Exam – CS 685

due Thursday Nov 11 at 9:30am EST on Gradescope

- This exam should take a couple hours to complete. However, we've given you 48 hours, so you can complete and submit it anytime during that window. Submission to Gradescope should be in PDF format.
- Feel free to use any resources (e.g., lecture slides, videos, conference papers, random websites) to help you answer the problems.
- We strongly prefer that you complete this exam using LaTeX (you can just copy this Overleaf project). If you'd rather type your answers using some different software, however, please feel free to do so. If you want to handwrite your answers and upload a scan of your exam, that's also fine. Just make sure we can easily read everything you wrote.
- Do **NOT** collaborate with other students on this exam. Do **NOT** even discuss exam problems with other students. Each answer should be your own work. We will check a random subset of answers for potential plagiarism using automatic NLP-based tools. Violations will be referred to the UMass Academic Honesty Board.
- If you have clarification questions for the instructors, please use **private Piazza posts** to ask them. We will delete any public posts about the exam. If the entire class would benefit from the answer to a privately-posted question (e.g., fixing a typo or unclear wording in a problem), the instructors will add them to a public Piazza post about exam-related issues.
- We will try to regularly reply to Piazza questions during the day (i.e., 9am-5pm EST Tuesday and Wednesday). If you post a question on Thursday at 3:30AM, do not expect to hear back from us :)

section	points
1. Prompt-based learning	/ 25
2. Long context language models	/ 30
3. Tokenization in language models	/ 20
4. Machine translation	/ 25
Total	/ 100

1 Prompt-based learning (25 pts)

Gilbert works for Zeta, a company that produces fashionable VR goggles. His boss tasks him with figuring out how much praise Zeta's latest luxury goggles are getting on the Internet. Gilbert collects a small dataset of 300 Tweets that contain mentions of these goggles. He tags each Tweet with whether it contains praise or not (i.e., a binary label). Since his dataset is so small, he decides to leverage one of Zeta's enormous pretrained left-to-right language models to solve his classification task via prompt-based learning.

Question 1.1. *[10 points]* Gilbert decides to start out with discrete prompts. First, he manually writes some prompts himself (e.g., *Does the following Tweet praise Zeta's amazing goggles?*), and then he applies a paraphrase generation model to obtain even more prompts. However, he eventually discovers that none of his prompts is able to match the validation performance of a continuous prompt (a sequence of 100 learned embeddings) obtained via prompt tuning.[1] If Gilbert enumerated every possible k-token-long discrete prompt where $1 < k < 100$, would he be guaranteed to find one that matches the validation performance of the learned continuous prompt? Why or why not?

Answer:

Even if Gilbert enumerates every possible k-token-long discrete prompt, there would be no guarantee that he would find a combination that matches the validation performance of the learned continuous prompt. Discrete prompt adaptation is limited in that it only checks parameters from a limited number of possible options, whereas continuous learned embeddings are able to learn over a continuous distribution of parameters and are more likely to find the ideal parameter values. This is analogous to grid search vs random search in hyperparameter optimization - random search returns more accurate hyperparameter values as "randomly chosen trials are more efficient for hyper-parameter optimization than trials on a grid" (Bergstra and Bengio, 2012).

Manually designed discrete prompts in general do not have best performance, as prompt engineering is a non-trivial task (small perturbations in prompt can significantly diminish the model's performance), and creating a perfect prompt requires understanding of the LM's inner workings, domain knowledge and repeated trial and error (Tianyu Gao, *The Gradient*, 2021). Learned continuous prompts are automated and generally more reliable and accurate.

[1] While the method in the prompt tuning paper (which we discussed in class) is implemented within the T5 encoder/decoder model, it can also be trivially applied to a decoder-only model like the one Gilbert is using.

Question 1.2. *[10 points]* Gilbert determines that enumerating every possible discrete prompt is computationally infeasible. He decides to instead develop a method that automatically maps discrete prompts to continuous prompts. To facilitate this process, he collects a large dataset of *(discrete prompt, learned continuous prompt)* pairs across a variety of tasks, where each continuous prompt is a sequence of 100 embeddings. Concretely, assume each discrete prompt contains tokens $d_1, d_2, \ldots, d_m$ where m varies across the prompts, while each continuous prompt contains tokens $c_1, c_2, \ldots, c_{100}$. Fully detail a model architecture, objective function, and training process that Gilbert can use to solve this problem, and make sure to also specify how he can use this model to produce new continuous prompts from discrete prompts at test-time.

