What we covered so far:

- Deterministic Dynamic Programming
  - Shortest Path Problem
  - HMM models
- Stochastic Dynamic Programming
  - LQR
  - MDP’s
- Approximate Dynamic Programming (Reinforcement Learning)
  - Value Iteration and Policy Iteration
  - “Model-free” methods(DQN, Policy Gradient, Actor-Critic)
  - Monte-Carlo Tree Search framework (e.g.: AlphaGo)
- Approximate Dynamic Programming (Model Predictive Control)
  - Model-based methods(Linear MPC, Robust MPC)
  - Learning-based MPC (LBMPC)

Goal of Dynamic Programming

- In all the methods we discussed so far, our main goal was to make good decisions in a dynamic environment:

- Hence our focus was on the **planning** and **policy** execution on the face of an environment that can be uncertain and complex.

- And the underlying fundamental principle is the **<u>Bellman's Principle</u>**

DP formulation

- The Bellman's Principle allow us the "break" the dynamic problem into stages and write a recursion where the "current" problem is given as a function of the "tail" problem.

- That is of course, the Bellman Recursion, which we have written so many times by now:

$$J_N(x_n) = g_N(x_N)$$

$$J_i(x_i) = \min_{u_i \in U_i(x_i)} \left\{ \mathbb{E}_{w_i} \left[g_i(x_i, u_i, w_i) + J_{i+1}(f_i(x_i, u_i, w_i)) \right] \right\}, \forall i \in \{0, ..., N-1\}$$

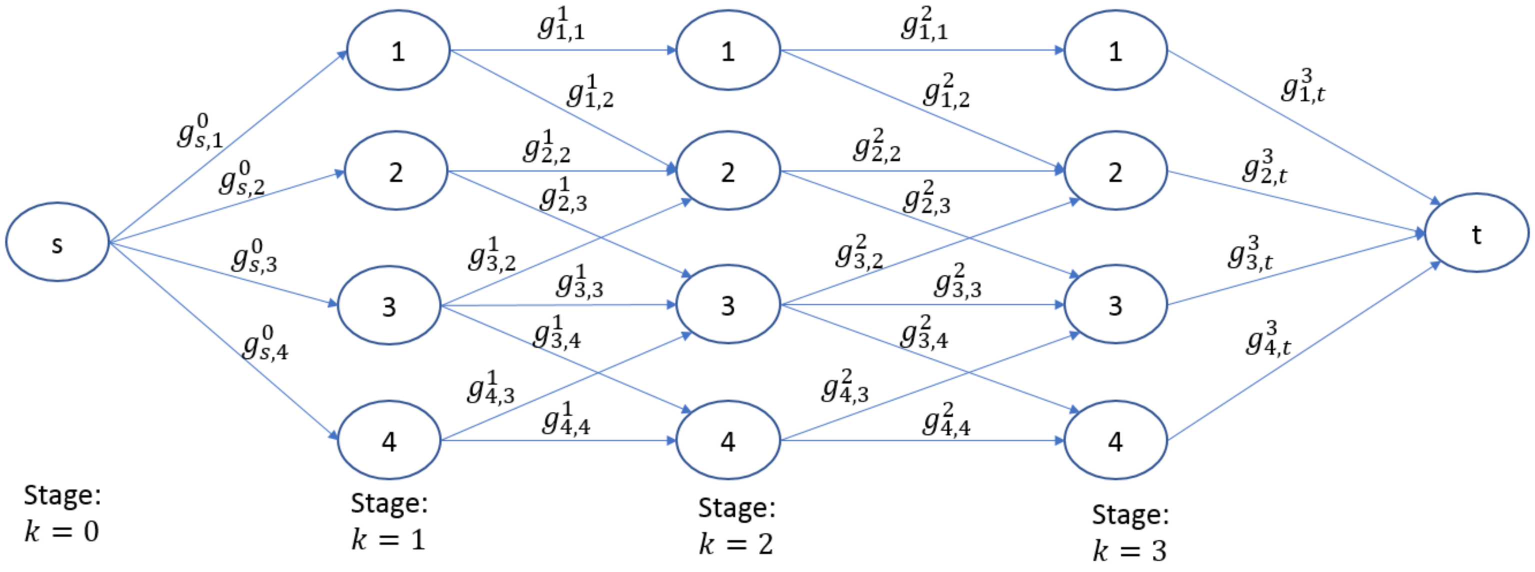
- And when attempting to solve those equations backwards, several problems arise:
  - (1) How can we compute expectations?
  - (2) How can we solve the optimization problem?
  - (3) How do we handle the curse of dimensionality?

