A Practitioner's Guide to Implementing and Scaling Reinforcement Learning for Large Models

Introduction: From Theory to Scalable Implementation

The objective of this guide is to provide a comprehensive, project-based curriculum for the technically proficient machine learning practitioner aiming to master Reinforcement Learning (RL) for training and fine-tuning large-scale models. The approach is predicated on the principle of learning by building, moving from the foundational implementation of core algorithms to the architecture of sophisticated, distributed RL systems. This journey is structured to cultivate a deep, transferable understanding of both the algorithmic mechanics and the systems engineering principles required to apply RL effectively in domains ranging from natural language processing to robotics.

The methodology is inspired by deep-dive, first-principles engineering efforts, where a complex problem is broken down into manageable, progressively more challenging projects.[1] Each project is designed to produce a functional, reusable code artifact and a more profound grasp of a core RL or systems concept. This ensures that theoretical knowledge is immediately grounded in practical application.

The curriculum is built upon three core tenets:

1. **Understand by Building:** True mastery comes from implementation. This guide will walk through the process of writing policies, loss functions, and even distributed communication patterns from scratch, ensuring a fundamental understanding that transcends any single library or framework.
2. **Progressive Complexity:** The projects begin with simple algorithms and environments, allowing for a clear focus on core concepts. Complexity is then layered on incrementally, scaling up to larger models, distributed hardware, and advanced techniques like Reinforcement Learning from Human Feedback (RLHF).
3. **System-Aware RL:** At scale, RL is as much a systems design challenge as it is an algorithmic one. This guide treats the two as inextricably linked, addressing the practical

infrastructure and engineering considerations necessary to train large models efficiently on modern hardware.

By the end of this guide, the practitioner will have not only implemented a range of RL algorithms but also architected a complete, multi-GPU training pipeline, and gained the perspective needed to evaluate, customize, and deploy state-of-the-art RL systems for complex, real-world tasks.

Part I: Foundations of Policy Gradient Methods from Scratch

This initial part of the guide is dedicated to building a solid, intuitive understanding of the core mechanics of policy gradient algorithms. By implementing these methods from the ground up on tasks that do not require complex physics simulators, the algorithmic challenges can be isolated from the environmental ones. This foundational knowledge is essential before tackling the complexities of large-scale models and distributed systems.

Section 1: Implementing Your First Policy Gradient Algorithm: REINFORCE

This section demystifies the core concept of policy gradients: directly optimizing a policy network to maximize expected rewards. The focus will be on implementing the simplest Monte Carlo policy gradient algorithm, REINFORCE. This algorithm serves as an excellent entry point because it directly illustrates the fundamental principle of increasing the probability of actions that lead to high rewards.

Project 1: REINFORCE for Controlled Text Generation

The goal of this first project is to build a complete, from-scratch REINFORCE agent in PyTorch. This agent will fine-tune a small language model, such as a character-level Recurrent Neural Network (RNN) or a pre-trained model like GPT-2, to generate text that possesses a specific, verifiable property. Examples of such properties include a desired sentiment or the inclusion of certain keywords. This task is chosen to abstract away the complexities of a physics simulator, allowing for a pure focus on the RL algorithm's implementation.

Theoretical Grounding

The REINFORCE algorithm is a foundational policy gradient method that optimizes a policy directly, without the use of a value function.[2] It operates by updating the policy's parameters based on the gradient of the expected total reward. The core idea is to collect complete trajectories, or "episodes," of experience by interacting with an environment and then using the observed returns to update the policy. This is a Monte Carlo approach, as it relies on the full return from an entire episode to inform the update.[2] The update rule essentially reinforces actions that led to higher returns by increasing their probabilities.

The training process for REINFORCE follows a simple loop [2]:

1. Use the current policy to generate a full episode (a trajectory of states, actions, and rewards).
2. For each timestep in the episode, calculate the total return (the sum of discounted rewards from that timestep to the end of the episode).
3. Update the policy parameters by performing gradient ascent on the objective function, which is weighted by the calculated returns.

Implementation Steps

The implementation will be constructed in PyTorch, following a logical progression from environment definition to the final training loop.[2]

1. Environment Setup: A Text-Based World
   Instead of using a standard library like gymnasium, a custom "text environment" will be defined. This approach is a powerful pedagogical tool, as it demonstrates that the RL paradigm is a generic framework for sequential decision-making, not one tied to specific domains like games or robotics. In this environment:
   - The **state** is the sequence of tokens generated thus far.
   - An **action** is the selection of the next token to append to the sequence.
   - An episode begins with a starting prompt and ends when an end-of-sequence token is generated or a maximum sequence length is reached.4
     This self-contained setup isolates the core RL components—policy, sampling, reward, and update—making the learned concepts more easily transferable to other domains.
2. Policy Network: The Decision-Maker
   A simple policy network will be implemented in PyTorch. This could be an LSTM or a Transformer decoder block that takes the current sequence of tokens as input and outputs a probability distribution over the entire vocabulary for the next token.6 The model's forward pass will produce these probabilities, from which an action (the next token) will be sampled.
3. The REINFORCE Loop: Learning from Experience

The main training loop will be coded to execute the REINFORCE algorithm.

- **Rollout:** For each episode, the agent will generate a complete text sequence. Starting with a prompt, it will iteratively sample the next token from the policy network's output distribution, feeding the new sequence back into the model for the next step. The log-probabilities of each chosen action (token) are stored for the subsequent update step.[2]
- **Reward Calculation:** A simple, verifiable reward function will be defined and implemented. For instance, the reward could be a single scalar value awarded at the end of the episode: +1 if the generated text contains a positive sentiment word (e.g., "excellent") and -1 otherwise.[4] This immediate feedback, though sparse, is the signal the agent will use to learn. This step immediately highlights a critical aspect of applied RL: reward engineering. A simple binary reward can lead to "reward hacking," where the model might learn to just output the target keyword and nothing else to maximize its reward. This observation serves as a crucial lesson and sets the stage for discussing more nuanced reward shaping in later projects.
- **Policy Update:** The REINFORCE loss function will be implemented. The objective is to maximize the expected reward, which is achieved by minimizing the negative of the expected reward. The loss for a single trajectory is calculated as: $L = -\sum_{t=0}^{T} G_t \log \pi(a_t|s_t)$, where G_t is the discounted return from timestep t and $\pi(a_t|s_t)$ is the probability of the action taken. For simplicity in this initial project, a single undiscounted return for the whole episode can be used, resulting in a loss of loss = - (total_reward) * (sum of log probabilities).[2] PyTorch's automatic differentiation is then used to compute the gradients and update the policy network's weights. This update directly teaches the model which sequences of token choices led to positive outcomes.

Code Reference and Walkthrough

The following provides a conceptual, minimal PyTorch implementation for the REINFORCE text generation task. It adapts the logic found in classic CartPole examples to the text domain.[2] While many available online resources for RL and text generation are complex or rely on extensive libraries [9], this implementation prioritizes minimalism for clarity.

