A Deep Dive into Incremental Pre-training for Large Language Models

> This guide provides a comprehensive overview of incremental pre-training for Large Language Models (LLMs). We will explore its fundamental purpose, the necessary preparations, and the strategic decisions involved in adapting a powerful base model to a specific domain. The document covers key concepts such as data curation, training frameworks for various scales (from massive clusters to single GPUs), data mixing strategies, and the detailed step-by-step workflow. It is designed to be a thorough resource for both understanding the theory and preparing for practical interview questions on the topic.

Knowledge Section

1. The LLM Training Trilogy: Pre-training, Fine-tuning, and Alignment

To understand incremental pre-training, it's crucial to place it within the broader context of LLM development. The lifecycle of a modern conversational LLM can be simplified into three main stages:

1. **Foundational Pre-training:** This is where the model learns general knowledge, grammar, reasoning abilities, and a vast array of facts about the world. It is trained on a massive, diverse corpus of text (terabytes of data) with a simple objective: predicting the next token in a sequence. This stage is computationally exorbitant and establishes the model's core capabilities.
2. **Instruction Fine-Tuning (SFT):** After pre-training, the model knows a lot but doesn't necessarily know how to follow instructions or act as a helpful assistant. SFT teaches the model the *format* of interaction. It's trained on a smaller, high-quality dataset of instruction-response pairs (e.g., "Question: What is the capital of France? Answer: The capital of France is Paris."). This teaches the model to be a helpful, instruction-following agent rather than just a text completer.
3. **Alignment (e.g., RLHF):** Reinforcement Learning from Human Feedback (RLHF) further refines the model's behavior to align with human preferences, ensuring it is

helpful, harmless, and honest. This involves training a reward model on human-ranked responses and then using reinforcement learning (like PPO) to optimize the LLM to generate outputs that maximize this reward.

So, where does incremental pre-training fit? It is a form of **continued foundational pre-training**, but on a specialized, domain-specific dataset. Its primary goal is to **inject deep domain knowledge** that was sparse or absent in the original general corpus. Trying to "teach" a model complex topics like legal case law or advanced molecular biology solely through instruction fine-tuning is inefficient and often ineffective, as SFT is better suited for learning interaction patterns, not for memorizing and internalizing vast new knowledge bases.

2. Preparing for Incremental Pre-training

Executing a successful incremental pre-training run requires careful planning and preparation across several key areas.

2.1. Selecting the Foundation (Base) Model

The choice of the base model is a critical first step, balancing performance, resource availability, and project goals.

- **Model Architecture and Size:** Larger models (e.g., 70B parameters) have greater capacity to absorb new knowledge but require significantly more computational resources. Smaller models (e.g., 7B or 13B) are more accessible and can be very effective for narrower domains.
- **Open-Source vs. Proprietary:** The community has largely converged on open-source models for this task. Leading candidates (as of mid-2024) include Meta's **LLaMA 3** series, Mistral AI's models (**Mistral 7B, Mixtral 8x7B**), and others from the TII Falcon family. The key is to choose a **base model**, not an instruction-tuned or chat variant, as the latter's training can interfere with the raw knowledge acquisition process.
- **License and Community:** Ensure the model's license (e.g., Apache 2.0, Llama 3 Community License) is compatible with your intended use case. A strong community and ecosystem (like Hugging Face) are invaluable for tools, support, and pre-existing configurations.

2.2. Curating the Domain Corpus

This is arguably the most crucial step. The quality and composition of your data directly determine the success of the project.

- **Data Sourcing:** Collect vast amounts of text relevant to your target domain (e.g., legal documents, medical research papers, software documentation). This data can be proprietary or sourced from the web.
- **Data Cleaning and Preprocessing:** Raw data is noisy. A rigorous cleaning pipeline is essential:
    - **Deduplication:** Remove duplicate or near-duplicate documents to improve training efficiency and prevent the model from overfitting to repeated samples.
    - **Quality Filtering:** Filter out low-quality content, such as boilerplate text, navigation menus, or poorly formatted text. Heuristics like language detection, text-to-code ratio, and perplexity filtering can be used.
    - **PII Removal:** Scrub Personally Identifiable Information (PII) to ensure privacy and security.
    - **Enhancing Information Density:** The core idea is to maximize the "signal" in your data. For instance, a well-structured textbook or a Wikipedia article has higher information density about a topic than a series of scattered news articles. Pre-processing should aim to structure and clean the data to make knowledge more easily learnable for the model.

