Topics in Artificial Intelligence (CPSC 532S): Assignment #3: *RNNs for Language Modeling*[*]

Due on Saturday, February 13, 2021 at 11:59pm PST

In this assignment you will build a series of recurrent neural network (RNN)-based *encoders* and *decoders*. The decoder is also sometimes called a *neural language model*. Effectively you will build a sequence-to-sequence translation model that is very common in language translation (*e.g.*, translating from English to French). However, for simplicity we will build a model that translates from English to Pig Latin. Note that you will reuse components of this assignment (the *decoders*) for Assignment 4, so it is to your benefit to not only write the required code for this assignment, but to do so in a structured manner that would be reusable.

Programming environment

You will be using **PyTorch** for this assignment. You are welcome to use Colab, Microsoft Azure or Amazon AWS for this assignment. If you opt to do this assignment on your personal computer, it is your responsibility to install PyTorch and ensure it is running properly. A number of tutorials for doing this are available on-line.

Data

To train the models you will be using sentences that correspond to (approximately 20K) training images we used for Assignment 2 from MS-COCO dataset (20,000 sentences in total) and 500 sentences for validation (corresponding to 100 validation images). Note that the full MS-COCO contains approximately 400,000 sentences, with 5 sentences for each image. Here we sub-sampled the data for faster processing. You will be reusing the data from previous assignment.

Assignment Instructions

You will need to follow instructions in the corresponding Jupyter/Colab notebook and submit the assignment as saved Jupiter/Colab notebook when done (including the results of executed code). As part of the assignment we are providing pre-processing code that tokenizes the text, defines a vocabulary of `vocabularySize` $= W = 2,000$ of most frequent words ($1,000$ words in English and $1,000$ in Pig Latin) and represents each word as either 1-hot encoding ($\mathbf{x} \in \mathbb{R}^{2,000}$) or word2vec encoding ($\mathbf{x} \in \mathbb{R}^{300}$), where the length of the encoding vector we denote `wordEncodingSize`. Note that word2vec encoding only make sense for English words and should not be used for Pig Latin words. Further note that our 1-hot encoding defined over *both* languages is a bit of a simplification. In more realistic translation models the 1-hot encoding is typically built individually for each language since the vocabulary and dictionary lengths for different languages tend to be different. We are not doing this in this assignment to avoid clutter. Once data is pre-processed you will follow the notebook through the following steps (as you do, you will find the Seq2seq PyTorch Tutorial extremely helpful).

[*]This assignment was heavily inspired and, in part, adopted from the Assignment 3 of University of Tornoto's CSC 421, initially provided by Paul Vicol. The adoption was done by Tanzila Rahman.

Part 1: Encoder-Decoder Language Translation with Teacher-Forcing

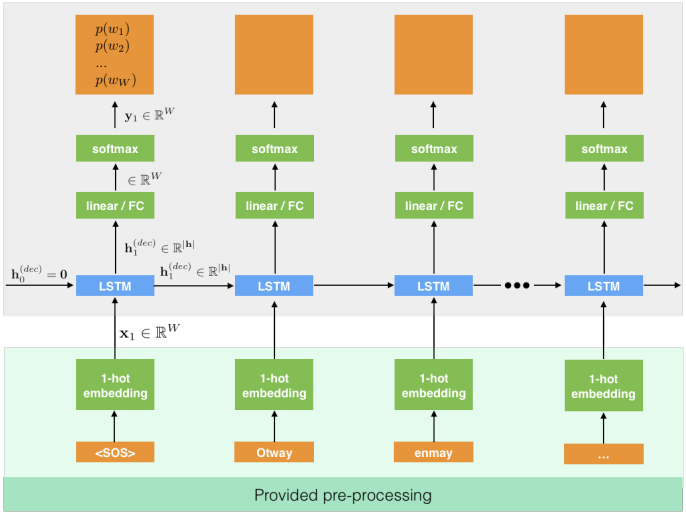
Language, or otherwise, translation is most often formulated as a sequence-to-sequence (seq2seq) problem. In this assignment, both the input and output are sequences of words. A common architecture used for seq2seq problems is the encoder-decoder model [1], composed of two Recurrent Neural Networks (RNNs).

