Homework 3: Part I
RNNs and Sequence to Sequence Models[1]

CMU 10-417/617: Intermediate Deep Learning (Fall 2024)

https://eshau.github.io/10417-24/
OUT: Tuesday, Oct 29th, 2024

DUE: Tuesday, Nov 12th, 2024, 11:59pm
TAs: Raunak Sood, Kaiwen Geng

START HERE: Instructions

Homework 3 covers topics on sequence-to-sequence models and Restricted Boltzmann Machines. The homework includes short answer questions, derivation questions, and a coding task.

- **Collaboration policy:** Collaboration on solving the homework is allowed, after you have thought about the problems on your own. It is also OK to get clarification (but not solutions) from books or online resources, again after you have thought about the problems on your own. There are two requirements: first, cite your collaborators fully and completely (e.g., "Jane explained to me what is asked in Question 2.1"). Second, write your solution *independently*: close the book and all of your notes, and send collaborators out of the room, so that the solution comes from you only. See the Academic Integrity Section on the course site for more information: https://eshau.github.io/10417-24/
- **Late Submission Policy:** See the late submission policy here: https://eshau.github.io/10417-24/
- **Submitting your work:**
  - **Written:** For both the programming portion and the written problems, we will be using Gradescope (https://gradescope.com/). For the programming portion, please submit your filled out "transformer.py" file to Gradescope under the assignment name "Homework 3 Part 1 Programming", and please submit your writeup to "Homework 3 Written". Please write your solution in the LaTeX files provided in the assignment and submit in a PDF form. **Handwritten solutions are accepted, but will receive zero credit if illegible.** Put your answers in the question boxes (between \begin{soln} and \end{soln}) below each problem. Please make sure you complete your answers within the given size of the question boxes. **We reserve the right to remove points if the template is not followed**. Regrade requests can be made, however this gives the TA the opportunity to regrade your entire paper, meaning if additional mistakes are found then points will be deducted. For more information about how to submit your assignment, see the following tutorial (note that even though the assignment in the tutorial is handwritten, submissions must be typed): https://www.youtube.com/watch?v=KMPoby5g_nE&feature=youtu.be
  - **Code:** All code must be submitted to Gradescope. Please submit `transformer.py` to Gradescope. **If you do not submit your code to Gradescope, you will not receive any credit for your assignment.** Gradescope will be used to check for plagiarism. Please make sure your familiarize yourself with the academic integrity information for this course.

[1]Compiled on Tuesday 29th October, 2024 at 09:59

1 Background: GRUs

One class of models for seq2seq learning is the (GRU), a type of recurrent neural network that has an "explicit memory" and is less vulnerable than vanilla RNNs to the vanishing gradient problem.

In general, a recurrent neural network is a function $\mathbf{h}_j = f(\mathbf{h}_{j-1}, \mathbf{x}_j)$ that consumes the *current input* $\mathbf{x}_j$ and the *previous hidden state* $\mathbf{h}_{j-1}$, and returns the *current hidden state* $\mathbf{h}_j$. In the most basic, "vanilla" type of RNN, the current hidden state is simply an affine function of the current input $\mathbf{x}_j$ and the previous hidden state $\mathbf{h}_{j-1}$, passed through a nonlinearity:

$$\mathbf{h}_j = \tanh(\mathbf{W}\mathbf{x}_j + \mathbf{U}\mathbf{h}_{j-1})$$

Unfortunately, due to the vanishing and exploding gradient problems that you will soon see below, it's difficult to learn long-term dependencies with a vanilla RNN. Intuitively, the "default" behavior of the vanilla RNN is to completely replace the hidden state at each time step; in contrast, a better inductive bias would be to retain most of the hidden state at each time step, carefully erasing and adding new information as appropriate. This is precisely what a GRU does.

In this assignment we adopt the original definition of the GRU from [Cho et al., 2014], in which a GRU is defined by the following update rules:

$$\begin{aligned}
\mathbf{r}_j &= \sigma(\mathbf{W}_r\mathbf{x}_j + \mathbf{U}_r\mathbf{h}_{j-1}) \\
\mathbf{z}_j &= \sigma(\mathbf{W}_z\mathbf{x}_j + \mathbf{U}_z\mathbf{h}_{j-1}) \\
\tilde{\mathbf{h}}_j &= \tanh(\mathbf{W}\mathbf{x}_j + \mathbf{U}(\mathbf{r}_j \circ \mathbf{h}_{j-1})) \\
\mathbf{h}_j &= (\mathbf{1} - \mathbf{z}_j) \circ \tilde{\mathbf{h}}_j + \mathbf{z}_j \circ \mathbf{h}_{j-1}
\end{aligned}$$

Here, σ is the sigmoid nonlinearity $\sigma(x) = 1/(1 + \exp(-x))$, and $\circ$ denotes the element-wise multiplication of two vectors.

Let's unpack these update rules. First, you compute the *reset gate* $\mathbf{r}_j$ and the *update gate* $\mathbf{z}_j$ as linear functions of the current input and the previous hidden state, passed through the sigmoid nonlinearity. Both $\mathbf{r}_j$ and $\mathbf{z}_j$ are h-dimensional vectors with entries between 0 and 1, where h is the dimension of the GRU's hidden state. Then you compute the candidate hidden state $\tilde{\mathbf{h}}_j$ as a function of the current input $\mathbf{x}_j$ and a version of the previous hidden state, $\mathbf{r}_j \circ \mathbf{h}_{j-1}$, where some entries of $\mathbf{h}_{j-1}$ have been "reset" via being scaled with the entries in $\mathbf{r}_j$. Finally, you compute the current hidden state $\mathbf{h}_j$ as an interpolation between the prior hidden state $\mathbf{h}_{j-1}$ and the candidate hidden state $\tilde{\mathbf{h}}_j$, where the weights of the interpolation are given by the update gate $\mathbf{z}_j$.

To summarize: The reset gate $\mathbf{r}_j$ controls whether the information from the old hidden state is erased or retained when computing the candidate hidden state. The update gate $\mathbf{z}_j$ controls whether the old hidden state is retained or replaced by the candidate hidden state $\tilde{\mathbf{h}}_j$ when computing the new hidden state $\mathbf{h}_j$.

2 Problem 1 - Vanishing/Exploding Gradients (15 points)

Consider the following recurrent neural network (RNN), which maps a length-m sequence $\{\mathbf{x}_1, \ldots, \mathbf{x}_m\}$ of vectors in $\mathbb{R}^d$ to a single vector $\hat{\mathbf{y}}$ in $\mathbb{R}^k$.

$$\begin{aligned} \mathbf{h}_0 &= \mathbf{0} \\ \mathbf{h}_j &= f(\mathbf{W}\mathbf{x}_j + \mathbf{U}\mathbf{h}_{j-1}) \quad j = 1 \ldots m \\ \hat{\mathbf{y}} &= \mathbf{V}\,\mathbf{h}_m \end{aligned}$$

Here, $\mathbf{W} \in \mathbb{R}^{h \times d}$ are input weights, $\mathbf{U} \in \mathbb{R}^{h \times h}$ are recurrent weights, and $\mathbf{V} \in \mathbb{R}^{k \times h}$ are output weights. The vector of hidden units $\mathbf{h}_j$ has dimension h. Assume that f is an element-wise activation function.