Answer:

Given that we have a large dataset of (*discrete prompt*, *learned continuous prompt*) pairs, we can build an encoder model with a feed forward layer on top to generate an embedded representation from the input discrete prompt.

Let us represent discrete prompt D and ground-truth learned continuous prompt embedding C as

$$D = (d_1, d_2, ..., d_m)$$
$$C = (c_1, c_2, ..., c_{100})$$

where

$$D \in \mathbb{R}^{m \times e}$$
$$C \in \mathbb{R}^{100 \times e}$$

The encoder model will generate an embedding sequence, which can then be brought to our required 100 length sequence C'. We will train the process based on similarity to the ground-truth learned continuous prompt embedding, and hence will use cosine distance as the objective function. We will backprop through the network during train time to minimise the cosine distance (or increase similarity) between the generated embedding and our ground-truth learned embedding.

During test time, Gilbert will only need to feed the discrete prompt as input to the model. The output embedding will be the continuous prompt.

Question 1.3. *[5 points]* Assume Gilbert successfully trains a model for the task in the previous subproblem and that for any held-out task, the generated continuous prompts exhibit no downstream performance degradation compared to those obtained via prompt tuning. What are two advantages of Gilbert's approach over prompt tuning?

Answer:

The two advantages of Gilbert's approach over prompt tuning are -

1. *Interpretability*: As the learned continuous prompts are associated with a discrete prompt, we can understand what the continuous prompt represents in plain text.
2. *Efficiency*: Gilbert's model would be more efficient than prompt tuning as it is a normal encoder based model. Optimizing Gilbert's model is equivalent to optimizing simple neural net.

2 Long context language models (30 pts)

Julia comes up with a clever approach to pack more context into Transformer language models without overly increasing the sequence length. Given an input sequence with tokens $x_1, x_2, \ldots, x_n$, she decides that instead of directly feeding the corresponding token embeddings $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n$ into the first Transformer block of her language model, she will perform a segment-level aggregation step to reduce the sequence length. Specifically, she decides to encode non-overlapping contiguous segments of 5 tokens (e.g., $s_1 = x_1x_2x_3x_4x_5$, $s_2 = x_6x_7x_8x_9x_{10}$, and so on) by individually feeding each segment s_i into a pretrained BERT model and extracting the final-layer [CLS] representation $\boldsymbol{s}_i$.

Question 2.1. *[10 points]* In her first attempt to build this model, she simply passes the resulting sequence of segment representations $\boldsymbol{s}_1, \boldsymbol{s}_2, \ldots, \boldsymbol{s}_{\lceil n/5 \rceil}$ through a standard 16-layer Transformer. She intends to train her model for next word prediction and compute the cross entropy loss across all timesteps in parallel, just like in a normal token-level Transformer. However, as she sets about implementing the loss function, she discovers a fundamental flaw in her design. What's wrong with her setup?

Answer:
Some fundamental flaws in Julia's design are -

1. The vocab used for next word is not the same as the vocab used for input sequence. This would lead to complications while calculating loss.

2. We would only have previous n segments, hence we would not know the prefix if we want to predict a word which is at 5n+i position ($i \leq 4$).

Question 2.2. *[10 points]* How would you modify Julia's approach to properly train a language model for next word prediction using her segment-level input representation idea? Do not add any new components to the model, and assume computational efficiency is not a concern.

Answer:

One method to fix Julia's approach is to replace fixed window conversion into contiguous segments to a sliding window conversion of 5 tokens into a segment. For edge tokens we can add padding in order to ensure 5 tokens in the segment representation. Hence when we are predicting next word we will always have complete prefix context available to accurately predict the next word.

Question 2.3. *[10 points]* After successfully implementing her model, Julia decides to experiment with a different task: given a sequence of segment representations $\boldsymbol{s}_1, \boldsymbol{s}_2, \ldots, \boldsymbol{s}_i$, retrieve the next *segment* s_{i+1} from the set of all possible five-word segments in the training set. She wants to formulate this as a retrieval problem, inspired by recent work in retrieval such as REALM and DPR. Describe how she could set up a new model and training objective for this task using both words and math. Feel free to introduce additional notation if you need to!

Answer:

We are given a sequence of segment representations $\boldsymbol{s}_1, \boldsymbol{s}_2, \ldots, \boldsymbol{s}_i$ where $\boldsymbol{s}_i$ is a BERT based segment representation of 5 contiguous tokens. We have to find the next segment s_{i+1} from set of all possible five-word segments in the training set.

