Language-Model Training — Notebook-style
Pretraining, Fine-tuning, and Preference Alignment (RLHF)

Compiled for learners

November 19, 2025

Abstract

This notebook-style manual explains the three main stages of modern large language model training: (1) pretraining via self-supervised maximum likelihood (next-token prediction), (2) supervised fine-tuning / instruction tuning (SFT), and (3) preference alignment via reinforcement learning from human feedback (RLHF). The document is written for beginners but is complete: it declares notation, gives clear formulas, a tiny numeric worked example, practical advice, and compact pseudocode for each stage. Where helpful, diagrams and boxed reminders clarify key steps.

Contents

1. Problem setup and notation (recap)

- Vocabulary size: V.
- A training sequence (token indices): $y_{1:T} = (y_1, \dots, y_T)$ where $y_t \in \{1, \dots, V\}$.
- Model parameters: θ.
- At time t the model produces a logits vector (pre-softmax scores) $u_t \in \mathbb{R}^V$. We write components $u_{t,i}$ for token i.
- The model probability for token i at time t is the softmax over logits:

$$p_{t,i} \equiv p_\theta(y_t = i \mid y_{<t}) = \mathrm{softmax}(u_t)_i = \frac{\exp(u_{t,i})}{\sum_{j=1}^{V} \exp(u_{t,j})}.$$

2. Sequence negative log-likelihood (per-sequence MLE objective)

For a single sequence $y_{1:T}$, the MLE (negative log-likelihood) loss is the sum of tokenwise negative log probabilities:

$$\mathcal{L}_{\text{seq}}(\theta; y_{1:T}) = -\sum_{t=1}^{T} \log p_\theta(y_t \mid y_{<t}) = -\sum_{t=1}^{T} \log p_{t,\, y_t}.$$

This is the same as token-level cross-entropy between the one-hot target distribution and the model distribution at each time step.

Notation / variables

- V = vocabulary size (number of distinct tokens).
- B = batch size (examples processed together).
- L = sequence length (tokens per example).
- $x = (x_1, \dots, x_T)$ = input/context tokens.
- $y = (y_1, \dots, y_T)$ = target token sequence (usually next-token targets).
- θ = model parameters (weights).
- $u_t \in \mathbb{R}^V$ = logits vector at time t.
- $p_\theta(y_t \mid y_{<t}) = \mathrm{softmax}(u_t)$ = model predicted probability for token y_t.
- $1[\cdot]$ = indicator function.

---

1 1) Pretraining (self-supervised MLE / next-token prediction)

1.1 Goal

Teach the model general-purpose language knowledge by predicting the next token given preceding context across massive unlabeled corpora.

1.2 Data

Large, diverse text corpora: books, web pages, code repositories, news articles, dialogs, etc. Sequences are tokenized into vocabulary indices.

1.3 Objective (per sequence)

For a single target sequence $y_{1:T}$ the negative log-likelihood (MLE) loss is:

$$\mathcal{L}_{\text{MLE}}(\theta; y_{1:T}) = -\sum_{t=1}^{T} \log p_\theta(y_t \mid y_{<t}).$$

This is the same as the per-token cross-entropy.

It represents how surprised the model is when trying to predict the correct next token at each step.If the model assigns high probability to the correct token → low lossIf the model assigns low probability → high lossSo the loss is literally measuring:How well does the model predict each next token in the training sequence?We want to minimize this surprise → meaning the model learns to assign high probability to correct next tokens.

1.4 Batch-averaged loss

Over a batch of B sequences (with per-example lengths L_b ignoring padding):

$$\mathcal{L}(\theta) = -\frac{1}{B} \sum_{b=1}^{B} \sum_{t=1}^{L_b} \log p_\theta(y_{b,t} \mid y_{b,<t}).$$

1.5 Training mechanics — step by step

1. For each sequence in the batch, feed tokens $y_{<t}$ (teacher forcing) to the model to obtain hidden state h_t.
2. Compute logits $u_t = h_t W_{\text{vocab}} + b$ (linear map to vocabulary dimension).
3. Stabilize logits: $u_t \leftarrow u_t - \max_j u_{t,j}$ (optional numerics trick).
4. Compute probabilities via softmax: $p_{t,i} = \exp(u_{t,i}) / \sum_j \exp(u_{t,j})$.
5. Compute token loss $\ell_t = -\log p_{t,y_t}$ and sum across timesteps and batch.
6. Backpropagate gradients $\nabla_\theta \mathcal{L}$ and update parameters with Adam/AdamW.

