Supervised fine-tuning content

Supervised Fine-Tuning (SFT)
The purpose of pretraining a model on large datasets is that it is able to
reproduce language and its meaning. During this process, the model learns
to complete input phrases as shown in Figure 12-4.
Figure 12-4. A base or pretrained LLM was trained to predict the next word(s).
This example also illustrates that the model was not trained to follow
instructions and instead will attempt to complete a question rather than
answer it (Figure 12-5).
Figure 12-5. A base LLM will not follow instructions but instead attempts to predict each next word.
It may even create new questions.
We can use this base model and adapt it to certain use cases, such as
following instructions, by fine-tuning it.
i want to explain this in seminar give content

Here's a clear and seminar-friendly explanation you can use for your **Supervised Fine-Tuning (SFT)** slide:

Supervised Fine-Tuning (SFT)

- **Pretraining phase**:
  - Large Language Models (LLMs) are first trained on massive datasets.
  - Goal: learn general language patterns, grammar, and meaning.
  - Example: Predicting the *next word* in a sentence (Figure 12-4).
- **Limitation of pretrained models**:
  - A base LLM is good at continuing text but **not at following instructions**.
  - Instead of directly answering a question, it tries to *complete the text* (Figure 12-5).
  - Example: If you ask, *"What is the capital of France?"* the model may continue with *"…and Germany?"* instead of simply saying *"Paris"*.
- **Why fine-tuning?**
  - We want models to **do more than just predict text**.
  - In real-world applications (like chatbots, summarizers, or assistants), the model must **follow instructions and provide useful responses**.
- **Supervised Fine-Tuning (SFT)**:
  - Takes the pretrained base model and trains it further using **instruction–response pairs** (human-labeled data).

- Example:
  - Input: *"What is the capital of France?"*
  - Desired Output: *"Paris"*
- Over time, the model learns to **respond appropriately instead of just predicting the next word**.

✅ **Takeaway for seminar**:
Pretraining gives the model its *language ability*.
Supervised Fine-Tuning gives the model its *instruction-following ability*, making it useful for practical tasks.

Do you want me to also **design this into a PPT slide format with speaker notes** so it's ready for your seminar?

Full Fine-Tuning
The most common fine-tuning process is full fine-tuning. Like pretraining
an LLM, this process involves updating all parameters of a model to be in
line with your target task. The main difference is that we now use a smaller
but labeled dataset whereas the pretraining process was done on a large
dataset without any labels (Figure 12-6).
Figure 12-6. Compared to language modeling (pretraining), full fine-tuning uses a smaller but
labeled dataset.
You can use any labeled data for full fine-tuning, making it also a great
technique for learning domain-specific representations. To make our LLM
follow instructions, we will need question-response data. This data, as
shown in Figure 12-7, is queries by the user with corresponding answers.
Figure 12-7. Instruction data with instructions by a user and corresponding answers. The
instructions can contain many different tasks.
During full fine-tuning, the model takes the input (instructions) and applies
next-token prediction on the output (response). In turn, instead of
generating new questions, it will follow instructions.

Here's a structured, **seminar-ready explanation** for your **Full Fine-Tuning** slide:

Full Fine-Tuning

- **What it is**
  - Full fine-tuning is the most common fine-tuning method.
  - Similar to pretraining, but instead of using massive unlabeled data, we use a **smaller, labeled dataset** (Figure 12-6).
- **Key difference from pretraining**

- **Pretraining** → model learns general language by predicting next words from huge, unlabeled text.
- **Full fine-tuning** → updates **all model parameters** using labeled data designed for a specific task.

- **Why it's useful**
  - Allows the model to learn **domain-specific representations** (e.g., medical, legal, financial language).
  - Example: If we want the model to act as a medical assistant, we fine-tune it with medical Q&A data.
- **Instruction-following with full fine-tuning**
  - To make an LLM **follow instructions**, we provide **instruction–response pairs** (Figure 12-7).
    - Input: *User instruction or query*
    - Output: *Correct response*
  - The model still uses **next-token prediction**, but now it predicts the tokens of the *desired answer*, not random completions.
- **Outcome**
  - Instead of generating unrelated questions or continuing text, the model learns to **directly follow instructions and provide correct answers**.

