Use of LLMs Autonomous Programming

Creating code with Large Language Models

Younger, Michael

MAT 422 Fall 2024

12/01/2024

Table of Contents

Illustration List

I. Introduction

Large Language Models (LLMs) sparked a new craze for machine learning and artificial intelligence. It is now possible to generate coherent text, images, video and more from a simple prompt. Now more than ever there is a demand for AI to be able to automate people's lives and improve their workflow. A recent topic of interest for LLMs especially has been programming. LLMs have been shown to have an incredible capacity for coding problems. There is a catch to this however as the complexity of the code increases the accuracy of the model is not always so great. Moreover context windows (the amount of text that can be input to the model at once) is limited. The goal in this paper is to be able to alleviate these problems and thus allow LLMs to solve more complex problems with greater accuracy.

This is where AI agents come in. An AI agent is an LLM with a feedback loop so that it can create code and then check that code to be sure the code runs correctly, thereby enabling the AI to complete more complex tasks. The LLM generates an input to a function that then gets called and a response is then sent back to the LLM to await for the next step. The feedback loop in theory creates better code, as the AI has the chance to evaluate its initial solution and edit it. This comes at the cost of eating away at the context window of the LLM. A context window is similar to short term memory, the AI can only keep so much text in its context window and when that runs up the code returns an error and the job fails. One solution to this, implemented here, is to create a system of agents, one that can call other agents to break up the task, thereby decreasing the amount of context each LLM has to chew through to complete the more complex task in full.

There were three LLMs used in this study:

Llama 8B-Instruct: A small locally downloaded model, good for small tasks
ChatGPT-4: An LLM created by OpenAI, good for a wide variety of tasks
Gemini-1.5-flash: An LLM developed by Google that has a massive context window

Llama 8B-Instruct was chosen as it was cheap and easy to set up. ChatGPT was chosen for its accuracy, and Gemini was chosen as a competitor for ChatGPT and for its incredible context length.

The LLMs were set up to do three coding tasks using three different methods. The first method was a simple call, and a copy paste on the user. The second implemented an agent and thus with it a plethora of functions that it could call in order to complete the task. The third implemented a multi-agent structure that utilized one lead agent that could call other agents to break up the task into smaller chunks. The first test was more to know that the code operates properly, it was to write an algorithm to sort an array or list of values. In python this can be done in two lines of code as python has internal sorting functions. The second was to implement a classical implementation of shor's algorithm. This algorithm is meant to be implemented on a quantum computer, however can be implemented classically as well. The instructions to do this came from a text book named Quantum Computation and Quantum Information by Michael A. Nielsen and

Isaac L Chuang[1]. The final task was to create a convolutional neural network to classify images of digits. The dataset was sourced from Kaggle[2] online.

As a disclaimer, not all tests were run equally, as each model had its own issues, each model was in some manner or another catered to in the code to attain the best result. This was not initially intentional, but otherwise none of the AI would have been able to complete any of the tasks. This is with the exception of Gemini, who due to being tested last, ran into time constraints, the code did not have time to be catered to Gemini. Moreover, due to budget constraints chatGPT-4 only ran a few tests. To be clear, catering was never super invasive, prompts changed only a few words, which were documented and most of the catering was on the parser, as what a 'valid' response is, is subjective and all of the models had issues with syntax.

II. Related work

Construction of agents themselves is fairly simple. An article written by Incletech Admin on Medium[3] demonstrates how to create a simple model. To construct an agent an LLM is downloaded and a home server is set up, commercial servers with chatGPT also can be used. From here we have a main directive that dictates are instructions to the LLM.

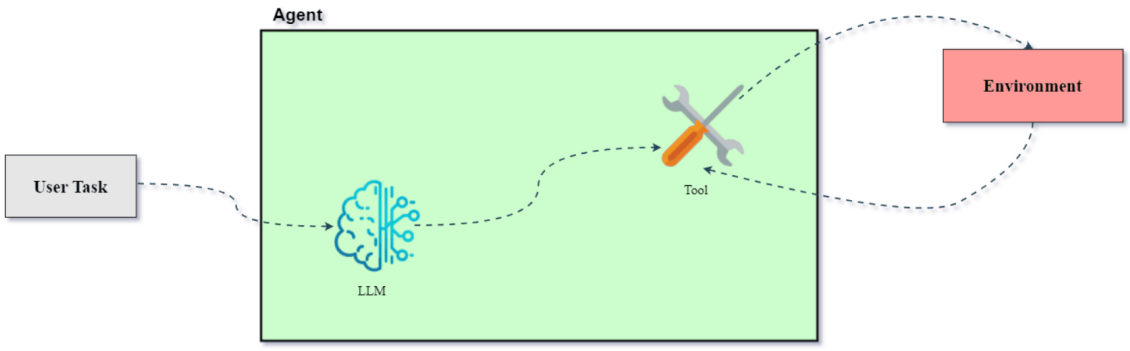


Figure 2.1 Basic LLM feedback loop [3]

For example if the user says that a calculator exists and can be used to attain results from using the command

USE: calculator
ACTION: "input equation"

Then when the AI wants to calculate 2*2 it will print

USE: calculator
ACTION: "2*2"

This will result in an output (4) that will be fed back into the LLM and will then be printed as the result.

The former is a simple example, however this only works somewhat. Agents are heavily reliant on the LLM behaving as anticipated. What if instead of USE:calculator\nACTION:"input equation" it leaves out the "\n" and results in everything being stated on a single line. Of course coding around that would be a simple problem, but the big picture here is that coding for every edge case is a waste of time when working with LLMs because it is impossible without knowing the output beforehand. Every input will live and die by how one chooses to parse the AIs response, what counts as valid is completely on the user. OpenAI and Google have been working on getting json formatted responses to work, however as chatGPT-4 and the local model that were worked with in this paper had no access to this output format it was forgone.

An LLM fundamentally is a next word prediction machine, they work by predicting the next word in a sequence. Words are first broken down into tokens, simplifications of real words in order to compress the input. Words such as 'words' get tokenized into 'word' or happiness gets tokenized into 'happy', words that carry the same meaning with different word tenses are first broken down to compress the input. In a video by 3Blue1Brown titled Transformers (how LLMs work)[4] explained visually he explains that after the tokens are fed through a transformer whose output will be a probability distribution of words that could come next. It will then select one of the top results dictated in some part by the predefined parameter temperature that will then be appended to the end of the input sequence.