Here's the unrolled computational graph of this RNN:

$$\begin{tikzcd}
\mathbf{0} \arrow[r] & \mathbf{h}_1 \arrow[r, "\mathbf{U}"] & \mathbf{h}_2 \arrow[r, "\mathbf{U}"] & \mathbf{h}_3 \arrow[r, "\mathbf{U}"] & \cdots \arrow[r, "\mathbf{U}"] & \mathbf{h}_m \arrow[r, "\mathbf{V}"] & \hat{\mathbf{y}} \\
& \mathbf{x}_1 \arrow[u, "\mathbf{W}"] & \mathbf{x}_2 \arrow[u, "\mathbf{W}"] & \mathbf{x}_3 \arrow[u, "\mathbf{W}"] & & \mathbf{x}_m \arrow[u, "\mathbf{W}"] &
\end{tikzcd}$$

In this problem, we are interested in the quantity $\frac{\partial \mathbf{h}_m}{\partial \mathbf{h}_1}$, a $h \times h$ Jacobian matrix which contains the partial derivatives of the coordinates of $\mathbf{h}_m$ with respect to the coordinates of $\mathbf{h}_1$.

$$\left(\frac{\partial \mathbf{h}_m}{\partial \mathbf{h}_1}\right)_{a,b} = \frac{\partial h_{ma}}{\partial h_{1b}}$$

The "vanishing gradient" problem arises when $\frac{\partial \mathbf{h}_m}{\partial \mathbf{h}_1}$ is small in norm, which means that time step 1 has negligible effect on the gradient $\nabla_{\mathbf{U}} L$. The "exploding gradient" problem arises when $\frac{\partial \mathbf{h}_m}{\partial \mathbf{h}_1}$ is large in norm, which means that time step 1 has an outsize effect on the gradient $\nabla_{\mathbf{U}} L$.

1. **(2 points)** Consider the simplified setting where the input, hidden, and output dimensions are all one ($d = h = k = 1$). Note that in this simplified setting, the weight matrices $\mathbf{W}$, $\mathbf{U}$, and $\mathbf{V}$ are just scalars w, u, and v.

   It will be helpful to define z_j as the pre-activation at time j, so that $z_j = wx_j + uh_{j-1}$ and $h_j = f(z_j)$.

   Give an explicit expression for the derivative dh_m/dh_1. Your expression should be in terms of u, the z_j's, and f', the derivative of the activation function.

Solution

2. **(2 points)** Suppose that the activation function is $f(z) = \tanh(z)$. Give an upper bound on $|dh_m/dh_1|$ in terms of u and m.

   Hint: the derivative of tanh is bounded by 1 (to see this, google "plot tanh(x) from -2 to 2").

   This is an example of the vanishing gradient problem. For large m and certain values of u, the norm of the gradient will be tiny.

   Solution

3. **(2 points)** Now suppose that the activation function is the identity function.

   (a) Give an explicit expression for dh_m/dh_1 in terms of u and m.

   (b) What is the numerical value of dh_m/dh_1 when $u = 1.2$ and $m = 100$?

   (c) What is the numerical value of dh_m/dh_1 when $u = 0.9$ and $m = 100$?

   (d) What is the issue here?

   Solution

4. **(3 points)** We now return to the fully general case, where $\mathbf{W}$, $\mathbf{U}$, and $\mathbf{V}$ are matrices and f is a general activation function.

   Give an explicit expression for the Jacobian $\partial\mathbf{h}_m/\partial\mathbf{h}_1$. It will be helpful to define $\mathbf{z}_j$ as the pre-activation at time j, so that $\mathbf{z}_j = \mathbf{W}\mathbf{x}_j + \mathbf{U}\mathbf{h}_{j-1}$ and $\mathbf{h}_j = f(\mathbf{z}_j)$. Your expression for $\partial\mathbf{h}_m/\partial\mathbf{h}_1$ should be in terms of $\mathbf{U}$, the $\mathbf{z}_j$'s, and f', the derivative of the activation function.

   Notice that the chain rule applies to vector-to-vector functions too: if $\mathbf{c}$ is a function of $\mathbf{b}$, which in turn is a function of $\mathbf{a}$, then $\frac{\partial\mathbf{c}}{\partial\mathbf{a}} = \frac{\partial\mathbf{c}}{\partial\mathbf{b}}\frac{\partial\mathbf{b}}{\partial\mathbf{a}}$.

   Solution

5. **(3 points)** Assume that f is the tanh activation function. Give an upper bound on $\|\frac{\partial\mathbf{h}_m}{\partial\mathbf{h}_1}\|_2$ the ℓ_2-operator norm of the Jacobian $\frac{\partial\mathbf{h}_m}{\partial\mathbf{h}_1}$. (Note that the ℓ_2 operater norm of a matrix is different from the ℓ_2 norm of a vector.) Your upper bound should be in terms of $\|\mathbf{U}\|_2$ and m.

   You'll want to use the following facts:

   (a) The operator norm is submultiplicative, i.e. $\|\mathbf{AB}\|_2 \leq \|\mathbf{A}\|_2\|\mathbf{B}\|_2$ for any matrices $\mathbf{A}$ and $\mathbf{B}$.

   (b) If $\|\mathbf{A}\mathbf{x}\|_2 \leq c\|\mathbf{x}\|_2$ for all $\mathbf{x}$, then $\|\mathbf{A}\|_2 \leq c$.

   (c) The derivative of tanh is bounded in magnitude by 1, i.e. $|f'(z)| \leq 1$ for all z.

   Solution

6. **(3 points)** Now suppose that the RNN was instead a GRU, i.e.

$$\mathbf{h}_j = \text{GRU}(\mathbf{h}_{j-1}, \mathbf{x}_j)$$

Moreover, suppose that, at each time step j, all update gates are set to 1: $\mathbf{z}_j = \mathbf{1}$ for $j = 1 \ldots m$.

Give an expression for $\partial \mathbf{h}_m / \partial \mathbf{h}_1$.

(Hint: first think of a (very!) simplified expression for $\mathbf{h}_m$ as a function of $\mathbf{h}_1$, then differentiate.)

Solution

3 Problem 2 - Deriving PyTorch's GELU (15 points)

Gaussian Error Linear Unit (GELU) is an activation function that you'll often see being used in transformers such as BERT and GPT. It is computed as

$$\mathrm{GELU}(x) = x\Phi(x)$$

where Φ is the cumulative distribution function of a standard Gaussian. GELU is more non-linear than ReLU which can help it better model complex functions. However, it is less efficient to implement than other activation functions. To alleviate this inefficiency, PyTorch's implementation has an approximation argument, which allows you to specify it to use a tanh approximation to GELU. In this problem, we will derive where this approximation comes from.

