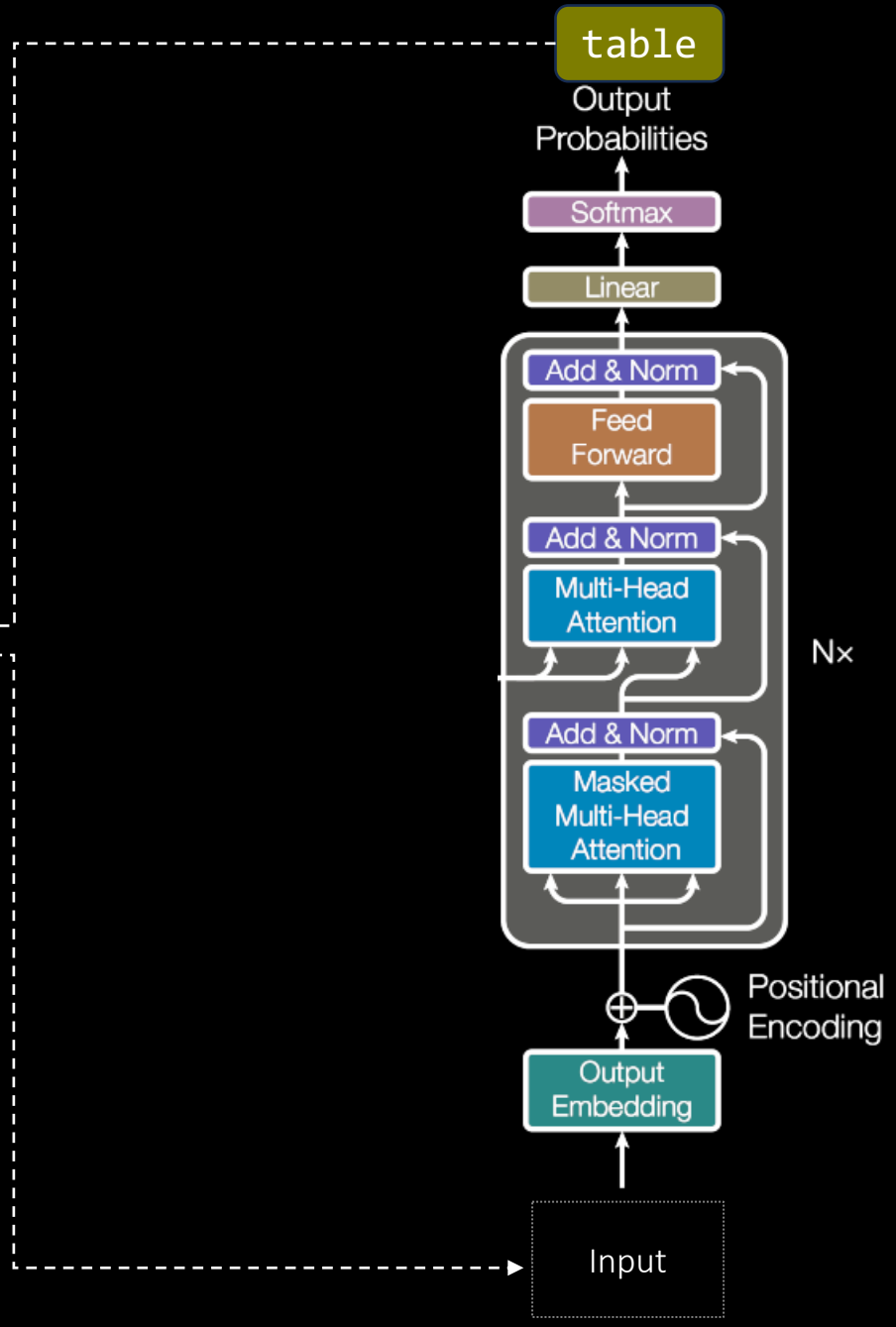
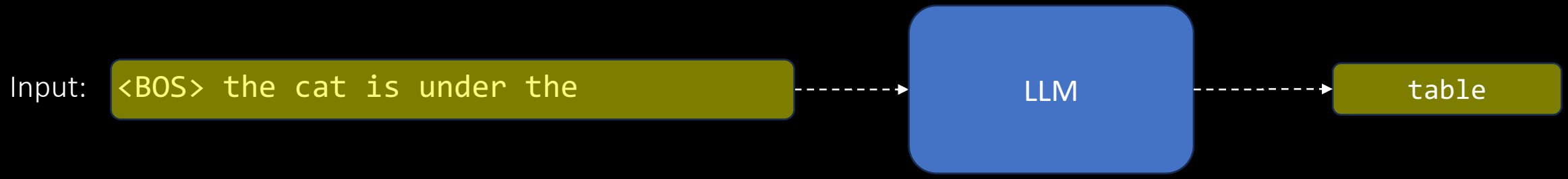
LLMs from Dummies: Part 4

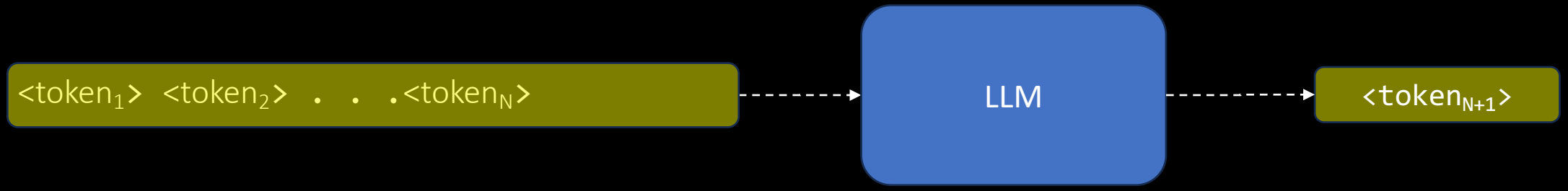
Topics

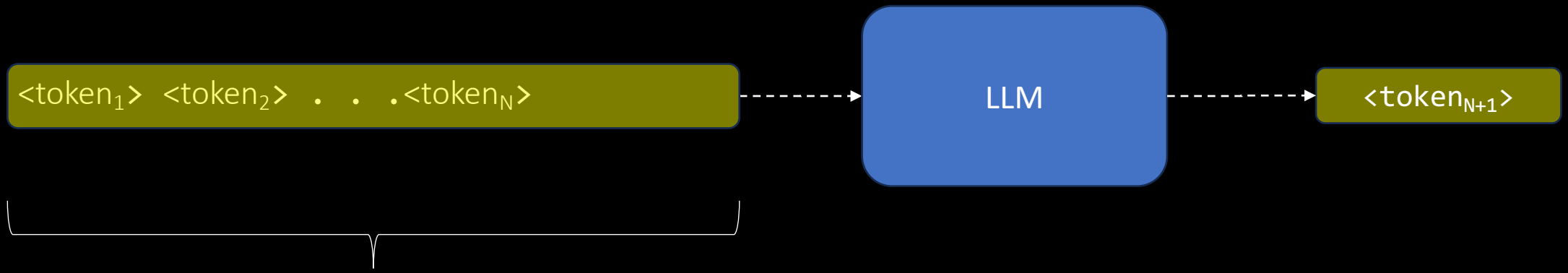
1. Language translation
    1. Simplest program: Literal translation
    2. Dealing with missing tokens
2. Literal translation using "ML”
    1. Encoding: One hot
    2. Using matrices
    3. A 'dictionary' using matrices
    4. Decoding: Cosine similarity
3. Attention
    1. Similar tokens: Softmax
    2. Matrix Query: Q
    3. Scaled dot-product attention
4. Attention Head
    1. Weight matrices
    2. Connecting matrices
5. Revisiting tokens & encoding
    1. Tokenizing: BPE
    2. Encoding: Embeddings
6. Transformer Block
    1. Multi-headed attention
    2. Non-linear (feed forward) layer
    3. Stack blocks
    4. Masked attention
7. Transformer
    1. Stacking deep networks
    2. Normalization layers
    3. Skip connections
    4. Dropout
8. Pre-training, Training, Fine-tunning, Adapting, Instruct
    8. Pre-training numbers (GPU hours, params, etc.)
    9. LORA / PERF: Basic concepts
    10. "Instruct" models
9. Prompts all the way down
    8. "Chat": Just isolated requests with "memory" of conversations
    9. "Context": Just add a sentence to the prompt
    10. "Prompt engineering": Similar to adding the right words in a Google search
10. Frameworks
    8. LangChain (API, Models, LLM, Prompts, Agents). Simple examples
    9. Vector databases
    10. Huggingface
11. Scaling inference & training
    8. Single GPU: Quantization, fp16, etc.
    9. Multiple GPUs: PP, ZeRO, TP, Sharding, etc.

Input: <BOS> the cat is under the



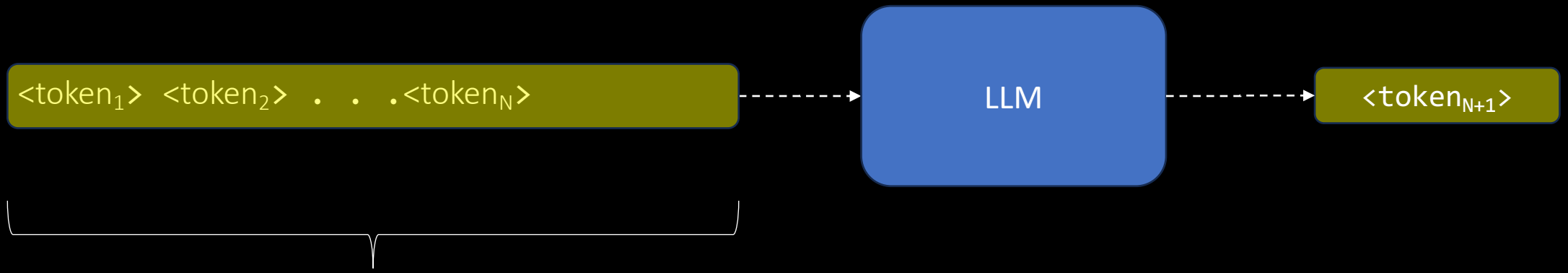

Input:
<BOS> the cat is under the
LLM
table


<token$_1$> <token$_2$> . . .<token$_N$>
LLM
<token$_{N+1}$>



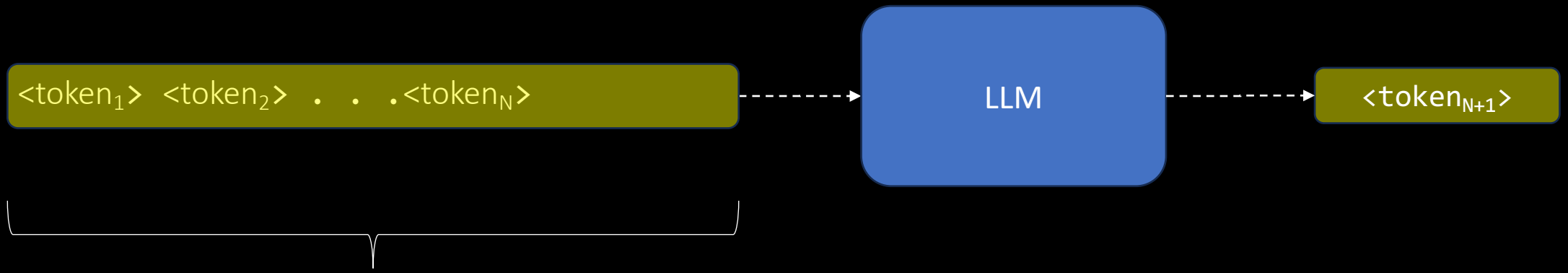
This is the prompt / context

This is the LLM's context to predict the output (next token)



The prompt is the **only** input to the LLM

The only way to interact with an LLM is by changing the prompt to do what we need to do



“the average one spaced page usually contains about 3000 characters or 500 words”
“For English text, 1 token is approximately 4 characters or 0.75 words”

- A 4KT context is ~**5.5 pages**
- An 8KT prompt is **~11 pages**
- A 32KT prompt is **~45 pages**

You can add a lot of data in 45 pages of text!