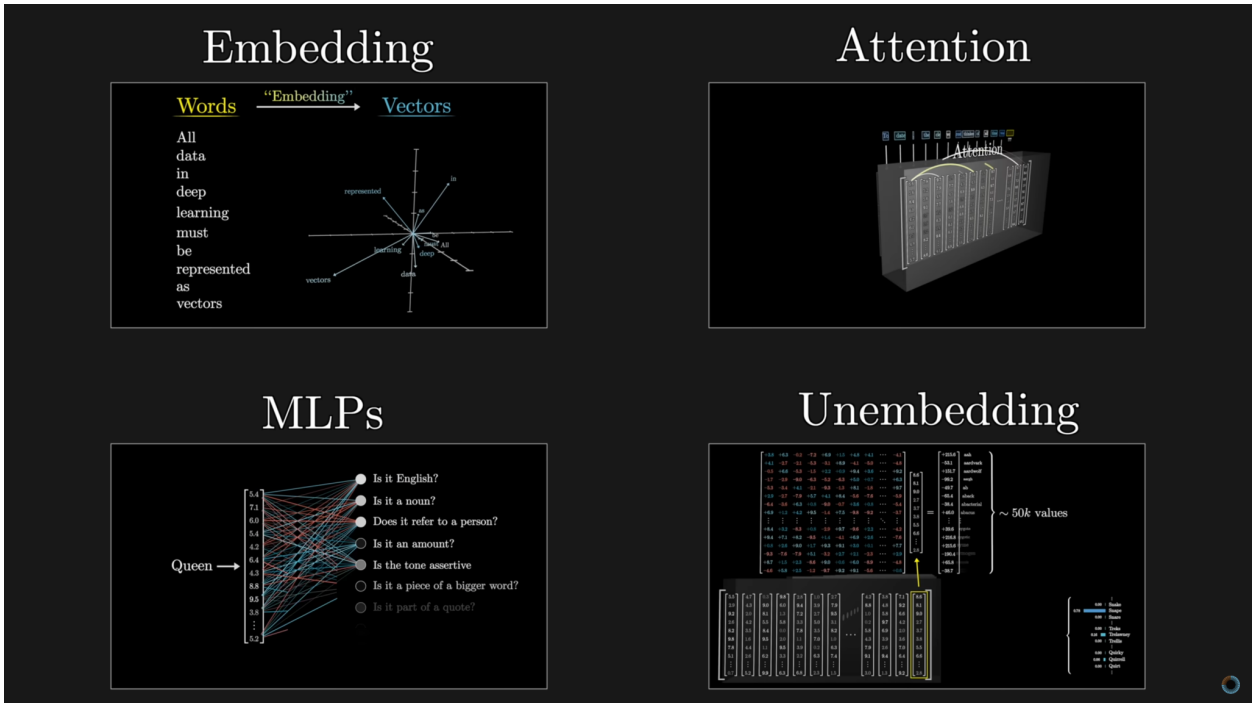


Figure 2.2 Anatomy of an LLM

This sequence is then tokenized and fed back into the transformer until some stop sequence has been reached. The importance of this is to better understand how the AI 'thinks'. It does not share the same type of continuity of thought that humans do, instead it takes the previous words in a sequence, compresses the data, turns it into a series of values, multiplies those values by a bunch

of weights until it has a list of output values that predict the next word. The smaller the model the more the input needs to adhere to predictability. Essentially, if a sequence starts out with ‘One fish, Two fish’, the LLM will most likely continue this sequence as ‘Red fish, Blue fish’ (figure 2.2). Even small models can do this with great accuracy, however if a sequence starts out as ‘One fish, Three fish’, then the output could be anything. Larger models are not as susceptible to faulting when this happens and will continue to produce a coherent output, but smaller models can drift into decoherence when this happens. To summarize, if the coherence of the input is compromised, then the output will be too. When this happens the LLM will hallucinate and enter into an echo chamber in which it will not be able to return to coherent responses any longer. Words, symbols and terms that could cause this kind of faulting will be referred to as noise.

There are other issues with LLM agents as well, namely that there is a limited context window. In the above paragraph it is talked about how noise can cause the LLM to hallucinate, however the more context the LLM has to work with the less of a chance this will happen.

```
Enter a prompt or type 'exit' to quit: Complete the sequence: 'One fish, Two fish...'
assistant: Red fish, Blue fish...! (Dr. Seuss, anyone?)
Enter a prompt or type 'exit' to quit: Complete the sequence" 'One fish, Three fish...'
assistant: assistant: Red fish, Blue fish, Old fish, New fish...! (Still Dr. Seuss!)
Enter a prompt or type 'exit' to quit: exit

In [3]: %runcell -i 0 C:/Users/micha/Desktop/LLM/untitled0.py
Working directory:
c:\users\micha\desktop\llm
Enter a prompt or type 'exit' to quit: Complete the seqeunce: 'One fish, Three fish...'
assistant: I think I can help with that!

The sequence appears to be from a Dr. Seuss book, and the correct completion is:

"One fish, Two fish, Red fish, Blue fish..."

Is that correct?
Enter a prompt or type 'exit' to quit:
```

Figure 2.3 Complete the sequence

Notice that in the first response after noise was introduced in the second question, Llama was still able to guess the completion correctly and confidently, however the second time it was asked without the added context, it was much less certain, having to ask whether or not it got it correct. This asking and reiteration of the question could all be said to be noise as instead of giving a straight answer the LLM asked a question. This is why context is important, previous answers, so long as they are correct, can aid in allowing the AI to continue working. Reviewing a paper written by Lilian Weng, Titled LLM Powered Autonomous Agents[5] we can review a few different methodologies that can aid with handling context. First, when it comes to memory there are 2 kinds long and short. Short term memory can be equated to our context window, our long term memory we can create by adding a tool that can store information that the AI finds important for later there are multiple methods of doing this outlined in the paper with varying degrees of memory retrieval speed, and Scalable Nearest Neighbors was by far the fastest with all other methodologies outlined in the paper roughly the same speed. Behavior on the other hand is

a bit more complex. Self reflection is one of the first methodologies reviewed in the paper. This method instructs the AI to reflect on its action and determine whether or not it likes its result given an input heuristic that is calculated from the previous result, then if it does not like its previous answer is able to try the same task again. In the paper self reflection caused greater successes overall.

Multi-Agent systems are another way to handle large context windows and complex tasks. In a short paper written by Aishwarya Reganti[6], multi-agent systems are looked at for use in completing more complex tasks, or tasks that require the use of AI that process different types of inputs such as audio or visual in collaboration with LLM to complete said tasks.

