Basic Overview of GenAI

Aryan Bhake

Abstract

Generative AI represents a significant shift in the field of artificial intelligence, enabling machines to generate realistic text, images, and code by learning patterns from massive datasets. This document presents a comprehensive theoretical and practical understanding of foundational modules in GenAI, including the Transformer architecture, training paradigms, and parameter-efficient fine-tuning techniques. We explore detailed mathematical formulations, present derivations, implementation code, and discuss recent innovations that have made large-scale deployment possible.

Contents

1 Introduction to Generative AI **4**
1.1 Applications of GenAI 4

2 Transformer Architecture **4**
2.1 Input Embeddings and Positional Encodings 4
2.2 Self-Attention Mechanism 4
2.3 Multi-Head Attention 5
2.4 Position-wise Feedforward Network (FFN) 5
2.5 Layer Normalization and Residual Connections 5
2.6 Transformer Block 5
2.7 Mathematical Summary 5
2.8 Computational Complexity 5
2.9 Types of Transformers 6
2.10 Training Dynamics 6
2.11 Gradient Checkpointing 6

3 Parameter-Efficient Fine-Tuning (PEFT) **6**
3.1 Motivation and Background 6
3.2 Adapter Layers 6
3.3 LoRA: Low-Rank Adaptation 7
3.4 BitFit 7
3.5 Prompt-Tuning and Prefix-Tuning 7
3.6 Comparison of PEFT Methods 7
3.7 Theoretical Insights 7
3.8 Implementation Example: LoRA in PyTorch 7

4 Optimizers in Generative AI **8**
4.1 Stochastic Gradient Descent (SGD) 8
4.2 Adam Optimizer 8
4.3 Learning Rate Scheduling 8
4.4 Weight Decay 8

5 Pretraining Objectives **9**
5.1 Language Modeling Objective . . . 9
5.2 Masked Language Modeling (MLM) . . . 9
5.3 Span Corruption and Permutation . . . 9
5.4 Contrastive Learning . . . 9
5.5 Code Example: MLM Loss . . . 9

6 Training Infrastructure **9**
6.1 Distributed Training . . . 10
6.2 Mixed Precision Training . . . 10
6.3 Gradient Accumulation . . . 10
6.4 Checkpointing . . . 10
6.5 Evaluation and Logging . . . 10
6.6 Hardware . . . 10
6.7 Software Stack . . . 11

7 Parameter-Efficient Fine-Tuning (PEFT) **11**
7.1 LoRA (Low-Rank Adaptation) . . . 11
7.2 Benefits . . . 11
7.3 Other PEFT Methods . . . 11

8 Prefix and Prompt Tuning **11**
8.1 Prompt Tuning . . . 11
8.2 Prefix Tuning . . . 12
8.3 Mathematics . . . 12
8.4 Comparison . . . 12

9 Text Generation Algorithms **12**
9.1 Greedy Decoding . . . 12
9.2 Beam Search . . . 12
9.3 Top-k Sampling . . . 12
9.4 Top-p (Nucleus) Sampling . . . 12
9.5 Temperature Scaling . . . 13

10 Serving LLMs Using FastAPI **13**
10.1 Basic Concepts . . . 13
10.2 Installation . . . 13
10.3 Serving a Text Generator . . . 13
10.4 Run the API . . . 13
10.5 Scalability . . . 14
10.6 Inference Acceleration . . . 14

11 Reinforcement Learning with Human Feedback (RLHF) **14**
11.1 Motivation . . . 14
11.2 Overview of the RLHF Pipeline . . . 14
11.3 Supervised Fine-Tuning (SFT) . . . 14
11.4 Reward Model Training . . . 15
11.5 Reinforcement Learning via PPO . . . 15
11.6 PPO Objective . . . 15
11.7 KL Penalty for Language Models . . . 15
11.8 Benefits and Challenges . . . 15

11.9 Modern Variants 16
11.10 Code Sketch: Reward Model Training 16
11.11 Conclusion 16

1 Introduction to Generative AI

Generative AI (GenAI) involves machine learning techniques that learn patterns from data to generate new, synthetic outputs. It is powered primarily by deep learning and is most commonly exemplified by transformer-based large language models (LLMs).