✅ **Takeaway for seminar**:
Full fine-tuning adapts a pretrained LLM to **specific domains and tasks** by updating *all parameters* with **labeled instruction–response data**, turning it into an **instruction-following assistant**.

Do you also want me to **make a comparison slide** (Pretraining vs. Full Fine-Tuning) so your audience sees the distinction clearly?

> Parameter-Efficient Fine-Tuning (PEFT)
> Updating all parameters of a model has a large potential of increasing its performance but comes with several disadvantages. It is costly to train, has slow training times, and requires significant storage. To resolve these issues, attention has been given to parameter-efficient fine-tuning (PEFT) alternatives that focus on fine-tuning pretrained models at higher computational efficiency

Here's a **seminar-ready explanation** you can use for **Parameter-Efficient Fine-Tuning (PEFT):**

Parameter-Efficient Fine-Tuning (PEFT)

- **Problem with Full Fine-Tuning**
  - Updates **all parameters** of a large model.

- While effective, it has **major drawbacks**:
  - High training cost
  - Slow training time
  - Large storage requirements
- **What PEFT does**
  - Instead of updating the **entire model**, PEFT updates only a **small subset of parameters** while keeping most of the pretrained model **frozen**.
  - This makes the process **faster, cheaper, and more memory-efficient**.
- **Key benefits**
  - Much lower computational cost
  - Requires less storage space for trained models
  - Allows fine-tuning for multiple tasks **without retraining the entire model**
- **Use case**
  - Adapting large language models to new tasks/domains **efficiently**.
  - Example: Fine-tuning a model for medical Q&A using PEFT methods instead of retraining all parameters.

✅ **Takeaway for seminar**:
PEFT provides a **smarter, resource-efficient alternative** to full fine-tuning, enabling organizations to adapt large models quickly and affordably without sacrificing too much performance.

Would you like me to also include a **visual comparison slide** (Full Fine-Tuning vs. PEFT) so your audience can easily grasp the difference?

what are the components of peft

Great question 👍 — in seminars, it's really powerful to show that **PEFT (Parameter-Efficient Fine-Tuning)** isn't just one method but a **family of techniques**.

Here are the **main components / approaches of PEFT** you can present:

Components of PEFT

1. **Adapters**
   - Small neural modules inserted between layers of the pretrained model.
   - Only the adapter parameters are trained, while the main model remains frozen.
   - Example: **AdapterFusion** (can combine multiple adapters for multi-task learning).
2. **Prefix-Tuning (Prompt-Tuning)**

- A set of trainable **prefix vectors** (like virtual tokens) are prepended to the input.
- The model learns how to interpret instructions without changing its main parameters.
- Example: *T5 Prompt-Tuning*.

3. **LoRA (Low-Rank Adaptation)**
   - Instead of updating full weight matrices, LoRA adds **small low-rank matrices**.
   - Much fewer trainable parameters but still very effective.
   - One of the most popular PEFT methods in practice today.
4. **BitFit (Bias-Only Fine-Tuning)**
   - Trains **only the bias terms** of the model while keeping all other weights frozen.
   - Extremely parameter-efficient (tiny number of trainable parameters).
5. **IA³ (Infused Adapter by Inhibiting and Amplifying Inner Activations)**
   - Uses trainable vectors to scale (amplify/inhibit) activations inside transformer layers.
   - Even fewer parameters than adapters/LoRA.

✅ Takeaway for Seminar

PEFT is not one technique—it's a **toolbox of strategies** (Adapters, Prefix-Tuning, LoRA, BitFit, IA³, etc.) that let us fine-tune large models **efficiently** while freezing most of their original parameters.

Do you want me to create a **table slide (Full Fine-Tuning vs. PEFT Components)** so you can show methods, trainable parameters, and efficiency side-by-side?

