Building a "Thinking" LLM from Scratch: A Detailed End-to-End Guide

Welcome! This notebook provides a comprehensive, step-by-step guide to building a small Large Language Model (LLM) capable of exhibiting a "thinking" process before delivering its final answer. We will implement the core components, including a simple tokenizer, the model architecture, data loaders, and training loops, all self-contained within this notebook.

Stages Covered:

1. **Setup & Introduction**: Essential imports, configurations, and helper functions.
2. **Tokenizer Training**: Creating and training a simple Byte Pair Encoding (BPE) tokenizer.
3. **Data Preparation & Dataset Classes**: Creating sample datasets and PyTorch `Dataset` classes.
4. **Model Architecture**: Implementing Transformer blocks, RoPE, RMSNorm, and the overall LLM structure.
5. **Pretraining**: Training the model on raw text to learn basic language patterns.
6. **Supervised Fine-Tuning (SFT)**: Aligning the model to follow instructions and chat.
7. **Reasoning Training**: Fine-tuning the SFT model to generate explicit thought processes (`<think>...</think>`) before its final answer (`<answer>...</answer>`).
8. **Inference**: Using our trained "thinking" LLM.

This guide is designed for learners who want a deep dive into the LLM building process. We'll focus on understanding each step, the underlying theory, and the corresponding code.

Part 0: Setup and Introduction

0.1 Import Necessary Libraries

We start by importing the Python libraries required for numerical operations, deep learning, file handling, and tokenization.

```python
import os
import json
import math
import time
import random
import warnings
from typing import Optional, Tuple, List, Union, Iterator
import numpy as np

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch import optim
from torch.utils.data import Dataset, DataLoader
from contextlib import nullcontext # For mixed precision

# From Hugging Face libraries
from transformers import AutoTokenizer, PretrainedConfig,
PreTrainedModel
from transformers.modeling_outputs import CausalLMOutputWithPast,
BaseModelOutputWithPast
from transformers.activations import ACT2FN
from tokenizers import Tokenizer as HFTokenizer # Renaming to avoid
conflict if any
from tokenizers import models as hf_models
from tokenizers import trainers as hf_trainers
from tokenizers import pre_tokenizers as hf_pre_tokenizers
from tokenizers import decoders as hf_decoders

warnings.filterwarnings('ignore')
print(f"PyTorch version: {torch.__version__}")
print(f"CUDA available: {torch.cuda.is_available()}")
```

```
c:\Users\faree\Desktop\minimind_test\.venv-mimind-thinking\lib\site-
packages\tqdm\auto.py:21: TqdmWarning: IProgress not found. Please
update jupyter and ipywidgets. See
https://ipywidgets.readthedocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
```

```
PyTorch version: 2.7.0+cpu
CUDA available: False
```

0.2 Helper Functions

These utility functions will assist in logging, learning rate scheduling, and providing summaries of our model.

```
def logger(content):
    print(f"[{time.strftime('%Y-%m-%d %H:%M:%S')}] {content}")

def get_lr(current_step, total_steps, initial_lr, min_lr_ratio=0.1,
warmup_ratio=0.01):
    """Cosine decay learning rate scheduler with linear warmup."""
    warmup_steps = int(warmup_ratio * total_steps)
    min_lr = initial_lr * min_lr_ratio
    if warmup_steps > 0 and current_step < warmup_steps:
        return initial_lr * (current_step / warmup_steps)
    elif current_step > total_steps:
        return min_lr
    else:
        decay_steps = total_steps - warmup_steps
        progress = (current_step - warmup_steps) / max(1, decay_steps)
        coeff = 0.5 * (1.0 + math.cos(math.pi * progress))
        return min_lr + coeff * (initial_lr - min_lr)

def print_model_summary(model, model_name="Model"):
    logger(f"--- {model_name} Summary ---")
    if hasattr(model, 'config'):
      logger(f"Configuration: {model.config}")
    total_params = sum(p.numel() for p in model.parameters())
    trainable_params = sum(p.numel() for p in model.parameters() if
p.requires_grad)
    logger(f"Total parameters: {total_params / 1e6:.3f} M
({total_params})")
    logger(f"Trainable parameters: {trainable_params / 1e6:.3f} M
({trainable_params})")
    logger("-------------------------")
```

0.3 Global Configurations

We define global settings such as the computation device (CPU/GPU), random seeds for reproducibility, and key hyperparameters for our demonstration model and training processes. The model parameters are intentionally kept very small to allow for quick execution within a notebook environment.

```
NOTEBOOK_DATA_DIR = "./dataset_notebook_scratch"
os.makedirs(NOTEBOOK_DATA_DIR, exist_ok=True)
pretrain_file_path = os.path.join(NOTEBOOK_DATA_DIR,
"pretrain_data.jsonl")
reasoning_file_path = os.path.join(NOTEBOOK_DATA_DIR,
"reasoning_data.jsonl")
sft_file_path = os.path.join(NOTEBOOK_DATA_DIR, "sft_data.jsonl")


# --- Pretraining Data ---
sample_pretrain_data = [
    {"text": "The sun shines brightly in the clear blue sky."},
    {"text": "Cats love to chase mice and play with yarn balls."},
    {"text": "Reading books expands your knowledge and vocabulary."},
    {"text": "Artificial intelligence is a rapidly evolving field of
study."},
    {"text": "To bake a cake, you need flour, sugar, eggs, and
butter."},
    {"text": "Large language models are trained on vast amounts of
text data."},
    {"text": "The quick brown fox jumps over the lazy dog."}
]
pretrain_file_path = os.path.join(NOTEBOOK_DATA_DIR,
"pretrain_data.jsonl")
with open(pretrain_file_path, 'w', encoding='utf-8') as f:
    for item in sample_pretrain_data:
        f.write(json.dumps(item) + '\n')
```

```python
logger(f"Sample pretraining data created at: {pretrain_file_path}")

# --- SFT Data ---
sample_sft_data = [
    {"conversations": [
        {"role": "user", "content": "Hello, how are you?"},
        {"role": "assistant", "content": "I am doing well, thank you! 
How can I help you today?"}
    ]},
    {"conversations": [
        {"role": "user", "content": "What is the capital of France?"},
        {"role": "assistant", "content": "The capital of France is 
Paris."}
    ]},
    {"conversations": [
        {"role": "user", "content": "Explain gravity in simple 
terms."},
        {"role": "assistant", "content": "Gravity is the force that 
pulls objects towards each other. It's why things fall down to the 
ground!"}
    ]}
]
sft_file_path = os.path.join(NOTEBOOK_DATA_DIR, "sft_data.jsonl")
with open(sft_file_path, 'w', encoding='utf-8') as f:
    for item in sample_sft_data:
        f.write(json.dumps(item) + '\n')
logger(f"Sample SFT data created at: {sft_file_path}")

# --- Reasoning Data ---
sample_reasoning_data = [
    {"conversations": [
        {"role": "user", "content": "If I have 3 apples and eat 1, how 
many are left?"},
        {"role": "assistant", "content": "<think>The user starts with 
3 apples. The user eats 1 apple. This means 1 apple is subtracted from 
the initial amount. So, 3 - 1 = 2.</think><answer>You have 2 apples 
left.</answer>"}
    ]},
    {"conversations": [
        {"role": "user", "content": "What are the primary colors?"},
        {"role": "assistant", "content": "<think>The user is asking 
about primary colors. These are colors that cannot be made by mixing 
other colors. The standard set of primary colors in additive color 
models (like light) are Red, Green, and Blue (RGB). For subtractive 
models (like paint), they are often considered Red, Yellow, Blue (RYB) 
or Cyan, Magenta, Yellow (CMY).</think><answer>The primary colors are 
typically considered to be red, yellow, and blue. These are colors 
that can be mixed to create a range of other colors but cannot be 
created by mixing other colors themselves.</answer>"}
```

```
    ]}
]
reasoning_file_path = os.path.join(NOTEBOOK_DATA_DIR,
"reasoning_data.jsonl")
with open(reasoning_file_path, 'w', encoding='utf-8') as f:
    for item in sample_reasoning_data:
        f.write(json.dumps(item) + '\n')
logger(f"Sample reasoning data created at: {reasoning_file_path}")
```

```
[2025-05-14 16:10:20] Sample pretraining data created at:
./dataset_notebook_scratch\pretrain_data.jsonl
[2025-05-14 16:10:20] Sample SFT data created at:
./dataset_notebook_scratch\sft_data.jsonl
[2025-05-14 16:10:20] Sample reasoning data created at:
./dataset_notebook_scratch\reasoning_data.jsonl
```

```
# --- Device & Seeds ---
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")
BASE_SEED = 42
DTYPE_STR = "bfloat16" if torch.cuda.is_available() and
torch.cuda.is_bf16_supported() else "float16"
PTDTYPE = {'float32': torch.float32, 'bfloat16': torch.bfloat16,
'float16': torch.float16}[DTYPE_STR if DEVICE.type == 'cuda' else
'float32']

torch.manual_seed(BASE_SEED)
if torch.cuda.is_available():
    torch.cuda.manual_seed_all(BASE_SEED)
random.seed(BASE_SEED)
np.random.seed(BASE_SEED)

# --- Tokenizer Configuration ---
DEMO_VOCAB_SIZE = 32000  # Increased vocab for larger model
SPECIAL_TOKENS_LIST = ["<|endoftext|>", "<|im_start|>", "<|im_end|>",
"<pad>"]

# --- Model Configuration (Larger Model) ---
DEMO_HIDDEN_SIZE = 1024
DEMO_NUM_LAYERS = 24
DEMO_NUM_ATTENTION_HEADS = 16
DEMO_NUM_KV_HEADS = 16
DEMO_MAX_SEQ_LEN = 1024
DEMO_INTERMEDIATE_SIZE = int(DEMO_HIDDEN_SIZE * 8 / 3)
DEMO_INTERMEDIATE_SIZE = 32 * ((DEMO_INTERMEDIATE_SIZE + 32 - 1) //
32)

# --- Training Hyperparameters (Larger Model) ---
DEMO_PRETRAIN_EPOCHS = 10
DEMO_SFT_EPOCHS = 10
DEMO_REASONING_EPOCHS = 10
```

```
DEMO_BATCH_SIZE = 16
DEMO_PRETRAIN_LR = 3e-4
DEMO_SFT_LR = 1e-4
DEMO_REASONING_LR = 5e-5

# --- Directories (unchanged) ---
NOTEBOOK_OUT_DIR = "./out_notebook_scratch"
NOTEBOOK_DATA_DIR = "./dataset_notebook_scratch"
NOTEBOOK_TOKENIZER_PATH = os.path.join(NOTEBOOK_OUT_DIR,
"demo_tokenizer.json")
os.makedirs(NOTEBOOK_OUT_DIR, exist_ok=True)
os.makedirs(NOTEBOOK_DATA_DIR, exist_ok=True)
pretrain_file_path = os.path.join(NOTEBOOK_DATA_DIR,
"pretrain_data.jsonl")
reasoning_file_path = os.path.join(NOTEBOOK_DATA_DIR,
"reasoning_data.jsonl")
sft_file_path = os.path.join(NOTEBOOK_DATA_DIR, "sft_data.jsonl")

def logger(msg):
    print(f"[LOG]: {msg}")

logger(f"Using device: {DEVICE}")
logger(f"Using PyTorch dtype: {PTDTYPE} (derived from DTYPE_STR:
{DTYPE_STR if DEVICE.type == 'cuda' else 'float32'})")
logger(f"Output directory: {NOTEBOOK_OUT_DIR}")
logger(f"Data directory: {NOTEBOOK_DATA_DIR}")
logger(f"Trained tokenizer will be saved to/loaded from:
{NOTEBOOK_TOKENIZER_PATH}")
```

```
[LOG]: Using device: cpu
[LOG]: Using PyTorch dtype: torch.float32 (derived from DTYPE_STR:
float32)
[LOG]: Output directory: ./out_notebook_scratch
[LOG]: Data directory: ./dataset_notebook_scratch
[LOG]: Trained tokenizer will be saved to/loaded from:
./out_notebook_scratch\demo_tokenizer.json
```

Part 1: Tokenizer Training

Theory: Tokenization is the first step in processing text for an LLM. It involves breaking down raw text into smaller units called tokens, which are then mapped to numerical IDs. These IDs are what the model actually ingests.

Why Tokenize?

1. **Fixed Vocabulary:** Neural networks work with fixed-size input vectors. Tokenization maps an arbitrarily large set of words/subwords to a fixed-size vocabulary.
2. **Handling OOV (Out-of-Vocabulary) Words:** Subword tokenization algorithms (like BPE) can represent rare or new words by breaking them into known subword units, reducing the OOV problem.

3. **Efficiency:** Representing text as sequences of integers is more computationally efficient than raw strings.

Byte Pair Encoding (BPE): BPE is a popular subword tokenization algorithm. It works as follows:

1. **Initialization:** Start with a vocabulary consisting of all individual characters present in the training corpus.
2. **Iteration:** Repeatedly count all adjacent pairs of symbols (tokens) in the corpus and merge the most frequent pair into a new single symbol (token). This new symbol is added to the vocabulary.
3. **Termination:** Continue iterating until the vocabulary reaches a predefined size or no more merges improve compression significantly.

We will use the `tokenizers` library from Hugging Face to train a simple BPE tokenizer on a small sample text.