The *encoder* RNN effectively compresses the input sequence into a fixed vector representation, encoded by a hidden state $\mathbf{h}_T^{(enc)}$. This is done by recursively processing the sequence of input tokens (words in our case) to produce a series of hidden states in a recursive manner, i.e., $\mathbf{h}_i^{(enc)} = RNN_{enc}\left(\mathbf{h}_{i-1}^{(enc)}, \mathbf{x}_i^{(\text{English})}\right)$; typically to initialize the recursion the initial hidden state is set to a zero vector $\mathbf{h}_0 = \mathbf{0}$. Note that T is simply the number of tokens/words in a given sentence[1].

The *decoder* RNN takes the compressed representation from the encoder, $\mathbf{h}_T^{(enc)}$, and produces, recursively, one token/word at a time, the output sentence in the target language. This is achieved by first producing a series of hidden representations $\mathbf{h}_i^{(dec)} = RNN_{dec}\left(\mathbf{h}_{i-1}^{(dec)}, \mathbf{x}_{i-1}^{(\text{Pig Latin})}\right)$ and then decoding each of these hidden representation $\mathbf{h}_i^{(dec)}$ to a token/word $\text{token}_i^{(\text{Pig Latin})}$. The conditioning on the encoding of the input English sentence, in the simplest form, is done by simply initializing the decoder RNN recursion to $\mathbf{h}_0^{(dec)} = \mathbf{h}_T^{(enc)}$.

The above description is the simplest encoder-decoder architecture. We will look at more sophisticated forms of decoders in Part 2 and Part 3 of this assignment.

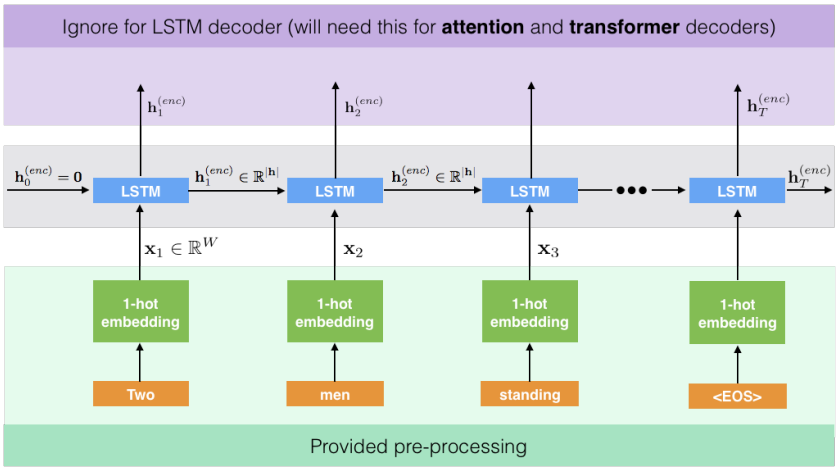
1. **Building Pig Latin Language Decoder.** First you will build a decoder (also known as language model) using RNN with LSTM units and 1-hot encoding for the words. The model is illustrated below.



The process and the sample code for building such a model is described in the PyTorch Seq2seq tutorial referenced above under "*Simple Decoder*", but you will need to use LSTM units instead of GRU. Starting with GRU implementation, as in the tutorial, may not be a bad starting point. The RNN you will build will need to take as input a tensor of sentences size ($1 \times \texttt{T} \times \texttt{wordEncodingSize}$) and output the distribution over words ($1 \times \texttt{T} \times \texttt{vocabularySize}$). In effect, at every time step of the RNN, the input is a current word and the output is the distribution over the following word. For the hidden state, use dimensionality $|\mathbf{h}| = 300$. Note, in training `T` can be assumed to be known, in inference `T` is determined by generated `<SOS>` token or `maxSequenceLength`.

