Project 2: Train Your Own GPT

May 2025

This project is in team. Please form a 2 to 3 people group to finish the project. In final report, you should explain in detail which work you are responsible for in the team. Also, you are encouraged to dig deeper in every method and analysis, insightful and extraordinary performance will be graded with extra bonus. Using any GPT to directly coding is forbidden.

1 Introduction

The era of large language models (LLMs) has transformed natural language processing, with models like GPT-3 demonstrating remarkable abilities in text generation, translation, and question answering. This assignment aims to provide hands-on experience with language models, from implementing foundational architectures to fine-tuning pre-trained models on specialized datasets. You will explore model training, evaluation metrics like perplexity, and domain-specific fine-tuning, gaining insights into the practical applications of LLMs.

2 Part A: Implementing Language Models

Background: Language Modeling. Language modeling is a central task in NLP and language models can be found at the heart of speech recognition, machine translation, and many other systems. In this homework, you are required to solve a language modeling problem by designing and implementing recurrent neural networks (RNNs), as well as Transformers. A language model is a probability distribution over sequences of Given such a sequence $(\mathbf{x}_1, \ldots, \mathbf{x}_m)$ with length m, it assigns a probability $P(\mathbf{x}_1, \ldots, \mathbf{x}_m)$ to the whole sequence of words. In detail, given a vocabulary dictionary of words $(\mathbf{v}_1, \ldots, \mathbf{v}_m)$ and a sequence of words $(\mathbf{x}_1, \ldots, \mathbf{x}_m)$, a language model predicts the following word $\mathbf{x}_{t+1}$ by modeling: $P(\mathbf{x}_{t+1} = \mathbf{v}_j \mid \mathbf{x}_1, \ldots, \mathbf{x}_t)$ where $\mathbf{v}_j$ is a word in the vocabulary dictionary.

Conventionally, we evaluate our language model in terms of perplexity (PP). *perplexity* is a standard metric to evaluate how well a model predicts a sample. Perplexity (PPL) is defined as the exponentiated average negative log-likelihood of a sequence. Formally, if a model with parameters θ assigns probabilities $p_\theta(x_i \mid x_{<i})$ to each token x_i in a sequence $X = (x_1, x_2, \ldots, x_T)$, then:

$$\mathrm{PPL}(X) = \exp\left(-\frac{1}{T} \sum_{i=1}^{T} \log p_\theta(x_i \mid x_{<i}) \right).$$

This is equivalent to the exponent of the cross-entropy loss. Intuitively, perplexity measures how surprised the model is by the test data: a lower perplexity indicates the model predicts the text

more confidently (and thus more accurately on average), whereas a higher perplexity means the model finds the text more unpredictable.

2.1 Dataset

We use the Penn Treebank dataset, a standard corpus for language modeling, available via the Hugging Face Datasets library. It contains approximately 929,000 training tokens from Wall Street Journal articles. This dataset have two parts: the training set and validation set. The directory structure consists of two parts:

- `"./src/"` contains the start code.
- `"./data/"` contains the train and test set.

You need to fill in the model definition in `./src/model.py` to implement RNN, LSTM and Transformer. **You need to implement at least one model from scratch**. For the rest two, you can directly use models from `torch.nn` package. Remember to tune the hyperparameter like learning rate, and set proper amount of parameters. Make sure your training go well by plotting the training loss and test loss.

Part B: Comparing Across Models and Domains

Do following things

1. **Calculate perplexity during training:** For each model (RNN, LSTM, Transformer) in Part A, compute the perplexity on the training set (and validation set if applicable) at various points of training (for example, after each epoch).
2. **Evaluate on a different corpus:** To investigate how well the models generalize to a new domain, take a different text corpus (preferably in a different style or domain than the training data). For example, evaluate the perplexity on WikiText-2 (Wikipedia articles), or a few chapters of a novel or the Tiny Shakespeare text as the new corpus, or vice versa. Ensure this new corpus is not something the model saw during training. Compute the perplexity of each of your trained models on this new corpus.
3. **Compare perplexities across domains:** Record the perplexity of each model on both the training domain and the new domain. Analyze the results. Which model has the smallest increase in perplexity when moving to the new domain? Which models performance degrades the most? This gives insight into which architecture generalizes better from the given training data.