Python

```python
import torch
import torch.nn as nn
import torch.optim as optim
from torch.distributions import Categorical
```

```
# 1. Policy Network (e.g., a simple RNN)
class Policy(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim):
        super(Policy, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.rnn = nn.LSTM(embedding_dim, hidden_dim, batch_first=True)
        self.fc = nn.Linear(hidden_dim, vocab_size)

    def forward(self, state_tokens):
        embedded = self.embedding(state_tokens)
        output, _ = self.rnn(embedded)
        # Use the output of the last token to predict the next
        logits = self.fc(output[:, -1, :])
        return torch.softmax(logits, dim=-1)

# Hyperparameters
vocab_size = 100 # Example vocabulary size
embedding_dim = 32
hidden_dim = 64
lr = 0.01
max_seq_len = 20

policy = Policy(vocab_size, embedding_dim, hidden_dim)
optimizer = optim.Adam(policy.parameters(), lr=lr)

# 2. The REINFORCE Loop
for episode in range(1000):
    # --- Rollout Phase ---
    state = torch.tensor([]) # Start with a <start> token
    log_probs =
    generated_sequence =

    for t in range(max_seq_len):
        probs = policy(state)
        dist = Categorical(probs)
        action = dist.sample() # Sample the next token (action)

        log_prob = dist.log_prob(action)
        log_probs.append(log_prob)

        generated_sequence.append(action.item())
        state = torch.cat([state, action.unsqueeze(0)], dim=1)
        if action.item() == 1: # Assume 1 is <end> token
            break

    # --- Reward Calculation Phase ---
    # Simple reward: +1 if a specific token (e.g., ID 42) is in the sequence
    reward = 1.0 if 42 in generated_sequence else -1.0

    # --- Policy Update Phase ---
    # We use a single return for the whole episode for simplicity
    returns = [reward] * len(log_probs)

    loss =
    for log_prob, R in zip(log_probs, returns):
        loss.append(-log_prob * R)

    optimizer.zero_grad()
    loss = torch.cat(loss).sum()
    loss.backward()
    optimizer.step()

    if episode % 100 == 0:
        print(f"Episode {episode}, Total Reward: {reward}, Loss: {loss.item()}")
```

This implementation, while simplified, captures the essence of the REINFORCE algorithm and its application to a generative task, providing a solid foundation for the more advanced methods to come.

Section 2: Advancing to Actor-Critic Methods: A PPO Deep Dive

This section addresses a primary weakness of the REINFORCE algorithm: high variance in the policy gradient estimate. This high variance stems from the reliance on noisy Monte Carlo returns from entire episodes, which can slow down and destabilize learning. The solution is to introduce a value function, which leads to the family of Actor-Critic methods. The focus here will be on a from-scratch implementation of Proximal Policy Optimization (PPO), the workhorse algorithm for a vast range of RL applications, including modern LLM fine-tuning.

Project 2: Implementing PPO for Classic Control

The goal of this project is to build a complete PPO agent from scratch in PyTorch. This agent will be capable of solving a classic control environment from the gymnasium library, such as CartPole-v1 or the more challenging LunarLander-v2.[20] Completing this project will result in a robust, reusable PPO implementation that will serve as the core of the RLHF pipeline in later sections.

Theoretical Grounding

Actor-Critic methods separate the policy and value function into two components:

- The **Actor** is the policy, which controls how the agent behaves (i.e., maps states to actions).
- The **Critic** is the value function, which evaluates the actions taken by the Actor by estimating the value of being in a particular state.

The Critic's role is to provide a lower-variance estimate of the return than the full Monte Carlo return used in REINFORCE. Instead of judging an action based on the total reward of the episode, it is judged based on how much better or worse it was than the Critic's expectation. This "better-or-worse" signal is called the **advantage**.

Proximal Policy Optimization (PPO) is an advanced Actor-Critic algorithm that improves training stability by constraining the size of policy updates at each step.[22] Its key innovation is the **clipped surrogate objective function**, which discourages the policy from changing too drastically from one iteration to the next. This creates a "trust region" where the policy can be updated safely, preventing the catastrophic performance collapses that can occur with overly aggressive updates. PPO also commonly uses **Generalized Advantage Estimation (GAE)**, a technique that provides a sophisticated trade-off between biased but low-variance value estimates and unbiased but high-variance Monte Carlo returns to calculate the advantage.

Implementation Steps

The implementation will be structured to clearly separate the different logical components of the PPO algorithm, drawing inspiration from high-quality educational repositories.[22]

1. Environment Setup: Interfacing with Gymnasium
   The agent will interface with a standard gymnasium environment.20 This involves using the standard API:
   - env.reset(): To start a new episode and get the initial observation.
   - env.step(action): To execute an action in the environment, which returns the next observation, reward, termination signal, truncation signal, and additional info.
2. Actor-Critic Network: A Shared Brain
   A single neural network will be implemented in PyTorch with a shared body and two separate heads:

    - **Actor Head (Policy):** Outputs a probability distribution over the possible actions (e.g., logits for a Categorical distribution in discrete action spaces).
    - Critic Head (Value Function): Outputs a single scalar value, which is the estimate of the state's value, $V(s)$.
      This shared-parameter architecture is common and efficient, as both the policy and value function often benefit from similar low-level feature representations of the state.
3. The PPO Loop: Collect, Estimate, Optimize
   The main training loop for PPO is more complex than that of REINFORCE and involves distinct phases.
    - **Data Collection (Rollout):** The agent interacts with the environment for a fixed number of steps (e.g., 2048 timesteps), using the current policy to select actions. For each step, it stores a tuple of (state, action, reward, next_state, done, log_prob). This collection of experiences forms the rollout buffer.
    - **Advantage Calculation (GAE):** Once the rollout is complete, Generalized Advantage Estimation (GAE) will be implemented. This is a crucial step that calculates the advantage for each timestep in the buffer. GAE uses the collected rewards and the Critic's value estimates for V(s) and V(s_next) to compute a more stable advantage signal. The GAE formula is a recursive one that balances bias and variance using a parameter λ.
    - **Optimization Phase:** After calculating advantages, the agent enters an optimization phase where it performs multiple gradient updates on the collected data. The rollout buffer is repeatedly sampled in mini-batches for a fixed number of epochs (e.g., 10 epochs). For each mini-batch:
        - **Actor Loss (PPO-Clip):** The core of PPO, the clipped surrogate objective, will be implemented. This involves calculating the ratio of the new policy's probability of an action to the old policy's probability. This ratio is then clipped to stay within a small interval around 1.0 (e.g., $[0.8, 1.2]$). The loss is the minimum of the clipped and unclipped objectives, which discourages large, potentially destabilizing policy updates.
        - **Critic Loss:** A simple mean squared error (MSE) loss is used to train the Critic. The loss is calculated between the Critic's value predictions and the actual returns computed from the rollout buffer (often called the value targets).
        - **Entropy Bonus (Optional):** An entropy term can be added to the Actor's loss to encourage exploration by penalizing policies that become too deterministic too quickly.