Prompt Engineering

Prompt:

```
The sky is
```

Ref: https://www.promptingguide.ai/

Prompt:

```
The sky is
```

Output:

```
The sky is blue on a clear day. On a cloudy day, the sky may be gray or white
```

Ref: https://www.promptingguide.ai/

Prompt Role

In a “chat” model, the context can contain several interactions between the user and the system

tell me a joke

why did the chicken cross the road?

I don't know, why did the chicken cross the road

Prompt Role

In a “chat” model, the context can contain several interactions between the user and the system

user: tell me a joke

assistant: why did the chicken cross the road?

user: I don't know, why did the chicken cross the road

Prompt Role

- `system`: System level instruction to guide your model's behavior
- `user`: The user’s input
- `assistant`: Represents the model’s output (e.g. from a previous chat answer)
- `function`: You can ask the model to generate a “function call” (i.e. use tools).

Prompt Role

Prompt

SYSTEM	You will be provided with statements, and your task is to convert them to standard English.
USER	She no went to the market.

Sample response

She did not go to the market.

Prompt Role

Prompt

SYSTEM	Summarize content you are provided with for a second-grade student.
USER	Jupiter is the fifth planet from the Sun and the largest in the Solar System. It is a gas giant with a mass one-thousandth that of the Sun, but two-and-a-half times that of all the other planets in the Solar System combined. Jupiter is one of the brightest objects visible to the naked eye in the night sky, and has been known to ancient civilizations since before recorded history. It is named after the Roman god Jupiter.[19] When viewed from Earth, Jupiter can be bright enough for its reflected light to cast visible shadows,[20] and is on average the third-brightest natural object in the night sky after the Moon and Venus.

Sample response

Jupiter is a really big planet in our Solar System. It is the fifth planet from the Sun and it is the largest planet. It is called a gas giant because it is made mostly of gas. Jupiter is much smaller than the Sun, but it is bigger than all the other planets combined. It is very bright and can be seen in the night sky without a telescope. People have known about Jupiter for a very long time, even before they started writing things down. It is named after a god from ancient Rome. Sometimes, Jupiter is so bright that it can make shadows on Earth. It is usually the third-brightest thing we can see in the night sky, after the Moon and Venus.

Prompt Engineering Techniques

- Zero-Shot
- Few-Shot
- Chain-of-Thought
- Self-Consistency
- Generated Knowledge
- Tree of Thoughts (ToT)
- Retrieval Augmented Generation (RAG)
- Automatic Reasoning and Tool-use (ART)
- Active-Prompt
- Directional Stimulus
- ReAct
- . . .

Zero-Shot Prompting

Large LLMs today, such as GPT-3, are tuned to follow instructions and are trained on large amounts of data; so they are capable of performing some tasks "zero-shot."

We tried a few zero-shot examples in the previous section. Here is one of the examples we used:

Prompt:

```
Classify the text into neutral, negative or positive.
Text: I think the vacation is okay.
Sentiment:
```

Output:

```
Neutral
```

Note that in the prompt above we didn't provide the model with any examples of text alongside their classifications, the LLM already understands "sentiment" -- that's the zero-shot capabilities at work.

Few-Shot Prompting

While large-language models demonstrate remarkable zero-shot capabilities, they still fall short on more complex tasks when using the zero-shot setting. Few-shot prompting can be used as a technique to enable in-context learning where we provide demonstrations in the prompt to steer the model to better performance. The demonstrations serve as conditioning for subsequent examples where we would like the model to generate a response.

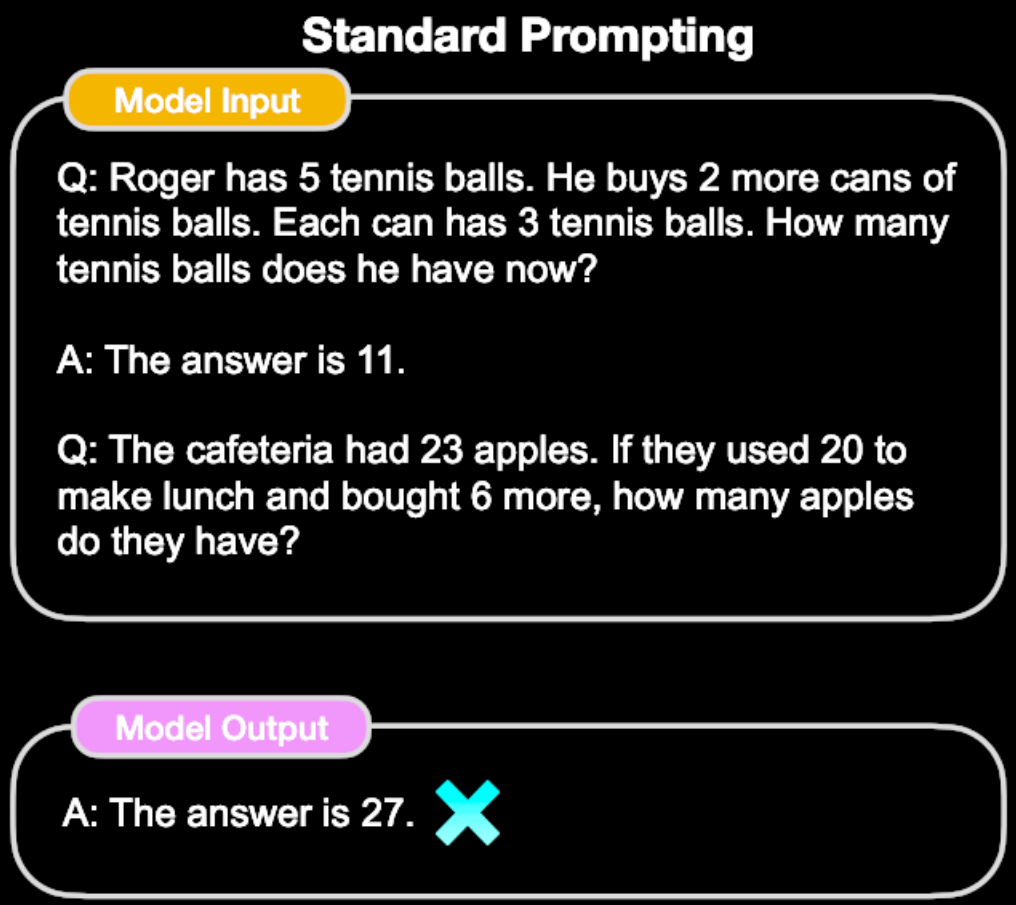
Prompt:

```
A "whatpu" is a small, furry animal native to Tanzania. An example of a sentence that uses
the word whatpu is:
We were traveling in Africa and we saw these very cute whatpus.
To do a "farduddle" means to jump up and down really fast. An example of a sentence that uses
the word farduddle is:
```

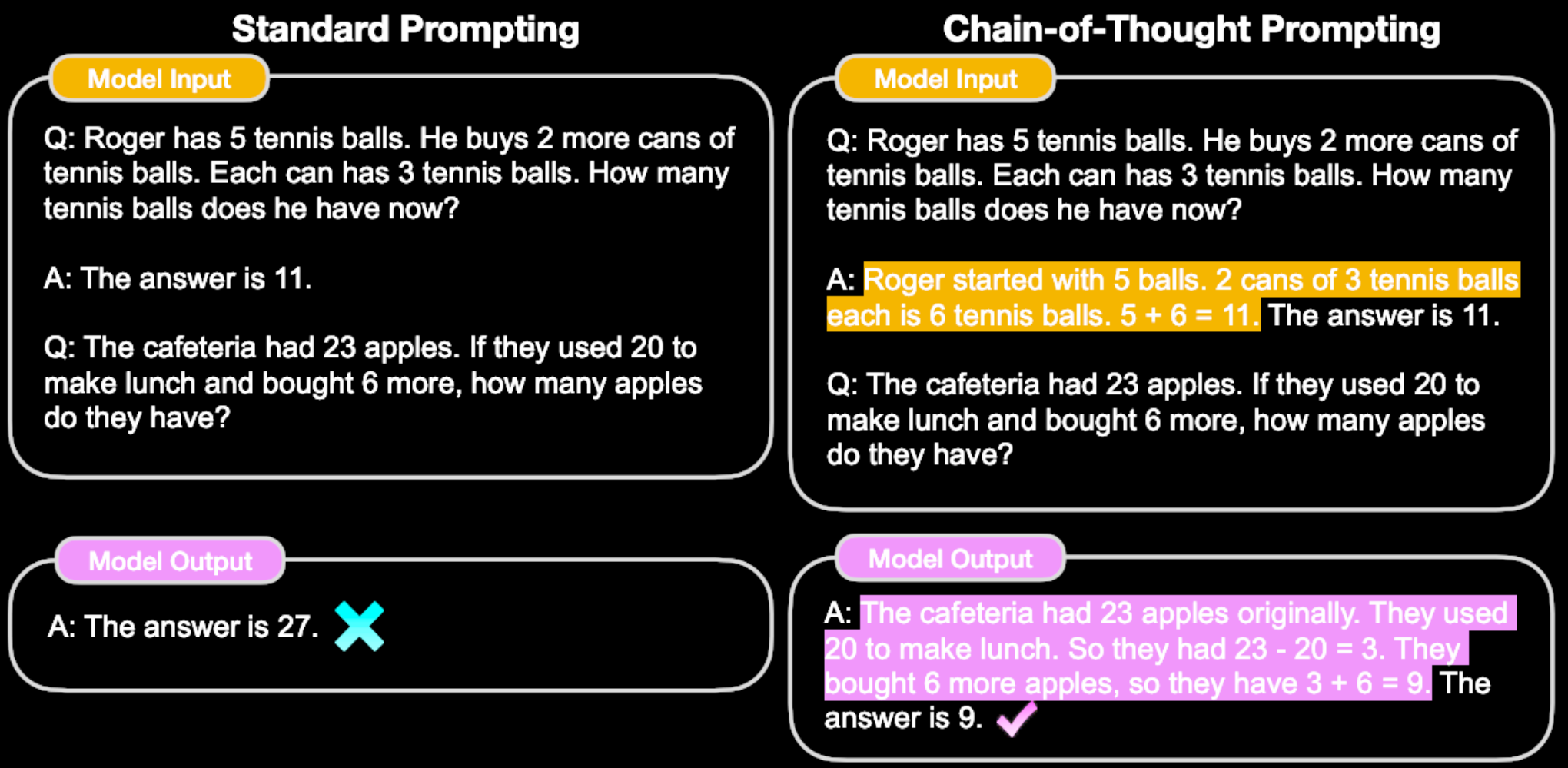
Output:

```
When we won the game, we all started to farduddle in celebration.
```