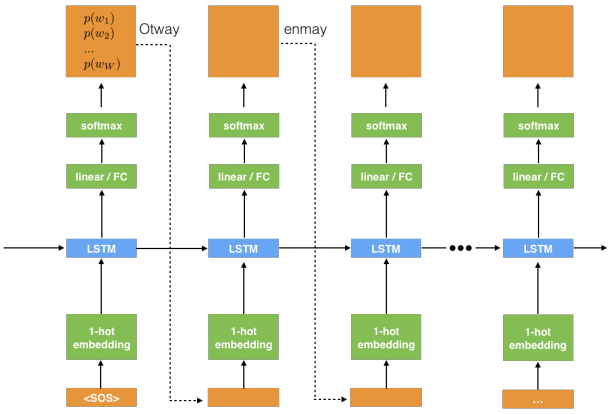
[1] In practice, in most implementations T could be defined as `maxSequenceLength` and fixed across the entire dataset. This is a strategy one must employ when running the model using mini-batches of more than one sample; something we will not require for this assignment.

2. **Building English Language Encoder.** Here you will follow the details in the Encoder portion of the PyTorch Seq2seq tutorial; again here you will need to replace GRU units by LSTM ones. The encoder should take the previous hidden state and one word at a time and progressively encode the sentence. The illustration of the model is shown in the Figure below. Again for the hidden state use dimensionality of $|\mathbf{h}| = 300$.



3. **Connecting Encoder to Decoder and Training End-to-End.** You will now connect the encoder to the decoder and train the Seq2seq encoder-decoder architecture end-to-end. Note that you will need to start the decoder from the `<SOS>` token and pass the last hidden state out of the encoder as the first hidden state of the decoder. Note that the input tokens for the decoder could be either ground truth tokens of translated sentence or the predicted tokens from prior steps. The proportion of each is dictated by the teacher forcing ratio. For this part you can start by implementing `teacher_forcing_ratio = 1.0`, which will only utilize the ground truth tokens. This will then need to be modified in step 7.

4. **Building Language Decoder MAP Inference.** Inference in the model follows the training and is referred to as "*Evaluation*" in the PyTorch Seq2seq tutorial. The function should start from a single word, predict the distribution over the next word, pick the most likely next word from the distribution and pass it as the input to the next LSTM unit and so on until `<EOS>` tag is reached. The inference is illustrated in the Figure below:



To evaluate the trained model using the inference procedure, infer sentences that result by translating 5 sentences of your choosing from English to Pig Latin (your notebook should show the 5 input and the corresponding 5 output sentences).

5. **Building Language Decoder Sampling Inference.** This is similar to above, but instead of picking the most likely word from the distribution predicted by the LSTM at every step, you should sample from the distribution according to the predicted probability. Evaluate the procedure by illustrating translations of the 5 same sentences from the above by drawing 4 sample output sentences for each (notebook should show 20 resulting sentences).

6. **Testing.** Take the validation set provided in the assignment that consists of 500 sentences from MS-COCO not used for training, for each run encoder then decoder using **MAP** inference. Report the average similarity between pairs of input and predicted output sentences using BLEU score (code for computing the score for pair of sentences is provided in the notebook).

7. **Experiment with Teacher Forcing.** Now redo steps 3-4 with `teacher_forcing_ratio = 0.9` and `teacher_forcing_ratio = 0.8`. Comment on the results, including speed of convergence and the overall quality of results.

8. **Encoding as Generic Feature Representation.** Use the final hidden state of the encoder to measure similarity between the first 10 validation sentences and the entire training set of 20,000 sentences. You would most likely want to pre-process the training set and save hidden states, so you do not need to re-compute those for finding nearest neighbor similarity.