2.3. Adapting the Tokenizer

The tokenizer translates raw text into a sequence of integer IDs that the model can process. If your domain contains a lot of specialized vocabulary not present in the base model's tokenizer, it's highly beneficial to extend it.

- **Why Extend?** Without extension, a specialized term like "immuno-oncology" might be split into multiple, less meaningful tokens (e.g., "im", "##muno", "-", "onco", "##logy"). A single, dedicated token is far more efficient for the model to learn and associate with a coherent concept.
- **Process:**

i. Train a new tokenizer (e.g., using `SentencePiece`) on your domain-specific corpus.
ii. Merge the new domain-specific tokens with the vocabulary of the original tokenizer.
iii. Resize the model's token embedding matrix to accommodate the new vocabulary size. The new embeddings are typically initialized with the average of the existing embeddings.

3. Training Frameworks and Parallelism

Training LLMs requires massive computational resources. The choice of framework depends on the scale of your hardware.

- **Massive Scale (Large Clusters): 3D Parallelism**
  When training on hundreds or thousands of GPUs, a combination of parallelism strategies is required. **Megatron-DeepSpeed** is a popular framework for this.
    - **Data Parallelism (DP):** The simplest form. The model is replicated on each GPU, and each GPU processes a different batch of data. Gradients are averaged across all GPUs before the weights are updated.
    - **Tensor Parallelism (TP):** Splits individual layers (matrix multiplications) of the model across multiple GPUs. For example, a weight matrix `W` can be split column-wise `W = [W_a, W_b]`, and the computation `Y = XW` becomes `[Y_a, Y_b] = [XW_a, XW_b]`. This requires high-speed interconnects like **NVIDIA's NVLink** for efficient communication between GPUs, as the results of the split computations must be exchanged at every forward and backward pass.
    - **Pipeline Parallelism (PP):** Splits the entire model vertically, placing different layers onto different GPUs. GPU 1 might handle layers 1-10, GPU 2 layers 11-20, and so on. This reduces the memory burden on each GPU but introduces "bubbles" where GPUs are idle waiting for the previous stage to complete.
- **Medium Scale (Multi-GPU Node): Tensor or FSDP**
  On a single server with multiple GPUs connected by NVLink, Tensor Parallelism is effective. Alternatively, **Fully Sharded Data Parallelism (FSDP)** from PyTorch is a

powerful and increasingly popular choice. FSDP shards the model's parameters, gradients, and optimizer states across data-parallel workers, offering a flexible way to manage memory.

- **Low Resource (Single/Few GPUs): Parameter-Efficient Fine-Tuning (PEFT)** If a full pre-training run is computationally infeasible, PEFT methods like **LoRA (Low-Rank Adaptation)** offer a highly efficient alternative for domain adaptation.
    - **LoRA:** Instead of updating the entire set of model weights (W_0), LoRA freezes them and injects small, trainable "adapter" matrices (A and B) into the layers. The update is represented by a low-rank decomposition: $\Delta W = BA$. The forward pass becomes $h = W_0x + BAx$. Since A and B are much smaller than W_0, the number of trainable parameters and memory usage is drastically reduced. While technically a fine-tuning method, LoRA can be used for "pre-training-like" domain adaptation on custom datasets.

4. Data Mixing Strategies

When performing incremental pre-training, you must decide how to combine your new domain-specific data with general-purpose data.

1. **Two-Stage Approach:** First, pre-train on a massive general corpus (this is the base model you start with). Then, conduct a second phase of pre-training exclusively on the domain corpus.
    - **Pros:** Simple to implement.
    - **Cons:** High risk of **catastrophic forgetting**, where the model's performance on general tasks degrades significantly as it over-specializes on the new domain.
2. **Domain-Only Approach:** Train a model from scratch using only a large-scale domain corpus.
    - **Pros:** Creates a true domain expert model.
    - **Cons:** Extremely expensive and often impractical. The model will lack general world knowledge and common-sense reasoning abilities.
3. **Mixed Corpus Approach (Recommended):** Create a training dataset that is a mixture of the original general-purpose data and your new domain-specific data.
    - **Pros:** This is the most effective and common strategy. It allows the model to learn the new domain knowledge while continuously

rehearsing its general capabilities, thus mitigating catastrophic forgetting.
    - **Cons:** Requires careful thought about the **mixing ratio**. A common strategy is to over-sample the high-quality domain data while still including a substantial amount of general data. The optimal ratio is an empirical question that depends on the domain and the dataset sizes.