Chain of thought



Chain of thought

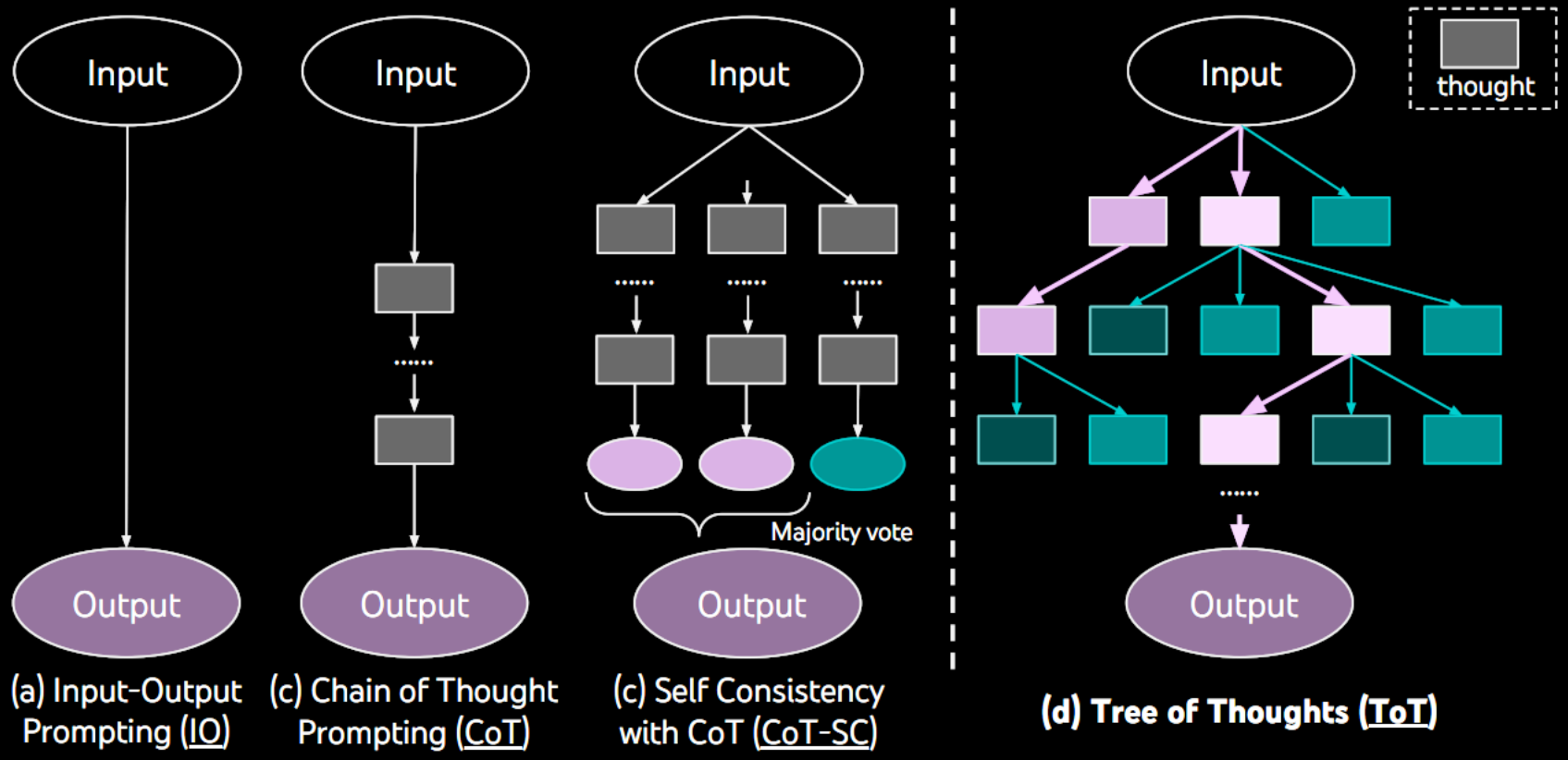


Tree of thought

For complex tasks that require exploration or strategic lookahead, traditional or simple prompting techniques fall short. Yao et el. (2023)(opens in a new tab) and Long (2023)(opens in a new tab) recently proposed Tree of Thoughts (ToT), a framework that generalizes over chain-of-thought prompting and encourages exploration over thoughts that serve as intermediate steps for general problem solving with language models.

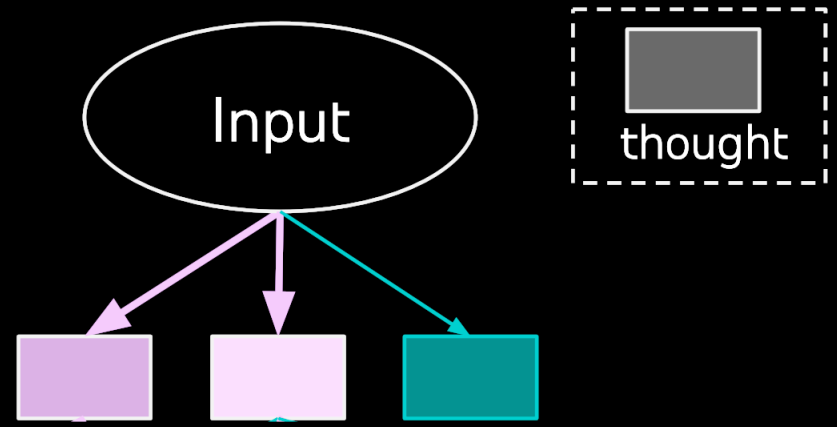
Tree of thought

For complex tasks that require exploration or strategic lookahead, traditional or simple prompting techniques fall short. Yao et al. (2023)(opens in a new tab) and Long (2023)(opens in a new tab) recently proposed Tree of Thoughts (ToT), a framework that generalizes over chain-of-thought prompting and encourages exploration over thoughts that serve as intermediate steps for general problem solving with language models.



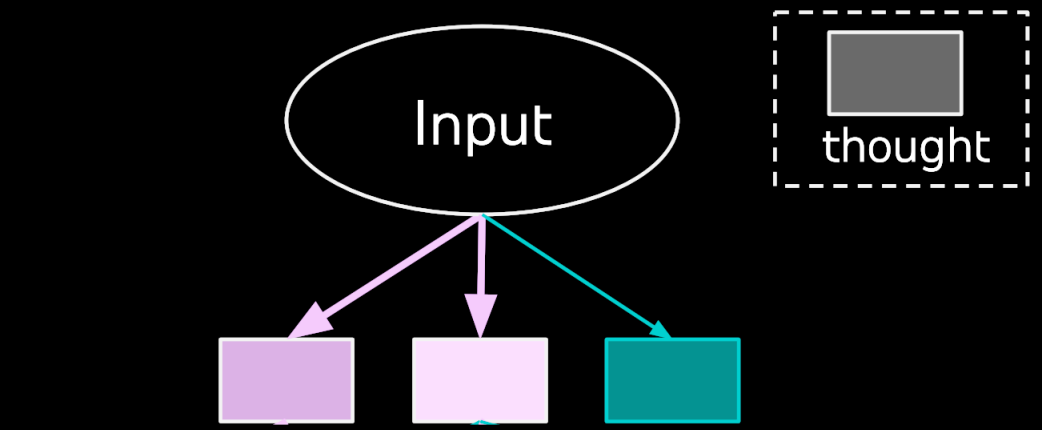
Tree of thought

- Ask the LLM to create create ONLY the first “thought”



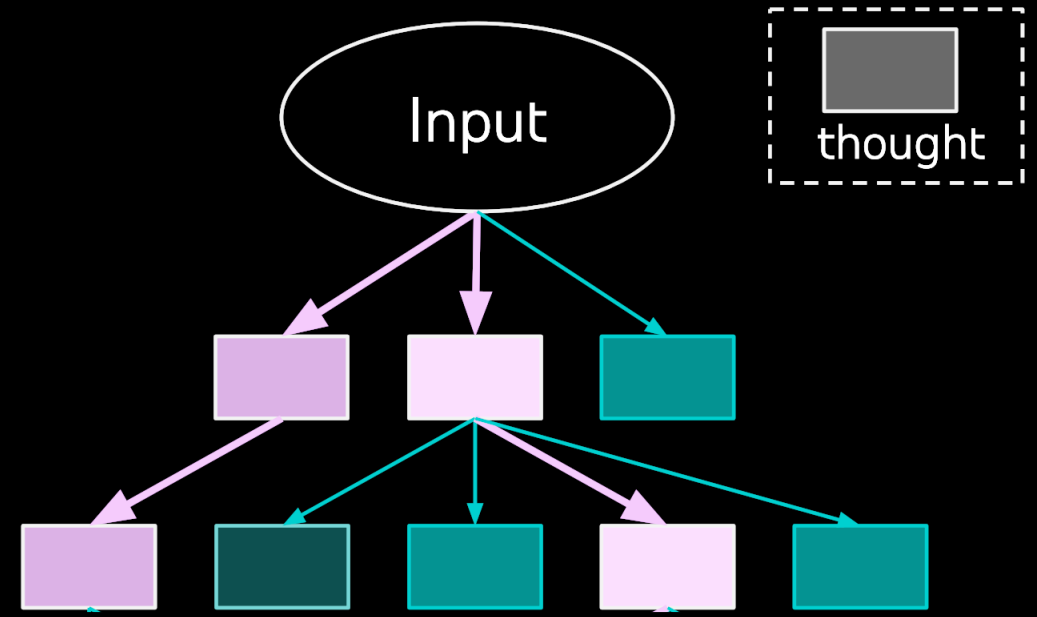
Tree of thought

- Ask the LLM to create create ONLY the first “thought”
- Self critique: Use LLM to evaluate these first thoughts



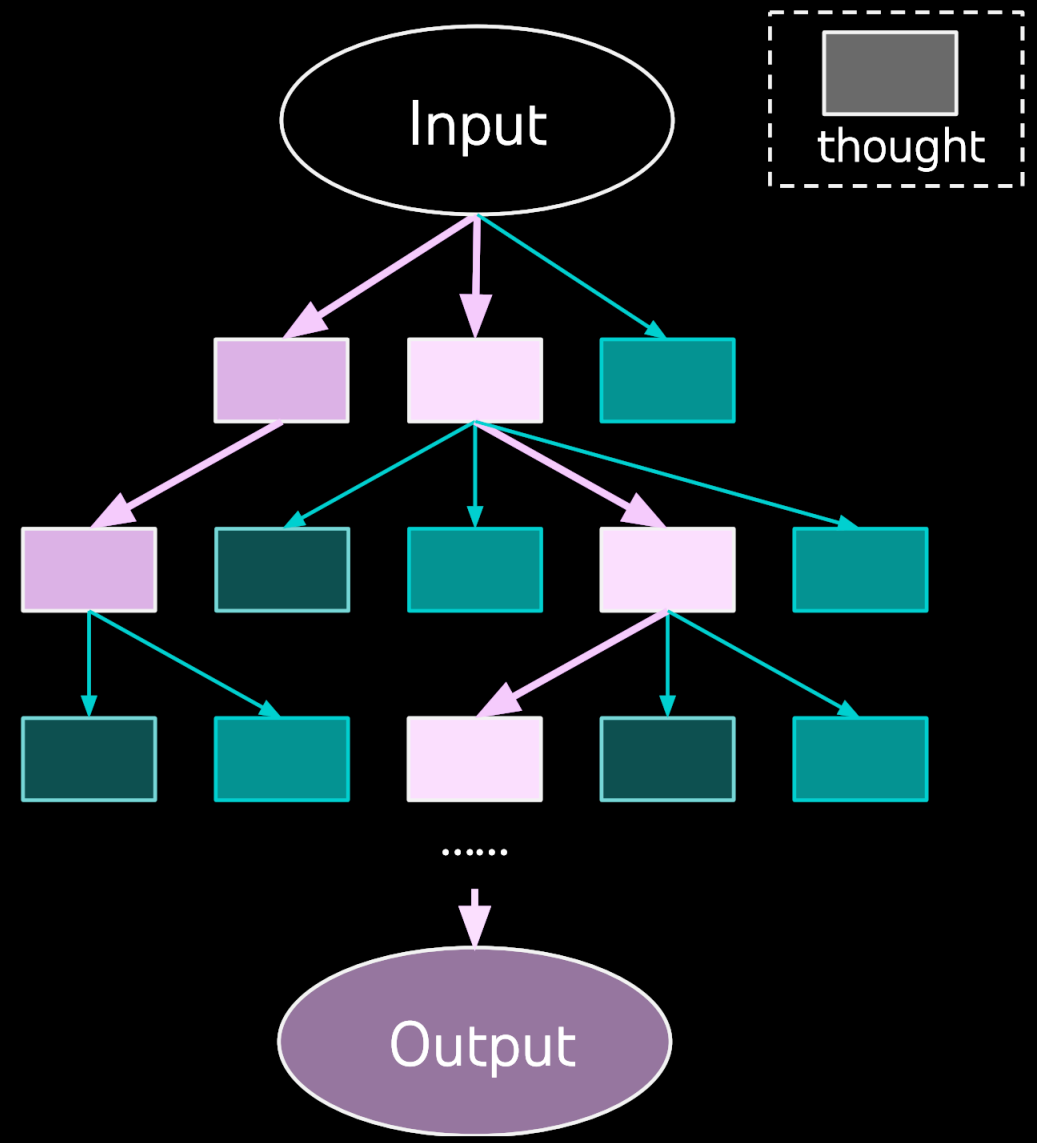
Tree of thought

- Ask the LLM to create create ONLY the first “thought”
- Self critique: Use LLM to evaluate these first thoughts
- Select the best thoughts
- Feed the LLM each thought and ask for the next “thought”



Tree of thought

- Ask the LLM to create create ONLY the first “thought”
- Self critique: Use LLM to evaluate these first thoughts
- Select the best thoughts
- Feed the LLM each thought and ask for the next “thought”
- Iterate until you reach an output



API

Create chat completion

`POST https://api.openai.com/v1/chat/completions`

Creates a model response for the given chat conversation.

