A Comprehensive Guide to Fine-Tuning Large Language Models

> This guide provides a deep dive into the essential concepts, strategies, and practical challenges of adapting Large Language Models (LLMs) to specific tasks and domains. We will cover the full lifecycle, from understanding VRAM requirements for full fine-tuning to advanced techniques like Parameter-Efficient Fine-Tuning (PEFT). Key topics include Supervised Fine-Tuning (SFT), mitigating catastrophic forgetting, data curation for domain adaptation, and overcoming common training obstacles like Out-of-Memory (OOM) errors. This document is structured to serve as both a conceptual knowledge base and a practical interview preparation resource.

Knowledge Section

1. The LLM Training Lifecycle: Pre-training vs. Fine-tuning

The power of modern LLMs comes from a two-stage process: pre-training and fine-tuning. Understanding the distinction is fundamental.

1.1. Pre-training: Building General Knowledge

In the **pre-training phase**, a model is trained on a vast, unlabeled corpus of text from the internet (e.g., Common Crawl, Wikipedia, books). The goal is to learn general patterns, grammar, semantics, reasoning abilities, and a broad understanding of the world. This is achieved through self-supervised objectives, where the data itself provides the labels.

- **Objective:** Learn a general representation of language.
- **Data:** Massive, unlabeled, and diverse text corpora (terabytes of data).
- **Knowledge Injected:** Broad, general-world knowledge and linguistic competence.
- **Process:** Extremely computationally expensive, requiring thousands of GPUs for weeks or months. This stage is typically performed only by large research labs and corporations (e.g., OpenAI, Google, Meta).

1.2. Fine-tuning: Specializing the Model

In the **fine-tuning phase**, a pre-trained model is further trained on a smaller, task-specific, labeled dataset. This adapts the model's general capabilities to a particular domain or task.

- **Objective:** Adapt the general model to a specific task (e.g., summarization, code generation, customer support chat) or domain (e.g., legal, medical, finance).
- **Data:** Smaller, curated, and labeled dataset (thousands to hundreds of thousands of examples).
- **Knowledge Injected:** Task-specific or domain-specific knowledge.
- **Process:** Far less computationally expensive than pre-training, feasible for smaller organizations and individuals.

Essentially, pre-training creates a knowledgeable "generalist," while fine-tuning molds it into a "specialist." For most applications, **fine-tuning is the recommended approach** to teach a model about a new domain, as it's far more efficient and leverages the powerful foundation built during pre-training.

2. Core Fine-Tuning Strategies

2.1. Supervised Fine-Tuning (SFT) and Instruction Tuning

Supervised Fine-Tuning (SFT) is the process of updating a model's weights using a labeled dataset. A highly effective form of SFT is **Instruction Tuning**, where the model is trained on examples of `(instruction, response)` pairs.

- **Benefit:** Instruction tuning teaches the model to follow user commands and behave in a helpful, conversational manner. It aligns the model's capabilities with human intent.
- **Data Format:** The data is typically structured as a prompt that includes the instruction, and a desired completion. For example, in JSON format:

```
[
  {
    "instruction": "Summarize the following text.",
    "input": "The Treaty of Westphalia was a series of peace treaties signed be
    "output": "The Treaty of Westphalia in 1648 ended the Thirty Years' War and
  },
  {
    "instruction": "Translate 'Hello, world!' to French.",
    "input": "",
    "output": "Bonjour, le monde!"
  }
]
```

- **Impact:** This is the primary method used to transform a raw "base" model into a helpful "chat" or "instruct" model, significantly improving its usability and controllability.

2.2. Continuous Pre-training for Domain Adaptation

If a target domain has a large amount of unlabeled text and a vocabulary that significantly differs from the general pre-training data, an intermediate step of **Continuous Pre-training** can be beneficial before SFT.

- **Process:** Continue the self-supervised pre-training objective (e.g., next-token prediction) on a large corpus of domain-specific unlabeled text (e.g., a company's internal wiki, legal case documents, or medical research papers).
- **Goal:** Familiarize the model with the domain's specific vocabulary, jargon, and stylistic patterns.
- **When to Use:** When the domain is highly specialized and you have access to a large corpus of unlabeled domain text.
- **Outcome:** This can improve the final performance of the SFT step, as the model already understands the domain's language.

