Assignment 2

COMP3361: Natural Language Processing - University of Hong Kong

Spring 2024

Goals The primary goal of this assignment is to give you hands-on experience implementing a Transformer language model. Understanding how these neural models work and building one from scratch will help you understand not just language modeling, but also systems for many other applications. Furthermore, you will also use open-source large language models to try cutting-edge tasks such as code generation and math reasoning, thereby gaining a better understanding of prompt technique.

Dataset and Code

You will find code and data here.

Part 1 & 2

Please use up-to-date versions of Python and PyTorch for this assignment. **See our released tutorial of PyTorch (code and video).**

Data The dataset for this paper is the `text8`[1] collection. This is a dataset taken from the first 100M characters of Wikipedia. Only 27 character types are present (lowercase characters and spaces); special characters are replaced by a single space and numbers are spelled out as individual digits (*20* becomes *two zero*). We will be splitting these into sequences of length 20 for Part 1.

Framework code The framework code you are given consists of several files. We will describe these in the following sections. `utils.py` implements an `Indexer` class, which can be used to maintain a bijective mapping between indices and features (strings). `letter_counting.py` contains the driver for Part 1, which imports `transformer.py`. `lm.py` contains the driver for Part 2 and imports `transformer_lm.py`.

Part 3

Framework code Because Part 3 requires GPU, we have released the code in iPython Notebook format to support Google Colab. Please check A2p3.ipynb on GitHub or Google Colab.

1 Building a "Transformer" Encoder (35%)

In this first part, you will implement a simplified Transformer (missing components like layer normalization and multi-head attention) from scratch for a simple task. **Given a string of characters, your task is to predict, for each position in the string, how many times the character at that position occurred before, maxing out at 2.** This is a 3-class classification task (with labels 0, 1, or > 2 which we'll just denote as 2). This task is easy with a rule-based system, but it is not so easy for a model to learn. However, Transformers are ideally set up to be able to "look back" with self-attention to count occurrences in the context. Below is an example string (which ends in a trailing space) and its corresponding labels:

[1] Original site: `http://mattmahoney.net/dc`

```
i like movies a lot
00010010002102021102
```

We also present a modified version of this task that counts both occurrences of letters before *and after* in the sequence:

```
i like movies a lot
22120120102102021102
```

Note that every letter of the same type always receives the same label no matter where it is in the sentence in this version. Adding the `-task BEFOREAFTER` flag will run this second version; default is the first version.

`lettercounting-train.txt` and `lettercounting-dev.txt` both contain character strings of length 20. **You can assume that your model will always see 20 characters as input.**

Getting started Run:

```
python letter_counting.py --task BEFOREAFTER
```

This loads the data for this part, but will fail out because the Transformer hasn't been implemented yet. (We didn't bother to include a rule-based implementation because it will always just get 100%.)

1.1 Q0 (not graded)

Implement Transformer and TransformerLayer for the BEFOREAFTER version of the task. You should identify the number of other letters of the same type in the sequence. This will require implementing both Transformer and TransformerLayer, as well as training in `train_classifier`.

Your Part 1 solutions **should not** use `nn.TransformerEncoder`, `nn.TransformerDecoder`, or any other off-the-shelf self-attention layers. You should only use Linear, softmax, and standard nonlinearities to implement Transformers from scratch.

TransformerLayer This layer should follow the format discussed in class: (1) self-attention (single-headed is fine; you can use either backward-only or bidirectional attention); (2) residual connection; (3) Linear layer, nonlinearity, and Linear layer; (4) final residual connection. With a shallow network like this, you likely don't need layer normalization, which is a bit more complicated to implement. Because this task is relatively simple, you don't need a very well-tuned architecture to make this work. You will implement all of these components from scratch.

