

# Contents

<b>1 Tutorials</b>	<b>1</b>
1.1 Prologue: What Problem Are We Solving?	1
1.1.1 The Practical Problem	1
1.1.2 Why This Is Hard	1
1.1.3 When to Use This Approach	2
1.1.4 What You Will Build	2
1.2 On the Purpose of This Collection	2
1.3 The Central Questions	3
1.4 The Methodological Framework	3
1.4.1 WHY: Problem Formulation	3
1.4.2 HOW: Methodology and Derivation	4
1.4.3 WHAT: Implementation and Verification	4
1.5 The Structure of the Chapters	4
1.6 The Chapter Sequence	5
1.7 Prerequisites	5
1.8 How to Use These Tutorials	6
1.9 Running Experiments	6
1.9.1 The Execution Pattern	6
1.9.2 Chapter Scripts	6
1.9.3 Understanding GPU Utilization	6
1.9.4 Monitoring Training	7
1.9.5 Long-Running Jobs	7
1.9.6 Output Artifacts	7
1.10 On the Voice of These Tutorials	7

## 1 Tutorials

**PDFs.** A rendered PDF copy is available under docs/pdfs/tutorials/ (auto-generated on push to main).

### 1.1 Prologue: What Problem Are We Solving?

Before diving into mathematics and algorithms, let us be concrete about the situation this curriculum addresses.

#### 1.1.1 The Practical Problem

You want to build a robot that manipulates objects to arbitrary goal positions. Not a robot that performs one fixed task--that is classical control engineering--but a robot that can:

- Reach any 3D position in its workspace
- Push any object to any target location
- Pick up any object and place it anywhere

The key word is **any**. The goal is specified at runtime, not at programming time. The robot must generalize across an infinite space of possible tasks.

#### 1.1.2 Why This Is Hard

**Sparse rewards.** The natural feedback is binary: did you succeed or not? You don't get partial credit for "almost" reaching the goal. This means most training episodes provide zero learning

signal.

**High-dimensional continuous control.** The robot issues 4D velocity commands at 50Hz. Random exploration in this space almost never reaches a specified goal by chance.

**Goal generalization.** You cannot train a separate policy for each possible goal--there are infinitely many. The policy must take the goal as input and generalize to goals it has never seen during training.

### 1.1.3 When to Use This Approach

The SAC + HER methodology we teach is appropriate when:

Condition	Example
Goals vary at runtime	"Move to position (0.3, 0.2, 0.1)" specified at test time
Rewards are sparse	Success/failure signal, no dense shaping
Actions are continuous	Velocity commands, torques--not discrete choices
Simulation is available	Training requires millions of trials

If your problem matches these conditions, this curriculum provides the methodologically appropriate solution. If your problem has dense rewards, discrete actions, or a single fixed goal, simpler methods exist and you should use them.

### 1.1.4 What You Will Build

By the end of this curriculum:

1. **A working policy** achieving >90% success on FetchReach, FetchPush, and FetchPickAndPlace
2. **Reproducible infrastructure** with Docker, versioned experiments, and statistical rigor
3. **Diagnostic capability** to understand *why* training succeeds or fails
4. **Deep understanding** of how problem structure dictates algorithm choice

#### What 500,000 training episodes produces:

Robot reaching goals

*No one programmed these movements. The robot discovered them through trial and error--mostly error, at first.*

This is a significant investment--expect weeks, not hours. The payoff is genuine research capability, not superficial familiarity with library calls.

---

## 1.2 On the Purpose of This Collection

This collection of tutorials constitutes a systematic treatment of goal-conditioned reinforcement learning for robotic manipulation. It is not a quickstart guide, nor a collection of recipes, nor a reference manual. It is, in the tradition of the great mathematical textbooks, a *course of study*: a carefully sequenced development of ideas, each building on what came before, designed to produce genuine understanding rather than superficial familiarity.

The reader who completes this course will not merely know how to train a policy on Fetch tasks. They will understand *why* goal-conditioned formulations enable certain algorithmic techniques; *how* the mathematical structure of the problem constrains the space of effective solutions; and

*what* properties a well-designed experimental pipeline must satisfy to produce reproducible results.