2.2 Questions to answer in your analysis:

Why does the perplexity change when you evaluate the model on a different corpus? Relate this to the concept of the training data distribution versus a new data distribution. Use the numerical results you obtained to support your reasoning.

2.3 Sampling and Evaluating Generated Text

Beyond perplexity and loss curves, it's informative to qualitatively evaluate your language models by examining the text they generate. In this task, you will use each trained model from part A to generate sample sequences and then assess their coherence and fluency.

1. **Sampling from the models:** For each model (RNN, LSTM, Transformer), generate a few sentences or a short paragraph of text. You can do this by feeding a starting prompt or even just a start-of-sequence token, then iteratively sampling the next token from the model's output distribution and feeding it back in as input for the subsequent prediction. You may use a simple stochastic sampling (possibly with a temperature parameter to control randomness) or greedy decoding (always pick highest probability token, though this can lead to repetitive outputs). Ensure you generate a sufficiently long sequence (e.g., 50-100 tokens) to see the model's ability to maintain context.

2. **Use the same prompts:** To compare models, it can be insightful to start them with the same initial text prompt. For example, you might input a prompt like "The meaning of life is" or the first few words of a sentence, then let each model continue from there. Alternatively, you can let them start with an empty prompt (or a special start token) to see what each model produces on its own.

Compare the generated texts from the RNN, LSTM, and Transformer. You can analyze from coherence, fluency, repetition, or other distinctive differences. Validate your analysis through concrete examples.

3 Part C: Fine-Tuning a Pretrained Language Model

Modern language modeling often leverages large pre-trained models that have been trained on massive general-purpose datasets. Instead of training from scratch, we can take an existing pre-trained language model and fine-tune it on a specific domain or task. In this task, you will pick an open-source pretrained language model (with around 100 million to 500 million parameters) and fine-tune it on a small domain-specific dataset.

Choosing a Pretrained Model: A good choice is **GPT-2** by OpenAI, which comes in various sizes. The "small" GPT-2 model has about 124 million parameters, and the "medium" GPT-2 has 355 million parameter. These models are available through Hugging Face's model hub. You may also use Qwen2-0.5B pretrained model, which is more powerful. You need to learn to use the Hugging Face Transformers library to load the model and tokenizer. For example, load GPT2 in Python be like:

```
from transformers import AutoTokenizer, AutoModelForCausalLM

tokenizer = AutoTokenizer.from_pretrained("gpt2")  # GPT-2 small
model = AutoModelForCausalLM.from_pretrained("gpt2")
```

This will download the pre-trained GPT-2 small model. The AutoTokenizer ensures your text is tokenized the same way the model was pre-trained (GPT-2 uses byte-pair encoding). Loading Qwen2-0.5B is similar.

3.1 Task: Supervised Finetuning

Preparing a Domain-Specific Dataset: Choose a domain that interests you (ideally related to your academic major or a field you know well) and collect a relatively small text dataset in that domain. Some examples:

- Medical domain MIMIC-III clinical notes for ICU stays (over 2 million de-identified notes)
- Legal domain AUEB-NLP ECtHR cases dataset (11 k enriched European Court of Human Rights decisions)
- Biomedical QA PubMedQA dataset (100 k+ research QA pairs from PubMed abstracts)
- Finance SEC Edgar 10-K filings dataset (annual reports filed to the U.S. SEC)
- Something else?

3.2 How to Finetune

We will fine-tune the model on the domain text using the language modeling objective (next-token prediction) essentially continuing the model's training on the new data.