Request body

`model` string **Required**

ID of the model to use. See the **model endpoint compatibility** table for details on which models work with the Chat API.

`messages` array **Required**

A list of messages comprising the conversation so far. **Example Python code**.

`role` string **Required**

The role of the messages author. One of `system`, `user`, `assistant`, or `function`.

`content` string **Required**

The contents of the message. `content` is required for all messages, and may be null for assistant messages with function calls.

`name` string Optional

The name of the author of this message. `name` is required if role is `function`, and it should be the name of the function whose response is in the `content`. May contain a-z, A-Z, 0-9, and underscores, with a maximum length of 64 characters.

`function_call` object Optional

The name and arguments of a function that should be called, as generated by the model

`temperature` number Optional Defaults to 1

What sampling temperature to use, between 0 and 2. Higher values like 0.8 will make the output more random, while lower values like 0.2 will make it more focused and deterministic.

We generally recommend altering this or `top_p` but not both.

`top_p` number Optional Defaults to 1

An alternative to sampling with temperature, called nucleus sampling, where the model considers the results of the tokens with top_p probability mass. So 0.1 means only the tokens comprising the top 10% probability mass are considered.

We generally recommend altering this or `temperature` but not both.

`n` integer Optional Defaults to 1

How many chat completion choices to generate for each input message.

`stream` boolean Optional Defaults to false

If set, partial message deltas will be sent, like in ChatGPT. Tokens will be sent as data-only **server-sent events** as they become available, with the stream terminated by a `data: [DONE]` message. **Example Python code**.

`stop` string or array Optional Defaults to null

Up to 4 sequences where the API will stop generating further tokens.

```
curl https://api.openai.com/v1/chat/completions \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer $OPENAI_API_KEY" \
  -d '{
     "model": "gpt-3.5-turbo",
     "messages": [{"role": "user", "content": "Say this is a test!"}],
     "temperature": 0.7
   }'
```

Function Calling

Function calling

Developers can now describe functions to `gpt-4-0613` and `gpt-3.5-turbo-0613`, and have the model intelligently choose to output a JSON object containing arguments to call those functions. This is a new way to more reliably connect GPT's capabilities with external tools and APIs.

These models have been fine-tuned to both detect when a function needs to be called (depending on the user's input) and to respond with JSON that adheres to the function signature. Function calling allows developers to more reliably get structured data back from the model. For example, developers can:

- Create chatbots that answer questions by calling external tools (e.g., like ChatGPT Plugins)

Convert queries such as "Email Anya to see if she wants to get coffee next Friday" to a function call like `send_email(to: string, body: string)`, or "What's the weather like in Boston?" to `get_current_weather(location: string, unit: 'celsius' | 'fahrenheit')`.

- Convert natural language into API calls or database queries

Convert "Who are my top ten customers this month?" to an internal API call such as `get_customers_by_revenue(start_date: string, end_date: string, limit: int)`, or "How many orders did Acme, Inc. place last month?" to a SQL query using `sql_query(query: string)`.

- Extract structured data from text

Define a function called `extract_people_data(people: [{name: string, birthday: string, location: string}])`, to extract all people mentioned in a Wikipedia article.

```python
import openai
import json

# Example from: https://platform.openai.com/docs/guides/gpt/function-calling

# Example dummy function hard coded to return the same weather
# In production, this could be your backend API or an external API
def get_current_weather(location, unit="fahrenheit"):
    """Get the current weather in a given location"""
    weather_info = {
        "location": location,
        "temperature": "72",
        "unit": unit,
        "forecast": ["sunny", "windy"],
    }
    return json.dumps(weather_info)
```

```python
# Step 1: send the conversation and available functions to GPT
messages = [{"role": "user", "content": "What's the weather like in Boston?"}]
functions = [
    {
        "name": "get_current_weather",
        "description": "Get the current weather in a given location",
        "parameters": {
            "type": "object",
            "properties": {
                "location": {
                    "type": "string",
                    "description": "The city and state, e.g. San Francisco, CA",
                },
                "unit": {"type": "string", "enum": ["celsius", "fahrenheit"]},
            },
            "required": ["location"],
        },
    }
]
response = openai.ChatCompletion.create(
    model="gpt-3.5-turbo-0613",
    messages=messages,
    functions=functions,
    function_call="auto",  # auto is default, but we'll be explicit
)
response_message = response["choices"][0]["message"]
```

```python
# Step 2: check if GPT wanted to call a function
if response_message.get("function_call"):
    # Step 3: call the function
    # Note: the JSON response may not always be valid; be sure to handle errors
    available_functions = {
        "get_current_weather": get_current_weather,
    }  # only one function in this example, but you can have multiple
    function_name = response_message["function_call"]["name"]
    fuction_to_call = available_functions[function_name]
    function_args = json.loads(response_message["function_call"]["arguments"])
    function_response = fuction_to_call(
        location=function_args.get("location"),
        unit=function_args.get("unit"),
    )

    # Step 4: send the info on the function call and function response to GPT
    messages.append(response_message)  # extend conversation with assistant's reply
    messages.append(
        {
            "role": "function",
            "name": function_name,
            "content": function_response,
        }
    )  # extend conversation with function response
    second_response = openai.ChatCompletion.create(
        model="gpt-3.5-turbo-0613",
        messages=messages,
    )  # get a new response from GPT where it can see the function response
    return second_response
```

Topics

1. Language translation
    1. Simplest program: Literal translation
    2. Dealing with missing tokens
2. Literal translation using "ML"
    1. Encoding: One hot
    2. Using matrices
    3. A 'dictionary' using matrices
    4. Decoding: Cosine similarity
3. Attention
    1. Similar tokens: Softmax
    2. Matrix Query: Q
    3. Scaled dot-product attention
4. Attention Head
    1. Weight matrices
    2. Connecting matrices
5. Revisiting tokens & encoding
    1. Tokenizing: BPE
    2. Encoding: Embeddings
6. Transformer Block
    1. Multi-headed attention
    2. Non-linear (feed forward) layer
    3. Stack blocks
    4. Masked attention
7. Transformer
    1. Stacking deep networks
    2. Normalization layers
    3. Skip connections
    4. Dropout
8. Pre-training, Training, Fine-tunning, Adapting, Instruct
    8. Pre-training numbers (GPU hours, params, etc.)
    9. LORA / PERF: Basic concepts
    10. "Instruct" models
9. Prompts all the way down
    8. "Chat": Just isolated requests with "memory" of conversations
    9. "Context": Just add a sentence to the prompt
    10. "Prompt engineering": Similar to adding the right words in a Google search
10. Frameworks
    8. LangChain (API, Models, LLM, Prompts, Agents). Simple examples
    9. Vector databases
    10. ~~Huggingface~~
11. Scaling inference & training
    8. Single GPU: Quantization, fp16, etc.
    9. Multiple GPUs: PP, ZeRO, TP, Sharding, etc.

LangChain

LangChain is a framework for developing applications powered by language models

Building an application

Now we can start building our language model application. LangChain provides many modules that can be used to build language model applications. Modules can be used as stand-alones in simple applications and they can be combined for more complex use cases.

The core building block of LangChain applications is the LLMChain. This combines three things:

- LLM: The language model is the core reasoning engine here. In order to work with LangChain, you need to understand the different types of language models and how to work with them.
- Prompt Templates: This provides instructions to the language model. This controls what the language model outputs, so understanding how to construct prompts and different prompting strategies is crucial.
- Output Parsers: These translate the raw response from the LLM to a more workable format, making it easy to use the output downstream.

LangChain trivial example

```
from langchain.chat_models import ChatOpenAI

# The ChatOpenAI objects are basically just configuration objects.
# You can initialize them with parameters like temperature and others.
chat_model = ChatOpenAI()

text = "What would be a good company name for a company that makes colorful socks?"

chat_model.predict(text)
# >> Socks O'Color
```

LangChain framework

- LLM
- Memory
- Prompt Template
- Output Parser
- Chain
- Tools
- Agent

LLMs

There are two types of language models, which in LangChain are called:

- LLMs: this is a language model which takes a string as input and returns a string
- ChatModels: this is a language model which takes a list of messages as input and returns a message

The input/output for LLMs is simple and easy to understand - a string. But what about ChatModels? The input there is a list of `ChatMessage`s, and the output is a single `ChatMessage`. A `ChatMessage` has two required components:

- `content`: This is the content of the message.
- `role`: This is the role of the entity from which the `ChatMessage` is coming from.

LangChain provides several objects to easily distinguish between different roles:

- `HumanMessage`: A `ChatMessage` coming from a human/user.
- `AIMessage`: A `ChatMessage` coming from an AI/assistant.
- `SystemMessage`: A `ChatMessage` coming from the system.
- `FunctionMessage`: A `ChatMessage` coming from a function call.

Prompt templates

Most LLM applications do not pass user input directly into an LLM. Usually they will add the user input to a larger piece of text, called a prompt template, that provides additional context on the specific task at hand.

In the previous example, the text we passed to the model contained instructions to generate a company name. For our application, it'd be great if the user only had to provide the description of a company/product, without having to worry about giving the model instructions.

PromptTemplates help with exactly this! They bundle up all the logic for going from user input into a fully formatted prompt. This can start off very simple - for example, a prompt to produce the above string would just be:

```
from langchain.prompts import PromptTemplate

prompt = PromptTemplate.from_template("What is a good name for a company that makes {product}?")
prompt.format(product="colorful socks")
```

```
What is a good name for a company that makes colorful socks?
```

Output Parsers

OutputParsers convert the raw output of an LLM into a format that can be used downstream. There are few main type of OutputParsers, including:

- Convert text from LLM -> structured information (eg JSON)
- Convert a ChatMessage into just a string
- Convert the extra information returned from a call besides the message (like OpenAI function invocation) into a string.