Code Reference and Walkthrough

The following conceptual code provides a skeleton for a PPO implementation, highlighting the key differences from REINFORCE. A full implementation would be more extensive, involving helper functions for GAE calculation and managing the rollout buffer. This structure is heavily influenced by clean, educational implementations like PPO-for-Beginners.[23]

Python

```
import torch
import torch.nn as nn
import gymnasium as gym

# 1. Actor-Critic Network
class ActorCritic(nn.Module):
    def __init__(self, state_dim, action_dim):
        super(ActorCritic, self).__init__()
        self.shared_layers = nn.Sequential(
            nn.Linear(state_dim, 64),
            nn.Tanh(),
            nn.Linear(64, 64),
            nn.Tanh()
        )
        # Actor head
        self.actor_head = nn.Linear(64, action_dim)
        # Critic head
        self.critic_head = nn.Linear(64, 1)

    def forward(self, state):
        features = self.shared_layers(state)
        action_logits = self.actor_head(features)
        state_value = self.critic_head(features)
        return action_logits, state_value

# Hyperparameters
clip_epsilon = 0.2
ppo_epochs = 10
mini_batch_size = 64
gamma = 0.99
gae_lambda = 0.95

#... (Assume env, model, optimizer are initialized)...
#... (Assume a RolloutBuffer class exists to store experiences)...

# 2. The PPO Loop
# --- Data Collection Phase ---
# (Loop for `num_steps_per_rollout` to fill the RolloutBuffer with experiences)

# --- Advantage Calculation Phase ---
# (Compute advantages and returns using GAE and store them in the buffer)
# This step is crucial. It uses the value estimates from the critic to create a
# baseline, allowing the actor to learn not just if an action was good in an absolute
# sense (positive return), but if it was *better than expected* (positive advantage).
# This is a fundamental leap from REINFORCE, which uses noisy raw returns and leads
# to much more stable and efficient learning.

# --- Optimization Phase ---
for _ in range(ppo_epochs):
    for mini_batch in buffer.get_mini_batches(mini_batch_size):
        old_states, old_actions, old_log_probs, returns, advantages = mini_batch

        # Get new log_probs and state_values from the current policy
        action_logits, state_values = model(old_states)
        dist = torch.distributions.Categorical(logits=action_logits)
        new_log_probs = dist.log_prob(old_actions)

        # --- Actor Loss (PPO-Clip) ---
        ratio = torch.exp(new_log_probs - old_log_probs)
        surr1 = ratio * advantages
        surr2 = torch.clamp(ratio, 1 - clip_epsilon, 1 + clip_epsilon) * advantages
        actor_loss = -torch.min(surr1, surr2).mean()

        # --- Critic Loss (MSE) ---
        critic_loss = nn.MSELoss()(state_values, returns)

        # Total loss and backpropagation
        loss = actor_loss + 0.5 * critic_loss
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

By implementing both REINFORCE and PPO, the role of the Critic as a variance reduction tool becomes viscerally clear. The transition from using raw, noisy returns to using stable advantage estimates is a fundamental step-up in algorithmic sophistication and performance. Furthermore, PPO's clipped objective introduces the concept of trust region optimization. While PPO is an on-policy algorithm (learning from data generated by the current policy), its ability to perform multiple updates on the same batch of data via importance sampling (corrected by the clipping mechanism) makes it significantly more sample-efficient than simpler on-policy methods like A2C, which perform only one update per data collection phase. This efficiency is a key reason for its widespread adoption.

Part II: Fine-Tuning Large Language Models with RL

With a solid, from-scratch implementation of PPO in hand, the focus now shifts to applying this powerful algorithm to the domain of large language models (LLMs). This part of the guide bridges the gap between classic RL in controlled environments and the modern application of RL for model alignment and behavior fine-tuning.

Section 3: The Reinforcement Learning from Human Feedback (RLHF) Pipeline

This section deconstructs the full, three-stage Reinforcement Learning from Human Feedback (RLHF) process. The culmination of this section will be a project to fine-tune a 7-billion-parameter LLM, demonstrating a practical and highly relevant application of the RL principles established in Part I.

Project 3: Building a Simplified RLHF Pipeline

The goal of this project is to fine-tune a 7B parameter LLM, such as Llama 3 8B Instruct or Qwen2 7B, using a simplified but complete RLHF pipeline. This will involve using a public preference dataset to train a reward model and then using that model to provide the learning signal for our PPO agent.

The Three Stages of RLHF

The RLHF process is a multi-stage, multi-model endeavor designed to align a powerful base LLM with human preferences and values.[17]

1. Stage 1: Supervised Fine-Tuning (SFT)
   The process begins not with RL, but with standard supervised fine-tuning. A pre-trained base LLM is fine-tuned on a high-quality dataset of curated prompt-response pairs. This initial step is crucial for adapting the model to the desired output format (e.g., a helpful assistant) and teaching it the basic style of interaction.17 For this project, to focus on the

RL aspects, a library like Hugging Face's TRL and its SFTTrainer can be leveraged to efficiently perform this stage.25 The output of this stage is the SFT model, which will serve as the initial policy for the RL stage.

2. Stage 2: Reward Model (RM) Training
   The next step is to create a proxy for human preferences by training a reward model. This model learns to predict which of two responses to a given prompt a human would prefer.
   - **Dataset:** A public preference dataset, such as Anthropic's HH-RLHF dataset, will be used.[27] This dataset contains a large number of prompts, each associated with two responses that have been labeled as "chosen" and "rejected" by human annotators.
   - **Implementation:** A reward model will be implemented. This is typically a language model (often initialized from the SFT model) with its final layer replaced by a linear layer that outputs a single scalar value (the reward). The model is trained on the preference dataset. For each prompt, it computes a reward score for both the "chosen" and "rejected" responses. The training objective is a binary classification-style loss that maximizes the margin between the scores of the chosen and rejected responses.[17] The model is trained to assign a higher score to the response that humans preferred.
3. Stage 3: RL Fine-Tuning with PPO
   This is the core of the project, where the PPO algorithm is used to fine-tune the SFT model to maximize the rewards given by the trained RM.
   - **Integration:** The PPO implementation from Project 2 will be adapted for this task. The "environment" is now the LLM itself, which generates a text response given a prompt. The "reward" for a generated response is no longer a simple, hard-coded value but is now provided by the sophisticated, learned reward model from Stage 2.
   - **KL-Divergence Penalty:** A critical component in modern RLHF is the inclusion of a Kullback-Leibler (KL) divergence penalty. In each training step, a penalty is added to the reward signal based on how much the current policy's output distribution deviates from that of the original SFT policy. This term acts as a constraint, preventing the model from "reward hacking"—finding esoteric sequences of tokens that exploit the reward model to get a high score but result in incoherent or nonsensical text. It keeps the policy anchored to the SFT model's well-behaved, fluent language generation capabilities.[17] This penalty will be added to the PPO objective function.