5. The Incremental Pre-training Workflow

1. **Data Preprocessing:** Take your curated text corpus and process it for the model. Using the model's tokenizer, convert the text into token IDs. Concatenate documents with a special End-of-Sequence (EOS) token and then chunk the resulting stream into fixed-length sequences (e.g., 2048 or 4096 tokens, matching the model's context window). Pad the last sequence if it's too short.
2. **Tokenizer and Model Setup:** Load the base model and tokenizer. If you extended the tokenizer, ensure you also resize the model's embedding layer (`model.resize_token_embeddings(len(tokenizer))`).
3. **Training Configuration:**
    - **Objective:** The training objective is **Causal Language Modeling** (or next-token prediction). The loss function is the **Cross-Entropy Loss** calculated over the vocabulary for each token position.
    - **Optimizer:** The **AdamW** optimizer is a standard choice.
    - **Learning Rate Scheduler:** A `Cosine` schedule with a warm-up period is commonly used. The peak learning rate for continued pre-training is typically smaller than for pre-training from scratch (e.g., 1e-5 vs 3e-4).
4. **Launch and Monitor Training:** Start the training job. Continuously monitor key metrics:
    - **Training Loss:** Should decrease steadily and smoothly. Spikes may indicate instability.
    - **Perplexity (on a held-out validation set):** Perplexity is a measure of how well the model predicts a sample of text. Lower is better. This is a crucial metric for tracking model quality.
5. **Checkpointing and Conversion:** Periodically save model checkpoints. These checkpoints can be large and split into multiple files depending on the parallelism

strategy. After training, you may need to convert and consolidate these files into a standard format, like Hugging Face's `safetensors`, for easy downstream use.

6. **Model Evaluation:** Perform a final evaluation.
    - **Quantitative:** Measure perplexity on domain-specific and general-purpose test sets.
    - **Qualitative:** Test the model's text generation and continuation capabilities with domain-specific prompts to check if it has acquired the desired knowledge and style.

Interview Questions

Theoretical Questions

Q1: Differentiate between pre-training from scratch, incremental pre-training, instruction fine-tuning, and RLHF.

Answer:
These four processes represent distinct stages in the lifecycle of a modern LLM, each with a different objective.

- **Pre-training from Scratch:** This is the foundational stage where a model is trained on a massive, general-purpose dataset (trillions of tokens). Its objective is **knowledge acquisition** through a self-supervised task, typically next-token prediction. It learns grammar, facts, reasoning, and language structure. This process is extremely computationally expensive.
- **Incremental Pre-training:** This is a **continuation** of the pre-training process, but on a smaller, specialized, domain-specific dataset (e.g., legal or medical text). Its objective is **knowledge injection**—to make the model an expert in a new domain that was underrepresented in the original training data. It uses the same next-token prediction objective.
- **Instruction Fine-Tuning (SFT):** This stage takes a pre-trained model and teaches it to follow instructions. Its objective is to learn the **format of interaction**. It is trained on a curated dataset of prompt-response pairs. This is what transforms a text-completion model into a helpful assistant.

- **RLHF (Reinforcement Learning from Human Feedback):** This is the final alignment stage. Its objective is to make the model's behavior align with human preferences (e.g., being helpful, harmless, and not generating biased content). It involves training a reward model on human-ranked outputs and then using RL to fine-tune the LLM to maximize this reward signal.

In short: **Pre-training** learns general knowledge, **Incremental Pre-training** learns domain knowledge, **SFT** learns the instruction-following format, and **RLHF** learns to be safe and helpful.

Q2: What is "catastrophic forgetting" in the context of LLMs, and how does incremental pre-training on a mixed corpus help mitigate it?

