A Comprehensive Guide to Parameter-Efficient Fine-Tuning (PEFT) for Large Language Models

This guide provides an in-depth exploration of Parameter-Efficient Fine-Tuning (PEFT) techniques, which are crucial for adapting large pre-trained models (LMs) to specific downstream tasks without incurring the prohibitive computational and storage costs of full fine-tuning. We will cover the motivation behind PEFT and delve into the specific mechanisms of key methods, including additive approaches like Adapter Tuning and Prefix/Prompt Tuning, selective methods like BitFit, and reparameterization techniques such as the highly influential Low-Rank Adaptation (LoRA) and its advanced variants, AdaLoRA and QLoRA. The guide concludes with a comparative analysis and a set of theoretical and practical interview questions to solidify understanding and prepare for technical interviews.

Knowledge Section

The "Why": The Need for Parameter-Efficient Fine-Tuning

The advent of large-scale pre-trained models like GPT-3, Llama, and BERT has revolutionized Natural Language Processing. However, their immense size (billions of parameters) presents significant challenges for adaptation to specific tasks:

1. **Prohibitive Fine-Tuning Costs:** Full fine-tuning, which involves updating all model weights, is computationally expensive. It requires substantial GPU resources (e.g., multiple A100s) and time, making it inaccessible for many researchers and organizations.
2. **Massive Storage Requirements:** Storing a unique, fully fine-tuned copy of a 100B+ parameter model for every single downstream task is impractical and inefficient.
3. **Catastrophic Forgetting:** When a model is fully fine-tuned on a new task, it often loses some of its powerful, generalizable knowledge acquired during pre-training. This phenomenon is known as catastrophic forgetting.

4. **Deployment Challenges:** Managing and serving dozens of unique, multi-billion parameter models in a production environment is a major operational hurdle.

PEFT methods address these challenges by freezing the vast majority of the pre-trained model's parameters and only tuning a small, targeted subset or adding a small number of new parameters. This approach drastically reduces computational overhead, memory usage, and storage costs while often achieving performance comparable to full fine-tuning.

Categories of PEFT Methods

PEFT techniques can be broadly classified into three main categories:

1. **Additive Methods:** These methods add new, trainable parameters or modules to the original model architecture. The pre-trained weights remain frozen.
2. **Selective Methods:** These methods freeze most parameters but select a small, specific subset of existing parameters (e.g., bias terms) for fine-tuning.
3. **Reparameterization Methods:** These methods reparameterize the weight matrices of the model, typically using low-rank decompositions, to create a low-dimensional, trainable representation of the weight updates.

---

1. Additive Methods

1.1. Adapter Tuning

Adapter Tuning involves inserting small, trainable modules called "Adapters" within each layer of the pre-trained Transformer model. The original model weights are frozen, and only the parameters of these new adapter layers and the Layer Normalization layers are updated during training.

An Adapter module typically consists of a bottleneck architecture:

1. A feed-forward down-projection layer that reduces the feature dimension.
2. A non-linear activation function (e.g., ReLU or GeLU).
3. A feed-forward up-projection layer that restores the original feature dimension.
4. A residual connection that adds the adapter's output to the original input from the Transformer block.

Characteristics:

- **Performance:** Achieves strong performance, often close to full fine-tuning.
- **Inference Overhead:** The added modules introduce a small amount of computational latency during inference, as the data must pass through these extra layers.

Variants:

- **AdapterFusion:** A technique for composing adapters from multiple tasks. It learns a routing mechanism to combine the representations learned by different task-specific adapters, enabling knowledge transfer.
- **AdapterDrop:** A method to improve inference efficiency by dynamically dropping some adapter layers (especially from lower Transformer layers) during inference without a significant drop in performance.

1.2. Prefix-Tuning

Prefix-Tuning prepends a sequence of continuous, task-specific vectors (or "virtual tokens") to the keys and values of the Multi-Head Attention mechanism in every Transformer layer. These prefix vectors are trainable, while the rest of the LM is frozen.