Deterministic DP vs Stochastic DP

- Our first attempt to solve the DP problem was to consider the case where there is no uncertainty on the dynamics:

$$x_{k+1} = f(x_k, u_k)$$

- And if the state-space was discrete, then we can formulate the DP as a Shortest-Path Problem:



Deterministic DP as Shortest Path

- The nice property about deterministic DP is that the optimal solution of the problem can be found by only searching for **sequences** of inputs:

$$(u_0, u_1, ..., u_{N-1})$$

- And we can use a forward DP algorithm to solve the Shortest Path problem.

- The **planning** on deterministic environments can be computed going forward in time.

- The forward DP algorithm receives different names depending on the applications:
  - (1) Dijkstra's Algorithm for shortest path problems on graphs
  - (2) Viterbi Algorithm for most-likely sequence estimation in decoding problems
  - (3) etc.

Stochastic DP and optimal policies

- Once we moved into the stochastic environment, we need to search for optimal **policies:**

$$(\mu_0(x_0), \mu_1(x_1), \dots, \mu_{N-1}(x_{N-1}))$$

- So a policy is a sequence of functions, where at each stage the function $\mu_k(x_k)$ specify our control/action, given that we are in state x_k.

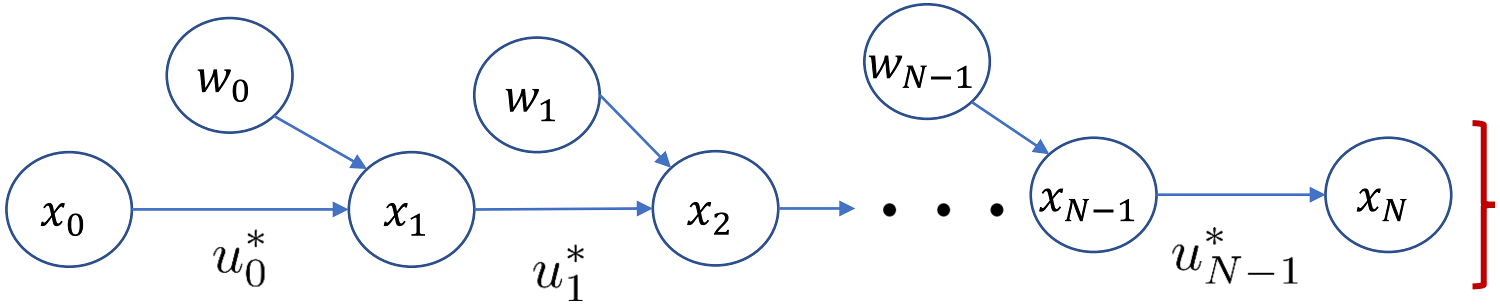
- This is the key distinction, between stochastic and deterministic DP

- It can be illustrated by the difference between **open-loop** and **closed-loop**.