1.1 Applications of GenAI

- Text generation (e.g., GPT, ChatGPT)
- Image generation (e.g., DALL~E)
- Music synthesis
- Code generation (e.g., Copilot)
- Text summarization, translation, Q&A

2 Transformer Architecture

Transformers are deep learning models that rely on self-attention mechanisms rather than recurrent or convolutional layers. Introduced by Vaswani et al. in 2017, they form the basis of most state-of-the-art generative models.

2.1 Input Embeddings and Positional Encodings

Each token in the input sequence is converted into a dense vector (embedding). Since Transformers lack recurrence, positional information is injected using positional encodings:

$$\text{PE} * (pos, 2i) = \sin\left(\frac{pos}{10000^{2i/d}}\right) \tag{1}$$

$$\text{PE} * (pos, 2i+1) = \cos\left(\frac{pos}{10000^{2i/d}}\right) \tag{2}$$

These are added to the token embeddings to produce position-aware representations.

2.2 Self-Attention Mechanism

The core of the Transformer is the self-attention mechanism which allows each token to attend to every other token in the sequence. Given an input sequence $X \in \mathbb{R}^{n \times d}$ (where n is the number of tokens and d is the embedding dimension), we compute:

$$Q = XW^Q, \tag{3}$$

$$K = XW^K, \tag{4}$$

$$V = XW^V, \tag{5}$$

where $W^Q, W^K, W^V \in \mathbb{R}^{d \times d_k}$ are learned projection matrices.

The attention scores are computed as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{6}$$

This scales dot products by $\sqrt{d_k}$ to counteract large values destabilizing softmax.

2.3 Multi-Head Attention

Rather than computing a single attention output, the input is split into h subspaces for parallel processing:

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \tag{7}$$

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O \tag{8}$$

This allows the model to attend to different parts of the sequence simultaneously.

2.4 Position-wise Feedforward Network (FFN)

Each attention output is passed through an FFN independently:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \tag{9}$$

Here, $W_1 \in \mathbb{R}^{d \times d_{ff}}$, $W_2 \in \mathbb{R}^{d_{ff} \times d}$, typically with $d_{ff} > d$.

2.5 Layer Normalization and Residual Connections

Each sub-layer (attention or FFN) is wrapped with residual connections and layer normalization:

$$\text{LayerNorm}(x + \text{Sublayer}(x)) \tag{10}$$

Residual connections help gradient flow; layer norm stabilizes training.

2.6 Transformer Block

A full Transformer encoder block is composed of:

1. Multi-head self-attention
2. Add
   Norm
3. Feedforward Network
4. Add
   Norm

The decoder variant adds encoder-decoder attention and masking.

2.7 Mathematical Summary

Let x be the token embedding + positional encoding. Then,

$$x_1 = \text{LayerNorm}(x + \text{MultiHead}(x, x, x)) \tag{11}$$

$$x_2 = \text{LayerNorm}(x_1 + \text{FFN}(x_1)) \tag{12}$$

This forms one layer. Stacking N such layers yields a deep Transformer.

2.8 Computational Complexity

Self-attention scales quadratically with sequence length n: $O(n^2 d)$. Many recent improvements (e.g., FlashAttention, Longformer) aim to reduce this.

2.9 Types of Transformers

- Encoder-only (e.g., BERT): good for classification.
- Decoder-only (e.g., GPT): suited for generation tasks.
- Encoder-decoder (e.g., T5, BART): good for translation, summarization.
- Efficient Transformers: Linformer, Performer, Reformer (reduce complexity).

2.10 Training Dynamics

Transformers are typically trained using the cross-entropy loss:

$$\mathcal{L} = -\sum_{t=1}^{T} \log P(x_t \mid x_t) \tag{13}$$

Gradients are computed via backpropagation and optimized using variants of stochastic gradient descent.