Characteristics:

- **No Architectural Change:** It does not alter the core architecture of the pre-trained model.
- **Computational Overhead:** The added prefix tokens increase the effective sequence length, leading to some additional computational cost in the attention mechanism.
- **Expressive Power:** By influencing the attention mechanism at every layer, it can effectively steer the model's behavior for the target task.

1.3. Prompt Tuning & P-Tuning

These methods are simplifications or variations of Prefix-Tuning.

- **Prompt Tuning:** This is the simplest approach. It only prepends trainable "virtual tokens" to the input embedding layer. These soft prompts are learned end-to-end and condition the frozen model to perform a specific task.

        - **Characteristics:** Extremely parameter-efficient (minimal new parameters). It works well for simple NLU tasks but may underperform on complex sequence labeling tasks.
- **P-Tuning:** An enhancement over Prompt Tuning. It also introduces trainable virtual tokens but uses a small neural network (e.g., an MLP+LSTM) called a `prompt_encoder` to model dependencies between these tokens, making them more expressive. The tokens can also be inserted at various positions, not just as a prefix.
    - **Characteristics:** The prompt encoder helps stabilize training and improves performance over basic Prompt Tuning.
- **P-Tuning v2:** A significant improvement that essentially makes P-Tuning behave like Prefix-Tuning. It applies trainable prompts (virtual tokens) to every layer of the model, not just the input. This "deep prompt tuning" gives it much more control over the model's internal representations.
    - **Characteristics:** Resolves the instability and performance issues of Prompt Tuning on smaller models and is highly effective for complex tasks like sequence labeling. It removes the complex `prompt_encoder` (e.g., LSTM) used in P-Tuning v1.

2. Selective Methods

2.1. BitFit (Bias-term Fine-tuning)

BitFit is an extremely simple and sparse fine-tuning method. It involves freezing all of the model's weights and only training the **bias** terms of the network (and the task-specific classification head). The underlying hypothesis is that modifying the bias terms is sufficient to adapt the model's representations to a new data distribution without fundamentally changing its core knowledge.

Characteristics:

- **Minimal Parameters:** Trains an extremely small fraction of the total parameters (often <0.1%).
- **Performance:** While surprisingly effective, its performance generally lags behind more comprehensive methods like LoRA or Adapter Tuning on most benchmarks.
- **Simplicity:** Trivial to implement.

3. Reparameterization Methods

3.1. LoRA (Low-Rank Adaptation)

LoRA is one of the most popular and effective PEFT methods. It is based on the hypothesis that the change in weights during model adaptation has a low "intrinsic rank." Therefore, instead of updating the full weight matrix W, LoRA models the *update* ΔW with a low-rank decomposition.

For a pre-trained weight matrix $W_0 \in \mathbb{R}^{d \times k}$, the update is represented by two smaller matrices, $B \in \mathbb{R}^{d \times r}$ and $A \in \mathbb{R}^{r \times k}$, where the rank $r \ll \min(d, k)$. The forward pass is modified as:

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

During training, W_0 is frozen, and only A and B are trainable.

Characteristics:

- **No Inference Latency:** After training, the weight update can be merged directly into the original weights: $W' = W_0 + BA$. The resulting model has the exact same architecture and inference speed as the original pre-trained model. This is a major advantage over Adapters.
- **High Efficiency:** Drastically reduces the number of trainable parameters. For example, for a 1024×1024 matrix, instead of training $1M$ parameters, LoRA with $r = 8$ only trains $2 \times 1024 \times 8 \approx 16K$ parameters.
- **Portability:** The trained LoRA weights (A and B) are very small and can be easily shared and "plugged into" the base model for different tasks.

