DS-GA-1011: Natural Language Processing with Representation Learning, Fall 2024

HW2 - Machine Translation

Name
NYU ID

Please write down any collaborators, AI tools (ChatGPT, Copliot, codex, etc.), and external resources you used for this assignment here.
Collaborators: Ziqi Liu, Yuanxin Zhang
AI tools: Chatgpt
Resources:

By turning in this assignment, I agree by the honor code of the College of Arts and Science at New York University and declare that all of this is my own work.

Acknowledgement: Problem 1 was developed by Yilun Kuang. Problem 2 is based off of Annotated Transformers from Sasha Rash and developed by Nitish Joshi.

Before you get started, please read the Submission section thoroughly.

Submission

Submission is done on Gradescope.

Written: You can either directly type your solution in the released `.tex` file, or write your solution using pen or stylus. A `.pdf` file must be submitted.

Programming: Questions marked with "coding" at the start of the question require a coding part. Each question contains details of which functions you need to modify. We have also provided some unit tests for you to test your code. You should submit all `.py` files which you need to modify, along with the generated output files as mentioned in some questions.

Compute Budget: For question 2.3, you should expect the total code execution time to be less than 2 hours on a single NVIDIA Quadro RTX 8000 GPU from NYU Greene HPC. Please plan ahead, as requesting GPU resources on the cluster can take several hours or even longer during peak times.

Due Date: This homework is due on October 9, 2024, at noon 12pm Eastern Time.

1 Recurrent Neural Network

In this problem, you will show the problem of vanishing and exploding gradients for Recurrent Neural Network (RNN) analytically. To show this, we will first expand the gradient of the loss function with respect to the parameters using the chain rule. Then, we will bound the norm of each individual partial derivative

with matrix norm inequalities. The last step will be to collect all of the partial derivative terms and show how repeated multiplication of a single weight matrix can lead to vanishing or exploding gradients.

1.1 RNN Derivatives

Let $S = (s_1, \cdots, s_T)$ be a sequence of input word tokens and T be the sequence length. For a particular token $s_t \in \mathcal{V}$ for $1 \leq t \leq T$, we can obtain its corresponding word embedding $x_t \in \mathbb{R}^d$ by applying equation (1), where $\phi_{\text{one-hot}}$ is the one-hot encoding function and W_e is the word embedding matrix.

The RNN forward pass computes the hidden state $h_t \in \mathbb{R}^{d'}$ using equation (2). Here $W_{\text{hh}} \in \mathbb{R}^{d' \times d'}$ is the recurrent weight matrix, $W_{\text{ih}} \in \mathbb{R}^{d' \times d}$ is the input-to-hidden weight matrix, $b_h \in \mathbb{R}^{d'}$ is the hidden states bias vector, and $\sigma : \mathbb{R}^{d'} \to [-1, 1]^{d'}$ is the tanh activation function. $W_{\text{hh}}, W_{\text{ih}}, b_h$ are shared across sequence index t.

The output of RNN $o_t \in \mathbb{R}^k$ at each sequence index t is given by equation (3), where $W_{h_o} \in \mathbb{R}^{k \times d'}$ is the hidden-to-output weight matrix and $b_o \in \mathbb{R}^k$ is the output bias vector. For an input sequence $S = (s_1, \cdots, s_T)$, we have a corresponding sequence of RNN hidden states $H = (h_1, \cdots, h_T)$ and outputs $O = (o_1, \cdots, o_T)$.

$$x_t = W_e \phi_{\text{one-hot}}(s_t) \tag{1}$$

$$h_t = \sigma(W_{\text{hh}} h_{t-1} + W_{\text{ih}} x_t + b_h) \tag{2}$$

$$o_t = W_{h_o} h_t + b_o \tag{3}$$

Let's now use this RNN model for classification. In particular, we consider the last output o_T to be the logits (scores for each class), which we then convert to the class probability vector $p_T \in [0, 1]^k$ by $p_T = g(W_{h_o} h_T + b_o)$ where $g(\cdot)$ is the softmax function and $\|p_T\|_1 = 1$.

