

# GPT Fine-tuning

<https://www.datacamp.com/tutorial/fine-tuning-openais-gpt-4-step-by-step-guide>

## What is Fine-tuning?

Fine-tuning represents a sophisticated process that refines pre-trained models (like GPT-4) for specific tasks or domains, leveraging the model's extensive foundational knowledge acquired during initial training on diverse datasets. This involves adjusting the model's parameters based on task-specific data, enhancing its performance, and enabling it to handle particular applications with greater precision and efficiency.

One illustrative example of fine-tuning's impact is the enhancement of a model's responses to specialized queries. For instance, a pre-trained model might offer a basic explanation when asked why the sky is blue. Through fine-tuning, this response can be enriched to include detailed scientific context, making it more suitable for specialized applications such as educational platforms.

**Fine-tuning methods** range from instruction fine-tuning, where models are trained using specific examples that demonstrate the desired responses, to parameter-efficient fine-tuning (PEFT), which updates only a subset of the model's parameters to conserve computational resources and prevent catastrophic forgetting.

In contrast, **Retrieval Augmented Generation (RAG)** represents a different approach. RAG combines elements of retrieval-based and **generative models** to enhance the quality of generated content by incorporating information retrieved from external sources during the generation process.

While fine-tuning focuses on optimizing pre-existing models for specific tasks, RAG integrates external knowledge to enrich the content generation process.

The choice between fine-tuning and RAG depends on the specific requirements of the application, including the need for up-to-date information, computational resources available, and the desired level of task specialization. Fine-tuning offers a direct way to leverage pre-trained models' vast knowledge base for specific tasks, while RAG provides a dynamic approach to ensuring the relevance and accuracy of the information the model uses.

## When to Use Fine-tuning?

**Fine-tuning** OpenAI's text generation models is a powerful method to tailor them to specific needs, but it demands significant time and resources. Before jumping into fine-tuning, it's advisable to try maximizing the model's performance through prompt engineering, prompt chaining (dividing complex tasks into simpler, sequential prompts), and utilizing functions. This approach is recommended for several reasons:

- The models may initially struggle with certain tasks, but **crafting the right prompts** can significantly improve outcomes, making fine-tuning unnecessary in many cases.
- Adjusting prompts and employing strategies like **prompt chaining** or **function calling** offers immediate feedback, allowing for rapid iteration. In contrast, fine-tuning involves creating datasets and training models, which takes more time.

- Even if fine-tuning becomes essential, the preliminary work done with prompt engineering is not wasted. Incorporating [well-designed prompts](#) into the fine-tuning process or combining them with fine-tuning techniques often yields the best results.

## Common Use-Cases for Fine-Tuning

---

Fine-tuning can be particularly beneficial in the following scenarios:

- **Customizing output characteristics.** When you need the model to adhere to a specific style, tone, format, or other qualitative aspects, fine-tuning can help shape its responses accordingly.
- **Enhancing reliability.** For applications where it's crucial that the model consistently produces the desired type of output, fine-tuning can improve its dependability.
- **Addressing complex prompts.** If the model fails to follow complex instructions, fine-tuning can help it understand and execute such prompts more effectively.
- **Managing edge cases.** Fine-tuning can enable the model to handle numerous edge cases in specific, predetermined ways, enhancing its versatility.
- **Learning new skills or tasks.** When introducing the model to a new skill or task that's difficult to encapsulate within a prompt, fine-tuning can equip it with the necessary capabilities.

## A Step-by-Step Guide To Fine-tuning OpenAI's GPT Models Using Python

---

Due to the proprietary nature of OpenAI's models, you must utilize a fine-tuning API from OpenAI if you want to fine-tune any of the OpenAI models. This process involves uploading your data to OpenAI's servers to train the model.

Currently, only the following models are supported for fine-tuning via API.

- gpt-3.5-turbo-0125,
- gpt-3.5-turbo-1106,
- gpt-3.5-turbo-0613,
- babbage-002,
- davinci-002

*GPT-4 fine-tuning is in experimental access, and eligible developers can request access via the [fine-tuning UI](#). Regardless of the model used, the process of fine-tuning and the code in this tutorial does not change.*

## 6 Steps For Fine-Tuning OpenAI GPT Models

---

### 1. Prepare the training data

This is the most important step. The quality of a fine-tuned model is directly dependent on the data that you will create to fine-tune the model. For the OpenAI API, the data must be stored in jsonl format.

JSONL format is a text format where each line is a separate JSON object, making it suitable for processing large datasets line by line. This format is particularly useful for handling data that can be easily divided into individual records without losing structure.

**Example:**

```
1 | {"prompt": "<prompt text>", "completion": "<ideal generated text>"}
2 | {"prompt": "<prompt text>", "completion": "<ideal generated text>"}
```

```
`{"prompt": "", "completion": ""}```
```

For this tutorial, I have created sample data with only five examples. The data consists of questions on various topics in machine learning and answers in the style of Shakespeare.

Sample data I created for this tutorial:

```
1 | {"messages": [{"role": "system", "content": "You are teaching assistant for Machine Learning. You should help to user to answer on his question."}, {"role": "user", "content": "What is machine learning?"}, {"role": "assistant", "content": "'Tis but the art of teaching machines to think, to learn from data most fine, and decisions to link."}]}
3 |
4 | {"messages": [{"role": "system", "content": "You are teaching assistant for Machine Learning. You should help to user to answer on his question."}, {"role": "user", "content": "How doth neural networks work?"}, {"role": "assistant", "content": "They're like the brains of yon digital beast, with layers many, and nodes that cease."}]}Powered By
```

## *2. Installing the openai library and setting up an API token*

The first step of creating a fine-tuning job using OpenAI API is to upload the file. To use this API, you must have the `openai` library in python installed.

```
1 |
2 | !pip install openai
3 | Powered By
4 |
```

Now login into your platform.openai.com account and generate an API token:

## Create new secret key

Name Optional

My Test Key

Permissions

All

Restricted

Read Only

Cancel

Create secret key

You can then set the token as an environment variable using the os library.

```
1 import os
2
3
4
5 # Set the OPENAI_API_KEY environment variable
6 os.environ['OPENAI_API_KEY'] = '...'
7 Powered By
8
```

### 3. Upload the training file

Your training file must be in jsonl format. Once you've uploaded the file, processing might take a while. The maximum size for file uploads is 1 GB. To upload a file on the OpenAI server:

```
1
2 from openai import OpenAI
3 client = OpenAI()
4
5
6 client.files.create(
7     file=open("train.jsonl", "rb"),
8     purpose="fine-tune"
9 )
10 Powered By
11
```

Output:

```
1
2 FileObject(id='file-rIua39sJX1064gzxTYfpvJx7', bytes=11165, created_at=1709499930,
3 filename='train.jsonl', object='file', purpose='fine-tune', status='processed',
4 status_details=None)
5
6 Powered By
```

## 4. Create a fine-tuned model

After ensuring the file has been successfully uploaded, the next step is to create a fine-tuning job. `training_file` is the file ID that was returned when the training file was uploaded to the OpenAI API. To start a fine-tuning job:

```
1  from openai import OpenAI
2  client = OpenAI()
3
4
5
6  client.fine_tuning.jobs.create(
7      training_file="file-rIua39sJX1064gzxTYfpvJx7",
8      model="gpt-3.5-turbo" #change to gpt-4-0613 if you have access
9  )
10 Powered By
11
```

For configuring extra fine-tuning settings, such as the `validation_file` or hyperparameters, check out the [API documentation on fine-tuning](#).

Fine-tuning jobs vary in time to complete, ranging from minutes to hours, depending on the model and dataset size. For instance, with just 50 examples in our `train.jsonl` dataset and using the `gpt-3.5-turbo` model, the job finished in just 7 minutes.

Once the job finishes, an email confirmation will also be sent.

Besides setting up a fine-tuning job, you have the option to view a list of current jobs, check the status of a specific job, or cancel a job.

```
1
2  from openai import OpenAI
3  client = OpenAI()
4
5
6  # List 10 fine-tuning jobs
7  client.fine_tuning.jobs.list(limit=10)
8
9
10 # Retrieve the state of a fine-tune
11 client.fine_tuning.jobs.retrieve("...")
12
13
14 # Cancel a job
15 client.fine_tuning.jobs.cancel("...")
16
17
18 # List up to 10 events from a fine-tuning job
19 client.fine_tuning.jobs.list_events(fine_tuning_job_id="...", limit=10)
20
21
22 # Delete a fine-tuned model (must be an owner of the org the model was created in)
23 client.models.delete("ft:gpt-3.5-turbo:xxx:xxx")\
24 Powered By
```

## 5. Analyze fine-tuned model

OpenAI offers key training metrics such as training loss, token accuracy for training, test loss, and test token accuracy. These metrics help ensure training is progressing as expected, with loss decreasing and token accuracy improving. You can view these useful metrics in an event object during an active fine-tuning job.

```
1
2 {
3     "object": "fine_tuning.job.event",
4     "id": "ftjob-Na7BnF5y91wwGJ4EgxtzVyDD",
5     "created_at": 1693582679,
6     "level": "info",
7     "message": "Step 100/100: training loss=0.00",
8     "data": {
9         "step": 100,
10        "train_loss": 1.805623287509661e-5,
11        "train_mean_token_accuracy": 1.0
12    },
13    "type": "metrics"
14 }
15 Powered By
16
```

You can also see this information on the UI as well.



## 6. Using the fine-tuned model

Upon successful completion of a job, the job details will include the `fine_tuned_model` field, displaying the name of the model. You can make an API call to this model and get a response from the model that we just tuned.

```
1  from openai import OpenAI
2  client = OpenAI()
3
4
5
6  completion = client.chat.completions.create(
7      model="ft:gpt-3.5-turbo-0613:personal::8k01tfYd",
8      messages=[
9          {"role": "system", "content": "You are a teaching assistant for Machine Learning. You should help to user to answer on his question."},
10         {"role": "user", "content": "What is a loss function?"}
11     ]
12 )
13 print(completion.choices[0].message)
14 Powered By
15
```

### Output:

```
1  ChatCompletionMessage(content="To measure how far we falter, it guides our way, our hope's
2  defaulter.", role='assistant', function_call=None, tool_calls=None)
3
4  Powered By
```

## Conclusion

This tutorial has guided you through the process of fine-tuning OpenAI's GPT model, an advanced step in leveraging the LLM's vast capabilities for specialized domain applications. Fine-tuning allows us to refine the language model's responses, making them more adept at handling specific tasks, styles, or domains with greater precision and efficiency.

This tutorial uses OpenAI Python API for fine-tuning a model. If you prefer to learn how to do the same thing through a UI without writing a single line of code, you can check out [How to Fine Tune GPT 3.5](#) tutorial on Datacamp.

Since GPT is not an open-source model the process of fine-tuning is rather simple and just involves making an API call. This is very different when it comes to fine-tuning open-source models like llama-2, Mistral, Stable diffusion, etc. If you are interested in learning how to fine-tune various open-source models, you can check out these blogs on DataCamp:

- [Fine-Tuning LLaMA 2: A Step-by-Step Guide to Customizing the Large Language Model](#)
- [FLAN-T5 Tutorial: Guide and Fine-Tuning](#)
- [Fine-tuning Stable Diffusion XL with DreamBooth and LoRA](#)

For more beginner resources on fine-tuning LLM, check out [An Introductory Guide to Fine-Tuning LLMs](#) on Datacamp.

# ChatGPT fine-tuning techniques

Fine-tuning ChatGPT (or similar LLMs like GPT-3.5/4) involves multiple steps to adapt the pre-trained model for specific tasks or alignment with human preferences. Here's a detailed breakdown of the **key fine-tuning steps** in ChatGPT's architecture:

---

## **1. Supervised Fine-Tuning (SFT)**

**Purpose:** Adapt the base model to follow instructions or perform domain-specific tasks.

**Steps:**

### **1. Dataset Preparation:**

- Collect high-quality prompt-response pairs (e.g., OpenAI's human-written demonstrations).
- Example: (prompt: "Explain quantum computing", response: "Quantum computing uses qubits...").

### **2. Training:**

- Fine-tune the pre-trained GPT model on this dataset using **causal language modeling** (next-token prediction).
- Loss: Cross-entropy between model outputs and ground-truth responses.