You will want to form queries, keys, and values matrices with linear layers, then use the queries and keys to compute attention over the sentence, then combine them with the values. You'll want to use `matmul` for this purpose, and you may need to transpose matrices as well. Double-check your dimensions and make sure everything is happening over the correct dimensions. Furthermore, the division by $\sqrt{d_k}$ in the attention paper may help stabilize and improve training, so don't forget it!

Transformer Building the Transformer will involve: (1) adding positional encodings to the input (see the `PositionalEncoding` class; but we recommend leaving these out for now) (2) using one or more of your TransformerLayers; (3) using Linear and softmax layers to make the prediction. You are simultaneously making predictions over each position in the sequence. Your network should return the log probabilities at the output layer (a 20x3 matrix) as well as the attentions you compute, which are then plotted for you for visualization purposes in `plots/`.

Training A skeleton is provided in `train_classifier`. We have already formed input/output tensors inside `LetterCountingExample`, so you can use these as your inputs and outputs. The training code

should make simultaneous predictions at all timesteps and accumulate losses over all of them simultaneously. NLLLoss can help with computing a "bulk" loss over the entire sequence.

Without positional encodings, your model may struggle a bit, but you should be able to get at least 85% accuracy with a single-layer Transformer in a few epochs of training. The attention maps should also show some evidence of the model attending to the characters in context.

1.2 Q1 (25 %)

Now extend your Transformer classifier with positional encodings and address the main task: identifying the number of letters of the same type **preceding** that letter. Run this with `python letter_counting.py`, no other arguments. Without positional encodings, the model simply sees a bag of characters and cannot distinguish letters occurring later or earlier in the sentence (although loss will still decrease and something can still be learned).

We provide a `PositionalEncoding` module that you can use: this initializes a `nn.Embedding` layer, embeds the *index* of each character, then adds these to the actual character embeddings.[2] If the input sequence is `the`, then the embedding of the first token would be $\text{embed}_{\text{char}}(\text{t}) + \text{embed}_{\text{pos}}(0)$, and the embedding of the second token would be $\text{embed}_{\text{char}}(\text{h}) + \text{embed}_{\text{pos}}(1)$.

Your final implementation should get **over 95% accuracy** on this task. **Our reference implementation achieves over 98% accuracy in 5-10 epochs of training taking 20 seconds each using 1-2 single-head Transformer layers (there is some variance and it can depend on initialization).** Also note that **the autograder trains your model on an additional task as well.** You will fail this hidden test if your model uses anything hardcoded about these labels (or if you try to cheat and just return the correct answer that you computed by directly counting letters yourself), but any implementation that works for this problem will work for the hidden test.

Debugging Tips As always, make sure you can overfit a very small training set as an initial test, inspecting the loss of the training set at each epoch. You will need your learning rate set carefully to let your model train. Even with a good learning rate, it will take longer to overfit data with this model than with others we've explored! Then scale up to train on more data and check the development performance of your model. Calling `decode` inside the training loop and looking at the attention visualizations can help you reason about what your model is learning and see whether its predictions are becoming more accurate or not.

If everything is stuck around 70%, you may not be successfully training your layers, which can happen if you attempt to initialize layers inside a Python list; these layers will not be "detected" by PyTorch and their weights will not be updated during learning.

Consider using small values for hyperparameters so things train quickly. In particular, with only 27 characters, you can get away with small embedding sizes for these, and small hidden sizes for the Transformer (100 or less) may work better than you think!

1.3 Q2 (5 %)

Look at the attention masks produced. Describe in 1-3 sentences what you see here, including what it looks like the model is doing and whether this matches your expectations for how it should work. Please include your Q2 and Q3 discussion in the ipynb file of Part 3.

[2]The drawback of this in general is that your Transformer cannot generalizes to longer sequences at test time, but this is not a problem here where all of the train and test examples are the same length. If you want, you can explore the sinusoidal embedding scheme from Attention Is All You Need **?**, but this is a bit more finicky to get working.