1.6 Why this works (intuitively)

Predicting next tokens forces the model to capture syntactic patterns, semantics, and statistical regularities in language — because predicting accurately requires modeling dependencies across tokens.

1.7 Tiny numeric example (digit-by-digit)

Single example, $B = 1$, $L = 3$, $V = 3$. At time $t = 3$ model logits: $u = [2.0, 0.5, -1.0]$.

1. Shift by max (2.0): $u' = [0, -1.5, -3.0]$.
2. Exponentiate: $e^{u'} = [1.0000, 0.2231, 0.0498]$.
3. Sum $S = 1.2729$.
4. Probabilities $p \approx [0.7859, 0.1754, 0.0391]$.
5. If true token index is 2, loss $\ell = -\log(0.1754) \approx 1.741$.

1.8 Practical training notes

- Use teacher forcing (feed true tokens) during training for stable gradients.
- Large batch sizes and long training schedules improve performance.
- Learning-rate schedules (warmup + decay) help optimization.
- Regularization: weight decay, dropout, and careful data curation.

2 2) Fine-tuning / Instruction Tuning (supervised tuning)

2.1 Goal

Adapt a pretrained model to a specific task or to follow instructions by training on labeled input→output pairs. This often improves format, tone, and task accuracy.

2.2 Objective

Same per-token cross-entropy, but conditional on task input $x^{(b)}$:

$$\mathcal{L}_{\text{SFT}}(\theta) = -\frac{1}{B}\sum_{b=1}^{B}\sum_{t=1}^{L_b}\log p_\theta(y_t^{(b)} \mid x^{(b)}, y_{<t}^{(b)}).$$

2.3 Why it helps

Gives direct supervision for the desired output style, structure, and task-specific behavior. Instruction datasets teach the model to respond in human-friendly formats.

2.4 Techniques and practicals

- **Adapters / LoRA:** low-rank updates reduce storage compute for fine-tuning.
- **Mixing data:** include some pretraining-like data or low learning rates to avoid catastrophic forgetting.
- **Evaluation:** use held-out instruction prompts, measure BLEU/ROUGE/EM where relevant, and human eval for instruction-following.

2.5 Tiny example (conceptual)

Prompt: "Summarize: <article>" → Target: 3-bullet summary. Train to minimize cross-entropy on the target summary tokens.

3 3) Preference alignment (RLHF — Reward modeling + RL)

3.1 Goal

Align model outputs with human judgments (helpfulness, safety, preference), beyond what label pairs provide.

3.2 High-level pipeline (step-by-step)

1. **Collect comparisons:** For many prompts, collect multiple candidate responses (from model(s) or humans). Human annotators rank or choose the preferred response(s).
2. **Train a reward model** $r_\phi(x, y)$ that scores a (prompt,response) pair to match human preferences. The model is trained with pairwise loss: if human prefers A over B,

$$-\log \frac{\exp r_\phi(x, A)}{\exp r_\phi(x, A) + \exp r_\phi(x, B)}.$$

3. **Optimize the policy** π_θ (the language model) to maximize expected reward under r_ϕ, usually with PPO-like updates while constraining divergence from a reference policy π_{ref}.
4. **Iterate:** collect more comparisons from newly generated outputs, retrain reward model, and repeat RL policy updates.

3.3 Formal (informal) objective for policy update

Maximize expected reward with a KL penalty to keep policy close to reference:

$$\max_\theta \mathbb{E}_{y\sim\pi_\theta(\cdot|x)}\big[r_\phi(x, y)\big] - \beta \ \mathrm{KL}\left(\pi_\theta(\cdot|x)\|\pi_{\text{ref}}(\cdot|x)\right).$$

Here β controls tradeoff between reward-seeking and staying close to the reference policy.