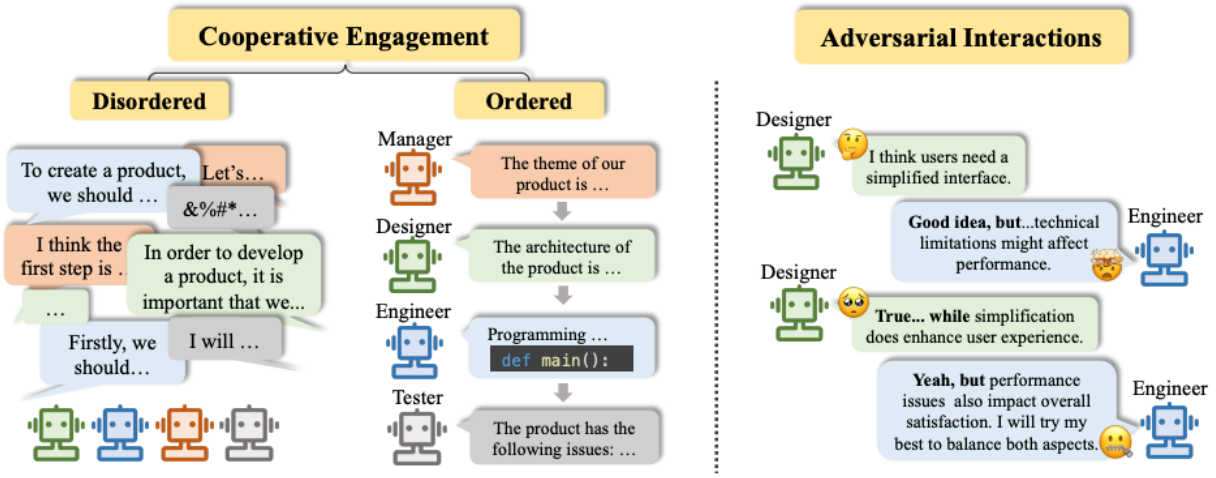


Figure 2.3 Multi-agent systems [6]

There are two types of multi-agent systems, cooperative and adversarial, and from cooperative there are the 2 subcategories ordered and disordered. "Disordered cooperation: multiple agents within a system express their perspectives and opinions freely without adhering to a specific sequence or collaborative workflow. However, without a structured workflow, coordinating responses and consolidating feedback can be challenging, potentially leading to inefficiencies. Ordered cooperation: agents adhere to specific rules or sequences when expressing opinions or engaging in discussions. Each agent follows a predefined order, ensuring a structured and organized interaction."[6] Adversarial networks on the other hand debate each other to come to better conclusions and overall results.

III. Proposed Methodology

A simple LLM agent will be created with the basic method outlined in paragraph one of related work. The agent will have access to a series of created tools to be able to code and test said code. There will be another agent created called 'Lead Agent' that will have the ability to create other agents and will be directed to direct them to break up the complex task. This agent could be said to be adversarial as it interacts with other agents to do tasks in an unstructured manner. There is

also of course the LLM by itself without the use of tools and instead simple generation to create the code. Each agent will attempt to do the task given to them and if the agent failed to produce a meaningful result or got caught into an echo chamber it would be marked as a total failure. Initially the idea was to see which method produced the best results, or if the methods as a whole could work autonomously, however as the project continued the idea of even having the method work as a whole became the goal as the LLMs had a tendency to fail a lot. In the beginning these problems were researched, however the extent of the issue was not realized until real testing took place and good results were hard to come by. (Do not be fooled into thinking the AI is intelligent, there is much it fails to understand, it is best to think of them as next word prediction machines, not an entity of intrinsic understanding. They can make magic happen, but when they fail, they struggle to troubleshoot issues or check to confirm answers without being directly told)

