Reinforcement Learning

Lecture 1: From Mathematical Models to Learning Sequential Decisions

Let's trace how data changes the usual problem-solving process, what counts as convincing evidence, and where reinforcement learning fits.

By the end of this lecture, you should be able to

- Separate problem specification, mathematical modeling, structural analysis, algorithm design, and verification.
- Explain how machine learning can act as either a *model surrogate* or an *algorithm surrogate*.
- Use route planning to compare a conventional solver, a hybrid learned model plus solver, and a learned decision policy.
- Distinguish the main machine-learning paradigms by their data, feedback, and objective.
- State what evaluation on a small set of models and datasets does—and does not—establish.
- Define reinforcement learning as sequential decision-making and explain its distinctive challenges.
- Place modern reinforcement learning within its major historical lineages.

Contents

1 How we usually solve problems **2**
- 1.1 Step 1: pin down the real problem . . . 2
- 1.2 Step 2: turn the problem into mathematics . . . 2
- 1.3 Step 3: look for useful structure . . . 3
- 1.4 Step 4: turn the structure into an algorithm . . . 3
- 1.5 Step 5: check the algorithm—and the model . . . 3

2 How learning from data changes the process **4**
- 2.1 Learn the model, keep the solver . . . 4
- 2.2 Learn the solver itself . . . 4
- 2.3 We are still modeling . . . 5

3 Let's work through route planning **5**
- 3.1 Start with known costs . . . 5
- 3.2 Learn the costs, keep the planner . . . 6
- 3.3 Learn the route directly . . . 7
- 3.4 When does this become reinforcement learning? . . . 7

3.5 Questions to discuss . . . 7

4 A quick map of machine-learning paradigms 8

5 Why a few benchmarks are not enough 9
5.1 Why a small test can mislead us . . . 9
5.2 What the statistics actually tells us . . . 10
5.3 Return to the route guarantee . . . 10
5.4 Match the evidence to the claim . . . 11

6 So, what is reinforcement learning? 12
6.1 Put the interaction into an MDP . . . 12
6.2 What makes RL different . . . 12

7 How reinforcement learning got here 14

8 Let's pull the ideas together 15

1 How we usually solve problems

Let's begin with the conventional way of solving a scientific or engineering problem. The details vary, but the basic sequence is usually the same:

$$\boxed{\text{real problem}} \longrightarrow \boxed{\text{mathematical model}} \longrightarrow \boxed{\text{structure}} \longrightarrow \boxed{\text{algorithm}} \longrightarrow \boxed{\text{verification and validation}} \tag{1.1}$$

These boxes may look like one continuous process, but they represent different jobs. Keeping them separate matters. Otherwise, it is easy to claim that we have solved the real problem when we have only solved a mathematical version of it.

1.1 Step 1: pin down the real problem

Before we write any equations, we need to decide what success actually means. Start with four questions:

1. What information is available as input?
2. Which outputs or actions are feasible?
3. What objective should be minimized or maximized?
4. Which constraints, risks, or failure modes are non-negotiable?

Consider the instruction "find a good route." What does *good* mean here? The shortest distance? The smallest expected travel time? The lowest fuel use? The smallest chance of arriving late? Are toll roads allowed? Must the route still work if a road closes? These are not minor implementation choices. They define the problem we are trying to solve.

1.2 Step 2: turn the problem into mathematics

Once the goal is clear, we need a mathematical model. We might represent the world using a graph, a vector, a probability distribution, or a differential equation. In route planning, for example, we

might treat the road network as a directed graph, assign a known nonnegative cost to every road, and define the cost of a route as the sum of its road costs.

Of course, the real world contains much more detail than this model. That is the point of abstraction: we deliberately leave out details so that we can reason about the problem. But this simplification creates a *model gap*. Later, we must ask whether that gap is small enough for the intended use.

1.3 Step 3: look for useful structure

Now ask: what useful structure does the model reveal? Here are some common examples:

- If the problem is convex, local optimality also gives us global optimality.
- If it is sparse, we can avoid working with quantities that are mostly zero.
- Monotonicity or submodularity may let us justify a greedy method.
- A graph makes paths, cuts, and connectivity visible.
- Decomposability and optimal substructure open the door to dynamic programming.
- Invariants or conservation laws tell us what cannot change.

This structure is more than a convenient description. It explains why a method should work on more than the few examples we happened to try.

1.4 Step 4: turn the structure into an algorithm

Once we see the structure, we can design an algorithm that uses it. An algorithm is a finite procedure that maps modeled inputs to outputs, but a good algorithm does more than return an answer. It uses the structure to improve correctness, running time, numerical stability, or approximation quality. Dijkstra's algorithm is a good example: it uses nonnegative edge costs to justify a greedy decision about which vertex to finalize next.