1. (1 point, written) Write down the per-example cross-entropy loss $\ell(y, p_T)$ for the classification task. Here $y \in \{0, 1\}^k$ is a one-hot vector of the label and p_T is the class probability vector where $p_T[i] = p(y[i] = 1 \mid S)$ for $i = 1, \ldots, k$. ($[i]$ denotes the i-th entry of the corresponding vector.)

$$\ell(y, p_T) = -\sum_{i=1}^{k} y[i] \log(p_T[i])$$

2. To perform backpropagation, we need to compute the derivative of the loss with respect to each parameter. Without loss of generality, let's consider the derivative with respect to a single parameter $w = W_{\text{hh}}[i, j]$ where $[i, j]$ denotes the (i, j)-th entry of the matrix. By chain rule, we have

$$\frac{\partial \ell}{\partial w} = \frac{\partial \ell}{\partial o_t} \frac{\partial o_t}{\partial h_t} \frac{\partial h_t}{\partial w} \tag{4}$$

Note that the first two derivatives in the 4 are easy to compute, so let's focus on the last term $\frac{\partial h_t}{\partial w}$. During the lecture, we have shown that

$$\frac{\partial h_t}{\partial w} = \sum_{i=1}^{t} \frac{\partial h_t}{\partial h_i} \frac{\partial h_i^+}{\partial w} \tag{5}$$

Here $\frac{\partial h_i^+}{\partial w}$ denotes the "immediate" gradient where h_{i-1} is taken as a constant.

(a) (1 point, written) Give an expression for $\frac{\partial h_i^+}{\partial w}$.

$h_t = \sigma(W_{hh} h_{t-1} + W_{ih} x_t + b_h)$

$w = W_{hh}[i, j]$

$\frac{\partial h_i^+}{\partial w} = \frac{\partial}{\partial w} \sigma(W_{hh} h_{i-1}) = \sigma'(W_{hh} h_{i-1} + W_{ih} x_i + b_h) \cdot h_{i-1}[j]$

$= (1 - h^2) \cdot h_{i-1}[j]$

(b) (2 points, written) Expand the gradient vector $\frac{\partial h_t}{\partial h_i}$ using the chain rule as a product of partial derivatives of one hidden state with respect to the previous hidden state. You do not need to explicitly do differentiations beyond that.

$$\frac{\partial h_t}{\partial h_i} = \frac{\partial h_t}{\partial h_{t-1}} \cdot \frac{\partial h_{t-1}}{\partial h_{t-2}} \cdots \frac{\partial h_{i+1}}{\partial h_i}$$

$$= \prod_{k=i+1}^{t} \frac{\partial h}{\partial h_{k-1}}$$

3. (2 points, written) Now let's further expand one of the partial derivatives from the previous question. Write down the Jacobian matrix $\frac{\partial h_{i+1}}{\partial h_i}$ by rules of differentiations. You can directly use σ' as the derivative of the activateion function in the expression.

$$\frac{\partial h_{i+1}}{\partial h_i} = \frac{\partial \sigma(W_{hh}h_i + W_{ih}x_{i+1} + b_h)}{\partial h_i}$$

$$= \sigma'(W_{hh}h_i + W_{ih}x_{i+1} + b_h)\ \frac{\partial}{\partial h_i}(W_{hh}h_i + W_{ih}x_{i+1} + b_h)$$

$$= \sigma'(W_{hh}h_i + W_{ih}x_{i+1} + b_h)\ W_{hh}^T$$

$$= \text{diag}(\sigma'(W_{hh}h_i + W_{ih}x_{i+1} + b_h))\ W_{hh}^T$$

1.2 Bounding Gradient Norm

To determine if the gradient will vanish or explode, we need a notion of magnitude. For the Jacobian matrix, we can use the induced matrix norm (or operator norm). For this question, we use the spectral norm $\|A\|_2 = \sqrt{\lambda_{\max}(A^\top A)} = s_{\max}(A)$ for a matrix $A \in \mathbb{R}^{m \times n}$. Here $\lambda_{\max}(A^\top A)$ denotes the maximum eigenvalue of the matrix $A^\top A$ and $s_{\max}(A)$ denotes the maximum singular value of the matrix A. You can learn more about matrix norms at this Wikipedia entry.
Now, to determine if the gradient $\frac{\partial \ell}{\partial w}$ will vanish or explode, we can focus on $\|\frac{\partial h_t}{\partial h_i}\|$. Note that if $\|\frac{\partial h_t}{\partial h_i}\|$ vanishes or explodes, $\|\frac{\partial \ell}{\partial w}\|$ also vanishes or explodes based on (4) and (5).