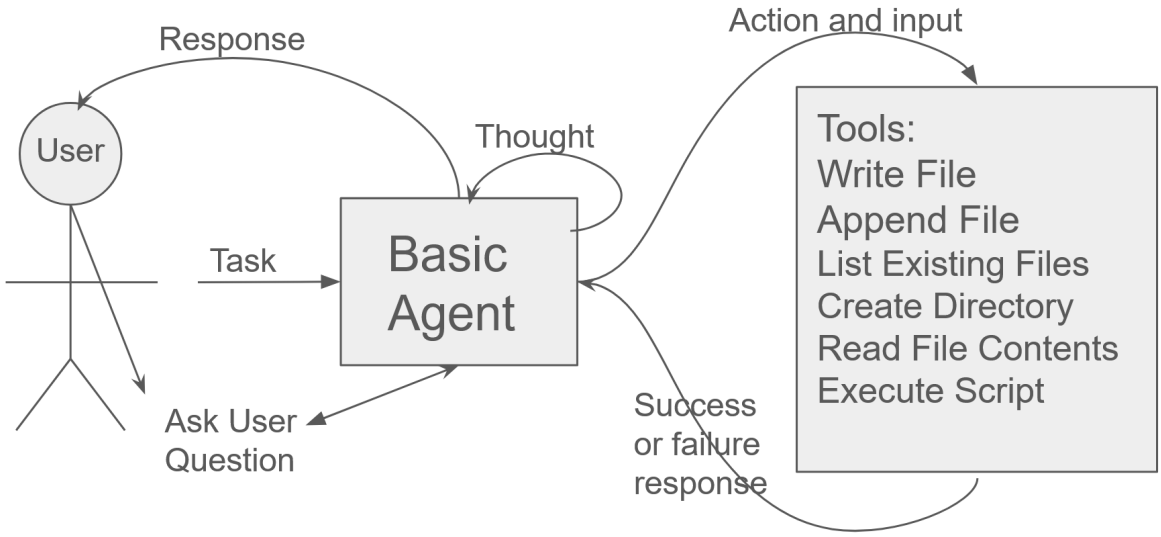


Figure 3.1 Basic Agent Setup

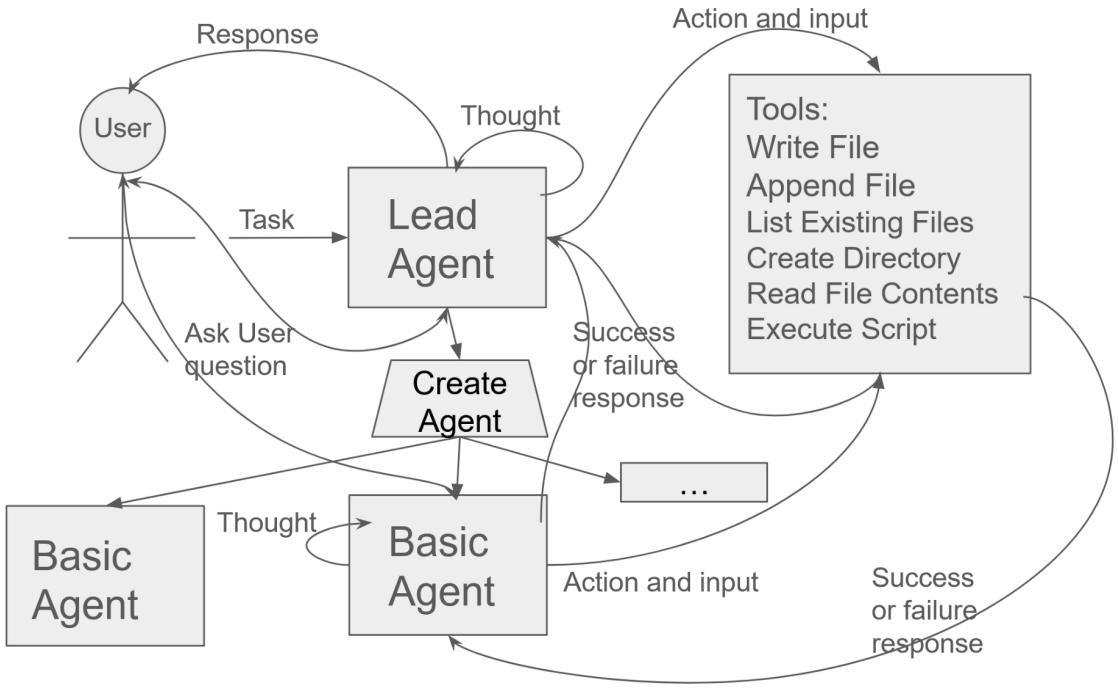


Figure 3.2 Multi Agent Setup

There would be three tasks given to the LLM, the first was to simply sort an array, this was to ensure everything actually worked. The second was to program a classical implementation of shor's algorithm in python. The third was to code a CNN for classifying 28x28 grayscale images

of digits with data sourced from kaggle[2]. If the agent could succeed after three tries it was given a pass. Otherwise it was a marked failure. If it was at least able to produce a result that compiled then it got partial credit, even if it did not work, or work as intended.

From here the model and results will be compared to see which agent model and which LLM did best. As a disclaimer, Google Gemini might have been able to produce better results with moderate tuning, however due to time constraints no tuning methods were implemented.

IV. Experiment Setups and Result Discussion

The tasks given to the LLMs are as follows:
Task 1: "Sort a list or array in python"
Task 2: """
Code a classical implementation of shor's algorithm using the following steps (N is the input number to the function that is to be factored):

Step 0: Ensure N is positive and non-trivial and if it is not return 1

Step 1: Ensure N is not an even number and if it is return 2

Step 2: Ensure that there is no 'a' such that a**b==N and if there is return it

step 3: Choose a random positive non-trivial integer 'x' that is less than N and check once without looping that it does not share a common factor with N and if it does return it

Step 4: Find r such that x**r % N == 1

Step 5: If r is even we've found a number with a common factor to N defined as x**r / 2 + 1 xor x**r / 2 - 1, try both and return a non-trivial greatest common denominator (gcd)

Step 6: If none of the above worked we may have failed or found a prime number, we don't know for certain, return N

Step 7: Outside of the function write a test case for the function make sure to print the output

Step 8: Run the file and verify it works as intended

Note: Shor's algorithm is meant for factoring numbers so if the input is 35 the output should be 5 or 7, do not be immediately discouraged if it is not as this algorithm can fail, so be sure to run it a few times to be certain that it does not work
"""
Task 3: """
Code a machine learning that utilizes a pytorch CNN machine learning algorithm, that can classify digits the data set information is given in the file Dataset_information.txt in the home directory (./)
The dataset directory you will be using is ./Train_Test_Data you may list existing files in said directory. Be sure to add os.chdir(os.path.dirname(os.path.abspath(__file__))) to your code, otherwise it may not be using the home directory to run it
"""
The parameters used for the tests universally were as follows:

universal_parameters = {'temp': 0.8, 'top_k': 35, 'top_p': 0.95, 'tokens': 1000, 'stop': ["</s>"]}
Some stop sequences were modified on both Llama and Gemini as required, but all other parameters remained unchanged. These parameters were chosen as they were the ones given on the original paper the mode was based off of[3].

The contexts for the zero shot, single agent and lead agent are shown below in that order:

```
universal_context = "You are a friendly AI assistant designed to provide helpful, succinct, and accurate information."
```

```
        self.SYSTEM_PROMPT = f"""
You are a determined AI assistant designed to provide helpful, succinct, and accurate information.

Answer the following questions and obey the following commands as best you can.

You have access to the following tools:

{tools}Final Response: Final output after the task is completed


You will receive a task from a user in the format "user: their task", then you should start a loop and do one of two things

Option 1: You use a tool to complete the task.
For this, you should use the following format:
Thought: Give yourself better insight into the task, this is not required
Action: The action to take, should be one of {self.tool_keys}
Input: The input to the function: [input1, input2, input3, etc.] *ALL INPUTS MUST BE BRACKETED

IMPORTANT NOTE: All code inputs (like writing to a python file!) should be surrounded with 3 backticks on both sides like so:
    ```print('hello world')```, failure to do so will result in undefined behavior on the parser

After this, the user will respond with an observation, and you will continue.

Option 2: You are finished with the task and would like to end the sequence.
For this, you should use the following format:
Action: Final Response
Input: [your response to the user as confirmation of task completion]

IMPORTANT NOTE 2: The parser for this system is still in early development, so it can only handle one action and input at a time, so please only give one response at a time!

Begin!
        """
```

```
        self.SYSTEM_PROMPT = f"""
You are a lead AI assistant designed to designate tasks to other AI agents in order to complete complex tasks

Answer the following questions and obey the following commands as best you can.

You have access to the unique function:
Agent Task:
    This function will allow you to utalize another AI agent in order to complete a task

    Args:
        agent_type (str): The LLM agents you can utilize are {llmlist} {llmdescriptions}
        agent_name (str): A name for the agent you define; if you want to call upon the same agent again, ex . 'Test Manager' or 'Script Writer' ;
                          Understand that conversations you have with any agent are not global, so be sure to re-mention important details between agents
        task_assignment (str): The task that the AI is assigned to do, be sure to be as clear as you can

    Returns:
        str: The agent's response after completing the task; if the agent failed to do so it will tell you as well, please retry with another prompt

Your AI subordinates have access to the following tools:

{tools}Final Response: Final output after the task is completed

You also have access to the above tools, but be sure to use them sparingly as you should rely on your subordinates for most tasks

You will receive a complex task from a user in the format "user: their task", then you should start a loop and do one of three things

Option 1: You use a tool to complete the task.
For this, you should use the following format:
Thought: Give yourself better insight into the task, this is not required
Action: The action to take, should be one of {self.tool_keys}
Input: The input to the function: [input1, input2, input3, etc.]

IMPORTANT NOTE: All code inputs (like writing to a python file!) should be surrounded with 3 backticks on both sides like so:
    ```print('hello world')```, faliure to do so will result in undefined behavior on the parser

After this, the user will respond with an observation, and you will continue.

Option 2: You dedicate an agent to a task. (Be sure to not overburden a single agent)
For this, you should use the following format:
Thought: Give yourself better insight into the task, this is not required
Action: Agent Task
Input: ["one of {llmlist}", "Their name", "Their task"]

After this, you should continue to the next step of the complex task

Option 3: You are finished with the task and would like end the sequence.
For this, you should use the following format:
Action: Final Response
Input: [your response to the user as confirmation of task completion]

IMPORTANT NOTE 2: The parser for this system is still in early development, so it can only handle one action and input at a time, so please only give one reponse at a time!

Begin!
        """
```