1.4 Q3 (5 %)

Try using more Transformer layers (3-4). Do all of the attention masks fit the pattern you expect? Describe in 1-3 sentences what you see in the "less clear" attention masks.

2 Transformer for Language Modeling (35 %)

In this second part, you will implement a Transformer language model. This should build heavily off of what you did for Part 1, although for this part you are allowed to use off-the-shelf Transformer components.

For this part, we use the first 100,000 characters of `text8` as the training set. The development set is 500 characters taken from elsewhere in the collection. Your model will need to be able to consume a chunk of characters and make predictions of the next character at each position simultaneously. Structurally, this looks exactly like Q1, although with 27 output classes instead of 3.

Getting started Run:

```
python lm.py
```

This loads the data, instantiates a `UniformLanguageModel` which assigns each character an equal $\frac{1}{27}$ probability, and evaluates it on the development set. This model achieves a total log probability of -1644, an average log probability (per token) of -3.296, and a perplexity of 27. Note that exponentiating the average log probability gives you $\frac{1}{27}$ in this case, which is the inverse of perplexity.

The `NeuralLanguageModel` class you are given has one method: `get_next_char_log_probs`. It takes a context and returns the log probability distribution over the next characters given that context as a numpy vector of length equal to the vocabulary size.

2.1 Q4 (35%)

Implement a Transformer language model. This will require: defining a PyTorch module to handle language model prediction, implementing training of that module in `train_lm`, and finally completing the definition of `NeuralLanguageModel` appropriately to use this module for prediction. Your network should take a chunk of indexed characters as input, embed them, put them through a Transformer, and make predictions from the final layer outputs.

Your final model must **pass the sanity and normalization checks, get a perplexity value less than or equal to 7, and train in less than 10 minutes.** Our Transformer reference implementation gets a perplexity of 6.3 in about 6 minutes of training. However, this is an unoptimized, unbatched implementation and you can likely do better.

Network structure You can use a similar input layer (Embedding followed by PositionalEncoding) as in Part 1 to encode the character indices. You can use the PositionalEncoding from Part 1. You can then use your Transformer architecture from Part 1 or you can use a real `nn.TransformerEncoder`,[3] which is made up of TransformerEncoderLayers.

Note that unlike the Transformer encoder you used in part 1, for Part 2 you must be careful to use a **causal mask** for the attention: tokens should not be able to attend to tokens occurring after them in the sentence, or else the model can easily "cheat" (consider that if token n attends to token $n+1$, the model can store the identity of token $n+1$ in the nth position and predict it at the output layer). Fortunately it should be very easy to spot this, as your perplexity will get very close to 1 very quickly and you will fail the sanity check. You can use the `mask` argument in `TransformerEncoder` and pass in a triangular matrix of zeros / negative infinities to prevent this.

[3] https://pytorch.org/docs/stable/generated/torch.nn.TransformerEncoder.html

Training on chunks Unlike in Part 1, you are presented with data in a long, continuous stream of characters. Nevertheless, your network should process a chunk of characters at a time, simultaneously predicting the next character at each index in the chunk.

You'll have to decide how you want to chunk the data for both training and inference. Given a chunk, you can either train just on that chunk or include a few extra tokens for context and not compute loss over those positions. This can improve performance a bit because every prediction now has meaningful context, but may only make a minor difference in the end.

Start of sequence In general, the beginning of any sequence is represented to the language model by a special start-of-sequence token. **For simplicity, we are going to overload space and use that as the start-of-sequence character.** That is, when give a chunk of 20 characters, you want to feed space plus the first 19 into the model and predict the 20 characters.

Evaluation In this case your model is evaluated on perplexity and likelihood, which rely on the probabilities that your model returns. **Your model must be a "correct" implementation of a language model.** Correct in this case means that it must represent a probability distribution $P(w_i|w_1, \ldots, w_{i-1})$. You should be sure to check that your model's output is indeed a legal probability distribution over the next word.