Answer:
Catastrophic forgetting is the phenomenon where a neural network, upon learning a new task, abruptly and completely forgets the knowledge it had learned from a previous task. In the context of LLMs, if you take a general-purpose model like LLaMA 3 and continue training it *only* on a narrow domain corpus (e.g., legal documents), it will become an expert in law but may forget how to perform basic math, answer general knowledge questions, or even write coherent conversational English. The model's weights are aggressively updated to minimize loss on the new data distribution, overwriting the parameters that encoded the previous general knowledge.

A mixed corpus approach mitigates this issue by combining the new domain-specific data with data from the original general pre-training distribution. During each training step, the model is exposed to both types of data.

- When it sees a batch of domain data, its weights are updated to improve its domain expertise.
- When it sees a batch of general data, its weights are updated to "rehearse" and reinforce its original, general-purpose knowledge.

This acts as a regularizer, preventing the model's parameters from drifting too far in the direction of the new domain at the expense of its foundational capabilities. It finds a balance in the parameter space that achieves low loss on both data distributions, thus learning the new information without forgetting the old.

Q3: You need to adapt a model like LLaMA 3 for the legal domain. Your legal corpus is much smaller than the original pre-training data. Should you extend the tokenizer? What are the pros and cons?

Answer:
Yes, you should strongly consider extending the tokenizer. Here's a breakdown of the pros and cons:

Pros of Extending the Tokenizer:

1. **Semantic Efficiency:** The legal domain has a rich, specific vocabulary (e.g., "estoppel," "subpoena," "res judicata"). By giving these terms their own unique tokens, the model can learn a single, dense semantic representation for each concept. Without this, "subpoena" might be tokenized as `["sub", "##po", "##ena"]`, forcing the model to learn the meaning from a combination of sub-word fragments, which is less efficient and less precise.
2. **Computational Efficiency:** Representing a common domain term with a single token instead of multiple tokens shortens the sequence length for a given piece of text. This reduces the computational and memory cost during both training and inference, as the attention mechanism's complexity is quadratic with respect to sequence length ($O(n^2)$).
3. **Improved Model Performance:** By creating more meaningful input representations, the model can grasp the domain's nuances more effectively, leading to better downstream performance on legal tasks.

Cons of Extending the Tokenizer:

1. **Model Architecture Modification:** You must resize the model's token embedding matrix (`nn.Embedding` layer). This is a structural change. The new token embeddings must be initialized, typically by averaging the existing embeddings, and then learned from scratch, which requires sufficient data.
2. **Training Overhead:** The new embeddings need to be trained. While the legal corpus is "smaller" than the original pre-training data, it still needs to be substantial enough for these new embeddings to converge to meaningful representations.
3. **Complexity:** It adds a step to the pipeline (training a new tokenizer, merging vocabularies, resizing the model), increasing the overall complexity of the project.

Conclusion: Despite the cons, the benefits of extending the tokenizer for a specialized domain like law almost always outweigh the drawbacks, provided you have a reasonably sized domain corpus (e.g., billions of tokens). The improvement in semantic and computational efficiency leads to a better and more effective final model.

Q4: Explain the mathematical formulation of LoRA. Why is it so memory-efficient compared to full fine-tuning?

Answer:
LoRA (Low-Rank Adaptation) is a parameter-efficient fine-tuning (PEFT) technique that avoids updating the full weight matrices of a pre-trained model.

Mathematical Formulation:
Let a pre-trained weight matrix in a specific layer be $W_0 \in \mathbb{R}^{d \times k}$. During full fine-tuning, we would update this entire matrix with an update ΔW, such that the new weight matrix is $W = W_0 + \Delta W$.

LoRA's key insight is that the update matrix ΔW can be approximated by a low-rank decomposition. It represents ΔW as the product of two much smaller matrices:

$$\Delta W = BA$$

where $B \in \mathbb{R}^{d \times r}$ and $A \in \mathbb{R}^{r \times k}$. Here, r is the rank, and it is a hyperparameter chosen to be much smaller than d and k (i.e., $r \ll \min(d, k)$).

During training, the original weights W_0 are **frozen**, and only the matrices A and B are trainable. The forward pass of the layer is modified from $h = W_0 x$ to:

$$h = W_0 x + BAx$$

This can be seen as a parallel "adapter" branch whose output is added to the original output.

Memory Efficiency:
The memory efficiency comes from the drastic reduction in the number of trainable parameters.