1.5 Step 5: check the algorithm—and the model

Finally, we need to ask two different questions. They sound similar, but they are not the same:

Verification Did the algorithm or implementation solve the mathematical problem we gave it? We might answer this with a proof of correctness, a complexity bound, a numerical error analysis, tests, or formal verification of the code.

Validation Did we write down the right mathematical problem in the first place? Here we need measurements, sensitivity analysis, calibration, field studies, or review by domain experts.

> **Key idea.** A proof can tell us that an algorithm exactly solves a model, provided the model's assumptions hold. It cannot tell us, by itself, that the model is a faithful description of the world.

One way to remember the distinction is to think of the classical pipeline as making two contracts:

$$\underbrace{\text{world} \approx \text{model}}_{\text{modeling contract}} \qquad \text{and} \qquad \underbrace{\text{algorithm solves model}}_{\text{algorithmic contract}}. \tag{1.2}$$

For a system to work reliably in practice, we need evidence for both contracts.

2 How learning from data changes the process

So where does machine learning enter this pipeline? Often, it enters where the real system is too complicated to describe completely by hand. Traffic, for example, depends on time, weather, events, human behavior, and many interacting factors. Instead of specifying every relationship explicitly, we can use data to learn a useful mapping.

In a simplified learning problem, we choose a hypothesis class $\mathcal{F}$, a loss ℓ, and a training procedure. Given a dataset $\mathcal{D}$, we then look for a function that performs well on that data:

$$\widehat{f} \in \arg\min_{f \in \mathcal{F}} \frac{1}{|\mathcal{D}|} \sum_{z \in \mathcal{D}} \ell(f; z) + \lambda\, \Omega(f), \tag{2.1}$$

Here, Ω expresses a preference such as simplicity or smoothness. The important point for us is that learning can replace two quite different parts of the conventional pipeline.

2.1 Learn the model, keep the solver

The first option is to learn a missing part of the model and then let a conventional algorithm do the final computation. We call the learned part a *model surrogate*:

$$x \xrightarrow{\text{learn from data}} \widehat{M}_x \xrightarrow{\text{known optimizer or planner}} y. \tag{2.2}$$

Think of predicting demand before solving an inventory problem, estimating aerodynamic forces before optimizing a design, or predicting travel times before running a shortest-path algorithm. In each case, learning supplies information that the model does not already contain.

Why is this hybrid design attractive? We can keep known constraints as hard constraints, and the established solver keeps its conditional correctness guarantee. But we now have a new question to answer: how do errors in $\widehat{M}_x$ affect the final decision?

2.2 Learn the solver itself

The second option is more direct: learn the solution map itself. This learned map is an *algorithm surrogate*:

$$x \xrightarrow{\text{learn from examples, demonstrations, or rewards}} \widehat{y}. \tag{2.3}$$

For example, a neural network could look at a road network and a destination and immediately recommend the next turn. Once trained, this may make repeated decisions very fast. In optimization, we sometimes say that training has *amortized* the computation across many problem instances.

The tradeoff is that we no longer inherit all the guarantees of the original hand-designed algorithm. The learned output may be infeasible or suboptimal, and its behavior outside the training distribution may be difficult to predict.

2.3 We are still modeling

It is tempting to say that data has replaced the model. A better description is that the modeling assumptions have moved. We still have to make choices about:

- what data to collect and which population it represents;
- how inputs and outputs are represented;
- which target or reward stands in for the real goal;
- which loss function assigns importance to different errors;
- which model class and optimization method are used;
- how evaluation cases are sampled and which metrics are reported.

Together, these choices form an *implicit model*. The assumptions are now spread across the dataset and training pipeline, so they may be harder to see than assumptions written in a short set of equations.

Table 1: What changes when part of the problem-solving pipeline is learned.

Question	Conventional emphasis	Learning-based emphasis
Where do assumptions live?	Explicit model, theorem hypotheses, and algorithm	Data collection, representation, loss, model class, and training
What supports correctness?	Deductive proof under stated assumptions	Statistical evidence plus any preserved structural guarantees
What is the common failure?	Wrong model or violated theorem assumptions	Distribution shift, proxy mismatch, rare failures, or optimization error
How is performance assessed?	Correctness, complexity, approximation, and stability	Generalization, calibration, robustness, downstream utility, and cost

In practice, the most dependable design is often a hybrid one. Learn the part that is genuinely unknown, keep trusted structure wherever we can, and make the boundary between the learned and engineered components clear.