3.2. AdaLoRA (Adaptive Low-Rank Adaptation)

AdaLoRA improves upon LoRA by adaptively allocating the parameter budget among weight matrices based on their importance. Instead of using a fixed rank r for all matrices, AdaLoRA dynamically adjusts the rank. It parameterizes the update ΔW as a product of three matrices, $\Delta W = P\Lambda Q$, which is based on Singular Value Decomposition (SVD). It then prunes the least important singular values in Λ during training, effectively reducing the rank for less important weight updates and preserving it for more critical ones.

Characteristics:

- **Efficient Budget Allocation:** More intelligently distributes the limited parameter budget, leading to better performance than LoRA for the same parameter count.
- **Complexity:** The importance scoring and pruning mechanism adds complexity compared to the simplicity of LoRA.

3.3. QLoRA (Quantized Low-Rank Adaptation)

QLoRA is a breakthrough technique that makes it possible to fine-tune massive models (e.g., 65B parameters) on a single consumer-grade GPU (like a 24GB RTX 4090). It combines a novel quantization strategy with LoRA.

The key innovations are:

1. **4-bit NormalFloat (NF4):** A new data type that is information-theoretically optimal for quantizing normally distributed weights. The base model's weights are quantized to 4-bit NF4 and stored in GPU memory.
2. **Double Quantization:** A technique to further reduce the memory footprint by quantizing the quantization constants themselves.
3. **Paged Optimizers:** Leverages NVIDIA unified memory to avoid out-of-memory errors during training by automatically paging optimizer states between CPU RAM and GPU VRAM when GPU memory is scarce.

In QLoRA, the 4-bit base model is frozen, and LoRA adapters are added on top. During the forward and backward passes, the 4-bit base model weights are de-quantized to a higher precision (e.g., BFloat16) on the fly, just in time for computation. This ensures that training quality is maintained while achieving massive memory savings.

Characteristics:

- **Unprecedented Memory Savings:** Democratizes the fine-tuning of very large models.
- **Slower Training:** The de-quantization step on-the-fly introduces a computational overhead, making QLoRA training slower than standard LoRA training if memory is not a constraint.

4. Hybrid and Unified Methods

4.1. MAM Adapter

This method seeks to find a middle ground and unify different PEFT approaches. It analyzes the mathematical relationship between Adapters, Prefix Tuning, and LoRA. The resulting `MAM Adapter` (Mixture-of-Attention-and-MLP) model is effectively a combination of a parallel Adapter for the Feed-Forward Network (FFN) and a form of soft prompt (Prefix-Tuning).

Characteristics:

- **Synergy:** Often outperforms individual PEFT methods by combining their strengths.

4.2. UniPELT

Instead of choosing one PEFT method, UniPELT combines several of them (LoRA, Prefix Tuning, and Adapters) as submodules within a single framework. It introduces a learnable gating mechanism that determines which PEFT submodule(s) to activate for a given input or task.

Characteristics:

- **Flexibility and Robustness:** The gating mechanism allows the model to dynamically choose the best adaptation strategy, leading to improved effectiveness and robustness across a wider range of tasks.
- **Increased Complexity:** It has more trainable parameters and higher inference latency than a single PEFT method due to the multiple submodules and gating mechanism.

Comparative Analysis of PEFT Methods

The following tables, adapted from the source material, summarize the key characteristics of these methods.

Table 1: Qualitative Comparison of PEFT Methods

Method	Type	Storage Efficient	Memory Efficient (Training)	Computationally Efficient (Backward Pass)	Computationa Efficient (Inference)
BitFit	Selective	✅ Yes	✅ Yes	✅ Yes	❌ No (no change)
Prefix-Tuning	Additive (Prompt)	✅ Yes	✅ Yes	✅ Yes	🔻 Minor Overhead
Prompt-Tuning	Additive (Prompt)	✅ Yes	✅ Yes	✅ Yes	🔻 Minor Overhead
Adapter	Additive (Module)	✅ Yes	✅ Yes	✅ Yes	🔻 Minor Overhead
LoRA	Reparameterization	✅ Yes	✅ Yes	✅ Yes	✅ Yes (no overhead after merge)
QLoRA	Reparam./Quantization	✅ Yes	✅✅ Very High	✅ Yes	🔻 Overhead (if not merged)
UniPELT	Hybrid	✅ Yes	✅ Yes	✅ Yes	🔻🔻 Significan Overhead

Figure 1: A qualitative comparison of various Parameter-Efficient Fine-Tuning methods across different efficiency dimensions.