Adapters
Adapters are a core component of many PEFT-based techniques. The
method proposes a set of additional modular components inside the
Transformer that can be fine-tuned to improve the model's performance on
a specific task without having to fine-tune all the model weights. This saves
a lot of time and compute.
Adapters are described in the paper "Parameter-efficient transfer learning
for NLP", which showed that fine-tuning 3.6% of the parameters of BERT
for a task can yield comparable performance to fine-tuning all the model's
weights.1 On the GLUE benchmark, the authors show they reach within
0.4% of the performance of full fine-tuning. In a single Transformer block,
the paper's proposed architecture places adapters after the attention layer
and the feedforward neural network as illustrated in Figure 12-8.
Figure 12-8. Adapters add a small number of weights in certain places in the network that can
be
fine-tuned efficiently while leaving the majority of model weights frozen.
It's not enough to only alter one Transformer block, however, so these
components are part of every block in the model, as Figure 12-9 shows.
Figure 12-9. Adapter components span the various Transformer blocks in the model.
Seeing all the adapter's components across the model like this enables us to

see individual adapters as shown in Figure 12-10, which is a collection of these components spanning all the blocks of the model. Adapter 1 can be a specialist in, say, medical text classification, while Adapter 2 can specialize in named-entity recognition (NER). You can download specialized adapters from https://oreil.ly/XraXg.

Figure 12-10. Adapters that specialize in specific tasks can be swapped into the same architecture (if they share the same original model architecture and weights).

The paper "AdapterHub: A framework for adapting transformers" introduced the Adapter Hub as a central repository for sharing adapters.2 A lot of these earlier adapters were more focused on BERT architectures. More recently, the concept has been applied to text generation Transformers in papers like "LLaMA-Adapter: Efficient fine-tuning of language models with zero-init attention".3

Low-Rank Adaptation (LoRA)

As an alternative to adapters, low-rank adaptation (LoRA) was introduced and is at the time of writing is a widely used and effective technique for PEFT. LoRA is a technique that (like adapters) only requires updating a small set of parameters. As illustrated in Figure 12-11, it creates a small subset of the base model to fine-tune instead of adding layers to the model.4

Figure 12-11. LoRA requires only fine-tuning a small set of parameters that can be kept separately from the base LLM.

Like adapters, this subset allows for much quicker fine-tuning since we only need to update a small part of the base model. We create this subset of parameters by approximating large matrices that accompany the original LLM with smaller matrices. We can then use those smaller matrices as a replacement and fine-tune them instead of the original large matrices. Take for example the 10 × 10 matrix we see in Figure 12-12.

Figure 12-12. A major bottleneck of LLMs is their massive weight matrices. Only one of these may have 150 million parameters and each Transformer block would have its version of these.

We can come up with two smaller matrices, which when multiplied, reconstruct a 10 × 10 matrix. This is a major efficiency win because instead of using 100 weights (10 times 10) we now only have 20 weights (10 plus 10), as we can see in Figure 12-13.

Figure 12-13. Decomposing a large weight matrix into two smaller matrices leads to a compressed, low-rank version of the matrix that can be fine-tuned more efficiently.

During training, we only need to update these smaller matrices instead of the full weight changes. The updated change matrices (smaller matrices) are then combined with the full (frozen) weights as illustrated in Figure 12-14.

Figure 12-14. Compared to full fine-tuning, LoRA aims to update a small representation of the original weights during training.

But you might suspect that performance would drop. And you would be right. But where does this trade-off make sense?

Papers like "Intrinsic dimensionality explains the effectiveness of language model fine-tuning" demonstrate that language models "have a very low intrinsic dimension."5 This means that we can find small ranks that approximate even the massive matrices of an LLM. A 175B model like GPT-3, for example, would have a weight matrix of 12,288 × 12,288 inside each of its 96 Transformer blocks. That's 150 million parameters. If we can successfully adapt that matrix into rank 8, that would only require two 12,288 × 2 matrices resulting in 197K parameters per block. These are major savings in speed, storage, and compute as explained further in the previously referenced LoRA paper.

This smaller representation is quite flexible in that you can select which parts of the base model to fine-tune. For instance, we can only fine-tune the Query and Value weight matrices in each Transformer layer.

Compressing the model for (more) efficient training

We can make LoRA even more efficient by reducing the memory requirements of the model's original weights before projecting them into smaller matrices. The weights of an LLM are numeric values with a given precision, which can be expressed by the number of bits like float64 or float32. As illustrated in Figure 12-15, if we lower the amount of bits to represent a value, we get a less accurate result. However, if we lower the number of bits we also lower the memory requirements of that model.

Figure 12-15. Attempting to represent pi with float 32-bit and float 16-bit representations. Notice the lowered accuracy when we halve the number of bits.

With quantization, we aim to lower the number of bits while still accurately representing the original weight values. However, as shown in Figure 12 16, when directly mapping higher precision values to lower precision values, multiple higher precision values might end up being represented by the same lower precision values.

