublayer}(x))$$

Implement the function `multi_head_attention` for **Deliverable 2**. We have already initialized all of the layers you will need in the constructor.

Complete code for **Deliverable 3** in `feedforward_layer`: the element-wise feed-forward layer consisting of two linear transformers with a ReLU layer in between.

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

Complete code for **Deliverable 4** in `final_layer`., to produce probability scores for all tokens in target language.

Put it all together by completing the method `forward`, where you combine all of the methods you have developed in the right order to perform a full forward pass.

Train the transformer architecture on the dataset with the default hyperparameters – you should get a perplexity better than that for seq2seq. Then perform hyperparameter tuning and include the improved results.

Include the accuracy of the Seq2Seq model and Transformer architecture before and after hyperparameter tuning.