Table 2: Quantitative Comparison of Parameter Usage

Method	Trainable Parameters (delta)	Modified Parameters	Evaluated Model Scale
BitFit	~0.1%	Bias Terms Only	< 1B
Prompt-Tuning	< 0.1%	New Prompt Embeds	<1B, <20B, >20B

Method	Trainable Parameters (delta)	Modified Parameters	Evaluated Model Scale
Prefix-Tuning	~0.1%	New Prefixes	<1B, <20B, >20B
P-Tuning v2	~0.1% - 1%	New Prompts	<1B, <20B
Adapter	~1% - 5%	New Modules	<1B, <20B
LoRA	~0.1% - 1%	New Matrices (A, B)	<1B, <20B, >20B
QLoRA	~0.1% - 1%	New Matrices (A, B)	>20B (e.g., 65B)
UniPELT	~1% - 5%	Multiple New Modules	<20B

Figure 2: A comparison of trainable parameter counts and evaluated model scales for prominent PEFT methods.

Summary and Best Practices

- For **maximum performance and versatility**, **P-Tuning v2** and **LoRA** are excellent, well-balanced choices.
- For **severely limited GPU memory**, **QLoRA** is the state-of-the-art solution, enabling fine-tuning of enormous models.
- For **zero inference latency**, **LoRA** is the ideal choice due to its ability to merge the adapter weights back into the base model.
- For **simple NLU tasks** where efficiency is paramount, **Prompt Tuning** can be a sufficient and highly economical option.

Interview Questions

Theoretical Questions

Question 1: What is "catastrophic forgetting" in the context of neural networks, and how do Parameter-Efficient Fine-Tuning (PEFT) methods help mitigate it?

Answer:

Catastrophic forgetting is the phenomenon where a neural network, upon being trained on a new task, abruptly and completely loses the knowledge it acquired from a previous task. In the context of LLMs, full fine-tuning on a specific dataset (e.g., medical literature) can degrade the model's general world knowledge and language fluency learned during its initial large-scale pre-training.

PEFT methods mitigate this problem by their core design: **freezing the pre-trained weights**.

1. **Preservation of Core Knowledge:** Since the vast majority (often >99%) of the model's parameters are frozen, the rich, general-purpose representations learned during pre-training are preserved. The fine-tuning process does not overwrite this foundational knowledge.
2. **Task-Specific Adaptation:** Instead of altering the entire model, PEFT methods introduce a small number of new parameters (Adapters, LoRA matrices, Prompts) that are specifically trained to steer the frozen model's behavior for the new task. This is like creating a small "control module" for the task, leaving the main "engine" (the pre-trained model) intact.
3. **Isolation of Changes:** The changes for each task are isolated to these small, separate modules. For LoRA, you can have different `(A, B)` matrix pairs for different tasks. For Adapters, you have different adapter modules. This prevents the learning for Task B from overwriting the adaptations made for Task A.

In essence, PEFT treats pre-trained knowledge as a solid foundation and builds small, task-specific extensions on top of it, rather than remodeling the entire foundation for each new project.

Question 2: Explain the mathematical formulation of LoRA. Why is the rank `r` a critical hyperparameter?

Answer:
The core hypothesis of LoRA is that the update to a weight matrix during fine-tuning, ΔW, has a low intrinsic rank. This means the update can be effectively approximated using a low-rank decomposition.