3. The Challenge of Catastrophic Forgetting

When a model is fine-tuned on a new task or domain, it risks degrading its performance on the original tasks or its general capabilities. This phenomenon is known as **Catastrophic**

Forgetting.

Why it happens:

- **Data Distribution Shift:** The fine-tuning dataset has a different statistical distribution from the massive pre-training dataset. The model's parameters shift significantly to minimize loss on this new, narrow distribution, effectively "forgetting" the patterns learned from the general data.
- **Parameter Update Conflict:** The gradients calculated for the new task may overwrite the parameters that were crucial for general knowledge.

This is often the reason why a fine-tuned model might feel "dumber" or less creative—it has over-specialized on the SFT data, losing some of its general reasoning or conversational ability.

Mitigation Strategies:

1. **Mixing General Data with SFT Data:** During SFT, include a small percentage (e.g., 5-10%) of general-purpose instruction data along with your domain-specific data. This forces the model to maintain its general abilities.
2. **Experience Replay:** Store a subset of samples from old tasks (or general data) and "replay" them by mixing them into the training batches for the new task.
3. **Elastic Weight Consolidation (EWC):** This technique adds a regularization term to the loss function that penalizes large changes to weights deemed important for previous tasks. The importance of a weight is measured by the Fisher Information Matrix. The modified loss function for a new task B, after learning task A, is:

$$\mathcal{L}_B(\theta) = \mathcal{L}_B(\theta) + \frac{\lambda}{2} \sum_i F_i (\theta_i - \theta_{A,i}^*)^2$$

   Where:
   - $\mathcal{L}_B(\theta)$ is the loss for the new task B.
   - λ is a hyperparameter controlling the importance of the old task.
   - F_i is the diagonal of the Fisher Information Matrix, signifying the importance of parameter i for task A.
   - $\theta_{A,i}^*$ is the optimal parameter value for task A.
4. **Parameter-Efficient Fine-Tuning (PEFT):** Methods like LoRA (see below)

implicitly mitigate catastrophic forgetting. By freezing the vast majority of the original model's weights and only training a small number of new parameters, the core knowledge of the base model is preserved by design.

4. Parameter-Efficient Fine-Tuning (PEFT)

Full fine-tuning requires updating all model parameters, which is computationally expensive and memory-intensive. PEFT methods offer a solution by freezing the pre-trained model weights and training only a small number of additional parameters.

4.1. LoRA (Low-Rank Adaptation)

LoRA is the most popular PEFT technique. It is based on the hypothesis that the change in weights during fine-tuning has a low "intrinsic rank."

- **Core Idea:** Instead of updating the original weight matrix $W_0 \in \mathbb{R}^{d \times k}$, LoRA learns a low-rank decomposition of the weight update ΔW. It represents ΔW as the product of two smaller matrices, $B \in \mathbb{R}^{d \times r}$ and $A \in \mathbb{R}^{r \times k}$, where the rank $r \ll \min(d, k)$.
- **Forward Pass:** The modified forward pass becomes:

$$h = W_0 x + \Delta W x = W_0 x + BAx$$

- **Benefits:**
    - **Memory Efficiency:** Only the matrices A and B need to be trained and stored. For a 7B parameter model, this can reduce trainable parameters from 7 billion to just a few million.
    - **No Inference Latency:** After training, the product BA can be merged back into the original weight matrix W_0 ($W' = W_0 + BA$), so there is no extra latency during inference.
    - **Task Switching:** You can have multiple lightweight LoRA adapters for different tasks and swap them out as needed without reloading the entire base model.

4.2. QLoRA (Quantized LoRA)

QLoRA further reduces memory usage by quantizing the base model to a lower precision (typically 4-bit) while performing the LoRA fine-tuning.

- **Key Innovations:**
    i. **4-bit NormalFloat (NF4):** A new data type that is information-theoretically optimal for normally distributed weights.
    ii. **Double Quantization:** Quantizes the quantization constants themselves to save additional memory.
    iii. **Paged Optimizers:** Uses NVIDIA unified memory to offload optimizer states to CPU RAM when the GPU runs out of memory, preventing OOM errors during training.
- **Impact:** QLoRA makes it possible to fine-tune large models (e.g., 33B or 65B parameter models) on a single consumer-grade GPU (like an RTX 3090/4090 with 24GB VRAM).