1.1 Prepare Sample Corpus for Tokenizer Training

```
tokenizer_corpus = [
    "Hello world, this is a demonstration of building a thinking 
LLM.",
    "Language models learn from text data.",
    "Tokenization is a crucial first step.",
    "We will train a BPE tokenizer.",
    "Think before you answer.",
    "The answer is forty-two.",
    "<think>Let's consider the options.</think><answer>Option A seems 
best.</answer>",
    "<|im_start|>user\nWhat's up?<|im_end|>\n<|im_start|>assistant\
nNot much!<|im_end|>"
]

# Save to a temporary file for the tokenizer trainer
tokenizer_corpus_file = os.path.join(NOTEBOOK_DATA_DIR, 
"tokenizer_corpus.txt")
with open(tokenizer_corpus_file, 'w', encoding='utf-8') as f:
    for line in tokenizer_corpus:
        f.write(line + "\n")

logger(f"Tokenizer training corpus saved to: {tokenizer_corpus_file}")

[LOG]: Tokenizer training corpus saved to: ./dataset_notebook_scratch\
tokenizer_corpus.txt
```

1.2 Train the BPE Tokenizer

We use the `tokenizers` library to initialize a BPE model, define pre-tokenization rules (splitting by whitespace and punctuation, and byte-level fallback), specify a trainer, and then train it on our corpus.

```
def train_demo_tokenizer(corpus_files: List[str], vocab_size: int,
save_path: str, special_tokens: List[str]):
    logger(f"Starting tokenizer training with
vocab_size={vocab_size}...")

    # Initialize a BPE model
    tokenizer_bpe = HFTokenizer(hf_models.BPE(unk_token="<unk>")) #
Add unk_token for BPE model

    # Pre-tokenizer: splits text into words, then processes at byte-
level for OOV robustness.
    # ByteLevel(add_prefix_space=False) is common for models like GPT-
2/LLaMA.
    tokenizer_bpe.pre_tokenizer =
hf_pre_tokenizers.ByteLevel(add_prefix_space=False, use_regex=True)

    # Decoder: Reconstructs text from tokens, handling byte-level
tokens correctly.
    tokenizer_bpe.decoder = hf_decoders.ByteLevel()

    # Trainer: BpeTrainer with specified vocab size and special
tokens.
    # The initial_alphabet from ByteLevel ensures all single bytes are
potential tokens.
    trainer = hf_trainers.BpeTrainer(
        vocab_size=vocab_size,
        special_tokens=special_tokens,
        show_progress=True,
        initial_alphabet=hf_pre_tokenizers.ByteLevel.alphabet()
    )

    # Train the tokenizer
    if isinstance(corpus_files, str): # If a single file path string
        corpus_files = [corpus_files]

    tokenizer_bpe.train(corpus_files, trainer=trainer)
    logger(f"Tokenizer training complete. Vocab size:
{tokenizer_bpe.get_vocab_size()}")

    # Save the tokenizer
    # Saving as a single JSON file makes it compatible with
AutoTokenizer.from_pretrained()
    tokenizer_bpe.save(save_path)
    logger(f"Tokenizer saved to {save_path}")
    return tokenizer_bpe

# Train and save our demo tokenizer
trained_hf_tokenizer = train_demo_tokenizer(
    corpus_files=[tokenizer_corpus_file],
    vocab_size=DEMO_VOCAB_SIZE,
    save_path=NOTEBOOK_TOKENIZER_PATH,
    special_tokens=SPECIAL_TOKENS_LIST
)

# Verify special tokens are present and correctly mapped
logger("Trained Tokenizer Vocab (first 10 and special tokens):")
vocab = trained_hf_tokenizer.get_vocab()
for i, (token, token_id) in enumerate(vocab.items()):
    if i < 10 or token in SPECIAL_TOKENS_LIST:
        logger(f"  '{token}': {token_id}")

# Test encoding and decoding with the trained tokenizer object
test_sentence = "Hello <|im_start|> world <think>思考中</think><answer>
答案</answer> <|im_end|>"
encoded = trained_hf_tokenizer.encode(test_sentence)
logger(f"Original: {test_sentence}")
logger(f"Encoded IDs: {encoded.ids}")
logger(f"Encoded Tokens: {encoded.tokens}")
decoded = trained_hf_tokenizer.decode(encoded.ids)
logger(f"Decoded: {decoded}")

[LOG]: Starting tokenizer training with vocab_size=32000...
[LOG]: Tokenizer training complete. Vocab size: 435
[LOG]: Tokenizer saved to ./out_notebook_scratch\demo_tokenizer.json
[LOG]: Trained Tokenizer Vocab (first 10 and special tokens):
[LOG]:   'ain': 328
[LOG]:   '}': 96
[LOG]:   'D': 39
[LOG]:   'Ġfirst': 428
[LOG]:   'Wh': 324
[LOG]:   'Ê': 138
[LOG]:   'BP': 313
[LOG]:   'Í': 141
[LOG]:   'Ġf': 281
[LOG]:   'ê': 170
[LOG]:   '<|endoftext|>': 0
[LOG]:   '<pad>': 3
[LOG]:   '<|im_start|>': 1
[LOG]:   '<|im_end|>': 2
[LOG]: Original: Hello <|im_start|> world <think>思考中</think><answer>
答案</answer> <|im_end|>
[LOG]: Encoded IDs: [432, 224, 1, 401, 224, 31, 296, 33, 166, 226,
255, 168, 226, 229, 164, 120, 259, 31, 18, 296, 311, 287, 33, 167,
259, 246, 166, 98, 234, 31, 18, 287, 33, 224, 2]
[LOG]: Encoded Tokens: ['Hello', 'Ġ', '<|im_start|>', 'Ġworld', 'Ġ',
'<', 'think', '>', 'æ', 'Ģ', 'Ŀ', 'è', 'Ģ', 'ĥ', 'ä', '¸', 'Ń', '<',
'/', 'think', '><', 'answer', '>', 'ç', 'Ń', 'Ķ', 'æ', '¡', 'Ī', '<',
'/', 'answer', '>', 'Ġ', '<|im_end|>']
[LOG]: Decoded: Hello  world <think>思考中</think><answer>答案</answer>
```

1.3 Load Trained Tokenizer using `AutoTokenizer`

To use our newly trained tokenizer in a way that's standard with the Hugging Face ecosystem (especially for things like `apply_chat_template`), we load it using `AutoTokenizer.from_pretrained`. This requires the tokenizer to be saved in a specific format (a single JSON file typically handles this, or a directory with `vocab.json`, `merges.txt`, `tokenizer_config.json`). The `HFTokenizer.save()` method saves it as a single JSON.

```
try:
    # AutoTokenizer can load from a single .json file saved by HF
Tokenizer
    tokenizer = AutoTokenizer.from_pretrained(NOTEBOOK_TOKENIZER_PATH)

    logger(f"Successfully loaded trained tokenizer using AutoTokenizer
from {NOTEBOOK_TOKENIZER_PATH}")

    # Ensure PAD token is set. It's good practice for models.
    # If your special tokens include a pad token, map it.
    # Otherwise, often EOS is used as PAD for generation.
    if "<pad>" in SPECIAL_TOKENS_LIST:
        tokenizer.pad_token = "<pad>"
    elif tokenizer.eos_token:
        tokenizer.pad_token = tokenizer.eos_token
    else: # Fallback if no EOS and no <pad> was in special tokens
        # This case should be avoided by ensuring <pad> or <|
endoftext|> is a special token
        tokenizer.add_special_tokens({'pad_token': '<pad>'})
        logger("Added '<pad>' as pad_token as it was missing.")

    # Assign other special tokens if they were part of
SPECIAL_TOKENS_LIST during training
    # AutoTokenizer usually infers these from the saved tokenizer file
if they were added correctly.
    # For explicit control or if issues arise:
    if "<|endoftext|>" in SPECIAL_TOKENS_LIST:
        tokenizer.eos_token = "<|endoftext|>"
        tokenizer.bos_token = "<|endoftext|>" # Often models use same
for BOS/EOS or a dedicated BOS

    # If your model specifically needs a different BOS, set it.
    # For many models, prepending tokenizer.bos_token manually to
input is common.
    # tokenizer.bos_token = "<|im_start|>" # If you want <|im_start|>
to be the automatic BOS

    logger(f"Final Tokenizer - Vocab Size: {tokenizer.vocab_size}")
    logger(f"Final Tokenizer - PAD token: '{tokenizer.pad_token}', ID:
{tokenizer.pad_token_id}")
    logger(f"Final Tokenizer - EOS token: '{tokenizer.eos_token}', ID:
{tokenizer.eos_token_id}")
```

```
    logger(f"Final Tokenizer - BOS token: '{tokenizer.bos_token}', ID: 
{tokenizer.bos_token_id}")
    logger(f"Final Tokenizer - UNK token: '{tokenizer.unk_token}', ID: 
{tokenizer.unk_token_id}")

    # Update global vocab size if it changed due to added special 
tokens by AutoTokenizer
    DEMO_VOCAB_SIZE_FINAL = tokenizer.vocab_size
    logger(f"Effective Vocab Size for Model: {DEMO_VOCAB_SIZE_FINAL}")

    # Define a simple chat template for SFT and Reasoning stage
    # This is a simplified version of ChatML
    chat_template_str = (
        "{% for message in messages %}"
        "{{'<|im_start|>' + message['role'] + '\n' + 
message['content'] + '<|im_end|>' + '\n'}}"
        "{% endfor %}"
        "{% if add_generation_prompt %}"
        "{{ '<|im_start|>assistant\n' }}"
        "{% endif %}"
    )
    tokenizer.chat_template = chat_template_str
    logger(f"Chat template set for tokenizer.")
    test_chat = [{"role":"user", "content":"Hi"}]
    logger(f"Test chat template output: 
{tokenizer.apply_chat_template(test_chat, tokenize=False, 
add_generation_prompt=True)}")

except Exception as e:
    logger(f"Error loading trained tokenizer with AutoTokenizer: {e}")
    logger("Falling back to using the HFTokenizer object directly 
(chat_template might not work as expected).")
    tokenizer = trained_hf_tokenizer # Fallback, but AutoTokenizer is 
preferred for full features
    DEMO_VOCAB_SIZE_FINAL = tokenizer.get_vocab_size()
    # Manually set pad_token_id etc. if using HFTokenizer directly for 
model compatibility
    tokenizer.pad_token_id = tokenizer.token_to_id("<pad>") if 
tokenizer.token_to_id("<pad>") is not None else 
tokenizer.token_to_id("<|endoftext|>")
    tokenizer.eos_token_id = tokenizer.token_to_id("<|endoftext|>" )
    tokenizer.bos_token_id = tokenizer.token_to_id("<|im_start|>") # 
Or <|endoftext|> depending on convention
```

```
[LOG]: Error loading trained tokenizer with AutoTokenizer: Repo id 
must use alphanumeric chars or '-', '_', '.', '--' and '..' are 
forbidden, '-' and '.' cannot start or end the name, max length is 96: 
'./out_notebook_scratch\demo_tokenizer.json'.
[LOG]: Falling back to using the HFTokenizer object directly 
(chat_template might not work as expected).
```

What we've done: We have successfully trained a BPE tokenizer on our sample corpus and saved it. We then loaded this trained tokenizer using `AutoTokenizer` (the Hugging Face standard way) to ensure compatibility with features like chat templating. We also verified its vocabulary and special token mappings. This tokenizer will now be used for all subsequent data processing and model interactions.

1.4 Self-Contained Dataset Classes

Now we define PyTorch `Dataset` classes. These will take file paths to our `.jsonl` data and the *trained tokenizer* to prepare data in the format our model expects.

1.4.1 `DemoCorpusDataset` for Pretraining

```python
class DemoCorpusDataset(Dataset):
    """Dataset for pretraining. Loads text, tokenizes, and prepares X, 
Y pairs."""
    def __init__(self, file_path: str, tokenizer, max_length: int):
        self.tokenizer = tokenizer
        self.max_length = max_length
        self.samples = []
        logger(f"Loading pretraining data from: {file_path}")
        with open(file_path, 'r', encoding='utf-8') as f:
            for line in f:
                self.samples.append(json.loads(line.strip())['text'])
        logger(f"Loaded {len(self.samples)} samples for pretraining.")

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

    def __getitem__(self, idx):
        text = self.samples[idx]

        # For pretraining, we typically add BOS and EOS if not 
implicitly handled by tokenizer during training.
        # Here, we ensure BOS is prepended. EOS will be part of the 
sequence if it fits.
        full_text_with_bos = self.tokenizer.bos_token + text

        encoding = self.tokenizer(
            full_text_with_bos,
            max_length=self.max_length,
            padding="max_length",
            truncation=True,
            return_tensors='pt'
        )
        input_ids = encoding.input_ids.squeeze(0) # (max_length)

        # Create loss mask: 1 for non-pad tokens, 0 for pad tokens.
        # Loss is calculated on Y (shifted input_ids), so the mask 
should align with Y.
        effective_loss_mask = (input_ids !=
self.tokenizer.pad_token_id).long()

        X = input_ids[:-1]  # (max_length - 1)
        Y = input_ids[1:]   # (max_length - 1)
        mask_for_loss_calculation = effective_loss_mask[1:] # Align
with Y

        return X, Y, mask_for_loss_calculation

logger("Testing DemoCorpusDataset...")
try:
    test_pretrain_ds = DemoCorpusDataset(pretrain_file_path,
tokenizer, DEMO_MAX_SEQ_LEN)
    X_pt_sample, Y_pt_sample, mask_pt_sample = test_pretrain_ds[0]
    logger(f"Sample X (Pretrain): {X_pt_sample.shape},
{X_pt_sample[:10]}...")
    logger(f"Sample Y (Pretrain): {Y_pt_sample.shape},
{Y_pt_sample[:10]}...")
    logger(f"Sample Mask (Pretrain): {mask_pt_sample.shape},
{mask_pt_sample[:10]}...")
    logger(f"Decoded X with BOS:
{tokenizer.decode(torch.cat([torch.tensor([tokenizer.bos_token_id]),
X_pt_sample[:torch.sum(mask_pt_sample)]]))}")
    logger(f"Decoded Y:
{tokenizer.decode(Y_pt_sample[:torch.sum(mask_pt_sample)])}")
except Exception as e:
    logger(f"Error loading trained tokenizer with AutoTokenizer: {e}")
    logger("Falling back to using the HFTokenizer object directly
(chat_template might not work as expected).")

    # Create a wrapper class that makes the tokenizer callable
    class CallableTokenizerWrapper:
        def __init__(self, base_tokenizer):
            self.tokenizer = base_tokenizer
            # Set token IDs
            self.pad_token_id = self.tokenizer.token_to_id("<pad>") if
self.tokenizer.token_to_id("<pad>") is not None else
self.tokenizer.token_to_id("<|endoftext|>")
            self.eos_token_id = self.tokenizer.token_to_id("<|
endoftext|>")
            self.bos_token_id = self.tokenizer.token_to_id("<|
im_start|>")

            # Set string attributes
            self.bos_token = "<|im_start|>"
            self.eos_token = "<|endoftext|>"
            self.pad_token = "<pad>" if
self.tokenizer.token_to_id("<pad>") is not None else "<|endoftext|>"
            self.unk_token = "<unk>"
            self.vocab_size = self.tokenizer.get_vocab_size()

        def __call__(self, text, max_length=None, padding=None,
truncation=None, return_tensors=None):
            # Implement basic functionality of AutoTokenizer.__call__
            if isinstance(text, list):
                encodings = [self.tokenizer.encode(t) for t in text]
            else:
                encodings = [self.tokenizer.encode(text)]

            # Convert encodings to lists of IDs if they aren't already
            token_ids = []
            for enc in encodings:
                if hasattr(enc, 'ids'):
                    token_ids.append(enc.ids)
                else:
                    # If encode returns a list directly, use it as is
                    token_ids.append(enc)

            # Handle max_length and padding
            if max_length is not None:
                if truncation:
                    token_ids = [ids[:max_length] for ids in
token_ids]
                if padding == "max_length":
                    token_ids = [ids + [self.pad_token_id] *
(max_length - len(ids)) if len(ids) < max_length else ids[:max_length]
for ids in token_ids]

            # Convert to tensors if requested
            if return_tensors == 'pt':
                import torch
                input_ids = torch.tensor(token_ids)

                # Create a proper TokenizerOutput class with a to()
method
                class TokenizerOutput:
                    def __init__(self, input_ids):
                        self.input_ids = input_ids

                    def to(self, device):
                        self.input_ids = self.input_ids.to(device)
                        return self

                return TokenizerOutput(input_ids)

            return token_ids
        def apply_chat_template(self, conversations, tokenize=True,
add_generation_prompt=False, return_tensors=None, max_length=None,
truncation=None, padding=None):
            """Applies a chat template to format conversation
messages."""
            # Define chat template similar to what was used in the
original tokenizer
            formatted_text = ""
            for message in conversations:
                formatted_text += f"<|im_start|>{message['role']}\
n{message['content']}<|im_end|>\n"

            # Add generation prompt if requested
            if add_generation_prompt:
                formatted_text += "<|im_start|>assistant\n"

            # Return the string if tokenize=False
            if not tokenize:
                return formatted_text

            # Otherwise tokenize and return tensor
            input_ids = self(
                formatted_text,
                max_length=max_length,
                padding=padding,
                truncation=truncation,
                return_tensors=return_tensors
            ).input_ids

            return input_ids
        def encode(self, text, add_special_tokens=True):
            return self.tokenizer.encode(text).ids

        def decode(self, token_ids, skip_special_tokens=False):
            if hasattr(token_ids, 'tolist'):  # If it's a tensor
                token_ids = token_ids.tolist()
            return self.tokenizer.decode(token_ids)

        def get_vocab_size(self):
            return self.tokenizer.get_vocab_size()

        def token_to_id(self, token):
            return self.tokenizer.token_to_id(token)

    # Wrap the tokenizer with our callable wrapper
    tokenizer = CallableTokenizerWrapper(trained_hf_tokenizer)
    DEMO_VOCAB_SIZE_FINAL = tokenizer.vocab_size

    logger(f"Fallback Tokenizer - Vocab Size: {tokenizer.vocab_size}")
    logger(f"Fallback Tokenizer - PAD token: '{tokenizer.pad_token}',
ID: {tokenizer.pad_token_id}")
    logger(f"Fallback Tokenizer - EOS token: '{tokenizer.eos_token}',
ID: {tokenizer.eos_token_id}")
    logger(f"Fallback Tokenizer - BOS token: '{tokenizer.bos_token}',
ID: {tokenizer.bos_token_id}")

[LOG]: Testing DemoCorpusDataset...
[LOG]: Loading pretraining data from: ./dataset_notebook_scratch\
pretrain_data.jsonl
[LOG]: Loaded 7 samples for pretraining.
[LOG]: Error loading trained tokenizer with AutoTokenizer:
'tokenizers.Tokenizer' object has no attribute 'bos_token'
[LOG]: Falling back to using the HFTokenizer object directly
(chat_template might not work as expected).
[LOG]: Fallback Tokenizer - Vocab Size: 435
[LOG]: Fallback Tokenizer - PAD token: '<pad>', ID: 3
[LOG]: Fallback Tokenizer - EOS token: '<|endoftext|>', ID: 0
[LOG]: Fallback Tokenizer - BOS token: '<|im_start|>', ID: 1
```