3 Let's work through route planning

Let's make these ideas concrete with route planning. The example will let us compare the conventional pipeline with both kinds of learned surrogate. It will also show us why good predictive accuracy does not automatically mean good decisions.

3.1 Start with known costs

Start by representing the road network as a directed graph $G = (V, E)$. We have an origin $s \in V$, a destination $t \in V$, and a known nonnegative cost $c_e \geq 0$ for every edge. The goal is:

$$P^* \in \operatorname*{arg\,min}_{P : s \rightsquigarrow t} c(P), \qquad c(P) = \sum_{e \in P} c_e. \tag{3.1}$$

What structure can we use? First, the cost of a path is the sum of its edge costs. Second, every prefix of a shortest path is itself a shortest path between its endpoints. Because all edge costs are nonnegative, we can also use a greedy finalization rule safely.

This is exactly what Dijkstra's algorithm does. It keeps a tentative distance for every vertex. At each step, it finalizes the unsettled vertex with the smallest tentative distance and then relaxes that vertex's outgoing edges. The key invariant is simple: whenever a vertex is finalized, its distance is the true shortest-path distance.

Why the invariant holds. Why is that true? Suppose the invariant held before we selected a vertex u. Imagine, for contradiction, that there were a strictly cheaper path to u. Look at the first unsettled vertex on that path and at its finalized predecessor. When we relaxed the edge from that predecessor, the first unsettled vertex would have received a tentative distance no greater than the cost of the supposedly cheaper path. Since all remaining edge costs are nonnegative, that tentative distance would be smaller than the value for u. But then the algorithm would not have selected u. This contradicts the greedy choice. Therefore u's label is final, and induction gives us correctness.

This is a strong guarantee, but notice what it says: the returned route is optimal for the graph and costs that we supplied. If the graph misses a road closure, or if static costs fail to capture congestion, Dijkstra's algorithm may correctly solve the wrong instance.

3.2 Learn the costs, keep the planner

What if the road costs are not known in advance? Suppose the cost of edge e depends on a context vector x, containing information such as the time, weather, nearby events, and recent traffic. We can use historical trips to train:

$$\widehat{c}_e(x) = f_\theta(e, x), \tag{3.2}$$

We constrain the predictions so that $\widehat{c}_e(x) \geq 0$, and then run Dijkstra's algorithm using those predicted costs. The roles are now cleanly separated: the predictor stands in for the unknown traffic model, while Dijkstra remains the planner.

Can we connect prediction error to route quality? Yes, under a stronger error assumption. Suppose every relevant candidate path has at most H edges and:

$$|\widehat{c}_e - c_e| \leq \varepsilon \quad \text{for every relevant edge } e. \tag{3.3}$$

Let $\widehat{P}$ be the path with the smallest predicted cost, and let P^* be the path with the smallest true cost. Then:

$$\begin{aligned} c(\widehat{P}) &\leq \widehat{c}(\widehat{P}) + H\varepsilon && (3.4)\\ &\leq \widehat{c}(P^*) + H\varepsilon && (3.5)\\ &\leq c(P^*) + 2H\varepsilon. && (3.6) \end{aligned}$$

Read the three lines one at a time. The first inequality converts the true cost of $\widehat{P}$ into predicted cost using the uniform error bound. The middle inequality uses the fact that $\widehat{P}$ is optimal under the predicted costs. The final inequality converts the predicted cost of P^* back into true cost.

Equation (3.6) is a small example of a *compositional guarantee*: we connect the learned component's error directly to the final decision quality. It also tells us why average test error may not be enough. A single badly underestimated edge can attract the optimizer even when the mean prediction error is small.

3.3 Learn the route directly

What if we skip the conventional planner and learn the route directly? A model could map the graph, context, origin, and destination to a complete route or to the next action. We might train it using optimal routes as supervised labels, expert demonstrations as imitation data, or rewards collected through interaction.

This direct model may be fast, but now we have to ask a new set of questions:

- Can it output a nonexistent edge or a route that never reaches the destination?
- Does it remember physical or legal constraints that were rare in the training data?
- What happens in a new city, during a closure, or under an unusual event?
- Does it optimize actual travel time or merely imitate historical routes?

These questions do not force us to abandon learning. They suggest practical safeguards. We might mask invalid actions, use the learned network only as a search heuristic, or send its output through a constrained planner. In this way, we can combine learned speed with structural protection.

3.4 When does this become reinforcement learning?

So far, we have mostly treated route planning as a one-shot computation. Static shortest-path planning is not necessarily reinforcement learning. The problem becomes sequential when the traveler moves, observes new congestion, decides whether to reroute, and accumulates travel-time costs throughout the trip. Each choice affects the next location, the available roads, future observations, and the decisions that remain. Learning a policy from this kind of feedback is an RL problem.