1. (2 points, written) Given the mathematical form of the Jacobian matrix $\frac{\partial h_{i+1}}{\partial h_i}$ we derived earlier, we can now bound the norm of the Jacobian with the following matrix norm inequality

$$\|AB\|_2 \leq \|A\|_2 \cdot \|B\|_2 \tag{6}$$

for matrices A, B with matched shapes. Write down a bound for $\left\|\frac{\partial h_i}{\partial h_{i-1}}\right\|_2$.

$$\left\|\frac{\partial h_i}{\partial h_{i-1}}\right\|_2 = \left\|\operatorname{diag}\left(\sigma'\left(W_{hh}h_{i-1} + W_{ih}x_i + b_h\right)\right) \cdot W_{hh}^\top\right\|_2$$

$$\leq \left\|\operatorname{diag}\left(\sigma'\left(W_{hh}h_{i-1} + W_{ih}x_i + b_h\right)\right)\right\| \cdot \left\|W_{hh}^\top\right\|_2$$

Since $\sigma : \mathbb{R}^{d'} \to [-1, 1]^{d'}$ is the tanh activation function

$\sigma'(\cdot) = 1 - \tanh^2(\cdot) \leq 1$

So $\left\|\operatorname{diag}\left(\sigma'\left(W_{hh}h_{i-1} + W_{ih}x_i + b_h\right)\right)\right\| \leq 1$

$\left\|W_{hh}^\top\right\|_2 = s_{\max}\left(W_{hh}^\top\right)$

Thus $\left\|\frac{\partial h_i}{\partial h_{i-1}}\right\|_2 \leq s_{\max}\left(W_{hh}^\top\right)$

2. (4 points, written) Now we have all the pieces we need. Derive a bound on the gradient norm $\|\frac{\partial h_t}{\partial h_i}\|$. Explain how the magnitude of the maximum singular value of W_{hh} can lead to either vanishing or exploding gradient problems. [**HINT:** You can use the fact that for the tanh activation function $\sigma(\cdot)$, the derivative $\sigma'(\cdot)$ is always less than or equal to 1.]

$$\left\| \frac{\partial h_i}{\partial h_{i-1}} \right\|_2 \leq \| W_{hh}^T \|_2 = s_{max}(W_{hh}^T)$$

$$\left\| \frac{\partial h_t}{\partial h_i} \right\|_2 \leq \prod_{i=1}^{t-1} \left\| \frac{\partial h_i}{\partial h_{i-1}} \right\|_2 = (s_{max}(W_{hh}^T))^{t-i}$$

If $s_{max}(W_{hh}) < 1$, then $\left\| \frac{\partial h_t}{\partial h_i} \right\|_2$ approaches zero as $t-i$ grow large.

For large time steps back, upper bound of the gradient vanishes.

If $s_{max}(W_{hh}) > 1$, then $\left\| \frac{\partial h_t}{\partial h_i} \right\|_2$ grows exponentially in further time step,

i.e. exploding gradient, resulting in unstable training.

If $s_{max}(W_{hh}) = 1$ ideally, the gradient neither vanishes nor explodes.

3. (1 point, written) Propose one way to get around the vanishing and exploding gradient problem.

I suggest using LSTM (Long Short-Term Memory) instead of RNN to deal with the vanishing and exploding gradient problem.

In LSTM, a Forget Gate is introduced to control part of information from previous time steps to be discussed. An Input Gate to decide part of new information needs to be added. An Output Gate determines the next hidden state and is passed to the next LSTM layer.

This should be able to solve vanishing and exploding gradient problem because we can selectively forget information from previous time steps.

2 Machine Translation

The goal of this homework is to build a machine translation system using sequence-to-sequence transformer models https://arxiv.org/abs/1706.03762. More specifically, you will build a system which translates German to English using the Multi30k dataset (https://arxiv.org/abs/1605.00459) You are provided with a code skeleton, which clearly marks out where you need to fill in code for each sub-question.

First go through the file `README.md` to set up the environment required for the class.

2.1 Attention

Transformers use scaled dot-product attention — given a set of queries Q (each of dimension d_k), a set of keys K (also each dimension d_k), and a set of values V (each of dimension d_v), the output is a weighted sum of the values. More specifically,

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V \tag{7}$$

Note that each of Q, K, V is a matrix of vectors packed together.

1. (2 points, written) The above function is called 'scaled' attention due to the scaling factor $\frac{1}{\sqrt{d_k}}$. The original transformers paper mentions that this is needed because dot products between keys and queries get large with larger d_k.

