**Fine-Tuning Large Language Models Efficiently**

**1. Introduction to Parameter-Efficient Fine-Tuning (PEFT)**

* The lecture focuses on how to fine-tune **Large Language Models (LLMs)** efficiently.
* Specifically, it discusses **Parameter-Efficient Fine-Tuning (PEFT)** techniques.

**2. Transfer Learning Before LLMs**

* Before LLMs, transfer learning was the dominant method in NLP.
* It involved:
  1. **Pre-training phase:**
     + Models were trained on large **unlabeled** corpora.
     + The goal was to develop **world knowledge** by learning **word representations**, **contextual word embeddings**, or **sentence embeddings**.
  2. **Fine-tuning phase:**
     + The pre-trained model was adapted for a **specific task** using **task-specific labeled data**.
     + There were different ways to fine-tune:
       - **Fully fine-tuning** the entire model.
       - **Adding extra layers** to the pre-trained model and fine-tuning only those.

**3. How LLMs Changed the Landscape**

* LLMs **improved world knowledge modeling** beyond traditional pre-trained models.
* A new **Instruction-Tuning & Alignment** phase was introduced:
  + This phase includes **some labeled data** but remains **general-purpose**.
  + The goal is to **enhance world knowledge** rather than fine-tune for specific tasks.

**`4. Emergence of In-Context Learning (ICL)**

* **In-Context Learning (ICL)** allows LLMs to perform tasks **without modifying model parameters**.
* **How it works:**
  + The task information is provided **in the input itself**.
  + Users can feed:
    - **Instructions on how to perform the task**.
    - **Examples of input-output pairs** to guide the model.
* This allowed **many NLP tasks** to be solved with **zero or few-shot prompting**.

**4.1 Why In-Context Learning Became Popular**

* **LLMs are expensive to host** due to high computational requirements.
* Many users **cannot host LLMs** but can **use APIs** to leverage them via **ICL**.
* This **reduced the need for direct fine-tuning** for many applications.

**5. Limitations of In-Context Learning**

Despite its advantages, **ICL is not always sufficient**.  
The key reasons are:

**5.1 Performance Limitations**

* Compared to **fully fine-tuned models**, **ICL often has lower accuracy**.
* If a task **demands high accuracy**, **full fine-tuning** is needed.
* **Two reasons why ICL underperforms:**
  1. **Prompt Engineering Issues**
     + **Poorly designed prompts** lead to **poor results**.
     + Even expert-designed prompts may not match the performance of **fine-tuned models**.
  2. **LLM Assumptions & Hidden Biases**
     + LLMs **make assumptions** about inputs that users cannot always control.
     + Certain **domains or nuances** might be **poorly learned** by the model.

**5.2 Sensitivity to Prompts**

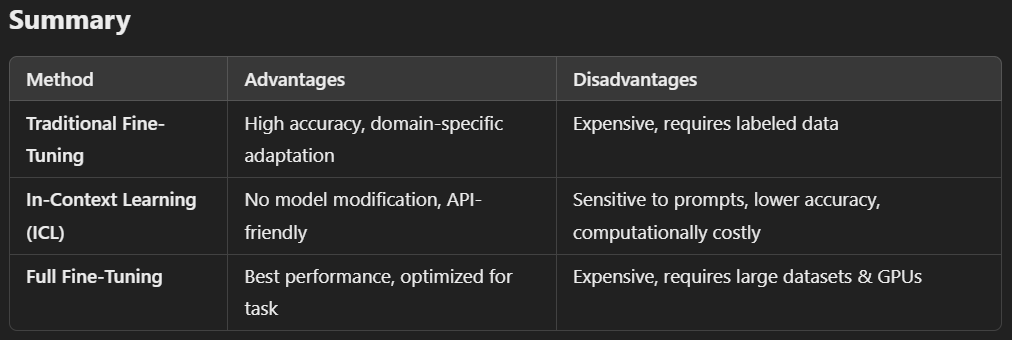
* **LLMs are highly sensitive to small prompt changes**.
* Earlier models **could fail** due to:
  + **Missing a preposition** in the prompt.
  + **Different phrasing** of the same question.
* Even modern LLMs still struggle with **prompt ordering**:
  + **Changing the order of input examples** in the context window **affects accuracy**.
  + Newer LLM versions (e.g., **Llama 3.1 vs. Llama 3**) may interpret prompts **differently**.
  + This means **prompt engineering must be redone** for every new model release.