> **Evidence check.** Match the evaluation to the role that learning plays. For a cost predictor, RMSE is useful, but it is not enough. We should also measure route regret, constraint violations, tail delays, calibration, and behavior under meaningful traffic shifts. For a policy, we must evaluate the trajectories that the policy itself creates.

3.5 Questions to discuss

Use the following questions to check where the assumptions and guarantees live:

1. Can a new cost predictor have lower RMSE but produce worse routes? Give a small graph where this occurs.
2. Which guarantee belongs to Dijkstra's algorithm, and which evidence is needed for the learned traffic model?
3. If deployment includes a city absent from training, what assumption has changed? Which experiment would reveal the failure?
4. In a safety-critical navigation system, which constraints should remain explicit rather than learned from examples?
5. At what point does adaptive navigation become an RL problem rather than repeated supervised prediction?

Running-example summary

The same route-planning application can represent several different problem-solving paradigms. The key questions are: What do we already know? What are we learning? And how do today's decisions affect the data we will see next?

Formulation	Learned component	Main assurance question
Known static graph	None; Dijkstra performs exact planning	Are the graph and nonnegative costs faithful to the world?
Predicted edge costs	Traffic model $\widehat{c}_e(x)$	Do prediction-error guarantees imply acceptable route regret?
Direct route prediction	The solution map or next-action rule	Are routes feasible, robust to shift, and good on the downstream metric?
Adaptive navigation	A policy, value function, environment model, or a combination	Does the policy achieve safe long-run performance on the trajectories it induces?

4 A quick map of machine-learning paradigms

Here is a simple way to tell the main machine-learning paradigms apart: ask what information the learner receives and what it is trying to achieve. These paradigms are not mutually exclusive boxes. A real system may use self-supervision to learn a representation, supervised learning to predict travel times, and reinforcement learning to choose actions.

Table 2: Common learning paradigms, organized by learning signal.

Paradigm	Learning signal	Primary goal	Route-planning example
Supervised learning	Labeled pairs (x, y)	Predict a label or numerical target on new examples	Predict an edge's travel time from context
Unsupervised learning	Unlabeled observations	Discover structure, clusters, or a data distribution	Cluster roads by traffic pattern
Self-supervised learning	Targets constructed from the observations themselves	Learn representations useful for later tasks	Predict masked segments of GPS trajectories
Semi-supervised learning	A small labeled set and a larger unlabeled set	Improve prediction when annotation is scarce	Combine measured travel times with raw trajectories
Imitation learning	Demonstrations of actions chosen by another policy	Reproduce demonstrated behavior	Imitate routes selected by human drivers or a planner
Contextual bandits	Reward for the chosen action, with no modeled long-term action effect	Maximize cumulative one-step reward while learning action quality	Choose one of several complete routes before a trip
Reinforcement learning	Rewards and observations along action-dependent trajectories	Learn a policy maximizing long-run expected return	Make adaptive turn and rerouting decisions during a trip

You will also hear several other labels. These answer different questions, so they cut across the paradigms in Table 2:

- **generative versus discriminative** asks whether we learn a data distribution or a conditional relationship used for prediction;
- **offline versus online** asks whether learning happens from a fixed dataset or continues while the system interacts with the world;

- **model-based versus model-free RL** asks whether the agent learns or uses an explicit model of the environment;
- **parametric versus nonparametric** asks how the model's capacity is controlled;
- **deep learning** tells us that multilayer neural networks are being used. It does not, by itself, tell us what learning signal is available.

> **Key idea.** Before attaching a paradigm label, pause and ask three questions: What feedback does the learner observe? Who chooses the actions that generate the data? Are we trying to make an immediate prediction or improve a long-term sequence of decisions?

5 Why a few benchmarks are not enough

Suppose we test several model variants on several datasets and one method comes out ahead. What have we learned? We have useful *empirical evaluation*: evidence about what happened in those particular experiments. In ordinary ML usage this is sometimes called "verification," but that word can suggest a much stronger conclusion. A short benchmark table does not, by itself, verify that "the method is robust," "the architecture is generally better," or "the system is safe." This distinction matters especially in deep RL, where random seeds and seemingly minor implementation choices can reverse the ranking of two methods (Henderson et al., 2018; Agarwal et al., 2021).