Part 2: Translation with Attention Decoder

One issue with the model implemented in Part 1 is that the encoding of the input sequence, due to its underlying recursive nature, tends to encode more recent tokens (appearing towards the end of the input sentence) with higher fidelity than the earlier ones. Consequently, this is one of the reasons in more realistic implementations of language-to-language translation that use architectures similar to the one in Part 1, the input sequences are often inverted as a pre-processing step.

Attention allows a model to look over the input sequence, and focus on relevant input tokens when producing the corresponding output tokens. In other words, attention can help the model to dynamically retrieve tokens from the input to enhance the decoding, e.g., focusing on the input word "Man" (English) when producing the output token "Anmay" (Pig Latin).

This can be achieved by modifying the decoder from Part 1 to take into account not only $\mathbf{h}_T^{(enc)}$, but also some dynamic combination of the encoder hidden states from sequence $\left\{\mathbf{h}_1^{(enc)}, \mathbf{h}_2^{(enc)}, \cdots, \mathbf{h}_T^{(enc)}\right\}$, specific to the output token being decoded. The reason this works is that the hidden states produced by the encoder can be interpreted as encoding of corresponding tokens in the input plus some context. At each decoding step, the attention-based decoder computes a normalized weighting over the encoder hidden states, where the weight given to each encoder hidden state indicates its relevance in determining the current output token.

Specifically, at decoding step t the decoder computes an attention weight $\alpha_i^{(t)}$ for each of the encoder hidden states $\mathbf{h}_i^{(enc)}$. These attention weights are defined such that $0 \leq \alpha_i^{(t)} \leq 1$ and sum up to 1:

$$\sum_{i=0}^{T} \alpha_i^{(t)} = 1. \tag{1}$$

In the most general form, $\alpha_i^{(t)}$ should be a function of the encoder hidden state i, the previous decoder hidden state $t-1$ and the current decoder input token encoding. In other words, $\alpha_i^{(t)} = f\left(\mathbf{h}_i^{(enc)}, \mathbf{h}_{t-1}^{(dec)}, \mathbf{x}_t^{(dec)}\right)$, where i ranges over the length of the input sequence. Note, in many implementations, this can be simplified to $\alpha_i^{(t)} = f\left(\mathbf{h}_i^{(enc)}, \mathbf{h}_{t-1}^{(dec)}\right)$ or $\alpha_i^{(t)} = f\left(\mathbf{h}_i^{(enc)}, \mathbf{x}_t^{(dec)}\right)$. Once the attentions are computed, the context vector

can be obtained as the weighted combination of encoder hidden states:

$$\mathbf{c}_t = \sum_{i=1}^{T} \alpha_i^{(t)} \mathbf{h}_i^{(enc)}. \tag{2}$$

This context vector can be concatenated to the *regular* input to the RNN cell input, e.g., as follows:

$$\mathbf{h}_t^{(dec)} = RNN_{dec}\left(\mathbf{h}_{t-1}^{(dec)}, \left[\mathbf{x}_t^{(dec)}; \mathbf{c}_t\right]\right). \tag{3}$$

Note: To reduce dimensionally and add a bit of robustness, for this part of the assignment instead of using 1-hot embedding vectors directly, for above computations, we will further project these vectors using the following operations $\hat{\mathbf{x}}_t^{(dec)} = \text{Dropout}(\text{Linear}(\mathbf{x}_t^{(dec)}), 0.1)$, where a linear layer is of size $2,000 \times 300$.

There are different ways to define the function f. To generalize across most attention models we consider attention as a function of the three inputs (queries, keys, values), denoted by $(\mathbf{Q}, \mathbf{K}, \mathbf{V})$, where query-keys are used to compute the attention weights and values are used to construct the context vector.