1. **(3 points)** Show that $\mathrm{GELU}(x) = \frac{1}{2}x\left(1 + \mathrm{erf}\left(\frac{x}{\sqrt{2}}\right)\right)$ where

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}\,dt$$

Hint: The cumulative distribution function of the standard Gaussian normal is $\Phi(x) = \frac{1}{\sqrt{2\pi}}e^{-\frac{x^2}{2}}$

Solution

2. **(3 points)** Write out the first few terms of the Maclaurin series for tanh. Recall that

$$\tanh'(x) = 1 - \tanh(x)^2$$

Write out the series until the third order term. Simplify your answer as much as possible.

Solution

3. **(1 points)** The first few terms of the Maclaurin series for the erf function are:

$$\operatorname{erf}(x) \approx \frac{2}{\sqrt{\pi}}\left(x - \frac{x^3}{3}\right)$$

Using this, show that

$$\operatorname{erf}\left(\frac{x}{\sqrt{2}}\right) \approx \sqrt{\frac{2}{\pi}}\left(x - \frac{x^3}{6}\right)$$

Solution

The tanh approximation to GELU used by PyTorch is given by:

$$\text{GELU}(x) \approx \frac{x}{2}\left[1 + \tanh\left(\sqrt{\frac{2}{\pi}}(x + ax^3)\right)\right]$$

In the next step, we will solve for a and compare it to the one used in PyTorch.

4. **(4 points)** Recall from earlier that $\text{GELU}(x) = \frac{x}{2}\left(1 + \text{erf}(\frac{x}{\sqrt{2}})\right)$. Thus we want to find the value of a for which $\tanh\left(\sqrt{\frac{2}{\pi}}(x + ax^3)\right) \approx \text{erf}(\frac{x}{\sqrt{2}})$. Use the tanh approximation that you calculated earlier to find an approximation for $\tanh\left(\sqrt{\frac{2}{\pi}}(x + ax^3)\right)$ and solve for a.

   Hint: You can toss out all terms with higher order than 3. Think about why this is a decent approximation.

   Solution

5. **(1 points)** Take a look at the PyTorch documentation for GELU. If it is different than the value you calculated above, can you explain, in 1-2 sentences, why that might have occurred?

   Solution

6. **(3 points)** Plot GELU, ReLU, and the GELU approximation that you computed earlier and describe some of the major differences that you see between the GELUs and ReLU. Why might some of these differences be desirable?

Solution

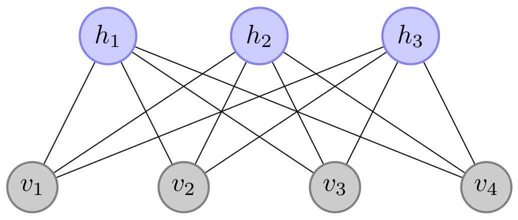
4 [10-617 ONLY] Problem 3 - Gradients in RBMs (10 points)

We learn that a deep belief network (DBN) as a generative graphical model, consists of multiple layers of latent variables with connections between the layers. One DBN can learn to probabilistically reconstruct its inputs and extract a deep hierarchical representation of the training data. DBNs can be viewed as a composition of simple, unsupervised networks such as restricted Boltzmann machines (RBMs), where each sub-network's hidden layer serves as the visible layer for the next.

In this problem, we will learn how RBMs and DBNs are trained.

4.1 Restricted Boltzmann Machine

In the RBM, visible units are conditionally independent on hidden units and vice versa.



.

For a given RBM, we relate the units with the energy function as follows:

$$Energy(\mathbf{v}, \mathbf{h}) = -\mathbf{b}^\top \mathbf{h} - \mathbf{c}^\top \mathbf{v} - \mathbf{h}^\top \mathbf{W} \mathbf{v}$$

where $\mathbf{b} \in \mathbb{R}^H, \mathbf{c} \in \mathbb{R}^V$ are offset/bias vectors and $\mathbf{W} \in \mathbb{R}^{H \times V}$ comprises the weights and connecting units.

The joint probability $P(\mathbf{v}, \mathbf{h})$ is presented as follows:

$$P(\mathbf{v}, \mathbf{h}) = \frac{1}{Z} \exp(-Energy(\mathbf{v}, \mathbf{h}))$$

where Z is the normalization term.

4.2 Conditional probabilities (2 pts)

Assuming $\mathbf{v}$ and $\mathbf{h}$ are binary units, show $P(v_i = 1|\mathbf{h})$ and $P(h_j = 1|\mathbf{v})$ are presented as follows:

$$P(v_i = 1|\mathbf{h}) = \sigma(c_i + \mathbf{h}^\top \mathbf{W}_{:,\mathbf{i}})$$

$$P(h_j = 1|\mathbf{v}) = \sigma(b_j + \mathbf{W}_{\mathbf{j},:} \mathbf{v})$$

where $\sigma(x) = \frac{1}{1+e^{-x}}$.

Solution

4.3 Free energy (2 pts)

We obtain $P(\mathbf{v})$ by marginalizing $\mathbf{h}$ as follows:

$$P(\mathbf{v}) = \frac{\sum_{\mathbf{h}\in\{0,1\}^H} \exp(-Energy(\mathbf{v},\mathbf{h}))}{Z} = \frac{\exp(-FreeEnergy(\mathbf{v}))}{Z}$$

where $FreeEnergy(\mathbf{v})$ form makes it easier to compute gradients with visible units only.

We rewrite the energy function as follows:

$$Energy(\mathbf{v},\mathbf{h}) = -\beta(\mathbf{v}) - \sum_{j=1}^{H} \gamma_j(\mathbf{v}, h_j)$$

Show $P(\mathbf{v})$ and $FreeEnergy(\mathbf{v})$ could be presented as follows (prove **all** 3 statements):

$$P(\mathbf{v}) = \frac{\exp(\beta(\mathbf{v}))}{Z} \prod_{j=1}^{H} \sum_{h_j\in\{0,1\}} \exp(\gamma_j(\mathbf{v}, h_j))$$

$$FreeEnergy(\mathbf{v}) = -\beta(\mathbf{v}) - \sum_{j=1}^{H} \log \sum_{h_j\in\{0,1\}} \exp(\gamma_j(\mathbf{v}, h_j))$$

$$FreeEnergy(\mathbf{v}) = -\mathbf{c}^\top\mathbf{v} - \sum_{j=1}^{H} \log(1 + \exp(b_j + \mathbf{W_{j,:}}\mathbf{v}))$$

Solution

4.4 Log-likelihood gradient (3 pts)

Show log-likelihood gradient of $P(\mathbf{v})$ could be presented as follows:

$$\frac{\partial \log P(\mathbf{v})}{\partial \theta} = -\frac{\partial FreeEnergy(\mathbf{v})}{\partial \theta} + \sum_{\tilde{v} \in \{0,1\}^V} P(\tilde{\mathbf{v}}) \frac{\partial FreeEnergy(\tilde{\mathbf{v}})}{\partial \theta}$$

Conceptually, we can think of $-\frac{\partial FreeEnergy(\mathbf{v})}{\partial \theta}$ as the "Positive Phase" and $\sum_{\tilde{v} \in \{0,1\}^V} P(\tilde{\mathbf{v}}) \frac{\partial FreeEnergy(\tilde{\mathbf{v}})}{\partial \theta}$ as the "Negative Phase". (Hint: Present $\frac{\partial \log Z}{\partial \theta}$ in terms of $FreeEnergy(\mathbf{v})$ and $P(\mathbf{v})$.)