5.1 Why a small test can mislead us

Here are seven reasons to be cautious when reading a small set of benchmark results:

1. **A familiar benchmark may stop behaving like a test set.** Researchers repeatedly inspect results on the same benchmarks and then adjust architectures, losses, and hyperparameters. Over time, those choices can indirectly fit the benchmark even if nobody explicitly trains on its test examples.
2. **Rows are not the same as domains.** A million road segments from one city may tell us a great deal about that city, but they still give us only one city-level example. If our claim concerns transfer to new cities, three datasets may amount to only three relevant observations.
3. **An average can hide the failures we care about most.** Mean accuracy or return may conceal rare catastrophes, poorly served subgroups, or large variation between runs.
4. **An in-distribution test is not a robustness test.** After deployment, the weather, users, sensors, incentives, or data-collection policy may change.
5. **Nearby model variants are correlated evidence.** Ten related architectures implemented in the same codebase are not ten independent confirmations of the underlying idea.
6. **The reported metric may only be a surrogate.** More accurate travel-time predictions do not necessarily produce better routes, just as a better one-step reward does not necessarily produce better long-term behavior.
7. **A deployed policy changes the data it sees.** This is especially important in RL. A new policy visits new states, and its errors can compound precisely in parts of the state space where the logged data are sparse.

5.2 What the statistics actually tells us

Let us make one part of the issue precise. Suppose $Z_1, \dots, Z_n$ are independent samples from the deployment distribution P. We fix a predictor f, assume that its loss satisfies $\ell(f; Z) \in [0, 1]$, and define its true and empirical risks by

$$R_P(f) = \mathbb{E}_{Z \sim P}[\ell(f; Z)], \qquad \widehat{R}_n(f) = \frac{1}{n} \sum_{i=1}^{n} \ell(f; Z_i). \tag{5.1}$$

Hoeffding's inequality gives

$$\mathbb{P}\Big(|R_P(f) - \widehat{R}_n(f)| > \epsilon\Big) \leq 2 \exp(-2n\epsilon^2). \tag{5.2}$$

The message is reassuring: if the test cases are genuinely independent, more of them reduce our sampling uncertainty. But notice the fine print. The predictor must already be fixed, the cases must be independent, and the test cases and deployment cases must come from the same distribution P. In real ML studies, each of these conditions deserves scrutiny.

Now imagine that we compare K models on the same test data and report the best one. We are no longer evaluating just one fixed predictor. For a fixed finite set of K models, a union bound gives, with probability at least $1 - \delta$,

$$\max_{f \in \mathcal{F}} |R_P(f) - \widehat{R}_n(f)| \leq \sqrt{\frac{\log(2K/\delta)}{2n}}, \qquad |\mathcal{F}| = K. \tag{5.3}$$

The $\log K$ term is the statistical price of making the comparison. The bound is still manageable when the candidate set is fixed and recorded in advance. It may no longer apply when we repeatedly inspect benchmark results, perform unrecorded tuning, or search through an effectively unbounded collection of designs.

The unit of generalization matters too. Suppose our test cases come from m cities. Road segments within one city share infrastructure, weather, and policy, so they are not independent evidence for a claim about an unseen city. Even if we observe millions of segments, the effective sample size for that claim may be closer to m. "Many examples" and "many environments" are therefore not interchangeable.

There is one more boundary to keep in mind. An in-distribution concentration bound says nothing by itself about an arbitrary shift from P to Q. Two possible worlds can agree on every case we observed and behave differently outside that support. To make a robustness claim, we need assumptions connecting P and Q, experiments that deliberately vary the environment, or a narrower claim.

5.3 Return to the route guarantee

Let us return to the route-planning example. Equation (3.6) tells us more than "the predictor worked well on average," but its guarantee has conditions. We need a uniform error bound for every edge the optimizer might use, and we need a bound on path length. The theorem handles the logical step from those assumptions to route quality; our evidence must tell us whether the assumptions are plausible in deployment.

This gives us a useful recipe:

$$\boxed{\text{component guarantee}} + \boxed{\text{system structure}} \Longrightarrow \boxed{\text{end-to-end guarantee}}. \tag{5.4}$$

In other words, first understand the learned component, then use the structure of the surrounding system to translate that understanding into an end-to-end claim. This usually tells us much more than reporting a component score without asking how the component will actually be used.

5.4 Match the evidence to the claim

So how much evidence is enough? That depends on what we want to claim. A claim about passing a particular test suite needs less evidence than a claim about generalization, safety, or reliable deployment. Table 3 shows how the evidence should grow with the scope of the claim.

Table 3: An evidence ladder: broader claims require additional evidence.