1.4.2 `DemoChatDataset` for SFT and Reasoning

This dataset class will handle conversational data. It will use the tokenizer's `apply_chat_template` method to format the input and then generate a loss mask to ensure that the model is only trained to predict the assistant's responses (including any `<think>` or `<answer>` tags within the assistant's turn).

```
class DemoChatDataset(Dataset):
    """Dataset for SFT and Reasoning. Uses chat templates and masks
non-assistant tokens."""
    def __init__(self, file_path: str, tokenizer, max_length: int):
        self.tokenizer = tokenizer
        self.max_length = max_length
        self.samples = []
        logger(f"Loading chat data from: {file_path}")
        with open(file_path, 'r', encoding='utf-8') as f:
            for line in f:
                self.samples.append(json.loads(line.strip())
['conversations'])
        logger(f"Loaded {len(self.samples)} chat samples.")

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

    def __getitem__(self, idx):
        conversations = self.samples[idx]

        # Tokenize the full conversation using the chat template
        # add_generation_prompt=False because the assistant's full
```

```
response is in the data
        input_ids = self.tokenizer.apply_chat_template(
            conversations,
            tokenize=True,
            add_generation_prompt=False,
            return_tensors="pt",
            max_length=self.max_length,
            truncation=True,
            padding="max_length"
        ).squeeze(0)

        # Create loss mask: only train on assistant's tokens
        loss_mask = torch.zeros_like(input_ids, dtype=torch.long)

        # This is a simplified approach to find assistant tokens.
        # A robust solution would parse based on special role tokens
during/after tokenization.
        # For this demo, we'll iterate turns and mark assistant tokens
if the template is consistent.
        # The tokenizer.apply_chat_template output helps, but
identifying exact assistant spans requires care.

        # Re-tokenize each part to find spans. This is not most
efficient but illustrative.
        current_token_idx = 0
        is_assistant_turn = False
        assistant_start_token_id = self.tokenizer.encode("<|im_start|
>assistant", add_special_tokens=False)[-1] # often includes role token
        user_start_token_id = self.tokenizer.encode("<|im_start|
>user", add_special_tokens=False)[-1]
        # im_end_token_id = self.tokenizer.eos_token_id # or specific
<|im_end|> id
        im_end_token_id = self.tokenizer.token_to_id("<|im_end|>")

        # More reliable: iterate token IDs to find assistant segments
        assistant_start_seq =
self.tokenizer.encode(self.tokenizer.apply_chat_template([{'role':'ass
istant', 'content':''}], add_generation_prompt=True,
tokenize=False).replace('\n',''), add_special_tokens=False)[:-1] #
remove placeholder token
        # The above is a bit hacky. The MiniMind SFTDataset uses
direct token ID matching for robust mask generation.
        # For this demo, we'll assume any token *after* an assistant
prompt and *before* the next user prompt or EOS is trainable.
        # This logic from the original dataset.py is more robust:
        bos_assistant_ids = self.tokenizer.encode("<|im_start|
>assistant\n", add_special_tokens=False)
        eos_ids = self.tokenizer.encode("<|im_end|>",
add_special_tokens=False)
```

```
        i = 0
        input_ids_list = input_ids.tolist()
        while i < len(input_ids_list):
            # Check for assistant start sequence
            if i + len(bos_assistant_ids) <= len(input_ids_list) and \
               input_ids_list[i : i + len(bos_assistant_ids)] ==
bos_assistant_ids:
                # Found assistant start
                start_of_response = i + len(bos_assistant_ids)
                # Find corresponding EOS
                end_of_response_marker = -1
                j = start_of_response
                while j < len(input_ids_list):
                    if j + len(eos_ids) <= len(input_ids_list) and \
                       input_ids_list[j : j + len(eos_ids)] ==
eos_ids:
                        end_of_response_marker = j
                        break
                    j += 1

                if end_of_response_marker != -1:
                    # Mark tokens from start of response up to and
including the EOS for loss
                    loss_mask[start_of_response :
end_of_response_marker + len(eos_ids)] = 1
                    i = end_of_response_marker + len(eos_ids) # Move
past this assistant block
                    continue
                else: # No EOS found, mask till end (might be
truncated)
                    loss_mask[start_of_response:] = 1
                    break
            i += 1

        loss_mask[input_ids == self.tokenizer.pad_token_id] = 0 #
Don't learn on padding

        X = input_ids[:-1]
        Y = input_ids[1:]
        mask_for_loss_calculation = loss_mask[1:]

        return X, Y, mask_for_loss_calculation

logger("Testing DemoChatDataset for SFT...")
try:
    test_sft_ds = DemoChatDataset(sft_file_path, tokenizer,
DEMO_MAX_SEQ_LEN)
    X_sft_sample, Y_sft_sample, mask_sft_sample = test_sft_ds[0]
    logger(f"Sample X (SFT): {X_sft_sample.shape},
{X_sft_sample[:20]}...")
    logger(f"Sample Y (SFT): {Y_sft_sample.shape},
{Y_sft_sample[:20]}...")
    logger(f"Sample Mask (SFT): {mask_sft_sample.shape},
{mask_sft_sample[:20]}...")
    full_sft_ids = torch.cat([X_sft_sample[:1], Y_sft_sample], dim=0)
    logger(f"Decoded SFT sample with mask applied (showing Y tokens
where mask=1):\
n{tokenizer.decode(Y_sft_sample[mask_sft_sample.bool()])}")
    logger(f"Full SFT sample decoded:\
n{tokenizer.decode(full_sft_ids)}")
except Exception as e:
    logger(f"Error testing DemoChatDataset: {e}. Tokenizer or chat
template might need adjustment.")

logger("Testing DemoChatDataset for Reasoning...")
try:
    test_reasoning_ds = DemoChatDataset(reasoning_file_path,
tokenizer, DEMO_MAX_SEQ_LEN)
    X_rsn_sample, Y_rsn_sample, mask_rsn_sample = test_reasoning_ds[0]
    logger(f"Sample X (Reasoning): {X_rsn_sample.shape},
{X_rsn_sample[:30]}...")
    logger(f"Sample Y (Reasoning): {Y_rsn_sample.shape},
{Y_rsn_sample[:30]}...")
    logger(f"Sample Mask (Reasoning): {mask_rsn_sample.shape},
{mask_rsn_sample[:30]}...")
    full_rsn_ids = torch.cat([X_rsn_sample[:1], Y_rsn_sample], dim=0)
    logger(f"Decoded Reasoning sample with mask applied (showing Y
tokens where mask=1):\
n{tokenizer.decode(Y_rsn_sample[mask_rsn_sample.bool()])}")
    logger(f"Full Reasoning sample decoded:\
n{tokenizer.decode(full_rsn_ids)}")
except Exception as e:
    logger(f"Error testing DemoChatDataset for Reasoning: {e}. Ensure
tokenizer and chat template are correctly set.")
```

```
[LOG]: Testing DemoChatDataset for SFT...
[LOG]: Loading chat data from: ./dataset_notebook_scratch\
sft_data.jsonl
[LOG]: Loaded 3 chat samples.
[LOG]: Sample X (SFT): torch.Size([1023]), tensor([  1, 364, 202, 432,
15, 224,  75,  82,  90, 265,  85,  72, 377,  34,
          2, 202,   1, 434, 202,  44])...
[LOG]: Sample Y (SFT): torch.Size([1023]), tensor([364, 202, 432,  15,
224,  75,  82,  90, 265,  85,  72, 377,  34,   2,
        202,   1, 434, 202,  44, 265])...
[LOG]: Sample Mask (SFT): torch.Size([1023]), tensor([0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1])...
[LOG]: Decoded SFT sample with mask applied (showing Y tokens where
mask=1):
```

```
I am doing well, thank you! How can I help you today?
[LOG]: Full SFT sample decoded:
user
Hello, how are you?
assistant
I am doing well, thank you! How can I help you today?

[LOG]: Testing DemoChatDataset for Reasoning...
[LOG]: Loading chat data from: ./dataset_notebook_scratch\
reasoning_data.jsonl
[LOG]: Loaded 2 chat samples.
[LOG]: Sample X (Reasoning): torch.Size([1023]), tensor([  1, 364,
202,  44,  73, 224,  44, 224,  75,  68,  89,  72, 224,  22,
        265,  83,  83,  79,  72,  86, 265,  81,  71, 224,  72, 268,
224,  20,
         15, 224])...
[LOG]: Sample Y (Reasoning): torch.Size([1023]), tensor([364, 202,
44,  73, 224,  44, 224,  75,  68,  89,  72, 224,  22, 265,
         83,  83,  79,  72,  86, 265,  81,  71, 224,  72, 268, 224,
20,  15,
        224,  75])...
[LOG]: Sample Mask (Reasoning): torch.Size([1023]), tensor([0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0])...
[LOG]: Decoded Reasoning sample with mask applied (showing Y tokens
where mask=1):
<think>The user starts with 3 apples. The user eats 1 apple. This
means 1 apple is subtracted from the initial amount. So, 3 - 1 =
2.</think><answer>You have 2 apples left.</answer>
[LOG]: Full Reasoning sample decoded:
user
If I have 3 apples and eat 1, how many are left?
assistant
<think>The user starts with 3 apples. The user eats 1 apple. This
means 1 apple is subtracted from the initial amount. So, 3 - 1 =
2.</think><answer>You have 2 apples left.</answer>
```

What we've done: We have prepared three sample datasets (`.jsonl` files) for pretraining, SFT, and reasoning. We also defined two PyTorch `Dataset` classes: `DemoCorpusDataset` for pretraining, and `DemoChatDataset` for both SFT and reasoning (as the core data structure and masking logic for assistant responses are similar). We've tested these dataset classes to see example outputs. The `tokenizer` used here is the one we trained in Part 1.

Part 3: Model Architecture (Self-Contained `DemoLLM`)

We will now implement the Transformer-based LLM architecture from scratch. This includes:

- `DemoLLMConfig`: Configuration class.

- `RMSNorm`: Root Mean Square Layer Normalization.
- `RotaryEmbedding`: Rotary Positional Embeddings (RoPE).
- `Attention`: Multi-head attention (with Grouped Query Attention consideration, though might be simplified to MHA for this demo's core).
- `FeedForward`: MLP with SwiGLU activation.
- `DemoTransformerBlock`: A single layer of the Transformer.
- `DemoLLMModel`: The stack of Transformer blocks.
- `DemoLLMForCausalLM`: The full model with a language modeling head for next-token prediction.

The implementation will be inspired by modern LLM designs but simplified.

3.1 `DemoLLMConfig` Class (Defined earlier in 0.3, re-shown for context)

Theory: The configuration class holds all hyperparameters that define the model's architecture, such as vocabulary size, number of layers, hidden dimension size, number of attention heads, etc. It's good practice to have a dedicated config class.

```
class DemoLLMConfig(PretrainedConfig):
    model_type = "demo_llm" # Generic name

    def __init__(
            self,
            vocab_size: int = DEMO_VOCAB_SIZE_FINAL, # Use the actual
vocab size from trained tokenizer
            hidden_size: int = DEMO_HIDDEN_SIZE,
            intermediate_size: int = DEMO_INTERMEDIATE_SIZE,
            num_hidden_layers: int = DEMO_NUM_LAYERS,
            num_attention_heads: int = DEMO_NUM_ATTENTION_HEADS,
            num_key_value_heads: Optional[int] = DEMO_NUM_KV_HEADS,
            hidden_act: str = "silu",
            max_position_embeddings: int = DEMO_MAX_SEQ_LEN,
            rms_norm_eps: float = 1e-5,
            rope_theta: float = 10000.0,
            bos_token_id: int = 1, # Will be updated from tokenizer if
possible
            eos_token_id: int = 2, # Will be updated from tokenizer if
possible
            pad_token_id: Optional[int] = None, # Will be updated from
tokenizer
            dropout: float = 0.0,
            use_cache: bool = True,
            flash_attn: bool = True,
            **kwargs
    ):
        self.vocab_size = vocab_size
        self.hidden_size = hidden_size
        self.intermediate_size = intermediate_size
        self.num_hidden_layers = num_hidden_layers
        self.num_attention_heads = num_attention_heads
        self.num_key_value_heads = num_key_value_heads if
num_key_value_heads is not None else num_attention_heads
        self.head_dim = self.hidden_size // self.num_attention_heads
        if self.num_attention_heads % self.num_key_value_heads != 0:
             raise ValueError(f"num_attention_heads
({self.num_attention_heads}) must be divisible by num_key_value_heads
({self.num_key_value_heads})")
        self.hidden_act = hidden_act
        self.max_position_embeddings = max_position_embeddings
        self.rms_norm_eps = rms_norm_eps
        self.rope_theta = rope_theta
        self.dropout = dropout
        self.use_cache = use_cache
        self.flash_attn = flash_attn

        # Update BOS/EOS/PAD from tokenizer if available
        if tokenizer is not None and hasattr(tokenizer,
'bos_token_id') and tokenizer.bos_token_id is not None:
            bos_token_id = tokenizer.bos_token_id
        if tokenizer is not None and hasattr(tokenizer,
'eos_token_id') and tokenizer.eos_token_id is not None:
            eos_token_id = tokenizer.eos_token_id
        if tokenizer is not None and hasattr(tokenizer,
'pad_token_id') and tokenizer.pad_token_id is not None:
            pad_token_id = tokenizer.pad_token_id
        else: # Default pad to eos if not defined
            pad_token_id = eos_token_id

        super().__init__(
            bos_token_id=bos_token_id,
            eos_token_id=eos_token_id,
            pad_token_id=pad_token_id,
            **kwargs,
        )
logger("DemoLLMConfig defined.")
```

```
[LOG]: DemoLLMConfig defined.
```