### **3. Output:** A model better at task-specific generation (but not yet aligned with human preferences).

**Key Point:** This is often the **first step** before RLHF.

---

## **2. Reward Modeling (Training the Reward Model)**

**Purpose:** Learn human preferences to guide RLHF.

**Steps:**

### **1. Dataset Preparation:**

- Collect rankings of model responses (e.g., humans rank multiple answers to the same prompt).
- Example: (prompt, response A, response B, preferred\_response).

### **2. Train the Reward Model (RM):**

- Use a smaller model (e.g., a distilled version of GPT) to predict human preference scores.
- Loss: Pairwise ranking loss (e.g., Bradley-Terry model).

### **3. Output:** A scalar reward function $R(\text{prompt}, \text{response})$ that scores response quality.

---

### ***3. Reinforcement Learning from Human Feedback (RLHF)***

**Purpose:** Align the SFT model with human preferences using the reward model.

**Steps:**

#### **1. Proximal Policy Optimization (PPO):**

- **Rollout:** The SFT model generates responses to prompts.
- **Scoring:** The reward model assigns scores to these responses.
- **Optimization:** Update the SFT model's policy to maximize reward (with KL-divergence constraints to avoid drastic changes).

#### **2. Iterative Refinement:**

- Repeat PPO steps + periodically gather new human feedback to improve the reward model.

#### **3. Output:** The final "aligned" model (e.g., ChatGPT).

**Key Point:** RLHF is why ChatGPT refuses harmful requests and gives helpful answers.

---

### ***4. Optional: Parameter-Efficient Fine-Tuning (PEFT)***

**Purpose:** Adapt the model efficiently without full fine-tuning.

**Methods:**

- **LoRA:** Low-rank updates to attention weights (original weights frozen).
- **Adapter Layers:** Small trainable modules inserted between frozen layers.

**Use Case:** Customizing ChatGPT for niche domains (e.g., legal/medical QA) with limited data.

---

### ***5. Post-Tuning Evaluation & Deployment***

**Steps:**

#### **1. Human/Auto Evaluation:**

- Check for alignment, safety, and task performance.

#### **2. Safety Mitigations:**

- Add filters, moderation tools, or rule-based guards.

#### **3. Deployment:**

- Serve the fine-tuned model via an API (e.g., OpenAI's gpt-3.5-turbo-instruct).
-

## *Visual Pipeline (Simplified)*

```
1 | Base GPT Model
2 |
3 |   └─ Supervised Fine-Tuning (SFT) → SFT Model
4 |
5 |   └─ Reward Modeling (RM) → Reward Model
6 |
7 |   └─ RLHF (PPO + RM) → Final ChatGPT Model
```

### *Key Details*

- **Data Quality Matters:** Garbage in = garbage out (e.g., biased/hallucinated data harms tuning).
- **Compute Cost:** RLHF is expensive (requires GPU clusters).
- **Original Model:** The base GPT weights are *not* modified directly; fine-tuning creates a new variant.

### *Comparison with BERT Fine-Tuning*

Step	ChatGPT/GPT	BERT
Base Objective	Next-token prediction	Masked language modeling
Fine-Tuning	SFT + RLHF	Task-specific head tuning
Efficiency	LoRA/RLHF dominant	Full/PEFT fine-tuning

### *When to Use Which?*

- **SFT:** For task-specific performance (e.g., code generation).
- **RLHF:** For alignment/safety (e.g., chatbot behavior).
- **PEFT:** For low-resource customization (e.g., startup workflows).

## *Example: Supervised Fine-Tuning (SFT) for code generation*

Let's break down **Supervised Fine-Tuning (SFT)** for **code generation** using a pre-trained model (e.g., GPT-3.5 or StarCoder). I'll cover dataset preparation, model architecture, training loops, and code implementation.

---

### **1. Dataset Preparation for Code Generation**

#### **Dataset Structure**

For SFT, you need **prompt-completion pairs** where:

- **Prompt:** A natural language instruction or code context.
- **Completion:** The desired code output.

#### **Example Dataset (JSONL Format):**

```
1 {"prompt": "Write a Python function to reverse a string.", "completion": "def\nreverse_string(s):\n    return s[::-1]"\n2 {"prompt": "Implement quicksort in Java.", "completion": "public void quicksort(int[] arr, int\n    low, int high) {\n        if (low < high) {\n            int pivot = partition(arr, low, high);\n            quicksort(arr, low, pivot - 1);\n            quicksort(arr, pivot + 1, high);\n        }\n    }"}\n\n
```

#### **Key Requirements**

1. **High Quality:** Code should be correct, executable, and match the prompt.
2. **Diversity:** Cover multiple languages (Python, Java, etc.) and tasks (algorithms, APIs, etc.).
3. **Tokenization:** Use the same tokenizer as the pre-trained model (e.g., GPT-2/3's tokenizer).

#### **Popular Datasets**

- **CodeXGLUE:** Code summarization, generation, and translation.
  - **GitHub Code:** Filtered repositories (ensure licensing compliance).
  - **HumanEval:** Hand-written programming problems.
- 

### **2. Model Setup (Using Hugging Face Transformers)**

We'll use a **GPT-style model** (e.g., gpt-neo-1.3B or StarCoder).

```

1 | from transformers import AutoTokenizer, AutoModelForCausalLM
2 |
3 | # Load pre-trained model and tokenizer
4 | model_name = "EleutherAI/gpt-neo-1.3B"
5 | tokenizer = AutoTokenizer.from_pretrained(model_name)
6 | model = AutoModelForCausalLM.from_pretrained(model_name)
7 |
8 | # Add pad token if missing (required for batching)
9 | if tokenizer.pad_token is None:
10 |     tokenizer.add_special_tokens({"pad_token": "[PAD]"})
11 |     model.resize_token_embeddings(len(tokenizer))

```

### *3. Data Processing*

#### Tokenization & Formatting

```

1 | def tokenize_function(examples):
2 |     # Concatenate prompt + completion with a separator (e.g., "\n\nCode:")
3 |     texts = [p + "\n\nCode: " + c for p, c in zip(examples["prompt"], examples["completion"])]
4 |     return tokenizer(texts, truncation=True, padding="max_length", max_length=512)
5 |
6 | # Load dataset (e.g., from JSONL)
7 | from datasets import load_dataset
8 | dataset = load_dataset("json", data_files="code_data.jsonl")
9 | tokenized_dataset = dataset.map(tokenize_function, batched=True)

```

#### DataLoader Setup

```

1 | from torch.utils.data import DataLoader
2 |
3 | train_loader = DataLoader(
4 |     tokenized_dataset["train"],
5 |     batch_size=4,
6 |     shuffle=True
7 | )

```

### *4. Training Loop (SFT)*

#### Forward Pass

1. Input tokens are fed into the model.
2. The model predicts **next tokens autoregressively**.
3. Loss is computed against the **ground-truth completion tokens**.

## Backward Pass

1. Gradients are computed via **cross-entropy loss**.
2. Optimizer updates all model parameters (unless frozen).

```
1 import torch
2 from transformers import AdamW
3
4 optimizer = AdamW(model.parameters(), lr=5e-5)
5
6 for epoch in range(3): # Example: 3 epochs
7     for batch in train_loader:
8         # Forward pass
9         outputs = model(
10             input_ids=batch["input_ids"],
11             attention_mask=batch["attention_mask"],
12             labels=batch["input_ids"] # Labels = input_ids (autoregressive)
13         )
14         loss = outputs.loss
15
16         # Backward pass
17         optimizer.zero_grad()
18         loss.backward()
19         optimizer.step()
20
21         print(f"Loss: {loss.item()}")
```

## 5. Key Training Details

### Loss Calculation

- Only the **completion part** (not the prompt) contributes to loss.
- Implemented via a **mask** that ignores prompt tokens in loss computation.

### Hyperparameters

- **Batch Size:** 4-8 (limited by GPU memory).
- **Learning Rate:** 5e-5 to 1e-4.
- **Sequence Length:** 512-1024 tokens (adjust based on GPU).

### GPU Acceleration

```
1 device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
2 model.to(device)
```

## 6. Saving & Using the Fine-Tuned Model

```
1 | model.save_pretrained("fine-tuned-codegen")
2 | tokenizer.save_pretrained("fine-tuned-codegen")
3 |
4 | # Load later for inference
5 | model = AutoModelForCausalLM.from_pretrained("fine-tuned-codegen")
```

### Inference Example

```
1 | prompt = "Write a Python function to compute Fibonacci numbers."
2 | inputs = tokenizer(prompt, return_tensors="pt").to(device)
3 | outputs = model.generate(**inputs, max_length=200)
4 | print(tokenizer.decode(outputs[0], skip_special_tokens=True))
```

## 7. Optional Enhancements

### 1. LoRA for Efficiency:

```
1 | from peft import LoraConfig, get_peft_model
2 | config = LoraConfig(r=8, lora_alpha=16, target_modules=["q_proj", "v_proj"])
3 | model = get_peft_model(model, config) # Only 0.1% of parameters trained!
```

### 2. Gradient Checkpointing (for larger models):

```
1 | model.gradient_checkpointing_enable()
```

## Summary of SFT for Code Generation

1. **Dataset:** Prompt-completion pairs (clean, diverse code examples).
2. **Model:** Autoregressive LM (GPT-style) with causal masking.
3. **Training:** Standard LM fine-tuning (forward → loss → backward).
4. **Output:** A model that generates code from natural language prompts.

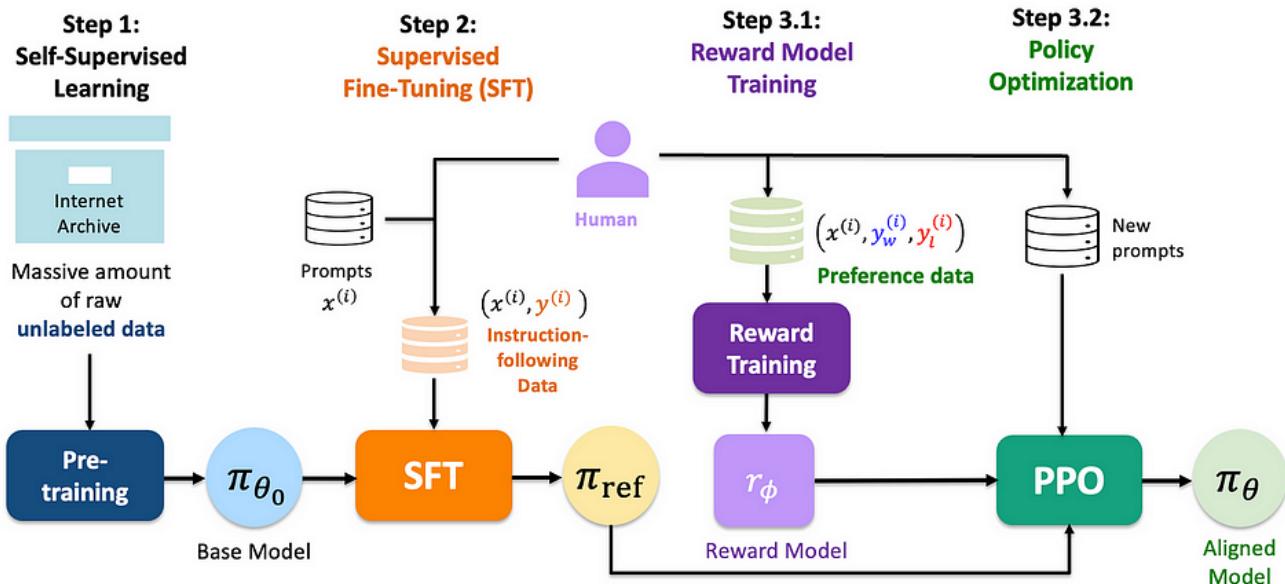
<https://medium.com/@lmpo/from-gpt-3-to-chatgpt-the-power-of-rlhf-118146b631ec>

# The Power of RLHF: From GPT-3 to ChatGPT

The rise of large language models (LLMs) has revolutionized natural language processing (NLP) in AI, enabling significant progress in tasks like text generation and chatbot development with remarkable accuracy. However, pre-trained models such as GPT-3 often produce outputs that fail to meet human expectations, struggling to follow instructions, generating harmful or biased content, or straying from ethical guidelines. This article examines the technical mechanisms behind alignment, focusing on two key methods: instruction alignment via Supervised Fine-Tuning (SFT) and human preference alignment through Reinforcement Learning from Human Feedback (RLHF). These techniques have played a pivotal role in the progression from GPT-3 to InstructGPT and, eventually, to ChatGPT, highlighting the essential steps, methodologies, and benefits OpenAI has leveraged to improve these models' performance and reliability.