Open-Loop vs Closed-Loop

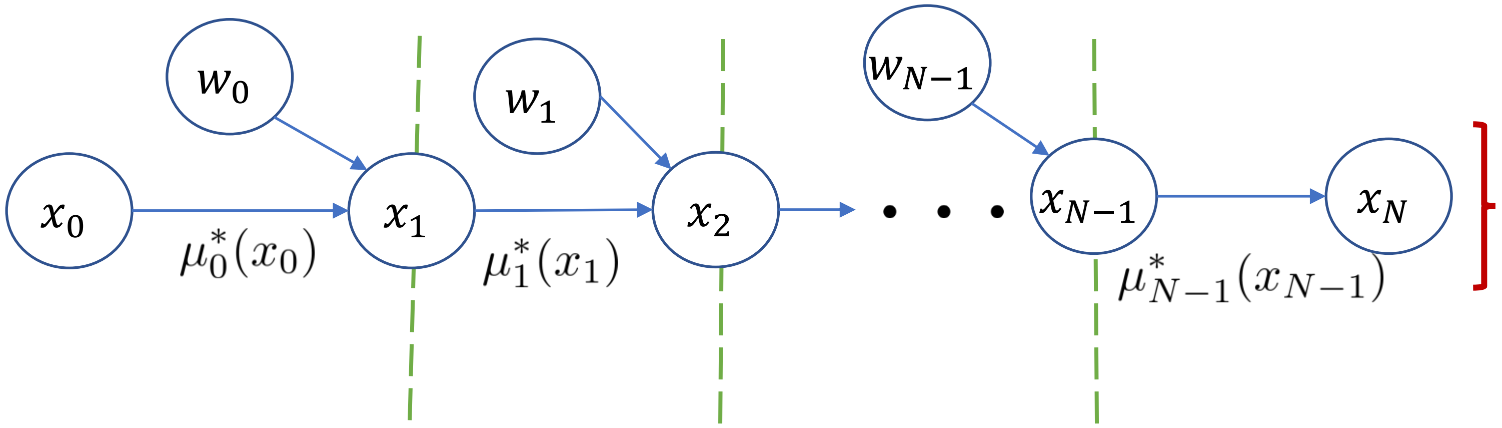
- The difference lies in the **timing** of decisions and when the uncertainty is resolved:

Open-Loop:



You "commit" on decisions at the start

Closed-Loop:



You time your choices after each uncertainty is "revealed"

Approximate Dynamic Programming

- Finding the optimal policies in Stochastic DP is **very hard**, in general.

- Even in deterministic cases is hard (see the Travelling Salesman).

- The abstract Backward DP Algorithm provides with the solution. But a computer (or us) cannot execute this algorithm in a reasonable amount of time.

- It is then that we introduce approximations with a clear goal:
  - (1) Reduce the amount of computation required (the curse of dimensionality)
  - (2) Provide a way to handle uncertainty (simulation and sampling)

Multistep Lookahead and Rollout

- The general approximation methods are the lookahead minimization:

$$\tilde{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} \left\{ \mathbb{E}_{w_k} \left[g_k(x_k, u_k, w_k) + \tilde{J}_{k+1}(f_k(x_k, u_k, w_k)) \right] \right\}, \forall k \in \{0, \ldots, N-1\}$$

- Where $\tilde{J}_{k+1}(x_{k+1})$ is an approximate cost-to-go function of l-step DP problem:

$$\tilde{J}_{k+1}(x_{k+1}) = \min_{(\mu_{k+1}, \ldots, \mu_{k+l-1})} \mathbb{E}_{w_{k+1}, \ldots, w_{k+l-1}} \left[\tilde{J}_{k+l}(x_{k+l}) + \sum_{i=k+1}^{k+l-1} g_i(x_i, \mu_i(x_i), w_i) \right]$$

- And we summarize it as:

First l-steps

"Future"

$$\min_{u_k, \mu_{k+1}, \ldots, \mu_{k+l-1}} \mathbb{E}_{w_{k+1}, \ldots, w_{k+l-1}} \left[g_k(x_k, u_k, w_k) + \sum_{i=k+1}^{k+l-1} g_i(x_i, \mu_i(x_i), w_i) + \tilde{J}_{k+l}(x_{k+l}) \right]$$

DP Minimization

Lookahead minimization

Cost-to-go Approximation

Multistep Lookahead and Rollout

- In the **Rollout Algorithm** the lookahead problem is still the same:

$$\tilde{\mu}_k(x_k) \in \arg \min_{u_k \in U_k(x_k)} \left\{ \mathbb{E}_{w_k} \left[g_k(x_k, u_k, w_k) + \tilde{J}_{k+1}(f_k(x_k, u_k, w_k)) \right] \right\}, \forall k \in \{0, \ldots, N-1\}$$