3.2 RMS Normalization

Theory: RMSNorm is a variant of Layer Normalization. Instead of subtracting the mean, it only re-scales the activations by their root mean square. This simplifies the computation and has been found to be effective in Transformer models. Formula: `output = (x / sqrt(mean(x^2) + eps)) * weight`

```
class DemoRMSNorm(nn.Module):
    def __init__(self, dim: int, eps: float = 1e-5):
        super().__init__()
        self.eps = eps
        self.weight = nn.Parameter(torch.ones(dim))

    def _norm(self, x):
        return x * torch.rsqrt(x.pow(2).mean(-1, keepdim=True) +
self.eps)

    def forward(self, x):
        output_dtype = x.dtype
        x = x.to(torch.float32) # Calculate in float32 for stability
        output = self._norm(x)
        return (output * self.weight).to(output_dtype)
logger("DemoRMSNorm defined.")
```

```
[LOG]: DemoRMSNorm defined.
```

3.3 Rotary Positional Embeddings (RoPE)

Theory: RoPE applies rotations to query and key vectors based on their absolute positions. This injects positional information in a relative way, as the rotation applied depends on the token's position, and the dot product between rotated query and key vectors inherently captures relative positional differences. It avoids adding separate positional embedding vectors to the input. Each head dimension `d` is conceptually split into `d/2` pairs. For a position `m` and a pair `i`, a rotation matrix `R_m,i` is applied. `R_m,i = [[cos(m*theta_i), -sin(m*theta_i)], [sin(m*theta_i), cos(m*theta_i)]]` where `theta_i = 10000^(-2i/d)`.

```
class RotaryEmbedding(nn.Module):
    def __init__(self, dim: int, max_seq_len: int, theta: float =
10000.0, device=None):
        super().__init__()
        # freqs: (max_seq_len, dim/2)
        freqs = 1.0 / (theta ** (torch.arange(0, dim, 2,
device=device).float() / dim))
        t = torch.arange(max_seq_len, device=device,
dtype=torch.float32)
        freqs = torch.outer(t, freqs)

        # freqs_cis: (max_seq_len, dim/2) holding complex numbers
cos(m*theta_i) + j*sin(m*theta_i)
        self.register_buffer("freqs_cis",
torch.polar(torch.ones_like(freqs), freqs), persistent=False)
        logger(f"Initialized RotaryEmbedding with dim={dim},
max_seq_len={max_seq_len}")

    def forward(self, xq: torch.Tensor, xk: torch.Tensor, seq_len:
int):
        # xq, xk: (bsz, num_heads, seq_len, head_dim)
        # freqs_cis: (max_seq_len, head_dim/2) -> slice to (seq_len,
head_dim/2)
```

```
        # Reshape xq, xk to (bsz, num_heads, seq_len, head_dim/2, 2)
to treat pairs for complex mul
        xq_r = xq.float().reshape(*xq.shape[:-1], -1, 2)
        xk_r = xk.float().reshape(*xk.shape[:-1], -1, 2)

        # Convert to complex: (bsz, num_heads, seq_len, head_dim/2)
        xq_c = torch.view_as_complex(xq_r)
        xk_c = torch.view_as_complex(xk_r)

        # Slice freqs_cis for the current sequence length
        # freqs_cis_pos: (seq_len, head_dim/2)
        freqs_cis_pos = self.freqs_cis[:seq_len]

        # Reshape freqs_cis for broadcasting: (1, 1, seq_len,
head_dim/2)
        freqs_cis_reshaped = freqs_cis_pos.unsqueeze(0).unsqueeze(0)

        # Apply rotation: q'_c = q_c * freqs_cis_pos
        xq_out_c = xq_c * freqs_cis_reshaped
        xk_out_c = xk_c * freqs_cis_reshaped

        # Convert back to real and reshape: (bsz, num_heads, seq_len,
head_dim)
        xq_out = torch.view_as_real(xq_out_c).flatten(3)
        xk_out = torch.view_as_real(xk_out_c).flatten(3)

        return xq_out.type_as(xq), xk_out.type_as(xk)

logger("RotaryEmbedding defined.")
```

```
[LOG]: RotaryEmbedding defined.
```

3.4 Attention Mechanism

Theory: Attention allows the model to weigh the importance of different tokens in the input sequence when producing an output for a given token. Multi-Head Attention (MHA) performs attention in parallel across several "heads," allowing the model to focus on different aspects of the input simultaneously.

- **Query (Q), Key (K), Value (V):** Input hidden states are linearly projected to Q, K, V vectors.
- **Scaled Dot-Product Attention:** `Attention(Q, K, V) = softmax( (Q @ K^T) / sqrt(d_k) ) @ V`
- **Grouped Query Attention (GQA):** A variation where multiple Q heads share the same K and V heads to reduce computation and memory for KV cache during inference. `num_key_value_heads` will be less than `num_attention_heads`.
- **Causal Masking:** For decoder-only models, a causal mask is applied to prevent tokens from attending to future tokens.

- **Flash Attention:** An optimized implementation of attention that reduces memory reads/writes, significantly speeding up training and inference (if available and conditions are met).

```
class DemoAttention(nn.Module):
    def __init__(self, config: DemoLLMConfig):
        super().__init__()
        self.config = config
        self.hidden_size = config.hidden_size
        self.num_q_heads = config.num_attention_heads
        self.num_kv_heads = config.num_key_value_heads
        self.num_kv_groups = self.num_q_heads // self.num_kv_heads #
Num Q heads per KV head
        self.head_dim = config.head_dim

        self.q_proj = nn.Linear(self.hidden_size, self.num_q_heads *
self.head_dim, bias=False)
        self.k_proj = nn.Linear(self.hidden_size, self.num_kv_heads *
self.head_dim, bias=False)
        self.v_proj = nn.Linear(self.hidden_size, self.num_kv_heads *
self.head_dim, bias=False)
        self.o_proj = nn.Linear(self.num_q_heads * self.head_dim,
self.hidden_size, bias=False)

        self.rotary_emb = RotaryEmbedding(
            self.head_dim,
            config.max_position_embeddings,
            theta=config.rope_theta,
            device=DEVICE # Initialize on target device
        )
        self.flash_available = hasattr(F,
'scaled_dot_product_attention') and config.flash_attn

    def _repeat_kv(self, x: torch.Tensor, n_rep: int) -> torch.Tensor:
        bs, num_kv_heads, slen, head_dim = x.shape
        if n_rep == 1:
            return x
        return (
            x[:, :, None, :, :]
            .expand(bs, num_kv_heads, n_rep, slen, head_dim)
            .reshape(bs, num_kv_heads * n_rep, slen, head_dim)
        )

    def forward(
        self,
        hidden_states: torch.Tensor, # (bsz, q_len, hidden_size)
        attention_mask: Optional[torch.Tensor] = None, # (bsz, 1,
q_len, kv_len) for additive mask
        position_ids: Optional[torch.LongTensor] = None, # (bsz,
q_len) -> Not directly used if RoPE applied based on seq_len
```

```
        past_key_value: Optional[Tuple[torch.Tensor, torch.Tensor]] =
None,
        use_cache: bool = False,
    ) -> Tuple[torch.Tensor, Optional[Tuple[torch.Tensor,
torch.Tensor]]]:
        bsz, q_len, _ = hidden_states.shape

        query_states = self.q_proj(hidden_states).view(bsz, q_len,
self.num_q_heads, self.head_dim).transpose(1, 2)
        key_states = self.k_proj(hidden_states).view(bsz, q_len,
self.num_kv_heads, self.head_dim).transpose(1, 2)
        value_states = self.v_proj(hidden_states).view(bsz, q_len,
self.num_kv_heads, self.head_dim).transpose(1, 2)
        # query_states: (bsz, num_q_heads, q_len, head_dim)
        # key_states/value_states: (bsz, num_kv_heads, q_len,
head_dim)

        kv_seq_len = q_len # Initially, before considering
past_key_value
        if past_key_value is not None:
            kv_seq_len += past_key_value[0].shape[2] # Add past length

        # Apply RoPE based on current q_len (for new tokens)
        # The RotaryEmbedding's forward method expects current seq_len
of Q and K
        cos, sin = None, None # Not passing these directly, RoPE is
self-contained
        query_states, key_states = self.rotary_emb(query_states,
key_states, seq_len=q_len) # RoPE applied to current q_len

        if past_key_value is not None:
            # key_states/value_states are for current q_len
            # past_key_value[0] is (bsz, num_kv_heads, past_seq_len,
head_dim)
            key_states = torch.cat([past_key_value[0], key_states],
dim=2)
            value_states = torch.cat([past_key_value[1],
value_states], dim=2)

        if use_cache:
            current_key_value = (key_states, value_states)
        else:
            current_key_value = None

        # Grouped Query Attention: Repeat K and V heads to match Q
heads
        key_states = self._repeat_kv(key_states, self.num_kv_groups)
        value_states = self._repeat_kv(value_states,
self.num_kv_groups)
        # Now key_states/value_states are (bsz, num_q_heads,
kv_seq_len, head_dim)

        attn_output = None
        # Check for Flash Attention compatibility
        # Flash Attn is_causal works when q_len == kv_seq_len for the
attention computation itself.
        # If past_kv is used, q_len for query_states is for new
tokens, kv_seq_len for key_states is total length.
        # Flash Attn handles this by taking full K/V and only new Qs.
        # The `is_causal` flag in F.sdpa handles masking correctly for
decoder style models.
        # The main condition for Flash Attn is no explicit additive
attention_mask.
        can_use_flash = self.flash_available and attention_mask is
None
        if can_use_flash:
            attn_output = F.scaled_dot_product_attention(
                query_states, key_states, value_states,
                attn_mask=None, # Causal mask handled by is_causal
                dropout_p=self.config.dropout if self.training else
0.0,
                is_causal= (q_len == kv_seq_len) # Only truly causal
if no KV cache or if generating first token
                                                # If q_len <
kv_seq_len (due to KV cache), is_causal should be False
                                                # and an explicit mask
would be needed for padding if any.
                                                # For simplicity in
decoder generation where new_q_len = 1, is_causal=False is fine.
                                                # And for training
where q_len = kv_seq_len, is_causal=True.
                                                # Let's make it always
causal for decoder, assuming no padding mask for flash path
            )
        else:
            # Manual attention with causal mask
            attn_weights = torch.matmul(query_states,
key_states.transpose(2, 3)) / math.sqrt(self.head_dim)

            if kv_seq_len > 0: # Avoid mask creation for empty
sequences
                # Causal mask (triangle mask)
                # query_states: (bsz, num_q_heads, q_len, head_dim)
                # key_states:   (bsz, num_q_heads, kv_seq_len,
head_dim)
                # attn_weights: (bsz, num_q_heads, q_len, kv_seq_len)
                mask = torch.full((q_len, kv_seq_len), float("-inf"),
device=query_states.device)
                # For causal, target token j can only attend to source
```

```
tokens i <= j + (kv_seq_len - q_len)
                # where (kv_seq_len - q_len) is the length of the past
context.
                # If q_len == kv_seq_len (no cache), it's a standard
upper triangle.
                # If q_len == 1 (generation with cache), it attends to
all kv_seq_len.
                causal_shift = kv_seq_len - q_len
                mask = torch.triu(mask, diagonal=1 + causal_shift) #
Corrected causal mask
                attn_weights = attn_weights + mask[None, None, :, :] #
Add to scores

            if attention_mask is not None: # Additive padding mask
                attn_weights = attn_weights + attention_mask

            attn_weights = F.softmax(attn_weights, dim=-1,
dtype=torch.float32).type_as(query_states)
            if self.config.dropout > 0.0:
                attn_weights = F.dropout(attn_weights,
p=self.config.dropout, training=self.training)
            attn_output = torch.matmul(attn_weights, value_states)

        attn_output = attn_output.transpose(1,
2).contiguous().view(bsz, q_len, -1)
        attn_output = self.o_proj(attn_output)
        return attn_output, current_key_value
logger("DemoAttention defined.")
```

```
[LOG]: DemoAttention defined.
```

3.5 FeedForward Network (MLP)

Theory: The FeedForward Network (FFN) is applied independently to each token position. It typically consists of two linear transformations with a non-linear activation function in between. Modern LLMs often use SwiGLU, which involves three linear layers and a SiLU (Sigmoid Linear Unit) activation. SwiGLU variant: `down_proj( silu(gate_proj(x)) * up_proj(x) )`

```
class DemoFeedForward(nn.Module):
    def __init__(self, config: DemoLLMConfig):
        super().__init__()
        self.gate_proj = nn.Linear(config.hidden_size,
config.intermediate_size, bias=False)
        self.up_proj = nn.Linear(config.hidden_size,
config.intermediate_size, bias=False)
        self.down_proj = nn.Linear(config.intermediate_size,
config.hidden_size, bias=False)
        self.act_fn = ACT2FN[config.hidden_act] # e.g., SiLU
        self.dropout = nn.Dropout(config.dropout)
    def forward(self, x):
        # This is the SwiGLU formulation: FFN_SwiGLU(x, W, V, W2) =
(Swish_1(xW) * xV)W2
        # Swish_1(x) = x * sigmoid(beta*x), where beta is often 1
(SiLU)
        # Here, gate_proj is W, up_proj is V, down_proj is W2
        return
self.dropout(self.down_proj(self.act_fn(self.gate_proj(x)) *
self.up_proj(x)))
logger("DemoFeedForward defined.")
```

```
[LOG]: DemoFeedForward defined.
```

3.6 Transformer Block (`DemoTransformerBlock`)

Theory: A Transformer block typically consists of a multi-head self-attention layer followed by a feed-forward network. Layer normalization is applied before each of these sub-layers (Pre-LN), and residual connections are used around each sub-layer. Block Structure:

1. `x_norm1 = RMSNorm(x)`
2. `attn_out = SelfAttention(x_norm1)`
3. `x = x + attn_out` (Residual 1)
4. `x_norm2 = RMSNorm(x)`
5. `ffn_out = FeedForward(x_norm2)`
6. `x = x + ffn_out` (Residual 2)

```
class DemoTransformerBlock(nn.Module):
    def __init__(self, config: DemoLLMConfig):
        super().__init__()
        self.self_attn = DemoAttention(config)
        self.mlp = DemoFeedForward(config)
        self.input_layernorm = DemoRMSNorm(config.hidden_size,
eps=config.rms_norm_eps)
        self.post_attention_layernorm =
DemoRMSNorm(config.hidden_size, eps=config.rms_norm_eps)

    def forward(
        self,
        hidden_states: torch.Tensor,
        attention_mask: Optional[torch.Tensor] = None,
        position_ids: Optional[torch.LongTensor] = None, # Passed to
attention for RoPE
        past_key_value: Optional[Tuple[torch.Tensor, torch.Tensor]] =
None,
        use_cache: bool = False,
    ) -> Tuple[torch.Tensor, Optional[Tuple[torch.Tensor,
torch.Tensor]]]:
        residual = hidden_states
        normed_hidden_states = self.input_layernorm(hidden_states)

        attn_outputs, present_key_value = self.self_attn(
            normed_hidden_states,
            attention_mask=attention_mask,
            position_ids=position_ids, # RoPE handled inside
DemoAttention using its internal RotaryEmbedding
            past_key_value=past_key_value,
            use_cache=use_cache
        )
        hidden_states = residual + attn_outputs

        residual = hidden_states
        normed_hidden_states =
self.post_attention_layernorm(hidden_states)
        feed_forward_hidden_states = self.mlp(normed_hidden_states)
        hidden_states = residual + feed_forward_hidden_states

        return hidden_states, present_key_value
logger("DemoTransformerBlock defined.")
```

```
[LOG]: DemoTransformerBlock defined.
```