5. Practical Training and Deployment Considerations

5.1. Estimating VRAM Requirements

For full-parameter fine-tuning, the required VRAM is a sum of several components:

1. **Model Weights:** The memory needed to store the model parameters. For a model with P parameters using 16-bit precision (FP16/BF16), this is $P \times 2$ bytes.
2. **Gradients:** Storing gradients for backpropagation requires the same amount of memory as the model weights ($P \times 2$ bytes).
3. **Optimizer States:** Adam and AdamW optimizers, which are commonly used, store two states (momentum and variance) for each parameter. This typically requires $P \times 8$ bytes (2 states $\times$ 4 bytes/state for FP32).
4. **Activations:** The memory for the intermediate outputs of each layer. This depends on batch size, sequence length, and model architecture. It is often the largest and most dynamic component.

A rough estimation formula for full fine-tuning with Adam is:
$$VRAM \approx (2 \times P) + (2 \times P) + (8 \times P) + \text{Activations} = 12P + \text{Activations}$$

This shows why full fine-tuning is so demanding. A 7B model would require at least $12 \times 7 = 84$ GB of VRAM *before* even accounting for activations.

5.2. Overcoming Out-of-Memory (OOM) Errors

When training results in an OOM error, consider these techniques:

1. **Reduce Batch Size:** The simplest solution. A smaller batch size reduces the memory required for activations. However, this can make training less stable.
2. **Gradient Accumulation:** Process several small batches sequentially and accumulate their gradients before performing a single optimizer step. This simulates a larger batch size without the corresponding memory footprint.
3. **Mixed-Precision Training (AMP):** Use a combination of 32-bit (FP32) and 16-bit (FP16/BF16) precision. It reduces memory usage for weights, gradients, and activations by up to 50% and can speed up training on modern GPUs with Tensor Cores.
4. **Use PEFT:** Switch from full fine-tuning to LoRA or QLoRA. This is the most effective method for drastically reducing memory usage.
5. **Distributed Training:**
    - **Data Parallelism (DP):** Replicate the model on multiple GPUs, split the data batch among them, and average the gradients.
    - **Tensor/Model Parallelism (TP):** Split the model itself (its layers or weight matrices) across multiple GPUs. This is necessary for models that are too large to fit on a single GPU even with a batch size of 1.

5.3. Data Curation and Evaluation

The quality of your fine-tuning is entirely dependent on the quality of your data.

- **Building an SFT Dataset:**
    - Collect raw data relevant to your task.
    - Carefully clean, preprocess, and format it into `(instruction, response)` pairs.
    - Ensure diversity and representativeness. Avoid biases that could lead the model to generate incorrect or harmful content.
- **Building an Evaluation Set:**
    - Create a held-out test set that reflects the real-world scenarios the model will face.
    - This set should be of high quality and should not be seen by the model during training.

    - Define clear evaluation metrics (e.g., accuracy for classification, ROUGE for summarization, or human evaluation for chat quality).

5.4. Model Selection: Base vs. Chat/Instruct Models

- **Base Models:** These are the raw output of the pre-training phase. They are incredibly knowledgeable but are not instruction-followers by default. They are good at "completing" text. Use a base model if you want maximum control and plan to do extensive instruction and safety tuning yourself.
- **Chat/Instruct Models:** These are base models that have already undergone extensive SFT and RLHF (Reinforcement Learning from Human Feedback) to be helpful, harmless, and follow instructions. For most applications, starting with a high-quality chat model (e.g., Llama-3-Instruct, Mistral-Instruct) is more efficient, as it already possesses the desired conversational abilities.

5.5. Vocabulary Expansion

For domains with many unique, out-of-vocabulary (OOV) tokens, you might consider expanding the model's vocabulary.

- **Pros:** Allows the model to learn representations for new domain-specific terms directly, potentially improving performance.
- **Cons:**
    - Increases model size.
    - Breaks compatibility with the original pre-trained weights for the input/output embedding layers, which must be retrained from scratch or carefully initialized. This often requires a significant amount of training data and compute.
- **Recommendation:** This should be a last resort. Modern tokenizers (like BPE) can often represent OOV words as a sequence of subword tokens. Often, continuous pre-training is a better approach to teach the model the meaning of these subword sequences in the new domain context.