This is the standard we set. Anything less is not worth the reader's time.

### 1.3 The Central Questions

Before diving into algorithms, we ask fundamental questions about the problem itself. This habit--common in applied mathematics and physics--helps avoid wasted effort on ill-posed problems. For reinforcement learning in robotic manipulation, the key questions are:

**The Existence Question.** Does there exist, within our hypothesis class of neural network policies, a function that achieves high success rate on the task distribution? This question is non-trivial. The hypothesis class is constrained (feedforward networks of bounded depth and width); the task distribution may include configurations that are kinematically difficult or dynamically unstable; the reward landscape may be deceptive, with local optima that trap gradient-based methods.

**The Learnability Question.** Given that a good policy exists, can our learning algorithm find it? The gap between existence and learnability is vast. A policy may exist in principle yet be unreachable by stochastic gradient descent from random initialization. The learning algorithm may require more samples than are practically available, or may be sensitive to hyperparameters in ways that make reliable training impossible.

**The Stability Question.** Suppose we find a good policy. Does it depend continuously on the training process? If small changes to the random seed, the hyperparameter settings, or the training data produce qualitatively different policies, then our results are not reproducible in any scientifically meaningful sense. The notorious instability of deep RL training makes this question particularly urgent.

These tutorials do not answer these questions in full generality--that would require a treatise, not a tutorial series. But they provide the conceptual framework and empirical tools to investigate these questions for specific tasks and algorithms. The reader who completes this course will know how to *ask* these questions precisely and how to *answer* them through careful experimentation.

### 1.4 The Methodological Framework

Every chapter in this collection follows a tripartite structure: **WHY, HOW, WHAT**.

#### 1.4.1 WHY: Problem Formulation

Before presenting any technique, we formulate the problem it addresses. What mathematical object are we seeking? What properties must it satisfy? What makes this problem difficult?

This is not mere throat-clearing. Problem formulation is itself a creative act that shapes everything that follows. A poorly formulated problem admits no clean solution; a well-formulated problem often suggests its own resolution.

When we introduce Hindsight Experience Replay in Chapter 4, we do not begin with the algorithm. We begin with the problem: sparse binary rewards provide no gradient signal when the goal is never reached; the probability of reaching an arbitrary goal by random exploration decreases exponentially with goal-space dimension; therefore, any practical algorithm must manufacture learning signal from failed attempts. The problem formulation makes the solution almost inevitable.

### 1.4.2 HOW: Methodology and Derivation

Having established *what* problem we are solving and *why* it is difficult, we turn to *how* we shall solve it.

Here we follow the pedagogical tradition exemplified by Sergei Levine's CS 285: algorithms are *derived*, not *declared*. We do not present update equations as fait accompli; we show where they come from. We do not invoke design choices as received wisdom; we justify them from first principles.

When we introduce Soft Actor-Critic, we begin with the maximum entropy objective and derive the soft Bellman equation, the Q-function update, and the policy improvement step as logical consequences. A student who has followed this derivation understands not just *what* SAC does but *why* it does it--and therefore knows how to adapt it when standard settings fail.

### 1.4.3 WHAT: Implementation and Verification

The transition from mathematical abstraction to working code is where many researchers falter. The gap between "I understand the algorithm" and "my implementation produces correct results" is vast, and it is bridged only by meticulous attention to detail.

Every chapter concludes with explicit deliverables: files that must exist, tests that must pass, metrics that must lie within specified ranges. These are not administrative overhead; they are the empirical verification that the mathematical ideas have been correctly instantiated in code.

A reader who cannot produce the specified deliverables has not completed the chapter, regardless of how well they believe they understand the material. Understanding without verification is indistinguishable from misunderstanding.

## 1.5 The Structure of the Chapters

Each chapter follows a consistent organization, modeled on the structure of mathematical monographs:

**Abstract.** A concise statement of what the chapter accomplishes: what problem is addressed, what methods are introduced, what deliverables are produced.

**Theoretical Foundations.** The mathematical context for the chapter's content. Definitions are stated precisely; connections to the broader literature are established; the reader understands *why* the material matters before seeing *how* it works.