1. **Formulation:**
   Let the original pre-trained weight matrix be $W_0 \in \mathbb{R}^{d \times k}$. In full fine-tuning, we would update this matrix to $W = W_0 + \Delta W$, where ΔW is a dense matrix with $d \times k$ trainable parameters.
   LoRA proposes to approximate ΔW with two smaller matrices: $B \in \mathbb{R}^{d \times r}$ and $A \in \mathbb{R}^{r \times k}$, where the rank $r \ll \min(d, k)$. The update is thus:

   $$\Delta W = BA$$

   The modified forward pass for a given input x becomes:

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

   During training, W_0 is frozen. The only trainable parameters are in matrices A and B. The number of trainable parameters is $d \times r + r \times k = r(d + k)$, which is significantly smaller than $d \times k$. Matrix A is typically initialized with random Gaussian values, and matrix B is initialized to all zeros, so at the beginning of training, $\Delta W = BA$ is zero, and the model is identical to the pre-trained one.
2. **Criticality of Rank `r`:**
   The rank `r` is a critical hyperparameter because it controls the trade-off between model expressivity (and performance) and parameter efficiency.
   - **Low `r` (e.g., 1, 2, 4):** A very small rank results in the fewest trainable parameters. This is highly efficient but may not provide enough capacity to capture the nuances of the downstream task, potentially leading to underfitting and lower performance. The update matrix ΔW is highly constrained.
   - **High `r` (e.g., 64, 128, 256):** A larger rank increases the number of trainable parameters. This gives the model more capacity to adapt to

the new task and can lead to better performance. However, it diminishes the efficiency benefits of LoRA. If `r` becomes too large, the number of parameters can approach that of full fine-tuning, and the model may be more prone to overfitting the small downstream dataset.

Therefore, `r` acts as a regularization knob. The optimal value for `r` must be found through experimentation and depends on the complexity of the task and the size of the fine-tuning dataset.

Question 3: Compare and contrast QLoRA's 4-bit NormalFloat (NF4) quantization with standard integer quantization. Why is NF4 more suitable for neural network weights?

Answer:

Standard Integer Quantization (e.g., INT8, INT4):

- **Mechanism:** Maps a range of floating-point values to a fixed number of integers. The mapping is typically uniform (linear). For example, values from [-1.0, 1.0] might be linearly mapped to integers [-128, 127].
- **Distribution Assumption:** It implicitly assumes that the values being quantized are uniformly distributed. If the data is not uniform, this method wastes "quantization bins" on ranges with few values and doesn't have enough precision for ranges with many values.

QLoRA's 4-bit NormalFloat (NF4):

- **Mechanism:** NF4 is a non-uniform quantization method specifically designed for data that follows a zero-mean normal distribution, which is a common characteristic of pre-trained neural network weights.
- **Distribution Assumption:** It explicitly assumes a normal distribution. The quantization bins (the representable values) are not evenly spaced. Instead, they are placed to have equal "expected" numbers of values from the normal distribution falling into each bin. This is achieved by setting the quantiles of the normal distribution as the boundaries for the bins.
- **Benefit:** This approach provides higher precision for values near the center of the distribution (around zero), where the majority of weights are concentrated. It allocates less precision to the tail ends of the distribution, where values are rare.

This makes NF4 "information-theoretically optimal" for normally distributed data, as it minimizes the quantization error for the most common weight values.

Conclusion:
NF4 is more suitable for neural network weights because its non-uniform, distribution-aware design better matches the actual distribution of the weights. Standard integer quantization's uniform approach is suboptimal and leads to greater information loss when quantizing normally distributed data, which can harm model performance more significantly. QLoRA's use of NF4 is a key reason why it can achieve such high model performance with extreme 4-bit quantization.

Practical & Coding Questions

Question 1: Implement a `LoRALinear` layer in PyTorch from scratch. Then, show how you would replace a `nn.Linear` layer in a simple model with your `LoRALinear` layer.

Answer:
Here is a complete, runnable Python code solution using PyTorch.