3.7 Main Model (`DemoLLMModel` - stack of blocks)

Theory: The main LLM model consists of an initial token embedding layer, followed by a stack of Transformer blocks, and a final normalization layer.

```
class DemoLLMModel(PreTrainedModel):
    config_class = DemoLLMConfig

    def __init__(self, config: DemoLLMConfig):
        super().__init__(config)
        self.config = config
        self.padding_idx = config.pad_token_id
        self.vocab_size = config.vocab_size

        self.embed_tokens = nn.Embedding(config.vocab_size,
config.hidden_size, self.padding_idx)
        self.layers = nn.ModuleList([DemoTransformerBlock(config) for
_ in range(config.num_hidden_layers)])
        self.norm = DemoRMSNorm(config.hidden_size,
eps=config.rms_norm_eps)
        self.dropout = nn.Dropout(config.dropout) # Added dropout
after embeddings
        self.gradient_checkpointing = False # For simplicity

    def forward(
        self,
        input_ids: torch.LongTensor = None,
        attention_mask: Optional[torch.Tensor] = None, # (bsz,
seq_len)
        position_ids: Optional[torch.LongTensor] = None,
        past_key_values: Optional[List[torch.FloatTensor]] = None,
        inputs_embeds: Optional[torch.FloatTensor] = None,
        use_cache: Optional[bool] = None,
        output_attentions: Optional[bool] = None, # Not implemented
        output_hidden_states: Optional[bool] = None, # Not implemented
        return_dict: Optional[bool] = None,
    ) -> Union[Tuple, CausalLMOutputWithPast]: # Adjusted return type
        use_cache = use_cache if use_cache is not None else
self.config.use_cache
        return_dict = return_dict if return_dict is not None else
self.config.use_return_dict

        if input_ids is not None and inputs_embeds is not None:
            raise ValueError("You cannot specify both input_ids and
inputs_embeds at the same time")
        elif input_ids is not None:
            batch_size, seq_length = input_ids.shape
        elif inputs_embeds is not None:
            batch_size, seq_length, _ = inputs_embeds.shape
        else:
            raise ValueError("You have to specify either input_ids or
inputs_embeds")

        past_key_values_length = 0
        if past_key_values is not None:
            past_key_values_length = past_key_values[0][0].shape[2] #
(bsz, num_kv_heads, seq_len, head_dim)

        if position_ids is None:
            device = input_ids.device if input_ids is not None else
inputs_embeds.device
            position_ids = torch.arange(
                past_key_values_length, seq_length +
past_key_values_length, dtype=torch.long, device=device
            )
            position_ids = position_ids.unsqueeze(0).view(-1,
seq_length)

        if inputs_embeds is None:
            inputs_embeds = self.embed_tokens(input_ids)

        hidden_states = self.dropout(inputs_embeds)

        # Create attention mask for padding (if any) and causality
        # For a decoder, the mask should be causal and also respect
```

```
padding tokens.
        # The shape for additive mask in Attention is (bsz, 1, q_len,
kv_len)
        # `attention_mask` from input is usually (bsz, seq_len)
        _expanded_mask = None
        if attention_mask is not None:
            # Expand padding mask: (bsz, seq_len) -> (bsz, 1, q_len,
kv_len_with_past)
            # This can get tricky with KV caching. For this simplified
version,
            # we assume attention_mask applies to current inputs.
            # Causal part is handled in DemoAttention.
            # An additive mask for padding would be:
            expanded_padding_mask = attention_mask[:, None,
None, :].expand(batch_size, 1, seq_length, seq_length +
past_key_values_length)
            _expanded_mask = torch.zeros_like(expanded_padding_mask,
dtype=hidden_states.dtype)
            _expanded_mask.masked_fill_(expanded_padding_mask == 0,
float("-inf"))

        next_decoder_cache = [] if use_cache else None

        for i, decoder_layer in enumerate(self.layers):
            past_kv = past_key_values[i] if past_key_values is not
None else None

            layer_outputs = decoder_layer(
                hidden_states,
                attention_mask=_expanded_mask, # Pass the combined
mask
                position_ids=position_ids, # RoPE will use this
implicitly or via seq_len
                past_key_value=past_kv,
                use_cache=use_cache,
            )
            hidden_states = layer_outputs[0]
            if use_cache:
                next_decoder_cache.append(layer_outputs[1])

        hidden_states = self.norm(hidden_states)

        # This model doesn't have MoE, so no aux_loss
        if not return_dict:
            return tuple(v for v in [hidden_states,
next_decoder_cache] if v is not None)

        return BaseModelOutputWithPast(
            last_hidden_state=hidden_states,
            past_key_values=next_decoder_cache,
            hidden_states=None,
            attentions=None,
        )
logger("DemoLLMModel defined.")
```

```
[LOG]: DemoLLMModel defined.
```

3.8 Causal LM Head Model (`DemoLLMForCausalLM`)

Theory: This class wraps the `DemoLLMModel` and adds a final linear layer (the Language Modeling head) on top of the Transformer block outputs. This head projects the hidden states to the vocabulary size, producing logits for each token in the vocabulary. It also handles the calculation of the loss for Causal Language Modeling if labels are provided.

```
class DemoLLMForCausalLM(PreTrainedModel):
    config_class = DemoLLMConfig

    def __init__(self, config: DemoLLMConfig):
        super().__init__(config)
        self.model = DemoLLMModel(config)
        self.lm_head = nn.Linear(config.hidden_size,
config.vocab_size, bias=False)
        # Weight tying is a common practice but optional
        # self.model.embed_tokens.weight = self.lm_head.weight
        self.post_init() # Initialize weights

    def get_input_embeddings(self):
        return self.model.embed_tokens

    def set_input_embeddings(self, value):
        self.model.embed_tokens = value

    def get_output_embeddings(self):
        return self.lm_head

    def set_output_embeddings(self, new_embeddings):
        self.lm_head = new_embeddings

    def prepare_inputs_for_generation(self, input_ids,
past_key_values=None, attention_mask=None, **kwargs):
        # if model is used as a decoder in encoder-decoder model, the
decoder attention mask is created on the fly
        if past_key_values:
            input_ids = input_ids[:, -1:] # Only take the last token
if past_key_values is not None

        position_ids = kwargs.get("position_ids", None)
        if attention_mask is not None and position_ids is None:
            # create position_ids on the fly for batch generation
            position_ids = attention_mask.long().cumsum(-1) - 1
            position_ids.masked_fill_(attention_mask == 0, 1)
            if past_key_values:
                position_ids = position_ids[:, -1].unsqueeze(-1)

        return {
            "input_ids": input_ids,
            "past_key_values": past_key_values,
            "use_cache": kwargs.get("use_cache"),
            "attention_mask": attention_mask,
            "position_ids": position_ids,
        }

    def forward(
        self,
        input_ids: Optional[torch.LongTensor] = None,
        attention_mask: Optional[torch.Tensor] = None,
        position_ids: Optional[torch.LongTensor] = None,
        past_key_values: Optional[List[torch.FloatTensor]] = None,
        inputs_embeds: Optional[torch.FloatTensor] = None,
        labels: Optional[torch.LongTensor] = None,
        use_cache: Optional[bool] = None,
        output_attentions: Optional[bool] = None,
        output_hidden_states: Optional[bool] = None,
        return_dict: Optional[bool] = None,
    ) -> Union[Tuple, CausalLMOutputWithPast]:
        return_dict = return_dict if return_dict is not None else
self.config.use_return_dict

        outputs = self.model(
            input_ids=input_ids,
            attention_mask=attention_mask,
            position_ids=position_ids,
            past_key_values=past_key_values,
            inputs_embeds=inputs_embeds,
            use_cache=use_cache,
            output_attentions=output_attentions,
            output_hidden_states=output_hidden_states,
            return_dict=True, # Internal call to base model should
always return dict
        )

        hidden_states = outputs.last_hidden_state
        logits = self.lm_head(hidden_states)

        loss = None
        if labels is not None:
            # Shift so that tokens < n predict n
            shift_logits = logits[..., :-1, :].contiguous()
            shift_labels = labels[..., 1:].contiguous()
            # Flatten the tokens
            loss_fct = nn.CrossEntropyLoss() # Default ignore_index is
-100, set if needed
            shift_logits = shift_logits.view(-1,
self.config.vocab_size)
            shift_labels = shift_labels.view(-1)
            # Ensure labels are on the same device as logits for loss
calculation
            shift_labels = shift_labels.to(shift_logits.device)
            loss = loss_fct(shift_logits, shift_labels)

        if not return_dict:
            output = (logits,) + (outputs.past_key_values if use_cache
else tuple()) # Keep it simple
            return (loss,) + output if loss is not None else output

        return CausalLMOutputWithPast(
            loss=loss,
            logits=logits,
            past_key_values=outputs.past_key_values,
            hidden_states=outputs.hidden_states,
            attentions=outputs.attentions,
        )
logger("DemoLLMForCausalLM defined.")
```

```
[LOG]: DemoLLMForCausalLM defined.
```

3.9 Verify Model Initialization

```
if tokenizer: # Ensure tokenizer is loaded before creating config
dependent on its vocab size
    demo_llm_config = DemoLLMConfig(vocab_size=DEMO_VOCAB_SIZE_FINAL)
# Use the final vocab size
    demo_llm_instance = DemoLLMForCausalLM(demo_llm_config).to(DEVICE)
    print_model_summary(demo_llm_instance, "Initial DemoLLM Instance")
    del demo_llm_instance # Clean up
else:
    logger("Skipping model verification as tokenizer was not loaded.")
```

```
DemoLLMForCausalLM has generative capabilities, as
`prepare_inputs_for_generation` is explicitly overwritten. However, it
doesn't directly inherit from `GenerationMixin`. From □v4.50□ onwards,
`PreTrainedModel` will NOT inherit from `GenerationMixin`, and this
model will lose the ability to call `generate` and other related
functions.
  - If you're using `trust_remote_code=True`, you can get rid of this
warning by loading the model with an auto class. See
https://huggingface.co/docs/transformers/en/model_doc/auto#auto-
classes
```

```
  - If you are the owner of the model architecture code, please modify
your model class such that it inherits from `GenerationMixin` (after
`PreTrainedModel`, otherwise you'll get an exception).
  - If you are not the owner of the model architecture class, please
contact the model code owner to update it.

[LOG]: Initialized RotaryEmbedding with dim=64, max_seq_len=1024
[LOG]: Initialized RotaryEmbedding with dim=64, max_seq_len=1024
[LOG]: Initialized RotaryEmbedding with dim=64, max_seq_len=1024
[LOG]: Initialized RotaryEmbedding with dim=64, max_seq_len=1024
[LOG]: Initialized RotaryEmbedding with dim=64, max_seq_len=1024
[LOG]: Initialized RotaryEmbedding with dim=64, max_seq_len=1024
[LOG]: Initialized RotaryEmbedding with dim=64, max_seq_len=1024
[LOG]: Initialized RotaryEmbedding with dim=64, max_seq_len=1024
[LOG]: Initialized RotaryEmbedding with dim=64, max_seq_len=1024
[LOG]: Initialized RotaryEmbedding with dim=64, max_seq_len=1024
[LOG]: Initialized RotaryEmbedding with dim=64, max_seq_len=1024
[LOG]: Initialized RotaryEmbedding with dim=64, max_seq_len=1024
[LOG]: Initialized RotaryEmbedding with dim=64, max_seq_len=1024
[LOG]: Initialized RotaryEmbedding with dim=64, max_seq_len=1024
[LOG]: Initialized RotaryEmbedding with dim=64, max_seq_len=1024
[LOG]: Initialized RotaryEmbedding with dim=64, max_seq_len=1024
[LOG]: Initialized RotaryEmbedding with dim=64, max_seq_len=1024
[LOG]: Initialized RotaryEmbedding with dim=64, max_seq_len=1024
[LOG]: Initialized RotaryEmbedding with dim=64, max_seq_len=1024
[LOG]: Initialized RotaryEmbedding with dim=64, max_seq_len=1024
[LOG]: Initialized RotaryEmbedding with dim=64, max_seq_len=1024
[LOG]: Initialized RotaryEmbedding with dim=64, max_seq_len=1024
[LOG]: Initialized RotaryEmbedding with dim=64, max_seq_len=1024
[LOG]: Initialized RotaryEmbedding with dim=64, max_seq_len=1024
[LOG]: --- Initial DemoLLM Instance Summary ---
[LOG]: Configuration: DemoLLMConfig {
  "_attn_implementation_autoset": true,
  "bos_token_id": 1,
  "dropout": 0.0,
  "eos_token_id": 0,
  "flash_attn": true,
  "head_dim": 64,
  "hidden_act": "silu",
  "hidden_size": 1024,
  "intermediate_size": 2752,
  "max_position_embeddings": 1024,
  "model_type": "demo_llm",
  "num_attention_heads": 16,
  "num_hidden_layers": 24,
  "num_key_value_heads": 16,
  "pad_token_id": 3,
  "rms_norm_eps": 1e-05,
  "rope_theta": 10000.0,
  "transformers_version": "4.51.3",
  "use_cache": true,
  "vocab_size": 435
}

[LOG]: Total parameters: 304.058 M (304058368)
[LOG]: Trainable parameters: 304.058 M (304058368)
[LOG]: -------------------------
```

What we've done in Model Architecture: We have defined all the necessary PyTorch modules for a Transformer-based decoder-only Language Model. This includes:

- `DemoLLMConfig`: Holds model hyperparameters.
- `DemoRMSNorm`: For layer normalization.
- `RotaryEmbedding`: Implements RoPE for positional encoding.
- `DemoAttention`: The core self-attention mechanism, including GQA logic and RoPE application.
- `DemoFeedForward`: The MLP layer using SwiGLU.
- `DemoTransformerBlock`: Combines attention and MLP with residual connections and normalization.
- `DemoLLMModel`: Stacks multiple Transformer blocks.
- `DemoLLMForCausalLM`: Adds the final language modeling head for prediction and loss calculation.

This self-contained architecture is now ready for training.

Part 4: Pretraining (Self-Contained `DemoLLM`)

Theory Recap: Pretraining teaches the model fundamental language understanding by having it predict the next token in a sequence on a large corpus. The loss is typically Cross-Entropy Loss calculated over the entire valid (non-padded) sequence.

We will use the `DemoLLMForCausalLM` and `DemoCorpusDataset` defined earlier.