1. **Implementing Additive Attention.** For this part of the assignment we will *learn* a function f, parametrized by a two layer fully-connected network with ReLU activation for the first layer. We will call this form of attention *additive attention*. In other words, we define the following network that produces unnormalized weights first:

   $$\beta_i^{(t)} = \mathbf{W}_2 \left[\text{ReLU}(\mathbf{W}_1[\mathbf{Q}_t; \mathbf{K}_i] + \mathbf{b}_1)\right] + \mathbf{b}_2 \tag{4}$$

   which we then normalize to sum up to 1 using softmax:

   $$\alpha^{(t)} = \text{softmax}(\beta^{(t)}) \tag{5}$$

   to obtain the final context vector:

   $$\mathbf{c}_t = \sum_{i=1}^{T} \alpha_i^{(t)} \mathbf{V}_i. \tag{6}$$

   Here, the notation $[\cdot;\cdot]$ implies concatenation.

   To implement the additive attention mechanism described above, fill in the `forward` method of the `AdditiveAttention` class. Use `self.softmax` function in the forward pass of the `AdditiveAttention` class to normalize the weights.

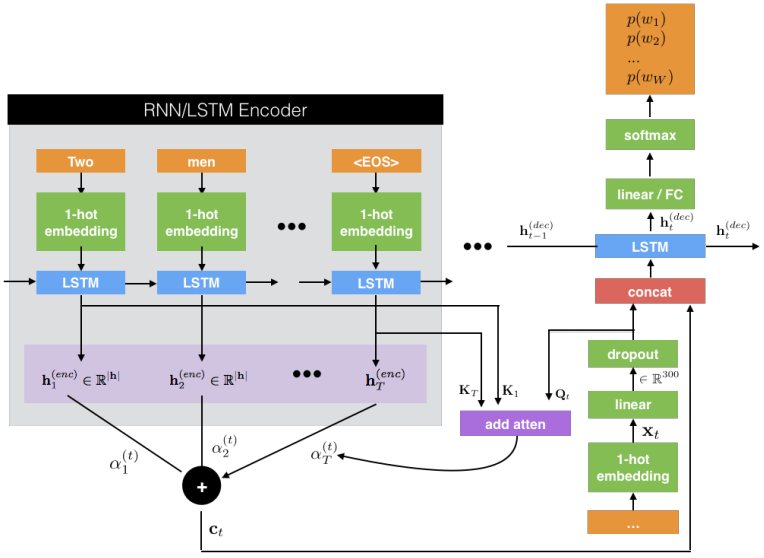
   Note, the code is setup to do potentially use batching, but we will only work with a simpler variant that **only** handles `batch_size = 1`. Hence, while in general, the query ($\mathbf{Q}$) would be a tensor of size `batch_size` $\times$ `hidden_size`, we will only deal with `1` $\times$ `hidden_size`, where `hidden_size = 300`. Similarly, keys/values will in general be the same sized tensors of dimensions `batch_size` $\times$ `seq_length` $\times$ `hidden_size`, but we will only deal with `1` $\times$ `seq_length` $\times$ `hidden_size`.

   The `AdditiveAttention` module should return both the context vector `batch_size` $\times$ `1` $\times$ `hidden_size` and the normalized attention weights `batch_size` $\times$ `seq_len` $\times$ `1`. Again, we will only be dealing with batch size equal to 1, which implies that the output should simply be two vectors, one the size of the encoder hidden state (context vector) and one the length of the input sequence (attention weights).

   We have provided some pseudo-code for `AdditiveAttention` class. You are free to use it, or code it from scratch, as long as the output is correct. You are also free to implement this with batching support if you so wish, but not required to do so.