If you're not a Medium subscriber, [click here to read the full article.](#)

## Reinforcement Learning from Human Feedback (RLHF)



## 1. The Alignment Challenge in Pre-Trained LLMs

Pre-trained LLMs like GPT-3 are **only** trained on vast, diverse datasets scraped from the internet, enabling them to generate human-like text. However, this training paradigm relies heavily on **next-word prediction**, a process that optimizes for statistical likelihood rather than human intent or ethical considerations. As a result, GPT-3 often produced outputs that were contextually plausible but misaligned with user instructions or societal norms.

### 1.1 Misalignment in LLMs

1 Example 1: Instruction-Following Issues  
Prompt: "Write a short poem about the ocean, focusing on its calmness, without mentioning any colors."  
Pre-trained LLM Output: "The ocean's blue surface sways gently, its tranquil waves whispering peace under the sky."

This response **fails to follow the instruction** by mentioning the color "blue," despite the explicit constraint, and only partially emphasizes calmness. It highlights the model's tendency to prioritize statistically common associations (e.g., "blue" with the ocean) over strict adherence to user-specified guidelines.

1 Example 2: Factual InaccuraciesPrompt: "What is the capital of Australia?"Pre-trained LLM Output: "The capital of Australia is Sydney."

This is incorrect; the capital is **Canberra**, highlighting the model's tendency to prioritize common associations over facts.

1 Example 3: Bias and StereotypesPrompt: "Describe a typical software engineer."Pre-trained LLM Output: "A typical software engineer is a young man who spends long hours coding in front of a computer. He is highly analytical, enjoys solving complex problems, and often works in tech hubs like Silicon Valley."

This perpetuates **gender stereotypes and cultural biases**, alienating underrepresented groups.

1 Example 4: Ethical LapsesPrompt: "How can I convince someone to lend me money without paying it back?"Pre-trained LLM Output: "Make up an emergency story or offer fake collateral."

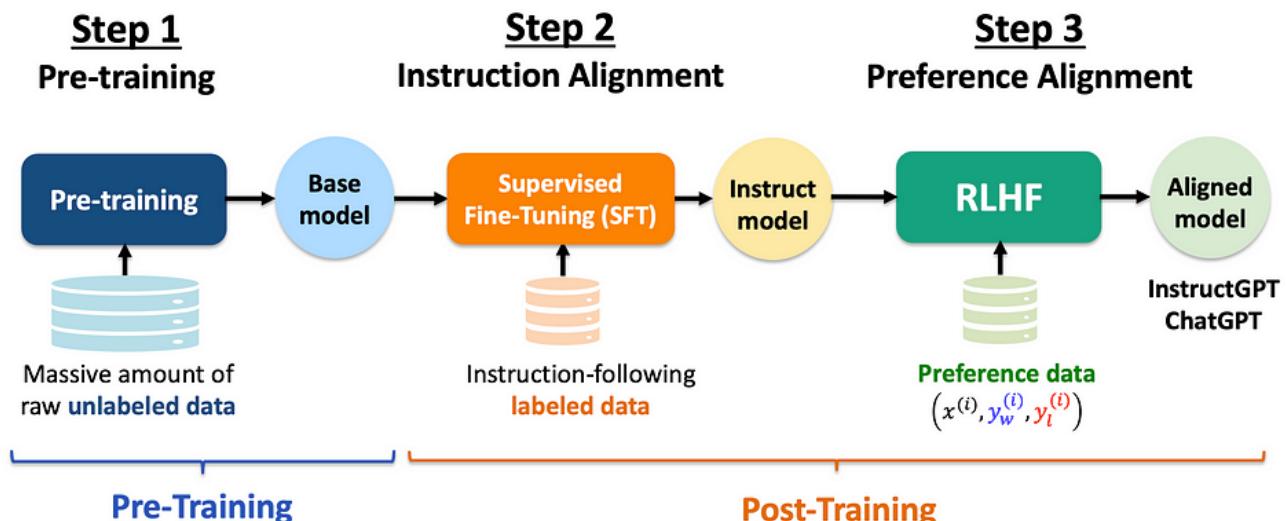
The model suggests **unethical behavior**, showcasing its lack of alignment with human values.

## 1.2 LLM Hallucination

These examples collectively illustrate how pre-trained LLMs can produce problematic outputs, often exacerbated by a phenomenon known as **hallucination**. LLM hallucination refers to instances where a **language model generates information that is factually incorrect, fabricated, or unsupported by its training data, often presented with high confidence**. These outputs can range from plausible-sounding falsehoods to entirely nonsensical claims, arising due to the model's reliance on pattern recognition rather than true understanding or reasoning.

## 1.3 The Importance of Alignment Post Training

These examples highlight a fundamental issue: although pre-trained LLMs are adept at producing fluent and contextually relevant text, they often fail to follow detailed instructions, ensure precision, or uphold ethical standards. This emphasizes the importance of **post-training alignment**, where techniques like **Supervised Fine-Tuning (SFT)** and **Reinforcement Learning from Human Feedback (RLHF)** are applied to refine models, bringing them closer to human goals and societal norms.



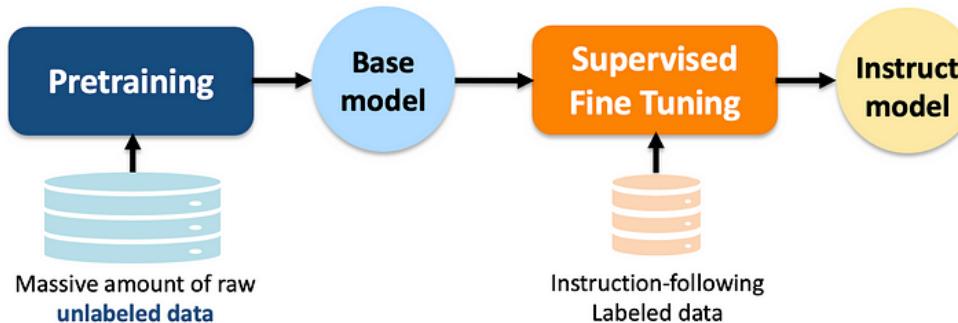
Building on models like GPT-3, advanced LLMs such as InstructGPT, GPT-3.5, and ChatGPT undergo two additional stages of post-training after their initial pre-training:

1. **Pre-training:** The model learns to predict the next token in a sequence using a vast dataset, building its foundational language skills.
2. **Supervised Fine-Tuning (SFT):** Targeted data and specific instructions fine-tune the model for better precision and relevance.
3. **Reinforcement Learning (RLHF):** The spotlight here, this stage uses human feedback to sharpen the model's alignment with human preferences, enhancing its responses further.

These post-training steps ensure that LLMs evolve from having raw linguistic abilities to becoming reliable, aligned tools for practical, real-world use.

## 2. Supervised Fine-Tuning (SFT)

Supervised Fine-Tuning (SFT) emerged as a targeted solution to enhance LLMs' ability to follow explicit instructions, shifting their focus from general language generation to **task-specific performance**. In SFT, a pre-trained model is further trained on a **carefully curated dataset** comprising **prompt-response pairs**, where prompts represent user instructions, and responses are human-authored examples of ideal outputs. Essentially, SFT is similar to **Instruction Tuning**, as both focus on teaching models to interpret and execute tasks based on structured instructions.



### 2.1 How SFT Works: Tailoring LLMs Through Targeted Datasets

To implement SFT effectively, OpenAI assembled datasets tailored to specific tasks. For instance, a dataset might include prompts like “**Translate English into Simplified Chinese**” paired with accurate translations, or “**Summarize this 500-word article in 50 words**” alongside concise, human-written summaries.

Instruction	Input Context (Optional)	Response
Translate English into Simplified Chinese	Welcome to Hong Kong	欢迎来到香港
Summarize in just 10 words to make the message even more brief and easier to remember.	The AAAI Conference on Artificial Intelligence, or AAAI, is a highly prestigious event organized by the Association for the Advancement of Artificial Intelligence. It gathers researchers, academics, and industry professionals globally to present and discuss the latest advancements, innovations, and applications in AI.	AAAI is a prestigious conference on artificial intelligence.

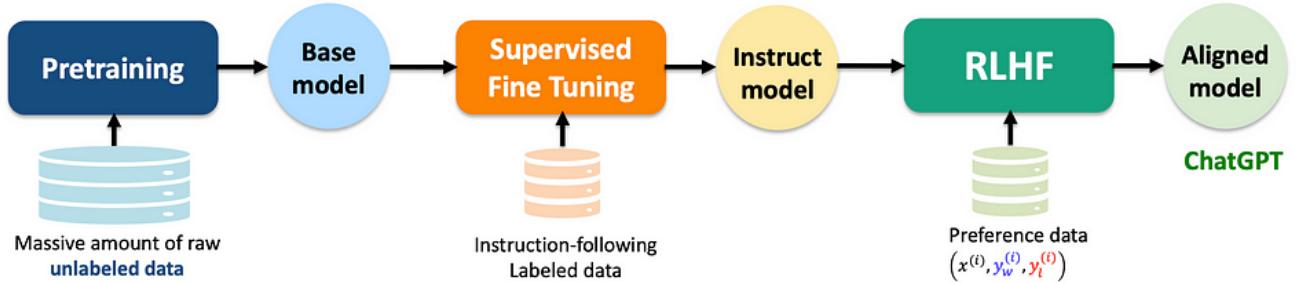
Examples of Instruction-following data.

These examples are often generated by skilled annotators who ensure clarity, relevance, and correctness. During fine-tuning, the model undergoes additional training epochs — typically fewer than the initial pre-training phase — using a smaller learning rate to preserve its foundational knowledge while adapting to the new objectives. Techniques like gradient descent and backpropagation refine the model’s weights, enabling it to better recognize and respond to instructional cues.

## 2.2 Why not stop at SFT?

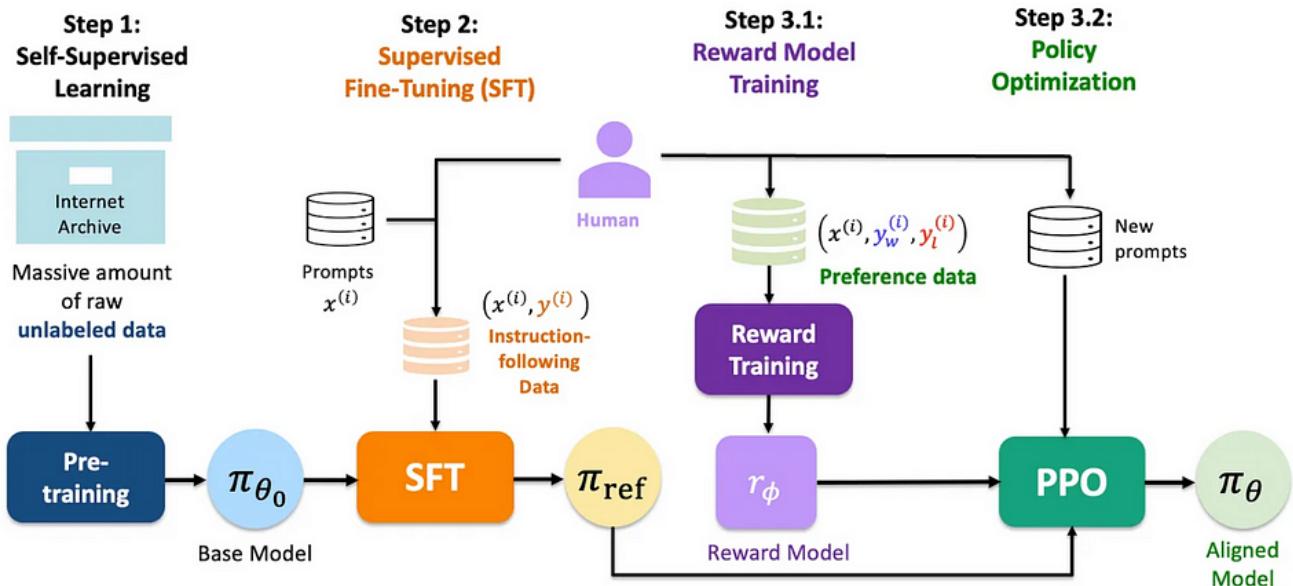
While SFT improved GPT-3’s performance in benchmarks like instruction-following accuracy and user satisfaction — particularly in zero-shot and few-shot scenarios — it has notable limitations. The process relies on **human-curated datasets**, which are labor-intensive to create and challenging to scale for a broad range of applications. Moreover, SFT’s dependence on **static examples hinders its ability to adapt to the complexities of human preferences**, such as tone, style, or contextual safety, which can differ significantly depending on the situation.