For our proposed model, we will first assume that we have a knowledge corpus Z pregenerated from training set, consisting of all possible five-word segments S in training data passed through BERT.

$$Z = \{S_1, S_2, \ldots, S_{\lceil n/5 \rceil}\}$$

The model, based on REALM's architecture, will have following two components -

1. *Knowledge Retriever*
   - Knowledge retriever module will be responsible to get the most relevant segments from the knowledge corpus.
   - First we will compose our input sequence of segments into a single representation using a transformer model.

     $$x = \text{Transformer}(\boldsymbol{s}_1, \boldsymbol{s}_2, \ldots, \boldsymbol{s}_i)$$

   - Retriever can be defined as a dense inner product model -

     $$p(S|x) = \frac{exp\ f(x, S)}{\sum_{S'} exp\ f(x, S')}$$

     $$f(x, S) = Embed_{input}(x)^\top Embed_{doc}(S)$$

     Where Embed function is nothing but BERT.

2. *Knowledge-Augmented Encoder*
   - This encoder will be a distinct transformer
   - With input x and retrieved segment S this encoder will define probability -

     $$p(s_{i+1}|S, x) = \sum_{S_k} exp\ f(x, S')$$

Our training objective for this model will be based on next word prediction. We can optimize the model following basic SGD.

3 Tokenization in language models (20 pts)

Terry is growing tired of the proliferation of NLP papers about different types of tokenization. He starts wondering whether it is possible to create a single language model that can encode inputs using multiple tokenizations. In particular, he wants his model to be able to represent text as sequences of bytes, characters, subwords, and words.

Terry decides to experiment on the WikiText-103 language modeling benchmark dataset. Given an input sequence from the training data, he first chooses a random segmentation of the sequence that could mix multiple tokenization schemes. Below are two examples of different tokenizations (tokens are shown separated by commas) for the sequence *the evil warlock* that could arise from this process:

- word-level: *the*, , *evil*, , *warlock*[2]
- mixture of tokenizations: *t*, *h*, *e*, , *evil*, , *war*, *##lock*[3]

Then, he feeds these segmented training sequences to a language model whose vocabulary (both token embeddings and softmax layer) includes every byte, character, subword, and word that occurs in WikiText-103's training set. The LM is trained using the standard next-token-prediction objective and its parameters are updated through backpropagation.

Question 3.1. *[20 points]* After successfully training his model (let's call it LM-A), he wants to compare it to a baseline language model (LM-B) trained on WikiText-103 using word-level tokenization. He decides to compute the log likelihood of the sentence *Language models are stupid* according to each model. Describe how he would do this for LM-A and LM-B using words and/or math, and assume that there are no out-of-vocabulary issues associated with this sentence.

Answer:
The likelihood of a sentence for a given language model is defined by -

$$\mathcal{L}(w_1 w_2 \dots w_n) = \prod_{i=1}^{n} p(w_i | w_1, w_2, \dots, w_{i-1})$$

where $w_1 w_2 \dots w_n$ is the input sentence, w_i is the final word/token and $w_1, w_2, \dots, w_{i-1}$ is the prefix.

Case I: LM-A
In this case, the tokenization can be a mixture of word, sub-word, character or byte level.

[2]For simplicity, we'll always treat spaces between words as individual tokens regardless of choice of segmentation.
[3]The ## in front of *lock* indicates that it is a subword.

Hence the likelihood of the sequence can be equivalent to the likelihood of sequence taking either one of the tokenzation methods (we cannot predict which tokenization method is being used). Hence the likelihood is given by sum of likelihoods considering each method -

$$log(\mathcal{L}(w_1 w_2 \dots w_n)) = log(\sum_{j=1}^{j=4} \prod_{i=1}^{n} p(w_{j_i} | w_{j_1}, w_{j_2}, \dots, w_{j_{i-1}}))$$

Case II: LM-B
For the baseline language model, the normal word level tokenization is followed. Hence the log likelihood for this model would be simply the log over the likelihood of the sequence -

$$log(\mathcal{L}(w_1 w_2 \dots w_n)) = log(\prod_{i=1}^{n} p(w_i | w_1, w_2, \dots, w_{i-1}))$$

$$\implies log(\mathcal{L}(w_1 w_2 \dots w_n)) = \sum_{i=1}^{n} log(p(w_i | w_1, w_2, \dots, w_{i-1}))$$

where $w_1 w_2 \dots w_n$ is the input sentence, w_i is the i^{th} token and $w_1, w_2, \dots, w_{i-1}$ are the word level token prefix.