This project reveals that production-grade RL is often a multi-model system. The RLHF pipeline requires the orchestration of at least three distinct models during the final training phase: the **policy model** (the one being trained), a **reference model** (the original SFT model, used for the KL penalty), and the **reward model**. If using an Actor-Critic algorithm like PPO, a fourth model, the **value model** (the Critic), is also involved. Managing these models—loading them, moving data between them, and distributing them across available hardware—is a significant step up in complexity from the single-agent RL of the previous projects. This complexity is the primary motivation for the development of robust, distributed training frameworks, which will be the focus of Part III.

Comparison with TRL and Discussion of DPO

After implementing the core RLHF logic, the from-scratch implementation will be compared to the PPOTrainer provided by Hugging Face's TRL (Transformer Reinforcement Learning) library.[25] TRL provides a high-level, optimized, and battle-tested implementation of the PPO-based RLHF loop. Analyzing its source code, particularly the ppo.py example script, will highlight the engineering abstractions, performance optimizations (like batching and device management), and additional features that a mature library provides over a minimal implementation.[25]

Finally, the discussion will turn to **Direct Preference Optimization (DPO)**, a modern and increasingly popular alternative to the three-stage RLHF process.[34] DPO is an elegant algorithm that cleverly bypasses the need for an explicitly trained reward model. It directly optimizes the language model on the preference pairs (chosen, rejected) using a simple binary cross-entropy loss function. The theoretical insight behind DPO is that the reward function can be expressed in terms of the optimal policy and a reference policy. By substituting this relationship into the standard RLHF objective, DPO derives a loss function that depends only on the policy being trained and the reference policy. This makes the training process simpler, more stable, and less computationally intensive than the traditional PPO-based approach, which is why it has seen rapid adoption for aligning models like Llama 3.[34] The introduction of the KL penalty in the PPO pipeline highlights the core tension in LLM alignment: optimizing for a specific reward signal (e.g., helpfulness, harmlessness) while simultaneously preserving the model's fundamental fluency and general capabilities. DPO's mathematical formulation elegantly captures this trade-off within a single, unified loss function, which is a key reason for its improved stability and popularity.

Section 4: Exploring Advanced Reward Mechanisms

This section moves beyond the paradigm of learning from subjective human preferences to explore the use of automated and verifiable reward sources. This is a cutting-edge area of RL research that promises to make the alignment process more scalable, objective, and robust.

Conceptual Project: Designing an RLVR System

The goal of this conceptual project is to architect a system for **Reinforcement Learning with**

Verifiable Rewards (RLVR). This system will be designed for a task where correctness can be objectively determined, such as solving math word problems or generating simple code that must pass unit tests.

Theoretical Grounding

RLVR represents a significant shift in the source of the reward signal. Instead of relying on a learned proxy for subjective human feedback, RLVR uses an objective, automated verification process to determine the reward.[35] The core principle is that the reward is granted only when an outcome is demonstrably correct. For example, a generated mathematical proof is only rewarded if each step is logically sound and the final answer is correct; a piece of code is only rewarded if it passes a predefined set of unit tests. This approach provides a high-quality, scalable, and unambiguous reward signal that is less prone to the biases and inconsistencies of human annotation and less susceptible to the "reward hacking" that can plague learned reward models.[35]

System Design

The architecture of an RLVR system involves three key components:

1. **The Verifier:** This is the heart of the RLVR system. It is an automated component responsible for evaluating the correctness of the model's generated output.
   - For a mathematical reasoning task, the verifier could be a Python script that uses a symbolic math library (like SymPy) or simply the Python interpreter to execute the model's chain-of-thought reasoning and check if the final numerical answer matches a known solution.
   - For a code generation task, the verifier would be a sandboxed execution environment that compiles and runs the generated code against a suite of unit tests. The number of passing tests would determine the outcome.
2. **The Reward Function:** The design of the reward function is critical for effective learning.
   - **Binary Reward:** The simplest approach is a binary reward: +1 for a fully correct solution, and 0 otherwise. While this signal is clean, it can be very sparse, making it difficult for the model to learn, especially on complex, multi-step problems.
   - **Soft or Graded Rewards:** To provide a denser learning signal, more nuanced reward schemes can be employed. Recent research has explored "soft" rewards, where a separate generative model provides a probabilistic score of correctness, or graded rewards that give partial credit for partially correct solutions (e.g., passing some but not all unit tests).[36] This helps guide the model more effectively during exploration.
3. **Integration with PPO:** The verifier-based reward function plugs directly into the existing PPO training loop developed in Project 3. It replaces the learned reward model (RM). In each step of the RL loop, the policy model generates a response, this response is passed to the verifier, and the verifier's output is used as the scalar reward to compute the advantages and update the policy.

This exploration of reward sources illustrates a critical design decision in any applied RL problem. Rewards can come from a spectrum of sources: sparse signals from the environment itself (as in classic control), dense learned proxies for complex objectives (like the RM in RLHF), or objective, automated verifiers (as in RLVR). The choice among these involves trade-offs in scalability, cost, signal density, and susceptibility to reward hacking. RLVR, in particular, represents a key step towards building more autonomous agents. By learning to generate outputs that satisfy an automated, objective criterion, the model is learning to achieve concrete, measurable goals without the need for constant human supervision. This concept connects directly to the domain of robotics and Vision-Language-Action (VLA) models, where success is often defined by verifiable physical outcomes, such as successfully grasping an object or navigating to a target location.

Part III: Architecting and Scaling Your RL Training Framework

This part of the guide forms the systems engineering core of the curriculum. Having implemented and understood the key RL algorithms, the focus now shifts to building the infrastructure required to run them efficiently on a multi-GPU, multi-node cluster. The user's available hardware of 6-8 H100 GPUs provides an ideal platform for exploring these advanced, large-scale training paradigms.

Section 5: Building a Distributed RL Training Framework

This section provides a hands-on guide to parallelizing and distributing the PPO-based RLHF pipeline from Project 3. It involves iteratively refactoring the single-GPU implementation to support multi-GPU training, comparing and contrasting three major distributed training paradigms: DistributedDataParallel (DDP), FullyShardedDataParallel (FSDP), and DeepSpeed.

Project 4: Scaling PPO to a Multi-GPU Cluster