Figure 12-16. Quantizing weights that are close to one another results in the same reconstructed weights thereby removing any differentiating factor.

Instead, the authors of QLoRA, a quantized version of LoRA, found a way to go from a higher number of bits to a lower value and vice versa without differentiating too much from the original weights.6

They used blockwise quantization to map certain blocks of higher precision values to lower precision values. Instead of directly mapping higher precision to lower precision values, additional blocks are created that allow for quantizing similar weights. As shown in Figure 12-17, this results in values that can be accurately represented with lower precision.

Figure 12-17. Blockwise quantization can accurately represent weights in lower precision through quantization blocks.

A nice property of neural networks is that their values are generally normally distributed between –1 and 1. This property allows us to bin the original weights to lower bits based on their relative density, as illustrated in Figure 12-18. The mapping between weights is more efficient as it takes into account the relative frequency of weights. This also reduces issues with outliers.

Figure 12-18. Using distribution-aware blocks we can prevent values close to one another from being represented with the same quantized value.

Combined with the blockwise quantization, this normalization procedure allows for accurate representation of high precision values by low precision values with only a small decrease in the performance of the LLM. As a result, we can go from a 16-bit float representation to a measly 4-bit

normalized float representation. A 4-bit representation significantly reduces the memory requirements of the LLM during training. Note that the quantization of LLMs in general is also helpful for inference as quantized LLMs are smaller in size and therefore require less VRAM.
There are more elegant methods to further optimize this like double quantization and paged optimizers, which you can read about more in th

Perfect — you've pasted the detailed text covering **Adapters, LoRA, and QLoRA**. I'll reshape it into **seminar-friendly content**, broken into clean sections you can use directly in your slides with key points and speaker notes.

Parameter-Efficient Fine-Tuning (PEFT) Techniques

---

1. Adapters

- **What they are**
  - Small, trainable modules inserted into each Transformer block.
  - Only adapter parameters are fine-tuned; the base model remains frozen.
- **Key insight**
  - Introduced in *"Parameter-efficient transfer learning for NLP"*.
  - Fine-tuning only **3.6% of BERT's parameters** gave performance **within 0.4% of full fine-tuning** (on GLUE benchmark).
- **Architecture**
  - Placed after **attention** and **feedforward** layers in every Transformer block.
  - Multiple adapters can be used: e.g.,
    - Adapter 1 → Medical text classification
    - Adapter 2 → Named Entity Recognition
- **Ecosystem**
  - **AdapterHub** repository → share/download ready-made adapters.
  - Extended to generative models like **LLaMA-Adapter**.

✅ **Takeaway**: Adapters make it possible to specialize one big model into multiple tasks **without retraining everything**.

---

2. LoRA (Low-Rank Adaptation)

- **What it does**
  - Instead of adding modules (like adapters), LoRA **decomposes weight matrices** into smaller low-rank matrices.
  - Train only these smaller matrices → huge parameter savings.

- **Example**
  - Original weight matrix: **10×10 = 100 parameters**.
  - Low-rank decomposition: **(10×2) + (2×10) = 20 parameters**.
  - Updates are merged back with frozen weights at inference.
- **Impact**
  - Reduces millions of parameters into a fraction (e.g., GPT-3's 150M params per block → rank-8 LoRA = only ~197K params).
  - Flexible: can target only specific parts like **Query & Value matrices**.

✅ **Takeaway**: LoRA enables **efficient fine-tuning** with minimal compute while maintaining strong performance.

---

3. QLoRA (Quantized LoRA)

- **Motivation**
  - Even with LoRA, storing huge weight matrices is costly.
  - Solution: **quantize** (reduce bit precision of weights).
- **How it works**
  - Standard float precision: 16-bit → consumes high memory.
  - QLoRA compresses to **4-bit normalized floats** with minimal accuracy loss.
  - Uses **blockwise quantization + normalization** to handle weight distribution and avoid losing important variations.
- **Benefits**
  - Much lower memory requirements.
  - Works well even with massive LLMs.
  - Useful for **both training and inference** (smaller VRAM usage).

✅ **Takeaway**: QLoRA combines **LoRA + quantization**, making fine-tuning large LLMs feasible on smaller GPUs.

Overall Summary

- **Adapters** → Add small modules into each layer.
- **LoRA** → Replace big matrices with low-rank approximations.
- **QLoRA** → Further compress with quantization for memory efficiency.