- Where the cost-to-go approximation $\tilde{J}_{k+1}(x_{k+1})$ is as the total cost of some *Base Policy* $\hat{\pi} = (\hat{\mu}_{k+1}, \ldots, \hat{\mu}_{N-1})$:

$$x_{i+1} = f_i(x_i, \hat{\mu}_i(x_i), w_i), \ \forall i = k+1, \ldots, N-1$$

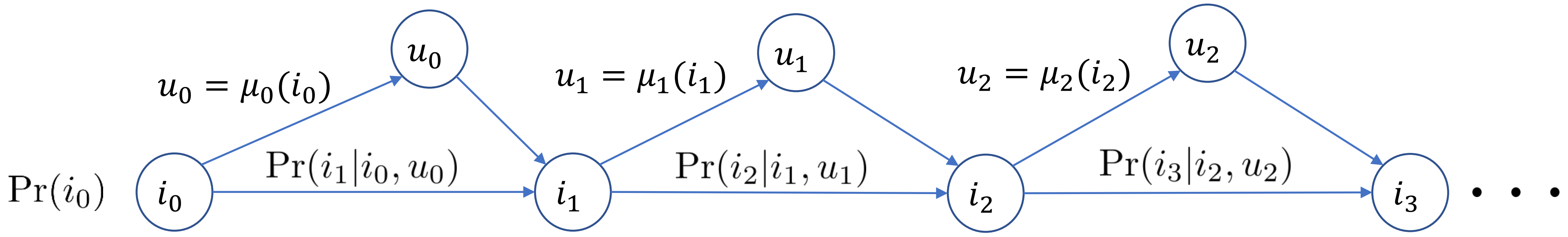
- For some simulated disturbances sequences $(w_k, \ldots, w_{N-1})$.

- Many of the Algorithms we saw are actually variations of the Rollout Algorithm with (multistep) lookahead minimization

Value Iteration and Policy Iteration

- The two cornerstones of Approximate DP are the Value Iteration and Policy Iteration.

- We studied those in the context of Markov Decision Problems (MDP's)
  - Dynamic Programs with discrete state space



- And we consider problems with infinite-horizon
  - Or problems where the horizon is very large (mas as well be infinite).

Value Iteration and Policy Iteration

Value Iteration

(VI-step) $$J^{(t+1)}(i) = \min_{u \in U(i)} \left\{ \sum_{j}^{n} p_{ij}(u)(g(i,u,j) + \alpha J^{(t)}(j)) \right\}$$

- We keep updating the value function (cost-to-go)
- The policy is obtained afterwards:

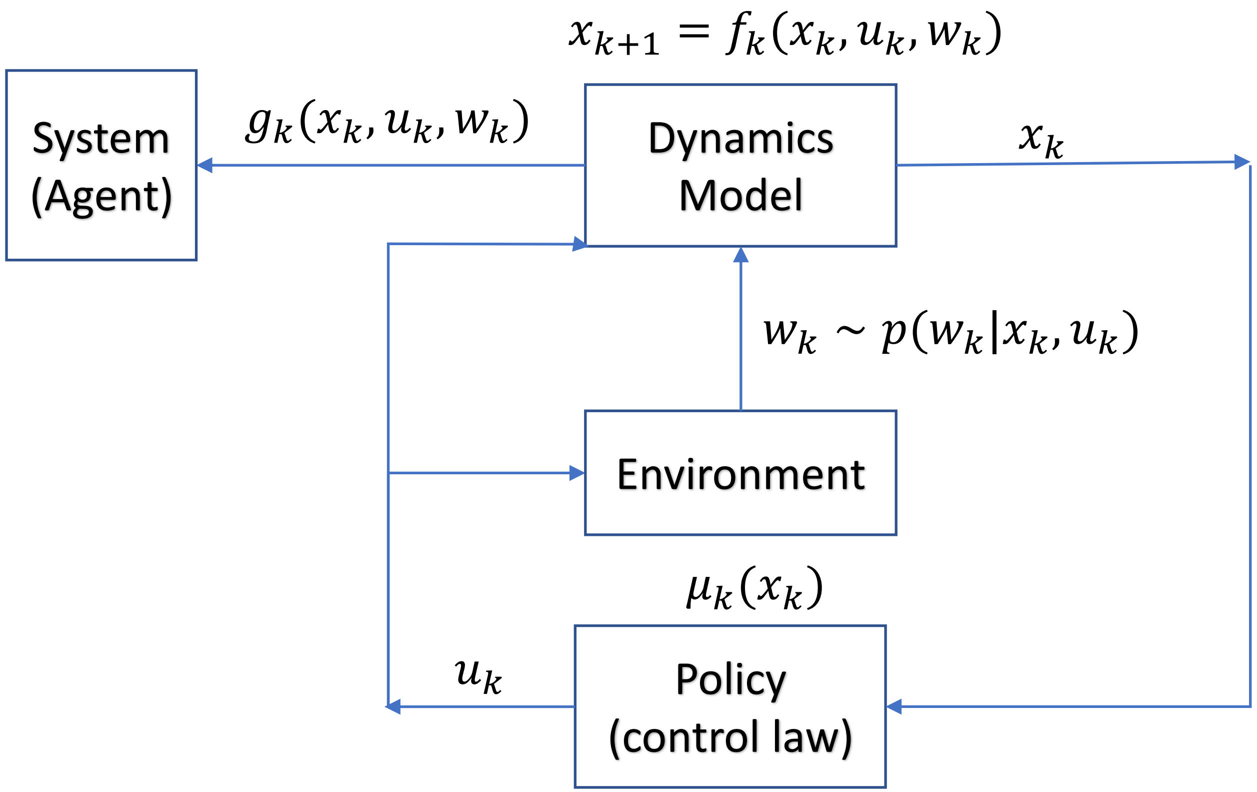
Policy Iteration

(Policy Evaluation) $$J_{\mu^{(t)}}(i) = \sum_{j=1}^{n} p_{ij}(\mu^{(t)}(i))\big(g(i,\mu^{(t)}(i),j) + \alpha J_{\mu^{(t)}}(j)\big)$$

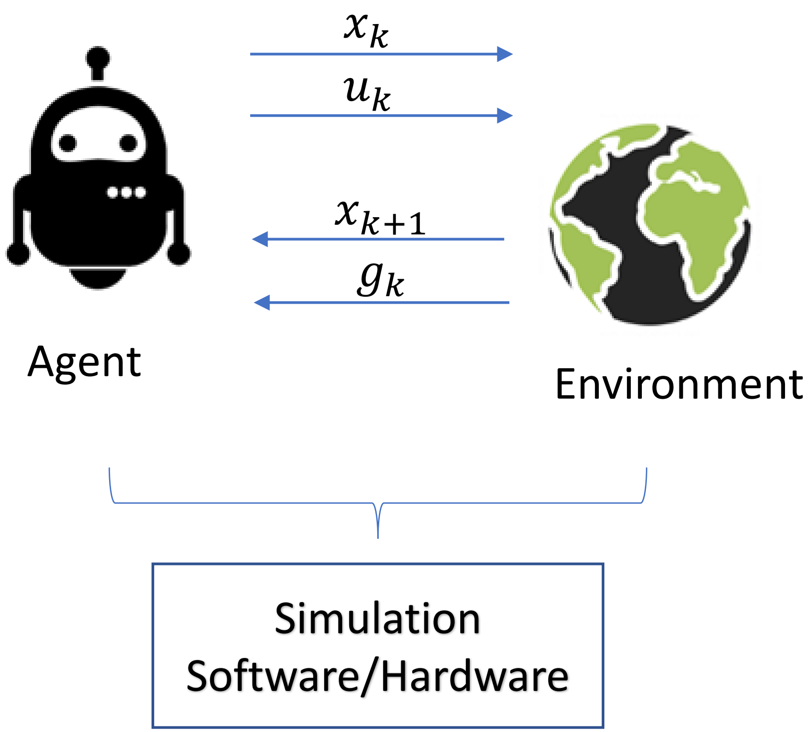
(Policy Improvement) $$\mu^{(t+1)}(i) \in \arg\min_{u \in U(i)} \left\{ \sum_{j=1}^{n} p_{ij}(u)\big(g(i,u,j) + \alpha J_{\mu^{(t)}}(j)\big) \right\}, \forall i \in \{1, ..., n\}$$