- **Full Fine-Tuning:** The number of trainable parameters for the layer is the size of

W_0, which is $d \times k$.
- **LoRA:** The number of trainable parameters is the sum of the sizes of A and B, which is $(d \times r) + (r \times k) = r(d + k)$.

Example:
Consider a layer where $d = 4096$ and $k = 4096$.

- Full fine-tuning parameters: $4096 \times 4096 = 16,777,216$.
- LoRA with a small rank $r = 8$: $8 \times (4096 + 4096) = 8 \times 8192 = 65,536$.

This is a reduction of over **250x** in trainable parameters for this single layer. When applied across the model, this massively reduces the memory required for storing gradients and optimizer states (like Adam's momentum and variance terms), making it possible to adapt huge models on consumer-grade GPUs.

Q5: What is the objective function for Causal Language Modeling during pre-training? Write it down using LaTeX and explain each term.

Answer:
The objective function for Causal Language Modeling (CLM), which is the standard for models like GPT and LLaMA, is to maximize the likelihood of the next token given the previous tokens. This is typically framed as minimizing the negative log-likelihood, which is equivalent to the **Cross-Entropy Loss**.

Given a sequence of tokens $T = (t_1, t_2, \ldots, t_L)$, the likelihood of the sequence is factorized by the chain rule of probability:

$$P(T) = \prod_{i=1}^{L} P(t_i | t_1, t_2, \ldots, t_{i-1})$$

The objective is to train a model with parameters θ to maximize this probability. Taking the logarithm and negating it gives us the loss function for a single sequence:

$$\mathcal{L}(\theta) = -\log P(T) = -\sum_{i=1}^{L} \log P(t_i | t_{<i}; \theta)$$

This is the **Negative Log-Likelihood (NLL)** loss.

In practice, for a large corpus of sequences, we minimize the average NLL, which is the **Cross-Entropy Loss**. For the i-th token in a sequence, the model outputs a probability distribution over the entire vocabulary V. Let p_i be this predicted distribution and y_i be the one-hot encoded vector representing the true next token t_i. The cross-entropy loss for this single prediction is:

$$H(y_i, p_i) = -\sum_{j=1}^{|V|} y_{i,j} \log(p_{i,j})$$

Since y_i is one-hot, only the term corresponding to the true token t_i is non-zero. So, this simplifies to $-\log(p_{i,t_i})$.

The final loss for a sequence is the average of these losses over all token positions:

$$\mathcal{L}_{CLM}(\theta) = \frac{1}{L} \sum_{i=1}^{L} -\log P(t_i | t_{<i}; \theta)$$

- L: The length of the token sequence.
- t_i: The token at position i.
- $t_{<i}$: The context, i.e., all tokens preceding position i.
- θ: The parameters of the model.
- $P(t_i | t_{<i}; \theta)$: The probability of the true token t_i being the next token, as predicted by the model with parameters θ given the context $t_{<i}$.