Solution

4.5 Gradients w.r.t. variables (3 pts)

Show that the gradients w.r.t. each variable $\mathbf{W}, \mathbf{b}, \mathbf{c}$ could be presented as follows:

$$-\frac{\partial \log P(\mathbf{v})}{\partial W_{ji}} = -P(h_j = 1|\mathbf{v}) \cdot v_i + \mathbb{E}_{\tilde{\mathbf{v}}}[P(\tilde{h}_j = 1|\tilde{\mathbf{v}}) \cdot \tilde{v}_i)]$$
$$-\frac{\partial \log P(\mathbf{v})}{\partial b_j} = -P(h_j = 1|\mathbf{v}) + \mathbb{E}_{\tilde{\mathbf{v}}}[P(\tilde{h}_j = 1|\tilde{\mathbf{v}})]$$
$$-\frac{\partial \log P(\mathbf{v})}{\partial c_i} = -v_i + \mathbb{E}_{\tilde{\mathbf{v}}}[\tilde{v}_i]$$

Hint: Use the results from 4.2.

Solution

5 Programming - Implementing Transformers (60 points)

In this question, you'll be implementing a sequence-to-sequence transformer model for English-to-French translation. Transformers are the current state of the art in many NLP tasks, and the paradigm of sequence modeling is now being extended to visual question answering, image generation, and even reinforcement learning, so the goal of this assignment is to give you a better understanding of how transformers work and their generality.

Task The task is to translate sentences from English to French. In the parlance of machine translation, English is the *source language* and French is the *target language*. Training a neural machine translation model on a real-world dataset would take days and would require multiple GPUs. Therefore, this assignment will use a simpler translation dataset, which we borrowed from `https://github.com/spro/practical-pytorch`.

The dataset consists of (English sentence, French sentence) pairs, 8824 for training and 1000 for testing. Here are some randomly drawn test examples:

English	French
I'll find a way to do it.	Je trouverai un moyen de faire ceci.
Don't monologue !	Ne monologue pas !
They speak Spanish.	Ils parlent l'espagnol.
What is this thing for?	À quoi sert cette chose ?
I get on very well with your cousin.	Je m'entends très bien avec ton cousin.
It is very likely.	C'est très probable.

References We've tried to make this assignment self-contained. Nevertheless, if anything is unclear, you might find the following references helpful:

- [Vaswani et al., 2017] introduced the transformer architecture as an alternative to previous sequence-to-sequence models, which primarily relied on variants of recurrent neural networks.
- The [Luong et al., 2017] paper, "Effective approaches to attention-based neural machine translation," gives an introduction to neural machine translation with attention-based recurrent neural networks, which can be seen as somewhat of an intermediate step between the earlier purely recurrent models, and modern fully attentional models.
- Graham Neubig's tutoral [Neubig, 2017] on neural machine translation, especially pp. 29-35 on RNNs, pp. 39-41 on seq2seq, greedy 1-best search, and beam search, and pp. 49-52 on attention.
- The TensorFlow NMT tutorial `https://github.com/tensorflow/nmt` has very nice diagrams, especially Figure 5.

Note that these references will use different notation from ours.

5.1 Problem Statement

We'll begin by discussing the general problem that machine translation is trying to solve, as well as how recurrent neural networks, which have been introduced to you in greater detail, have been used in this capacity. **If you have a good understanding of seq2seq RNNs, feel free to skip to the transformers section (5.2). However, we do recommend reviewing the sections on maximum-likelihood training of RNNs, beam search, and perplexity, as we will be asking you to implement those concepts.**

Machine Translation We'll now describe the machine translation problem in detail. Each train/test example consists of a paired source sentence $(x_1, \dots, x_S)$ and target sentence $(y_1, \dots, y_T)$. S is the length of the source sentence and T is the length of the target sentence. We'll use the letter s to index the source sentence, and t to index the target sentence.

The whole dataset consists of N source/target sentence pairs. Each sentence has been preprocessed so that the first word is a special start-of-sentence (SOS) token and final word is a special end-of-sentence (EOS) token. Sentences are also padded to have equal length.

A seq2seq architecture consists of several components:

1. Two networks called the *encoder* and the *decoder*. In this assignment, both the encoder f^{enc} and decoder f^{dec} will be deep transformers, but past approaches have also used stacked GRUs and LSTMs.
2. Word embeddings in both the source language and the target language (these can either be learned or provided beforehand).
3. An output layer, which produces a probability distribution over words in the vocabulary.

We'll slightly abuse notation by using $\mathbf{x}_s$ to denote the source language embedding for the source word x_s, and $\mathbf{y}_t$ to denote the target language embedding for the target word y_t.

Seq2Seq A seq2seq model is a neural network which takes as input both a source sentence $x_1 \ldots x_S$ and a target sentence $y_1 \ldots y_T$, and returns the conditional probability of the target sentence given the source sentence, $p(y_1 \ldots y_T | x_1 \ldots x_S)$. You can use a trained seq2seq model to do translations: given a source sentence $x_1 \ldots x_S$, search for the target sentence $y_1 \ldots y_T$ with the highest probability under the model.

For any paired source/target sentences $(x_1 \ldots x_S,\ y_1 \ldots y_T)$, the conditional probability of the target given the source can be written via the chain rule of probability as:

$$p(y_1 \ldots y_T | x_1 \ldots x_S) = \prod_{t=1}^{T} p(y_t | y_1 \ldots y_{t-1}, x_1 \ldots x_S)$$

In a seq2seq model, these T probabilities are computed consecutively in a single left-to-right pass. Specifically, we first pass in the source target sequence to the encoder, and the model typically generates some compact representation of the sentence. For example, in an RNN, we would have a hidden vector along with the context vectors in an attentional model. As you'll see below, in a transformer, we'll have several sets of keys and queries at each layer of the encoder.

Then, these representations are used by the decoder to autoregressively perform inference. At the first timestep, we pass in the context from the encoder, and at the output layer, the decoder produces a probability distribution over the first word in the target sequence using a softmax over the logits.

Maximum likelihood training You train a seq2seq model by maximizing the conditional log-likelihood of the training dataset wrt the model weights. The conditional log-likelihood of the training dataset is simply the sum of the conditional log-probability of each (source sentence, target sentence) pair.

Attention Here, we'll briefly describe the attention mechanism in standard RNNs, as a similar mechanism is used in transformers.