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

class LoRALinear(nn.Module):
    """
    Implements a LoRA (Low-Rank Adaptation) layer as a replacement for nn.Linear.
    """
    def __init__(
        self,
        in_features: int,
        out_features: int,
        rank: int,
        alpha: float = 1.0, # Scaling factor, often set to the same as rank
    ):
        super().__init__()
        self.in_features = in_features
        self.out_features = out_features
        self.rank = rank
        self.alpha = alpha

        # Create the standard linear layer and freeze it
        self.linear = nn.Linear(in_features, out_features)
        self.linear.weight.requires_grad = False

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

        # Scaling factor for the LoRA update
        self.scaling = self.alpha / self.rank

        self.reset_parameters()

    def reset_parameters(self):
        """Initialize the parameters."""
        # Initialize the original linear layer's weights as they would be by default
```

```
        # (This is already done by nn.Linear, but good practice to know)
        self.linear.reset_parameters()

        # Initialize LoRA matrices
        # A is initialized with Kaiming uniform for non-linearity, B is initialized to ze
        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 for the LoRALinear layer.

        Computes: h = Wx + (alpha/r) * B * A * x
        """
        # Original frozen path
        frozen_output = self.linear(x)

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

        return frozen_output + lora_output

    def extra_repr(self):
        return f'in_features={self.in_features}, out_features={self.out_features}, rank={

# --- Example Usage ---

# 1. Define a simple model with standard nn.Linear layers
class SimpleModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.layer1 = nn.Linear(128, 256)
        self.relu = nn.ReLU()
        self.layer2 = nn.Linear(256, 10)

    def forward(self, x):
        x = self.layer1(x)
        x = self.relu(x)
```

```
        x = self.layer2(x)
        return x

# 2. Create an instance of the model
model = SimpleModel()
print("--- Original Model ---")
print(model)
print(f"Total parameters: {sum(p.numel() for p in model.parameters() if p.requires_grad)}"

# 3. Replace a standard nn.Linear layer with our LoRALinear layer
lora_rank = 8
lora_alpha = 8

# Let's replace the second linear layer with our LoRA version
model.layer2 = LoRALinear(
    in_features=model.layer2.in_features,
    out_features=model.layer2.out_features,
    rank=lora_rank,
    alpha=lora_alpha
)

print("\n--- Model with LoRA Layer ---")
print(model)

# 4. Verify which parameters are trainable
trainable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
total_params = sum(p.numel() for p in model.parameters())

print(f"\nTotal parameters in model: {total_params}")
print(f"Trainable parameters (LoRA): {trainable_params}")

# Let's inspect the trainable parameters
print("\nTrainable parameter names:")
for name, param in model.named_parameters():
    if param.requires_grad:
        print(f"  - {name}")