2.11 Gradient Checkpointing

To reduce memory usage, intermediate activations are recomputed during backward pass:

- Saves GPU memory
- Trade-off with extra compute

3 Parameter-Efficient Fine-Tuning (PEFT)

Large language models (LLMs) contain billions of parameters, making full fine-tuning resource-intensive and often unnecessary for downstream tasks. Parameter-Efficient Fine-Tuning (PEFT) techniques allow practitioners to adapt LLMs using a small number of additional parameters, significantly reducing computational and memory requirements while maintaining performance.

3.1 Motivation and Background

Given a pre-trained model $f(x;\theta)$, where $\theta \in \mathbb{R}^N$ are the model weights, traditional fine-tuning optimizes all parameters:

$\theta^* = \arg\min_\theta \mathbb{E}_{(x,y)\sim\mathcal{D}} \left[\mathcal{L}(f(x;\theta), y)\right]$

In contrast, PEFT freezes the base parameters θ and introduces trainable parameters $\phi \ll \theta$, learning them via:

$\phi^* = \arg\min_\phi \mathbb{E}_{(x,y)\sim\mathcal{D}} \left[\mathcal{L}(f(x;\theta,\phi), y)\right]$

Here, $f(x;\theta,\phi)$ represents the modified forward pass incorporating ϕ.

3.2 Adapter Layers

Adapters insert small bottleneck networks within Transformer blocks. A typical adapter is defined as:

$\text{Adapter}(h) = h + W_{\text{up}}\sigma(W_{\text{down}}h)$

where:

- $h \in \mathbb{R}^d$ is the hidden representation,
- $W_{\text{down}} \in \mathbb{R}^{r\times d}$, $W_{\text{up}} \in \mathbb{R}^{d\times r}$, with $r \ll d$,

- σ is a non-linear activation, e.g., ReLU or GELU.

Only W_{up} and W_{down} are trained during PEFT.

3.3 LoRA: Low-Rank Adaptation

LoRA hypothesizes that model adaptation lies in a low-rank subspace. Instead of updating $W \in \mathbb{R}^{d\times k}$, LoRA introduces low-rank matrices $A \in \mathbb{R}^{d\times r}$, $B \in \mathbb{R}^{r\times k}$ and modifies the forward pass as:

$Wx \rightarrow (W + \alpha AB)x$

where α is a scaling factor. Only A and B are optimized. The rank r controls the parameter-efficiency trade-off.

3.4 BitFit

BitFit fine-tunes only bias terms b in each linear layer:

$y = Wx + b$ (only b is updated)

This technique is extremely efficient but may underperform on complex tasks.

3.5 Prompt-Tuning and Prefix-Tuning

Prompt-tuning appends learnable embeddings $P \in \mathbb{R}^{l\times d}$ to the input sequence:

$[x_1, \ldots, x_n] \rightarrow [P_1, \ldots, P_l, x_1, \ldots, x_n]$

Prefix-tuning prepends learned keys and values $(K_{\text{prefix}}, V_{\text{prefix}})$ to the attention mechanism in every Transformer layer:

$\text{Attention}(Q, [K_{\text{prefix}}, K], [V_{\text{prefix}}, V])$

3.6 Comparison of PEFT Methods

Let $P_{\text{trainable}}$ denote the number of trainable parameters, and P_{total} the full model size.

Method	**Trainable Parameters**	**Inference Overhead**
Full Fine-Tuning	$P_{\text{trainable}} = P_{\text{total}}$	High
Adapter	$\sim 1\% - 5\%$	Medium
LoRA	$\sim 0.1\% - 1\%$	Low
BitFit	$\ll 1\%$	Minimal
Prompt/Prefix-Tuning	$\sim 0.1\% - 1\%$	Medium

3.7 Theoretical Insights

Let $\mathcal{H} * \theta$ denote the function class spanned by the original model and $\mathcal{H} * \phi \subset \mathcal{H} * \theta$ the subspace induced by PEFT. If the downstream task lies near $\mathcal{H} * \phi$, fine-tuning only a subset is sufficient for high performance, minimizing the generalization gap.