**The Experimental Environment.** The computational context for the chapter's experiments. We specify the container, the hardware assumptions, the environment variables, and any other infrastructure requirements. Reproducibility demands that these details be explicit.

**Implementation Details.** The core of the chapter: code (in the form of versioned scripts), analysis, and explanation. We do not merely show what to type; we explain what it means and why it works.

**Verification and Sanity Checks.** Concrete tests that confirm correct implementation. A reader who passes these tests has correctly instantiated the chapter's ideas; a reader who fails them has made an error that must be diagnosed before proceeding.

**Deliverables.** The artifacts that constitute "done": files, metrics, plots. The reader knows unambiguously what they must produce.

## 1.6 The Chapter Sequence

The chapters are ordered to build systematically from foundations to applications:

**Chapter 0: A Containerized "Proof of Life" on Spark DGX.** The existence of a working experimental environment is the precondition for all subsequent work. This chapter establishes that precondition: we verify GPU access, enter a reproducible container, validate MuJoCo rendering, and confirm that a training loop executes without error. No learning happens here; we are merely establishing that the laboratory is functional.

*WHY:* Without a reproducible environment, no experimental result can be trusted. Containerization is not a convenience; it is a requirement for scientific validity.

*HOW:* We use Docker with the NVIDIA runtime, mount the repository into the container, manage dependencies via a virtual environment, and configure rendering backends for headless operation.

*WHAT:* A working `docker/dev.sh` entrypoint; a rendered frame proving MuJoCo works; a saved checkpoint proving the training loop completes.

**Chapter 1: The Anatomy of Goal-Conditioned Fetch Environments.** Before training any agent, we must understand precisely what the agent perceives, what actions it can take, and how rewards are computed. This chapter dissects the Fetch environment interface.

*WHY:* Goal-conditioned environments have a specific structure--observation dictionaries with `achieved_goal` and `desired_goal` keys, reward functions that can be queried for arbitrary goals--that enables techniques like Hindsight Experience Replay. Understanding this structure is prerequisite to using those techniques correctly.

*HOW:* We inspect observation and action spaces programmatically, verify reward consistency between `env.step()` and `compute_reward()`, and collect baseline metrics with random policies.

*WHAT:* JSON schemas describing the environment interface; verified reward consistency; random baseline metrics establishing the performance floor.

Subsequent chapters--on PPO baselines, SAC, HER, robustness, and beyond--follow the same structure, each building on the foundations laid by its predecessors.

## 1.7 Prerequisites

**Mathematical.** The reader should be comfortable with the fundamentals of probability theory (random variables, expectations, conditional distributions), optimization (gradients, stochastic gradient descent), and the basic formalism of Markov Decision Processes (states, actions, transitions, rewards, policies, value functions). Sutton and Barto's *Reinforcement Learning: An Introduction* provides sufficient background; familiarity with Bertsekas and Tsitsiklis's *Neuro-Dynamic Programming* is helpful but not required.

**Computational.** The reader should be comfortable with Python programming, command-line operations, and basic Docker usage. Familiarity with PyTorch is helpful; familiarity with Stable Baselines 3 is not required and will be developed in the tutorials.

**Infrastructural.** The reader should have access to a machine with an NVIDIA GPU and a properly configured Docker installation. The tutorials are written with the Spark DGX cluster in mind but should be adaptable to any similar environment.

## 1.8 How to Use These Tutorials

**Sequential completion is expected.** The chapters build on each other; skipping ahead will leave gaps in understanding that will manifest as confusion or incorrect implementations later. A reader who cannot complete Chapter 0 has no business attempting Chapter 3.

**Verification is not optional.** Every chapter includes sanity checks and deliverables. Complete them. A reader who skips verification to save time will spend more time later debugging problems that could have been caught early.

**Understanding is the goal, not speed.** These tutorials are designed to be worked through carefully, not skimmed. A reader who spends a week on Chapter 1 and emerges with genuine understanding has accomplished more than one who rushes through all chapters in a day and emerges with recipes they cannot adapt.

**The scripts are the source of truth.** Tutorials reference scripts in the `scripts/` directory; they do not embed code to be copied and pasted. The scripts are versioned, tested, and maintained. The prose explains what the scripts do and why; the scripts are what you actually run.