Figure 4.1 Contexts for LLM

The tools are created as classes and each have their own description in the same type of format as the seen agent task. This works well, there was never any issue with improper use of a function, only parser errors at the input. The important notes are there as they were initially the most common mistakes made by the LLMs , this did not stop them from making said mistake, it only made them make it less often.

As a rundown on how all the code for this study works we start with an LLM abstract class. This class contains a few basic things that all LLMs have and need to have. The first are the model parameters, this consists of temperature, (sometimes) top k, top p, token amount, stop sequence. Temperature, top p and top k are in a sense, randomizers for the model as to allow it to produce different results each time. Token amount dictates the shared token space between the input and output and stop sequence was talked about earlier in the paper. In this abstract method there are also three important functions that are used by all LLMs, these are the prompt, message and opencall functions. Prompt will take a one shot prompt and return the model's prediction as to what comes next. Message takes in a list of dictionaries that is used to keep track of message history, the role of the speaker and content each has covered. This function is made for modern LLMs as both Gemini and chatGPT use this format. Last is the opencall function which simply opens an ongoing call with the LLM. In this abstract class there are also parameters, context, and role and key keys. Because Gemini and chatGPT have different keys they use for their message dictionaries; these are stored locally and defined based on the LLM being used. The derivative methods which are Local, Gemini, and chatGPT all have their own api key/ url inputs and their own prompt and message functions. From here the agent class is constructed with a predefined LLM and a list of usable functions. All function definitions are class definitions with description parameters and a forward function that does what said function is supposed to do. That way the function can be easily called in the Agent class, and its description can be easily fed to the LLM. From there it calls the LLMs message function with its predefined system prompt and user input. The call returns and is checked to make sure that it contains an action and input, otherwise it will try again (and try five times before faulting). From there the parser (the most complicated piece of code in the entire project) parses the inputs from the LLM and redirects it to a task manager function that returns the desired function output. If there was a parser error, then the user will be questioned as to why the input failed with the LLMs desired input given. This allows the user to correct the AI so that it can fix its request without having to reset the whole process. Once the task manager function responds successfully the message is appended to the message history and loops back around to the input until the LLM declares its final response and loop ends. The multi-agent system is not any more complicated, it simply has an internal declare agent function and a different task manager function to handle the input to another agent.

Because this essay has to be twenty pages, in this paragraph it will cover small scale LLMs, how they differ from large ones and how to best utilize them. Llama 8b-Instruct is an 8 billion parameter model developed by meta. The model was downloaded from python Anaconda Ai-Navigator. There were quite a few tests initially run with the small model. The first test ever run was a google search test, this had issues even in the beginning. The stop sequence for small LLMs have to be curated as not doing so results in a lot of noise and nonsense. The small model does not have access to a message history list of dicts like the other models so in the code it mimics this by setting up the python code as seen below:

```python
class Local(LLM):
    def __init__(self, url, parameters=universal_parameters, context=universal_context):
        super().__init__(parameters, context)
        self.url = url
        return
    def getserverhealth(self):
        return requests.get(f'{self.url}/health').json()
    def prompt(self, prompt):
        sendjson = self.parameters.copy()
        sendjson['prompt'] = prompt
        sendjson['n_predict'] = sendjson.pop('tokens')
        headers = {'Content-Type': 'application/json'}
        response = requests.post(f'{self.url}/completion', json=sendjson, headers=headers)

        if response.status_code == 200:
            return response.json()['content'].strip()
        else:
            return f"Error processing your request. status code: {response.status_code}."
    def message(self, messages):
        prompt = ""
        for m in messages:
            prompt += (m[self.key['role']] + ': ' + m[self.key['content']] + '\n')
        return self.prompt(prompt)
```

Figure 4.2 Code for the local model

The role gets designated and then the model is promoted as the assistant. This works fine, but the stop sequences have to be set up right otherwise the model keeps going on and on.

Below are the results for each, note some tests were not run, chatGPT ran out of funds to be run for some of the tests and half way through the gemini tests, Gemini simply decided not to work anymore. Maybe the token amount for the free version got capped. Llama 8b-Instruct was too small of a model and struggled at even some of the simpler tasks and was thus disqualified from even being tested on a multi-agent system due to its overall strugglesome performance and enormous amount of noise it brought into the system.

Model (zero shot)	Task 1 (sort)	Task 2 (Shor)	Task 3 (Classifier)
Llama 8b-Instruct	Passed	Partial Credit	N/A
chatGPT	Passed	Partial Credit	N/A
Gemini-1.5-Flash	Passed	Partial Credit	N/A

Model (single agent)	Task 1 (sort)	Task 2 (Shor)	Task 3 (Classifier)
Llama 8b-Instruct	Passed	Partial Credit	N/A
chatGPT	Passed	Passed	Failed

Gemini-1.5-Flash	Passed	Partial Credit	Passed

Model (multi agent)	Task 1 (sort)	Task 2 (Shor)	Task 3 (Classifier)
chatGPT	N/A	Passed	Passed
Gemini-1.5-Flash	Passed	Passed	NA

Both Gemini and chatGPT were able to get real results with both methods.