Practical & Coding Questions

Q1: You are given a base LLaMA model from Hugging Face and a new list of domain-specific terms. Write a Python script to add these new tokens to the tokenizer and resize the model's embedding layer accordingly.

```python
import torch
from transformers import AutoTokenizer, AutoModelForCausalLM

def add_new_tokens_to_model(model_name, new_tokens):
    """
    Adds new tokens to a model's tokenizer and resizes its embedding layer.

    Args:
        model_name (str): The Hugging Face model identifier (e.g., 'meta-llama/Llama-2-7b-
        new_tokens (list of str): A list of new string tokens to add.

    Returns:
        tuple: A tuple containing the updated model and tokenizer.
    """
    # 1. Load the tokenizer and model
    # Note: For gated models like LLaMA, you need to provide an access token.
    # from huggingface_hub import login; login("YOUR_HF_TOKEN")
    tokenizer = AutoTokenizer.from_pretrained(model_name)
    model = AutoModelForCausalLM.from_pretrained(model_name)

    # LLaMA tokenizers often don't have a pad token, add one if needed.
    if tokenizer.pad_token is None:
        # Add a pad token. We use a new special token to avoid conflicts.
        # It's common to use the eos_token as a pad_token for LLaMA models.
        tokenizer.add_special_tokens({'pad_token': '[PAD]'})
        print("Added [PAD] token.")

    # 2. Add new tokens to the tokenizer
    print(f"Original vocabulary size: {len(tokenizer)}")
    tokenizer.add_tokens(new_tokens)
    print(f"New vocabulary size: {len(tokenizer)}")

    # 3. Resize the model's token embedding matrix
    # The new embeddings will be randomly initialized.
    model.resize_token_embeddings(len(tokenizer))

    # It's good practice to initialize the new embeddings intelligently.
    # For example, initialize the new token embeddings with the mean of the old ones.
    # Also initialize the new padding token embedding to zero.
    with torch.no_grad():
        # Get the average of the existing embeddings
        old_embeddings = model.get_input_embeddings().weight.data
        avg_embeddings = old_embeddings[:-len(new_tokens)-1].mean(dim=0, keepdim=True)

        # Set the new token embeddings to this average
        model.get_input_embeddings().weight.data[-len(new_tokens):] = avg_embeddings

        # Specifically handle the pad token if we added it.
        # Find its ID and set its embedding to zero.
        pad_token_id = tokenizer.pad_token_id
        if pad_token_id is not None:
             model.get_input_embeddings().weight.data[pad_token_id] = torch.zeros_like(avg
        print("New token embeddings initialized with the average of existing embeddings."
        print(f"Pad token embedding at ID {pad_token_id} initialized to zeros.")


    # 4. Verify the changes
    print(f"Model embedding matrix size: {model.get_input_embeddings().weight.size()}")

    # Test encoding one of the new tokens
    test_token = new_tokens[0]
    encoded = tokenizer.encode(test_token)
    print(f"Encoded '{test_token}': {encoded}")
    # The new token should be a single ID > original vocab size
    assert len(encoded) == 1, "New token should be encoded as a single ID."

    return model, tokenizer

# --- Example Usage ---
# You need to have access to the LLaMA model on Hugging Face Hub.
# Replace with a model you have access to, like a smaller open model.
# For this example, let's use a widely available smaller model.
model_id = "gpt2" # Using gpt2 for accessibility
# model_id = "meta-llama/Meta-Llama-3-8B" # Example for Llama 3

domain_specific_tokens = ["<legal_case>", "</legal_case>", "res_judicata", "force_majeure"

updated_model, updated_tokenizer = add_new_tokens_to_model(model_id, domain_specific_toke

# You can now save the updated model and tokenizer for your training job
# updated_model.save_pretrained("./my_adapted_model")
# updated_tokenizer.save_pretrained("./my_adapted_model")
```

Q2: Implement a simple data processing script using Hugging Face `datasets` that takes a large text file, tokenizes it, and chunks it into fixed-length sequences for pre-training.

```python
from datasets import load_dataset
from transformers import AutoTokenizer

def create_pretraining_dataset(text_file_path, tokenizer_name, block_size=2048):
    """
    Loads a text file, tokenizes it, and groups it into fixed-length chunks.

    Args:
        text_file_path (str): Path to the input text file (one document per line).
        tokenizer_name (str): Hugging Face identifier for the tokenizer.
        block_size (int): The desired sequence length for model input.

    Returns:
        datasets.Dataset: The processed dataset, ready for training.
    """
    # 1. Load the tokenizer
    tokenizer = AutoTokenizer.from_pretrained(tokenizer_name)

    # 2. Load the raw text file as a dataset object
    # The 'text' type assumes one document per line.
    raw_dataset = load_dataset('text', data_files=text_file_path, split='train')

    # 3. Define the tokenization function
    def tokenize_function(examples):
        # The tokenizer will convert the text into a list of token IDs.
        return tokenizer(examples['text'])

    # Tokenize the entire dataset. `batched=True` speeds this up significantly.
    # `remove_columns` gets rid of the original 'text' column.
    tokenized_dataset = raw_dataset.map(
        tokenize_function,
        batched=True,
        remove_columns=raw_dataset.column_names
    )

    # 4. Define the chunking function
    def group_texts(examples):
        # Concatenate all texts from the batch
        concatenated_examples = {k: sum(examples[k], []) for k in examples.keys()}
        total_length = len(list(concatenated_examples.values())[0])

        # We drop the small remainder, but you could pad it instead if you want to use it
        total_length = (total_length // block_size) * block_size

        # Split by chunks of block_size
        result = {
            k: [t[i : i + block_size] for i in range(0, total_length, block_size)]
            for k, t in concatenated_examples.items()
        }
        # The trainer will handle creating the 'labels' for causal LM
        result["labels"] = result["input_ids"].copy()
        return result

    # Apply the chunking function to the tokenized dataset
    lm_dataset = tokenized_dataset.map(
        group_texts,
        batched=True, # Process multiple texts at once
    )

    print(f"Created a dataset with {len(lm_dataset)} samples of size {block_size}.")
    print("Example sample:")
    print(lm_dataset[0].keys())
    print("Length of input_ids:", len(lm_dataset[0]['input_ids']))

    return lm_dataset


# --- Example Usage ---
# First, create a dummy text file for demonstration
dummy_text = """This is the first document. It contains some text.
This is a second, slightly longer document. It will be concatenated with the first.
And a third one to ensure we have enough tokens to create at least one full block. Large
The most common objective is next-token prediction, also known as causal language modelin
This process requires text to be chunked into fixed-size blocks for efficient batching on
"""
with open("my_domain_data.txt", "w") as f:
    f.write(dummy_text)
```

```python
# Use a common tokenizer
tokenizer_id = "gpt2"
# The block size should match your model's context window
context_window_size = 128 # Using a small size for demonstration

processed_dataset = create_pretraining_dataset(
    text_file_path="my_domain_data.txt",
    tokenizer_name=tokenizer_id,
    block_size=context_window_size
)
```