2. **Implementing Attention Decoder.** Now we will apply the `AdditiveAttention` module to the RNN decoder (which uses LSTM). You will do this by filling in the `forward` method of the `AttentionDecoder` class, to implement the interface shown below by setting:

$$\begin{aligned} \mathbf{Q}_t &\leftarrow [\hat{\mathbf{x}}_t^{(dec)}] \quad \text{alternatively you can do } \mathbf{Q}_t \leftarrow [\mathbf{h}_{t-1}^{(dec)}] \\ \mathbf{K} &\leftarrow [\mathbf{h}_1^{(enc)}, \mathbf{h}_2^{(enc)}, \cdots, \mathbf{h}_T^{(enc)}] \\ \mathbf{V} &\leftarrow [\mathbf{h}_1^{(enc)}, \mathbf{h}_2^{(enc)}, \cdots, \mathbf{h}_T^{(enc)}] \end{aligned}$$



To do this you will need to fill in the `forward` method of the `AttentionDecoder` class, by doing the following:

   (a) Compute the context vector and the attention weights using `self.attention`.

   (b) Concatenate the context vector with the current decoder input.

   (c) Feed the concatenation to the decoder LSTM cell to obtain the new hidden state.

3. **Training.** You will need to train the new model with teacher forcing similar to Step 3, Part 1 of the assignment. This will require updating of the `train()` method to handle `AttentionLSTM`.

4. **Testing.** Now redo Step 6 of Part 1 to test the model and produce an BLEU score. This will require updating of the `inference()` method to handle `AttentionLSTM`. Note that you should see a substantial improvement in performance here.

5. **Visualizing Attention.** One of the many benefits of attention is that it allows us to gain some insight into the model. By visualizing the attention weights generated for the input tokens in each decoder step, we can see where the model focuses while producing each output token. We are already providing the code for you to visualize the attention. Use this code to visualize attention for translation of two sentences. Comment on the visualized attention.

Part 3: Translation with Transformer Decoder

One of the disadvantages of the attention decoder implemented in the last section is that while it can, in-principle, focus on any encoder token(s) when generating a token in the decoder, it has limited ability to

keep track of the generated tokens themselves. The reason is that those are only encoded in the hidden state that is recursively passed from a decoder step to the next. Transformer architecture alleviates this constraint by allowing the decoder to focus on both the encoded token representations and the decoded ones.

1. **Implementing Scaled Dot-product Attention.** For this part of the assignment, we will implement a slightly more sophisticated version of attention. In this variant, the function f is a dot product similarity between the linearly transformed / embedded query and keys vectors. Specifically,

$$\beta_i^{(t)} = \frac{(\mathbf{W}_q \mathbf{Q}_t)^T (\mathbf{W}_k \mathbf{K}_i)}{\sqrt{d}} \tag{7}$$

   where $\mathbf{W}_q$ and $\mathbf{W}_k$ are embedding/projection weight matrices that result in the same dimensional vectors over which dot product is computed (numerator). The d is the dimension of the query. Similarly to earlier attention variant, we normalize weights to sum up to 1 using softmax:

$$\alpha^{(t)} = \text{softmax}(\beta^{(t)}) \tag{8}$$

   and obtain the final context vector:

$$\mathbf{c}_t = \sum_{i=1}^{T} \alpha_i^{(t)} \mathbf{W}_v \mathbf{V}_i. \tag{9}$$

   by accumulating transformed value vectors, transformed by learned matrix $\mathbf{W}_v$.

   To implement this in the assignment, fill in the `__init__` and forward methods of the `ScaledDotAttention` class. Your forward pass needs to work with both 1D query tensor (`batch_size` × (1) × `hidden_size`) and 2D query tensor (`batch_size` × `k` × `hidden_size`) (remember that function only needs to support `batch_size = 1`.