3.4 Reward-model training (pairwise) — step-by-step

1. Collect labeled pairs (A preferred over B) for many prompts.
2. For each pair, compute scalar scores $s_A = r_\phi(x, A)$ and $s_B = r_\phi(x, B)$.
3. Minimize pairwise cross-entropy loss: $-\log \frac{e^{s_A}}{e^{s_A}+e^{s_B}}$.
4. Validate on held-out human comparisons.

3.5 PPO / policy updates — specifics

- Use trajectories of token-level log-probabilities under the current policy and compute rewards via r_ϕ (optionally with a per-token shaping term).

- Compute advantages (reward minus baseline) and update policy with clipped surrogate objective.
- Add KL penalty or constraint relative to π_{ref} to prevent runaway optimization.
- Use many PPO epochs per batch but keep learning rates small; normalize rewards and advantages.

3.6 Practical worries and mitigations

- Reward hacking: policy finds loopholes in reward model; mitigate by improving reward data, adding adversarial prompts, and penalizing shortcuts.
- Over-optimization reduces factuality: preserve capabilities via KL-to-ref and mixed objectives.
- Annotation noise: use multiple annotators, calibration, and quality checks.

4 Why three stages? (intuition and picture)

Intuition: Pretraining builds general intelligence (knowledge + language). Fine-tuning instructs the model how to format and perform tasks. RLHF sculpts behavior to align with messy human preferences that are hard to encode directly.

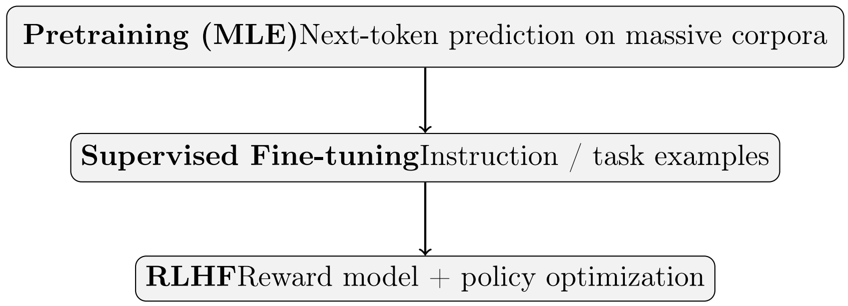

Figure 1: Three-stage training pipeline: each stage refines capabilities or behavior.

5 Quick checklist / cheat sheet

- Pretrain = MLE next-token on huge corpora → general language.
- Fine-tune = supervised training on task/instruction examples → specialized behavior.
- RLHF = train reward from human prefs → optimize policy with RL and KL constraint → aligned behavior.
- Loss (per-token): $-\log p_\theta(y_t|context)$; batch loss = average over tokens and examples.
- Typical optimizers: AdamW; regularization: weight decay, dropout, early stopping.

6 Appendix: compact PyTorch-style pseudocode

6.1 Pretraining (simplified)

```
# Pseudocode: Pretraining loop (simplified)
for epoch in range(epochs):
    for batch in dataloader:
        inputs, targets = batch  # teacher-forcing
        logits = model(inputs)   # shape [B, L, V]
        loss = cross_entropy_loss(logits.view(-1,V), targets.view(-1))
        loss.backward()
        optimizer.step(); optimizer.zero_grad()
```

6.2 Supervised Fine-tuning

```
# Pseudocode: SFT loop
for epoch in range(epochs):
    for batch in sft_dataloader:
        prompt, target = batch
        logits = model(prompt, target_input)
        loss = cross_entropy_loss(logits, target)
        loss.backward(); optimizer.step(); optimizer.zero_grad()
```

6.3 RLHF (sketch)

```
# Collect responses
for prompt in prompts:
    samples = [sample_from(model, prompt) for _ in range(K)]
    # humans rank samples; build (A>B) pairs
# Train reward model r_phi to predict human prefs
# Use PPO to update policy pi_theta using reward r_phi and KL penalty to
   pi_ref
for ppo_iter in range(N):
    trajectories = generate_trajectories(pi_theta)
    rewards = [r_phi(x,y) for (x,y) in trajectories]
    advantages = compute_advantages(rewards)
    ppo_update(pi_theta, trajectories, advantages, kl_penalty=beta)
```

End of notebook-style entry. If you want a one-page visual summary, or a downloadable PDF compiled from this LaTeX, say the word and I will prepare it.