Batching Batching across multiple sequences can further increase the speed of training. While you do not need to do this to complete the assignment, you may find the speedups helpful. You should be able to do this by increasing the dimension of your tensors by 1, a batch dimension which should be the first dimension of each tensor. The rest of your code should be largely unchanged. Note that you only need to apply batching during training, as the two inference methods you'll implement aren't set up to pass you batched data anyway.

Tensor manipulation `np.asarray` can convert lists into numpy arrays easily. `torch.from_numpy` can convert numpy arrays into PyTorch tensors. `torch.FloatTensor(list)` can convert from lists directly to PyTorch tensors. `.float()` and `.int()` can be used to cast tensors to different types. `unsqueeze` allows you to add trivial dimensions of size 1, and `squeeze` lets you remove these.

3 Applying Large Language Models to Code Generation and Math Reasoning (30%)

In this section, you will work with pre-trained large language models (LLMs) to tackle tasks in code generation and mathematical reasoning. The focus is on exploring different prompting strategies to enhance the model's performance on these tasks without additional training. This part of the assignment is designed to give you practical experience in leveraging LLMs for specific applications by using prompts effectively.

You will use the HumanEval [Chen et al., 2021] and GSM8K [Cobbe et al., 2021] benchmarks for evaluating the model's performance on code generation and math reasoning tasks, respectively. These benchmarks are designed to test the model's ability to generate code and solve math problems that require logical reasoning and multi-step calculations.

3.1 Load model and tokenizer (not graded)

First, you will set up a Large Language Model (LLM) for use in the subsequent tasks. The model of choice is CodeLlama-7B [Rozière et al., 2023], selected for its balance between size and performance, making it suitable for inference on Colab T4 GPUs. You will load this model in a 4-bit inference mode to optimize resource usage. This setup involves implementing a class named `LLM` that facilitates loading the model

and generating text completions based on provided prompts. Refer to the HuggingFace LLM Tutorial for guidance on using their toolkit, ensuring you do not use the `pipeline` function for this assignment.

3.2 Q5: Zero-shot Code Generation (5%)

In the zero-shot code generation task, your objective is to prompt the model to generate code solutions without any prior examples specific to the task at hand. This approach tests the model's ability to apply its pre-trained knowledge to generate relevant code based on the problem description provided in the prompt. The HumanEval benchmark, consisting of 164 programming challenges, will be used to evaluate the model's performance in generating correct and executable code solutions.

In this problem, you will implement `HumanEvalEvaluator` inherit from `Evaluator` to evaluate the model on the HumanEval code generation benchmark. Before you dive into `HumanEvalEvaluator`, please implement the `generate_completions` function of `Evaluator`.

After that, you can switch to `HumanEvalEvaluator`, where we have implemented almost all functions except the `postprocess_output` function. Since the language model keeps generating in the loop until archive `max_new_tokens`, we need to extract the solution from generated completions based on the stop sequences provided in the `postprocess_output` function.

After implementing `HumanEvalEvaluator`, you can run `human_eval_evaluator.evaluate` to evaluate the HumanEval benchmark. Based on the original paper and our reference code solution, you will get near 30% execution accuracy.

3.3 Q6: Few-shot Math Reasoning (10%)

Few-shot learning in math reasoning involves providing the model with a small number of example problems and solutions to help it grasp the task's requirements. This technique is aimed at enhancing the model's ability to solve new math problems by learning from the structure and logic of the provided examples.

The GSM8K benchmark, which includes a diverse set of grade-school level math word problems, will be used for this task. In this part, we evaluate the first 100 examples of GSM8K's test set, employing 8-shot in-context examples provided in the ipython notebook. You should implement all functions in `GSM8KEvaluator`.

`load_data` You should load this dataset with the HuggingFace datasets library from GSM8K. Make sure that you load the first 100 examples from the test split in the main subset. You may consider processing the decimal separator of answer numbers.