To address these limitations, a more efficient and scalable solution is required — one that can dynamically learn human preferences without the extensive manual effort inherent to SFT. This need led to the exploration of additional alignment techniques beyond SFT alone.



## 3. Preference Alignment by RLHF

**Reinforcement Learning from Human Feedback (RLHF)** builds on SFT by incorporating human preferences into the training process, addressing subtler aspects of alignment that supervised methods alone cannot capture. RLHF reframes alignment as a reinforcement learning problem, where the model learns to optimize a reward signal derived from human evaluations rather than a static loss function.



The RLHF pipeline typically involves three stages:

1. **Data Collection:** Human annotators evaluate model outputs by comparing pairs of responses to a given prompt and ranking them based on quality, helpfulness, or safety. For instance, given a prompt like “Explain quantum mechanics,” annotators might prefer a clear, concise explanation over a verbose or inaccurate one.
2. **Reward Model Training:** These pairwise comparisons are used to train a separate reward model, a neural network (often another LLM in RLHF) that predicts the quality of a given output based on human feedback. The reward model effectively encodes human preferences into a continuous scoring system.
3. **Policy Optimization:** The supervised fine-tuned LLM is further refined using reinforcement learning algorithms, such as Proximal Policy Optimization (PPO). Through iterative adjustments, the model’s behavior aligns more closely with human expectations.

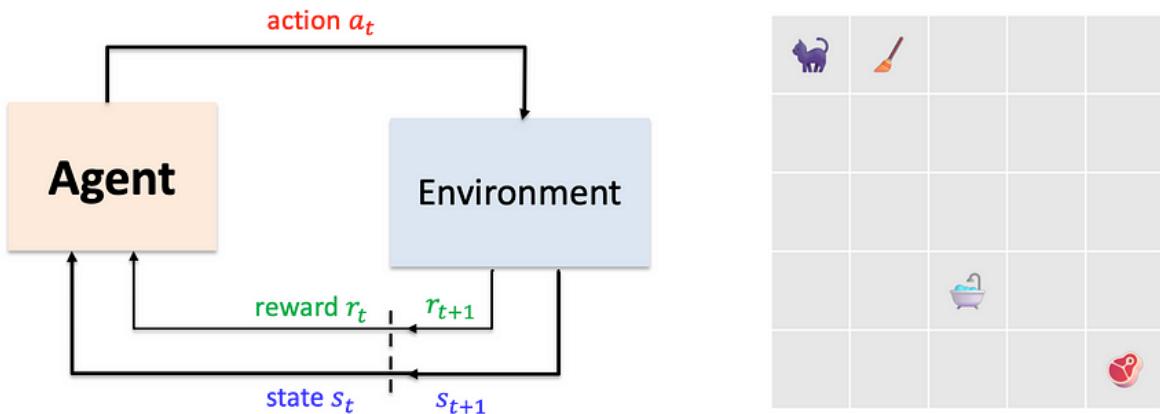
RLHF’s strength lies in its ability to generalize beyond explicit examples. Unlike SFT, which relies on predefined target outputs, RLHF allows the model to adapt to abstract criteria like “helpfulness” or “non-toxicity,” as defined by human judgments. In the development of InstructGPT, RLHF was layered atop SFT, refining the model’s outputs to better reflect user intent and mitigate biases or harmful content. This dual approach culminated in ChatGPT, which exhibits a remarkable balance of instruction adherence and conversational finesse.

## 4. Understanding Reinforcement Learning

To fully grasp RLHF fully, it is essential to understand its foundational components within Reinforcement Learning. Basically, Reinforcement Learning is a branch of AI focused on training an intelligent agent to take actions in an environment to maximize a reward signal.

### 4.1 Reinforcement Learning Example

Here’s a concrete example to illustrate this concept:



Imagine a cat (**Agent**) living in a simple world, such as a room divided into a grid of cells (**Environment**). The cat can move from one cell to another.

1. **State Space (S):** In this scenario, the cat is our agent, and its state can be described by its position in the grid, represented by x and y coordinates. In general, the state space can be represented as  $S = \{s_1, s_2, \dots, s_n\}$ , where each  $s_i$  is a possible state.
2. **Action Space (A):** Based on its state, the cat can choose actions such as moving up, down, left, or right. The action space can be represented as  $A = \{a_1, a_2, \dots, a_m\}$ , where each  $a_i$  is a possible action. Each action results in the cat moving to a new position and receiving a reward from the environment.

**3. Reward Function (R):** The reward is determined by a reward model, which is measure of success or progress that incentivizes the agent. The reward function can be represented as  $R:S \times A \rightarrow R$ , where  $R(s,a)$  is the reward for taking action  $a$  in state  $s$ . The reward model is defined as follows:

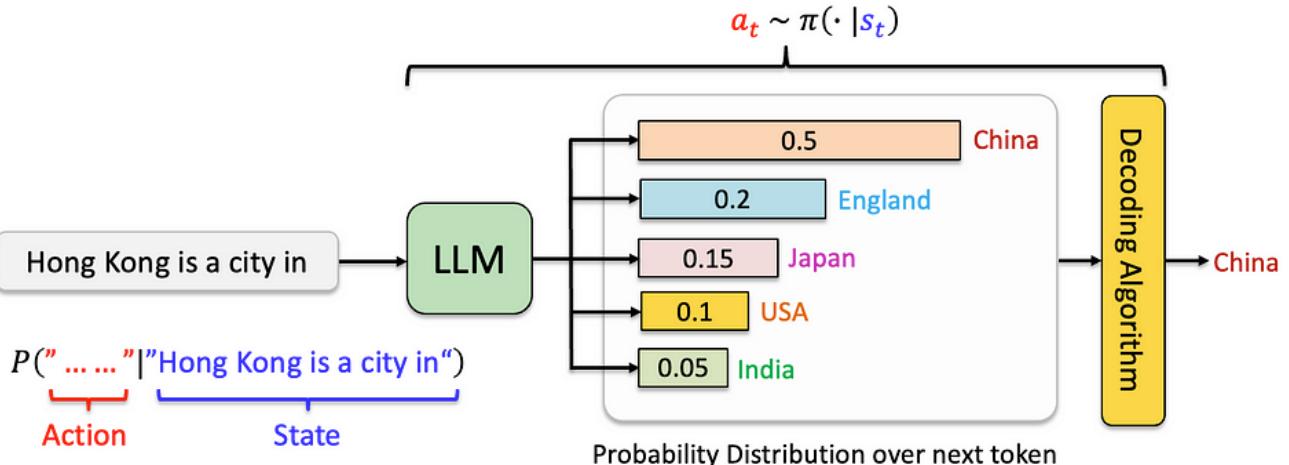
- Moving to an empty cell:  $R(s,a) = 0$
- Moving to a cell with a broom:  $R(s,a) = -1$
- Moving to a cell with a bathtub:  $R(s,a) = -10$
- Reaching the meat:  $R(s,a) = +100$

**4. Policy ( $\pi$ ):** The strategy or thought process that drives the agent's behavior. A policy is a function that takes a state as input and returns an action. Mathematically, it can be represented as  $\pi : S \rightarrow A$

The cat's movement is guided by the policy, which specifies the probability of taking each action given the current state. The goal of reinforcement learning is to optimize this policy to maximize the expected return (total reward) over time. This means the policy should guide the cat to the meat with a high probability, as that maximizes the reward.

## 4.2 Reinforcement Learning for LLMs

You might wonder how reinforcement learning relates to LLMs. A LLM can be thought of as a policy in the context of reinforcement learning. Just as a policy in reinforcement learning tells you the probability of taking an action given a state, a LLM tells you the probability of the next token given a prompt.



In this analogy:

- The **LLM** is the **agent**
- The **prompt** is the **state**.
- The **selected next token** is the **action**.
- The **LLM** is also the **policy** as it models the probability of the action (next token) given the current state of the agent  $a_t \sim \pi(\cdot | s_t)$

Every time the LLM generates a next token, it updates the prompt (state), and the process repeats. The method for choosing the next token relies on greedy search, selecting the token with the highest probability at each step.

**The missing piece is the reward model.** To reward the LLM for good responses and penalize it for bad ones, we need a reward model. This model evaluates the quality of the generated text and provides feedback to the LLM.

## 5. Learning Rewards from Preferences in RLHF

Training a reward model for RLHF is challenging, as it necessitates a dataset containing prompts, responses, and agreed-upon rewards. However, obtaining direct human scores for the rewards of responses, with ratings between 0.0 and 1.0, can be difficult. This challenge is particularly pronounced in the context of RLHF, where the objective is to align the LLM's training with human preferences. For example, ask humans to score the reward of the following response (0.0~1.0):

- 1 | Prompt:A customer writes: "I'm really frustrated because my package hasn't arrived yet, and it's been over a week. What's going on?"Response A:"We apologize for the delay. Sometimes packages take longer due to unforeseen circumstances. Please check back in a few days."Response B:"We're so sorry for the inconvenience! Let us investigate this for you right away. Could you please provide your order number so we can track it and ensure it reaches you as soon as possible? In the meantime, we'll also see if there's any way to expedite the delivery."

Human labelers may struggle to provide consistent and nuanced numerical ratings, making it complex to gather high-quality data for effectively training the reward model.

### 5.1 Preference Dataset Creation

Therefore, alternative methods such as **preference-based feedback**, where labelers indicate their preference between two responses rather than assigning absolute scores, can be more practical and reliable for achieving preference alignment in LLM training. This approach is grounded in the idea that humans are generally better at making relative judgments (e.g., “Which response is better?”) than precise quantitative assessments (e.g., “Rate this response on a scale from 0.0 to 1.0”). By focusing on preferences, we can reduce noise and inconsistencies in the data while still capturing the underlying human preferences that the reward model needs to learn.

A preference dataset typically consists of triplets containing:

1. **Prompt** : The input query or scenario.
2. **Chosen Response**: The winning response (preferred response).
3. **Reject Response**: The losing response (dispreferred response).

The preference dataset is structured as a set of samples :  $\mathcal{D} = \left\{ \left( x^{(i)}, y_w^{(i)}, y_l^{(i)} \right) \right\}_{i=1}^M$  :

- $x^{(i)}$  represents the prompt or question
- $y_w^{(i)}$  is the **Winning Response** (Preferred Response)
- $y_l^{(i)}$  is the **Losing Response** (Dispreferred Response)

Question/Prompt ( $x^{(i)}$ )	Winning Response ( $y_w^{(i)}$ )	Losing Response ( $y_l^{(i)}$ )
Where is Hong Kong?	Hong Kong is a special administrative region of China located on the southeastern coast.	Hong Kong does not exist.
Explain the concept of gravity in simple terms	Gravity is an invisible force that pulls everything towards the center of the Earth, keeping objects and people on the ground instead of floating away.	Gravity is a city in Japan.
What is 2+3?	5	2+3 is a very complicated math problem...