Interview Questions

Theoretical Questions

Question 1: Explain catastrophic forgetting. Why does it occur, and how does the Elastic Weight Consolidation (EWC) algorithm attempt to solve it? Please provide the mathematical formulation for the EWC loss term.

Answer:
Catastrophic forgetting is the tendency of an artificial neural network to abruptly and drastically forget previously learned information upon learning new information. It's a common problem in continual learning scenarios where a model is sequentially trained on multiple tasks.

Why it occurs:
It happens because the optimization process (like Stochastic Gradient Descent) adjusts the model's weights to minimize the loss on the *current* task. These adjustments are made without regard for their effect on previously learned tasks. If a weight was critical for Task A, but irrelevant or detrimental for Task B, its value will be changed to optimize for Task B, thereby destroying the knowledge of Task A. This is exacerbated by the high-dimensional and non-convex nature of the loss landscape.

Elastic Weight Consolidation (EWC):
EWC provides a solution by viewing learning as a probabilistic inference problem. It assumes a Bayesian perspective where the model parameters have a prior distribution. After training on Task A, the posterior distribution of the parameters, $p(\theta|\mathcal{D}_A)$, becomes the prior for Task B. To prevent forgetting, we want the parameters to stay close to the optimal solution for Task A, θ_A^*, especially for weights that were important for Task A.

EWC approximates this by adding a quadratic penalty term to the loss function of Task B. The importance of each parameter is measured by the diagonal of the **Fisher Information Matrix (FIM)**, which approximates the posterior's precision. A high Fisher information value for a parameter indicates it is very sensitive and important for the task's output.

Mathematical Formulation:
The loss function for the new task B is modified as follows:

$$\mathcal{L}(\theta) = \mathcal{L}_B(\theta) + \sum_i \frac{\lambda}{2} F_i(\theta_i - \theta^*_{A,i})^2$$

Where:

- $\mathcal{L}_B(\theta)$ is the standard loss function for the new task B.
- λ is a hyperparameter that controls the relative importance of the old task versus the new task.
- $\theta^*_{A,i}$ is the value of the i-th parameter after training on task A (the "anchor" point).
- F_i is the i-th diagonal element of the Fisher Information Matrix, calculated after training on Task A. It acts as a weight-specific learning rate, heavily penalizing changes to important parameters. The FIM is formally the expected value of the squared gradient of the log-likelihood: $F = \mathbb{E}_{x \sim p(x)}[\nabla_\theta \log p(y|x, \theta) \nabla_\theta \log p(y|x, \theta)^T]$.

Question 2: What is the core idea behind LoRA (Low-Rank Adaptation), and why is it so effective for reducing memory usage during fine-tuning?

Answer:
The core idea behind LoRA is the hypothesis that the update to a pre-trained model's weight matrix during fine-tuning has a low "intrinsic rank." This means that even though the weight matrix W and its update ΔW are full-rank, high-dimensional matrices, the change (ΔW) can be effectively approximated by a much lower-rank matrix.

Instead of directly training the large ΔW matrix, LoRA decomposes it into the product of two much smaller, low-rank matrices: A and B.

$$\Delta W = B \cdot A$$

Where if $W \in \mathbb{R}^{d \times k}$, then $B \in \mathbb{R}^{d \times r}$ and $A \in \mathbb{R}^{r \times k}$. The rank r is a hyperparameter and is chosen to be much smaller than d and k (e.g., $r = 8, 16, 32$).

Effectiveness for Memory Reduction:

1. **Drastically Fewer Trainable Parameters:** The original weights W_0 of the LLM are frozen. The only parameters that are trained are those in matrices A and B.

The number of parameters in A and B is $r \times k + d \times r = r(d+k)$. In a full fine-tuning scenario, we would train $d \times k$ parameters. Since $r \ll d, k$, the number of trainable parameters is reduced by orders of magnitude.

2. **Reduced Optimizer State Memory:** Optimizers like Adam store states (momentum and variance) for each trainable parameter. Since LoRA has far fewer trainable parameters, the memory required for these optimizer states is proportionally reduced. This is often the largest source of memory savings.
3. **No Gradient Memory for Base Model:** Since the base model parameters are frozen, there is no need to store their gradients, saving a significant amount of VRAM.

This combined effect allows for fine-tuning very large models on hardware with limited VRAM.