3.8 Implementation Example: LoRA in PyTorch

```
class LoRALinear(nn.Module):
def **init**(self, in_features, out_features, r=4, alpha=1.0):
super().**init**()
self.W = nn.Linear(in_features, out_features, bias=False)
```

```
self.A = nn.Linear(in_features, r, bias=False)
self.B = nn.Linear(r, out_features, bias=False)
self.alpha = alpha

'''
def forward(self, x):
    return self.W(x) + self.alpha * self.B(self.A(x))
'''
```

4 Optimizers in Generative AI

Optimizers play a crucial role in updating the weights of the Transformer model during training to minimize the loss function. Commonly used optimizers include SGD, Adam, and their variants.

4.1 Stochastic Gradient Descent (SGD)

SGD updates weights using the gradient of the loss function:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta \mathcal{L}(\theta_t)$$

Here, η is the learning rate and $\nabla_\theta \mathcal{L}$ is the gradient of the loss with respect to the model parameters θ.

4.2 Adam Optimizer

Adam (Adaptive Moment Estimation) maintains a moving average of both gradients and their squared values:

$$\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1 - \beta_1) g_t \\
v_t &= \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \\
\hat{m}_t &= \frac{m_t}{1 - \beta_1^t} \\
\hat{v}_t &= \frac{v_t}{1 - \beta_2^t} \\
\theta_t &= \theta_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}
\end{aligned}$$

where g_t is the gradient at time t, and ϵ is a small constant for numerical stability.

4.3 Learning Rate Scheduling

Often, a scheduler is used to adaptively change the learning rate:

- Warmup + linear decay
- Cosine annealing
- Step decay

4.4 Weight Decay

Weight decay is used to regularize the model:

$$\theta_{t+1} = \theta_t - \eta \left(\nabla_\theta \mathcal{L} + \lambda \theta_t \right)$$

5 Pretraining Objectives

The objective of pretraining is to teach the model general linguistic and semantic knowledge before fine-tuning on downstream tasks.

5.1 Language Modeling Objective

For autoregressive models like GPT:

$$\mathcal{L} = -\sum_{t=1}^{T} \log P(x_t \mid x_{<t})$$

5.2 Masked Language Modeling (MLM)

Used in BERT-like models:

$$\mathcal{L}_{MLM} = -\sum_{i \in \text{masked}} \log P(x_i \mid x_{\setminus i})$$

Where x_i is masked and the model predicts its value given the rest of the input.

5.3 Span Corruption and Permutation

Used in T5 and XLNet respectively:

- T5 corrupts spans and asks the model to fill them.
- XLNet uses permutation-based language modeling.

5.4 Contrastive Learning

Used in representation learning where the model pulls semantically similar representations closer and dissimilar ones apart.

5.5 Code Example: MLM Loss

```
import torch
import torch.nn.functional as F

def masked_language_loss(logits, labels, mask):
    logits = logits[mask]
    labels = labels[mask]
    return F.cross_entropy(logits, labels)
```

6 Training Infrastructure

Training LLMs at scale requires massive computational infrastructure and software optimizations.

6.1 Distributed Training

Due to GPU memory constraints, training is distributed across multiple GPUs:

- Data Parallelism: Split data across GPUs
- Model Parallelism: Split model across GPUs
- Pipeline Parallelism: Split layers as pipeline stages

6.2 Mixed Precision Training

Uses FP16 or BF16 for faster computation and reduced memory:

- Needs loss scaling to maintain numerical stability

6.3 Gradient Accumulation

Allows effective large batch sizes:

```
accum_steps = 4
optimizer.zero_grad()
for i in range(accum_steps):
    loss = model(input[i]) / accum_steps
    loss.backward()
optimizer.step()
```

6.4 Checkpointing

Checkpoints save model weights and optimizer states periodically for resuming training or evaluation.

6.5 Evaluation and Logging

Tools like TensorBoard, Weights & Biases help track:

- Loss
- Accuracy / Perplexity
- Gradients
- Learning rates

6.6 Hardware

- GPUs: A100, V100, RTX series
- TPUs (for Google Cloud)
- Multi-node clusters

6.7 Software Stack

- Frameworks: PyTorch, TensorFlow, JAX
- Libraries: DeepSpeed, FairScale, HuggingFace Transformers
- Accelerators: NVIDIA Apex, Triton

7 Parameter-Efficient Fine-Tuning (PEFT)

Large-scale models are expensive to fine-tune due to billions of parameters. PEFT techniques reduce the number of trainable parameters while maintaining performance.