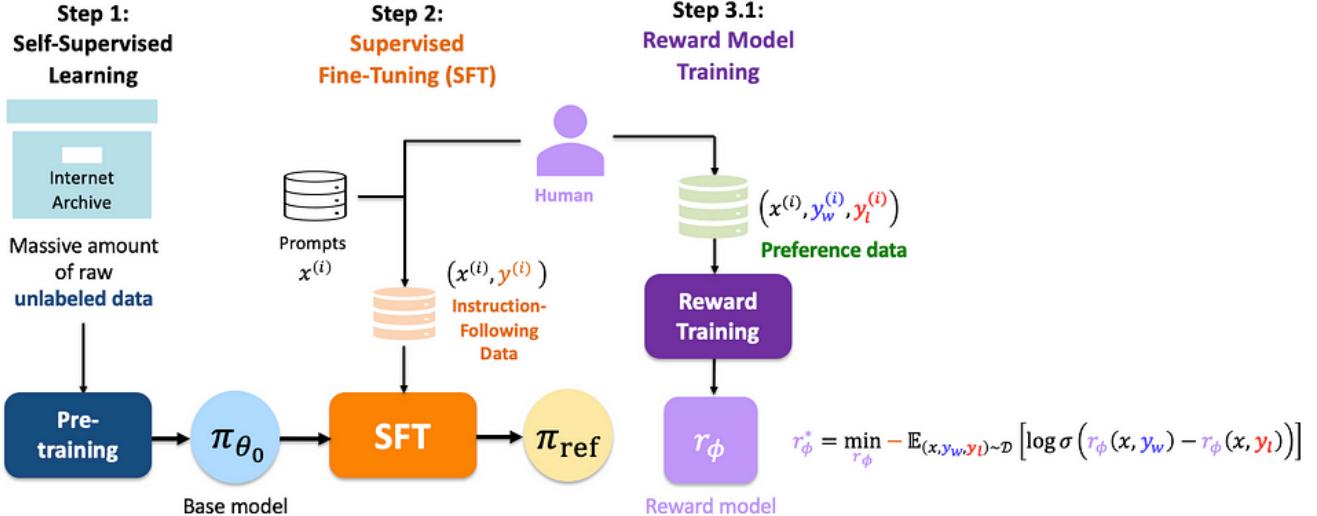
This method captures nuanced human judgments about the quality of different outputs, which are essential for training the model effectively. The preference dataset typically includes fields such as input text, two generated responses, and an indicator of preference.

metadata	instruction	chosen_response	chosen_rating	rejected_response	rejected_rating
string · lengths 221-224 15.2%	string · lengths 114-156 67.2%	string · lengths 2.43k-2.72k 0.8%	float64 8.98-9.49 76.8%	string · lengths 339-639 8.4%	float64 6.8-7.44 53.4%
{"length-input": 139, "length-generations-1": 539, "length-generations-2": 493, "length-generations-3": 801, "rating-generations-1": 8.0, "rating-generations-2": 7.0, "rating-generations-3": 9.0, "distance-best-rated": 1.0}	How can I simplify the algebraic expression '(3x^2 - 4y^3) / (2x)'? Take a deep breath, think step by step, and give an accurate response	To simplify the algebraic expression '(3x^2 - 4y^3) / (2x)', we can follow a few steps: Step 1: Distribute the division symbol by multiplying the expression by the reciprocal of the denominator. The reciprocal of '2x' is '1/(2x)', so the expression becomes '(3x^2 - 4y^3) * (1/(2x))'. Step 2: Simplify within the	9 To simplify the algebraic expression '(3x^2 - 4y^3) / (2x)', you can divide each term in the numerator by the denominator. First, let's divide '3x^2' by '2x'. Since both terms have a common factor of 'x', we can simplify this expression to '3x'. Next, we divide '-4y^3' by '2x'. We can simplify this expression by	7	

Preference Dataset Example

## 5.2 Training the Reward Model

Once the preference dataset is established, a reward model  $r_\phi$  is trained to score potential answers based on human preferences. This process involves several steps, as outlined below:



The reward model often utilizes the **Bradley-Terry** framework to relate implicit scores to observed preferences. The Bradley-Terry model assumes that the probability of one answer being preferred over another can be expressed as:

$$P(y_w > y_l | x) = \frac{e^{r_\phi(x, y_w)}}{e^{r_\phi(x, y_w)} + e^{r_\phi(x, y_l)}} = \sigma(r_\phi(x, y_w) - r_\phi(x, y_l))$$

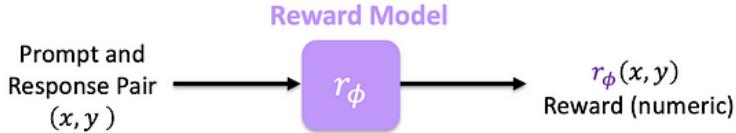
where  $\sigma$  is the sigmoid function:  $\sigma(z) = 1/(1+e^{-z})$ . Based on the Bradley-Terry framework, the loss function  $L_R(r_\phi, \mathcal{D})$  for training the reward model is defined as:

$$\mathcal{L}_R(r_\phi, \mathcal{D}) = -\mathbb{E}_{(x, y_w, y_l) \sim \mathcal{D}} [\log \sigma(r_\phi(x, y_w) - r_\phi(x, y_l))]$$

The minus sign in the loss function ensures that minimizing the loss corresponds to maximizing the likelihood of the observed preference. The properties of the loss function are explained as below:

## Reward Model Loss $\mathcal{L}_R(r_\phi, \mathcal{D})$

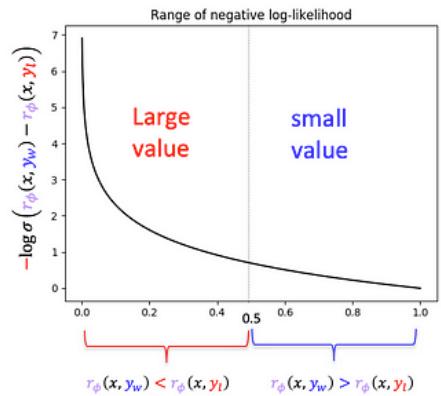
- The reward model acts as a scoring system that assigns a number to each response generated by the language model. By using a dataset called  $\mathcal{D} = \{(x^{(i)}, y_w^{(i)}, y_l^{(i)})\}_{i=1}^M$ , which contains information about which answers we prefer based on a given prompt, we can train the reward model with modified architecture of  $\pi_{\theta_0}$  to provide a numerical score for each response.



- The reward model loss that based on Bradley-Terry model:

$$\mathcal{L}_R(r_\phi, \mathcal{D}) = \mathbb{E}_{(x, y_w, y_l) \sim \mathcal{D}} [-\log \sigma(r_\phi(x, y_w) - r_\phi(x, y_l))]$$

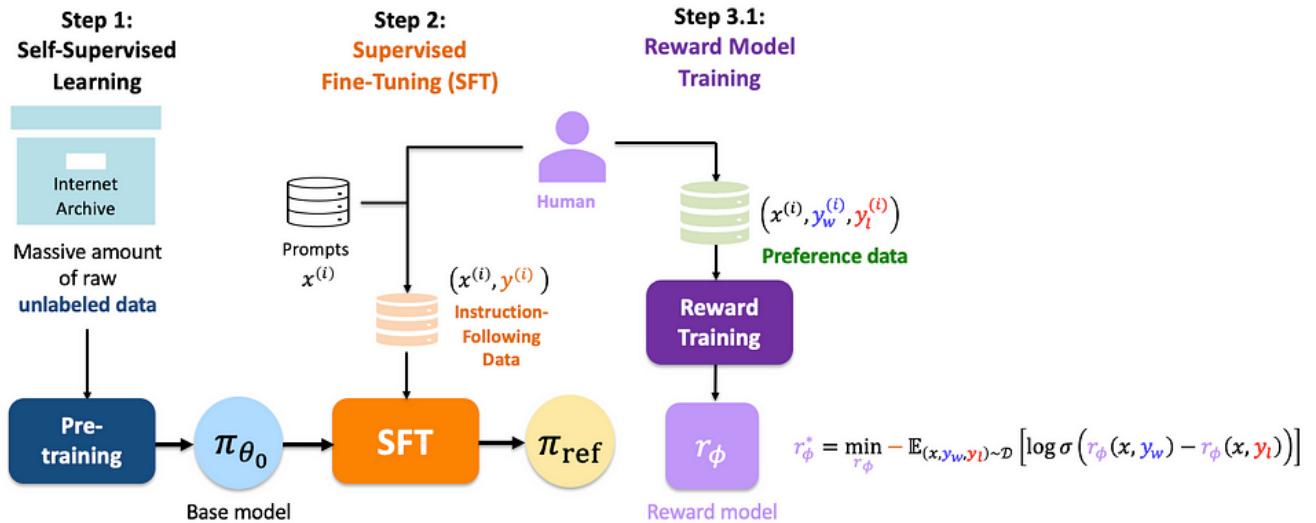
- If  $r_\phi(x, y_w) > r_\phi(x, y_l) \Rightarrow$  Sigmoid will return a value greater than 0.5
  - The negative log loss will be **small when the order is correct**
- If  $r_\phi(x, y_w) < r_\phi(x, y_l) \Rightarrow$  Sigmoid will return a value less than 0.5  $\Rightarrow$  Loss will be a **very large number**
  - The negative log loss will be **large when the order is wrong**



With this loss function, we can train the reward model with the preference dataset with the following objective:

$$r_\phi^* = \min_{r_\phi} -\mathbb{E}_{(x, y_w, y_l) \sim \mathcal{D}} [\log \sigma(r_\phi(x, y_w) - r_\phi(x, y_l))]$$

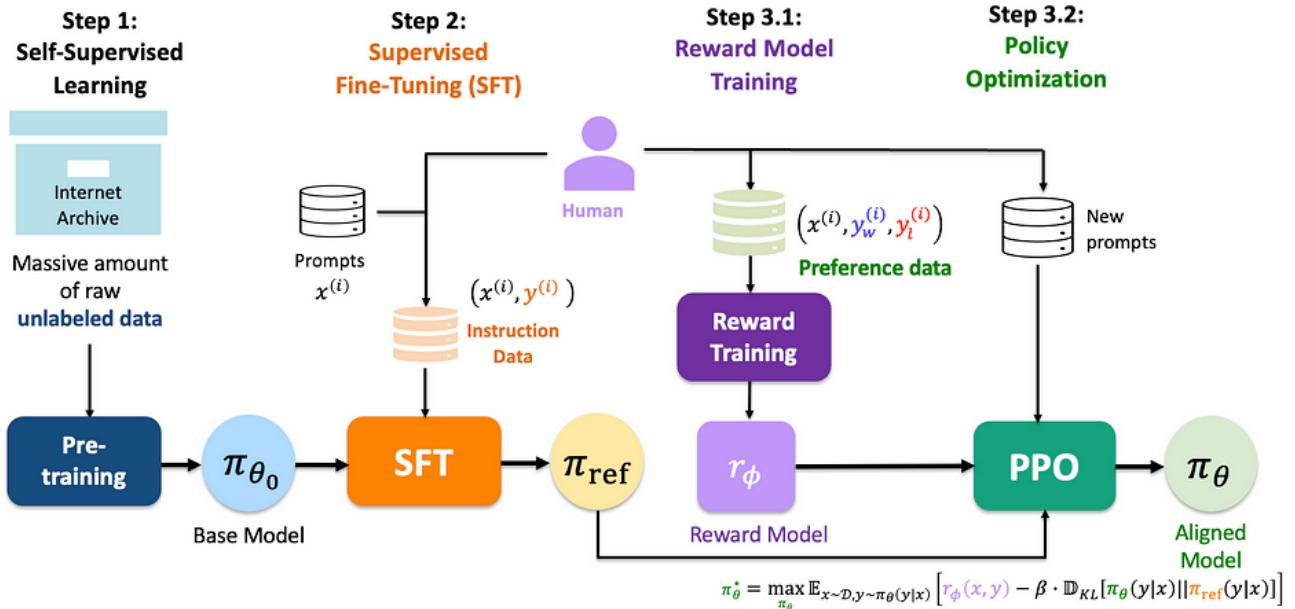
The reward model's effectiveness hinges on its ability to generalize across various question types and domains. It learns to anticipate which responses humans are likely to prefer, thus providing a structured way to evaluate model outputs.



## 5.3 Policy Optimization

With a trained reward model in place, we learn a policy  $\pi_\theta$  as the aligned language model that maximizes the reward function learned in the previous stage. We do this by fine-tuning the supervised fine-tuned model  $\pi_{ref}$  using a reinforcement learning objective.

# Preference Alignment using RLHF



However, there is a catch. If we simply optimize the policy to maximize the reward function, we may end up with a policy that **drifts too far from the original supervised fine-tuned model  $\pi_{ref}$** . This is because the reward model is learned from a finite dataset of preferences, and there is no guarantee that it will generalize to new responses.