4 Machine translation (25 pts)

Your friend Sophia recommends you a novel written in Korean, claiming it's the best novel ever written. Unfortunately, you don't speak or read Korean, and you don't have money to hire a professional Korean-to-English translator. Thus, you purchase an ebook of the novel and shamefully resort to machine translation, copying chunks of the text and pasting it into Google Translate. However, you quickly discover that you cannot follow the story at all due to many jarring translation errors.

Since you trust Sophia's book recommendations, you decide not to give up so easily. You wonder if it's possible to build a model to perform automatic *post-editing* on top of Google Translate's output. In particular, given a Korean sentence and an English translation of this sentence produced by Google Translate, your model should produce a corrected English translation. To enable this, you collect a large parallel dataset of classic Korean novels paired with human-written English translations.

Question 4.1. *[15 points]* Describe an approach for this post-editing task that leverages the ByT5 model we discussed in class. Make sure to specify the format of the inputs and outputs to this model, as well as any preprocessing you would need to do on your dataset as well as the output of Google Translate. Feel free to introduce any notation (or figures) to make your answer more clear.

Answer:

A neural language model approach to solving Automatic Post-Editing (APE) task can be followed, using the ByT5 approach for tokenization. Our training input will be of the form of (*source*, *MT output*, *target*) triplets described as -

- Source (*src*): Input Korean text
- MT Output (*mt*): Output from Google Translate when fed with the Korean source input text.
- Target (*tgt*): During training, this will be the human generated translation of corresponding Korean text.

Test time input will be of the form of (*source*, *MT output*) pairs while expected output generated will be the post-edited *target*.

As ByT5 is pretty robust and does not get easily affected by errors in the input text, not much preprocessing is required. The basic preprocessing would be conversion of text to UTF-8 encoded bytes. Masking is done on the input bytes, in the form of 20 byte spans. Finally, we drop any illegal bytes in the model's output.

Existing models for APE use joint encoders to model the input (*src*, *mt*) (Lee, Shen et al). This is primarily due to the fact that *src* and *mt* are in different languages and may follow different tokenization methods. However as the ByT5 model works on byte level tokens, it is language independent and both *src* and *mt* can be passed to a single encoder post tokenization. I'd suggest to input the *src* and *mt* to the encoder with a special token [*SEP*] to separate them. Following ByT5's "span corruption" pre-training objective, we would be able to learn a joint representation embedding of both *src* and *mt* which can then be fed to the decoder for the decoding step.

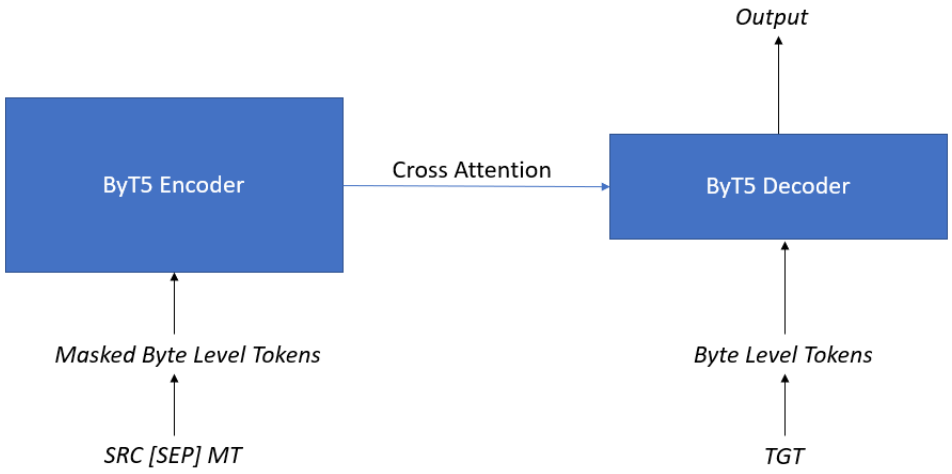

Figure 1: Proposed ByT5 Model for APE Task

Question 4.2. *[10 points]* Describe two ways that you could evaluate whether your post-editing model is producing superior translations than Google Translate.

Answer:
The two ways according to which the post-editing model can be evaluated against the Google Translation are -

1. *Human Evaluation*:
   Human evaluators can be used to compare the post-edited model against the Google Translation output. Evaluators can be instructed to compare the two based on metrics such as fluency, clarity, adequacy etc. either using a scale (say 1-5) or a simple binary choice (”which one is better?”).

2. *BLEU Score*:
   Assuming we have access to a reference translation, we can use BLEU metric to see which translation scores better on average over all translated sentences - the Google Translate output or our post-edited output. BLEU measures the n-gram overlap between a machine translation output and reference translation using precision measure. In our case, the output with more overlap with the reference translation will have the better BLEU score.