Intended claim	Minimum starting point	Important additions
Runs correctly on given cases	Unit, integration, and regression tests	Edge cases, determinism checks, static analysis
Improves a named benchmark	Untouched test split and strong baselines	Multiple seeds, uncertainty intervals, effect sizes, compute matching
Improves because of a proposed idea	Controlled comparison	Ablations, sensitivity analysis, negative results, alternative mechanisms
Generalizes across environments	Multiple independently sampled domains	Leave-one-domain-out tests, meaningful shifts, hierarchical uncertainty
Is robust or safe in a region	Explicit threat and operating model	Stress tests, tail metrics, formal certificates where possible, red teams
Performs well after deployment	End-to-end task metrics	Monitoring, shift detection, rollback rules, incident analysis

A stricter standard does not mean that every neural network needs a universal theorem. It means that the breadth and stakes of our claim should determine the strength of our evidence. In practice, we should:

- say clearly which population and operating conditions the claim covers, and which failures matter;
- keep a genuinely untouched evaluation for confirmation;
- sample meaningful sources of variation rather than relying only on convenient benchmarks;
- report uncertainty, effect sizes, random seeds, compute, and enough implementation detail to interpret the comparison;
- compare against strong baselines and use ablations, calibration checks, tail metrics, and failure analysis to understand the result;
- evaluate the downstream decision objective and constraints, not only a convenient component-level metric;
- derive formal or compositional guarantees when their assumptions are defensible, and then test whether those assumptions hold;
- make data, code, protocols, and exclusions reproducible when possible;

- continue monitoring after deployment, because the environment can change after the evaluation is complete.

> **Key idea.** Keep both sides of the evidence story in view. Theory tells us what follows if its assumptions hold; experiments tell us what happened in the environments we sampled. The strongest assurance combines those two sources with continued monitoring after deployment.

6 So, what is reinforcement learning?

We are now ready to ask: where does reinforcement learning fit into this picture? At its heart, RL asks how an agent can learn from experience to choose actions that lead to good outcomes *over time*. The phrase "over time" is doing important work here. What defines RL is the *sequential decision structure*, not the use of a particular neural network or optimization algorithm.

6.1 Put the interaction into an MDP

To make that idea precise, we need a mathematical language for an agent, its environment, and their interaction. A discounted Markov decision process (MDP) is a tuple

$$\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, r, \rho_0, \gamma), \tag{6.1}$$

where $\mathcal{S}$ is the set of possible states and $\mathcal{A}$ is the set of possible actions. The transition model $P(s' \mid s, a)$ tells us how the environment may move from state s to state s' after action a, while $r(s, a)$ gives the expected immediate reward. Finally, ρ_0 describes where an episode begins, and $\gamma \in [0, 1)$ determines how strongly we weight rewards that arrive later.

Now picture one step of interaction. At time t, the agent observes S_t, chooses $A_t \sim \pi(\cdot \mid S_t)$, receives reward R_{t+1}, and moves to S_{t+1}. The policy π is simply the agent's decision rule. A standard goal is to find a policy with a large expected discounted return:

$$J(\pi) = \mathbb{E}_{\tau \sim p^\pi}\left[\sum_{t=0}^{\infty} \gamma^t R_{t+1}\right]. \tag{6.2}$$

In plain language, we add up the rewards along a trajectory, give later rewards a weight of γ^t, and average over trajectories generated by the policy. The superscript in p^π is worth noticing: when the agent changes its policy, it also changes the states it visits and therefore the data it will see.

There is one caveat. The MDP model assumes that the current state contains the information needed to predict what happens next. If the agent sees only part of that state, its decision rule may instead depend on a history of observations or on a learned memory state.

6.2 What makes RL different

At first glance, this may sound like ordinary prediction with a reward as the target. The crucial difference is that the learner sits inside a feedback loop: its actions help determine both its future outcomes and its future evidence. That creates several distinctive challenges.

1. **Feedback is evaluative rather than instructive.** A reward tells the agent how well things went, but it usually does not reveal which action would have been correct.

2. **Consequences can be delayed.** An action taken now may affect rewards much later, so the agent must work out which earlier decisions deserve credit or blame.

3. **The policy shapes the data.** Actions change later states, so the current policy determines which parts of the environment the agent gets to observe.

4. **Exploration competes with exploitation.** The agent may have to try an uncertain action to learn whether it is valuable, even when a familiar action currently looks safer.

5. **Evaluation is counterfactual.** Logged data reveal what happened after the actions that were actually taken, not what would have happened under every alternative policy.

6. **Errors can compound.** One poor decision may send the agent into an unfamiliar state, where further mistakes become more likely.

The neighboring formulations are easiest to compare side by side.

Table 4: How RL differs from neighboring problem formulations.