👉 For your **seminar slides**, you could have:

- **One overview slide** ("PEFT = Adapters, LoRA, QLoRA")
- **One slide each** explaining them with diagrams/figures.

- **One final comparison table** showing **trainable parameters, efficiency, performance trade-off**.

Do you want me to **make that final comparison table (Adapters vs LoRA vs QLoRA)** for your slide deck?

Supervised Fine-Tuning (SFT)

Pretraining phase:

Large Language Models (LLMs) are first trained on massive datasets.

Goal: learn general language patterns, grammar, and meaning.

Example: Predicting the next word in a sentence (Figure 12-4).

Limitation of pretrained models:

A base LLM is good at continuing text but not at following instructions.

Instead of directly answering a question, it tries to complete the text (Figure 12-5).

Example: If you ask, "What is the capital of France?" the model may continue with "...and Germany?" instead of simply saying "Paris".

Why fine-tuning?

We want models to do more than just predict text.

In real-world applications (like chatbots, summarizers, or assistants), the model must follow instructions and provide useful responses.

Supervised Fine-Tuning (SFT):

Takes the pretrained base model and trains it further using instruction–response pairs (human-labeled data).

Example:

Input: "What is the capital of France?"

Desired Output: "Paris"

Over time, the model learns to respond appropriately instead of just predicting the next word.
give as explanation sentences

Got it ✅ — here's your **Supervised Fine-Tuning (SFT)** section rewritten as smooth **explanation sentences** instead of bullet points, so you can directly use it in your seminar talk or report:

Supervised Fine-Tuning (SFT)

Large Language Models (LLMs) are first pretrained on massive datasets with the primary goal of learning general language patterns, grammar, and meaning. In this pretraining phase, the model is trained to predict the next word in a sentence, which helps it develop a broad understanding of language.

However, pretrained models come with limitations. While they are very good at continuing text, they are not designed to follow instructions. For example, if you ask a base LLM, *"What is the capital of France?"*, instead of giving the correct answer *"Paris"*, it might continue the text with something like *"...and Germany?"*, because it is simply trying to complete the sequence rather than respond to the question.

This is why fine-tuning is necessary. In real-world applications such as chatbots, summarization systems, or virtual assistants, we expect the model to follow instructions and provide useful, task-specific responses, not just generate more text.

Supervised Fine-Tuning (SFT) addresses this by taking a pretrained base model and training it further using instruction–response pairs created by humans. For instance, the instruction could be *"What is the capital of France?"* and the desired response would be *"Paris"*. By repeatedly training on such examples, the model gradually learns to respond appropriately to instructions instead of merely predicting the next word.

Do you want me to do the same **sentence-style explanation** for **Full Fine-Tuning and PEFT** as well, so your whole seminar content stays consistent?

yes

Perfect ✅ — here's the **sentence-style explanation** for **Full Fine-Tuning** and **PEFT**, written in the same style as your SFT explanation so your seminar content flows consistently.

Full Fine-Tuning

Full fine-tuning is the most common fine-tuning approach. It is similar to pretraining in the sense that the model's parameters are updated, but instead of using massive amounts of unlabeled data, a smaller, labeled dataset is used. This allows the model to be adapted to a specific task or domain.

The key difference is that, in full fine-tuning, **all parameters of the model are updated**. For example, to make a model follow instructions, we provide it with question–response pairs where the input is the user's instruction and the output is the correct answer. The model still relies on next-token prediction, but now it predicts the tokens of the desired response instead of generating random text continuations.

Because the model's entire parameter set is updated, full fine-tuning can be very effective, but it also comes with high computational cost, longer training time, and large storage requirements. Nonetheless, it is a powerful method for teaching a model to follow instructions or for adapting it to domain-specific applications like medical or legal text.

Parameter-Efficient Fine-Tuning (PEFT)

Although full fine-tuning can yield strong results, it is often expensive and resource-intensive, especially with very large models. Updating all parameters requires massive compute power, slows down training, and demands significant storage.

To address this, researchers developed **Parameter-Efficient Fine-Tuning (PEFT)** techniques. Instead of updating the entire model, PEFT methods only fine-tune a **small subset of parameters**, while keeping most of the pretrained weights frozen. This makes training faster, cheaper, and more memory-efficient, while still allowing the model to adapt effectively to new tasks.