One problem in RNNs without attention is that in a seq2seq model, the only information the decoder gets about the source sentence is the hidden state of the encoder after it has ingested the entire sequence - a potentially low-dimensional representation compared to the length of the sequence. Since the decoder may need to "focus" more on certain parts of the source sentence when producing a given word in the target sequence, compressing the entire sentence into the hidden state of the encoder can be inefficient. Moreover, since a given word in the beginning of the source sentence only affects a word in the target sequence through several passes of the RNN, this often leads to unstable gradient-based training.

To ameliorate this, the attention mechanism was introduced. Given the hidden states of the encoder at each time step $[\mathbf{h}_1, \cdots, \mathbf{h}_S]$, the decoder tries to find the hidden state that provides the most information about generating the next word at timestep t. To do so, the decoder produces a query vector $\mathbf{q}_t$ and tries to find the hidden state that best aligns with this query using a score function $\text{score}(\mathbf{q}_t, \mathbf{h}_s) \triangleq \mathbf{q}_t^T \mathbf{A} \mathbf{h}_s$. However, since the argmax operation is non-differentiable, we use a softmax. Specifically, the attention weights $\boldsymbol{\alpha}_t$ are calculated as

$$\boldsymbol{\alpha}_{t,s} = \frac{\exp(\text{score}(\mathbf{q}_t, \mathbf{h}_s))}{\sum_{s'} \exp(\text{score}(\mathbf{q}_t, \mathbf{h}_s))}.$$

The context vector $\mathbf{c}_t$ of information from the source sentence is then calculated as a linear combination of the $[\mathbf{h}_1, \cdots \mathbf{h}_S]$, where the weights are the attention scores. This context vector is used in the decoder, along with the decoder's current hidden state $\mathbf{h}_t$ to calculate the next hidden state, as well as to generate the word of the target sentence at time step t.

Greedy decoding At test time, we are given a source sentence $x_1 \dots x_S$, and we need to translate it into the target language. The most principled thing to do would be to return the target sentence with the highest conditional probability, i.e. $\text{argmax}_{y_1 \dots y_T} p(y_1 \dots y_T | x_1 \dots x_S)$. However, this would be expensive to compute. A computationally cheaper alternative is *greedy decoding*.

First you encode the source sentence and run the decoder for one step to compute $p(y_1|x_1 \dots x_S)$, a distribution (vector) over the target vocabulary words. Then you predict the first word to be the word with the highest probability in $p(y_1|x_1 \dots x_S)$ — call this word $\hat{y}_1$. Next, you run the decoder for another step to compute $p(y_2|x_1 \dots x_S, \hat{y}_1)$, and you predict the second word to be the word $\hat{y}_2$ with the highest probability in $p(y_2|x_1 \dots x_S, \hat{y}_1)$. More generally, you predict the t-th word to be the word $\hat{y}_t$ with the highest probability in $p(y_t|x_1 \dots x_S, \hat{y}_1 \dots \hat{y}_{t-1})$. This process continues until $\hat{y}_t$ is the EOS token, at which point you return $\hat{y}_1 \dots \hat{y}_t$ as the predicted sentence.

In order to avoid a situation where a (poorly trained or misimplemented) model keeps outputting words without ever outputting the EOS token, it helps to have a cutoff parameter `MAX_LENGTH`. Once greedy decoding has outputted `MAX_LENGTH` words, then translation stops regardless of whether or not the last one was the EOS token.

Beam search Previously, we generated a translation using "greedy decoding," in which we compute the probability distribution $p(y_t|x_1 \dots x_S, y_1 \dots y_{t-1})$ at time step t, and use the word with the highest score in this distribution as the next word in our translation. While simple to implement, greedy decoding is not guaranteed to find the target sentence with the highest conditional probability, given the source sentence, under the model.

Beam search is one way to alleviate this problem. Instead of greedily choosing the most likely next word, beam search expands all possible next words, and keeps the B most likely ones, where B is the width of the beam.

In decoding process, beam search maintains a list of B candidate translations at each time step (except for the first time step when it has not produced any translation). Let $y_{j,b}$ denote the target word for time j contained in the b-th candidate translation ($b = 1 \dots B$). At time t, for all candidate translations in the beam, we compute the conditional log-likelihood $\log p(y_{1,b}, \dots, y_{t-1,b}, y_t | x_1, \dots, x_S)$ for every word y_t in the target vocabulary. In other words, starting with B candidate translations, we temporarily generate $B\times$`target_vocab_size` translations, among which we pick the top B translations with the highest conditional log-likelihood. This process of calculating the log-likelihood for $B\times$`target_vocab_size` continuations of the current B translations, and then pruning back down to the top B is continued until the search terminates. Figure 1 shows an example beam search with beam width $B = 2$ given a vocabulary of $\{\text{a}, \text{b}\}$.

One thing to consider in using beam search is that, with the following conditional log-likelihood of a translation $y_1, \dots, y_t$ given a source sentence $x_1, \dots, x_S$,

$$\log(y_1, \dots, y_t | x_1, \dots, x_S) = \sum_{j=1}^{t} \log p(y_j | x_1, \dots, x_S, y_1, \dots, y_{j-1}),$$

beam search tends to prefer shorter sentences. This is because each time we append a new target word to a translation, another negative term is added, which decreases the above log-likelihood for the resulting translation. To counteract this length bias towards shorter translations, we will use a heuristic that normalizes

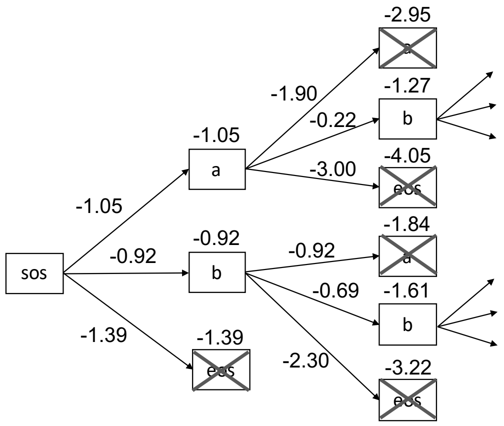


Figure 1: An example beam search with beam width $B = 2$ given a vocabulary of $\{a, b\}$ [Neubig, 2017]. Numbers next to arrows are conditional log-likelihood for a single word, $\log p(y_t|x_1, \ldots, x_S, y_1, \ldots, y_{t-1})$, and numbers above words are conditional log-likelihood for a sequence of words, $\log p(y_1, \ldots, y_t|x_1, \ldots, x_S)$.

the log-likelihood by the length of a translation. With length normalization, the criterion for selecting the top candidate now becomes identifying a translation that has the highest average log-likelihood per word, i.e., $\frac{1}{t} \log p(y_{1,b}, \ldots, y_{t-1,b}, y_t|x_1, \ldots, x_S)$.