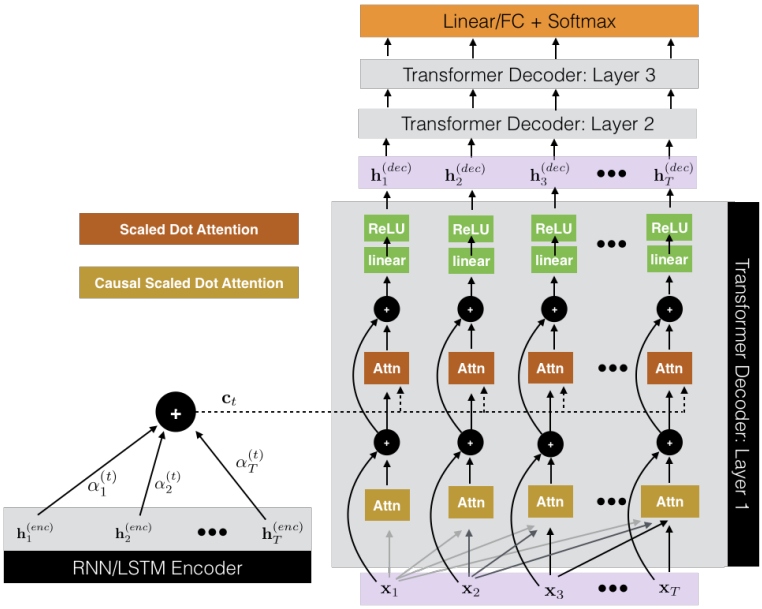
   Note that this attention can be used as a seamless replacement for Additive Attention in Part 2 and this is not a bad way to test out that you implemented the function correctly. However, this is not required for the assignment.

2. **Implementing Causal Scaled Dot-product Attention.** We will implement another version of the Scaled Dot-Product Attention, that will be helpful in attending over the decoder hidden states. To do so, fill in the `forward` method in the `CausalScaledDotAttention`. It will be very similar to the `ScaledDotAttention` class. The only additional computation that is needed is to mask out the attention to the future decoder steps. You will need to add `self.neg_inf` to some of the entries in the unnormalized attention weights.

3. **Implementing Simplified Transformer.** You will now use `ScaledDotAttention` as the building blocks for a simplified transformer [2] decoder. Consider being given a batch of decoder input embeddings, $\mathbf{X}^{(dec)} = [\mathbf{x}_1^{(dec)}, \mathbf{x}_2^{(dec)}, \cdots]$ across all time steps, which has dimension `batch_size` × `decoder_seq_len` × `hidden_size` and a batch of encoder hidden states, $\mathbf{H}^{(enc)} = [\mathbf{h}_1^{(enc)}, \cdots, \mathbf{h}_T^{(enc)}]$, for each time step in the input sequence, which has dimension `batch_size` × `encoder_seq_len` × `hidden_size`. Again, recall that we will keep `batch_size = 1`.

   Note that while in training (assuming teacher forcing), we have access to all $\mathbf{X}^{(dec)}$. However, when testing, we will incrementally generate portions of $\mathbf{X}^{(dec)}$ tensor.

   The transformer solves the translation problem using layers of attention modules. In each layer, we first apply the `CausalScaledDotAttention` self-attention to the decoder inputs followed by `ScaledDotAttention` attention module to the encoder hidden staters, similar to the attention decoder from Part 2. The output of the attention layers are fed into an hidden layer using ReLU activation. The final output of the last transformer layer are used to compute the word prediction. To improve the optimization, we add residual connections between the attention layers and ReLU layers. The illustration of the architecture you will implement is above.



4. **Training.** You will need to train the new model with teacher forcing similar to Step 3, Part 1 of the assignment. This will require updating of the `train()` method to handle `Transofmer`.

5. **Testing.** Now redo Step 6 of Part 1 to test the model and produce an BLEU score. This will require updating of the `inference()` method to handle `Transformer`. Note that you should see a substantial improvement in performance here.

6. **Visualizing Attention.** Visualize attention for the transformer. Given that there are multiple layers of attention, you will need to visualize the attention for each layer. Visualize attention for 3 sentence translations of your choosing.

References

[1] Samy Bengio, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. Scheduled sampling for sequence prediction with recurrent neural networks. *Advances in Neural Information Processing Systems*, pages 1171–1179, 2015.

[2] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in Neural Information Processing Systems*, pages 5998–6008, 2017.