## 1.9 Running Experiments

### 1.9.1 The Execution Pattern

All commands run through Docker via `docker/dev.sh`. This wrapper handles GPU access, dependencies, and rendering configuration:

```
# Pattern: bash docker/dev.sh <command>
bash docker/dev.sh python scripts/ch02_ppo_dense_reach.py all
bash docker/dev.sh python train.py --algo sac --env FetchReach-v4 --total-steps 1000000
```

### 1.9.2 Chapter Scripts

Each chapter provides a self-contained orchestration script that runs the full pipeline:

Script	Description
<code>scripts/ch00_proof_of_life.py</code>	GPU/MuJoCo/render validation
<code>scripts/ch01_env_anatomy.py</code>	Environment inspection and baselines
<code>scripts/ch02_ppo_dense_reach.py</code>	PPO training, eval, reporting
<code>scripts/ch03_sac_dense_reach.py</code>	SAC training with diagnostics

Usage pattern:

```
# Full pipeline
bash docker/dev.sh python scripts/ch03_sac_dense_reach.py all --seed 0

# Individual steps
bash docker/dev.sh python scripts/ch03_sac_dense_reach.py train --total-steps 100000
bash docker/dev.sh python scripts/ch03_sac_dense_reach.py eval
bash docker/dev.sh python scripts/ch03_sac_dense_reach.py compare
```

### 1.9.3 Understanding GPU Utilization

You may observe low GPU utilization (~5-10%) during training. **This is expected.** RL training is CPU-bound:

Component	Hardware	Notes
MuJoCo physics simulation	CPU	Dominant cost (~60-70%)
Neural network forward/backward	GPU	Small batch sizes = low utilization
Replay buffer operations	CPU	Memory bandwidth limited

With `batch_size=256` and simple MLPs, GPU operations complete in microseconds. The GPU idles while waiting for CPU simulation. Typical throughput: ~600 fps on DGX-class hardware.

### 1.9.4 Monitoring Training

TensorBoard provides real-time training visibility:

```
bash docker/dev.sh tensorboard --logdir runs --bind_all
# Open http://localhost:6006 in browser
```

Key metrics to watch:

Metric	Meaning	Healthy Behavior
rollout/success_rate	Task success	Increases to >0.9
rollout/ep_rew_mean	Episode return	Less negative over time
train/value_loss	Critic error	Decreases, then stabilizes
replay/q_min_mean (SAC)	Q-value estimates	Stabilizes, doesn't explode
replay/ent_coef (SAC)	Entropy temperature	Decreases from ~1 to ~0.1-0.5

### 1.9.5 Long-Running Jobs

Use tmux to persist sessions across SSH disconnections:

```
tmux new -s rl # Create session
bash docker/dev.sh python scripts/ch03_sac_dense_reach.py all
# Ctrl-b d # Detach (job keeps running)
tmux attach -t rl # Reattach later
```

### 1.9.6 Output Artifacts

Artifact	Location	Purpose
Checkpoints	checkpoints/*.zip	Trained model weights
Metadata	checkpoints/*.meta.json	Hyperparams, versions, timing
Eval reports	results/*.json	Per-episode + aggregate metrics
TensorBoard	runs/	Training curves and diagnostics
Videos	videos/	Rendered policy rollouts

## 1.10 On the Voice of These Tutorials

These tutorials strive for precision and clarity. Where the material is complex, we try to explain the complexity rather than hide it. Where shortcuts exist, we try to show the full path first so the reader understands what the shortcut elides.

This approach reflects a belief that reinforcement learning for robotic manipulation is genuinely difficult, and that readers deserve to be told when something is hard rather than left to wonder

why they are struggling. The formality is not a barrier; it is an attempt at respect--for the subject and for the reader's time.

These tutorials are imperfect. They contain errors, unclear passages, and explanations that may not work for every learner. Corrections, questions, and suggestions are welcome. The goal is understanding, and if these materials fail to produce it, the fault lies with the materials, not the reader.

---

*"The purpose of a tutorial is not to make the subject seem easy. It is to make the subject become understood."*