The goal of this project is to transform the PPO implementation into a high-performance, distributed training system. This process will not only accelerate training but also enable the fine-tuning of models that are too large to fit into the memory of a single GPU.

Step 1: Data Parallelism with DistributedDataParallel (DDP)

- **Concept:** DDP is the most fundamental form of distributed training in PyTorch. The model is replicated on each GPU, and each process (or "rank") works on a different slice of the input data batch. After the backward pass, the gradients computed on each GPU are synchronized and averaged across all GPUs using an efficient all-reduce communication collective. This ensures that all model replicas remain identical at the end of each training step.[37]
- **Implementation:** The PPO training script will be refactored to incorporate DDP. This involves several key steps [38]:
  1. Initialize the distributed process group using torch.distributed.init_process_group(). This sets up the communication backend (typically nccl for NVIDIA GPUs).
  2. Wrap the models (policy, value, and reward models) in the torch.nn.parallel.DistributedDataParallel container.
  3. Use torch.utils.data.distributed.DistributedSampler for the data loader to ensure that each process receives a unique, non-overlapping subset of the training data.
  4. Launch the training script using torchrun or a similar launcher, which manages the spawning of processes for each GPU.
     This DDP implementation will serve as the performance baseline for distributed training.

Step 2: Sharded Data Parallelism with FullyShardedDataParallel (FSDP)

- **Concept:** When models become too large to fit on a single GPU (even with a batch size of one), DDP is no longer viable. FSDP is PyTorch's native solution for this problem. Instead of replicating the entire model on each GPU, FSDP shards the model's parameters, gradients, and optimizer states across the available GPUs. This dramatically reduces the memory footprint per device.[39] During computation, each FSDP-wrapped module gathers the full parameters it needs just for the forward or backward pass using an all-gather collective, and then discards them immediately afterward. Gradients are synchronized using a reduce-scatter collective, which is more communication-efficient than DDP's all-reduce for this sharded setup.[42]
- **Implementation:** The DDP wrappers will be replaced with torch.distributed.fsdp.FullyShardedDataParallel. This is not a simple drop-in replacement and requires more careful configuration [43]:
  1. **Auto-Wrap Policy:** An auto_wrap_policy must be defined. This is a function that tells FSDP how to group the model's layers into sharded units. A common strategy is to wrap each transformer block individually.
  2. **State Dict Management:** Saving and loading checkpoints is more complex with FSDP. The model state is sharded across all GPUs, so special functions are needed to either save the sharded state or gather the full state onto a single GPU for saving.[45]
  3. **Optimizer Initialization:** The optimizer must be initialized *after* the model has been

wrapped with FSDP, as FSDP replaces the model's parameters with its own sharded parameter handles.

Step 3: Advanced Optimization with DeepSpeed

- **Concept:** DeepSpeed is a comprehensive library from Microsoft that provides a suite of optimizations for large-scale training. Its flagship feature is the Zero Redundancy Optimizer (ZeRO), which offers multiple stages of sharding that are conceptually similar to FSDP.[47] DeepSpeed goes further by offering advanced features like offloading optimizer states and even model parameters to CPU RAM or NVMe storage (ZeRO-Infinity), enabling the training of truly massive models on hardware with limited GPU memory.[48]
- **Implementation:** Integrating DeepSpeed is typically done through a configuration file and a lightweight wrapper around the model. Libraries like Hugging Face Accelerate simplify this integration significantly.[51] The process involves:
  1. Creating a deepspeed_config.json file that specifies the desired optimizations, such as zero_optimization stage 3.
  2. Using the DeepSpeed initialization function, deepspeed.initialize(), which wraps the model, optimizer, and data loader.
  3. Launching the training script with the deepspeed command-line launcher.

Comparative Analysis

After implementing all three backends, the RLHF pipeline will be run with each, and performance will be systematically measured. This analysis is crucial for developing an engineer's intuition about which tool to use in which scenario. The results will be summarized in a table, providing a practical decision-making guide. A senior engineer needs to understand not just *how* to use a tool, but *when* and *why*. For an RLHF pipeline with multiple models of varying sizes (e.g., a 70B policy model and a 1B reward model), the choice of strategy is a nuanced architectural decision. This comparison distills complex documentation [54] into a concrete framework for making those decisions.

Feature	**DistributedDataParallel (DDP)**	**Fully Sharded Data Parallel (FSDP)**	**DeepSpeed (ZeRO-3 + Offload)**
Core Mechanism	Model Replication	Sharding (Parameters, Gradients, Optimizer)	Advanced Sharding + CPU/NVMe Offload

Memory Efficiency	Baseline (Lowest)	High	Highest
Communication Pattern	all-reduce gradients	all-gather params, reduce-scatter grads	Dynamic, optimized collectives
Implementation	Manual setup with torch.distributed [38]	PyTorch native, requires wrap policy [43]	Library-driven via config file [51]
Best for RL Policy/Value Models...	Small models (<10B) that fit on one GPU.	Large models (>10B) that must be sharded.	Extreme-scale models or memory-constraine d hardware.
Considerations for Multi-Model RL	Simple to apply to all models, but inefficient for large policy models.	Ideal for the large policy model; may be overkill for smaller reward/value models.	Offers fine-grained control but adds dependency. Great for heterogeneous memory needs.

This project solidifies the understanding that modern RL at scale is fundamentally a distributed systems problem. The challenge extends beyond implementing the PPO loss to orchestrating the flow of data (rollouts, experiences) and model states across a cluster of GPUs. A key observation from building a synchronous distributed pipeline is the "RL training stall": the powerful training GPUs sit idle while waiting for the policy model to generate rollouts. This inefficiency is a major bottleneck and directly motivates the asynchronous architectures used in advanced frameworks like SkyRL, providing a natural transition to the final part of this guide.

Part IV: Advanced Applications and the Broader Ecosystem

This final part connects the custom-built framework to real-world applications in robotics and provides a critical evaluation of when and how to adopt a full-stack, pre-built RL library. Having constructed the core components of an RL framework from first principles, the practitioner is now equipped to understand, extend, and make informed decisions about using more advanced tools.

Section 6: Bridging RL and Robotics Simulation

This section provides the necessary primers to apply the developed RL framework to the domain of robotics and Vision-Language-Action (VLA) models, an area of keen interest. The focus is on interfacing with industry-standard physics simulators and adapting the agent to handle visual inputs and continuous action spaces.

Project 5: PPO for a Robotic Manipulation Task

The goal of this project is to connect the distributed PPO framework to a simulated robotics environment and train a simple vision-based manipulation policy. This will demonstrate the generality of the RL framework and highlight the specific challenges of applying RL to physical systems.

Environment Primers

A prerequisite for this project is familiarity with a high-fidelity robotics simulator. Two excellent options are NVIDIA Isaac Sim and MuJoCo.