**5.3 Trust & Interpretability Issues**

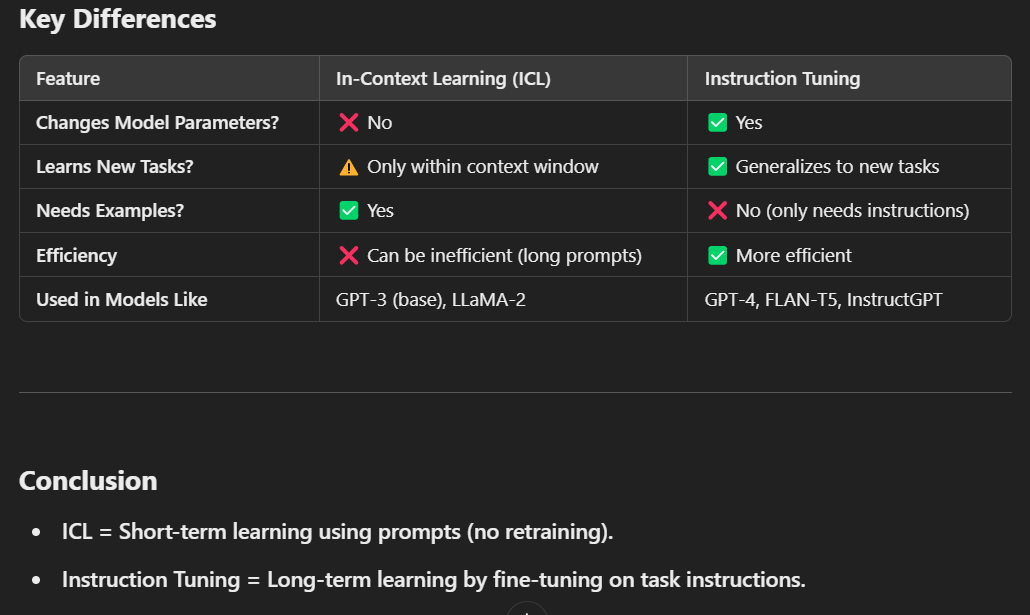
* LLMs do **not always learn what users expect** from in-context examples.
* Problems include:
  + **Unexpected outputs** when prompts are slightly modified.
  + **LLMs learning "spurious patterns"** rather than actual task rules.
  + Users **don’t know when the model will fail**, making it **hard to trust**.

**5.4 Computational Inefficiency**

* **Long prompts waste computational resources**.
* If a task requires **detailed instructions & multiple examples**, the **context window fills up quickly**.
* This **increases latency**, **computational costs**, and **energy consumption**.
* Many **real-world tasks** (e.g., **summarization, reading comprehension**) require handling **long inputs**.
  + Example: **Summarizing different types of news articles** requires **detailed instructions** and **examples for each category**.
  + **ICL becomes inefficient** in such cases.



**Instruction tuning vs in context learning:**



**6. Why Not Just Use Full Fine-Tuning Instead?**

* Given the issues with ICL, full fine-tuning seems like a better option.
* **Next discussion**: How to fine-tune LLMs efficiently using **Parameter-Efficient Fine-Tuning (PEFT)**.

**Fine-Tuning Large Language Models (LLMs) - Detailed Notes**

**1. Why Full Fine-Tuning is Challenging**

**1.1 Memory Constraints**

* Fine-tuning a large LLM like **Llama 8B** requires significant memory.
* If all model weights are in **FP16 (16-bit floating point precision)**, it may require **16GB of memory**.
* However, memory usage isn’t just determined by the number of model parameters.
* Additional memory is needed for:
  + **Optimizer state**
  + **Gradient storage**
  + **Forward activations** (intermediate results during forward propagation)
  + **Temporary memory allocations**
* Different training methods impact memory usage:
  + **Full-precision training** (higher accuracy, more memory)
  + **Mixed-precision training** (compromise between precision and memory usage)
  + **Low-precision training** (reduces memory, possible accuracy loss)
* Typically, full fine-tuning requires about **12-20% of the trainable parameters** in memory.
* Example: Fine-tuning **GPT-3 (175 billion parameters)** is infeasible for most due to extreme memory requirements.

**1.2 Infrastructure Limitations**

* Not all researchers or companies have the **hardware resources** to fully fine-tune an LLM.
* Hosting and fine-tuning LLMs require **high-end GPUs and TPUs**, which are expensive and power-intensive.
* The cost of **storage, compute power, and cooling** makes it impractical for many users.
* Accessibility is reduced, limiting research progress and democratization of AI.

**2. Storage Challenges**

**2.1 Large Checkpoint Sizes**

* Every time the model learns a new task, a **checkpoint** (model snapshot) must be saved.
* Example: A single LLM checkpoint can be **350GB**.
* If each new task requires a separate fine-tuned model, storage becomes overwhelming.
* The serving infrastructure must handle multiple fine-tuned models, increasing complexity.

**2.2 Inefficient Hosting**

* Without **in-context learning**, multiple fine-tuned LLMs must be served separately.
* Example: A company may need **10-15 LLMs** for different applications.
* Hosting multiple models is **computationally expensive** and inefficient.
* Energy consumption increases, making it less sustainable.
* Instead, **in-context learning** allows a **single model to handle multiple tasks**, improving efficiency.

**3. The Overfitting Problem**

**3.1 Data Requirements**

* Even if enough **hardware and storage** is available, fine-tuning is still challenging due to **overfitting**.
* Overfitting occurs when the model memorizes training data instead of learning generalizable patterns.
* Example: A **175B parameter model** trained on only **1,000-2,000 examples** will memorize them completely.
* Training accuracy will be high, but validation performance will suffer, leading to poor generalization.

**3.2 Impact on Transfer Learning**

* Transfer learning is meant to **apply pre-trained knowledge** to new tasks.
* If a model overfits during fine-tuning, the benefits of **transfer learning** are lost.
* To avoid overfitting, **a vast amount of training data** is needed, which is often impractical to collect.

**4. Parameter-Efficient Fine-Tuning (PEFT)**

* Full fine-tuning and in-context learning are two extremes.
* **Parameter-efficient fine-tuning (PEFT)** is a middle-ground approach.
* PEFT modifies only a **small subset of model parameters** instead of all weights.
* Techniques like **LoRA (Low-Rank Adaptation)** and **Adapter Layers** reduce resource requirements.
* This approach retains most pre-trained knowledge while allowing task-specific adaptation.

**Conclusion**

* Full fine-tuning is impractical due to **memory, infrastructure, storage, and overfitting concerns**.
* In-context learning offers a more efficient way to handle multiple tasks without separate fine-tuned models.
* PEFT methods provide a **balance** between full fine-tuning and in-context learning, making LLMs more accessible and scalable.

