ChatGPT

Reinforcement Learning (RL) Project Roadmap: From Basics to Large-Scale Fine-Tuning

To get started with RL for large models, plan a sequence of projects from simple to complex. Begin with fundamental RL tasks on toy environments, then move to language-model fine-tuning (e.g. small LLMs with RL rewards), and parallel robotics simulations. Use well-known RL tools (PyTorch, Gym/Isaac Gym, Stable-Baselines3, Ray RLlib, HuggingFace TRL, etc.) and leverage multi-GPU frameworks (DDP, FSDP, Monarch, Accelerate/DeepSpeed) as you scale up. Below is a structured project roadmap, mixing **general RL** and **robotics/RLHF-style** tasks, each with suggested resources and goals. Emphasize implementing the policy and loss functions yourself, and use open-source datasets or simulators where possible.

Core Tools and Libraries

- **RL Frameworks:** Use libraries like Stable-Baselines3 (SB3) and Ray RLlib for ready-made algorithms (PPO, SAC, DQN, etc.). For example, SB3 provides reliable PyTorch implementations with simple APIs [1]. Ray's RLlib offers scalable, fault-tolerant RL suited for large workloads [2]. These let you focus on experimenting rather than re-implementing algorithms.
- **Language Model Fine-Tuning:** For text-based RL (RLHF or similar), use HuggingFace's TRL or TRLX libraries. TRL explicitly supports PPO for LLM fine-tuning [3] and uses Accelerate/DeepSpeed under the hood for multi-GPU training [4]. You can also look at open-source RLHF pipelines (e.g. CarperAI's TRLX or HuggingFace examples) for guidance.
- **Simulation Environments:** For robotics, start with OpenAI Gym/Gymnasium environments (classic control, continuous control, Atari, etc.) and Gym robotics (FetchReach, Ant, etc.) [5]. NVIDIA's Isaac Gym/Isaac Lab offer GPU-accelerated physics sims and RL scripts for complex robots (multi-GPU support, Hydra configs) [6] [5]. For offline or real-data tasks, consider D4RL (offline RL datasets) or imitation-learning benchmarks.
- **Distributed Training:** On multi-GPU setups, use PyTorch's Distributed DataParallel (DDP) or FSDP. PyTorch's tutorials show how to convert single-GPU code to DDP for 4+ GPUs [7]. For very large models, FSDP shards parameters/optimizer states across GPUs, greatly reducing memory usage [8]. PyTorch Monarch is a new framework that lets you program cluster-wide workloads as if on one machine [9]. HuggingFace Accelerate or DeepSpeed can also manage multi-GPU/large-model training. For example, TRL docs explain using Accelerate to spread PPO training across GPUs and mention DeepSpeed/ZERO for even bigger models [4] [10].

Starter Projects (General RL and Language Tasks)

1. **Classic Control from Scratch (Gym):** Train a simple agent (e.g. CartPole-v1, MountainCar-v0) with a policy-gradient method (REINFORCE or PPO) you implement from scratch. Use a small neural network in PyTorch. This teaches RL fundamentals (state, action, reward loops). Verify your code by matching a known baseline (e.g. CartPole solved). Use OpenAI Gym and PyTorch (or SB3) [1]. This project needs no external dataset beyond the gym envs and is ideal for testing multi-GPU DDP once working on one GPU [7].

2. **Continuous Control / Robotics Gym (Gymnasium-Robotics):** Next, pick a continuous-control task like Pendulum or a simple Gym robotic task (e.g. `FetchReach-v3`, `Ant-v2`, etc.). Implement an off-policy RL algorithm (e.g. DDPG, TD3, or SAC). Use Gym environments from the Farama Gymnasium-Robotics library [5]. This teaches you handling continuous actions and more complex state spaces. You can start on one GPU, then distribute across many using PyTorch DDP. Optionally use SB3 to benchmark your implementation.