- **NVIDIA Isaac Sim:** This is a powerful, photorealistic, and physically accurate robotics simulator built on the Omniverse platform. A primer on Isaac Sim would cover setting up a new scene, loading a robot's Universal Scene Description (USD) file, defining the physics properties of the environment, and using its Python API to access observations (such as camera images and joint states) and send action commands to the robot's actuators.[60]
- **MuJoCo (Multi-Joint dynamics with Contact):** This is a fast and accurate physics engine widely used in robotics research. A primer on MuJoCo would involve loading a model from an MJCF (MuJoCo XML) file, stepping the simulation, and using the Python bindings to read sensor data and apply controls.[63]

Implementation Steps

1. Environment Wrapper: The Universal Adapter
   The most critical step is to create a gymnasium-compatible wrapper around the chosen simulator's Python API. This wrapper will expose the standard reset() and step() methods, translating the simulator's specific data structures into the observation and

action spaces that the PPO agent expects. This act of creating a standardized interface is a powerful demonstration of software engineering principles in RL, as it makes the learning algorithm completely independent of the underlying environment.

2. Vision-Based Policy: Learning from Pixels
   The Actor-Critic network from Project 2 will be modified to handle visual inputs. This involves adding a Convolutional Neural Network (CNN) backbone (e.g., a ResNet) to the front of the network. The CNN will process the raw image observations from the simulated camera and extract a compact feature vector, which is then fed into the existing fully-connected layers of the policy and value heads.
3. Continuous Actions: Controlling the Robot
   Most robotics tasks involve continuous actions, such as setting joint torques or target end-effector positions. The policy head must be adapted to handle this. Instead of outputting logits for a categorical distribution, it will output the parameters of a continuous probability distribution, typically a Gaussian. For each action dimension, it will output a mean (μ) and a standard deviation (σ), and the final action is sampled from this Gaussian distribution.
4. Training and Robotics-Specific Challenges
   Finally, the distributed PPO trainer from Project 4 is connected to the new, wrapped robotics environment, and a training run is launched. This phase will bring to light challenges specific to robotics, such as:
   - **Sample Inefficiency:** RL in robotics is notoriously sample-inefficient. It can take millions or even billions of simulation steps to learn a complex task.
   - **Reward Shaping:** Designing a good reward function is even more critical and difficult in robotics. A sparse reward (e.g., +1 only upon successful task completion) can make learning nearly impossible. Dense reward shaping, where the agent is given intermediate rewards for making progress towards the goal (e.g., moving its hand closer to an object), is often necessary but can introduce unintended biases.

This project powerfully illustrates the value of standardized APIs. The gymnasium API acts as the glue that allows a highly optimized, distributed RL algorithm to be plugged into a completely new and complex domain like robotics with minimal changes to the core learning logic. This fulfills the objective of building a general-purpose framework. Furthermore, while this project remains in simulation, it naturally opens the door to a discussion of the critical **sim-to-real gap**—the challenge of transferring policies trained in simulation to real-world hardware. The fidelity of the physics simulation, the realism of sensor data, and the robustness of the learned policy to real-world noise are all paramount concerns for practical robotics applications.

Section 7: Evaluating Full-Stack RL Libraries: A Case Study of SkyRL

Having built the core components of a distributed RL training framework from scratch, the practitioner is now in an excellent position to critically evaluate a full-stack, pre-built library. This section uses SkyRL as a case study to understand the architectural choices made in production-grade RL systems and to develop a framework for deciding when to build from scratch versus when to adopt an existing solution.

SkyRL Architecture Deep Dive

SkyRL is a modular, full-stack RL library from the Berkeley Sky Computing Lab, designed specifically for training large language models on complex, long-horizon, and agentic tasks.[66] An analysis of its architecture reveals how it addresses the challenges encountered in the previous projects.

- **Deconstruction:** SkyRL's architecture is composed of several key modules: skyrl-train, skyrl-gym, and skyrl-agent. Its logical components—Trainer, Generator, and InferenceEngine—can be mapped directly to the concepts built in this guide.
  - The most significant architectural choice is the separation of the **Generator** and the **Trainer**. The Generator is responsible for rollouts—interacting with the environment to generate trajectories. The Trainer is responsible for the optimization step—consuming these trajectories to update the model weights. This is a direct and robust solution to the "training stall" problem identified in Part III. It allows for an **asynchronous pipeline**, where the Generator can run on one set of hardware (perhaps cheaper CPUs or older GPUs) to constantly produce data, while the Trainer runs on the powerful H100s, ensuring they are always utilized for their primary purpose: gradient computation.
  - skyrl-gym provides a library of pre-built, tool-use environments, abstracting away the environment creation and wrapping step that was a key part of Project 5.[66]
  - skyrl-train encapsulates the complex distributed training logic built in Project 4, with native support for different backends (like FSDP and Megatron) and hardware configurations.[69]
- **When to Build vs. When to Adopt:** The journey through this guide provides the context for this critical engineering decision.
  - **Building from scratch** (as has been done here) is invaluable for deep learning, debugging, and full control, enabling the implementation of novel algorithms or highly custom system architectures.
  - **Adopting a library** like SkyRL is the pragmatic choice for rapid prototyping and for leveraging a battle-tested, highly optimized, and scalable infrastructure. For complex, long-horizon agentic tasks like software engineering (SWE-Bench), SkyRL provides the necessary scaffolding out of the box.[67]

Using and Customizing SkyRL

To demonstrate the power and flexibility of a mature framework, a brief walkthrough of a

SkyRL example will be conducted.

- **Walkthrough:** A quickstart example, such as fine-tuning a model on the GSM8K math dataset using the GRPO algorithm (a PPO variant), will be executed.[71] SkyRL's integration with SkyPilot can be used to demonstrate how such a job can be seamlessly launched on a cloud cluster with a single command, abstracting away the complexities of provisioning and setup.[70]
- **Customization:** A key advantage of a well-designed framework is extensibility. Using a framework does not mean sacrificing flexibility. SkyRL's documentation provides clear tutorials and API extension points for adding new components. For example, a new custom environment can be registered within skyrl-gym, or a novel advantage estimator can be implemented and registered within skyrl-train, demonstrating that the framework is designed to be built upon, not just used as a black box.[72]