**Detailed Notes on Parameter-Efficient Fine-Tuning (PEFT)**

**Introduction to Parameter-Efficient Fine-Tuning (PEFT)**

PEFT is an alternative approach to fine-tuning large language models (LLMs) while mitigating the major challenges associated with **full fine-tuning** and **in-context learning (ICL)**. Instead of training all the model’s parameters, PEFT selectively fine-tunes only a small subset, reducing memory usage, computational cost, and storage requirements.

**Why PEFT?**

The need for PEFT arises from the challenges of both **fully fine-tuning** and **in-context learning**:

1. **Challenges with Full Fine-Tuning:**
   * Requires **high memory** (12-20 times the trainable parameters).
   * Needs **huge computational power** (multiple GPUs with high VRAM capacity).
   * Can lead to **overfitting**, especially when training data is limited.
   * Large storage overhead for multiple checkpoints and fine-tuned models.
   * Risks **catastrophic forgetting**, where models forget pre-trained knowledge after learning new tasks.
2. **Challenges with In-Context Learning:**
   * **Higher inference cost**, as prompts must include example demonstrations.
   * Limited capacity for long input sequences due to **context window constraints**.
   * **Task-specific adaptation** is inefficient, as every prompt needs handcrafted examples.

PEFT provides a **middle ground**, leveraging the benefits of both approaches while minimizing their drawbacks.

**How PEFT Works**

PEFT involves **freezing most of the model parameters** and only training a **small subset** for new tasks. This approach maintains the model’s general knowledge while allowing task-specific adaptation.

1. **Task-Specific Incremental Parameters**
   * Instead of modifying the entire model, PEFT introduces **small trainable parameters**.
   * These parameters capture new task-specific knowledge without altering the core pre-trained model.
2. **Examples of PEFT Applications:**
   * Fine-tuning for **question-answering (QA)**
   * Fine-tuning for **text summarization**
   * Fine-tuning for **classification tasks**

**Advantages of PEFT**

**1. Reduced Memory Requirements**

* **Lower optimizer state size**: Since fewer parameters are trainable, the optimizer’s memory footprint decreases.
* **Fewer gradient calculations**: Memory used for storing gradients is significantly reduced.
* **Temporary memory usage decreases**, leading to better GPU utilization.

**2. Faster Convergence**

* Since **most of the model remains frozen**, it retains its pre-trained world knowledge.
* The small subset of trainable parameters **learns task-specific mappings quickly**, leading to faster convergence than full fine-tuning.
* Requires **fewer GPU hours**, making training more cost-effective.

**3. Hardware Efficiency**

* Works on **older GPUs** with limited memory (e.g., NVIDIA V100, Tesla models).
* Reduces dependency on **high-end hardware** like A100 or H100 GPUs.
* Allows **smaller institutions and researchers** to fine-tune models without expensive infrastructure.

**4. Reduced Overfitting**

* Since **not all parameters are modified**, the model retains its ability to generalize.
* Helps **prevent memorization** of limited training data, ensuring better performance on unseen examples.

**5. Avoids Catastrophic Forgetting**

* Traditional fine-tuning risks **overwriting pre-trained knowledge** when learning new tasks.
* PEFT allows models to **retain world knowledge** while adapting to new domains.
* Example: A model fine-tuned for spam detection in **India** can generalize to spam detection in **the U.S.**, despite regional differences in language and spam patterns.

**6. Lower Storage Requirements**

* Unlike full fine-tuning, which requires storing a **full model copy** for each task, PEFT only requires storing **incremental parameters**.
* Saves significant disk space, making deployment easier.

**Comparison: Full Fine-Tuning vs. PEFT vs. In-Context Learning**

| **Feature** | **Full Fine-Tuning** | **In-Context Learning** | **PEFT** |
| --- | --- | --- | --- |
| **Memory Usage** | Very High | Low | Low-Medium |
| **Training Time** | Long | No Training Required | Short |
| **Inference Cost** | Low | High | Low |
| **Generalization** | May Overfit | Generalizes Well | Generalizes Well |
| **Storage Overhead** | High | None | Low |
| **Hardware Requirements** | High-End GPUs | Minimal | Moderate |
| **Risk of Forgetting** | High | No Forgetting | Low |

**Conclusion**

PEFT is an **efficient fine-tuning strategy** that balances the strengths and weaknesses of **full fine-tuning and in-context learning**. By freezing most of the model’s parameters and training only a small subset, PEFT significantly reduces:

* **Memory usage**
* **Compute cost**
* **Storage requirements**
* **Overfitting risks**
* **Catastrophic forgetting**

This technique enables **efficient deployment of LLMs** on **lower-end hardware** while retaining the ability to adapt to **new tasks** with minimal storage overhead.

PEFT represents a crucial development in **scalable and accessible LLM fine-tuning**, ensuring that both research and industry can efficiently **leverage AI advancements** without prohibitive costs.

**Detailed Notes on Parameter Efficient Fine-Tuning Techniques in LLMs**

**1. Introduction to Parameter Efficient Fine-Tuning**

* The lecture dives into popular techniques used for **parameter-efficient fine-tuning** of Large Language Models (LLMs).
* The focus is on methods like **Prompt Tuning**, **Prefix Tuning**, and how they compare to traditional fine-tuning methods.