   For a query q and key k both of dimension d_k and each component being an independent random variable with mean 0 and variance 1, compute the mean and variance (with steps) of the dot product $q.k$ to demonstrate the point.

Mean of $q \cdot k$ $\quad q \cdot k = \sum_{i=1}^{d_k} q_i k_i \quad E[q_i k_i] = E[q_i]\,E[k_i] = 0 \cdot 0 = 0$

$E[q \cdot k] = E[\sum_{i=1}^{d_k} q_i k_i] = \sum_{i=1}^{d_k} E[q_i k_i] = 0$

Variance of $q \cdot k$. $Var[q \cdot k] = E[(q \cdot k)^2] - (E[q \cdot k])^2$

$E[(q \cdot k)^2] = E[(\sum_{i=1}^{d_k} q_i k_i)^2] = E[\sum_{i=1}^{d_k} (q_i k_i)^2 + \sum_{i \neq j} q_i k_i q_j k_j]$

$E[q_i k_i q_j k_j] = 0$ for all $i \neq j$.

$\sum_{i=1}^{d_k} E[(q_i k_i)^2] = \sum_{i=1}^{d_k} E[q_i^2]\,E[k_i^2] = \sum_{i=1}^{d_k} 1 \cdot 1 = d_k$

$Var[q \cdot k] = E[(q \cdot k)^2] - (E[q \cdot k])^2 = d_k - 0 = d_k$

So variance increases with d_k, softmax output will be closed to 0 or 1, and our attention model will focus its "attention" on one word, causing ineffectness. So we need to scale our function to avoid increase in output with respect to increase in d_k.

2. (2 points, coding) Implement the above scaled dot-product attention in the `attention()` function present in `layers.py`. You can test the implementation after the next part.

See coding

3. (2 point, coding) In this part, you will modify the `attention()` function by making use of the parameters `mask` and `dropout` which were input to the function. The `mask` indicates positions where the attention values should be zero (e.g. when we have padded a sentence of length 5 to length 10, we do not want to attend to the extra tokens). `dropout` should be applied to the attention weights for regularization.

   To test the implementation against some unit tests, run `python3 test.py --attention.`

   See coding

4. (3 points, coding) Instead of a single attention function, transformers use multi-headed attention function. For original keys, queries and values (each of dimension say d_{model}), we use h different projection matrices to obtain queries, keys and values of dimensions d_k, d_k and d_v respectively. Implement the function `MultiHeadedAttention()` in `layers.py`. To test the implementation against some unit tests, run `python3 test.py --multiheaded_attention`.

See coding

2.2 Positional Encoding

Since the underlying blocks in a transformer (namely attention and feed forward layers) do not encode any information about the order of the input tokens, transformers use 'positional encodings' which are added to the input embeddings. If d_{model} is the dimension of the input embeddings, pos is the position, and i is the dimension, then the encoding is defined as:

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{model}}) \tag{8}$$

$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{model}}) \tag{9}$$

1. (2 points, written) Since the objective of the positional encoding is to add information about the position, can we simply use $PE_{pos} = sin(pos)$ as the positional encoding for pos position? Why or why not?

   No. This will make encoding for every dimension exactly the same, and lose the ability to capture information from different dimensions. Using the combination of cosine and sine functions introduces different frequencies of periods and helps to better represent more complex positional information.

2. (2 points, coding) Implement the above positional encoding in the function `PositionalEncoding()` in the file `utils.py`. To test the implementation against some unit tests, run `python3 test.py --positional_encoding`.

See coding

`

2.3 Training

1. (2 points, written) The above questions should complete the missing parts in the training code and we can now train a machine translation system!

   Use the command `python3 main.py` to train your model. For the purpose of this homework, you are not required to tune any hyperparameters. You should submit the generated `out_greedy.txt` file containing outputs. You must obtain a BLEU score of atleast 35 for full points (By default we are using BLEU-4 for this and all subsequent questions).

   See coding

2.4 Decoding & Evaluation

In the previous question, the code uses the default `greedy_decode()` to decode the output. In practice, people use algorithms such as beam search decoding, which have been shown to give better quality outputs. (Note: In the following questions, use a model trained with the default i.e. given hyperparameters)

1. (2 points, written) In the file `utils.py` you will notice a function `subsequent_mask()`. What does that function do and why is it required in our model?