The design of a framework like SkyRL is not arbitrary. Its separation of concerns, particularly between generation and training, is an architectural pattern that naturally emerges from the need to build efficient, asynchronous, distributed RL systems. By building the components from first principles first, one can fully appreciate *why* SkyRL is designed the way it is. The user can see the Generator/Trainer split not as an abstract design choice, but as a concrete solution to the GPU idle time problem they personally identified. This provides a powerful, practical understanding of production-grade RL systems design. Finally, SkyRL's focus on long-horizon, real-world tasks and tool use represents the frontier of applied RL for LLMs, connecting this learning journey to the most active and exciting areas of current research and development.[66]

Conclusion

This guide has charted a comprehensive path from the fundamental principles of Reinforcement Learning to the architecture and deployment of large-scale, distributed training systems for fine-tuning modern language and robotic models. By adhering to a project-based, first-principles approach, a practitioner can develop a deep and robust understanding that is not tied to any single library but is grounded in the core mechanics of the algorithms and the systems that run them.

The journey began with the from-scratch implementation of REINFORCE and PPO, solidifying the core concepts of policy gradients, variance reduction through actor-critic methods, and trust region optimization. These algorithms were then applied to the complex, multi-stage RLHF pipeline, revealing the systems-level challenges of orchestrating multiple models and the critical importance of alignment mechanisms like the KL-divergence penalty. The exploration of RLVR further broadened the perspective on reward engineering, highlighting the power of objective, automated verification.

The transition to Part III marked a crucial shift from an algorithmic to a systems engineering mindset. By implementing and comparing DDP, FSDP, and DeepSpeed, the practitioner gains first-hand experience with the trade-offs in memory, communication, and implementation complexity inherent in distributed training. This process culminates in the key architectural realization of the "training stall" in synchronous RL and the necessity of asynchronous designs, as embodied in advanced frameworks like SkyRL.

Finally, the guide connected this custom-built framework to the domain of robotics, demonstrating the power of standardized APIs like gymnasium to bridge disparate fields. The critical analysis of SkyRL provided a capstone, illustrating how production-grade frameworks are not merely collections of tools but are well-designed solutions to recurring architectural patterns discovered through the process of scaling RL systems.

By completing this curriculum, the practitioner is equipped not just to use existing RL tools, but to reason about their design, customize them, and, when necessary, build new components from the ground up. This holistic skill set—spanning algorithm theory, practical implementation, and distributed systems engineering—is precisely what is required to push the boundaries of what is possible with reinforcement learning and large-scale models.

Works cited

1. Training AI to play Super Smash Bros. Melee - Eric Gu, accessed October 29, 2025, https://ericyuegu.com/melee-pt1
2. REINFORCE - A Quick Introduction (with Code) | Dilith Jayakody, accessed October 29, 2025, https://dilithjay.com/blog/reinforce-a-quick-introduction-with-code
3. Reinforce Algorithm Explained: Python Implementation, Use Cases & Intuition - upGrad, accessed October 29, 2025, https://www.upgrad.com/tutorials/ai-ml/machine-learning-tutorial/reinforce-algorithm/
4. Seq2Seq and Reinforcement Learning Text Generation influenced ..., accessed October 29, 2025, https://www.reddit.com/r/learnmachinelearning/comments/7wefqd/seq2seq_and_reinforcement_learning_text/
5. Understanding reinforcement learning for model training from scratch | by Rohit Patel | Data Science Collective | Medium, accessed October 29, 2025, https://medium.com/data-science-collective/understanding-reinforcement-learning-for-model-training-from-scratch-8bffe8d87a07
6. Tutorial: Text Generation with LSTMs and GRUs - Wandb, accessed October 29, 2025, https://wandb.ai/wandb_fc/wb-tutorials/reports/Tutorial-Text-Generation-with-LSTMs-and-GRUs--Vmlldzo0NTIxMjY1
7. Building Your Own Text Generation Model: A Beginner's Guide | by

Abhijithcprakash, accessed October 29, 2025, https://medium.com/@abhijithcprakash/building-your-own-text-generation-model-a-beginners-guide-5261fad1e6bc
8. Building a text generation model from scratch - Wingedsheep: Artificial Intelligence Blog, accessed October 29, 2025, https://wingedsheep.com/building-a-language-model/
9. Minimalist implementation of a GPT2 with Language Model Head with PyTorch Lightning, Transformers and PyTorch-NLP. - GitHub, accessed October 29, 2025, https://github.com/minimalist-nlp/gpt2-text-generation
10. bharathgs/Awesome-pytorch-list: A comprehensive list of pytorch related content on github,such as different models,implementations,helper libraries,tutorials etc., accessed October 29, 2025, https://github.com/bharathgs/Awesome-pytorch-list
11. pytorch/rl: A modular, primitive-first, python-first PyTorch library for Reinforcement Learning. - GitHub, accessed October 29, 2025, https://github.com/pytorch/rl
12. Lightning-AI/pytorch-lightning: Pretrain, finetune ANY AI model of ANY size on 1 or 10,000+ GPUs with zero code changes. - GitHub, accessed October 29, 2025, https://github.com/Lightning-AI/pytorch-lightning
13. A set of examples around pytorch in Vision, Text, Reinforcement Learning, etc. - GitHub, accessed October 29, 2025, https://github.com/pytorch/examples
14. unslothai/unsloth: Fine-tuning & Reinforcement Learning for LLMs. Train OpenAI gpt-oss, DeepSeek-R1, Qwen3, Gemma 3, TTS 2x faster with 70% less VRAM. - GitHub, accessed October 29, 2025, https://github.com/unslothai/unsloth
15. Introducing torchforge – a PyTorch native library for scalable RL post-training and agentic development, accessed October 29, 2025, https://pytorch.org/blog/introducing-torchforge/
16. Reinforcement Learning (DQN) Tutorial - PyTorch documentation, accessed October 29, 2025, https://docs.pytorch.org/tutorials/intermediate/reinforcement_q_learning.html
17. Reinforcement Learning from Human Feedback (RLHF): A Practical Guide with PyTorch Examples | by Sam Ozturk, accessed October 29, 2025, https://themeansquare.medium.com/reinforcement-learning-from-human-feedback-rlhf-a-practical-guide-with-pytorch-examples-139cee11fc76
18. Generating Text with PyTorch Cheatsheet - Codecademy, accessed October 29, 2025, https://www.codecademy.com/learn/generating-text-with-py-torch/modules/generating-text-with-pytorch/cheatsheet
19. Reinforcement Learning (PPO) with TorchRL Tutorial - PyTorch documentation, accessed October 29, 2025, https://docs.pytorch.org/tutorials/intermediate/reinforcement_ppo.html
20. Classic Control - Gymnasium Documentation, accessed October 29, 2025, https://gymnasium.farama.org/environments/classic_control/
21. Farama-Foundation/Gymnasium: An API standard for single-agent reinforcement learning environments, with popular reference environments and related utilities

(formerly Gym) - GitHub, accessed October 29, 2025, https://github.com/Farama-Foundation/Gymnasium
22. ikostrikov/pytorch-a2c-ppo-acktr-gail: PyTorch implementation of Advantage Actor Critic (A2C), Proximal Policy Optimization (PPO), Scalable trust-region method for deep reinforcement learning using Kronecker-factored approximation (ACKTR) and Generative Adversarial Imitation Learning (GAIL). - GitHub, accessed October 29, 2025, https://github.com/ikostrikov/pytorch-a2c-ppo-acktr-gail
23. ericyangyu/PPO-for-Beginners: A simple and well styled ... - GitHub, accessed October 29, 2025, https://github.com/ericyangyu/PPO-for-Beginners
24. RLHF_with_Custom_Datasets.ipynb - Colab, accessed October 29, 2025, https://colab.research.google.com/github/heartexlabs/RLHF/blob/master/tutorials/RLHF_with_Custom_Datasets.ipynb