Question 3: You are tasked with fine-tuning a 70B LLM but are constantly hitting Out-of-Memory (OOM) errors on your A100 (80GB) GPU. Describe, in order, the top three techniques you would apply to solve this problem.

Answer:
Given a 70B model and an 80GB GPU, full fine-tuning is impossible. The first priority is to drastically reduce the memory footprint.

1. **Implement QLoRA (Quantized LoRA):** This is the single most impactful step.
    - **Why:** QLoRA combines the parameter efficiency of LoRA with 4-bit quantization of the base model. The base model's weights, which are the largest component, would be loaded in 4-bit precision instead of 16-bit, reducing their memory footprint by 75% (from ~140GB to ~35GB). The LoRA adapters add only a few million trainable parameters, keeping the memory for gradients and optimizer states minimal. This step alone will likely solve the OOM issue and fit the model in memory.
2. **Enable Gradient Accumulation:**
    - **Why:** Even with QLoRA, memory usage scales with batch size due to activations. If I still need to use a batch size that is too large for memory (e.g., to maintain training stability), I would use gradient accumulation. By setting a small per-device batch size (e.g., 1 or 2) and accumulating gradients over several steps (e.g., 8 or 16 steps), I can simulate a much larger effective batch size without the

corresponding memory cost for activations.

3. **Use Paged Optimizers:**
    - **Why:** This is a feature often used with QLoRA via libraries like `bitsandbytes`. If the GPU VRAM is still under pressure from optimizer states during gradient updates, a paged optimizer (like paged AdamW) will automatically offload the optimizer states to the much larger CPU RAM when the GPU memory is full and page them back when needed. This prevents OOM crashes during the optimizer step, trading a bit of speed for stability.

Question 4: What is the difference between a "Base" LLM and a "Chat/Instruct" LLM? For the task of summarizing legal documents into a specific JSON format, which would you start with and why?

Answer:

- **Base LLM:** This is the model resulting from the initial pre-training phase. It excels at text completion and has a vast amount of world knowledge, but it is not inherently conversational or instruction-following. It predicts the most statistically likely next token based on the input text.
- **Chat/Instruct LLM:** This is a base model that has undergone further alignment tuning, typically using a combination of Supervised Fine-Tuning (SFT) on instruction-response pairs and Reinforcement Learning from Human Feedback (RLHF). This tuning makes the model adept at following user commands, engaging in dialogue, and adhering to safety guidelines.

Which to choose for summarizing legal documents into JSON?

I would choose the **Chat/Instruct model** as the starting point.

Reasoning:

1. **Instruction Following:** The task is fundamentally an instruction: "Summarize this legal document into this specific JSON format." An instruct-tuned model is already optimized to understand and execute such commands. A base model would likely just continue the legal text or produce an unstructured summary.
2. **Formatting Capability:** Chat models are extensively trained on data that includes structured formats like Markdown, code, and JSON. They are generally better at

reliably generating well-formed structured output than base models.

3. **Efficiency:** Starting with an instruct model means I can focus my fine-tuning effort on teaching the model the *specifics* of legal language and my target JSON schema, rather than spending time and data teaching it the basic concept of following instructions in the first place. The alignment work has already been done, saving significant development time.

Practical & Coding Questions

Question 1: Implement the core logic of a LoRA layer in PyTorch. The layer should take a standard `nn.Linear` layer and apply the LoRA logic to it.

Answer:

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

class LoRALayer(nn.Module):
    """
    Implements a LoRA (Low-Rank Adaptation) layer that wraps a standard nn.Linear layer.
    """
    def __init__(
        self,
        original_layer: nn.Linear,
        rank: int,
        lora_alpha: float = 1.0,
    ):
        super().__init__()
        self.rank = rank
        self.lora_alpha = lora_alpha

        # The original pre-trained layer is frozen
        self.original_layer = original_layer
        for param in self.original_layer.parameters():
            param.requires_grad = False

        in_features, out_features = original_layer.in_features, original_layer.out_feature

        # Create the low-rank matrices A and B
        self.lora_B = nn.Parameter(torch.zeros(out_features, rank))
        self.lora_A = nn.Parameter(torch.zeros(rank, in_features))

        # Scaling factor
        self.scaling = self.lora_alpha / self.rank

        # Initialize the weights
        # A is initialized with Kaiming uniform, B is initialized to zero
        nn.init.kaiming_uniform_(self.lora_A, a=math.sqrt(5))
        nn.init.zeros_(self.lora_B)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """
```