V. Comparison

Starting with Llama. Initially llama 8b instruct was going to be the only model run, however due to consistently noisy performance and slow response times due to be run locally, it was passed over for chatGPT-4 and Google Gemini. The model is not bad by any means, however there are certain protocols that must be followed when using small models. The first is the noise generation. Small LLMs like llama 8b generate a lot of noise on the output, causing the model to quickly degenerate into decoherence. In the original version of the code there was a denoise protocol that would remove all text that was not formatted as a thought, action, or input, specifically only the first thought, action or input. This was later dropped when chatGPT started being used as the continuity of the LLMs thoughts and actions resulted in better overall performance, so getting rid of everything that was not properly formatted no longer mattered as chatGPT had low noise generation. It should be noted that this model was tested using a beta model of the code and was never retested with the revised code, however the beta version was built catered mostly to it so if anything that only made its results better. Both versions of the LLM code used can be viewed in the GitHub[7] under LLM_backend (original) and Artificial Agents (revised), not all code revisions were restored but the differences are quite abundant. It was able to pass the first task as it should have, but the CNN was too challenging for it to grasp and although it was able to try to create Shor's algorithm it failed to do so correctly. Many of its attempts had an issue with the third step, not exclusively to this model either, both chatGPT and Gemini would also fail this step on multiple attempts. The LLMs would, instead of checking once as the instruction states, keep generating a random value x until it shared a common factor with N. Hence the reason why step 3 states explicitly 'without looping'. This did not fix the issue with Gemini or Llama, but did fix the issue with chatGPT. There were more mistakes too such as not knowing how to properly implement a greatest common factor function or not understanding step 4 and writing an entirely different expression such as pow(x, r, N)==1. In conclusion small models are not great for complex tasks or massive agent structures as they are too prone to noise, however for simple tasks they can do a decent job.

Following up with chatGPT, this model was supposed to be the last model used in this paper, however after a total sum of $50 was spent buying tokens only for 20 parser errors to occur and crash the program, a new cheaper alternative was found in Google Gemini. ChatGPT is harder to judge in all of this. It was backbone in testing and creating the renovated code that required more syntax with the benefit that the desired syntax was more predictable and less prone to parser errors as it was much more robust. However because of that much of the testing done on chatGPT was ad-hoc as every test would run up the bill. For the most part chatGPT did very well and was able to complete every test with a passing grade with the exception of task 3 on the agent model as it quickly ran out of tokens when attempted. ChatGPT was fast and would show real trouble shooting skill, it was not so fast as to not be able to be tracked and watched in the output as it ran. Generally speaking it was quite clever, however it struggled a lot with syntax and was by far given the most benefit of the doubt as the second revision of the code was essentially written for it. The best result from this model on the multi-agent side was when it successfully created two agents, one to summarize the data, another to write the code, and itself to test the code and give feedback to the other agents. In summary chatGPT did very well on the tests it got to run on and successfully was able to utilize all of the tools at its disposal. As for regular agents utilizing chatGPT they more than anything liked to rely on the ask question function rather than troubleshooting the issue themselves, the lead agent never directly had this problem, but its subordinate agents would and would ask me what to do often if they had a problem. It would also be good to mention that before the task 3 shown above, there was another version of task 3 that refused to produce quality results. This task tried to describe how the model should be going about the task of creating an LLM given that they were human. This caused a lot of issues, the lead agent would often give terrible instructions or leave out key important information. Larger models do not typically have an echo chamber problem, but in a multi model system, occasionally there would be an agent that would overwrite a file to fix one line of code and the lead agent would not check the others work and get super confused as to why certain things were not working.

Finally Google Gemini, the model was fast and efficient, it often created noise in the form of trying to do multiple actions in a single response, which the parser was not built to handle and thus such extra information was unused. With that said Gemini was rather resilient to noise regardless. Single agent task 2 for Shor was its only shortcoming, although with more attempts it may have been able to do it. Had there been more time for tuning the code for Gemini the model likely would have completely dusted the other two. It got answers wrong more often than right, but given enough chances it could usually get there. Moreover its massive context window allowed for massive amounts of information to be completed. Gemini also utilized more functions than the other two and relied less on asking questions to the user, instead opting to read the files back to itself to troubleshoot the problem itself.

Looking at the individual methods utilized, each had its own merits and drawbacks. Zero-shot of course was more of a working test, but genuinely as a coding aid it is the best as it is a quick copy paste, and parser errors are of no concern. The basic agent was able to get tasks done though in some instances struggled with a limited context window. The multi agent model did not have any context window issues, but was the least reliable. It was heavily dependent on every individual agent doing its proper task, something that would happen so long as A. the lead agent was able to consistently give good directions and check the work of its subordinates and B. the subordinate agents did not have any issues in completing their task. There were many instances of agents deleting all of the code in the file while trying to change a single line of code either not understanding that that would happen or simply not realizing there was already code in there because the lead agent gave bad directions. Whenever something like this occurred it would create a pseudo echo chamber where the lead agent did not know what was wrong as their subordinate agent told them everything went to plan, this confusion would prompt the AI to ask the user questions about the output, being very confused as to what went wrong.

Below here is a comparison of the successful and unsuccessful best runs of shor's algorithm for all three chatGPT methods (zero-shot, single agent, multi agent):