Q3: Write a PyTorch code snippet that demonstrates the core logic of a LoRA layer. Show how the low-rank matrices `A` and `B` are used to modify the output of a standard linear layer.

```python
import torch
import torch.nn as nn
import math

class LoRALinearLayer(nn.Module):
    def __init__(self, in_features, out_features, rank, alpha):
        """
        A LoRA-augmented Linear layer.

        Args:
            in_features (int): Number of input features for the linear layer.
            out_features (int): Number of output features for the linear layer.
            rank (int): The rank 'r' of the low-rank update.
            alpha (int): A scaling factor for the LoRA update.
        """
        super().__init__()
        self.rank = rank
        self.alpha = alpha

        # 1. The original, pre-trained linear layer
        self.linear = nn.Linear(in_features, out_features, bias=False)
        # Freeze the original weights
        self.linear.weight.requires_grad = False

        # 2. The low-rank adaptation matrices A and B
        self.lora_A = nn.Parameter(torch.empty(rank, in_features))
        self.lora_B = nn.Parameter(torch.empty(out_features, rank))

        # Initialize the LoRA matrices
        nn.init.kaiming_uniform_(self.lora_A, a=math.sqrt(5))
        nn.init.zeros_(self.lora_B) # Initialize B to zero so the initial update is zero

    def forward(self, x):
        # Original forward pass (frozen)
        original_output = self.linear(x)

        # LoRA adapter forward pass (trainable)
        # The update is scaled by alpha/rank
        lora_update = (self.lora_B @ self.lora_A) * (self.alpha / self.rank)
        adapter_output = torch.nn.functional.linear(x, lora_update)

        # Combine the outputs
        final_output = original_output + adapter_output

        return final_output

# --- Example Usage ---
# Configuration
input_dim = 512  # e.g., model's hidden dimension
output_dim = 512 # e.g., model's hidden dimension
lora_rank = 8
lora_alpha = 16

# Create the LoRA layer
lora_layer = LoRALinearLayer(input_dim, output_dim, rank=lora_rank, alpha=lora_alpha)

# Print the number of parameters
original_params = sum(p.numel() for p in lora_layer.linear.parameters())
lora_params = sum(p.numel() for p in [lora_layer.lora_A, lora_layer.lora_B])
total_params = sum(p.numel() for p in lora_layer.parameters() if p.requires_grad)

print(f"Original Linear Layer Parameters: {original_params}")
print(f"LoRA (A+B) Parameters: {lora_params}")
print(f"Total Trainable Parameters in this Layer: {total_params}")
print("-" * 30)

# Verify which parameters are trainable
for name, param in lora_layer.named_parameters():
    print(f"Parameter '{name}' is trainable: {param.requires_grad}")

# Example forward pass
input_tensor = torch.randn(4, 10, input_dim) # (batch_size, sequence_length, features)
output_tensor = lora_layer(input_tensor)

print(f"\nInput shape: {input_tensor.shape}")
print(f"Output shape: {output_tensor.shape}")
```