```
        Forward pass combines the original layer's output with the LoRA update.
        """
        # Original layer forward pass (with frozen weights)
        original_output = self.original_layer(x)

        # LoRA path forward pass
        lora_update = (self.lora_B @ self.lora_A) * self.scaling
        lora_output = x @ lora_update.T

        return original_output + lora_output

# --- Example Usage ---
# 1. Define a standard linear layer (simulating a pre-trained layer)
input_dim = 128
output_dim = 256
pretrained_linear = nn.Linear(input_dim, output_dim)

# 2. Create the LoRA wrapper around it
lora_rank = 8
lora_alpha = 16
lora_linear_layer = LoRALayer(
    original_layer=pretrained_linear,
    rank=lora_rank,
    lora_alpha=lora_alpha
)

# 3. Check trainable parameters
trainable_params = sum(p.numel() for p in lora_linear_layer.parameters() if p.requires_gra
total_params = sum(p.numel() for p in lora_linear_layer.parameters())
original_params = sum(p.numel() for p in pretrained_linear.parameters())

print(f"Original Linear Layer Parameters: {original_params}")
print(f"Total LoRA Layer Parameters: {total_params}")
print(f"Trainable LoRA Parameters (A and B): {trainable_params}")
print(f"Expected Trainable Params: {lora_rank * input_dim + lora_rank * output_dim}")

# 4. Dummy forward pass
input_tensor = torch.randn(10, input_dim) # Batch size of 10
output_tensor = lora_linear_layer(input_tensor)
```

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

Question 2: Write a Python function using only NumPy to demonstrate catastrophic forgetting. Train a simple linear model on Task A, then on Task B, and show that its performance on Task A degrades significantly.

Answer:

This code demonstrates catastrophic forgetting with a simple linear regression model.

```
import numpy as np
import matplotlib.pyplot as plt

# --- Helper Functions ---
def generate_data(n_samples, slope, intercept, noise_std=0.1):
    """Generates data for a linear relationship."""
    X = np.random.rand(n_samples, 1) * 10
    y = slope * X + intercept + np.random.randn(n_samples, 1) * noise_std
    return X, y

def mse_loss(y_true, y_pred):
    """Calculates Mean Squared Error loss."""
    return np.mean((y_true - y_pred)**2)

class SimpleLinearModel:
    """A simple linear model trained with gradient descent."""
    def __init__(self):
        self.weights = np.random.randn(1, 1)
        self.bias = np.random.randn(1)

    def predict(self, X):
        return X @ self.weights + self.bias

    def train(self, X, y, learning_rate=0.001, epochs=100):
        for epoch in range(epochs):
            # Predict and calculate loss
            y_pred = self.predict(X)
            loss = mse_loss(y, y_pred)

            # Calculate gradients
            grad_weights = -2 * np.mean(X * (y - y_pred))
            grad_bias = -2 * np.mean(y - y_pred)

            # Update weights and bias
            self.weights -= learning_rate * grad_weights
            self.bias -= learning_rate * grad_bias

        print(f"Final Loss: {loss:.4f}, Learned Weights: {self.weights[0,0]:.2f}, Bias: {s
```

```python
# --- Demonstration ---

# 1. Define two different tasks
# Task A: y = 2x + 5
X_a, y_a = generate_data(100, slope=2, intercept=5)
# Task B: y = -3x + 10
X_b, y_b = generate_data(100, slope=-3, intercept=10)

# 2. Initialize the model
model = SimpleLinearModel()

# 3. Train on Task A
print("--- Training on Task A ---")
model.train(X_a, y_a, epochs=200)

# 4. Evaluate performance on Task A (should be good)
preds_a_after_a = model.predict(X_a)
loss_a_after_a = mse_loss(y_a, preds_a_after_a)
print(f"\nLoss on Task A after training on Task A: {loss_a_after_a:.4f}\n")

# 5. Train the *same* model on Task B
print("--- Training on Task B (Forgetting Task A) ---")
model.train(X_b, y_b, epochs=200)

# 6. Evaluate performance on Task A again (should be bad)
preds_a_after_b = model.predict(X_a)
loss_a_after_b = mse_loss(y_a, preds_a_after_b)
print(f"\nLoss on Task A after training on Task B: {loss_a_after_b:.4f} <--- CATASTROPHIC