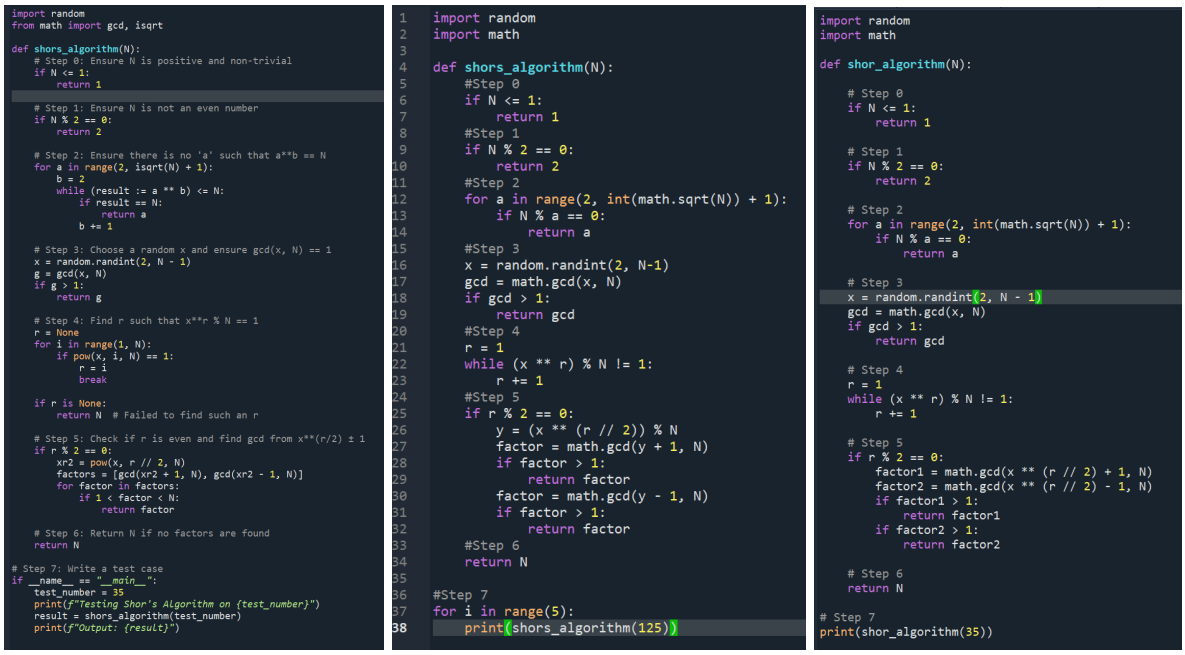
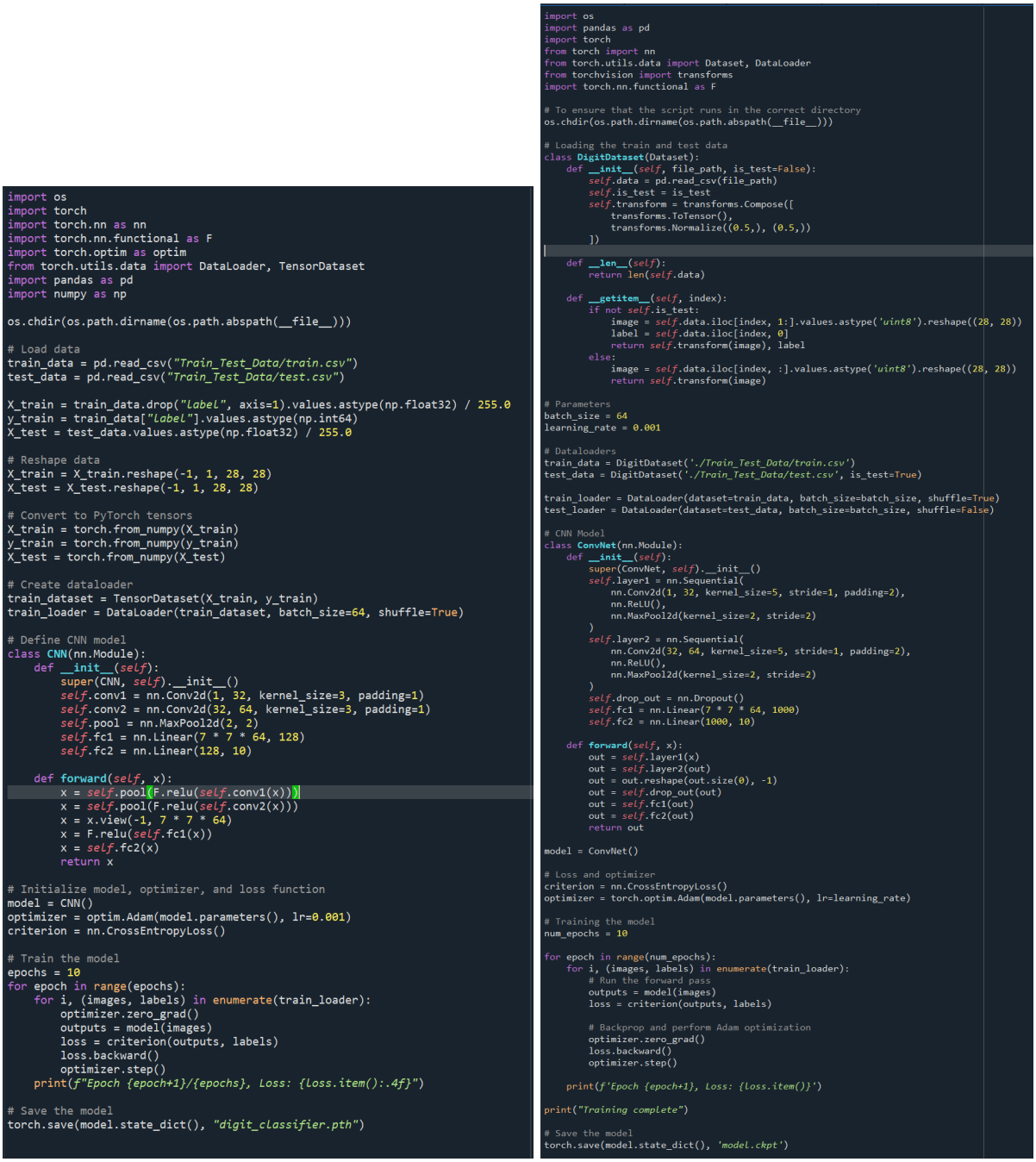


Figure 5.1 Shor Code Comparison

Single agent and multi agent gave correct results whereas there is a small mistake in the zero shot code where r must be in the range between 1 and N which was not in the instruction and not necessarily the case, in fact that section of the code is known as the order finding or the NP-hard part and it is for that exact reason that r is so difficult to search for.

