HW 1: CalcGPT

Version: April 10, 2023

Your goal is to disrupt the calculator industry using GPT. Instead of old-fashioned custom-built algorithms for computing answers to arithmetic problems, we will design and test a system that uses a pre-trained large language model to solve these problems.

Summary of steps:

1. Convert a simple addition problem to a string using **two** methods:
   Baseline: Input problem directly as a string "1+2="
   Choose your own method to compare. These are just some ideas, feel free to be creative here:
   - Convert digits to words
   - Convert arithmetic to a word problem ("Jack has three apples, then receives two more…")?
   - Use a prompt ("You are a calculator…") ?
   - Roman numerals ("VI + IV = X")?
   - Use a code ("A=5, B=6, A+B=")
   - Use algebra ("if x=5 and y=7 and z = x+y, then z=")
   - In-context learning, use examples of desired result ("2+3=5, 7+11=18, 8+4=")
   - Modify the problem in ways that are trivial for humans. E.g. ("700 + 1100 = 1800")

   Note that the goal is not necessarily to get the best accuracy. Getting the *worst* accuracy with something that is easy for humans is also interesting.

2. Use a LLM (default choice is GPT2-large) to generate a response
   A very simple approach is sufficient, you could generate a single response and take the first "word" to be the answer. Some things you might consider:
   a. Change temperature / sampling options to get deterministic results?
   b. Purposely sample multiple responses and merge or take the best candidate?

3. Convert answer to an integer
   To measure the accuracy, we'll need to convert the answer to an integer somehow. I recommend using torch.nan for cases that cannot be decoded.

4. Analyze and visualize the results. Detailed instructions about what should be required are given in the write-up section.
   a. Top-line accuracy for baseline and new strategy
   b. Scatter plots of correct/incorrect answers
   c. Train a classifier to see if you can predict which sums are more likely to produce incorrect answers.

5. Submit code and write-up on canvas.

Extra credit:

- Compare other math problems (multiplication, exponentiation, algebra, calculus…)
- Compare with other large LLMs not on Huggingface (Alpaca, …)
- Use API calls to compare with closed LLMs (ChatGPT, GPT4). This may require going to harder problems, as it is difficult to get ChatGPT to make a mistake
- Check if fine-tuning the model on some range of numbers improves performance on numbers outside of that range
- Self-reflection prompting to verify if answers are correct
- Any other creative experiment that goes far beyond the assignment
- Study the next token distributions - are there multiple numbers with high probability for each problem, or is it peaked on one number? Is the entropy of the next token distribution different for different problems?

Code:

You can find some template code attached at the bottom, with TODOs of things to fill in. Comments also contain info about points and requirements. Please keep the function names for easier grading. (If you want to try a complex strategy that requires refactoring, just put a comment in the predefined function telling us where to look.) We prefer Ipython notebooks for the code (also easier if you're running on Google Colab).

Write-up instructions:

The write-up should be as succinct as possible. Include the following sections, with the requirements in each section.

(a) Method for encoding strings (1 point)
Include an example, and an example of how it is tokenized

(b) Method for generating text (1 point)
Briefly describe the LLM used, and any choices of hyper-parameters

(c) Method for decoding strings (1 point)
Include an example, and example of how the tokenized output look

(d) Results (6 points)
Show the following results for both the baseline and comparison strategy

- How accurate was the baseline and your comparison method? (i.e., what fraction of problems give the correct answers)
- Plot a scatter plot for each method of problems "x1+x2" with x1,x2 on each axis, and different plot markers to indicate whether the answer was correct. Axes must be labeled. Include a legend so we know which markers are correct versus incorrect. Each subplot should have a title so we know which is the baseline and which is your comparison strategy.
- Then train a classifier (using scikit-learn, for example) to see if you can predict which sum problems will be answered correctly. You can use any classifier that seems reasonable (e.g. decision tree, linear).

- How accurately did the classifier predict whether the answer was correct (on the training data)? This is attempting to gauge whether the LLM is wrong randomly or in some systematic ways.
- Use "contour" to visualize the classifier boundary on the same plot as your scatter plot. Comment in the text about the choice of classifier and how we should interpret the results.

(e) AI collaboration statement (1 point to include)
AI "collaborators" are allowed (Github copilot, ChatGPT). I'm just curious how they were used and what worked, it won't affect the grade.

(f) Extra credit (optional)

Collaboration:

Feel free to discuss ideas with classmates, but you should do the coding and write-up on your own. I also encourage you to discuss ideas and preliminary results on a Canvas discussion board. I think we could discover some interesting phenomena by pooling observations.
Also use the discussion board to discuss common issues - I suspect some of your fellow students will have more expertise at running these models than I do.
AI tools are allowed, please declare how they were used in the write-up.