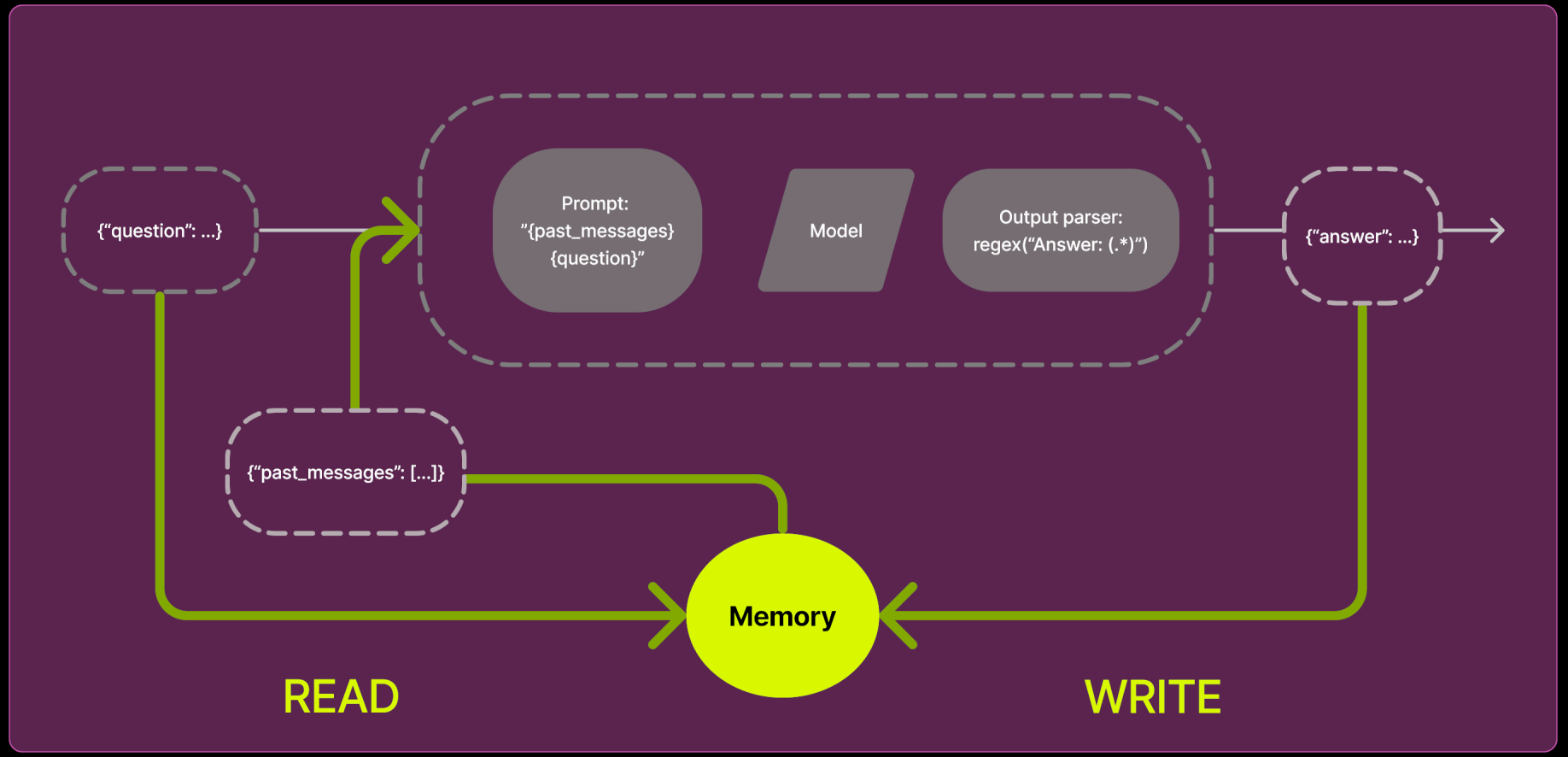
For full information on this, see the section on output parsers

In this getting started guide, we will write our own output parser - one that converts a comma separated list into a list.

```
from langchain.schema import BaseOutputParser

class CommaSeparatedListOutputParser(BaseOutputParser):
    """Parse the output of an LLM call to a comma-separated list."""


    def parse(self, text: str):
        """Parse the output of an LLM call."""
        return text.strip().split(", ")

CommaSeparatedListOutputParser().parse("hi, bye")
# >> ['hi', 'bye']
```

Memory

Most LLM applications have a conversational interface. An essential component of a conversation is being able to refer to information introduced earlier in the conversation. At bare minimum, a conversational system should be able to access some window of past messages directly. A more complex system will need to have a world model that it is constantly updating, which allows it to do things like maintain information about entities and their relationships.



Chains

Using an LLM in isolation is fine for simple applications, but more complex applications require chaining LLMs - either with each other or with other components.

LangChain provides the **Chain** interface for such "chained" applications. We define a Chain very generically as a sequence of calls to components, which can include other chains. The base interface is simple:

Why do we need chains?

Chains allow us to combine multiple components together to create a single, coherent application. For example, we can create a chain that takes user input, formats it with a PromptTemplate, and then passes the formatted response to an LLM. We can build more complex chains by combining multiple chains together, or by combining chains with other components.

LLMChain

We can now combine all these into one chain. This chain will take input variables, pass those to a prompt template to create a prompt, pass the prompt to an LLM, and then pass the output through an (optional) output parser. This is a convenient way to bundle up a modular piece of logic. Let's see it in action!

```
class CommaSeparatedListOutputParser(BaseOutputParser):
    """Parse the output of an LLM call to a comma-separated list."""


    def parse(self, text: str):
        """Parse the output of an LLM call."""
        return text.strip().split(", ")

template = """You are a helpful assistant who generates comma separated lists.
A user will pass in a category, and you should generated 5 objects in that category in a comma separated list.
ONLY return a comma separated list, and nothing more."""
system_message_prompt = SystemMessagePromptTemplate.from_template(template)
human_template = "{text}"
human_message_prompt = HumanMessagePromptTemplate.from_template(human_template)

chat_prompt = ChatPromptTemplate.from_messages([system_message_prompt, human_message_prompt])
chain = LLMChain(
    llm=ChatOpenAI(),
    prompt=chat_prompt,
    output_parser=CommaSeparatedListOutputParser()
)
chain.run("colors")
# >> ['red', 'blue', 'green', 'yellow', 'orange']
```

Tools

> **INFO**
>
> Head to Integrations for documentation on built-in tool integrations.

Tools are interfaces that an agent can use to interact with the world.

Get started

Tools are functions that agents can use to interact with the world. These tools can be generic utilities (e.g. search), other chains, or even other agents.

Currently, tools can be loaded with the following snippet:

```
from langchain.agents import load_tools
tool_names = [...]
tools = load_tools(tool_names)
```

Some tools (e.g. chains, agents) may require a base LLM to use to initialize them. In that case, you can pass in an LLM as well:

```
from langchain.agents import load_tools
tool_names = [...]
llm = ...
tools = load_tools(tool_names, llm=llm)
```

Completely New Tools - String Input and Output

The simplest tools accept a single query string and return a string output. If your tool function requires multiple arguments, you might want to skip down to the `StructuredTool` section below.

There are two ways to do this: either by using the Tool dataclass, or by subclassing the BaseTool class.

Tool dataclass

The 'Tool' dataclass wraps functions that accept a single string input and returns a string output.

```
# Load the tool configs that are needed.
search = SerpAPIWrapper()
llm_math_chain = LLMMathChain(llm=llm, verbose=True)
tools = [
    Tool.from_function(
        func=search.run,
        name="Search",
        description="useful for when you need to answer questions about current events"
        # coroutine= ... <- you can specify an async method if desired as well
    ),
]
```

Agents

The core idea of agents is to use an LLM to choose a sequence of actions to take. In chains, a sequence of actions is hardcoded (in code). In agents, a language model is used as a reasoning engine to determine which actions to take and in which order.

There are several key components here:

Agent

This is the class responsible for deciding what step to take next. This is powered by a language model and a prompt. This prompt can include things like:

1. The personality of the agent (useful for having it respond in a certain way)
2. Background context for the agent (useful for giving it more context on the types of tasks it's being asked to do)
3. Prompting strategies to invoke better reasoning (the most famous/widely used being ReAct)

LangChain provides a few different types of agents to get started. Even then, you will likely want to customize those agents with parts (1) and (2). For a full list of agent types see agent types

Agent types

Action agents

Agents use an LLM to determine which actions to take and in what order. An action can either be using a tool and observing its output, or returning a response to the user. Here are the agents available in LangChain.

Zero-shot ReAct

This agent uses the ReAct framework to determine which tool to use based solely on the tool's description. Any number of tools can be provided. This agent requires that a description is provided for each tool.

Note: This is the most general purpose action agent.

Structured input ReAct

The structured tool chat agent is capable of using multi-input tools. Older agents are configured to specify an action input as a single string, but this agent can use a tools' argument schema to create a structured action input. This is useful for more complex tool usage, like precisely navigating around a browser.

OpenAI Functions

Certain OpenAI models (like gpt-3.5-turbo-0613 and gpt-4-0613) have been explicitly fine-tuned to detect when a function should to be called and respond with the inputs that should be passed to the function. The OpenAI Functions Agent is designed to work with these models.

Conversational

This agent is designed to be used in conversational settings. The prompt is designed to make the agent helpful and conversational. It uses the ReAct framework to decide which tool to use, and uses memory to remember the previous conversation interactions.

Self ask with search

This agent utilizes a single tool that should be named `Intermediate Answer`. This tool should be able to lookup factual answers to questions. This agent is equivalent to the original self ask with search paper, where a Google search API was provided as the tool.

ReAct document store

This agent uses the ReAct framework to interact with a docstore. Two tools must be provided: a `Search` tool and a `Lookup` tool (they must be named exactly as so). The `Search` tool should search for a document, while the `Lookup` tool should lookup a term in the most recently found document. This agent is equivalent to the original ReAct paper, specifically the Wikipedia example.

ReAct

This walkthrough showcases using an agent to implement the ReAct logic.

```
from langchain.agents import load_tools
from langchain.agents import initialize_agent
from langchain.agents import AgentType
from langchain.llms import OpenAI
```

First, let's load the language model we're going to use to control the agent.

```
llm = OpenAI(temperature=0)
```

Next, let's load some tools to use. Note that the `llm-math` tool uses an LLM, so we need to pass that in.

```
tools = load_tools(["serpapi", "llm-math"], llm=llm)
```

Finally, let's initialize an agent with the tools, the language model, and the type of agent we want to use.

```
agent = initialize_agent(tools, llm, agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION, verbose=True)
```

```
agent.run("Who is Leo DiCaprio's girlfriend? What is her current age raised to the 0.43 power?")
```

```
> Entering new AgentExecutor chain...
 I need to find out who Leo DiCaprio's girlfriend is and then calculate her age raised to the 0.43
Action: Search
Action Input: "Leo DiCaprio girlfriend"
Observation: Camila Morrone
Thought: I need to find out Camila Morrone's age
Action: Search
Action Input: "Camila Morrone age"
Observation: 25 years
Thought: I need to calculate 25 raised to the 0.43 power
Action: Calculator
Action Input: 25^0.43
Observation: Answer: 3.991298452658078

Thought: I now know the final answer
Final Answer: Camila Morrone is Leo DiCaprio's girlfriend and her current age raised to the 0.43 po

> Finished chain.


"Camila Morrone is Leo DiCaprio's girlfriend and her current age raised to the 0.43 power is 3.9912
```

OpenAI functions

Certain OpenAI models (like gpt-3.5-turbo-0613 and gpt-4-0613) have been fine-tuned to detect when a function should to be called and respond with the inputs that should be passed to the function. In an API call, you can describe functions and have the model intelligently choose to output a JSON object containing arguments to call those functions. The goal of the OpenAI Function APIs is to more reliably return valid and useful function calls than a generic text completion or chat API.

The OpenAI Functions Agent is designed to work with these models.