Whenever an EOS token is selected among the highest scoring candidates, the beam width is reduced by one, and the corresponding translation is stored into a list of final candidates. When the beam width becomes zero, the search process stops. The search also stops if the length of a translation becomes MAX_LENGTH as in greedy decoding. All remaining candidates in the beam at this point, if any, are also put into the final candidate list. Then, among B final candidates, the translation with the highest normalized log-likelihood is selected as the final translation.

BLEU Score (10-617 ONLY) The BLEU score is a popular metric to measure the quality of a machine translation model. For a given source sentence, there may be a number of reasonably correct target translations, so we would like the generated translation to match with a large number of the targets.

First, we introduce the concept of an n-gram. An n-gram is a contiguous substring of size n: a sentence of length S will have $S - n + 1$ total n-grams. As an example, the 3-grams in the sentence "[I, went, to, the, store]" are "[(I, went, to), (went, to, the), (to, the, store)]. To measure the correctness of the generated sentence, we will use a measure called clipped precision.

Consider the case of single-sentence target y and single-sentence prediction $\hat{y}$, which is what you will be implementing. Suppose the value of n is fixed/given. We first calculate the number of times each n-gram from the predicted sequence appears in the target sentence and sum these values. For example, if the target sentence is "I went to the store" and the predicted sentence is "I went to the shop," the 1-gram "I" appears once in the target sentence. We then divide by the total number of n-grams in the predicted sentence. However, this creates an incentive for the model, for example, to predict "store store store store store," since this would give a score of 1.0, the max achievable. Rather than calculating the number of times each n-gram appears in the target sentence, we instead calculate the number of times each **unique** n-gram appears in the target sentence, and take the minimum of this count with the number of times it appears in the predicted sentence. With this modification, the score for the sentence "store store store store store" would be 0.2.

More formally, given a value of n, we define the clipped precision score for n as

$$P_n(y, \hat{y}) \triangleq \frac{\sum_{x \in \{\text{unique n-grams in } \hat{y}\}} \min(C(\hat{y}, x), C(y, x))}{|\hat{y}| - n + 1}$$

where $C(y, x)$ is the number of times x appears in y and $C(\hat{y}, x')$ is the number of times it appears in $\hat{y}$.

To compute the BLEU score, we fix the maximimum number of n-grams we are considering. We denote the BLEU-k score to be the score if we consider all n-grams up to size k. Then, we compute the weighted geometric mean of the BLEU-scores, where the weights are typically uniform:

$$\prod_{n=1}^{k} P_n(y, \hat{y})^{1/k}.$$

The last consideration is that of sentence length. If we were to predict just one word correctly by generating a one-word sentence, the score would be 1. To remedy against this, we introduce a brevity penalty, which is $\min(1, \exp\left(1 - \frac{|y|}{|\hat{y}|}\right))$. The BLEU-$k$ score is then the product of the brevity penalty and the weighted geometric mean.

5.2 Transformers

Motivation and General Architecture Transformers aim to address a number of the deficiencies of seq2seq RNNs. In particular, while the decoder can search for information in the encoder's hidden weights, the encoder itself still relies on a recurrent structure, where information between two words in a T-word sentence can require $O(T)$ connections to be propagated. Due to this, training deep RNNs is still a poorly-conditioned optimization problem. Transformers address this issue by introducing a key mechanism called self-attention. First, in the encoder, the words are represented as a sequence of tokens through the input

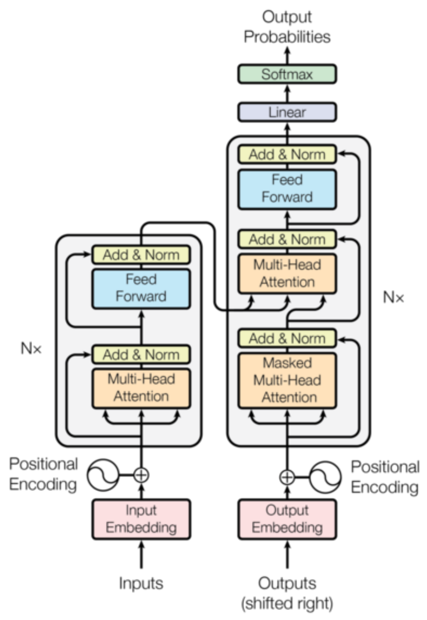


Figure 2: The transformer architecture consists of an encoder and a decoder, which are composed of transformer layers. Self-attention is used in the encoder and decoder to establish dependencies in the source and target, and standard attention is used to allow the decoder to look at the encoder's embeddings.

embedding $[\mathbf{x}_1^0, \mathbf{x}_2^0, \cdots, \mathbf{x}_T^0]$ where each $\mathbf{x}_t^0 \in \mathbb{R}^d$. A positional encoding $[\mathbf{p}_1, \mathbf{p}_2, \cdots, \mathbf{p}_T]$ is added to each token. Then, this sequence of tokens is "transformed" into a sequence of tokens $[\mathbf{a}_1^0, \mathbf{a}_2^0, \cdots, \mathbf{a}_T^0]$ via the multi-head self attention mechanism, which we will specify in greater detail in a later section. Then, for each timestep t, we compute $\mathbf{b}_t^0 = \text{LayerNorm}(\mathbf{x}_t^0 + \text{Dropout}(\mathbf{a}_t^0))$. This is analogous to the skip connections in ResNets, and the LayerNorm module performs an analogue of BatchNorm for seq2seq models.

After the attention module, the values $\mathbf{b}_t^0$ are passed into a position-wise feedforward network - an MLP applied to each token. Another skip-connection and LayerNorm are used to compute $\mathbf{c}_t^0 = \text{LayerNorm}(\mathbf{b}_t^0 + \text{Dropout}(\mathbf{b}_t^0))$. These values $\mathbf{c}_t^0$ are used as the context vectors for the first layer in the decoder. This process corresponds to one layer of the transformer, which is repeated. More generally, at layer L, the transformer converts the previous sequence of tokens $[\mathbf{c}_1^{L-1}, \mathbf{c}_2^{L-1}, \cdots, \mathbf{c}_T^{L-1}]$ into $[\mathbf{c}_1^{L}, \mathbf{c}_2^{L}, \cdots, \mathbf{c}_T^{L}]$, which, as mentioned before, are used as context vectors for the L-th layer of the decoder.

The decoder works similarly, albeit with some slight modifications. For the first attention block, the "Masked Multi-Head Attention", the tokens in the output sequence are transformed via a variant of self-attention. Next, these transformed tokens are used as the decoder encodings in the standard variant of attention, as described in section 3.2. Finally, the tokens generated from this procedure are passed to another position-wise MLP. At a given time-step t, the translated word is generated in the same manner, with the output distribution over words parameterized by a softmax function over the logits at the last layer's token.