Setting	Feedback	Does action affect future data?	Central question
Supervised prediction	Target label or value	Usually no; the dataset is treated as fixed	How accurately can a target be predicted?
Imitation learning	Demonstrated action	Yes at deployment, even if training data are fixed	How can demonstrated behavior be reproduced without drifting?
Contextual bandit	Reward for selected action	No long-term state consequence in the formulation	Which action has highest expected one-step reward?
Planning with a known model	Specified dynamics and objective	Yes, but the main task is computation rather than learning the model	Which policy optimizes the known model?
Reinforcement learning	Rewards along trajectories	Yes; interaction or logged policy determines coverage	Which policy maximizes long-term expected return?

Our route-planning example makes these distinctions concrete. The same application can fall into different paradigms depending on what is known, what feedback is available, and whether today's action changes tomorrow's decision problem:

- computing a shortest path on a known graph is **planning**;
- predicting edge travel time from recorded pairs is **supervised learning**;
- copying expert turns is **imitation learning**;
- selecting one complete route before a trip can be a **contextual bandit** abstraction if the choice has no modeled future state effect;
- learning adaptive rerouting decisions with delayed trip cost is **reinforcement learning**.

So the label "RL" comes from the structure of the problem, not from one particular implementation. An RL method may be model-based or model-free, online or offline, and tabular or approximate. Offline RL does not explore while it is being trained, but the logged data were still produced by earlier action choices, and judging unseen actions remains a counterfactual problem. Likewise,

deep RL simply means that deep networks are used as function approximators; it is not a different definition of reinforcement learning (Sutton and Barto, 2018).

7 How reinforcement learning got here

How did these ideas come together? Reinforcement learning does not have a single birth date. It grew out of several traditions that were initially quite separate: trial-and-error psychology, dynamic programming and optimal control, multi-armed bandits, and animal-inspired learning systems. The milestones below are selective, but they show how those threads gradually became the field we now call RL.

1950s: foundations

Two foundational questions came into focus. First, how should a learner balance trying uncertain choices with using what it already knows? The bandit work of Robbins (1952) formalized this exploration problem through sequential experimental design. Second, how can a long decision problem be broken into smaller ones? Bellman's dynamic programming and principle of optimality supplied the recursive answer (Bellman, 1957). Dynamic programming assumes that a model is already available, so it is not by itself a learning method. Its value functions and Bellman equations, however, became central to RL.

1959: learning through play

Samuel's checkers program offered an early glimpse of what learning through experience could look like in practice. It combined experience, search, and value estimation, and used self-play to improve its game (Samuel, 1959).

1980s: modern learning mechanisms

The next step was to learn useful predictions while acting. Actor–critic ideas appeared in adaptive control systems (Barto et al., 1983), separating a component that evaluates behavior from one that improves it. Temporal-difference learning then showed how to update predictions by bootstrapping from later estimates, without waiting until the final outcome (Sutton, 1988).

Late 1980s–1990s: core algorithms and theory

Several algorithms that still shape the field arrived during this period. Watkins introduced Q-learning in his 1989 thesis, and the later result with Dayan proved convergence of the tabular algorithm under stated sampling conditions (Watkins, 1989; Watkins and Dayan, 1992). REINFORCE gave a general likelihood-ratio estimator for policy gradients (Williams, 1992). TD-Gammon then showed that temporal-difference learning and self-play could produce remarkably strong backgammon play (Tesauro, 1995).

2013/2015: deep reinforcement learning

Deep Q-Networks brought several existing ingredients together: Q-learning, convolutional networks, experience replay, and a target network. The result was an agent that learned Atari control directly from pixels. A preliminary result appeared in 2013, followed by the widely cited journal article in 2015 (Mnih et al., 2015).

2016–2018: learning plus search at scale

AlphaGo combined supervised learning, reinforcement learning, policy and value net-

works, and Monte Carlo tree search (Silver et al., 2016). AlphaZero then removed human game demonstrations and learned chess, shogi, and Go through self-play plus search (Silver et al., 2018). It is useful to pause over this point: these were not "pure RL" systems. Their strength came from combining learning, planning, representation, and large-scale computation.

Late 2010s onward: broader regimes
The field expanded beyond online benchmark control. Stable policy optimization, scalable model-based methods, multi-agent RL, offline RL, and learning from human preferences all became prominent. Representative examples include PPO (Schulman et al., 2017), modern offline-RL formulations (Levine et al., 2020), and preference-based deep RL (Christiano et al., 2017). Along the way, the questions also broadened: not only "How high is the return?" but also "How much data did it take?", "How robust and safe is it?", "Can others reproduce it?", and "Does the objective reflect what people actually want?"