25. Examples - Hugging Face, accessed October 29, 2025, https://huggingface.co/docs/trl/main/example_overview
26. huggingface/trl: Train transformer language models with reinforcement learning. - GitHub, accessed October 29, 2025, https://github.com/huggingface/trl
27. Hh Rlhf — Unitxt, accessed October 29, 2025, https://www.unitxt.ai/en/main/catalog/catalog.cards.hh_rlhf.html
28. Hh Rlhf — Unitxt, accessed October 29, 2025, https://www.unitxt.ai/en/1.10.0/catalog/catalog.cards.hh_rlhf.html
29. Anthropic/hh-rlhf · Datasets at Hugging Face, accessed October 29, 2025, https://huggingface.co/datasets/Anthropic/hh-rlhf
30. What is RLHF and how to use it to train an LLM — Part 4 | by Jim Wang | Medium, accessed October 29, 2025, https://medium.com/@jimwang3589/what-is-rlhf-and-how-to-use-it-to-train-an-llm-part-4-1146228b74ef
31. TRL - Transformer Reinforcement Learning - Hugging Face, accessed October 29, 2025, https://huggingface.co/docs/trl/index
32. Quickstart - Hugging Face, accessed October 29, 2025, https://huggingface.co/docs/trl/v0.1.1/quickstart
33. Examples - Hugging Face, accessed October 29, 2025, https://huggingface.co/docs/trl/v0.7.2/example_overview
34. Direct Preference Optimization: Your Language Model is Secretly a ..., accessed October 29, 2025, https://arxiv.org/pdf/2305.18290
35. Reinforcement Learning with Verifiable Rewards: Unlocking reliable AI reasoning, accessed October 29, 2025, https://toloka.ai/blog/reinforcement-learning-with-verifiable-rewards-unlocking-reliable-ai-reasoning/
36. Crossing the Reward Bridge: Expanding RL with Verifiable Rewards ..., accessed October 29, 2025, https://arxiv.org/abs/2503.23829
37. Distributed and parallel training... explained - Part 1 (2020) - fast.ai Course Forums, accessed October 29, 2025, https://forums.fast.ai/t/distributed-and-parallel-training-explained/73892
38. HOWTO: PyTorch Distributed Data Parallel (DDP) | Ohio ..., accessed October 29,

2025, https://www.osc.edu/resources/getting_started/howto/howto_pytorch_distributed_data_parallel_ddp
39. Getting Started with Fully Sharded Data Parallel(FSDP) - PyTorch documentation, accessed October 29, 2025, https://docs.pytorch.org/tutorials/intermediate/FSDP1_tutorial.html
40. Fully Sharded Data Parallel (FSDP) - GeeksforGeeks, accessed October 29, 2025, https://www.geeksforgeeks.org/deep-learning/fully-sharded-data-parallel-fsdp/
41. PyTorch Fully Sharded Data Parallel (FSDP) on AMD GPUs with ROCm, accessed October 29, 2025, https://rocm.blogs.amd.com/artificial-intelligence/fsdp-training-pytorch/README.html
42. Fully Sharded Data Parallel (FSDP) Theory of Operations - Habana Documentation, accessed October 29, 2025, https://docs.habana.ai/en/latest/PyTorch/PyTorch_FSDP/Theory_of_operations.html
43. Run Torch FSDP - Prime Intellect Docs, accessed October 29, 2025, https://docs.primeintellect.ai/tutorials-multi-node-cluster/torch-run-fsdp-multi-node
44. HOWTO: PyTorch Fully Sharded Data Parallel (FSDP) | Ohio Supercomputer Center, accessed October 29, 2025, https://www.osc.edu/resources/getting_started/howto/howto_pytorch_fully_sharded_data_parallel_fsdp
45. FullyShardedDataParallel — PyTorch 2.9 documentation, accessed October 29, 2025, https://docs.pytorch.org/docs/stable/fsdp.html
46. Fully Sharded Data Parallel - Hugging Face, accessed October 29, 2025, https://huggingface.co/docs/accelerate/usage_guides/fsdp
47. Implementing DeepSpeed for Scalable Transformers: Advanced Training with Gradient Checkpointing and Parallelism - MarkTechPost, accessed October 29, 2025, https://www.marktechpost.com/2025/09/06/implementing-deepspeed-for-scalable-transformers-advanced-training-with-gradient-checkpointing-and-parallelism/
48. DeepSpeed — PyTorch Lightning 2.5.5 documentation, accessed October 29, 2025, https://lightning.ai/docs/pytorch/stable/advanced/model_parallel/deepspeed.html
49. DeepSpeed: Latest News, accessed October 29, 2025, https://www.deepspeed.ai/
50. Training Overview and Features - DeepSpeed, accessed October 29, 2025, https://www.deepspeed.ai/training/
51. Getting Started - DeepSpeed, accessed October 29, 2025, https://www.deepspeed.ai/getting-started/
52. DeepSpeed - Hugging Face, accessed October 29, 2025, https://huggingface.co/docs/transformers/deepspeed
53. DeepSpeed - Hugging Face, accessed October 29, 2025,

https://huggingface.co/docs/accelerate/usage_guides/deepspeed
54. DeepSpeed vs FSDP #245 - Lightning-AI/lit-llama - GitHub, accessed October 29, 2025, https://github.com/Lightning-AI/lit-llama/issues/245
55. FSDP vs DeepSpeed - Hugging Face, accessed October 29, 2025, https://huggingface.co/docs/accelerate/concept_guides/fsdp_and_deepspeed
56. DeepSpeed vs FSDP: A Comprehensive Comparison - BytePlus, accessed October 29, 2025, https://www.byteplus.com/en/topic/499106
57. Accelerate vs. DeepSpeed vs. FSDP - Ben Gubler, accessed October 29, 2025, https://www.bengubler.com/posts/2023-08-29-accelerate-deepspeed-fsdp
58. [D] What do you all use for large scale training? Normal pytorch or do you use libraries like HF Accelerate. - Reddit, accessed October 29, 2025, https://www.reddit.com/r/MachineLearning/comments/1dxtaez/d_what_do_you_all_use_for_large_scale_training/
59. Different performance between deepspeed and fsdp - distributed - PyTorch Forums, accessed October 29, 2025, https://discuss.pytorch.org/t/different-performance-between-deepspeed-and-fsdp/174131
60. Isaac Sim Basic Usage Tutorial — Isaac Sim Documentation, accessed October 29, 2025, https://docs.isaacsim.omniverse.nvidia.com/5.1.0/introduction/quickstart_isaacsim.html
61. Getting started with Isaac™ Sim - Stereolabs, accessed October 29, 2025, https://www.stereolabs.com/docs/isaac-sim/isaac_sim
62. Getting Started Tutorials - Isaac Sim Documentation - NVIDIA, accessed October 29, 2025, https://docs.isaacsim.omniverse.nvidia.com/4.5.0/introduction/quickstart_index.html
63. Python - MuJoCo Documentation, accessed October 29, 2025, https://mujoco.readthedocs.io/en/stable/python.html
64. tutorial.ipynb - Colab - Google, accessed October 29, 2025, https://colab.research.google.com/github/google-deepmind/mujoco/blob/main/python/tutorial.ipynb
65. tayalmanan28/MuJoCo-Tutorial - GitHub, accessed October 29, 2025, https://github.com/tayalmanan28/MuJoCo-Tutorial
66. NovaSky-AI/SkyRL: SkyRL: A Modular Full-stack RL Library for LLMs - GitHub, accessed October 29, 2025, https://github.com/NovaSky-AI/SkyRL
67. SkyRL - UC Berkeley Sky Computing Lab, accessed October 29, 2025, https://sky.cs.berkeley.edu/project/skyrl/
68. SkyRL: A Reinforcement Learning Pipeline for Real-World, Long-Horizon Agents - Medium, accessed October 29, 2025, https://medium.com/@bravekjh/skyrl-a-reinforcement-learning-pipeline-for-real-world-long-horizon-agents-59aaa63a2677
69. SkyRL-v0: Transforming AI Agent Training with Next-Gen Reinforcement Learning, accessed October 29, 2025, https://www.xugj520.cn/en/archives/skyrl-v0-reinforcement-learning-framework.

html
70. SkyRL: Modular Full-Stack RL Training for LLMs — SkyPilot Docs, accessed October 29, 2025, https://docs.skypilot.co/en/latest/examples/training/skyrl.html
71. Train LLMs with reinforcement learning using SkyRL - Anyscale Docs, accessed October 29, 2025, https://docs.anyscale.com/tutorials/train-llm-with-skyrl
72. Development Guide - SkyRL documentation - Read the Docs, accessed October 29, 2025, https://skyrl.readthedocs.io/en/latest/getting-started/development.html
73. Distributed Multi-Node Jobs — SkyPilot documentation, accessed October 29, 2025, https://docs.skypilot.co/en/stable/running-jobs/distributed-jobs.html
74. Distributed Training with PyTorch — SkyPilot Docs, accessed October 29, 2025, https://docs.skypilot.co/en/latest/examples/training/distributed-pytorch.html
75. Creating a New Environment or Task - SkyRL documentation - Read the Docs, accessed October 29, 2025, https://skyrl.readthedocs.io/en/latest/tutorials/new_env.html
76. Welcome to SkyRL's documentation! — SkyRL documentation, accessed October 29, 2025, https://skyrl.readthedocs.io/