3. **Imitation Learning and RL Fine-Tuning (Offline → Online):** Create or use a small dataset of expert trajectories (e.g. trajectories from a rule-based agent or human play). First train a policy via supervised learning (behavior cloning). Then use reinforcement learning (e.g. PPO) to fine-tune this policy with a designed reward function. This illustrates a common pipeline: offline pretraining followed by online RL. Example: mimic stock data actions or play logs, then refine via RL for better performance.

4. **RL on Simple Text Tasks (Policy Gradient on Sequences):** Design a toy text environment. For example, let an agent generate a sentence and give a reward if it matches a target sentiment or category. Implement a policy network (e.g. an RNN or small Transformer) that generates tokens, and optimize it with REINFORCE/PPO. Alternatively, use a question-answering or sentiment classification dataset from Hugging Face and define a reward (correct label = +1, else 0). This project forces you to handle sequences and define custom reward functions. HuggingFace TRL's "sentiment" example is a reference (see PPO Trainer docs) [11]. Start with a small model (GPT-2 small) and few GPUs, then try multi-GPU training with Accelerate.

5. **Verifiable-Reward Tasks (Math/Code):** Use a dataset where correctness can be *automatically checked*. E.g. the GSM8K math word problems or a coding problems dataset (like APPS or CodeAlly). Let your agent (a language model) output a solution, and define reward=1 for exactly correct answer (e.g. string-match or unit tests), else 0 [12]. Train with PPO or another RL method to maximize the verifiable reward. This leverages "verifiable rewards" (precise, rule-based signals) [12]. It is simpler than human feedback, so great for learning RL pipelines on text.

6. **RLHF on a Toy Task:** Build a mini RLHF pipeline. For example, fine-tune a small GPT-2 model to prefer answers scored higher by a pretrained reward model. Use any dataset of prompts and two candidate responses with "preferences" (e.g., Rotten Tomatoes reviews or a toy preference dataset). Train a reward model (or use a heuristic) and then apply PPO via HuggingFace TRL. This ties together supervised fine-tuning (SFT) and RLHF. The TRL documentation provides PPO examples with dummy or real reward models [11].

Each of the above projects lets you *implement the policy and loss functions yourself* and see end-to-end RL training. Use stable-baselines or HuggingFace TRL for reference code, but write your own training loops first. For multi-GPU, use PyTorch DDP or Accelerate. Once a project works on one GPU, scale it by launching with `torch.distributed.launch` or Accelerate (the TRL docs show how) [4].

Starter Projects (Robotics and Simulation)

1. **Classic Robotics Env (Gym Fetch):** Use OpenAI Gym's robotics tasks (e.g. `FetchReach-v3`, `FetchPush-v3`, `FetchPickAndPlace-v3`) [5]. These simulate a 7-DoF arm doing simple

manipulation. Implement an algorithm like DDPG or SAC to control the arm. This teaches continuous control in higher dimensions. Use Mujoco or PyBullet as backend (open-source). You can also try using Isaac Gym if available for faster parallel sim.

2. **Locomotion (MuJoCo):** Train on Ant or Humanoid tasks (`Ant-v3`, `Humanoid-v3`). These are harder. Try PPO or SAC. This step tests scaling to higher-dimensional policies and long horizons. Use Stable-Baselines3 to compare results, then implement your own agent. Multi-GPU training (via RLlib or VecEnv) can speed learning by running many simulators in parallel.

3. **Sim2Real Pipeline (Off-Policy RL + Domain Randomization):** If interested in robotics, set up a small sim2real pipeline. For example, use OpenAI's Fetch environment with randomized physics parameters. First train in simulation (core and high-fidelity sim), then test policies on a "realistic" version of the sim (or real robot if available). This follows ideas from literature on staged pipelines [13]. Document how performance changes across stages.

4. **NVIDIA Isaac Gym / Lab Project:** Try an advanced robotics sim. Isaac Gym (GPU-based physics) can simulate many robots in parallel. Use NVIDIA's sample tasks (like quadruped locomotion or robot arm tasks). Isaac Lab has RL examples and supports distributed training and Hydra configs [6]. Study their provided scripts (they support DDP, population-based training, etc.) and try customizing one environment or creating a simple custom scene. This exposes you to state-of-the-art robotics RL frameworks.