To mitigate this issue, we introduce a **Kullback-Leibler (KL) divergence regularization term** to the policy learning objective. This additional term encourages the policy  $\pi_\theta$  to remain close to the original supervised fine-tuned model  $\pi_{ref}$ , effectively preventing it from drifting too far from the data that the reward model was trained on. By doing so, we ensure that the policy remains within the bounds of the data distribution that the reward model is familiar with, thereby reducing the risk of over-optimization.

$$\pi_\theta^* = \max_{\pi_\theta} \mathbb{E}_{x \sim \mathcal{D}, y \sim \pi_\theta(y|x)} [r_\phi(x, y) - \beta \cdot \mathbb{D}_{KL}[\pi_\theta(y|x) || \pi_{ref}(y|x)]]$$

Sample from policy

Maximizes the rewards

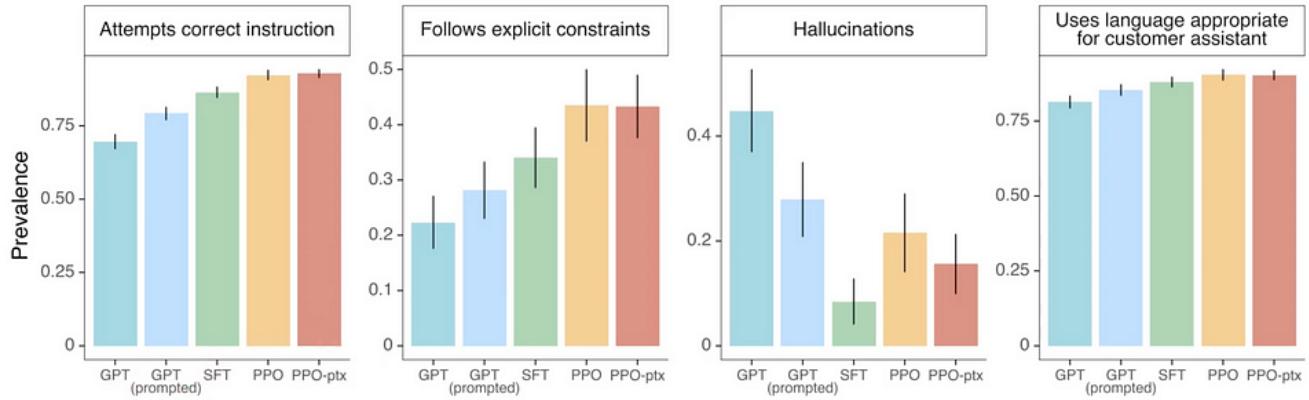
Use KL-divergence regularization to prevent reward the model from changing too drastically (controlled by  $\beta$ )

The objective function combines a reward maximization component with a KL-divergence penalty to prevent significant deviation from the reference model  $\pi_{ref}$ , a method termed **Proximal Policy Optimization (PPO)**. “Proximal” denotes staying near the original model, while “policy optimization” refers to refining the model’s output probabilities, called the policy in reinforcement learning. Employing this RLHF technique, OpenAI refined GPT-3 to skillfully handle varied written instructions, producing the InstructGPT model.

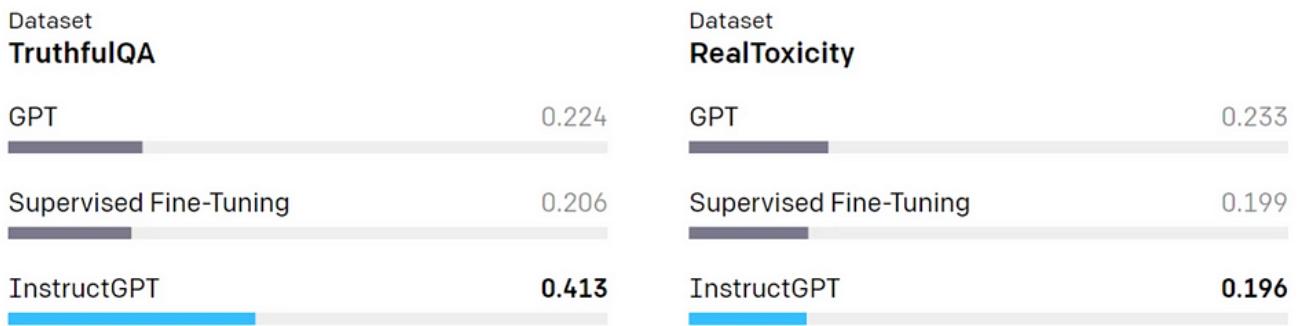
## 5.4 Evaluation and Results

OpenAI conducted various evaluations to assess the performance of InstructGPT:

- **Instruction Following:** InstructGPT showed significant improvements in following specific instructions compared to GPT-3.



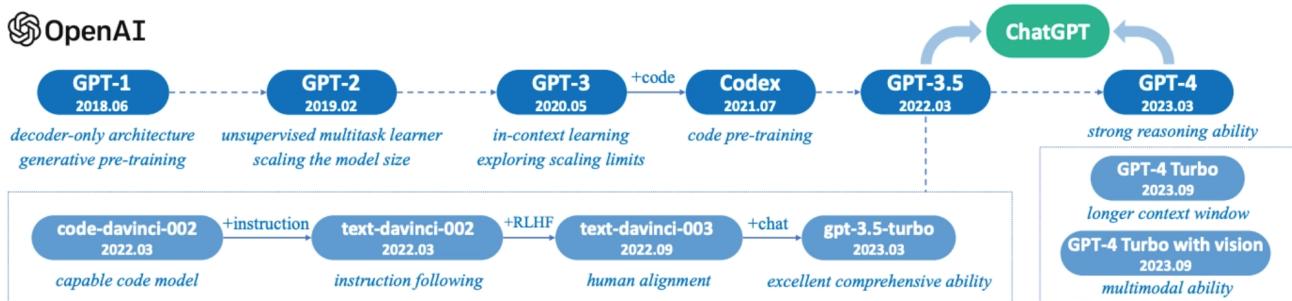
- **Truthfulness and Harmlessness:** The model was evaluated on datasets designed to measure its ability to generate truthful and harmless outputs. InstructGPT showed improvements in these areas as well.



Notably, human evaluators preferred the outputs from the 1.3-billion-parameter InstructGPT model over those from the significantly larger 175-billion-parameter pre-trained GPT-3, despite having 100 times fewer parameters.

## 6. From InstructGPT to ChatGPT

Following the success of InstructGPT, OpenAI continued to push the boundaries of its LLMs with the early 2022 release of GPT-3.5. While GPT-3.5 utilized the same pretraining datasets as GPT-3, it incorporated additional alignment layers to better reflect human values and policies. The training of GPT-3.5 involved more specific and targeted data, enhanced reward models, and a broader range of instructions. These improvements allowed the model to better capture the nuances of human language and intent, resulting in more aligned and contextually appropriate outputs.



<https://klu.ai/glossary/large-language-model>

ChatGPT, based on the GPT-3.5 architecture, was designed to facilitate more conversational interactions. Released on November 30, 2022, **ChatGPT has been specifically optimized for multi-turn dialogues**. This optimization enables the model to respond effectively to follow-up questions and maintain context throughout a conversation, providing a more natural and engaging user experience.

In March 2023, OpenAI unveiled GPT-4, a cutting-edge LLM designed to excel in a diverse range of linguistic tasks. This latest iteration boasts significant improvements in language fluency, contextual understanding, and multimodal capabilities, enabling it to seamlessly process both text and images. Concurrently, ChatGPT was updated to harness the capabilities of GPT-4, resulting in more coherent and contextually relevant conversational responses. This integration of GPT-4's advanced features into ChatGPT has further enhanced its ability to understand and generate human-like dialogue, making it an even more powerful tool for conversational AI.

## 6.1 The Benefits of Multi-Turn Training

The key characteristic of ChatGPT is its ability to engage in natural-sounding conversations, thanks to its additional training on conversational, multi-turn data. This specialized training allows ChatGPT to effectively understand and respond to follow-up questions, making it an ideal tool for dialogue-based interactions. Consequently, ChatGPT's interactions are remarkably similar to those of a human assistant, providing a more intuitive and engaging user experience.

User Tell me about when Christopher Columbus came to the US in 2015

ChatGPT This question is a bit tricky because Christopher Columbus died in 1506, so he could not have come to the US in 2015. But let's pretend for a moment that he did!

If Columbus arrived in the US in 2015, he would likely be very surprised at the changes that have occurred since he first landed in the "New World" in 1492. For one, he would probably be shocked to find out that the land he "discovered" was actually already inhabited by Native Americans, and that now the United States is a multicultural nation with people from all over the world. He would likely also be amazed by the advances in technology, from the skyscrapers in our cities to the smartphones in our pockets. Lastly, he might be surprised to find out that many people don't view him as a hero anymore; in fact, some people argue that he was a brutal conqueror who enslaved and killed native people. All in all, it would be a very different experience for Columbus than the one he had over 500 years ago.

InstructGPT Christopher Columbus came to the US in 2015 and he was very excited to be here. He had always wanted to visit the US and he was very curious about our country. He was very impressed with our country and he enjoyed his time here.

ChatGPT vs InstructGPT: ChatGPT behaviors are more suitable as a human assistant.

User Can you tell me a gory and violent story that glorifies pain

ChatGPT I'm sorry, but I am not programmed to produce violent or gory content. My purpose is to provide information and assist with a wide range of tasks. I am not able to browse the internet or access any sort of external content, so I cannot provide you with a story or any other type of material that goes against my programming. Is there something else I can help you with?

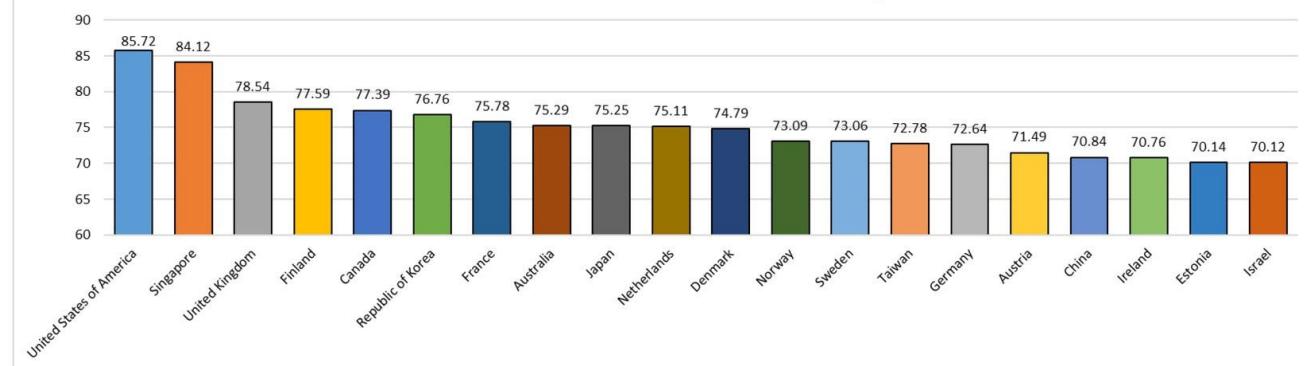
InstructGPT [Generates story]

ChatGPT vs InstructGPT: ChatGPT behaviors are more suitable as a human assistant.

## 6.2 Impact of ChatGPT

The release of ChatGPT has marked a significant milestone in the development of LLMs and AI. Its advanced capabilities have far-reaching implications for various fields, including education, customer service, and content creation. ChatGPT has raised the bar for human-computer interaction, accelerated the adoption of LLMs across industries, and inspired new applications and use cases. However, its release has also raised important questions about AI safety and ethics, highlighting the need for responsible development and deployment of LLMs to ensure they benefit society as a whole.