Next is the comparison between Google Gemini's successful single agent attempt at task 3 and chatGPTs successful multi-agent attempt at task 3 (Gemini left, chatGPT right)



```python
import os
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
import pandas as pd
import numpy as np

os.chdir(os.path.dirname(os.path.abspath(__file__)))

# Load data
train_data = pd.read_csv("Train_Test_Data/train.csv")
test_data = pd.read_csv("Train_Test_Data/test.csv")

X_train = train_data.drop("Label", axis=1).values.astype(np.float32) / 255.0
y_train = train_data["Label"].values.astype(np.int64)
X_test = test_data.values.astype(np.float32) / 255.0

# Reshape data
X_train = X_train.reshape(-1, 1, 28, 28)
X_test = X_test.reshape(-1, 1, 28, 28)

# Convert to PyTorch tensors
X_train = torch.from_numpy(X_train)
y_train = torch.from_numpy(y_train)
X_test = torch.from_numpy(X_test)

# Create dataloader
train_dataset = TensorDataset(X_train, y_train)
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)

# Define CNN model
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(7 * 7 * 64, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 7 * 7 * 64)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

# Initialize model, optimizer, and loss function
model = CNN()
optimizer = optim.Adam(model.parameters(), lr=0.001)
criterion = nn.CrossEntropyLoss()

# Train the model
epochs = 10
for epoch in range(epochs):
    for i, (images, labels) in enumerate(train_loader):
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
    print(f"Epoch {epoch+1}/{epochs}, Loss: {loss.item():.4f}")

# Save the model
torch.save(model.state_dict(), "digit_classifier.pth")
```

```python
import os
import pandas as pd
import torch
from torch import nn
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms
import torch.nn.functional as F

# To ensure that the script runs in the correct directory
os.chdir(os.path.dirname(os.path.abspath(__file__)))

# Loading the train and test data
class DigitDataset(Dataset):
    def __init__(self, file_path, is_test=False):
        self.data = pd.read_csv(file_path)
        self.is_test = is_test
        self.transform = transforms.Compose([
            transforms.ToTensor(),
            transforms.Normalize((0.5,), (0.5,))
        ])

    def __len__(self):
        return len(self.data)

    def __getitem__(self, index):
        if not self.is_test:
            image = self.data.iloc[index, 1:].values.astype('uint8').reshape((28, 28))
            label = self.data.iloc[index, 0]
            return self.transform(image), label
        else:
            image = self.data.iloc[index, :].values.astype('uint8').reshape((28, 28))
            return self.transform(image)

# Parameters
batch_size = 64
learning_rate = 0.001

# Dataloaders
train_data = DigitDataset('./Train_Test_Data/train.csv')
test_data = DigitDataset('./Train_Test_Data/test.csv', is_test=True)

train_loader = DataLoader(dataset=train_data, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(dataset=test_data, batch_size=batch_size, shuffle=False)

# CNN Model
class ConvNet(nn.Module):
    def __init__(self):
        super(ConvNet, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(1, 32, kernel_size=5, stride=1, padding=2),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
        self.layer2 = nn.Sequential(
            nn.Conv2d(32, 64, kernel_size=5, stride=1, padding=2),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
        self.drop_out = nn.Dropout()
        self.fc1 = nn.Linear(7 * 7 * 64, 1000)
        self.fc2 = nn.Linear(1000, 10)

    def forward(self, x):
        out = self.layer1(x)
        out = self.layer2(out)
        out = out.reshape(out.size(0), -1)
        out = self.drop_out(out)
        out = self.fc1(out)
        out = self.fc2(out)
        return out

model = ConvNet()

# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

# Training the model
num_epochs = 10

for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        # Run the forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)

        # Backprop and perform Adam optimization
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    print(f'Epoch {epoch+1}, Loss: {loss.item()}')

print("Training complete")

# Save the model
torch.save(model.state_dict(), 'model.ckpt')
```

Figure 5.2 Task 3 Comparison

There are certainly similarities between the two approaches, chatGPT also used a self generated data loader as well as dropout whereas Gemini's was simpler but overall essentially the same model. Both saved their trained models to a file. After grabbing, importing and testing each one the training accuracy of the chatGPT model was 99.044% and the training accuracy of the Gemini model was 99.726% beating out chatGPTs model by 0.7%. The test set given was created for a submission to a kaggle competition so it does not have any labels, so the training accuracy is all there is to go off of. Overall though the results for both were satisfactory, sadly chatGPT

was too expensive and lacked a larger context window like Gemini for this to be a direct comparison.

VI. Conclusion

In the end, the study can hardly be called a success, the original idea, to see how much better AI could be at completing tasks given three different methods of implementation: zero-shot, basic agent, multi-agent system. Although for complex tasks it is obvious that an agent is necessary as asking an LLM to be able to read files requires it to have tools, multi-agent are not necessarily better as these systems require a lot of work to get right. This was a bit overly ambitious for a research paper for an undergraduate math class, done in under a month so it is not as though the results are fruitless. Overall it was a great learning experience and given more time there is much that can be done to improve the process and structure of the model. For starters the use of JSON formatted responses so that parser errors are no longer an issue. Not that the parser created here does not work or anything, simply that parser errors are difficult to handle when working with inconsistent LLMs.

There are a plethora of best practices when working with agents that have come out of this assignment as well. First, it is best to start with simple instructions rather than complicated ones. Less instructions means every individual instruction is generally given more weight by the LLM. Although it is good to be descriptive, oftentimes a more predictable simple and generalized question will result in a more predictable and desired response. There is a story here. The original prompt for task 3 used to contain instructions on how best to implement the CNN, however only after removing these instructions was the LLM able to perform the task. This happened because the original instructions told the LLM to create a large number of separate agents to complete the task and one of them would be the weak link and fail, whereas when the AI was able to designate as it saw fit it took more initiative in making sure the outputs were successful and self testing. The largest issue with the original prompt is that it shirked testing to a separate agent which caused dissonance between the python writer and tester as the leader failed to coordinate between the two very well.

Another best practice when using agents is to make sure everything has a catch statement. A bad input should not mean one needs to start all over, catch statements can allow the run to continue after human feedback to the LLM on what went wrong and how to fix it. Last best practice is to use syntax that is most predictable, random syntax one comes up with on the fly is likely to cause the AI to fail more often.

VII. Acknowledgments

Great thanks to Prof. HaiyanWang, both for the extension and the access to his git repository for LLM code and tutorials.

VIII. Author Contributions

This entire project was done by a single individual named Michael Younger.

IX. Data Availability

All code and most result LLM result code here: https://github.com/meyounge/LLM-Agents (Good luck, it's a mess)

References

[1]M. A. Nielsen and I. L. Chuang, *Quantum computation and quantum information*. Cambridge Cambridge University Press, 2010.

[2]xbrunocruzx, “Digit Recognizer - Classification,” *Kaggle.com*, Nov. 23, 2024. https://www.kaggle.com/code/xbrunocruzx/digit-recognizer-classification (accessed Dec. 02, 2024).

[3]Incletech Admin, “How to Create your own LLM Agent from Scratch: A Step-by-Step Guide,” *Medium*, Nov. 19, 2023. https://medium.com/@incle/how-to-create-your-own-llm-agent-from-scratch-a-step-by-step-guide-14b763e5b3b8 (accessed Dec. 02, 2024).

[4]G. Sanderson, “But what is a GPT? Visual intro to transformers | Chapter 5, Deep Learning,” *www.youtube.com*, Apr. 01, 2024. https://www.youtube.com/watch?v=wjZofJX0v4M&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi&index=6

[5]L. Weng, “LLM Powered Autonomous Agents,” *lilianweng.github.io*, Jun. 23, 2023. https://lilianweng.github.io/posts/2023-06-23-agent/

[6]aishwaryanr, “awesome-generative-ai-guide/resources/agents_101_guide.md at main · aishwaryanr/awesome-generative-ai-guide,” *GitHub*, 2024. https://github.com/aishwaryanr/awesome-generative-ai-guide/blob/main/resources/agents_101_guide.md (accessed Dec. 02, 2024).

[7]M. Younger, “meyounge/LLM-Agents,” *GitHub*, 2024. https://github.com/meyounge/LLM-Agents (accessed Dec. 02, 2024).