**1. Types of prompt tuning:**

* **prompt tuning**
* **prefix tuning**
* **adapters**
* **lora**

**2. Prompt Tuning (Soft Prompting)**

* **Hard Prompting vs. Soft Prompting**:
  + **Hard Prompting**: This involves human intervention to write and adjust the prompt for a model, such as rephrasing or adding examples.
  + **Soft Prompting**: A more efficient approach where the model learns a task-specific prompt automatically through fine-tuning. The model doesn't alter its core weights but only learns special "soft" tokens (trainable parameters) that serve as prompts.
* **Key Idea**:
  + In soft prompting, only the **task-specific soft prompt tokens** are trainable, while the rest of the model's architecture remains unchanged (frozen).
  + These soft prompt tokens are inserted into the input and guide the model's behavior for specific tasks.
* **Advantages**:
  + **Efficiency**: Soft prompting requires minimal additional parameters compared to full fine-tuning. For instance, even with a 4K tokenizer size, adding five special tokens is a small increase relative to the size of large LLMs.
  + **Multitask Serving**: When using multiple tasks, different prompts (tokens) can be associated with each task. This allows for efficient **multitask learning** with a single LLM model. During inference, different prompts can be selected based on the specific task at hand.
  + **Scalability**: It's easier to scale up or down the number of instances of the LLM during inference. Since only the soft tokens are modified, there’s no need to run separate instances for different tasks.
* **Comparison to Full Fine-Tuning**:
  + Full fine-tuning involves adjusting all parameters of the model, making it computationally expensive and less efficient.
  + Soft prompting reduces the number of trainable parameters, making it more efficient for tasks that can be adapted with minimal tuning.
* **Model Size and Efficiency**:
  + **For Smaller Models**: Full fine-tuning is often more effective than soft prompting because it provides better accuracy.
  + **For Larger Models**: Soft prompting starts catching up to full fine-tuning in terms of task performance, especially when model size increases.
* **Prompt Length**:
  + Longer prompts (up to 20 tokens) improve task performance by providing more context.
  + However, after a certain point (around 20 tokens for T5 models), further increasing prompt length doesn’t improve results significantly.
* **Initialization of Soft Prompts**:
  + The way soft prompts are initialized is critical. Using random initialization can lead to poor convergence.
  + Better performance is achieved if the initial prompts are chosen to be semantically relevant to the task.
* **Generalization**:
  + **Soft prompting** shows **better generalization** on unseen tasks, particularly when the domain differs from the training data. For example, when tested on a **books dataset** (unseen domain), soft prompting showed an improvement in task performance over full fine-tuning.

**3. Prefix Tuning**

* **Introduction to Prefix Tuning**:
  + Prefix tuning is a **contemporary technique** to soft prompting. It improves upon soft prompting by inserting trainable parameters not only at the input layer but also in every layer of the transformer model.
  + This method introduces small, task-specific prefixes that influence the input at each layer of the network, making it a **more flexible and powerful approach**.
* **How Prefix Tuning Works**:
  + A **prefix** is a sequence of trainable parameters (tokens) inserted at the beginning of each layer in the transformer model. The prefix influences every subsequent token processed by the model.
  + Unlike soft prompts, which only modify the input layer, prefix tuning extends the influence to the entire network, ensuring a deeper impact on model predictions.
* **Key Architecture**:
  + The prefix is inserted at the beginning of the network and influences how each word in the input is processed at every layer of the model.
  + The architecture involves creating trainable embeddings for the prefix tokens, which are then incorporated into the network’s internal layers.
* **Benefits of Prefix Tuning**:
  + **Improved Performance**: Prefix tuning, by influencing each layer of the model, allows it to achieve better results compared to soft prompting.
  + **Efficient Fine-Tuning**: Similar to soft prompting, prefix tuning also reduces the number of parameters required for full fine-tuning, making it highly efficient.
  + **Training Stability**: Using an **MLP (Multilayer Perceptron)** function to process the prefix before feeding it into the model helps stabilize the training process. This prevents issues like exploding or vanishing gradients, which are common with smaller parameter spaces.
* **Challenges and Solutions**:
  + **Unstable Training**: One of the challenges in prefix tuning is the potential for unstable training, especially when parameters are initialized incorrectly.
  + The solution is to use a **smaller embedding size** for the initial prefix embeddings, and later combine them with a larger MLP layer for stability.
  + This approach helps avoid instability, such as the gradients becoming too large or too small, which could disrupt the training process.
* **Performance Evaluation**:
  + Prefix tuning has been tested on tasks like **text-to-text generation** (e.g., summarization and table-to-text generation). With only **0.1% of the model’s parameters** being used for tuning, it can achieve results comparable to full fine-tuning.
  + **Unseen Domains**: Like soft prompting, prefix tuning also generalizes well to tasks from **unseen domains**, improving task performance even on data that differs from the model’s training data.

**Comparison of Prompt Tuning and Prefix Tuning**

* **Parameter Efficiency**: Both methods significantly reduce the number of parameters that need to be tuned compared to full fine-tuning, making them highly parameter-efficient.
* **Training Stability**: Prefix tuning requires careful architecture to avoid unstable training, whereas soft prompting generally has fewer stability concerns.
* **Task Adaptability**: Both techniques allow models to adapt to multiple tasks effectively, with prefix tuning offering slightly more flexibility by influencing every layer of the model.

**3.Adapters in Large Language Models (LLMs)**

**Introduction to Adapters**

* **Definition:**  
  Adapters are a technique used in fine-tuning transformer-based models by introducing small additional layers into each transformer block without modifying the main architecture significantly. This allows the model to adapt to specific tasks with minimal parameters.
* **Origin:**  
  Adapters were first introduced in 2019 as a technique for fine-tuning models for specific tasks. They were initially the most popular approach before other techniques like soft prompting and prefix tuning gained prominence.