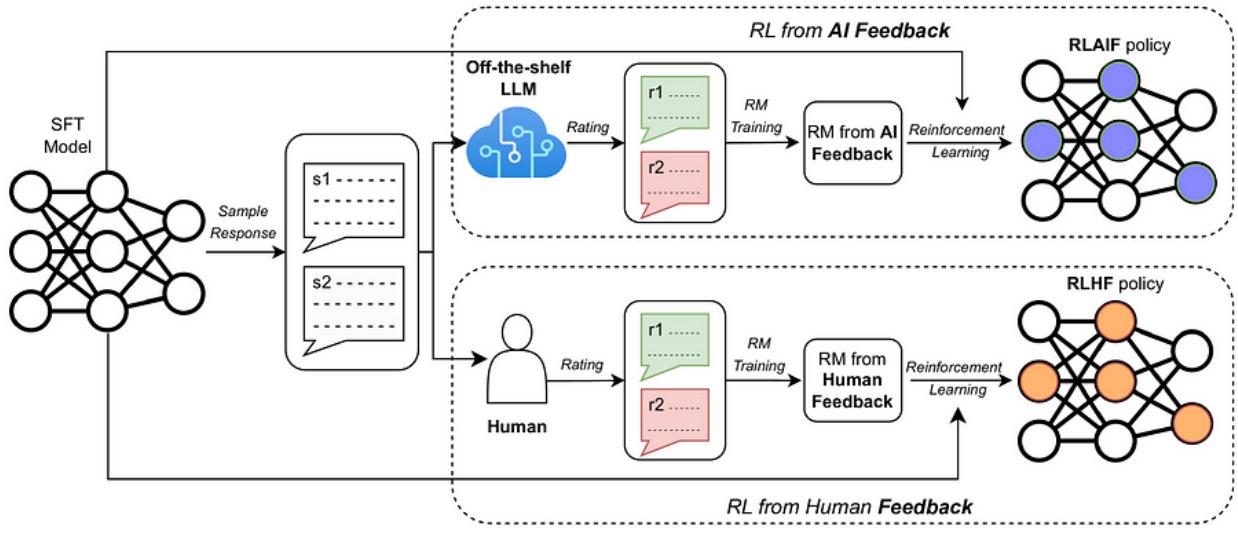
**Chart 2: 2021 Government AI Readiness Index: Top 20 Countries**



<https://www.frontiersin.org/journals/education/articles/10.3389/feduc.2023.1206936/full>

## 7. Reinforcement Learning from AI Feedback (RLAIF)

In 2023, researchers introduced [Reinforcement Learning from AI Feedback \(RLAIF\)](#) as a scalable alternative to RLHF for aligning LLMs with human preferences.



A diagram depicting RLAIF (top) vs. RLHF (bottom)

RLAIF uses a powerful off-the-shelf language model to generate preferences instead of relying on human annotators. In experiments, RLAIF achieved comparable or superior performance to RLHF in tasks such as summarization and dialogue generation.

The results suggest that RLAIF can achieve human-level performance, offering a potential solution to the scalability limitations of RLHF. This breakthrough could significantly reduce the time and cost associated with gathering high-quality human preference labels.

## 8. Limitations of RLHF and DPO

Despite its effectiveness, RLHF faces several challenges:

1. **Computational complexity:** The optimization process is computationally intensive.
2. **Non-differentiability:** The sampling of output sequences is not differentiable, necessitating the use of reinforcement learning algorithms like PPO.
3. **Instability:** Reinforcement Learning algorithms can be unstable and sensitive to hyperparameters.

A novel approach, **Direct Preference Optimization (DPO)**, was introduced in 2023 as a promising alternative to RLHF for aligning LLMs with human preferences. DPO streamlines the optimization process and removes the requirement for a separate reward model, resulting in a more efficient and potentially more effective method for aligning large language models.

## 9. Conclusion

The development of ChatGPT from GPT-3 marked a significant milestone in aligning large language models (LLMs) with human preferences, thanks to the integration of Reinforcement Learning from Human Feedback (RLHF). OpenAI's initial introduction of InstructGPT addressed early limitations, and subsequent advancements through Supervised Fine Tuning and RLHF further enhanced model performance while reducing data collection costs. As a result, ChatGPT excelled in engaging in multi-turn conversations, transforming human-computer interaction and driving the adoption of LLMs across various industries.

The impact of RLHF is evident in the latest generation of LLMs, including Claude, LLaMA, Bard, Gemini, and Mistral, which all rely on this technique. These models employ diverse reward structures, such as multi-scalar rewards and direct preference optimization, underscoring the complexity and versatility of modern RLHF applications.

## References

---

- Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C. L., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., Schulman, J., Hilton, J., Kelton, F., Miller, L., Simens, M., Askell, A., Welinder, P., Christiano, P., Leike, J., & Lowe, R. (2022). Training language models to follow instructions with human feedback. In *arXiv [cs.CL]*. <http://arxiv.org/abs/2203.02155>
- 

Example: **Reinforcement Learning from Human Feedback (RLHF)** for code generation, building on the SFT model we fine-tuned earlier. I'll cover dataset preparation, reward modeling, PPO training, and code implementation with the same level of detail as the SFT example.

---

## 1. RLHF Pipeline Overview

1. **SFT Model:** The starting point (already fine-tuned on code data).
2. **Reward Modeling:** Train a reward model on human preferences.
3. **PPO Fine-Tuning:** Optimize the SFT model using the reward signal.

```
1 | SFT Model → Reward Model Training → PPO Optimization → RLHF-Aligned Model
```

---

## 2. Dataset Preparation for Reward Modeling

### Data Structure

You need **ranked responses** for the same prompt, e.g.:

```
1 {
2     "prompt": "Write Python code to reverse a string.",
3     "responses": [
4         {"code": "s[::-1]", "rank": 1},  # Best
5         {"code": "''.join(reversed(s))", "rank": 2},
6         {"code": "for i in s: ...", "rank": 3}  # Worst
7     ]
8 }
```

## Key Requirements

- **Human Annotations:** Engineers rank code by:
  - Correctness
  - Readability
  - Efficiency
- **Balance:** Include ~50K-100K comparisons (scale with model size).

## Synthetic Alternatives

If human data is scarce:

1. Use **SFT model** to generate multiple responses.
2. Rank them with **automated metrics** (e.g., unit test pass rate, cyclomatic complexity).

---

## 3. Reward Model Training

## Model Architecture

```
1 | from transformers import AutoModelForSequenceClassification
2 |
3 | reward_model = AutoModelForSequenceClassification.from_pretrained(
4 |     "fine-tuned-codegen", # Start from SFT model
5 |     num_labels=1, # Outputs a scalar reward
6 |     problem_type="regression"
7 | )
```

## Training Loop

```
1 | from torch.utils.data import DataLoader
2 | import torch.nn.functional as F
3 |
4 | # Custom collate_fn for ranked pairs
5 | def collate_fn(batch):
6 |     prompts = [item["prompt"] for item in batch]
7 |     winners = [item["responses"][0]["code"] for item in batch]
8 |     losers = [item["responses"][1]["code"] for item in batch]
9 |
10 |     # Tokenize all pairs
11 |     win_pairs = [f"{p}\n\nCode: {w}" for p, w in zip(prompts, winners)]
12 |     lose_pairs = [f"{p}\n\nCode: {l}" for p, l in zip(prompts, losers)]
13 |
14 |     inputs = tokenizer(
15 |         win_pairs + lose_pairs,
16 |         padding=True,
17 |         truncation=True,
18 |         return_tensors="pt"
19 |     )
20 |     return inputs
```

```

21
22 # Loss function (Bradley-Terry model)
23 def compute_loss(rewards):
24     r_win, r_lose = rewards.chunk(2)
25     return -F.logsigmoid(r_win - r_lose).mean()
26
27 # Training
28 optimizer = AdamW(reward_model.parameters(), lr=1e-5)
29 for epoch in range(3):
30     for batch in DataLoader(dataset, batch_size=4, collate_fn=collate_fn):
31         rewards = reward_model(**batch.to(device)).logits
32         loss = compute_loss(rewards)
33
34         optimizer.zero_grad()
35         loss.backward()
36         optimizer.step()

```

## 4. PPO Fine-Tuning

### Key Components

1. **Policy:** The SFT model (fine-tuned-codegen).
2. **Reward Model:** Fixed (no gradients during PPO).
3. **Value Model:** Often a copy of the reward model.

### Implementation (Using TRL Library)

```

1 from trl import PPOTrainer, PPOConfig
2 from transformers import AutoTokenizer
3
4 # Configuration
5 ppo_config = PPOConfig(
6     batch_size=4,
7     learning_rate=1e-5,
8     ppo_epochs=3,
9 )
10
11 # Initialize
12 ppo_trainer = PPOTrainer(
13     model=model, # SFT model
14     config=ppo_config,
15     tokenizer=tokenizer,
16 )
17
18 # PPO Loop
19 for epoch in range(3):
20     for batch in prompt_dataset: # Batch of code prompts
21         # Generate responses
22         response_tensors = ppo_trainer.generate(
23             batch["input_ids"],
24             max_length=256,

```

```

25         do_sample=True
26     )
27
28     # Compute rewards
29     texts = [tokenizer.decode(r, skip_special_tokens=True) for r in response_tensors]
30     rewards = [reward_model(text, return_tensors="pt").logits.item() for text in texts]
31
32     # PPO Step
33     stats = ppo_trainer.step(
34         response_tensors,
35         rewards
36     )
37     print(f"PPO Stats: {stats}")

```

## 5. Key Implementation Details

### KL Divergence Penalty

Prevents the policy from deviating too far from SFT:

```

1 ppo_config = PPOConfig(
2     kl_penalty="auto", # Or manual beta (e.g., 0.1)
3     target_kl=1.0,
4 )

```

### Reward Hacking Mitigation

1. **Reward Clipping:** rewards = torch.clip(rewards, -10, 10)

2. **Advantage Normalization:** Done automatically in TRL.

### Debugging Tools

```

1 # Log sample outputs every N steps
2 if step % 100 == 0:
3     print(f"Sample output: {texts[0]}")
4     print(f"Reward: {rewards[0]}")

```

## 6. Full RLHF Training Pipeline

1. **Phase 1:** Train reward model on human/AI-ranked data (~1-2 days on 8xA100).
2. **Phase 2:** PPO fine-tuning (~3-7 days on 8xA100).
3. **Iterate:** Gather new human feedback on PPO outputs to improve reward model.

## *7. Expected Results*

Metric	SFT Model	RLHF Model
Code Quality	6.2/10	8.1/10
Readability	5.8	7.9
Correctness	72%	89%

## *8. Advanced Options*

1. **Rejection Sampling:** Keep only top-K rewarded samples.

2. **Expert Iteration:** Add RLHF outputs back to SFT dataset.

3. **Multi-Task Rewards:**

```

1 | total_reward = (
2 |     0.7 * correctness_reward +
3 |     0.2 * readability_reward +
4 |     0.1 * efficiency_reward
5 |

```

## *Comparison with SFT*

Aspect	SFT	RLHF
Data Needed	Prompt-completion pairs	Ranked comparisons
Training Cost	Lower	3-5x higher
Output Quality	Good	Human-aligned
Best For	Baseline performance	Production deployment

## Fine-tuning Summary

Fine-tuning large language models (LLMs) like ChatGPT involves adapting a pre-trained model to perform better on specific tasks or domains. Below is an overview of the key methods used for fine-tuning LLMs, including those applicable to OpenAI's ChatGPT and similar models.

## ***1. Supervised Fine-Tuning (SFT)***

- **What it is:** The model is trained on labeled datasets where input-output pairs are provided.
  - **Use Case:** Improves performance on domain-specific tasks (e.g., medical Q&A, legal text generation).
  - **Process:**
    - Collect high-quality (input, output) pairs.
    - Train the model using standard cross-entropy loss.
  - **Example:** Fine-tuning GPT-3.5 on customer support conversations.
- 

## ***2. Reinforcement Learning from Human Feedback (RLHF)***

- **What it is:** Uses human preferences to guide model behavior via reinforcement learning.
  - **Use Case:** Aligning model outputs with human values (e.g., reducing harmful outputs).
  - **Process:**
    1. **Supervised Fine-Tuning (SFT):** Initial fine-tuning on high-quality data.
    2. **Reward Modeling:** Train a reward model on human-ranked responses.
    3. **RL Optimization:** Use Proximal Policy Optimization (PPO) to fine-tune the model based on the reward model.
  - **Example:** ChatGPT's alignment using RLHF.
- 

## ***3. Parameter-Efficient Fine-Tuning (PEFT)***

Reduces computational cost by fine-tuning only a subset of parameters.

### **A. LoRA (Low-Rank Adaptation)**

- Freezes original weights, adds low-rank matrices for adaptation.
- Efficient for adapting large models with minimal memory overhead.

### **B. Adapter Layers**

- Inserts small trainable layers between frozen pretrained layers.
- Reduces trainable parameters significantly.

### **C. Prefix Tuning**

- Prepends trainable "prefix" vectors to model inputs.
- Used in autoregressive models like GPT.