Install openai,google-search-results packages which are required as the langchain packages call them internally

```
llm = ChatOpenAI(temperature=0, model="gpt-3.5-turbo-0613")
search = SerpAPIWrapper()
llm_math_chain = LLMMathChain.from_llm(llm=llm, verbose=True)
db = SQLDatabase.from_uri("sqlite:///../../../../../notebooks/Chinook.db")
db_chain = SQLDatabaseChain.from_llm(llm, db, verbose=True)
tools = [
    Tool(
        name = "Search",
        func=search.run,
        description="useful for when you need to answer questions about current events. You should ask
    ),
    Tool(
        name="Calculator",
        func=llm_math_chain.run,
        description="useful for when you need to answer questions about math"
    ),
    Tool(
        name="FooBar-DB",
        func=db_chain.run,
        description="useful for when you need to answer questions about FooBar. Input should be in the
    )
]
```

```
agent = initialize_agent(tools, llm, agent=AgentType.OPENAI_FUNCTIONS, verbose=True)
```

```
agent.run("Who is Leo DiCaprio's girlfriend? What is her current age raised to the 0.43 power?")
```

```
> Entering new  chain...

Invoking: `Search` with `{'query': 'Leo DiCaprio girlfriend'}`


Amidst his casual romance with Gigi, Leo allegedly entered a relationship with 19-year old model, E
Invoking: `Calculator` with `{'expression': '19^0.43'}`


> Entering new  chain...
19^0.43```text
19**0.43
```
...numexpr.evaluate("19**0.43")...

Answer: 3.547023357958959
> Finished chain.
Answer: 3.547023357958959Leo DiCaprio's girlfriend is reportedly Eden Polani. Her current age raise

> Finished chain.

"Leo DiCaprio's girlfriend is reportedly Eden Polani. Her current age raised to the power of 0.43 i
```
```

Vector Databases

Retrieval-Augmented Generation

Since the inception of large language models (LLMs), developers and researchers have tested the capabilities of combining information retrieval and text generation. By augmenting the LLM with a search engine, we no longer need to fine-tune the LLM to reason about our particular data.This method is known as Retrieval-augmented Generation (RAG).

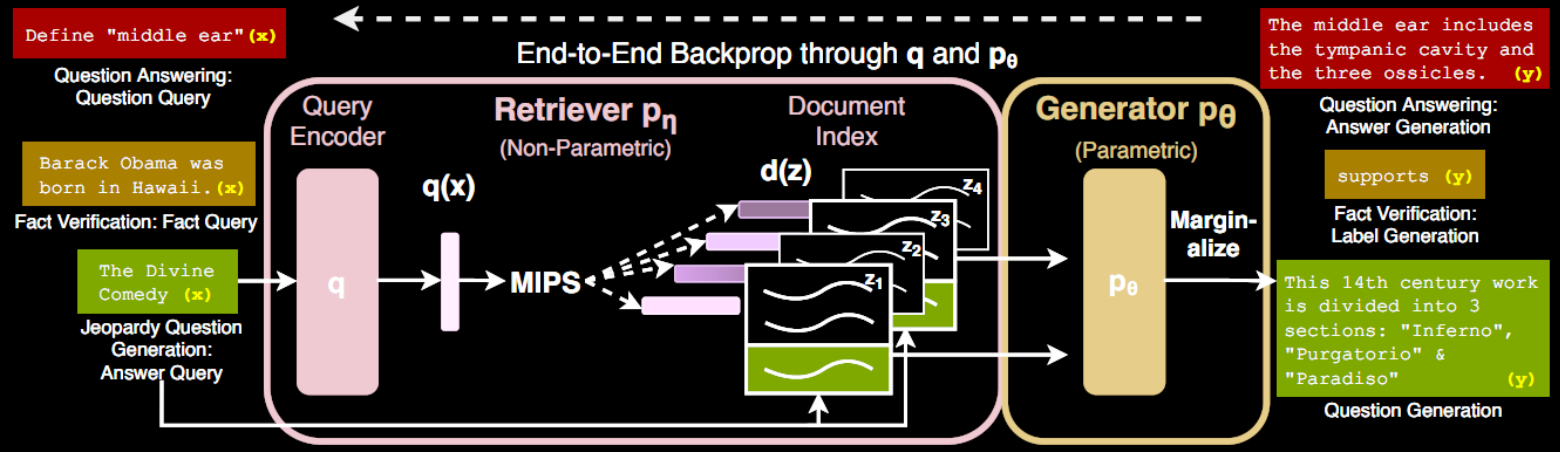
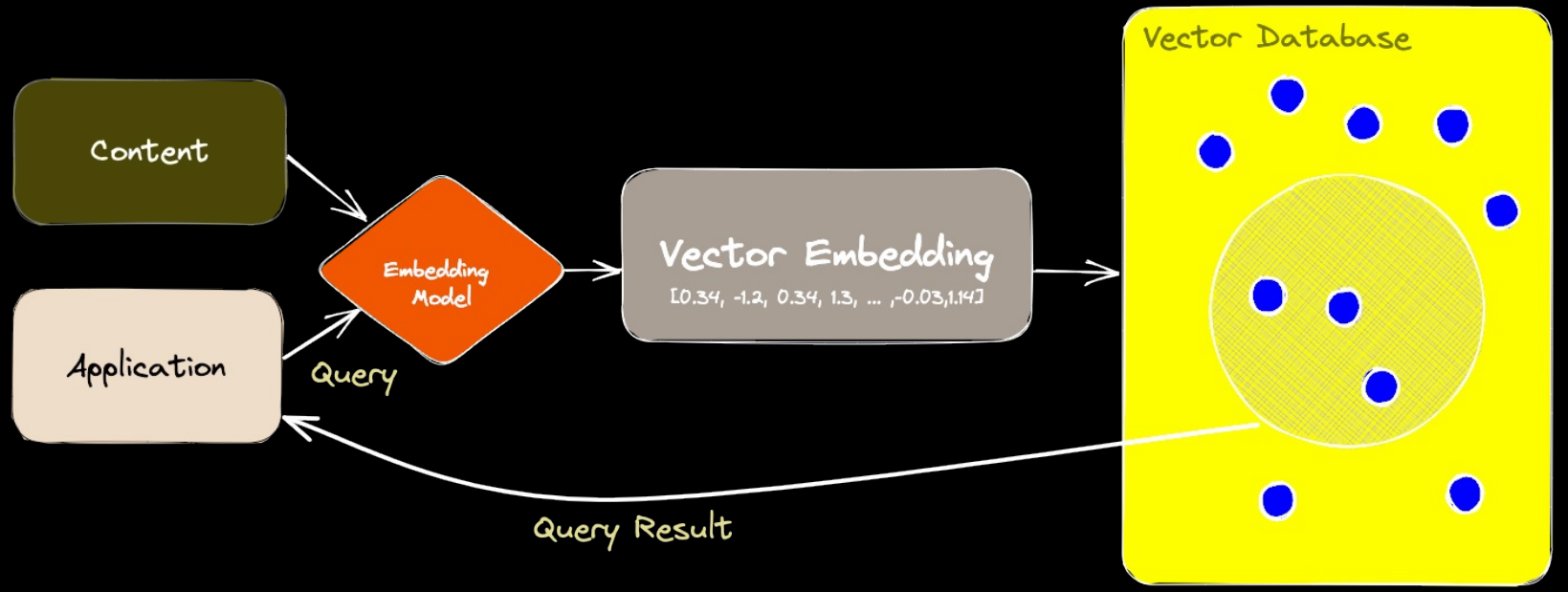


Figure 1: Overview of our approach. We combine a pre-trained retriever (*Query Encoder* + *Document Index*) with a pre-trained seq2seq model (*Generator*) and fine-tune end-to-end. For query x, we use Maximum Inner Product Search (MIPS) to find the top-K documents z_i. For final prediction y, we treat z as a latent variable and marginalize over seq2seq predictions given different documents.


{ "data": [{
"Wine": "Covey Run 2005 Chardonnay",
"Description": "... good with fish ..."
}]}
Traditional search engine
"Wine for seafood"
No products found...
Vector search engine
"Wine for seafood"
Covey Run 2005 Chardonnay


Content
Application
Embedding Model
Query
Vector Embedding
[0.34, -1.2, 0.34, 1.3, ... ,-0.03,1.14]
Vector Database
Query Result

How LLMs send Search Queries

Let's dig a little deeper into how prompts are transformed into search queries. The traditional approach is simply to use the prompt itself as a query. With the innovations in semantic vector search, we can send less formulated queries like this to search engines. For example, we can find information about "the eiffel tower" by searching for "landmarks in France".

However, sometimes we may find a better search result by formulating a question rather than sending a raw prompt. This is often done by prompting the language model to extract or formulate a question based on the prompt and then send that question to the search engine, i.e. "landmarks in France" translated to, "What are the landmarks in France?". Or a user prompt like "Please write me a poem about landmarks in France" translated to, "What are the landmarks in France?" to send to the search engine.

Refine

Map Reduce

Topics

1. Language translation
    1. Simplest program: Literal translation
    2. Dealing with missing tokens
2. Literal translation using "ML”
    1. Encoding: One hot
    2. Using matrices
    3. A 'dictionary' using matrices
    4. Decoding: Cosine similarity
3. Attention
    1. Similar tokens: Softmax
    2. Matrix Query: Q
    3. Scaled dot-product attention
4. Attention Head
    1. Weight matrices
    2. Connecting matrices
5. Revisiting tokens & encoding
    1. Tokenizing: BPE
    2. Encoding: Embeddings
6. Transformer Block
    1. Multi-headed attention
    2. Non-linear (feed forward) layer
    3. Stack blocks
    4. Masked attention
7. Transformer
    1. Stacking deep networks
    2. Normalization layers
    3. Skip connections
    4. Dropout
8. Pre-training, Training, Fine-tunning, Adapting, Instruct
    8. Pre-training numbers (GPU hours, params, etc.)
    9. LORA / PERF: Basic concepts
    10. "Instruct" models
9. Prompts all the way down
    8. "Chat": Just isolated requests with "memory" of conversations
    9. "Context": Just add a sentence to the prompt
    10. "Prompt engineering": Similar to adding the right words in a Google search
10. Frameworks
    8. LangChain (API, Models, LLM, Prompts, Agents). Simple examples
    9. Vector databases
    10. Huggingface
11. Scaling inference & training
    8. Single GPU: Quantization, fp16, etc.
    9. Multiple GPUs: PP, ZeRO, TP, Sharding, etc.

Scaling inference & Training

Batch sizes

One gets the most efficient performance when batch sizes and input/output neuron counts are divisible by a certain number, which typically starts at 8, but can be much higher as well. That number varies a lot depending on the specific hardware being used and the dtype of the model.