# --- Test forward pass ---
input_tensor = torch.randn(4, 128) # Batch size 4, feature size 128
```

```
output = model(input_tensor)
print(f"\nOutput shape: {output.shape}") # Should be [4, 10]
```

Question 2: You are asked to fine-tune a language model on a new task. How would you design an experiment to find the optimal rank `r` for LoRA? Write a Python script using `matplotlib` to visualize the potential results.

Answer:

To find the optimal rank `r` for LoRA, I would design an experiment that evaluates the model's performance on a validation set across a range of `r` values. The goal is to find the "sweet spot" that maximizes performance without adding too many parameters.

Experimental Design:

1. **Define a Search Space:** Choose a range of `r` values to test. A logarithmic scale is often a good starting point, for example: `r = [1, 2, 4, 8, 16, 32, 64]`.
2. **Split Data:** Divide the task-specific dataset into training, validation, and test sets. The validation set is crucial for hyperparameter tuning.
3. **Training Loop:** For each value of `r` in the search space:
    - Instantiate the model with LoRA layers using the current `r`.
    - Train the model on the training set for a fixed number of epochs or until convergence.
    - After each epoch (or at the end of training), evaluate the model on the validation set using a key metric (e.g., accuracy, F1-score, perplexity).
    - Record the best validation score achieved for that `r`.
    - Also, calculate and record the number of trainable parameters for that `r`.
4. **Analysis and Visualization:** Plot the validation performance against the rank `r`. Also, plot the number of trainable parameters against `r`. This helps visualize the trade-off.
5. **Selection:** Choose the smallest `r` that achieves a performance level close to the maximum observed performance (the "elbow point"). This balances performance and efficiency. Avoid choosing a much larger `r` that gives only a marginal performance gain.
6. **Final Evaluation:** After selecting the best `r`, train the final model on the

combined training and validation sets, and report its performance on the held-out test set.

Python Script for Visualization (Mock Results):

This script simulates the outcome of such an experiment and visualizes it.

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

# --- Mock Data from a simulated experiment ---
# This data represents the hypothetical results of the experiment described above.

ranks = [1, 2, 4, 8, 16, 32, 64]
# Let's assume a hypothetical base model with 7B parameters and LoRA is applied to
# linear layers with average size of 4096x4096.
# Trainable params = r * (in_features + out_features) * num_lora_layers
# This is a rough estimation for illustration.
trainable_params_millions = np.array([0.2, 0.4, 0.8, 1.6, 3.2, 6.4, 12.8])

# Validation accuracy (hypothetical). Performance saturates at higher ranks.
validation_accuracy = [82.5, 85.0, 87.5, 88.8, 89.1, 89.2, 89.25]

# --- Visualization ---

fig, ax1 = plt.subplots(figsize=(12, 7))

# Plotting Validation Accuracy
color = 'tab:blue'
ax1.set_xlabel('LoRA Rank (r)', fontsize=14)
ax1.set_ylabel('Validation Accuracy (%)', color=color, fontsize=14)
ax1.plot(ranks, validation_accuracy, 'o-', color=color, markerfacecolor='white', markersi
ax1.tick_params(axis='y', labelcolor=color)
ax1.grid(True, linestyle='--', alpha=0.6)
ax1.set_xscale('log', base=2) # Use a log scale for ranks as they grow exponentially
ax1.set_xticks(ranks)
ax1.get_xaxis().set_major_formatter(plt.ScalarFormatter())


# Create a second y-axis for the number of trainable parameters
ax2 = ax1.twinx()
color = 'tab:red'
ax2.set_ylabel('Trainable Parameters (Millions)', color=color, fontsize=14)
ax2.plot(ranks, trainable_params_millions, 's--', color=color, markerfacecolor='white', ma
ax2.tick_params(axis='y', labelcolor=color)
```

```python
# Adding a title and legend
plt.title('Finding the Optimal LoRA Rank (r)', fontsize=16, pad=20)
fig.tight_layout()
fig.legend(loc='upper center', bbox_to_anchor=(0.5, 0.95), ncol=2, fontsize=12)

# Highlighting the "elbow point" or "sweet spot"
optimal_r = 8
optimal_idx = ranks.index(optimal_r)
ax1.axvline(x=optimal_r, color='green', linestyle=':', linewidth=2, label=f'Optimal r = {
ax1.plot(ranks[optimal_idx], validation_accuracy[optimal_idx], 'g*', markersize=15, label=

# Add annotation to explain the choice
ax1.annotate(
    'The "Elbow Point":\nBest trade-off between\nperformance and efficiency',
    xy=(ranks[optimal_idx], validation_accuracy[optimal_idx]),
    xytext=(ranks[optimal_idx] * 0.8, validation_accuracy[optimal_idx] - 3),
    arrowprops=dict(facecolor='black', shrink=0.05, width=1, headwidth=8),
    ha='right',
    fontsize=12,
    bbox=dict(boxstyle="round,pad=0.3", fc="yellow", ec="black", lw=1, alpha=0.5)
)


plt.show()
```