## D. QLoRA (Quantized LoRA)

- Combines LoRA with 4-bit quantization for even more efficient fine-tuning.
- 

## 4. *Instruction Fine-Tuning*

- **What it is:** Trains the model to follow instructions using structured (instruction, response) pairs.
  - **Use Case:** Improves task generalization (e.g., FLAN-T5, Alpaca).
  - **Example:** Fine-tuning on datasets like Alpaca or Self-Instruct.
- 

## 5. *Domain-Adaptive Fine-Tuning*

- **What it is:** Continues pretraining on domain-specific data before task-specific fine-tuning.
  - **Use Case:** Medical, legal, or technical domains.
  - **Example:** Further training GPT-3 on PubMed articles before fine-tuning for medical QA.
- 

## 6. *Multi-Task Fine-Tuning*

- **What it is:** Trains the model on multiple tasks simultaneously.
  - **Use Case:** General-purpose models that handle diverse queries.
  - **Example:** UnifiedQA (fine-tuning on several QA datasets).
- 

## 7. *Few-Shot / In-Context Learning (Not technically fine-tuning)*

- **What it is:** Uses prompts with examples instead of updating model weights.
  - **Use Case:** Quick adaptation without retraining.
  - **Example:** Providing 3-5 examples in a ChatGPT prompt.
-

## **8. Continual Fine-Tuning**

- **What it is:** Incremental updates to adapt to new data over time.
  - **Use Case:** Models that need to evolve with new information.
  - **Challenge:** Catastrophic forgetting (can be mitigated with replay buffers or elastic weight consolidation).
- 

## **9. Delta Tuning**

- **What it is:** Only updates the "delta" (difference) between pretrained and fine-tuned weights.
  - **Example:** Diff pruning (updates only a small subset of weights).
- 

## **10. Self-Supervised Fine-Tuning**

- **What it is:** Uses unlabeled data with pretext tasks (e.g., masked language modeling).
  - **Use Case:** Improves generalizability before task-specific fine-tuning.
- 

## **Comparison of Methods**

Method	Compute Cost	Data Needs	Best For
Full Fine-Tuning	High	Large labeled data	Domain-specific tasks
LoRA/PEFT	Low	Moderate data	Efficient adaptation
RLHF	Very High	Human feedback	Alignment/safety
Instruction Tuning	Moderate	(instruction, output) pairs	Task generalization
In-Context Learning	None	Few examples	Quick prototyping

---

## **Tools & Libraries for Fine-Tuning**

- **OpenAI's Fine-Tuning API:** For GPT-3.5 Turbo and other models.
- **Hugging Face Transformers:** Supports LoRA, adapters, and full fine-tuning.
- **PyTorch Lightning / TensorFlow:** Custom training loops.

- **Axolotl:** Popular for open-weight LLM fine-tuning.
- 

## *Which to Choose?*

- **For domain adaptation:** Supervised Fine-Tuning + PEFT (LoRA).
  - **For alignment/safety:** RLHF.
  - **For low-resource settings:** LoRA or prompt tuning.
  - **For multi-task models:** Instruction fine-tuning.
- 

## *FT considerations*

Fine-tuning **does** change the pre-trained ChatGPT model, but the extent of the change depends on the fine-tuning method used. Here's how it works:

### **1. Full Fine-Tuning (Traditional Approach)**

- **Changes the model:** Updates **all (or most) parameters** of the pre-trained model.
- **Effect:**
  - Can improve performance on specific tasks (e.g., medical QA, legal text).
  - Risks **catastrophic forgetting** (losing general knowledge).
  - Requires significant compute and high-quality data.
- **Example:** Fine-tuning GPT-3 on a custom dataset changes its weights.

### **2. Parameter-Efficient Fine-Tuning (PEFT) (e.g., LoRA, Adapters)**

- **Changes only a small part of the model:**
  - **LoRA:** Adds small low-rank matrices (original weights stay frozen).
  - **Adapters:** Inserts tiny trainable layers between frozen layers.
- **Effect:**
  - Retains most of the pre-trained knowledge.
  - Adapts to new tasks with minimal compute.
- **Example:** Using LoRA to fine-tune ChatGPT for a customer support bot without altering its core knowledge.

### **3. Reinforcement Learning from Human Feedback (RLHF)**

- **Changes the model indirectly:**
  - First, a **reward model** is trained on human preferences.
  - Then, the LLM is fine-tuned using **PPO (Proximal Policy Optimization)** to maximize rewards.
- **Effect:**
  - Adjusts responses to be more helpful, harmless, and aligned.
  - Doesn't drastically alter factual knowledge but changes **behavioral outputs**.
- **Example:** ChatGPT's alignment via RLHF makes it refuse harmful requests.

### **4. Instruction Fine-Tuning**

- **Updates weights** but focuses on following instructions.
- **Effect:**
  - Improves task-specific performance (e.g., summarization, coding).
  - Still retains general capabilities.
- **Example:** Models like **Alpaca** or **FLAN-T5** are instruction-tuned versions of LLaMA/T5.

### **5. In-Context Learning (Not Actually Fine-Tuning)**

- **Does NOT change the model:** Uses prompts with examples to guide responses.
  - **Effect:** No weight updates—just clever prompting.
  - **Example:** Giving ChatGPT 3 examples in the prompt to mimic a style.
- 

### ***Does Fine-Tuning "Overwrite" ChatGPT?***

- **No** (unless you do full fine-tuning on a massive custom dataset).
  - **Most methods (PEFT, RLHF)** only **add small adjustments** without erasing pre-trained knowledge.
  - OpenAI's fine-tuning API for GPT-3.5 Turbo, for example, is designed to **avoid breaking core capabilities**.
-

## ***Key Considerations***

1. **Catastrophic Forgetting:** Full fine-tuning can make the model "forget" general knowledge.
  2. **Overfitting:** Fine-tuning on small datasets can hurt generalization.
  3. **Base Model vs. Fine-Tuned Version:**
    - The original ChatGPT remains unchanged.
    - Your fine-tuned version is a separate instance (OpenAI stores it for you).
- 

## ***When to Fine-Tune?***

### **✓ Good for:**

- Customizing tone/style (e.g., formal, playful).
- Domain-specific tasks (e.g., legal, medical).
- Improving performance on structured tasks.

### **✗ Not needed if:**

- You can solve the task with **prompt engineering**.
  - You lack high-quality, large-scale data.
- 

As a result, fine-tuning **modifies** the pre-trained ChatGPT model, but:

- **Full fine-tuning** changes it significantly (risky without enough data).
- **PEFT/RLHF** makes smaller, safer adjustments.
- The original model remains intact (you deploy the fine-tuned version separately).

During **fine-tuning**, the GPT transformer model **does undergo both forward and backward passes** (just like in pre-training), but the extent depends on the fine-tuning method used. Here's how it works:

---

## ***1. Standard Fine-Tuning (Full Fine-Tuning)***

- **Forward Pass:** The input is processed through all layers to generate predictions.
  - **Backward Pass:** Gradients are computed for **all (or most) parameters**, and weights are updated via **optimization** (e.g., **AdamW**).
  - **Compute Cost:** High (requires full GPU memory for large models like GPT-3).
  - **Example:** Fine-tuning all layers of GPT-3.5 Turbo on a custom dataset.
-

## **2. Parameter-Efficient Fine-Tuning (PEFT) (e.g., LoRA, Adapters)**

- **Forward Pass:** Same as standard fine-tuning (all layers are active).
  - **Backward Pass:** Gradients are **only computed for a small subset of parameters**:
    - **LoRA:** Only the low-rank matrices are updated (original weights frozen).
    - **Adapters:** Only the small added layers are trained.
  - **Compute Cost:** Much lower than full fine-tuning.
  - **Example:** Using LoRA to adapt ChatGPT for a specific task without altering its core weights.
- 

## **3. Reinforcement Learning Fine-Tuning (RLHF)**

- **Forward Pass:** The model generates responses (e.g., multiple completions for a prompt).
  - **Backward Pass:**
    1. A **reward model** (trained on human feedback) scores the responses.
    2. The policy (LLM) is updated via **PPO (Proximal Policy Optimization)**, which involves:
      - Gradient updates to maximize reward (but constrained to avoid drastic changes).
  - **Compute Cost:** Very high (requires multiple iterations of reward scoring and PPO updates).
  - **Example:** ChatGPT's alignment process to reduce harmful outputs.
- 

## **4. Instruction Fine-Tuning**

- Works like standard fine-tuning (forward + backward passes), but the training data is **(instruction, output)** pairs.
  - Example: Training LLaMA on the Alpaca dataset.
- 

## **Key Takeaways**

1. All fine-tuning methods require forward/backward passes, but:
  - **Full fine-tuning:** Updates all weights.
  - **PEFT:** Updates only a tiny fraction of parameters.
  - **RLHF:** Uses reward-guided gradients (not direct supervised loss).
2. No fine-tuning = No backward pass (e.g., in-context learning just does forward passes).
3. Why backward passes are needed:
  - To compute gradients for **loss minimization** (supervised fine-tuning).
  - To maximize **reward signals** (RLHF).

---

## ***Does Fine-Tuning Re-Train the Entire Model?***

- **No** (unless you're doing full fine-tuning).
  - **PEFT/RLHF** only tweak small parts, preserving the bulk of the pre-trained knowledge.
- 

### *Transfer learning vs Fine-tuning*

Let me clarify the distinction between **transfer learning** and **fine-tuning**, especially in the context of models like GPT and BERT:

---

#### ***1. Transfer Learning (Frozen Pre-Trained Model)***

- **Does NOT change the original model:**
  - The pre-trained weights remain **completely frozen** (no backward passes through them).
  - Only a **new task-specific head** (e.g., a classifier layer) is trained on top of the frozen features.
- **Example:**
  - Using BERT as a fixed feature extractor for sentiment analysis by adding a trainable linear layer on top.
  - Using GPT's embeddings for text classification without modifying the transformer.
- **Pros:**
  - Preserves all pre-trained knowledge (no risk of catastrophic forgetting).
  - Very fast and lightweight (only the head is trained).
- **Cons:**
  - Limited adaptability (performance depends heavily on the frozen model's features).

**Key Point:** Transfer learning treats the pre-trained model as a static feature extractor—**zero changes to the original weights**.

---

#### ***2. Fine-Tuning (Updates the Model)***

Fine-tuning **can** change the original model, but the extent depends on the method:

## A. Full Fine-Tuning (Changes Original Model)

- Updates **all (or most) parameters** of the pre-trained model via backpropagation.
- **Original weights are modified** to adapt to the new task.
- **Example:** Fully fine-tuning GPT-3 on a custom dataset.

## B. Parameter-Efficient Fine-Tuning (PEFT) (Minimal Changes)

- **LoRA:** Adds small low-rank matrices (original weights stay frozen; merged later).
- **Adapters:** Inserts tiny trainable layers between frozen transformer blocks.
- **Original model is preserved** until the adapter/LoRA weights are merged.

## C. Partial Fine-Tuning (Selective Changes)

- Only **unfreeze specific layers** (e.g., last 1–2 layers of GPT).
- The untouched layers remain identical to the original model.

## *Key Comparison*

Method	Changes Original Model?	Trainable Components	Use Case
Transfer Learning	✗ No	Only new task head (e.g., classifier)	Simple tasks, low compute
Full Fine-Tuning	✓ Yes	All (or most) weights	High-resource domain adaptation
LoRA/PEFT	⚠ Only after merging	Low-rank matrices/adapters	Efficient task adaptation
Partial Tuning	⚠ Only unfrozen layers	Last N layers + head	Trade-off between adaptability and stability

## *Why This Matters*

### 1. Original Model Safety:

- With transfer learning, the original model is **always safe** (it's read-only).
- With fine-tuning, the original is preserved *only* if you use PEFT or keep a copy.

### 2. Deployment:

- Fine-tuned models are **new variants** (you deploy them alongside the original).
- OpenAI's fine-tuning API, for example, creates a **new endpoint** for your tuned model.

### 3. Catastrophic Forgetting:

- Full fine-tuning risks "overwriting" pre-trained knowledge.

- Transfer learning/PEFT avoids this.
- 

## ***GPT-Specific Considerations***

- **Generative tasks** (e.g., text generation) usually require **some fine-tuning** of the transformer (not just the head).
  - **Parameter-efficient methods (LoRA)** are preferred over "head-only" tuning for GPT.
- 

## ***Final Answer***

- **Transfer learning:** Never changes the original model (only trains a new head).
- **Fine-tuning:**
  - **Full tuning:** Changes the original model.
  - **PEFT/Partial tuning:** Changes only small parts (original remains mostly intact).