Some of the most widely used PEFT methods include **Adapters**, which insert small trainable modules inside Transformer blocks; **LoRA (Low-Rank Adaptation)**, which replaces large weight matrices with smaller decomposed versions; and **QLoRA**, which further compresses weights using quantization for even more efficiency. These methods allow large models to be fine-tuned for many tasks without the heavy costs of retraining all parameters.

✅ **Now you have all three (SFT, Full Fine-Tuning, and PEFT) in the same explanatory style.**
Would you like me to also create a **final comparison explanation** (like a story tying all three together: Pretraining → SFT → Full Fine-Tuning → PEFT) so your seminar feels like a smooth journey?

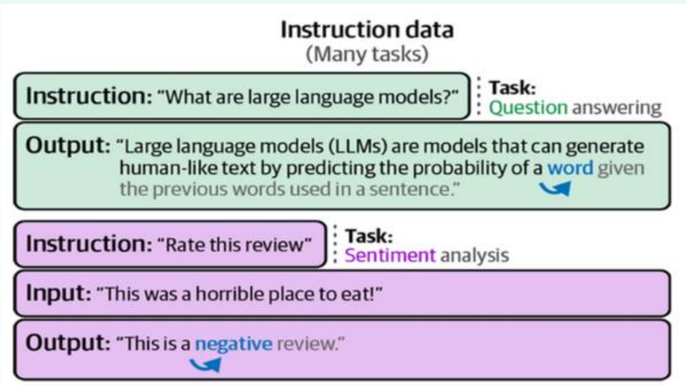

Figure 12-7. Instruction data with instructions by a user and corresponding answers. The instructions can contain many different tasks.

```
explain this imaghe
```

Great — let me explain this image clearly so you can use it in your seminar:

Explanation of the Image (Figure 12-7: Instruction Data)

This image shows how **instruction data** is used in **Supervised Fine-Tuning (SFT)** to train Large Language Models (LLMs).

1. **Instruction–Response Format**
   - The model is given an **instruction** (a task to perform) and the correct **output** (the response we expect).
   - These pairs are created by humans and cover many possible tasks.

2. **Example 1: Question Answering**
    - **Instruction:** "What are large language models?"
    - **Task:** Question answering
    - **Output:** "Large language models (LLMs) are models that can generate human-like text by predicting the probability of a word given the previous words used in a sentence."
    - Here, the model learns how to properly answer a factual question instead of just continuing text.
3. **Example 2: Sentiment Analysis**
    - **Instruction:** "Rate this review"
    - **Input:** "This was a horrible place to eat!"
    - **Task:** Sentiment analysis
    - **Output:** "This is a negative review."
    - Here, the model is trained to classify the sentiment of a user's text.
4. **Key Insight**
    - Instruction data allows one pretrained model to learn **many different tasks** (question answering, sentiment analysis, summarization, translation, etc.) depending on the instructions given.
    - This is what makes fine-tuned LLMs much more useful in real-world applications compared to base pretrained models.

✅ **In simple words for your seminar**:
This figure shows how instruction–response pairs teach a model to follow instructions for different tasks. Instead of just predicting the next word, the model learns to answer questions, classify reviews, and handle many other tasks by being trained with examples like these.

Do you want me to also **redraw this into a simpler diagram (Instruction → Model → Output)** for your presentation slides?

are these the dataset for training

Yes ✅

The **instruction–response pairs shown in that figure are examples of the dataset used for training** during **Supervised Fine-Tuning (SFT)**.

Here's how it works:

- **Pretraining phase:**
  The base LLM is trained on massive unlabeled text (e.g., books, Wikipedia, websites).

- **Fine-tuning phase (SFT):**
  The model is trained again, but this time on a **smaller labeled dataset** where each sample has:
  - An **instruction** (the user's query or task).
  - An **expected response** (the correct answer/output).

So yes — datasets like the ones in your figure (Question → Answer, Review → Sentiment, etc.) are exactly what's used to fine-tune the model to follow instructions.

👉 In practice, such datasets are usually called **Instruction-Tuning datasets** or **Human-Labeled QA pairs**.

Do you want me to also give you **real-world examples of commonly used instruction datasets** (like Dolly, Alpaca, or OASST) so you can mention them in your seminar?