**Adapter Architecture**

* **Adapter Layers:**  
  In adapters, new layers called *adapter layers* are added inside each transformer block. These layers consist of:
  + A **down-projection** layer, which reduces the dimensionality of the hidden states (i.e., the hidden size of the transformer model).
  + A **non-linearity** (typically ReLU activation function).
  + An **up-projection** layer, which brings the dimensionality back to the original size of the hidden states.
* **Residual Connections:**  
  Adapters use **residual connections** both within the transformer block (as in standard transformers) and inside the adapter layers.
  + **Why are residual connections important?**  
    These connections help maintain stability in training, especially when new layers are introduced. The residual connection ensures that if the adapter layers are initialized with random weights, the output from the first epoch would still closely resemble the original transformer model without adapters, preserving the original performance.
* **Bottleneck Structure:**  
  The architecture of the adapter is similar to a **bottleneck structure** found in autoencoders. This means that the dimensionality is reduced in the middle of the network (through the down-projection layer) and then increased again (through the up-projection layer).
  + This helps in achieving **parameter efficiency**, significantly reducing the number of parameters compared to a full feed-forward network.
  + **Benefit of Bottleneck Structure:**
    - It reduces the number of parameters and computation, making it more memory efficient.
    - If the original hidden dimension (D) is large (e.g., 1024), the bottleneck structure down-projects it to a much smaller value (e.g., 24). This reduces the parameter count drastically.
    - Example: If a full feed-forward network requires 1024×10241024 \times 10241024×1024 parameters, using a bottleneck with a value mmm of 24 reduces the parameters to 2×1024×242 \times 1024 \times 242×1024×24, which is orders of magnitude lower.

**Advantages and Trade-offs**

* **Parameter Efficiency:**  
  Adapters allow fine-tuning with only a small fraction of the model's parameters. For instance, adapters might require only 3-5% of the parameters compared to full fine-tuning.
* **Stable Training:**  
  The residual connections help ensure that the model doesn’t "forget" the task-specific features learned during the pre-training phase, making training more stable when adding new adapter layers.
* **Hyperparameter Tuning:**  
  The down-projection dimension (value **m**) in the adapter acts as a **hyperparameter** that allows a trade-off between **performance** and **overfitting**. The smaller the bottleneck, the fewer the parameters, but it might impact the performance.

**Challenges and Disadvantages of Adapters**

* **Latency Overhead:**  
  Since adapters modify the architecture by adding new layers in the middle of the transformer blocks, they increase the computation time, which can lead to **increased latency**. This is particularly problematic during inference when fast predictions are needed.
* **Difficulty in Changing Adapters:**  
  Adapters are tightly integrated into the model architecture, making it difficult to swap out different adapter types or configurations. For example, replacing adapter A with adapter B becomes challenging, especially when hosted in production.

**Performance Comparison with Other Techniques**

* **Adapter vs Full Fine-Tuning:**  
  When compared to full fine-tuning, adapters can achieve similar performance with only a small fraction of the parameters. For instance, with only 3.6% of the model’s parameters, adapters can perform as well as fully fine-tuning all layers of a large model.
  + **Graph Comparison:**
    - Fine-tuning the top layers of a model (as shown in the graph) allows efficient training, with performance reaching the optimal point after training only the top layers. Adapters can achieve similar results by tuning only a small fraction of the model’s parameters (around 3%).
* **Adapter vs Prefix Tuning:**  
  Prefix tuning, a more recent technique, is even more parameter-efficient than adapters. Prefix tuning can achieve similar performance with just **0.1% of the parameters** compared to adapters, which require around **3%**.
  + **Recommendation:**  
    Given the efficiency of prefix tuning, it is usually preferred over adapters for certain use cases, especially when parameter efficiency is critical.

**Use Cases and Applicability of Adapters**

* **Task-Specific Fine-Tuning:**  
  Adapters are ideal when a pre-trained model needs to be fine-tuned for a specific task but without altering the main architecture too much. This is useful for adapting large pre-trained models like BERT, GPT, etc., to specialized domains (e.g., medical language processing, legal texts, etc.).
* **Model Size and Parameter Efficiency:**  
  For environments where memory and computation resources are limited, adapters offer a way to fine-tune a model without incurring the computational cost of training an entirely new model. This makes them suitable for edge devices or environments with resource constraints.

**Conclusion**

* **Key Benefits:**
  + Adapter-based fine-tuning allows efficient adaptation of large models with minimal changes to the architecture.
  + By adding small adapter layers, the performance of LLMs can be optimized for specific tasks without requiring full fine-tuning, offering a good balance between performance and parameter efficiency.
* **Trade-offs:**
  + Adapters are not without their challenges, including increased inference latency and the difficulty of dynamically replacing adapters in a live system.
  + Prefix tuning, while more efficient, may offer an advantage in certain contexts over adapters due to its significantly lower parameter requirements.

**4. Detailed Notes on Low-Rank Adaptation (LoRA) and Intrinsic Dimensionality in LLMs**

**1. Introduction to LoRA**

* **LoRA (Low-Rank Adaptation)** is an efficient fine-tuning technique for LLMs.
* It is a well-established method with a strong theoretical foundation rooted in deep learning.
* Implementations of LoRA are readily available in Hugging Face libraries, making it easy to apply to various LLMs.

**2. Background: The Evolution of LoRA**

* LoRA’s development was influenced by research on **low-rank composition** in machine learning models.
* Early studies explored how models could be fine-tuned by modifying only a subset of parameters.
* The key idea was that only a small number of dimensions are needed for specific tasks, which led to the concept of **intrinsic dimensionality**.