4.1 Initialize Model for Pretraining

```
logger("Initializing model for Pretraining...")
if tokenizer: # Ensure tokenizer is loaded
    pt_config = DemoLLMConfig(
        vocab_size=DEMO_VOCAB_SIZE_FINAL,
        hidden_size=DEMO_HIDDEN_SIZE,
        intermediate_size=DEMO_INTERMEDIATE_SIZE,
        num_hidden_layers=DEMO_NUM_LAYERS,
        num_attention_heads=DEMO_NUM_ATTENTION_HEADS,
        num_key_value_heads=DEMO_NUM_KV_HEADS,
        max_position_embeddings=DEMO_MAX_SEQ_LEN,
        bos_token_id=tokenizer.bos_token_id,
        eos_token_id=tokenizer.eos_token_id,
        pad_token_id=tokenizer.pad_token_id
    )
    pt_model = DemoLLMForCausalLM(pt_config).to(DEVICE)
    print_model_summary(pt_model, "Demo Pretrain Model")
else:
    logger("Cannot initialize pretrain model: tokenizer not
available.")
    pt_model = None
```

```
[LOG]: Initializing model for Pretraining...
[LOG]: Initialized RotaryEmbedding with dim=64, max_seq_len=1024
[LOG]: Initialized RotaryEmbedding with dim=64, max_seq_len=1024
[LOG]: Initialized RotaryEmbedding with dim=64, max_seq_len=1024
[LOG]: Initialized RotaryEmbedding with dim=64, max_seq_len=1024
[LOG]: Initialized RotaryEmbedding with dim=64, max_seq_len=1024
[LOG]: Initialized RotaryEmbedding with dim=64, max_seq_len=1024
[LOG]: Initialized RotaryEmbedding with dim=64, max_seq_len=1024
[LOG]: Initialized RotaryEmbedding with dim=64, max_seq_len=1024
[LOG]: Initialized RotaryEmbedding with dim=64, max_seq_len=1024
[LOG]: Initialized RotaryEmbedding with dim=64, max_seq_len=1024
[LOG]: Initialized RotaryEmbedding with dim=64, max_seq_len=1024
[LOG]: Initialized RotaryEmbedding with dim=64, max_seq_len=1024
[LOG]: Initialized RotaryEmbedding with dim=64, max_seq_len=1024
[LOG]: Initialized RotaryEmbedding with dim=64, max_seq_len=1024
[LOG]: Initialized RotaryEmbedding with dim=64, max_seq_len=1024
[LOG]: Initialized RotaryEmbedding with dim=64, max_seq_len=1024
[LOG]: Initialized RotaryEmbedding with dim=64, max_seq_len=1024
[LOG]: Initialized RotaryEmbedding with dim=64, max_seq_len=1024
[LOG]: Initialized RotaryEmbedding with dim=64, max_seq_len=1024
[LOG]: Initialized RotaryEmbedding with dim=64, max_seq_len=1024
[LOG]: Initialized RotaryEmbedding with dim=64, max_seq_len=1024
[LOG]: Initialized RotaryEmbedding with dim=64, max_seq_len=1024
[LOG]: Initialized RotaryEmbedding with dim=64, max_seq_len=1024
[LOG]: Initialized RotaryEmbedding with dim=64, max_seq_len=1024
[LOG]: --- Demo Pretrain Model Summary ---
[LOG]: Configuration: DemoLLMConfig {
  "_attn_implementation_autoset": true,
  "bos_token_id": 1,
  "dropout": 0.0,
  "eos_token_id": 0,
  "flash_attn": true,
  "head_dim": 64,
  "hidden_act": "silu",
  "hidden_size": 1024,
  "intermediate_size": 2752,
  "max_position_embeddings": 1024,
  "model_type": "demo_llm",
  "num_attention_heads": 16,
  "num_hidden_layers": 24,
  "num_key_value_heads": 16,
  "pad_token_id": 3,
  "rms_norm_eps": 1e-05,
  "rope_theta": 10000.0,
  "transformers_version": "4.51.3",
  "use_cache": true,
  "vocab_size": 435
}

[LOG]: Total parameters: 304.058 M (304058368)
[LOG]: Trainable parameters: 304.058 M (304058368)
[LOG]: -------------------------
```

4.2 Prepare Pretraining DataLoader

```
if tokenizer and pt_model: # Proceed only if tokenizer and model are
initialized
    demo_pt_dataset = DemoCorpusDataset(pretrain_file_path, tokenizer,
max_length=DEMO_MAX_SEQ_LEN)
    demo_pt_dataloader = DataLoader(demo_pt_dataset,
batch_size=DEMO_BATCH_SIZE, shuffle=True, num_workers=0)
    logger(f"Demo Pretrain dataset size: {len(demo_pt_dataset)}")
else:
    logger("Skipping pretrain dataloader: tokenizer or model not
initialized.")
    demo_pt_dataloader = [] # Empty dataloader

[LOG]: Loading pretraining data from: ./dataset_notebook_scratch\
pretrain_data.jsonl
[LOG]: Loaded 7 samples for pretraining.
[LOG]: Demo Pretrain dataset size: 7
```

4.3 Pretraining Loop

```
if pt_model and demo_pt_dataloader: # Check if model and dataloader
are ready
    optimizer_pt_demo = optim.AdamW(pt_model.parameters(),
lr=DEMO_PRETRAIN_LR)
    loss_fct_pt_demo = nn.CrossEntropyLoss(reduction='none')

    # Mixed precision context for GPU
    autocast_ctx = nullcontext() if DEVICE.type == 'cpu' else
torch.amp.autocast(device_type=DEVICE.type, dtype=PTDTYPE)
    scaler_pt_demo = torch.cuda.amp.GradScaler(enabled=(DTYPE_STR !=
'float32' and DEVICE.type == 'cuda'))

    total_steps_pt_demo = len(demo_pt_dataloader) *
DEMO_PRETRAIN_EPOCHS
    logger(f"Starting DEMO Pretraining for {DEMO_PRETRAIN_EPOCHS}
epochs ({total_steps_pt_demo} steps)...")

    pt_model.train()
    current_training_step_pt = 0
    for epoch in range(DEMO_PRETRAIN_EPOCHS):
        epoch_loss_pt_val = 0.0
        for step, (X_batch, Y_batch, mask_batch) in
enumerate(demo_pt_dataloader):
            X_batch, Y_batch, mask_batch = X_batch.to(DEVICE),
Y_batch.to(DEVICE), mask_batch.to(DEVICE)

            current_lr_pt = get_lr(current_training_step_pt,
total_steps_pt_demo, DEMO_PRETRAIN_LR)
            for param_group in optimizer_pt_demo.param_groups:
                param_group['lr'] = current_lr_pt

            with autocast_ctx:
                # For our custom DemoLLMForCausalLM, if `labels` is
not passed, it returns logits.
                # We need to compute loss manually using the mask.
                outputs_pt = pt_model(input_ids=X_batch)
                logits_pt = outputs_pt.logits # (bsz, seq_len-1,
vocab_size)

                # logits_pt are for predicting Y_batch. mask_batch
aligns with Y_batch.
                raw_loss_pt = loss_fct_pt_demo(logits_pt.view(-1,
logits_pt.size(-1)), Y_batch.view(-1))
                masked_loss_pt = (raw_loss_pt * mask_batch.view(-
1)).sum() / mask_batch.sum().clamp(min=1)

            scaler_pt_demo.scale(masked_loss_pt).backward()
            scaler_pt_demo.step(optimizer_pt_demo)
            scaler_pt_demo.update()
            optimizer_pt_demo.zero_grad(set_to_none=True)

            epoch_loss_pt_val += masked_loss_pt.item()
            current_training_step_pt += 1

            if (step + 1) % 1 == 0: # Log frequently for demo
                logger(f"PT Epoch {epoch+1}, Step
{step+1}/{len(demo_pt_dataloader)}, Loss: {masked_loss_pt.item():.4f},
LR: {current_lr_pt:.3e}")

        avg_epoch_loss_pt = epoch_loss_pt_val /
len(demo_pt_dataloader)
        logger(f"End of PT Epoch {epoch+1}, Average Loss:
{avg_epoch_loss_pt:.4f}")

    logger("DEMO Pretraining finished.")
    # Save the final pretrained model
    final_pretrained_model_path = os.path.join(NOTEBOOK_OUT_DIR,
"demo_llm_pretrained.pth")
```

```
    torch.save(pt_model.state_dict(), final_pretrained_model_path)
    logger(f"Demo pretrained model saved to:
{final_pretrained_model_path}")
else:
    logger("Skipping Pretraining loop as model or dataloader was not
initialized.")
    final_pretrained_model_path = None
```

```
[LOG]: Starting DEMO Pretraining for 10 epochs (10 steps)...
```

4.4 Quick Test of Self-Contained Pretrained Model

Let's check if our model learned anything, even if minimal.

```
if pt_model and final_pretrained_model_path and
os.path.exists(final_pretrained_model_path):
    logger("Testing pretrained model generation...")
    pt_model.eval() # Set to evaluation mode
    test_prompt_str_pt = "Language models learn"
    # Prepend BOS for generation consistency with training if
tokenizer doesn't do it automatically
    pt_test_input_ids = tokenizer(tokenizer.bos_token +
test_prompt_str_pt, return_tensors="pt").input_ids.to(DEVICE)

    with torch.no_grad(), autocast_ctx:
        generated_output_pt = pt_model.generate(
            pt_test_input_ids,
            max_new_tokens=15,
            do_sample=False, # Greedy for this test
            eos_token_id=tokenizer.eos_token_id,
            pad_token_id=tokenizer.pad_token_id
        )
    decoded_generated_pt = tokenizer.decode(generated_output_pt[0],
skip_special_tokens=True)
    logger(f"Prompt: '{test_prompt_str_pt}' -> Generated:
'{decoded_generated_pt}'")
else:
    logger("Skipping pretrained model test as it was not trained or
saved.")
```

```
[2025-05-14 15:52:28] Testing pretrained model generation...
[2025-05-14 15:52:28] Prompt: 'Language models learn' -> Generated:
'Language models learn learn learn learn learn learn learn learn learn
learn learn learn learn learn learn learn'
```

What we've done in Pretraining: We initialized our `DemoLLMForCausalLM` with a very small configuration. We then trained it for a few epochs on our tiny sample pretraining dataset. The goal was for the model to learn to predict the next token in a sequence, thereby capturing some basic statistical properties of the language. The generated output will likely be repetitive or nonsensical at this stage due to the extremely limited data and model size, but the process is what matters for this guide.

Part 5: Supervised Fine-Tuning (SFT) (Self-Contained `DemoLLM`)

Theory Recap: SFT adapts the pretrained LLM to follow instructions or engage in conversations. It uses a dataset of input prompts and desired high-quality responses. The key is to train the model to generate the assistant's part of the dialogue, often using a loss mask so that only the assistant's tokens contribute to the loss calculation.

We will load our pretrained `DemoLLMForCausalLM` and fine-tune it using the `DemoChatDataset` with our `sample_sft.jsonl` data.

5.1 Initialize Model for SFT (Load Pretrained)

```
logger("Initializing model for SFT, loading pretrained weights...")
if tokenizer and final_pretrained_model_path and
os.path.exists(final_pretrained_model_path):
    sft_config = DemoLLMConfig( # Re-use the same config as
pretraining
        vocab_size=DEMO_VOCAB_SIZE_FINAL,
        hidden_size=DEMO_HIDDEN_SIZE,
        intermediate_size=DEMO_INTERMEDIATE_SIZE,
        num_hidden_layers=DEMO_NUM_LAYERS,
        num_attention_heads=DEMO_NUM_ATTENTION_HEADS,
        num_key_value_heads=DEMO_NUM_KV_HEADS,
        max_position_embeddings=DEMO_MAX_SEQ_LEN,
        bos_token_id=tokenizer.bos_token_id,
        eos_token_id=tokenizer.eos_token_id,
        pad_token_id=tokenizer.pad_token_id
    )
    sft_model_demo = DemoLLMForCausalLM(sft_config).to(DEVICE)

sft_model_demo.load_state_dict(torch.load(final_pretrained_model_path,
map_location=DEVICE))
    print_model_summary(sft_model_demo, "Demo SFT Model (Loaded
Pretrained)")
else:
    logger("Cannot initialize SFT model: Pretrained model path or
tokenizer is invalid.")
    sft_model_demo = None
```

```
[2025-05-14 15:52:28] Initializing model for SFT, loading pretrained
weights...
[2025-05-14 15:52:28] Initialized RotaryEmbedding with dim=16,
max_seq_len=64
[2025-05-14 15:52:28] Initialized RotaryEmbedding with dim=16,
max_seq_len=64
[2025-05-14 15:52:28] --- Demo SFT Model (Loaded Pretrained) Summary
```

```
---
[2025-05-14 15:52:28] Configuration: DemoLLMConfig {
  "_attn_implementation_autoset": true,
  "bos_token_id": 1,
  "dropout": 0.0,
  "eos_token_id": 0,
  "flash_attn": true,
  "head_dim": 16,
  "hidden_act": "silu",
  "hidden_size": 64,
  "intermediate_size": 192,
  "max_position_embeddings": 64,
  "model_type": "demo_llm",
  "num_attention_heads": 4,
  "num_hidden_layers": 2,
  "num_key_value_heads": 2,
  "pad_token_id": 3,
  "rms_norm_eps": 1e-05,
  "rope_theta": 10000.0,
  "transformers_version": "4.51.3",
  "use_cache": true,
  "vocab_size": 435
}

[2025-05-14 15:52:28] Total parameters: 0.126 M (126464)
[2025-05-14 15:52:28] Trainable parameters: 0.126 M (126464)
[2025-05-14 15:52:28] -------------------------
```

5.2 Prepare SFT DataLoader

```python
if tokenizer and sft_model_demo:
    demo_sft_dataset = DemoChatDataset(sft_file_path, tokenizer,
max_length=DEMO_MAX_SEQ_LEN)
    demo_sft_dataloader = DataLoader(demo_sft_dataset,
batch_size=DEMO_BATCH_SIZE, shuffle=True, num_workers=0)
    logger(f"Demo SFT dataset size: {len(demo_sft_dataset)}")

    logger("Verifying SFT data sample and mask from DataLoader:")
    for x_s_dl, y_s_dl, m_s_dl in demo_sft_dataloader:
        # Reconstruct full sequence for one sample to verify mask
logic
        idx_to_check = 0
        # The first token of X is often BOS. The actual sequence
starts from the second token of input_ids.
        # So, to reconstruct from X, Y: use X's BOS, then Y.
        # Original input_ids was tokenized_chat. X = input_ids[:-1], Y
= input_ids[1:]
        # So, input_ids = torch.cat([X_s_dl[idx_to_check, :1],
Y_s_dl[idx_to_check]], dim=0)
```

```
        # However, the tokenizer already added BOS if specified by
template. Let's just decode Y with its mask.
        logger(f"  Tokens from Y for sample {idx_to_check} where mask
is 1: {tokenizer.decode(y_s_dl[idx_to_check]
[m_s_dl[idx_to_check].bool()])}")
        break
else:
    logger("Skipping SFT Dataloader: SFT model or tokenizer not
initialized.")
    demo_sft_dataloader = []
```

```
[2025-05-14 15:52:28] Loading chat data from:
./dataset_notebook_scratch\sft_data.jsonl
[2025-05-14 15:52:28] Loaded 3 chat samples.
[2025-05-14 15:52:28] Demo SFT dataset size: 3
[2025-05-14 15:52:28] Verifying SFT data sample and mask from
DataLoader:
[2025-05-14 15:52:28]   Tokens from Y for sample 0 where mask is 1:
Gravity is the force that pulls objects toward
```