1. **Data preprocessing:** Tokenize your domain text with the same tokenizer (for GPT-2, use the GPT-2 tokenizer loaded above). You may need to split the text into sequences (for example, chunks of a certain length, like 128 or 256 tokens, depending on memory and context length considerations).
2. **Training setup:** Set the model to training mode and use an optimizer like Adam. Because the model is already pre-trained, you can use a relatively low learning rate (e.g., 5e-5 or 1e-4) to avoid damaging the pre-trained weights too quickly. Optionally, you can use the Hugging Face *Trainer* API for convenience.
3. **Fine-tune the model:** Train the model on your domain dataset for a certain number of epochs (maybe a few epochs, since the dataset is small and the model can overfit quickly). Watch the training loss; it should decrease. If you have a bit of unrelated validation text in the same domain, monitor the validation loss to avoid overfitting.
4. **Save the fine-tuned model:** After training, save your model (and tokenizer if needed) so you can load it for evaluation. If using `Trainer`, it might do this automatically. If doing manually, you can call `model.save_pretrained()` and `tokenizer.save_pretrained()` to save the weights and tokenizer files.

Write your own finetuning code on selected dataset. Report the training loss along the process. Now you have a model fine-tuned to your domain, it's time to evaluate how the fine-tuning affected its performance. We want to see if the model has improved its capability in the domain-specific context and whether it has maintained its general language abilities. You may design a couple of simple evaluation tasks or experiments to measure changes after fine-tuning. Consider both qualitative and quantitative evaluations.
Some possible evaluation approaches are:

- Test the perplexity on same domain text.
- Domain-Specific Question Answering or Prompt Completion.
- Fill-in-the-Blank Test.
- Some else?

Reflection: Based on your evaluation, write a brief reflection on the impact of fine-tuning: Did the fine-tuned model indeed improve on domain-specific tasks? What differences did you observe in its behavior compared to the base model? How might the size of the fine-tuning dataset and the fine-tuning duration affect these outcomes?
This analysis will help solidify your understanding of transfer learning in language models and the trade-offs involved in specializing a general model to a narrower domain.

Bonus (Optional): RLHF

(This task is optional and for the curious, going beyond the core assignment.) Modern state-of-the-art LLMs (like ChatGPT or GPT-4) undergo a fine-tuning stage known as *Reinforcement Learning from Human Feedback (RLHF)* to make them more aligned with user expectations and values. In this task, you are encouraged to explore, at a high level or through a small experiment, how RLHF techniques like Proximal Policy Optimization (PPO) or Direct Preference Optimization (DPO) can be applied to language models using open-source tools.

3.3 What is RLHF?

After a model is trained on vast data (predicting next tokens), we often want it to follow instructions or prefer certain kinds of responses. RLHF involves collecting human feedback on model outputs (for example, humans rank which responses they like better), training a *reward model* from this feedback, and then further fine-tuning the language model to maximize this reward (making it more likely to produce responses humans prefer). This is typically done with reinforcement learning algorithms. PPO is a popular reinforcement learning algorithm used in this context to gently nudge the model's outputs towards higher reward while not deviating too much from the pre-trained policy. DPO is a newer approach that simplifies the process by turning the preference optimization into a direct supervised objective, avoiding some complexities of reinforcement learning.

3.4 Some suggestions

Read about the **TRL (Transformer Reinforcement Learning)** library by Hugging Face, which provides implementations of PPO and DPO for language models. The TRL library allows you to plug in a language model and a reward function (or preference dataset) to perform RLHF training. Their documentation and examples (such as using TRL to fine-tune a model on a dataset of human preferences) are a great starting point.
Describe what approach or resource you investigated. For example, you might summarize the objective of PPO vs DPO, or outline the steps needed to perform RLHF. If you run any code, describe the setup (which model, what reward or preference data, how long training was, etc.) and the outcomes (did the model's behavior change as expected?). Even if you do not run code, you can discuss conceptually how RLHF would be applied to your fine-tuned model to further align it, demonstrating understanding of the process.