**3. The Concept of Intrinsic Dimensionality**

* Intrinsic dimensionality refers to the minimum number of dimensions required to retain **90% of the accuracy** of a fully fine-tuned model.
* It depends on:
  + The **base model's capability** (e.g., larger models require fewer changes for adaptation).
  + The **complexity of the task** (e.g., a simple classification task may require fewer dimensions).
* A key result: **For any given task, a subset of parameters suffices to achieve high accuracy.**

**4. The 2018 Paper on Low-Rank Composition in ML Models**

* Proposed a technique where instead of fine-tuning all parameters, a **random projection matrix** is used.
* The approach involves:
  + Representing the entire set of trainable parameters as **one large vector**.
  + Identifying a smaller subset of dimensions that are sufficient for learning a task.
  + Using a **randomly initialized projection matrix** (not trained) to project from a high-dimensional space to a low-dimensional one.
* Finding: A much smaller **intrinsic dimension (d)** exists for each task, allowing efficient fine-tuning.

**5. Intrinsic Dimensionality in LLMs (2021 Study)**

* While earlier studies treated all model parameters as a **single vector**, LLMs have a **layered structure**.
* The study introduced **structure-aware intrinsic dimensionality**:
  + Instead of considering the model as a single large vector, it is **divided into layers**.
  + Fine-tuning is applied **layer-wise**, reducing computational requirements significantly.
  + A **scaling factor** is introduced to optimize parameter updates for different layers.
* Major challenge: **Memory constraints**—Direct application of intrinsic dimensionality concepts to LLMs requires excessive memory (e.g., fine-tuning BERT requires **1TB** memory).
* Solution: Use layer-wise representation to **reduce projection matrix size**, making fine-tuning feasible.

**6. Scaling Factor in Intrinsic Dimensionality**

* A scaling parameter (Lambda) is introduced to improve fine-tuning efficiency.
* The model is structured as follows:
  + Each layer gets a separate parameter vector (Theta).
  + Scaling factors (Lambda) adjust updates for each layer.
* Benefits:
  + Reduces required memory.
  + Maintains effective performance while using fewer parameters.

**7. Experimental Findings**

* The study found that **larger models require fewer dimensions to adapt to a task**.
* Graphical representation:
  + **X-axis**: Model size (number of parameters).
  + **Y-axis**: Intrinsic dimensionality required to maintain 90% accuracy.
* Key takeaway:
  + **As model size increases, intrinsic dimensionality decreases**, implying that **larger models generalize better with fewer modifications**.

**8. Applications and Implications**

* **Memory-Efficient Fine-Tuning:**
  + LoRA enables fine-tuning without modifying the entire model, reducing storage and computational needs.
* **Task-Specific Adaptation:**
  + Tasks with **lower intrinsic dimensionality** can be fine-tuned more efficiently.
* **Improved Expressivity:**
  + Layer-wise tuning enhances model performance by leveraging structural knowledge.

**9. Comparison with Other Fine-Tuning Methods**

* **Full Fine-Tuning**: Requires training all parameters, consuming extensive memory.
* **Prompt Tuning & Prefix Tuning**: Modify only the input embeddings but lack efficiency in deeper model changes.
* **Adapters**: Introduce additional layers but require more memory than LoRA.
* **LoRA**: Reduces fine-tuning overhead by selectively modifying rank-decomposed matrices.

**10. Conclusion**

* LoRA leverages the concept of **intrinsic dimensionality** to efficiently fine-tune LLMs.
* Layer-wise structuring **optimizes parameter usage** and **reduces computational cost**.
* Larger models can **generalize better** with fewer trainable parameters.
* LoRA is a **practical, memory-efficient, and widely adopted** fine-tuning method in NLP and LLM applications.

These notes provide a **comprehensive summary** of the lecture on LoRA and intrinsic dimensionality, covering theoretical foundations, experimental findings, and practical applications.

**Introduction to Parameter Efficient Fine-Tuning (PEFT)**

* The lecture discusses how to fine-tune models efficiently by modifying only specific parts of the model rather than updating all weights.
* The challenge in standard fine-tuning is the large parameter space, which can be computationally expensive and memory-intensive.
* The LoRA (Low-Rank Adaptation) technique is introduced as a way to reduce this parameter space while maintaining efficiency.

**Regular Fine-Tuning and Weight Updates**

* In traditional fine-tuning:
  + Start with pre-trained model weights (W).
  + Compute the weight update (ΔW) using backpropagation.
  + The new weight is given by: W' = W + ΔW.
* This can be seen as two parallel networks:
  + One with the original pre-trained weights.
  + One with the updated fine-tuned weights.
  + Adding them results in the final fine-tuned model.

**Contributions of LoRA**

LoRA introduced two main contributions:

1. **Identification of Essential Weights to Modify**
   * LoRA focuses only on modifying a subset of weights within the Transformer architecture.
   * Reduces the **intrinsic dimensionality** of the model significantly.
   * Updates only the **query (Q), key (K), value (V), and output projection (O) matrices** in Transformer blocks.
2. **Factorized Representation of Weight Updates**
   * Instead of updating full weight matrices, LoRA factorizes weight updates into two smaller matrices:
     + **Down Projection Matrix (A)**: Reduces the dimensionality.
     + **Up Projection Matrix (B)**: Expands back to original dimensions.
   * The final weight update is represented as **ΔW = A \* B**.
   * No non-linearity is applied, keeping the update process efficient.