5. **Multi-Agent RL or Control:** As a fun stretch, try a simple multi-agent RL scenario (e.g., two-agent pong or pursuit/evasion). Use Stable-Baselines3 or RLlib which have multi-agent capabilities. This is less essential but shows how to scale to more agents.

Throughout robotics projects, use **open-source simulation data**. Gym and Isaac come with benchmarks and environments. No proprietary data needed. You can log training curves to understand sample efficiency.

Scaling Up: Distributed Training and Large Models

Once you have small projects working, incorporate distributed training:

- **Distributed Data Parallel (DDP):** Convert single-GPU code to DDP using PyTorch's tutorial as a guide [7]. Ensure each process handles its GPU and synchronize gradients with NCCL. This lets you use all 6–8 H100s for larger models or faster learning.
- **Fully Sharded Data Parallel (FSDP):** For very large networks (LLMs or wide nets), wrap your model with PyTorch's FSDP. As the docs note, FSDP shards weights/gradients/optimizer state across GPUs, enabling models that wouldn't fit on one GPU [8]. This is crucial if you move beyond tens of millions of parameters.
- **Hugging Face Accelerate / DeepSpeed:** For LLM training, use Accelerate (which TRL uses) to abstract away the multi-GPU launch [4]. For extreme scale, DeepSpeed with ZeRO optimizes memory (offloading, partitioning) and can handle billion-parameter models [10].
- **PyTorch Monarch:** Experimentally, PyTorch Monarch promises a unified control framework for cluster-based training [9]. It lets you orchestrate multiple GPUs (even across machines) as a single

program. If your RL pipeline logic is complex (e.g. custom rollout servers), Monarch could simplify coding.

Implementing Your Own Framework: The question mentions "making your own training frameworks." Once comfortable, try writing a mini-RL training system: define an `Environment` class, a `Policy` model, and a `Trainer` loop that runs episodes, collects rewards, computes losses, and updates the model (distributed with DDP/FSDP). This is ambitious but hugely instructive. Use libraries like PyTorch Lightning or Ray as inspiration for organizing code.

Advanced Projects

1. **RLHF on a Larger Scale:** Fine-tune a moderately large LLM (e.g. 7B-size) with RL. For example, use a public dialogue or summarization dataset and a synthetic or small human preference reward model. Implement SFT then PPO (via TRL). With 80GB GPUs and FSDP, you can train models that are too big for a single GPU. This practices RLHF-like pipelines in a controlled setting.

2. **Long-Horizon / Tool-Use Tasks:** Explore tasks requiring long sequences of actions. The Sky Computing Lab's *SkyRL* framework is built for multi-turn, real-world LLM tasks (e.g. tool use, coding, SQL) [14]. Use SkyRL-Gym's tool-use environments (math, coding, search, SQL) and train an agent with PPO via SkyRL-Train or their skyrl-agent system. The SkyRL-SQL example (trained on only 653 samples to beat GPT-4) shows the power of focused RL pipelines [15]. Building even a smaller-scale version of such a pipeline (e.g. an LLM querying a search tool and learning from success/failure) would be a cutting-edge project.

3. **Simulation-to-Real Policy Transfer:** If you have a real robot or advanced sim, train a policy in sim (maybe with domain randomization) and then test it in a real-world-like setting. This uses the multi-stage pipeline idea [13]. Document how to refine the policy at each stage.

4. **Curriculum or Population-Based Training:** For extra credit, implement curriculum learning or population-based training (PBT). Start RL in easy environments and gradually increase difficulty. Or maintain a population of agents with varied hyperparameters (as in AlphaStar-style self-play) and evolve them. PyTorch and Ray have utilities for these (e.g. Ray Tune for PBT).