If we step back from the timeline, three themes keep reappearing:

1. **Prediction and control support each other.** Value functions predict future return so that the agent can make better decisions.
2. **Learning and planning work well together.** Many landmark systems combine learned estimates with search or dynamic programming; we do not have to choose one or the other.
3. **Scale changes capability, not the evidence problem.** Larger models and broader benchmarks can produce striking results, but they make careful evaluation more important, not less.

8 Let's pull the ideas together

Take-away messages

Let us pull the main thread together. If you remember only a few ideas from this lecture, make them these:

1. Start with the problem, not the algorithm. Conventional problem solving moves from specification to model, structure, algorithm, and verification—and validating the model remains a separate obligation.
2. Machine learning can fill in a missing part of a model, or it can learn a solution map that stands in for an algorithm. Either way, assumptions do not disappear; they move and often multiply.
3. Hybrid designs are often especially useful: we can learn the uncertain component from data while preserving known constraints, structure, and proofs.
4. Results from a few model variants on a few datasets justify only a narrow empirical claim. Claims about generalization, robustness, or safety need evidence across the relevant sources of variation and, when possible, theory connecting component error to system behavior.
5. Reinforcement learning is about sequential choice under evaluative feedback. Actions affect not only future outcomes, but also the future data available to the learner.

6. Modern RL is best understood as a meeting point for ideas from dynamic programming, trial-and-error learning, bandits, function approximation, search, and statistical learning.

Try these questions

The following exercises ask you to reuse the lecture's ideas in settings of your own. For each one, make your assumptions explicit; that is part of the answer.

1. **Reconstruct the pipeline.** Pick a familiar algorithm from outside route planning. What real problem is it meant to solve? Write down its mathematical model, the structure it exploits, its main invariant, and its correctness claim. Then identify one gap between the model and the real setting.

2. **Prediction versus decision.** Build a graph with two s-to-t routes in which predictor A has lower average edge error than predictor B, yet A produces a route with higher true cost. What does your example teach us about choosing a metric that matches the deployment goal?

3. **Audit a generalization claim.** Suppose a paper evaluates four neural architectures, five random seeds, and three datasets. For claims about random initialization, examples within a dataset, and entirely new application domains, what are the independent units of evidence? Sketch a confirmatory experiment that matches the broadest claim.

4. **Classify the problem.** Revisit each routing formulation in Section 6. Explain why it belongs to supervised learning, imitation learning, a contextual bandit, planning, or RL. For each classification, name one modeling assumption whose change could move the problem into a different category.

5. **Design a stricter standard.** Imagine that an adaptive routing agent will be deployed across several cities. Design an evidence plan: which success metrics, distribution shifts, tail risks, baselines, uncertainty estimates, safety constraints, and monitoring procedures would you require before and after deployment?

References

Rishabh Agarwal, Max Schwarzer, Pablo Samuel Castro, Aaron Courville, and Marc G. Bellemare. Deep reinforcement learning at the edge of the statistical precipice. In *Advances in Neural Information Processing Systems*, volume 34, pages 29304–29320, 2021.

Andrew G. Barto, Richard S. Sutton, and Charles W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(5):834–846, 1983. doi: 10.1109/TSMC.1983.6313077.

Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.

Paul F. Christiano, Jan Leike, Tom B. Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. In *Advances in Neural Information Processing Systems*, volume 30, 2017.

Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018. doi: 10.1609/aaai.v32i1.11694.

Sergey Levine, Aviral Kumar, George Tucker, and Justin Fu. Offline reinforcement learning: Tutorial, review, and perspectives on open problems. *arXiv preprint arXiv:2005.01643*, 2020. URL https://arxiv.org/abs/2005.01643.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, 2015. doi: 10.1038/nature14236.

Herbert Robbins. Some aspects of the sequential design of experiments. *Bulletin of the American Mathematical Society*, 58(5):527–535, 1952. doi: 10.1090/S0002-9904-1952-09620-8.

Arthur L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229, 1959. doi: 10.1147/rd.33.0210.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017. URL https://arxiv.org/abs/1707.06347.

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529:484–489, 2016. doi: 10.1038/nature16961.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419): 1140–1144, 2018. doi: 10.1126/science.aar6404.

Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988. doi: 10.1007/BF00115009.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2 edition, 2018. URL http://incompleteideas.net/book/the-book-2nd.html.

Gerald Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3):58–68, 1995. doi: 10.1145/203330.203343.

Christopher J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, University of Cambridge, 1989.

Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8:279–292, 1992. doi: 10.1007/BF00992698.

Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256, 1992. doi: 10.1007/BF00992696.