`build_prompts` In this function, you should build 8-shot direct prompts with the short answer. The prompt template should be:

Answer the following questions.

Question: {$Q_{example1}$}

Answer: {$A_{example1}$}

Question: {$Q_{example2}$}

Answer: {$A_{example2}$}

...

Question: {$Q_{inference}$}

Answer:

`calculate_metrics` After inference, you could implement postprocess and accuracy metrics to evaluate the performance.

3.4 Q7: Few-shot Chain-of-Thought Math Reasoning (5%)

Chain-of-thought prompting [Wei et al., 2022] encourages the model to generate intermediate steps or reasoning processes when solving problems, rather than jumping directly to the final answer. In few-shot chain-of-thought math reasoning, the model is given a few examples where the solution process is detailed step-by-step. This teaches the model to approach new problems by generating a reasoned sequence of steps leading to the solution, which can improve performance on complex reasoning tasks.

We provided the chain-of-thought demo answers `cot_answer` and you should implement the `GSM8KCoTEvaluator` to finish the CoT evaluation process.

3.5 Q8: Few-shot Program-of-Thought Math Reasoning (10%)

Program-of-thought prompting [Chen et al., 2022] is a variation of chain-of-thought reasoning where the model generates not just a sequence of reasoning steps, but a structured, program-like set of instructions or operations that solve the problem. This can involve more explicit structuring of the solution process, potentially making it easier for the model to handle more complex or multi-step problems. In a few-shot setting, the model learns from a small number of examples how to construct these program-like solutions for new problems.

These approaches leverage the flexibility and generative capabilities of large language models to tackle specific tasks in novel ways. By carefully designing prompts and leveraging few-shot learning techniques, it's possible to significantly enhance a model's performance on tasks like code generation and mathematical reasoning without the need for extensive retraining or fine-tuning on task-specific datasets.

This part involves executing the solution function to get the final results. We provided a Pythono executor to execute generated code solution. Please implement `GSM8KPoTEvaluator` in your solution. The program-of-thought prompt template should be:

Q: $\{Q_{example1}\}$

solution in Python:

$\{A_{example1}\}$

...

Question: $\{Q_{inference}\}$

solution in Python:

Deliverables and Submission

You will submit three files in this submission: `trainsformer.py` and `trainsformer_lm.py` for Part 1 & Part 2, `UNIVERSITYNUMBER.ipynb` for Part 3.

Part 1 & Part 2

You should only upload two files (`trainsformer.py` and `trainsformer_lm.py`) for Part 1 and Part 2 on Moodle.

Make sure that the following commands work (for Parts 1 and 2, respectively) before you submit and you pass the sanity and normalization checks for `lm.py`:

```
python letter_counting.py
python lm.py --model NEURAL
```

These commands should run without error and train in the allotted time limits.

Part 3

Similar to assignment 1, please clear all output and rerun all cells to ensure they are executable and retain all logs. Submit your ipynb file with the name `UNIVERSITYNUMBER.ipynb`

References

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harrison Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, David W. Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William H. Guss, Alex Nichol, Igor Babuschkin, Suchir Balaji, Shantanu Jain, Andrew Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew M. Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *ArXiv*, abs/2107.03374, 2021. URL https://api.semanticscholar.org/CorpusID:235755472.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *ArXiv*, abs/2211.12588, 2022. URL https://api.semanticscholar.org/CorpusID:253801709.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *ArXiv*, abs/2110.14168, 2021. URL https://api.semanticscholar.org/CorpusID:239998651.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, I. Evtimov, Joanna Bitton, Manish P Bhatt, Cristian Cantón Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre D'efossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code. *ArXiv*, abs/2308.12950, 2023. URL https://api.semanticscholar.org/CorpusID:261100919.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Huai hsin Chi, F. Xia, Quoc Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models. *ArXiv*, abs/2201.11903, 2022. URL https://api.semanticscholar.org/CorpusID:246411621.