Code Template

```
import torch as t
from transformers import GPT2LMHeadModel, GPT2Tokenizer  # pip install
transformers
from transformers import AutoModelForCausalLM, AutoTokenizer
import matplotlib.pyplot as plt
import time
import numpy as np
import sklearn
import pickle
import re  # regular expressions, useful for decoding the output


def create_dataset(i_start=0, i_end=50, operation=t.add):
   """(1 pt) Create a dataset of pairs of numbers to calculate an operation
on.
   DO NOT USE A FOR LOOP. Use pytorch functions, possibilities include
meshgrid, stack, reshape, repeat, tile.
   (Note you'll have to use for loops on string stuff in other functions)

   The dataset should be a tuple of two tensors, X and y, where X is a Nx2
tensor of numbers to add,
   and y is a N tensor of the correct answers.
   E.g., if i_start=0, i_end=2, then X should be
tensor([[0,0,1,1],[0,1,0,1]]).T and y should be tensor([0,1,1,2]).
```

```
    I recommend doing all pairs of sums involving 0-49, but you may modify
this.
    """
    # TODO
    return X, y

def load_LLM(default="EleutherAI/gpt-neo-2.7B", device='cpu'):
    """(1 pt) Load a pretrained LLM and put on device. Default choice is a
large-ish GPT-neo-2.7B model on Huggingface.
    Could also consider the "open GPT" from facebook: "facebook/opt-2.7b", or
others
    here: https://huggingface.co/models?pipeline_tag=text-generation
    Explicitly load model and tokenizer, don't use the huggingface "pipeline"
which hides details of the model
    (and it also has no batch processing, which we need here)
    """
    # TODO
    return model, tokenizer

def encode_problems(X, strategy='baseline'):
    """(1 pts) Encode the problems as strings. For example, if X is
[[0,0,1,1],[0,1,0,1]],
    then the baseline output should be ["0+0=", "0+1=", "1+0=", "1+1="]"""
    output_strings = []
    for xi in X:
        if strategy == 'baseline':
            # TODO: encode_string =
        else:
            # TODO: encode_string =
        output_strings.append(encode_string)
    return output_strings

def generate_text(model, tokenizer, prompts, verbose=True, device='cpu'):
    """(3 pts) Complete the prompt using the LLM.
    1. Tokenize the prompts:
https://huggingface.co/docs/transformers/preprocessing
        Put data and model on device to speed up computations
        (Note that in real life, you'd use a dataloader to do this efficiently
in the background during training.)

    2. Generate text using the model.
        Turn off gradient tracking to save memory.
        Determine the sampling hyper-parameters.
        You may need to do it in batches, depending on memory constraints

    3. Use the tokenizer to decode the output.
    You will need to optionally print out the tokenization of the input and
output strings for use in the write-up.
    """
```

```
    t0 = time.time()
    # TODO: tokenize
    # TODO: generate text, turn off gradient tracking
    # TODO: decode output, output_strings = ...

    if verbose:
        # TODO: print example tokenization for write-up
    print("Time to generate text: ", time.time() - t0)  # It took 4 minutes to
do 25000 prompts on an NVIDIA 1080Ti.
    return output_strings

def decode_output(output_strings, strategy='baseline', verbose=True):
    """(1 pt) Decode the output strings into a list of integers. Use "t.nan"
for failed responses.
    One suggestion is to split on non-numeric characters, then convert to int.
And use try/except to catch errors.
    """
    y_hat = []
    for s in output_strings:
        # TODO: y = f(s)
        y_hat.append(y)
    return y_hat

def analyze_results(X, y, y_hats, strategies):
    """(3 pts) Analyze the results.
    Output the accuracy of each strategy.
    Plot a scatter plot of the problems “x1+x2” with x1,x2 on each axis,
    and different plot markers to indicate whether the answer from your LLM was
correct.
    (See write-up instructions for requirements on plots)
    Train a classifier to predict whether the LLM gave the correct response
(using scikit-learn, for example)
    and plot the classifier boundary over the scatter plot with “contour”. (Use
whatever classifier looks appropriate)"""
    # TODO


if __name__ == "__main__":
    device = t.device("cuda" if t.cuda.is_available() else "cpu")  # Use GPU if
available
    device = t.device('mps') if t.backends.mps.is_available() else device  # Use
Apple's Metal backend if available

    X, y = create_dataset(0, 50)
    model, tokenizer = load_LLM(device=device)

    y_hats = []  # list of lists of predicted answers, y_hat, for each strategy
    strategies = ['baseline', 'new']
    for strategy in strategies:
```

```
        input_strings = encode_problems(X, strategy=strategy)
        output_strings = generate_text(model, tokenizer, input_strings,
device=device)
        output_strings = [out_s[len(in_s):] for in_s, out_s in
zip(input_strings, output_strings)]  # Remove the input string from generated
answer
        y_hats.append(decode_output(output_strings, strategy=strategy))

    analyze_results(X, y, y_hats, strategies)
```