7.1 LoRA (Low-Rank Adaptation)

Instead of updating the full weight matrix $W \in \mathbb{R}^{d \times d}$, LoRA learns low-rank updates:

$$W' = W + \Delta W = W + AB$$
$$A \in \mathbb{R}^{d \times r}, \quad B \in \mathbb{R}^{r \times d}, \quad r \ll d$$

Only A and B are learned; W is frozen.

7.2 Benefits

- Reduces training cost
- Enables multiple tasks or domains with lightweight adapters
- Easy storage and deployment

7.3 Other PEFT Methods

- Adapter Layers: Insert small bottleneck MLPs between layers
- BitFit: Tune only bias terms
- IA3: Input-activation based scaling

8 Prefix and Prompt Tuning

Prompt engineering manipulates the input. Prompt tuning and prefix tuning provide trainable methods to optimize prompts.

8.1 Prompt Tuning

- Learn a small set of continuous embeddings prepended to input tokens
- Model weights remain frozen

8.2 Prefix Tuning

- Learn prefix tokens inserted into key and value projections at each Transformer layer
- Changes the behavior of attention

8.3 Mathematics

Assume original attention: $\text{softmax}(QK^\top)V$. In prefix tuning, we augment K and V with learned prefixes:

$$K' = [K_{\text{prefix}}; K], \quad V' = [V_{\text{prefix}}; V]$$

8.4 Comparison

- Prompt tuning affects only the input layer
- Prefix tuning influences all Transformer layers

9 Text Generation Algorithms

9.1 Greedy Decoding

At each timestep, choose the token with the highest probability:

$$x_t = \arg\max_w P(w \mid x_{<t})$$

Can lead to repetitive and uncreative outputs.

9.2 Beam Search

Maintains k best sequences at each step. At timestep t, expand each of the k sequences with all possible next tokens and keep top k:

$$\text{Score}(x_{1:t}) = \log P(x_1) + \cdots + \log P(x_t)$$

9.3 Top-k Sampling

Randomly sample from the top k probable tokens:

$$P_k(w) = \frac{P(w)}{\sum_{i=1}^{k} P(w_i)} \text{ for } w \in \text{Top-k}$$

9.4 Top-p (Nucleus) Sampling

Sample from the smallest set of tokens with cumulative probability $\geq p$:

$$\sum_{i=1}^{n} P(w_i) \geq p$$

Adaptive to output distribution.

9.5 Temperature Scaling

Controls randomness by modifying logits:

$$P(w) \propto \exp\left(\frac{\text{logit}_w}{T}\right)$$

Low T makes distribution sharp (greedy), high T flattens it (diverse).

10 Serving LLMs Using FastAPI

FastAPI is a modern, high-performance web framework for building APIs with Python.

10.1 Basic Concepts

- Async I/O powered by `asyncio`
- Automatic documentation via OpenAPI
- Dependency injection for modularity

10.2 Installation

```
pip install fastapi uvicorn
```

10.3 Serving a Text Generator

```
from fastapi import FastAPI, Request
from transformers import AutoModelForCausalLM, AutoTokenizer
import torch

app = FastAPI()

model_name = "gpt2"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(model_name)

@app.post("/generate")
async def generate_text(req: Request):
    body = await req.json()
    prompt = body["prompt"]
    input_ids = tokenizer(prompt, return_tensors="pt").input_ids
    output = model.generate(input_ids, max_length=100)
    return {"response": tokenizer.decode(output[0])}
```

10.4 Run the API

```
uvicorn app:app --reload
```

10.5 Scalability

- Use Gunicorn + Uvicorn workers for multi-process serving
- Add caching with Redis
- Monitor with Prometheus + Grafana

10.6 Inference Acceleration

- ONNX export
- DeepSpeed inference engine
- vLLM, TensorRT, or TGI for optimized backends

11 Reinforcement Learning with Human Feedback (RLHF)

Reinforcement Learning with Human Feedback (RLHF) is a fine-tuning methodology used to align large language models (LLMs) with human preferences. It bridges the gap between raw model pretraining (which relies on maximum likelihood objectives) and desired human-like behavior, such as helpfulness, harmlessness, and honesty.