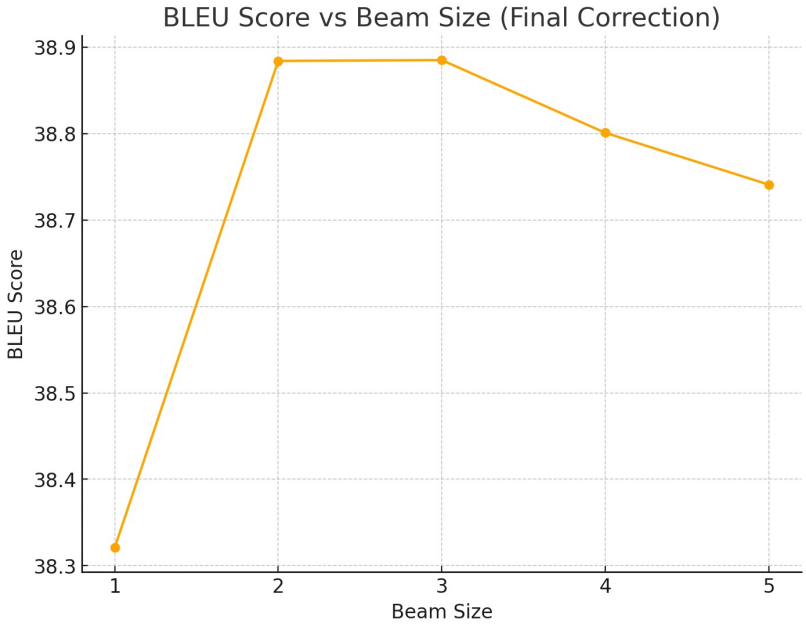
This function generates a mask that blocks the model from future steps, ensuring it to use only current and previous time steps. It is required because Transformer decoder should generate words only based on previously generated words. If it can access future words, it is kind of cheating and would lead to unrealistic inference. The function ensures the model to be autoregressive.

2. (5 points, coding) Implement the `beam_search()` function in `utils.py`. We have provided the main skeleton for this function and you are only required to fill in the important parts (more details in the code). You can run the code using the arguments `--beam_search` and `--beam_size`. You should submit the generated file `out_beam.txt` when `beam_size = 2`.

   To test the implementation against some unit tests, run `python3 test.py --beam_search`.

See coding

3. (3 points, written) For the model trained in question 1.3, plot the BLEU score as a function of beam size. You should plot the output from beam size 1 to 5. Is the trend as expected? Explain your answer.



BLEU score is pretty low with beam_size=1. When increase beam_size to 2, it increases to almost 38.9 and it about the same score when beam_size=3. It decreases after we gradually increase beam_size to 4 and 5.

This can be expected beam_size is a matter of accuracy and range of exploration. BLEU measures how similar our result is compared to truth based on n-gram algorithms. A larger beam size allows the model to try out more candidate words for translation, improving the likelihood for selecting a better translation. When it reaches a balance between accuracy and exploration (beam_size={2,3}), BLEU reaches its peak. When beam_size is too large ({4,5} in our case), beam search degeneration appears, which is likely because the model choose between to big a range, and tend to choose fluent flow over accurate translation. A larger beam_size means the model needs to consider more possible translation which would also include more noise.

4. (2 points, written) You might notice that some of the sentences contain the '⟨unk⟩' token which denotes a word not in the vocabulary. For systems such as Google Translate, you might not want such tokens in the outputs seen by the user. Describe a potential way to avoid (or reduce) the occurrence of these tokens in the output.

1. One reason for '<unk>' is that a word is not in the vocabulary. So we can simply increase the size of of vocabulary to train your model on.
2. Subword tokenization. If we break down unknown words into known words, they are more likely to be in the vocabulary.
3. Copy mechanism. Some rare and unknown words are names, nouns, and domain-specific terms. For these words we can directly copy them into translation destination.

5. (2 points, written) In this homework, you have been using BLEU score as your evaluation metric. Consider an example where the reference translation is "I just went to the mall to buy a table.", and consider two possible model generations: "I just went to the mall to buy a knife." and "I just went to the mall to buy a desk.". Which one will BLEU score higher? Suggest a potential fix if it does not score the intended one higher.

BLEU only sees the "knife" and "desk" are both different than table, but it cannot recognize desk is a closer word to table than knife. The two generations should have similar BLEU score.

A potential fix is to use embedding-based metrics. So "desk" could have a higher score because it have more similar neighbors and embedding space with "table".

Or can incorporate synonym dictionaries. In this way semantically related words like "desk" and "table" can be treated as matches.