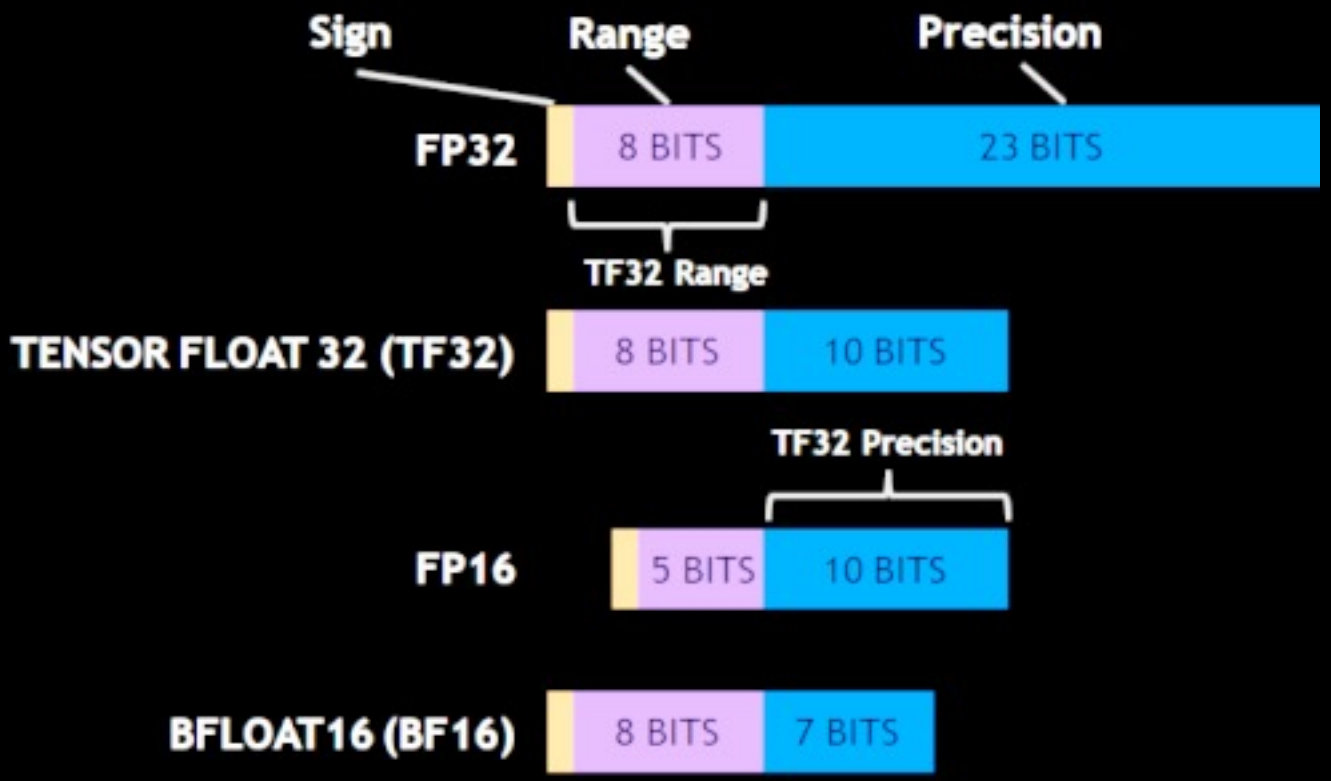
For example for fully connected layers (which correspond to GEMMs), NVIDIA provides recommendations for input/output neuron counts and batch size.

Tensor Core Requirements define the multiplier based on the dtype and the hardware. For example, for fp16 a multiple of 8 is recommended, but on A100 it's 64!

For parameters that are small, there is also Dimension Quantization Effects to consider, this is where tiling happens and the right multiplier can have a significant speedup.

Gradient Accumulation

The idea behind gradient accumulation is to instead of calculating the gradients for the whole batch at once to do it in smaller steps. The way we do that is to calculate the gradients iteratively in smaller batches by doing a forward and backward pass through the model and accumulating the gradients in the process. When enough gradients are accumulated we run the model's optimization step. This way we can easily increase the overall batch size to numbers that would never fit into the GPU's memory. In turn, however, the added forward and backward passes can slow down the training a bit.


Sign
Range
Precision
FP32
8 BITS
23 BITS
TF32 Range
TENSOR FLOAT 32 (TF32)
8 BITS
10 BITS
TF32 Precision
FP16
5 BITS
10 BITS
BFLOAT16 (BF16)
8 BITS
7 BITS

Multiple GPUs

NVIDIA NCCL

The NVIDIA Collective Communication Library (NCCL) implements multi-GPU and multi-node communication primitives optimized for NVIDIA GPUs and Networking. NCCL provides routines such as all-gather, all-reduce, broadcast, reduce, reduce-scatter as well as point-to-point send and receive that are optimized to achieve high bandwidth and low latency over PCIe and NVLink high-speed interconnects within a node and over NVIDIA Mellanox Network across nodes.

Multiple GPUs, One Node, Model fits in 1 GPU

DDP:

- At the start time the main process replicates the model once from gpu 0 to the rest of gpus
- Then for each batch:
  1. each gpu consumes each own mini-batch of data directly
  2. during `backward`, once the local gradients are ready, they are then averaged across all processes

ZeRO

La	Lb	Lc
a0	b0	c0
a1	b1	c1
a2	b2	c2

ZeRO

La	Lb	Lc
a0	b0	c0
a1	b1	c1
a2	b2	c2

GPU0:

La	Lb	Lc
a0	b0	c0

GPU1:

La	Lb	Lc
a1	b1	c1

GPU2:

La	Lb	Lc
a2	b2	c2

ZeRO

```
La | Lb | Lc
---|----|---
a0 | b0 | c0
a1 | b1 | c1
a2 | b2 | c2
```

```
GPU0:
La | Lb | Lc
---|----|---
a0 | b0 | c0

GPU1:
La | Lb | Lc
---|----|---
a1 | b1 | c1

GPU2:
La | Lb | Lc
---|----|---
a2 | b2 | c2
```

```
x0 => GPU0
x1 => GPU1
x2 => GPU2
```

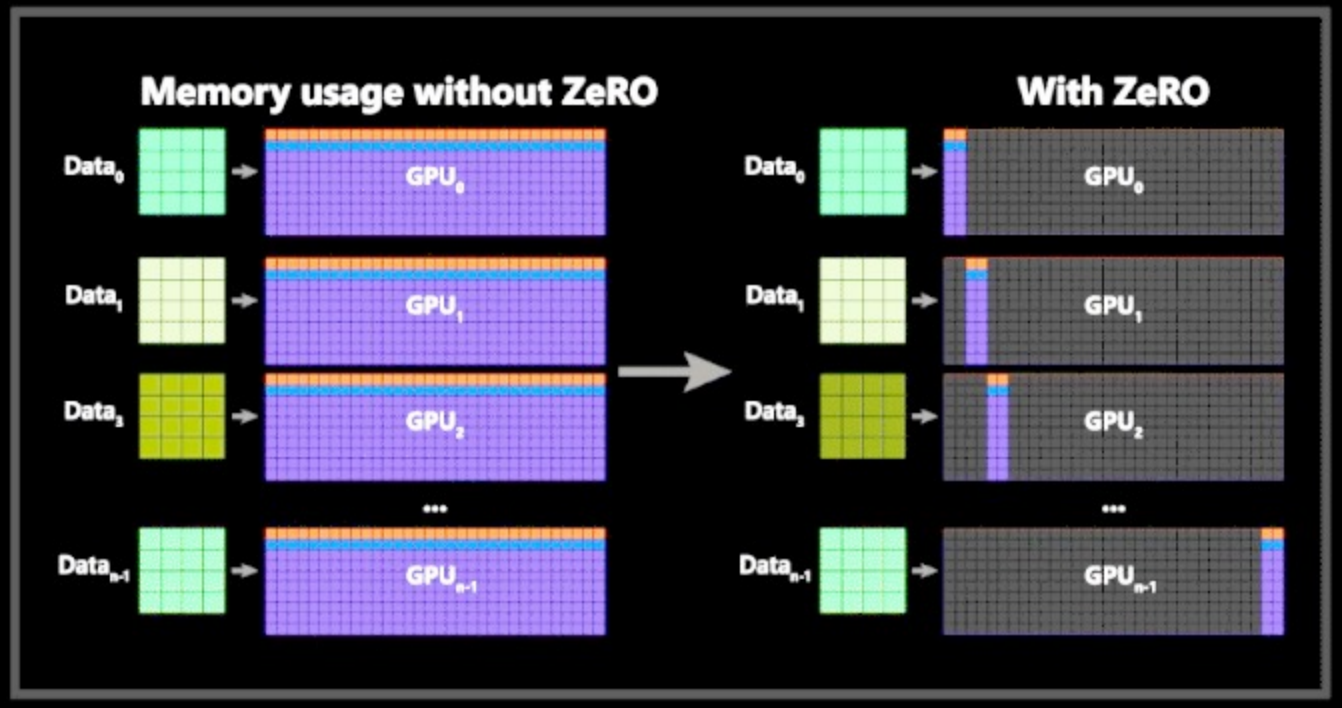
Let's focus just on GPU0: x0 needs a0, a1, a2 params to do its forward path, but GPU0 has only a0 - it gets sent a1 from GPU1 and a2 from GPU2, bringing all pieces of the model together.

In parallel, GPU1 gets mini-batch x1 and it only has a1, but needs a0 and a2 params, so it gets those from GPU0 and GPU2.

Same happens to GPU2 that gets input x2. It gets a0 and a1 from GPU0 and GPU1, and with its a2 it reconstructs the full tensor.

All 3 GPUs get the full tensors reconstructed and a forward happens.

ZeRO


Memory usage without ZeRO
With ZeRO
Data0
Data1
Data2
Data n-1
GPU0
GPU1
GPU2
GPU n-1

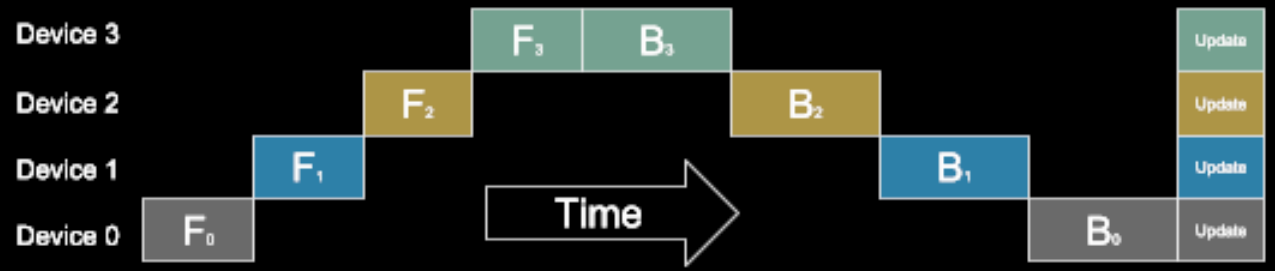
Naïve Model Parallelism

Naive Model Parallelism (Vertical) and Pipeline Parallelism

Naive Model Parallelism (MP) is where one spreads groups of model layers across multiple GPUs. The mechanism is relatively simple - switch the desired layers `.to()` the desired devices and now whenever the data goes in and out those layers switch the data to the same device as the layer and leave the rest unmodified.

We refer to it as Vertical MP, because if you remember how most models are drawn, we slice the layers vertically. For example, if the following diagram shows an 8-layer model:

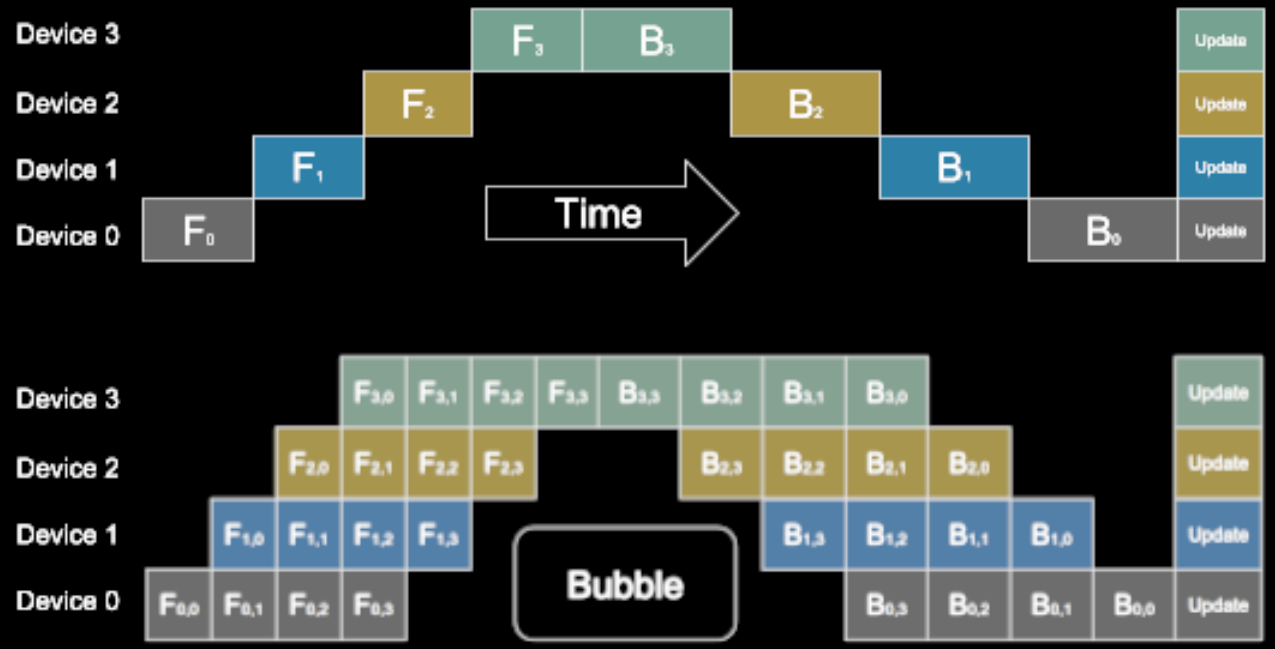
```
=================  =================
|  0 | 1 | 2 | 3  |  |  4 | 5 | 6 | 7  |
=================  =================
        gpu0               gpu1
```



The naive model parallelism strategy leads to severe underutilization due to the sequential nature of the network. Only one accelerator is active at a time.