**Comparison of Fine-Tuning Techniques**

Different fine-tuning methods were compared based on the fraction of trainable parameters and accuracy:

* **Full Fine-Tuning (FT)**: Updates all weights, requiring large memory and compute.
* **BitFit**: Updates only the biases in the model.
* **Soft Prompting (PRM)**: Learns continuous input embeddings without modifying model weights.
* **Prefix Tuning (Pre-Layer)**: Introduces additional tunable layers in early stages.
* **Adapters**: Introduce bottleneck layers into Transformer blocks.
* **LoRA**: Provides competitive performance with minimal trainable parameters.

For complex tasks like **WikiSQL (NL-to-SQL), NLI (Natural Language Inference), and Summarization**, LoRA achieved **better accuracy than full fine-tuning while using fewer parameters**.

**Ablation Studies on LoRA**

To validate LoRA’s efficiency, ablation studies were conducted:

* By fixing the total number of trainable parameters and training different subsets of weight matrices.
* Findings:
  + Training **only the query (Q) matrix** resulted in suboptimal performance.
  + Training **only the key (K) matrix** also led to lower accuracy.
  + The best performance was achieved by training a combination of **query, key, value, and output projection matrices**.

**LoRA Configuration for Fine-Tuning**

* LoRA allows flexibility in choosing:
  + Which matrices to fine-tune.
  + The **rank (R)** of the factorized matrices.
  + The **learning rate** for A and B matrices.
* The choice of rank **determines the trade-off between memory efficiency and performance**.
  + Increasing rank improves accuracy up to a point, after which it saturates or slightly declines.

**Intrinsic Dimensionality and Rank in LoRA**

* The rank (R) serves as an indicator of **intrinsic dimensionality** of the task.
* If the task is **complex** and the base model is **not very strong**, a **higher R** is needed.
* If the task is **simple** and the base model is **powerful**, a **lower R** can suffice.
* Experiments show that performance peaks at a certain rank and then slightly declines.

**Variations and Extensions of LoRA**

Due to LoRA’s success, several variants have been introduced:

1. **QLoRA**
   * Quantized version of LoRA to **reduce memory footprint**.
2. **LongLoRA**
   * Optimized for **long-context processing**.
3. **LoRA+**
   * Uses **different learning rates** for A and B matrices, improving convergence speed.
4. **DORA (Dynamic LoRA)**
   * Automatically finds the **optimal rank (R)** without needing multiple experiments.

**Training Stability in LoRA**

To ensure stable training, specific initialization techniques are used:

1. **B Matrix Initialization**
   * Initialized as **zero** to maintain the original model behavior in early training steps.
   * Ensures that **AB starts as zero**, meaning the fine-tuned model initially behaves like the base model.
2. **A Matrix Initialization**
   * Initialized using **Gaussian distribution (zero mean, small variance)**.
   * Prevents drastic changes when B updates, ensuring smooth training.

**Non-Linearity in LoRA**

* LoRA does not apply non-linearity to weight updates because:
  + The goal is to **add ΔW to W**, mimicking a direct weight update.
  + Non-linearity is unnecessary since we are **not applying additional transformations**.
  + This allows for easy fusion of LoRA weights with the base model after fine-tuning.

**Hot Swapping and Model Flexibility**

* Unlike prompt tuning, LoRA updates are not easily swappable because they modify internal weights.
* However, research is ongoing to enable **hot swapping** for LoRA:
  + This would allow switching between different LoRA-trained models **without downtime**.

**Summary of the Lecture**

* Discussed **Parameter Efficient Fine-Tuning (PEFT)** and why it is necessary.
* Introduced **LoRA** and its contributions:
  + Selective fine-tuning of Q, K, V, and O matrices.
  + Factorized weight updates with low-rank adaptation.
* Compared LoRA with other fine-tuning techniques.
* Explained ablation studies showing optimal weight selection and rank choices.
* Highlighted LoRA variants like **QLoRA, LongLoRA, LoRA+, and DORA**.
* Covered training stability measures and non-linearity considerations.
* Discussed emerging techniques to **enable hot swapping for LoRA**.

**Key Takeaways**

1. **LoRA achieves near full fine-tuning accuracy with only a small fraction of trainable parameters**.
2. **It modifies only essential weight matrices (Q, K, V, O) to reduce memory and computation requirements**.
3. **The rank (R) plays a crucial role in balancing model efficiency and accuracy**.
4. **Variants like QLoRA, LongLoRA, and DORA further optimize LoRA for different use cases**.
5. **LoRA’s initialization strategy ensures stable and efficient fine-tuning**.

This lecture provided an in-depth understanding of LoRA and its impact on fine-tuning large language models efficiently.

**KeyWords:**

**1. General Concepts**

* **Transfer Learning** – Using knowledge from one task (pre-training) and applying it to another (fine-tuning).
* **Pre-Training** – Training a model on large-scale, unlabelled data to acquire general world knowledge.
* **Fine-Tuning** – Adjusting the pre-trained model using labeled, task-specific data for better performance.
* **World Knowledge** – Information that the model has learned during pre-training from large corpora.

**2. Large Language Models (LLMs)**

* **LLM Era** – The period when AI models became much larger and more capable of reasoning and knowledge retention.
* **Instruction Alignment** – Adjusting LLMs with labeled examples to improve their ability to follow instructions.
* **Instruction Tuning** – Training models to follow human-like task instructions without requiring examples.
* **Alignment Phase** – The step where an LLM is refined using human feedback and labeled instructions.

**3. In-Context Learning (ICL)**

* **In-Context Learning (ICL)** – Providing instructions and examples in the input prompt to guide LLM behavior without updating its parameters.
* **Prompting** – Giving structured input (task descriptions, examples) to make LLMs generate desired outputs.
* **Few-Shot Learning** – Providing a few example input-output pairs in the prompt to guide LLM behavior.
* **Zero-Shot Learning** – Giving only an instruction without examples and relying on the model's pre-existing knowledge.
* **Prompt Sensitivity** – The tendency of LLMs to give different results based on small prompt changes.