11.1 Motivation

Large models trained purely via next-token prediction may produce toxic, biased, or unhelpful responses. RLHF incorporates human judgment to shape model behavior post-pretraining.

11.2 Overview of the RLHF Pipeline

RLHF typically proceeds in three main stages:

1. **Supervised Fine-Tuning (SFT)**: Fine-tune a pretrained LLM on high-quality human-annotated prompts and completions.
2. **Reward Model (RM) Training**: Train a reward model that ranks multiple responses based on human preferences.
3. **Policy Optimization via RL**: Fine-tune the model using reinforcement learning (typically PPO) guided by the reward model.

11.3 Supervised Fine-Tuning (SFT)

Given a dataset $\mathcal{D}_{\text{SFT}} = \{(x_i, y_i)\}$ of prompts x_i and ideal completions y_i, the objective is:

$$\mathcal{L}_{\text{SFT}}(\theta) = -\sum_i \log P_\theta(y_i \mid x_i)$$

This stage ensures the model understands the task and provides coherent outputs.

11.4 Reward Model Training

Given a set of k candidate responses $\{y_1, y_2, \ldots, y_k\}$ for a prompt x, humans rank them by preference. These are used to train a reward function $r_\phi(x, y)$ parameterized by ϕ, typically via pairwise loss:

$$\mathcal{L}_{\text{RM}} = -\log \sigma(r_\phi(x, y^+) - r_\phi(x, y^-))$$

where (y^+, y^-) is a preferred and dispreferred pair, and σ is the sigmoid function.

11.5 Reinforcement Learning via PPO

The reward model $r_\phi(x, y)$ acts as a proxy for human feedback. We define the reward:

$$R(x, y) = r_\phi(x, y)$$

We then fine-tune the language model (policy π_θ) using PPO (Proximal Policy Optimization) to maximize expected reward:

$$\max_\theta \mathbb{E}_{y \sim \pi_\theta(\cdot|x)} [R(x, y)]$$

11.6 PPO Objective

Let $\pi_{\theta_{\text{old}}}$ be the policy before update, and π_θ be the current one. PPO maximizes:

$$\mathcal{L}^{\text{PPO}}(\theta) = \mathbb{E}_t \left[\min \left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon)\hat{A}_t \right) \right]$$

where:

$$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}, \quad \hat{A}_t = R_t - V(s_t)$$

$V(s_t)$ is the value function used for advantage estimation, and ϵ is a small clip factor to ensure stability.

11.7 KL Penalty for Language Models

To prevent the policy from drifting too far from the initial (helpful but safe) model π_{ref}, a KL penalty is added:

$$\tilde{R}(x, y) = r_\phi(x, y) - \beta \cdot \text{KL}[\pi_\theta(\cdot \mid x) \| \pi_{\text{ref}}(\cdot \mid x)]$$

where β controls the regularization strength.

11.8 Benefits and Challenges

- **Pros**:
  - Enables alignment with human values
  - Produces more helpful and safe outputs
- **Cons**:
  - Human labeling is expensive and noisy
  - Difficult to generalize across cultures or value systems

11.9 Modern Variants

- DPO (Direct Preference Optimization): An alternative to PPO with simpler implementation and faster training.
- RLAIF (RL with AI Feedback): Uses LLMs themselves to generate feedback instead of humans.
- Constitutional AI (Anthropic): Feedback is guided by a set of predefined principles (e.g., helpfulness, non-toxicity).

11.10 Code Sketch: Reward Model Training

```
# Pseudo-code for pairwise reward model training
import torch
import torch.nn.functional as F

def loss_fn(reward_model, x, y_pos, y_neg):
    r_pos = reward_model(x, y_pos)
    r_neg = reward_model(x, y_neg)
    return -F.logsigmoid(r_pos - r_neg)
```

11.11 Conclusion

RLHF is a cornerstone of aligning generative models like ChatGPT with human expectations. By integrating human preferences via reward modeling and reinforcement learning, RLHF pushes the model beyond statistical language modeling to responsible, safe AI generation.