5.3 SFT Loop

```
if sft_model_demo and demo_sft_dataloader:
    optimizer_sft_d = optim.AdamW(sft_model_demo.parameters(),
lr=DEMO_SFT_LR)
    loss_fct_sft_d = nn.CrossEntropyLoss(reduction='none')

    autocast_ctx_sft = nullcontext() if DEVICE.type == 'cpu' else
torch.amp.autocast(device_type=DEVICE.type, dtype=PTDTYPE)
    scaler_sft_d = torch.cuda.amp.GradScaler(enabled=(DTYPE_STR !=
'float32' and DEVICE.type == 'cuda'))

    total_steps_sft_d = len(demo_sft_dataloader) * DEMO_SFT_EPOCHS
    logger(f"Starting DEMO SFT for {DEMO_SFT_EPOCHS} epochs
({total_steps_sft_d} steps)...")

    sft_model_demo.train()
    current_training_step_sft = 0
    for epoch in range(DEMO_SFT_EPOCHS):
        epoch_loss_sft_val = 0.0
        for step, (X_batch_sft, Y_batch_sft, mask_batch_sft) in
enumerate(demo_sft_dataloader):
            X_batch_sft, Y_batch_sft, mask_batch_sft =
X_batch_sft.to(DEVICE), Y_batch_sft.to(DEVICE),
mask_batch_sft.to(DEVICE)

            current_lr_sft = get_lr(current_training_step_sft,
total_steps_sft_d, DEMO_SFT_LR)
            for param_group in optimizer_sft_d.param_groups:
                param_group['lr'] = current_lr_sft
            with autocast_ctx_sft:
                outputs_sft_loop =
sft_model_demo(input_ids=X_batch_sft)
                logits_sft_loop = outputs_sft_loop.logits # (bsz,
seq_len-1, vocab_size)

                raw_loss_sft = loss_fct_sft_d(logits_sft_loop.view(-1,
logits_sft_loop.size(-1)), Y_batch_sft.view(-1))
                # mask_batch_sft corresponds to Y_batch_sft
                masked_loss_sft = (raw_loss_sft *
mask_batch_sft.view(-1)).sum() / mask_batch_sft.sum().clamp(min=1)

            scaler_sft_d.scale(masked_loss_sft).backward()
            scaler_sft_d.step(optimizer_sft_d)
            scaler_sft_d.update()
            optimizer_sft_d.zero_grad(set_to_none=True)

            epoch_loss_sft_val += masked_loss_sft.item()
            current_training_step_sft += 1

            if (step + 1) % 1 == 0:
                logger(f"SFT Epoch {epoch+1}, Step
{step+1}/{len(demo_sft_dataloader)}, Loss:
{masked_loss_sft.item():.4f}, LR: {current_lr_sft:.3e}")

        logger(f"End of SFT Epoch {epoch+1}, Avg Loss:
{epoch_loss_sft_val / len(demo_sft_dataloader):.4f}")

    logger("DEMO SFT finished.")
    final_sft_model_path = os.path.join(NOTEBOOK_OUT_DIR,
"demo_llm_sft.pth")
    torch.save(sft_model_demo.state_dict(), final_sft_model_path)
    logger(f"Demo SFT model saved to: {final_sft_model_path}")
else:
    logger("Skipping SFT loop as model or dataloader was not
initialized.")
    final_sft_model_path = None
```

```
[2025-05-14 15:52:28] Starting DEMO SFT for 5 epochs (10 steps)...
[2025-05-14 15:52:28] SFT Epoch 1, Step 1/2, Loss: 60.2210, LR:
5.000e-05
[2025-05-14 15:52:28] SFT Epoch 1, Step 2/2, Loss: 55.7710, LR:
4.890e-05
[2025-05-14 15:52:28] End of SFT Epoch 1, Avg Loss: 57.9960
[2025-05-14 15:52:28] SFT Epoch 2, Step 1/2, Loss: 60.0502, LR:
4.570e-05
[2025-05-14 15:52:28] SFT Epoch 2, Step 2/2, Loss: 55.5945, LR:
4.073e-05
[2025-05-14 15:52:28] End of SFT Epoch 2, Avg Loss: 57.8224
```

```
[2025-05-14 15:52:28] SFT Epoch 3, Step 1/2, Loss: 57.3038, LR: 
3.445e-05
[2025-05-14 15:52:28] SFT Epoch 3, Step 2/2, Loss: 59.5190, LR: 
2.750e-05
[2025-05-14 15:52:28] End of SFT Epoch 3, Avg Loss: 58.4114
[2025-05-14 15:52:28] SFT Epoch 4, Step 1/2, Loss: 59.8299, LR: 
2.055e-05
[2025-05-14 15:52:28] SFT Epoch 4, Step 2/2, Loss: 55.3519, LR: 
1.427e-05
[2025-05-14 15:52:28] End of SFT Epoch 4, Avg Loss: 57.5909
[2025-05-14 15:52:28] SFT Epoch 5, Step 1/2, Loss: 57.2238, LR: 
9.297e-06
[2025-05-14 15:52:28] SFT Epoch 5, Step 2/2, Loss: 60.3055, LR: 
6.101e-06
[2025-05-14 15:52:28] End of SFT Epoch 5, Avg Loss: 58.7647
[2025-05-14 15:52:28] DEMO SFT finished.
[2025-05-14 15:52:28] Demo SFT model saved to: ./out_notebook_scratch\
demo_llm_sft.pth
```

5.4 Quick Test of SFT Model

```python
if sft_model_demo and final_sft_model_path and 
os.path.exists(final_sft_model_path):
    logger("Testing SFT model chat capability...")
    sft_model_demo.eval()
    sft_test_chat_history = [{"role": "user", "content": "What is the 
capital of France?"}]
    sft_test_prompt = 
tokenizer.apply_chat_template(sft_test_chat_history, tokenize=False, 
add_generation_prompt=True)
    sft_test_inputs = tokenizer(sft_test_prompt, 
return_tensors="pt").to(DEVICE)

    with torch.no_grad(), autocast_ctx_sft:
        sft_generated_outputs = sft_model_demo.generate(
            sft_test_inputs.input_ids,
            max_new_tokens=200,
            do_sample=True,
            eos_token_id=tokenizer.eos_token_id,
            pad_token_id=tokenizer.pad_token_id
        )
    sft_decoded_response = tokenizer.decode(sft_generated_outputs[0]
[sft_test_inputs.input_ids.shape[1]:], skip_special_tokens=True)
    logger(f"SFT Prompt: '{sft_test_chat_history[0]['content']}' -> 
Generated: '{sft_decoded_response}'")
else:
    logger("Skipping SFT model test as it was not trained or saved.")
```

```
[2025-05-14 15:52:28] Testing SFT model chat capability...
```

```
This is a friendly reminder - the current text generation call will
exceed the model's predefined maximum length (64). Depending on the
model, you may observe exceptions, performance degradation, or nothing
at all.

[2025-05-14 15:52:29] SFT Prompt: 'What is the capital of France?' ->
Generated: '
```

```
'
```

What we've done in SFT: We took the pretrained model and fine-tuned it on a small dataset of conversations. The `DemoChatDataset` class helped format the data using the tokenizer's chat template and created a loss mask so that the model only learned to predict the assistant's responses. After this stage, the model should be better at engaging in simple Q&A or chat compared to the purely pretrained model.

Part 6: Reasoning Training (Self-Contained `DemoLLM`)

Theory Recap: The goal here is to teach the SFT model to produce structured reasoning outputs, specifically using `<think>...</think>` for the thought process and `<answer>...</answer>` for the final result. We achieve this by fine-tuning on a dataset where assistant responses follow this format. A crucial technique, inspired by `train_distill_reason.py` from the original project, is to increase the loss penalty on the special tag tokens (`<think>`, `</think>`, `<answer>`, `</answer>`). This encourages the model to learn to generate these structural elements correctly.

We'll load our SFT `DemoLLMForCausalLM` and fine-tune it using the `DemoChatDataset` with our `sample_reasoning.jsonl`.

6.1 Initialize Model for Reasoning (Load SFT)

```
logger("Initializing model for Reasoning, loading SFT weights...")
if tokenizer and final_sft_model_path and
os.path.exists(final_sft_model_path):
    rsn_config = DemoLLMConfig( # Same config as SFT
        vocab_size=DEMO_VOCAB_SIZE_FINAL,
        hidden_size=DEMO_HIDDEN_SIZE,
        intermediate_size=DEMO_INTERMEDIATE_SIZE,
        num_hidden_layers=DEMO_NUM_LAYERS,
        num_attention_heads=DEMO_NUM_ATTENTION_HEADS,
        num_key_value_heads=DEMO_NUM_KV_HEADS,
        max_position_embeddings=DEMO_MAX_SEQ_LEN,
        bos_token_id=tokenizer.bos_token_id,
        eos_token_id=tokenizer.eos_token_id,
        pad_token_id=tokenizer.pad_token_id
    )
    reasoning_model_d = DemoLLMForCausalLM(rsn_config).to(DEVICE)
    reasoning_model_d.load_state_dict(torch.load(final_sft_model_path,
map_location=DEVICE))
    print_model_summary(reasoning_model_d, "Demo Reasoning Model
(Loaded SFT)")
else:
    logger("Cannot initialize Reasoning model: SFT model path or
tokenizer is invalid.")
    reasoning_model_d = None
```

```
[2025-05-14 15:52:29] Initializing model for Reasoning, loading SFT
weights...
[2025-05-14 15:52:29] Initialized RotaryEmbedding with dim=16,
max_seq_len=64
[2025-05-14 15:52:29] Initialized RotaryEmbedding with dim=16,
max_seq_len=64
[2025-05-14 15:52:29] --- Demo Reasoning Model (Loaded SFT) Summary
---
[2025-05-14 15:52:29] Configuration: DemoLLMConfig {
  "_attn_implementation_autoset": true,
  "bos_token_id": 1,
  "dropout": 0.0,
  "eos_token_id": 0,
  "flash_attn": true,
  "head_dim": 16,
  "hidden_act": "silu",
  "hidden_size": 64,
  "intermediate_size": 192,
  "max_position_embeddings": 64,
  "model_type": "demo_llm",
  "num_attention_heads": 4,
  "num_hidden_layers": 2,
  "num_key_value_heads": 2,
  "pad_token_id": 3,
  "rms_norm_eps": 1e-05,
  "rope_theta": 10000.0,
  "transformers_version": "4.51.3",
  "use_cache": true,
  "vocab_size": 435
}

[2025-05-14 15:52:29] Total parameters: 0.126 M (126464)
[2025-05-14 15:52:29] Trainable parameters: 0.126 M (126464)
[2025-05-14 15:52:29] -------------------------
```

6.2 Prepare Reasoning DataLoader

```
if tokenizer and reasoning_model_d:
    demo_reasoning_dataset = DemoChatDataset(reasoning_file_path,
tokenizer, max_length=DEMO_MAX_SEQ_LEN)
    demo_reasoning_dataloader = DataLoader(demo_reasoning_dataset,
batch_size=DEMO_BATCH_SIZE, shuffle=True, num_workers=0)
    logger(f"Demo Reasoning dataset size:
{len(demo_reasoning_dataset)}")

    logger("Verifying Reasoning data sample and mask from
DataLoader:")
    for x_r_dl, y_r_dl, m_r_dl in demo_reasoning_dataloader:
        idx_to_check = 0
        logger(f"  Tokens from Y for sample {idx_to_check} where mask
is 1: {tokenizer.decode(y_r_dl[idx_to_check]
[m_r_dl[idx_to_check].bool()])}")
        full_ids_rsn_dl = torch.cat([x_r_dl[idx_to_check,:1],
y_r_dl[idx_to_check]], dim=0)
        logger(f"  Full Decoded Reasoning sample {idx_to_check}:\
n{tokenizer.decode(full_ids_rsn_dl)}")
        break
else:
    logger("Skipping Reasoning Dataloader: model or tokenizer not
initialized.")
    demo_reasoning_dataloader = []
```

```
[2025-05-14 15:52:29] Loading chat data from:
./dataset_notebook_scratch\reasoning_data.jsonl
[2025-05-14 15:52:29] Loaded 2 chat samples.
[2025-05-14 15:52:29] Demo Reasoning dataset size: 2
[2025-05-14 15:52:29] Verifying Reasoning data sample and mask from
DataLoader:
[2025-05-14 15:52:29]   Tokens from Y for sample 0 where mask is 1:
<think>The user is asking about primary colors. These are colors
[2025-05-14 15:52:29]   Full Decoded Reasoning sample 0:
user
What are the primary colors?
assistant
<think>The user is asking about primary colors. These are colors
```