**4. Fine-Tuning Techniques**

* **Full Fine-Tuning** – Training all model parameters on task-specific data to optimize performance.
* **Parameter-Efficient Fine-Tuning (PEFT)** – Adjusting only a small subset of parameters to save computational resources.
* **Adapter Layers** – Additional neural layers added to a pre-trained model to specialize it for specific tasks.

**5. Challenges in LLM Adaptation**

* **Cost of Hosting LLMs** – High resource demands for running and fine-tuning large models.
* **Knowledge Assumptions** – LLMs make implicit assumptions based on training data, sometimes leading to unexpected behaviors.
* **Prompt Instability** – Minor changes in prompt wording can drastically alter model outputs.
* **Model Transparency Issue** – Difficulty in understanding what the model has actually learned.
* **Inference Latency** – Increased response times due to complex input prompts and long sequences.

**1. Fine-Tuning**

Fine-tuning refers to adapting a pre-trained model to a specific task by further training it on task-specific data.

* **Full Fine-Tuning** – Modifying all model parameters by training on new data.
* **Parameter-Efficient Fine-Tuning (PEFT)** – Updating only a small subset of parameters to reduce computational cost.

**2. Checkpoints**

Snapshots of a model’s weights during training, allowing resumption from a specific stage instead of starting over. Essential for:

* Preventing progress loss due to system failures.
* Enabling model selection and hyperparameter tuning.

**3. Optimizer State**

Refers to additional data (like momentum and learning rate adjustments) stored by optimizers such as Adam or SGD, required for efficient training.

**4. Gradient Storage**

Gradients represent the direction and magnitude of model updates during training. They must be stored in memory to adjust weights but increase memory overhead.

**5. Forward Activations**

Intermediate values produced in the forward pass of neural network computation. Required for backpropagation but also consume memory.

**6. Mixed Precision Training**

A training technique that reduces memory usage and increases speed by using both **full precision (FP32)** and **half precision (FP16/BF16)** floating-point formats.

**7. Storage Overhead**

The additional disk space required to store multiple versions of fine-tuned models. Large models (e.g., GPT-3) require **hundreds of gigabytes** per checkpoint.

**8. Transfer Learning**

The concept of using a pre-trained model as a starting point for a new task. Fully fine-tuned models risk losing this generalization ability if overfitted to small datasets.

**9. Overfitting**

A model memorizing training data instead of learning generalizable patterns. Large models with **few training examples** overfit easily, reducing real-world accuracy.

**10. In-Context Learning (ICL)**

A method where a model **learns from examples given in the input prompt** rather than through parameter updates. It enables:

* Adaptation to multiple tasks without fine-tuning.
* Avoiding storage overhead from multiple fine-tuned models.

**11. Hosting Multiple LLMs**

Refers to running multiple separately fine-tuned models on different servers. This approach has **high infrastructure costs** compared to in-context learning.

**12. Throughput Utilization**

A measure of how efficiently computational resources (GPUs, TPUs) are used. Running multiple LLMs separately can lead to inefficiency, whereas **a single model with ICL** optimizes GPU usage.

**13. Compute Cost**

The amount of **GPU/TPU power and memory** required for training or inference. Full fine-tuning is costly, whereas ICL allows **task flexibility without retraining**.

**14. Dynamic Task Adaptation**

A feature of in-context learning where a model **switches between different tasks instantly** by altering the prompt instead of re-training.

**1. Fine-Tuning Related Terms**

* **Fine-Tuning:** Adapting a pre-trained language model to a specific task by training on task-specific data.
* **Fully Fine-Tuning:** Updating all model parameters during training, requiring massive computational resources.
* **Parameter-Efficient Fine-Tuning (PEFT):** A method that fine-tunes only a small subset of parameters while freezing the rest to save memory and computation.
* **Catastrophic Forgetting:** When a model forgets previously learned tasks after learning a new one, common in fine-tuning approaches.
* **Transfer Learning:** A technique where a model trained on a large dataset is adapted for another related task with minimal training data.

**2. In-Context Learning (ICL) Related Terms**

* **In-Context Learning:** The model learns from examples provided in the prompt without updating any parameters.
* **Few-Shot Learning:** Providing a few labeled examples in the prompt to guide model behavior.
* **Zero-Shot Learning:** The model performs a task without any examples, relying only on its pre-trained knowledge.
* **Prompt Engineering:** Crafting effective prompts to get desired responses from an LLM.

**3. Hardware & Optimization Related Terms**

* **Optimizer States:** Additional memory required for storing gradient updates during training.
* **Gradient Storage:** Memory needed to keep track of changes during backpropagation.
* **Activation Storage:** Memory required to store intermediate outputs (activations) during forward propagation.
* **Mixed-Precision Training:** Using lower precision (e.g., FP16 instead of FP32) to reduce memory consumption and speed up training.
* **Low-Rank Adaptation (LoRA):** A PEFT technique where only a small low-rank matrix is fine-tuned, significantly reducing memory usage.

**4. Model Deployment & Hosting**

* **Checkpointing:** Saving intermediate model states during training to resume from a particular step in case of failure.
* **Serving LLMs:** Deploying models for inference, ensuring efficient resource utilization.
* **Throughput:** The number of inferences a model can handle per second, crucial for real-time applications.
* **Latency:** The time taken by the model to respond to an input.