# --- Visualization ---
plt.figure(figsize=(12, 6))
plt.scatter(X_a, y_a, label='Task A Data', color='blue', alpha=0.6)
plt.scatter(X_b, y_b, label='Task B Data', color='red', alpha=0.6)

# Plot model trained on A
plt.plot(X_a, preds_a_after_a, 'b-', label='Model after Task A training')
# Plot model after it was retrained on B
plt.plot(X_a, preds_a_after_b, 'r--', label='Model after Task B training (predictions on A
```

```
plt.title('Demonstration of Catastrophic Forgetting')
plt.xlabel('X')
plt.ylabel('y')
plt.legend()
plt.grid(True)
plt.show()

print(f"The loss on Task A increased from {loss_a_after_a:.4f} to {loss_a_after_b:.4f} af
```

Question 3: Using `matplotlib`, create a visualization that explains the concept of L2 Regularization. Plot the loss contours and the L2 penalty region to show why the optimal solution tends to have smaller weights.

Answer:

This code will generate a plot showing the unregularized loss function (as elliptical contours) and the L2 regularization constraint (as a circle). The optimal regularized solution is where the contours first "touch" the circle, which pulls the solution closer to the origin (smaller weights) compared to the unconstrained minimum.

```python
import numpy as np
import matplotlib.pyplot as plt

# Define a simple quadratic loss function (e.g., from linear regression)
# Loss = (w1 - 2)^2 + (w2 - 3)^2
def loss_function(w1, w2):
    return (w1 - 2)**2 + (w2 - 3)**2

# Create a grid of weight values
w1 = np.linspace(-1, 5, 400)
w2 = np.linspace(-1, 5, 400)
W1, W2 = np.meshgrid(w1, w2)
Z = loss_function(W1, W2)

# Define the L2 penalty region
# w1^2 + w2^2 <= C (a circle)
circle_angles = np.linspace(0, 2 * np.pi, 100)
radius = 1.5 # Example radius for the constraint circle
circle_w1 = radius * np.cos(circle_angles)
circle_w2 = radius * np.sin(circle_angles)

# Find the L2 regularized solution
# This is where the loss contour just touches the L2 circle.
# Analytically, it's on the line from the origin to the unregularized minimum.
w_unreg = np.array([2, 3])
w_reg = w_unreg * radius / np.linalg.norm(w_unreg)


# Plotting
plt.style.use('seaborn-v0_8-whitegrid')
fig, ax = plt.subplots(figsize=(8, 8))

# 1. Plot the loss contours
contour = ax.contour(W1, W2, Z, levels=np.logspace(0, 3, 10), cmap='viridis_r')
ax.clabel(contour, inline=True, fontsize=8)
ax.set_xlabel('$w_1$ (Weight 1)')
ax.set_ylabel('$w_2$ (Weight 2)')
ax.set_title('L2 Regularization: Finding the Optimal Weights', fontsize=14)
```

```
# 2. Plot the L2 regularization constraint region
ax.plot(circle_w1, circle_w2, 'r-', linewidth=2.5, label='L2 Constraint ($w_1^2 + w_2^2 \
ax.fill(circle_w1, circle_w2, 'red', alpha=0.1)


# 3. Mark the optimal points
# Unregularized minimum
ax.plot(2, 3, 'bo', markersize=10, label='Unregularized Minimum ($w_{unreg}$)')
ax.text(2.1, 3.1, '$w_{unreg}$', color='blue', fontsize=12)

# Regularized minimum
ax.plot(w_reg[0], w_reg[1], 'ro', markersize=10, label='L2 Regularized Minimum ($w_{reg}$
ax.text(w_reg[0] - 0.5, w_reg[1] - 0.2, '$w_{reg}$', color='red', fontsize=12)

# Add an arrow to show the "pull" towards the origin
ax.arrow(2, 3, w_reg[0] - 2, w_reg[1] - 3, head_width=0.1, head_length=0.15, fc='black', e
ax.text(1.5, 2.2, 'Pulled towards origin', style='italic', fontsize=10)


ax.axhline(0, color='grey', linewidth=0.5)
ax.axvline(0, color='grey', linewidth=0.5)
ax.set_aspect('equal', adjustable='box')
ax.legend(loc='upper left')
plt.show()
```