Resources and Further Reading

- **RL Textbooks/Papers:** *Sutton & Barto's* "Reinforcement Learning" book for fundamentals; OpenAI's RLHF papers and blog posts for fine-tuning LMs.
- **Libraries/Docs:** Stable-Baselines3 docs [1], Ray RLlib docs [2], HuggingFace TRL docs [3] [4], PyTorch distributed/FSDP tutorials [7] [8], NVIDIA Isaac Lab RL docs [6], SkyRL repo/docs [14].
- **Datasets:** Gym environments (built-in tasks) [5]; D4RL for offline RL; CIFAR or Text datasets (SST2, GSM8K, APPS) for language tasks; RoboTurk or slippi (for Smash/Melee type imitation).
- **GPU Tools:** NVIDIA Megatron, DeepSpeed, or PyTorch FSDP for big LLMs. Cloud or on-prem clusters since you have H100s.
- **Example Projects:** Eric Gu's Melee-PT blog [16] shows building a custom RL pipeline (Transformer on 3B frames). While domain-specific, it illustrates data processing, SFT, and RL phases.

Each project above should start small (e.g. a tiny network, few GPUs, fewer timesteps) and then scale up. Document your experiments, note how adding GPUs or larger models changes performance. By the end you'll have hands-on experience with full RL training pipelines – from writing policy networks and losses to orchestrating distributed training and fine-tuning large models.

Citations: We have used official docs and papers for guidance. For example, HuggingFace's TRL docs note that PPO is used for RLHF fine-tuning [3], and Label Studio's blog explains "verifiable rewards" for LLM tasks [12]. Stable-Baselines3 is recommended for reliable off-the-shelf RL algorithms [1], and Ray RLlib is cited for scalable RL workloads [2]. PyTorch tutorials cover multi-GPU DDP [7] and FSDP memory-saving sharding [8]. NVIDIA's Gym robotics docs list Fetch arm tasks [5]. SkyRL (UC Berkeley) provides an LLM-centric RL framework for long-horizon tasks [14]. Finally, an inspiring example project (Melee-PT) shows a custom RL pipeline on game data [16].

---

[1] Stable-Baselines3: Reliable Reinforcement Learning Implementations | Antonin Raffin | Homepage
https://araffin.github.io/post/sb3/

[2] RLlib: Industry-Grade, Scalable Reinforcement Learning — Ray 2.51.0
https://docs.ray.io/en/latest/rllib/index.html

[3] [11] PPO Trainer
https://huggingface.co/docs/trl/main/en/ppo_trainer

[4] [10] Distributing Training
https://huggingface.co/docs/trl/main/en/distributing_training

[5] Fetch - Gymnasium-Robotics Documentation
https://robotics.farama.org/envs/fetch/index.html

[6] Reinforcement Learning — Isaac Lab Documentation
https://isaac-sim.github.io/IsaacLab/main/source/overview/reinforcement-learning/index.html

[7] Multi GPU training with DDP — PyTorch Tutorials 2.9.0+cu128 documentation
https://docs.pytorch.org/tutorials/beginner/ddp_series_multigpu.html

[8] Getting Started with Fully Sharded Data Parallel (FSDP2) — PyTorch Tutorials 2.9.0+cu128 documentation
https://docs.pytorch.org/tutorials/intermediate/FSDP_tutorial.html

[9] Introducing PyTorch Monarch – PyTorch
https://pytorch.org/blog/introducing-pytorch-monarch/

[12] Reinforcement Learning from Verifiable Rewards | Label Studio
https://labelstud.io/blog/reinforcement-learning-from-verifiable-rewards/

[13] A Simulation Pipeline to Facilitate Real-World Robotic Reinforcement Learning Applications
https://arxiv.org/html/2502.15649v1

[14] [15] GitHub - NovaSky-AI/SkyRL: SkyRL: A Modular Full-stack RL Library for LLMs
https://github.com/NovaSky-AI/SkyRL

[16] Training AI to play Super Smash Bros. Melee - Eric Gu
https://ericyuegu.com/melee-pt1