Positional Encoding Unlike in RNNs, where there is a natural temporal structure, a priori, the transformer has no knowledge of positional information - i.e., the output of the transformer is permutation equivariant with respect to the input. To combat this, prior to the first layer of the transformer, a positional encoding is applied to the input. This positional encoding is a fixed value that is added to each token in the sequence. Specifically, we construct a $(T \times d)$ positional encoding matrix $\mathbf{P}$ whose values are

$$\mathbf{P}_{(\text{pos},2i)} = \sin\left(\frac{\text{pos}}{10000^{2i/d}}\right) \quad \mathbf{P}_{(\text{pos},2i+1)} = \cos\left(\frac{\text{pos}}{10000^{2i/d}}\right)$$

The positional encoding at a given dimension is a periodic function of the timestep t, and the frequency of the function decreases with the dimension d. This positional encoding is typically just added to the input matrix (as they have the same size).

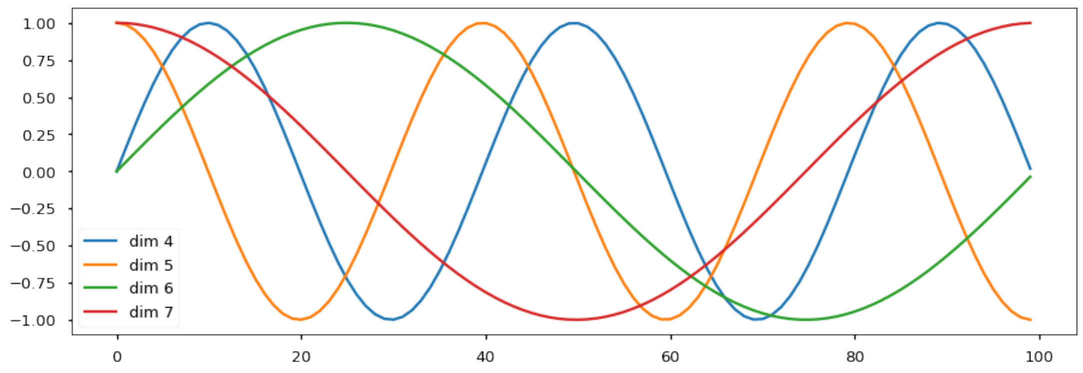


Figure 3: The positional encoding allows each dimension to be associated with a different frequency periodic function that varies with the timestep in the sequence.

Self-Attention In this section, we will describe self-attention, the mechanism that allows transformers to share information across the entire sequence. Self-attention takes in a sequence of input query tokens $\{\mathbf{x}_q\}$, a set of input key tokens $\{\mathbf{x}_k\}$, and a set of input value tokens $\{\mathbf{x}_v\}$. For simplicity, we will describe the case of single-headed self-attention and then describe the case of multi-headed self-attention. Also, for now, you can think of these three being the same sequence - the input tokens $\{\mathbf{x}_t^0\}$, but we will explain later the convenience of having greater generality. Each token $\mathbf{x}$ is linearly projected into a key, query, and value vector:

$$\mathbf{q}_t^0 = \mathbf{W}_Q^0 \mathbf{x}_{q,t}^0 \quad \mathbf{k}_t^0 = \mathbf{W}_K^0 \mathbf{x}_{k,t}^0 \quad \mathbf{v}_t^0 = \mathbf{W}_V^0 \mathbf{x}_{v,t}^0$$

where $\mathbf{W}_Q^0, \mathbf{W}_K^0$, and $\mathbf{W}_V^0$ are learned weights.

In matrix form, this is expressed as

$$\mathbf{Q}^0 = \mathbf{W}_Q^0 \mathbf{X}_Q^{0,\top}, \quad \mathbf{K}^0 = \mathbf{W}_K^0 \mathbf{X}_K^{0,\top}, \quad \mathbf{V}^0 = \mathbf{W}_V^0 \mathbf{X}_V^{0,\top}$$

In each of these matrices, the t-th row is the t-th query, key, or value. The attention weights are then calculated as

$$\boldsymbol{\alpha}^0 = \text{softmax}\left(\frac{\mathbf{Q}^0 \mathbf{K}^{0,\top}}{\sqrt{d}}\right).$$

Prior to the softmax and scaling, the ij-th value of $\mathbf{Q}^0\mathbf{K}^{0,\top}$ is $\text{score}(\mathbf{q}_i^0, \mathbf{k}_j^0)$, where the score is the standard inner project. In other words, this matrix product computes the same attention mechanism, except across all values of the sequence. The values of the matrix are then scaled to have unit variance, and a softmax is applied to ensure that each row of $\boldsymbol{\alpha}^0$ sums to one. Each row $\boldsymbol{\alpha}_t^0$ is analogous to $\boldsymbol{\alpha}^t$ in RNN attention, representing how much each position t attends to every other position in the input sequence.

Finally, we get the values $\mathbf{a}_t^0$ by multiplying $\boldsymbol{\alpha}^0$ by the value matrix $\mathbf{V}^0$, which takes a weighted average of each row of the values, where the weights are the attention weights. Note that now, each token in the source sequence can attend to any other token in $O(1)$ steps, significantly improving the conditioning of the problem.

The reason why we allow the input query, key, and value tokens to be arbitrary (not just the input tokens to a given layer) is because in the decoder's second attention, attention is used in a similar fashion to RNNs, where the tokens in the decoder attend to the hidden tokens of the encoder. In this case, we can achieve the same affect as RNN attention by setting the query to be the decoder tokens and the keys and values to be the encoder tokens.

Encoding and Masked Decoding To encode the source sequence and get the set of attention weights, it suffices to perform the forward passes through the encoder, as described above. However for the decoder, based on our current description, there exists one key flaw. If we were to naively perform a forward pass with self attention, then the decoder could trivially "read off" the correct word in the target sequence. This is because we pass in the whole sentence at once, meaning that if we use self attention, the model at a given word can just put all its weight on the next position, since that is the transformed token for the correct word in the sequence.

To get around this, the authors use masked self attention. Essentially, in the attention weights computed via $\mathbf{Q}^l\mathbf{K}^{l,\top}$, we mask out all positions ij where $j \geq i$, so that the attention can't "look ahead" in the sequence. This also preserves the autoregressive property, where we want each target word to be conditioned only on the words coming strictly before it. There are a number of ways to implement masked self attention, but we recommend directly computing the mask that contains zeros at positions where $j \geq i$ and ones elsewhere. This should look like an upper triangular matrix without the diagonal. To finish the implementation, one would set the attention weights to $-\infty$ where the mask is 0, so that the exponentiation sets the value to 0 in the softmax. This allows us to get the correct log-likelihood scores when performing inference. To generate new sequences, we can simply keep a list of generated tokens, and each time we generate a new token, we pass all previously generated tokens into the network and sample from the last layer.

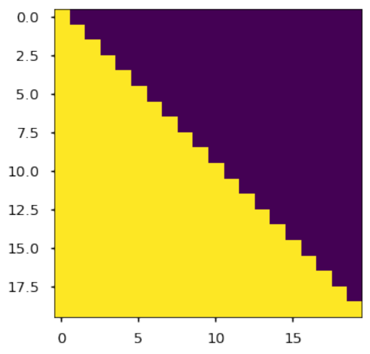


Figure 4: Masked decoding: We mask out all positions from future timesteps.