6.3 Reasoning Training Loop with Special Token Weighting

```
if reasoning_model_d and demo_reasoning_dataloader and tokenizer:
    optimizer_rsn_d = optim.AdamW(reasoning_model_d.parameters(),
lr=DEMO_REASONING_LR)
    loss_fct_rsn_d = nn.CrossEntropyLoss(reduction='none')

    autocast_ctx_rsn = nullcontext() if DEVICE.type == 'cpu' else
torch.amp.autocast(device_type=DEVICE.type, dtype=PTDTYPE)
    scaler_rsn_d = torch.cuda.amp.GradScaler(enabled=(DTYPE_STR !=
'float32' and DEVICE.type == 'cuda'))

    # Get token IDs for special reasoning tags. This might create
multiple IDs if tokenized into subwords.
    # For simplicity, we'll check for the presence of the first token
of each tag sequence.
    think_start_id_rsn  = tokenizer.encode('<think>',
add_special_tokens=False)[0]
    think_end_id_rsn    = tokenizer.encode('</think>',
add_special_tokens=False)[0]
    answer_start_id_rsn = tokenizer.encode('<answer>',
add_special_tokens=False)[0]
    answer_end_id_rsn   = tokenizer.encode('</answer>',
add_special_tokens=False)[0]

    # Create a tensor of these first-token IDs for quick checking
    special_tag_first_token_ids = torch.tensor([
        think_start_id_rsn, think_end_id_rsn,
        answer_start_id_rsn, answer_end_id_rsn
    ], device=DEVICE).unique()
    logger(f"Special Reasoning Tag First Token IDs for weighting:
{special_tag_first_token_ids.tolist()}")
    REASONING_TAG_LOSS_WEIGHT = 5.0
    total_steps_rsn_d = len(demo_reasoning_dataloader) *
DEMO_REASONING_EPOCHS
    logger(f"Starting DEMO Reasoning training for
{DEMO_REASONING_EPOCHS} epochs ({total_steps_rsn_d} steps)...")

    reasoning_model_d.train()
    current_training_step_rsn = 0
    for epoch in range(DEMO_REASONING_EPOCHS):
        epoch_loss_rsn_val = 0.0
        for step, (X_batch_rsn, Y_batch_rsn, sft_style_mask_rsn) in
enumerate(demo_reasoning_dataloader):
            X_batch_rsn, Y_batch_rsn, sft_style_mask_rsn =
X_batch_rsn.to(DEVICE), Y_batch_rsn.to(DEVICE),
sft_style_mask_rsn.to(DEVICE)

            current_lr_rsn = get_lr(current_training_step_rsn,
total_steps_rsn_d, DEMO_REASONING_LR)
            for param_group in optimizer_rsn_d.param_groups:
                param_group['lr'] = current_lr_rsn

            with autocast_ctx_rsn:
                outputs_rsn_loop =
reasoning_model_d(input_ids=X_batch_rsn)
                logits_rsn_loop = outputs_rsn_loop.logits

                raw_loss_per_token_rsn =
loss_fct_rsn_d(logits_rsn_loop.view(-1, logits_rsn_loop.size(-1)),
Y_batch_rsn.view(-1))

                # Start with the SFT mask (train only on assistant
response tokens)
                effective_loss_weights = sft_style_mask_rsn.view(-
1).float().clone()

                # Identify positions of (first tokens of) special tags
in the target Y_batch_rsn
                is_special_target_token_rsn =
torch.isin(Y_batch_rsn.view(-1), special_tag_first_token_ids)

                # Apply higher weight where it's an assistant token
AND a special tag token
                apply_extra_weight = is_special_target_token_rsn &
(sft_style_mask_rsn.view(-1) == 1)
                effective_loss_weights[apply_extra_weight] *=
REASONING_TAG_LOSS_WEIGHT

                # Calculate final weighted loss, normalized by the sum
of the original SFT mask counts
                # This maintains a somewhat comparable loss magnitude
```

```
to SFT, while upweighting tags.
                weighted_loss_rsn = (raw_loss_per_token_rsn *
effective_loss_weights).sum() / sft_style_mask_rsn.sum().clamp(min=1)

            scaler_rsn_d.scale(weighted_loss_rsn).backward()
            scaler_rsn_d.step(optimizer_rsn_d)
            scaler_rsn_d.update()
            optimizer_rsn_d.zero_grad(set_to_none=True)

            epoch_loss_rsn_val += weighted_loss_rsn.item()
            current_training_step_rsn += 1

            if (step + 1) % 1 == 0:
                logger(f"Reasoning Epoch {epoch+1}, Step
{step+1}/{len(demo_reasoning_dataloader)}, Loss:
{weighted_loss_rsn.item():.4f}, LR: {current_lr_rsn:.3e}")

        logger(f"End of Reasoning Epoch {epoch+1}, Avg Loss:
{epoch_loss_rsn_val / len(demo_reasoning_dataloader):.4f}")

    logger("DEMO Reasoning training finished.")
    final_reasoning_model_path = os.path.join(NOTEBOOK_OUT_DIR,
"demo_llm_reasoning.pth")
    torch.save(reasoning_model_d.state_dict(),
final_reasoning_model_path)
    logger(f"Demo reasoning model saved to:
{final_reasoning_model_path}")
else:
    logger("Skipping Reasoning loop as model or dataloader was not
initialized.")
    final_reasoning_model_path = None
```

```
[2025-05-14 15:52:29] Special Reasoning Tag First Token IDs for
weighting: [31]
[2025-05-14 15:52:29] Starting DEMO Reasoning training for 5 epochs (5
steps)...
[2025-05-14 15:52:29] Reasoning Epoch 1, Step 1/1, Loss: 67.5133, LR:
2.000e-05
[2025-05-14 15:52:29] End of Reasoning Epoch 1, Avg Loss: 67.5133
[2025-05-14 15:52:29] Reasoning Epoch 2, Step 1/1, Loss: 67.4562, LR:
1.828e-05
[2025-05-14 15:52:29] End of Reasoning Epoch 2, Avg Loss: 67.4562
[2025-05-14 15:52:29] Reasoning Epoch 3, Step 1/1, Loss: 67.4038, LR:
1.378e-05
[2025-05-14 15:52:29] End of Reasoning Epoch 3, Avg Loss: 67.4038
[2025-05-14 15:52:29] Reasoning Epoch 4, Step 1/1, Loss: 67.3643, LR:
8.219e-06
[2025-05-14 15:52:29] End of Reasoning Epoch 4, Avg Loss: 67.3643
[2025-05-14 15:52:29] Reasoning Epoch 5, Step 1/1, Loss: 67.3407, LR:
3.719e-06
```

```
[2025-05-14 15:52:29] End of Reasoning Epoch 5, Avg Loss: 67.3407
[2025-05-14 15:52:29] DEMO Reasoning training finished.
[2025-05-14 15:52:29] Demo reasoning model saved to:
./out_notebook_scratch\demo_llm_reasoning.pth
```

What we've done in Reasoning Training: We took our SFT-trained model and further fine-tuned it on a specialized dataset containing `<think>...</think><answer>...</answer>` structures. The key modification in the training loop was to apply a higher loss weight to the special tag tokens (`<think>`, `</think>`, etc.) when they appeared in the assistant's target response. This encourages the model to prioritize learning and correctly generating these structural elements, leading to the desired "thinking" output format.

Part 7: Inference with the "Thinking" LLM

Theory: Now we test our final model. We'll provide it with prompts and observe if it generates the structured `<think>...</think><answer>...</answer>` output. The quality of the thinking and answer will depend heavily on the (very limited) training data, but the structure should be present if the reasoning training was somewhat effective.

```
if final_reasoning_model_path and
os.path.exists(final_reasoning_model_path) and tokenizer:
    logger("Loading final reasoning model for inference...")
    final_model_config = DemoLLMConfig( # Ensure config matches saved
model
        vocab_size=DEMO_VOCAB_SIZE_FINAL,
        hidden_size=DEMO_HIDDEN_SIZE,
        intermediate_size=DEMO_INTERMEDIATE_SIZE,
        num_hidden_layers=DEMO_NUM_LAYERS,
        num_attention_heads=DEMO_NUM_ATTENTION_HEADS,
        num_key_value_heads=DEMO_NUM_KV_HEADS,
        max_position_embeddings=DEMO_MAX_SEQ_LEN,
        bos_token_id=tokenizer.bos_token_id,
        eos_token_id=tokenizer.eos_token_id,
        pad_token_id=tokenizer.pad_token_id
    )
    final_thinking_llm =
DemoLLMForCausalLM(final_model_config).to(DEVICE)

final_thinking_llm.load_state_dict(torch.load(final_reasoning_model_pa
th, map_location=DEVICE))
    final_thinking_llm.eval()
    logger("Final 'thinking' model loaded.")
    print_model_summary(final_thinking_llm, "Final Thinking LLM")

    def get_structured_response(model, user_query,
max_new_toks=DEMO_MAX_SEQ_LEN - 10, temp=0.7, tk=10):
        logger(f"\n--- Generating Structured Response for:
'{user_query}' ---")
        chat_history = [{"role": "user", "content": user_query}]
        prompt_text = tokenizer.apply_chat_template(chat_history,
tokenize=False, add_generation_prompt=True)

        input_ids = tokenizer(prompt_text,
return_tensors="pt").input_ids.to(DEVICE)
        logger(f"Input prompt (len {input_ids.shape[1]}):
{prompt_text.strip()}")

        with torch.no_grad(), autocast_ctx_rsn: # Using autocast
context from reasoning training
            generated_ids = model.generate(
                input_ids,
                max_new_tokens=max_new_toks,
                do_sample=True,
                temperature=temp,
                top_k=tk,
                eos_token_id=tokenizer.eos_token_id,
                pad_token_id=tokenizer.pad_token_id
            )

        assistant_response_ids = generated_ids[0][input_ids.shape[1]:]
        assistant_response_text =
tokenizer.decode(assistant_response_ids, skip_special_tokens=True)
        logger(f"Raw Assistant Response:\n{assistant_response_text}")

        # Simple parsing for <think> and <answer> tags
        think_part = "Not found"
        answer_part = assistant_response_text # Default to full
response if tags are not perfectly formed
        try:
            if "<think>" in assistant_response_text and "</think>" in
assistant_response_text:
                think_start_idx =
assistant_response_text.find("<think>") + len("<think>")
                think_end_idx =
assistant_response_text.find("</think>")
                think_part =
assistant_response_text[think_start_idx:think_end_idx].strip()

                if "<answer>" in
assistant_response_text[think_end_idx:] and "</answer>" in
assistant_response_text[think_end_idx:]:
                    # Search for answer tag *after* the think_end tag
                    search_after_think =
assistant_response_text[think_end_idx + len("</think>"):]
                    if "<answer>" in search_after_think:
                        answer_start_idx_rel =
search_after_think.find("<answer>") + len("<answer>")
                        answer_start_idx_abs = think_end_idx +
len("</think>") + answer_start_idx_rel
                        if "</answer>" in
assistant_response_text[answer_start_idx_abs:]:
                            answer_end_idx =
assistant_response_text.find("</answer>", answer_start_idx_abs)
                            answer_part =
assistant_response_text[answer_start_idx_abs:answer_end_idx].strip()
                        else: # No closing answer tag after opening
                            answer_part =
assistant_response_text[answer_start_idx_abs:].strip()
                    else: # No answer tag found after think block
                         answer_part = search_after_think.strip() #
Consider rest as answer
            elif "<answer>" in assistant_response_text and "</answer>"
in assistant_response_text: # Only answer tags
                answer_start_idx =
assistant_response_text.find("<answer>") + len("<answer>")
                answer_end_idx =
assistant_response_text.find("</answer>")
                answer_part =
assistant_response_text[answer_start_idx:answer_end_idx].strip()
        except Exception as e:
            logger(f"Error parsing think/answer tags: {e}")
            # Fallback to full response handled by default answer_part
initialization

        logger(f"==> Parsed <think>: {think_part}")
        logger(f"==> Parsed <answer>: {answer_part}")
        logger("------------------------")
        return think_part, answer_part

    # Test queries
    get_structured_response(final_thinking_llm, "If I have 3 apples
and eat 1, how many are left?")
    get_structured_response(final_thinking_llm, "What are the primary
colors?")
    get_structured_response(final_thinking_llm, "Hello! Tell me a
joke.")

else:
    logger("Skipping final inference test as reasoning model or
tokenizer was not available/trained.")
```

```
[2025-05-14 15:52:29] Loading final reasoning model for inference...
[2025-05-14 15:52:29] Initialized RotaryEmbedding with dim=16,
max_seq_len=64
[2025-05-14 15:52:29] Initialized RotaryEmbedding with dim=16,
max_seq_len=64
[2025-05-14 15:52:29] Final 'thinking' model loaded.
[2025-05-14 15:52:29] --- Final Thinking LLM Summary ---
```

```
[2025-05-14 15:52:29] Configuration: DemoLLMConfig {
  "_attn_implementation_autoset": true,
  "bos_token_id": 1,
  "dropout": 0.0,
  "eos_token_id": 0,
  "flash_attn": true,
  "head_dim": 16,
  "hidden_act": "silu",
  "hidden_size": 64,
  "intermediate_size": 192,
  "max_position_embeddings": 64,
  "model_type": "demo_llm",
  "num_attention_heads": 4,
  "num_hidden_layers": 2,
  "num_key_value_heads": 2,
  "pad_token_id": 3,
  "rms_norm_eps": 1e-05,
  "rope_theta": 10000.0,
  "transformers_version": "4.51.3",
  "use_cache": true,
  "vocab_size": 435
}

[2025-05-14 15:52:29] Total parameters: 0.126 M (126464)
[2025-05-14 15:52:29] Trainable parameters: 0.126 M (126464)
[2025-05-14 15:52:29] -------------------------
[2025-05-14 15:52:29]
--- Generating Structured Response for: 'If I have 3 apples and eat 1,
how many are left?' ---
[2025-05-14 15:52:29] Input prompt (len 51): <|im_start|>user
If I have 3 apples and eat 1, how many are left?<|im_end|>
<|im_start|>assistant
[2025-05-14 15:52:29] Raw Assistant Response:
```

```
[2025-05-14 15:52:29] ==> Parsed <think>: Not found
[2025-05-14 15:52:29] ==> Parsed <answer>:
```

```
[2025-05-14 15:52:29] ------------------------
[2025-05-14 15:52:29]
--- Generating Structured Response for: 'What are the primary colors?'
---
[2025-05-14 15:52:29] Input prompt (len 25): <|im_start|>user
What are the primary colors?<|im_end|>
<|im_start|>assistant
[2025-05-14 15:52:30] Raw Assistant Response:
```

```
[2025-05-14 15:52:30] ==> Parsed <think>: Not found
[2025-05-14 15:52:30] ==> Parsed <answer>:
```

```
[2025-05-14 15:52:30] ------------------------
[2025-05-14 15:52:30]
--- Generating Structured Response for: 'Hello! Tell me a joke.' ---
```

```
[2025-05-14 15:52:30] Input prompt (len 24): <|im_start|>user
Hello! Tell me a joke.<|im_end|>
<|im_start|>assistant
[2025-05-14 15:52:30] Raw Assistant Response:
```

```
[2025-05-14 15:52:30] ==> Parsed <think>: Not found
[2025-05-14 15:52:30] ==> Parsed <answer>:
```

```
[2025-05-14 15:52:30] ------------------------
```

Part 8: Conclusion and Next Steps

This notebook has provided a detailed, self-contained walkthrough of creating a small "thinking" Large Language Model. We covered:

1. **Tokenizer Training:** A simple BPE tokenizer was trained from scratch using the `tokenizers` library.
2. **Data Preparation:** Custom `Dataset` classes were implemented to handle pretraining, SFT, and reasoning data, all using our trained tokenizer.
3. **Model Architecture:** Key components of a Transformer decoder (RMSNorm, RoPE, Attention, FFN, Blocks, LM Head) were implemented directly within the notebook under the `DemoLLM` family of classes.
4. **Training Pipeline:** We executed three distinct training phases:
   - **Pretraining:** To instill basic language understanding.
   - **Supervised Fine-Tuning (SFT):** To teach instruction following and conversational ability, with loss masking on assistant responses.
   - **Reasoning Training:** To encourage the model to output explicit `<think>...</think>` processes before an `<answer>...</answer>`, using weighted loss for special tags.
5. **Inference:** We demonstrated how to use the final model and attempt to parse its structured output.

Key Observations from this Demo:

- **Complexity:** Building even a simplified LLM from scratch involves many interconnected components.
- **Data is King:** The structure and content of training data at each stage are paramount. The reasoning data with explicit tags was crucial for the desired output format.
- **Computational Demands:** Even with tiny data and model sizes, training can be slow, highlighting the immense resources needed for state-of-the-art LLMs.
- **Fragility of Small Models:** The outputs from this demo model will be very basic and likely not robust due to the extremely limited scale. The "reasoning" observed is more pattern imitation than deep understanding.

Further Exploration (Beyond this Notebook):

- **Scale Up:** Use significantly larger datasets, vocabulary sizes, and model dimensions.
- **Advanced Tokenization:** Explore more sophisticated tokenizer training or use well-established pretrained tokenizers if starting a new project from scratch isn't a hard requirement for the tokenizer itself.

- **Efficient Model Architectures:** Implement features like Mixture of Experts (MoE), Flash Attention (if hardware supports it more robustly than the basic check here), and Grouped Query Attention (GQA) more thoroughly.
- **Sophisticated Training:** Use distributed training (DDP/FSDP), more advanced optimizers and learning rate schedulers, gradient checkpointing, etc.
- **RLHF/DPO:** For better alignment with human preferences and more nuanced control over generation, explore Reinforcement Learning from Human Feedback or Direct Preference Optimization.
- **Rigorous Evaluation:** Employ standard NLP benchmarks (e.g., GLUE, SuperGLUE, MMLU, domain-specific tests for reasoning) to quantitatively assess model performance.

This notebook serves as an educational tool to demystify the core mechanics. Building production-grade LLMs is a significant engineering and research effort. Hopefully, this detailed guide provides a solid foundation for your LLM journey!