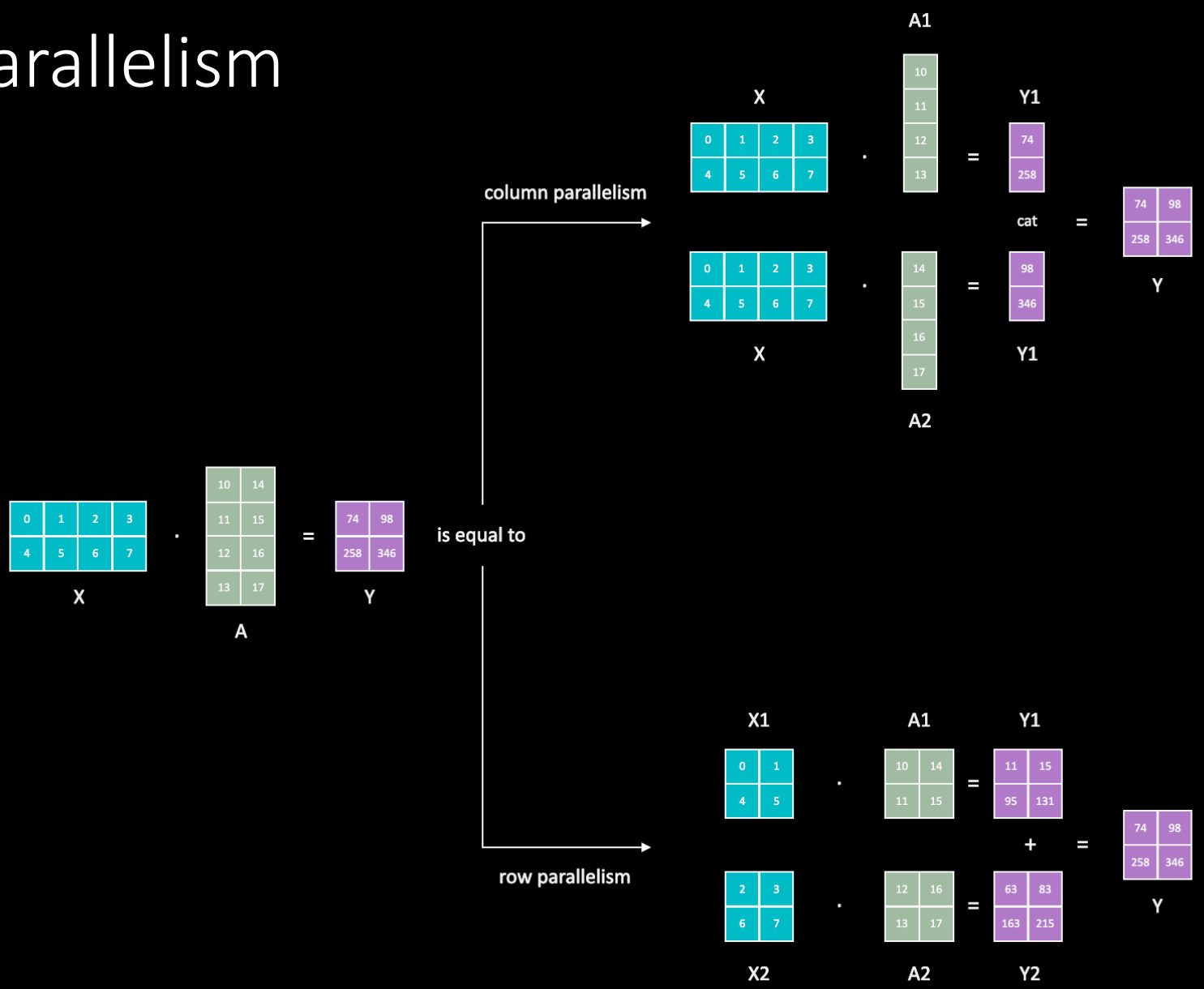
Pipe Parallelism

Pipeline Parallelism (PP) is almost identical to a naive MP, but it solves the GPU idling problem, by chunking the incoming batch into micro-batches and artificially creating a pipeline, which allows different GPUs to concurrently participate in the computation process.



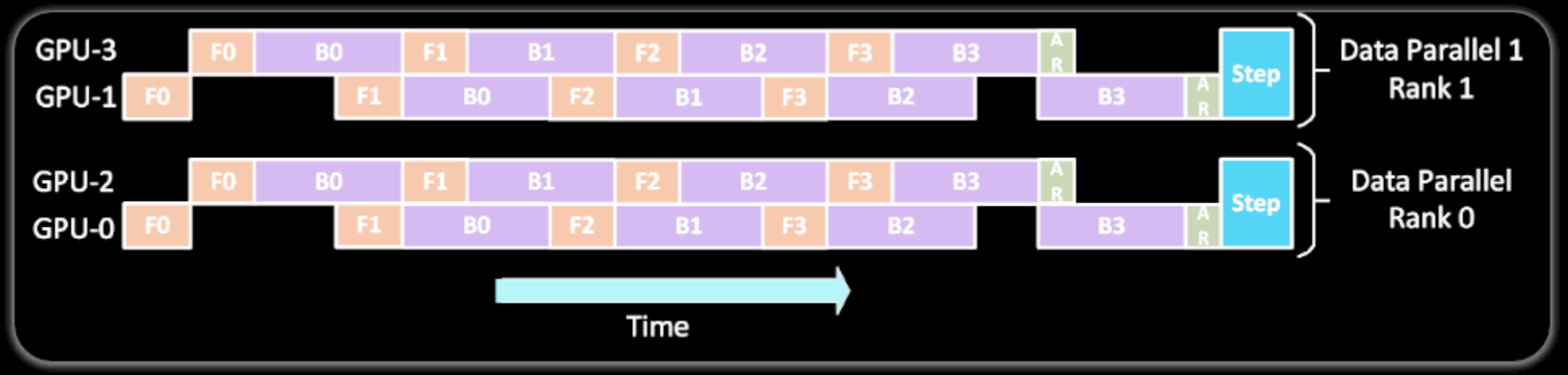
***Top:** The naive model parallelism strategy leads to severe underutilization due to the sequential nature of the network. Only one accelerator is active at a time. **Bottom:** GPipe divides the input mini-batch into smaller micro-batches, enabling different accelerators to work on separate micro-batches at the same time.*

Tensor Parallelism



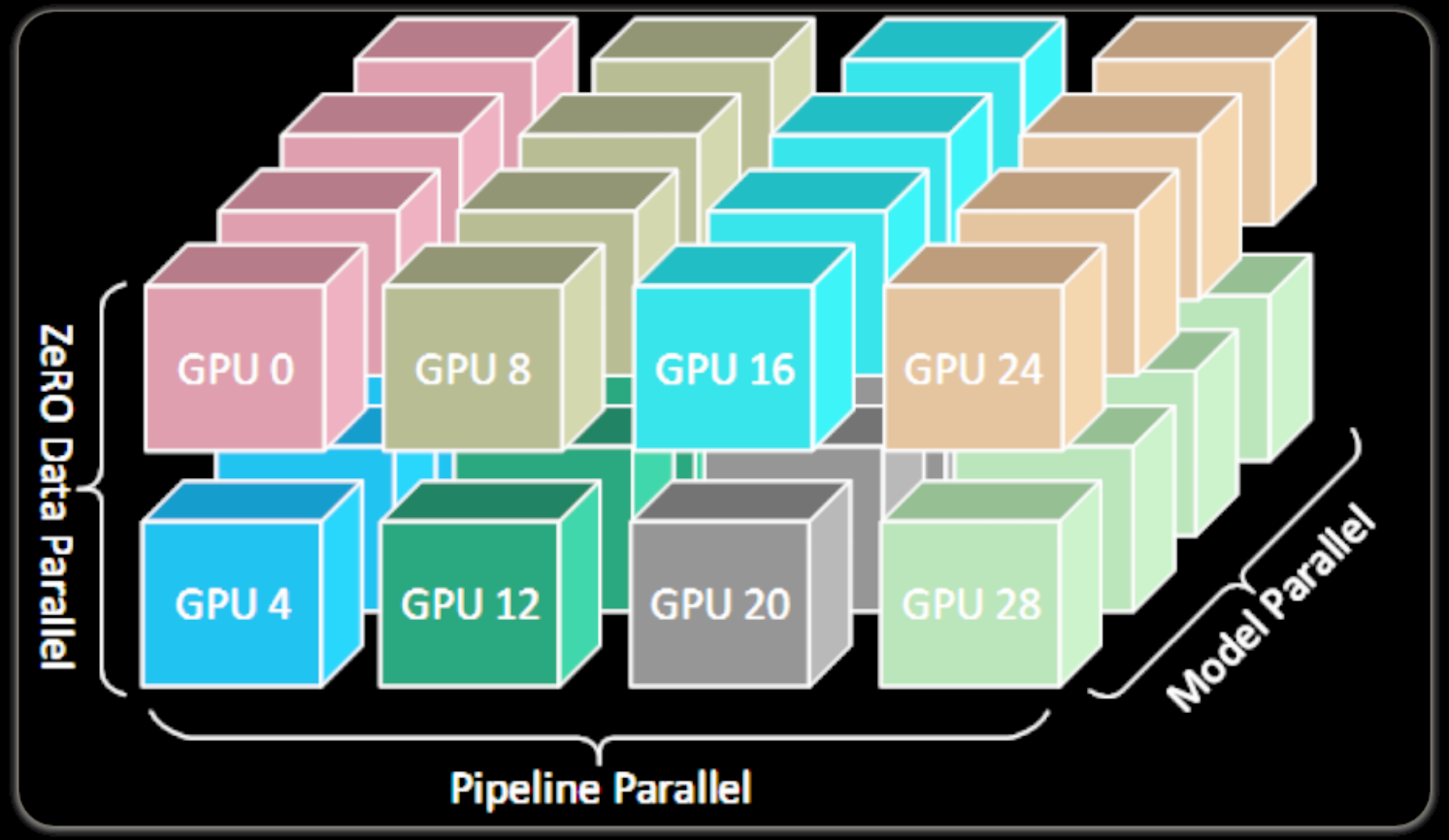
DP+PP

The following diagram from the DeepSpeed pipeline tutorial demonstrates how one combines DP with PP.



DP+PP+TP

To get an even more efficient training a 3D parallelism is used where PP is combined with TP and DP. This can be seen in the following diagram.



End