5.3 Programming Details

Implementation Details Since transformers are significantly more complicated than the models you have been implementing so far, to reduce your load, we highly recommend that you use PyTorch to implement your solution. **To simplify the implementation, we have included the init functions for each class you will implement. Please do not change these, and use the object variables we have defined within them to implement the functions below.** For this assignment, the `torch.nn` library will be most helpful - in particular, the following modules -

- `torch.nn.Linear`
- `torch.nn.Softmax`
- `torch.nn.LayerNorm`
- `torch.nn.Dropout`
- `torch.nn.ModuleList`
- `torch.nn.ReLU`
- `torch.nn.Embedding`
- `torch.nn.CrossEntropyLoss`

are some building blocks we recommend you use to simplify the coding. You are also free to use standard numpy-style torch operations (i.e., `torch.matmul`) to perform matrix operations. **You are not allowed to use PyTorch implementations of transformers, such as the `torch.nn.MultiheadAttention` module. If it is unclear whether the module you are using is allowed or not, please reach out to us beforehand and we can clarify.**

Dependencies In order for your code to run properly on Gradescope, we recommend using PyTorch 2.4 or above and any numpy verioon below 2.0. In addition, our starter code uses Python 3 type hints, so you need to use Python 3.

We recommend using Anaconda to manage dependencies. To create an Anaconda environment named `seq2seq` with these dependencies, run the following:

```
$ conda create -n seq2seq python=3.10.6
$ conda activate seq2seq
$ python -m pip install -r requirements.txt
```

Loading and processing data We have provided you with helper functions in `transformer.py` that will allow you to easily generate training and test sets that can be directly passed into the `train` function in the file. To load the data with our recommended settings, you can use the following commands:

```
# Loads data from Spanish -> English machine translation task
train_sentences, test_sentences, source_vocab, target_vocab = load_data()
# Generates training/test data based on english and spanish vocabulary sizes
# and caps max length of sentence at 12
train_source, train_target = preprocess_data(train_sentences,
    len(source_vocab), len(target_vocab), 12)
test_source, test_target = preprocess_data(test_sentences,
    len(source_vocab), len(target_vocab), 12)
```

5.4 Tasks

WARNING: Running all the experiments can take several hours (more than 8). Please start the assignment early.

1. **(30 points)** Implement a transformer model. This question is autograded.
   (a) **(3 points)** Implement the `PositionalEncodingLayer` class in `transformer.py`.
   (b) **(8 points)** Implement the `SelfAttentionLayer` class in `transformer.py`.
   (c) **(2 points)** Implement the `_lookahead_mask` function in the `Decoder` class in `transformer.py`.
   (d) **(11 points)** Implement the `predict` function in the `Transformer` class in `transformer.py` using beam search.
   (e) **(6 points) 10-617 ONLY:** Implement the `bleu_score` function in `transformer.py`.

   As with previous assignments, you can test your implementation of these functions locally by using the `TestPositionalEncoding`, `TestSelfAttention`, `TestSelfAttentionWithMask`, `TestMask`, `TestBeamSearch`, and `TestBleuScore` functions defined in `tests.py`.

2. **(15 points)** Run the following experiments on your seq2seq model. For all the experiments below, use a hidden dimension of 256, and keep the default settings from the `train` function in the `transformer.py` file. Make sure to save your model, since the second question will require you to generate plots and sentences from each of these models. You can save a PyTorch model as follows:

```
torch.save(model.state_dict(), 'filename.pkl') # Save the model
model.load_state_dict(torch.load('filename.pkl')) # Load a saved model
```

   (a) **(3 points)** Using the code you just implemented, train a transformer with 1 block in the encoder, 1 block in the decoder, and 1 attention head in each block. Plot the training and test loss.

Solution

(b) **(3 points)** Using the code you just implemented, train a transformer with 1 block in the encoder, 1 block in the decoder, and 2 attention heads in each block. Plot the training and test loss.

Solution

(c) **(3 points)** Using the code you just implemented, train a transformer with 2 blocks in the encoder, 1 blocks in the decoder, and 1 attention head in each block. Report the final training and test loss. Plot the training and test loss.

Solution

(d) **(3 points)** Using the code you just implemented, train a transformer with 2 blocks in the encoder, 2 blocks in the decoder, and 2 attention heads in each block. Report the final training and test loss. Plot the training and test loss.

Solution

(e) **(3 points)** Using the code you just implemented, train a transformer with 2 blocks in the encoder, 3 blocks in the decoder, and 4 attention heads in each block. Report the final training and test loss. Plot the training and test loss.

Solution

3. **(4 points)** Using the last model you train, generate translations for any 6 sequences in the test set with a beam size of 3. Consider using the `decode_sentence` helper function provided in the starter code. For each of them, include the source sentence, the target sentence, the predicted sentence, and the average log-likelihood of the predicted sentence.

Solution

4. **(4 points)** Using the last model, visualize the attention matrix generated by each attention head in the second self-attention module of the first decoder block for the first 3 **train** sentences. The forward method of the `Transformer` class returns 2 things: the output sentence and the attention matrix you should visualize (first dimension is number of heads). You can use them in the `visualize_attention` helper function that is provided in the starter code. There should be 12 plots in total. Additionally, discuss in 1-2 sentences what you see.

Solution

5. **(3 points)** We want to see the effect of the beam size in beam search. Using the last model you trained, compute beam search predictions for the first 100 test examples, using beam size 1, 2, ..., 8. For each beam size, compute the average normalized log-likelihood of the predicted sentences. Plot your results. Additionally, discuss your results in 1-2 sentences.

Solution

6. **(3 points) 10-617 ONLY:** For every model you trained, use beam search with beam size 3 to make predictions for every test examples, and compute the average BLEU-1, BLEU-2, BLEU-3, and BLEU-4 scores. Reported your results up to the 4th decimal.

Solution

7. **(1 point) 10-617 ONLY**: Discuss, in 1-2 sentences, a limitation of the BLEU score.

Solution

Collaboration Questions Please answer the following: After you have completed all other components of this assignment, report your answers to the collaboration policy questions detailed in the Academic Integrity Policies found here.

1. Did you receive any help whatsoever from anyone in solving this assignment? Is so, include full details.
2. Did you give any help whatsoever to anyone in solving this assignment? Is so, include full details.
3. Did you find or come across code that implements any part of this assignment ? If so, include full details even if you have not used that portion of the code.

Solution

References

[Cho et al., 2014] Cho, K., van Merriënboer, B., Gülçehre, Ç., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using rnn encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

[Luong et al., 2017] Luong, T., Pham, H., and Manning, C. D. (2017). Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*.

[Neubig, 2017] Neubig, G. (2017). Neural machine translation and sequence-to-sequence models: A tutorial. *arXiv preprint arXiv:1703.01619*.

[Vaswani et al., 2017] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need. *CoRR*, abs/1706.03762.