- We keep updating the policy
- The value-function is a by-product

Model-free vs Model-based Methods



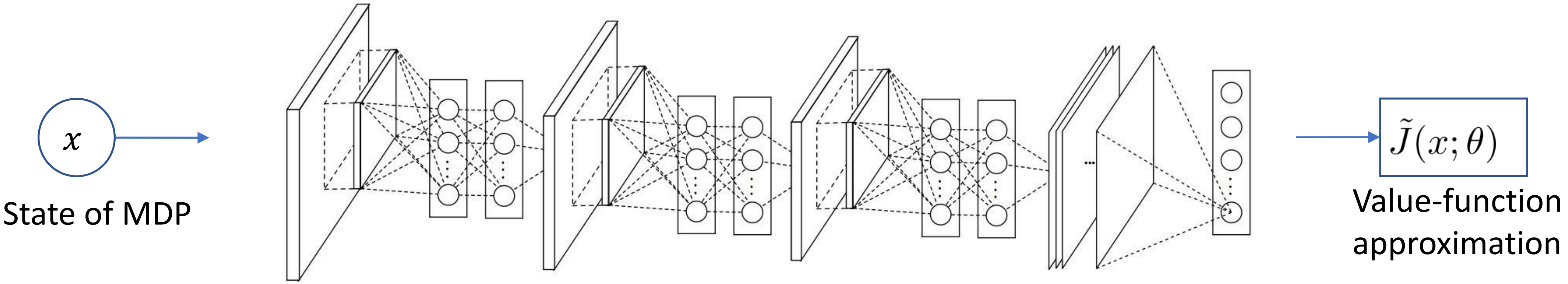
Model-based Case



Model-free Case

Model-free Methods

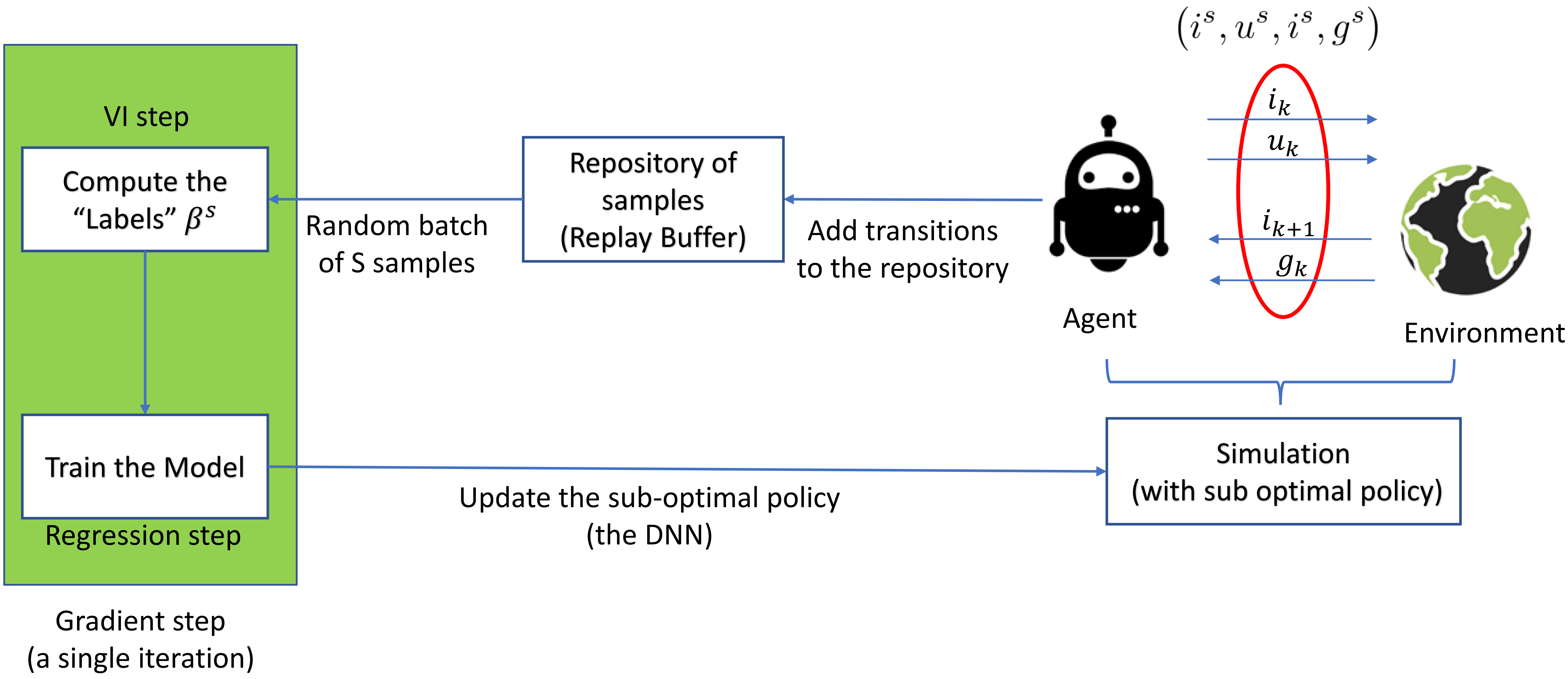
- Model-free methods rely heavily on **simulation** and **approximation architectures**
- We studied Approximated Value Iteration, using DNN's:



- So the DNN's maps the state of MPD to the associated cost-to-go function approximation.
- That led us to the Fitted VI Algorithm and the Deep Q-Learning Algorithm.

Deep Q-Learning Algorithm

- So we can summarize the Deep Q-learning algorithm by the following diagram:



Approximate PI and the Policy Gradient

- On the Policy Iteration side of things, we studied the Policy Gradient, an “actor-only” method:

$$\min_{\theta} \mathbb{E}_{p(z|\theta)} \Big[\sum_{k=0}^{\infty} \alpha^k g(i_k, u_k) \Big] = \mathbb{E}_{p(z|\theta)} \big[F(z) \big]$$

$$\theta^{(t+1)} = \theta^{(t)} - \gamma^{(t)} \nabla_\theta \Big(\mathbb{E}_{p(z|\theta)} \big[F(z) \big] \Big), \quad \forall t \geq 0$$

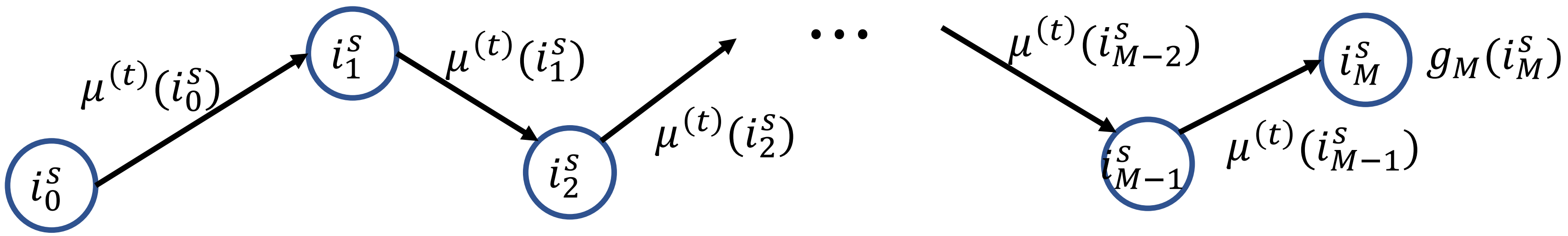
- Where we saw the Policy Gradient as Weithed Maximum-Likelihood:

$$\nabla_\theta \Big(\mathbb{E}_{p(z|\theta)} \big[F(z) \big] \Big) \approx \frac{1}{S} \sum_{s=1}^{S} \underbrace{\Bigg(\sum_{k=0}^{M-1} \nabla_\theta \Big(\ln(p(u_k | i_k^s, \theta)) \Big) \Bigg)}_{\boxed{\text{Maximum-Likelihood}}} \underbrace{\Bigg(\sum_{k=0}^{M-1} \alpha^k g(i_k^s, u_k^s) + \alpha^M \hat{J}_M(i_m^s) \Bigg)}_{\boxed{\text{Costs “weights”}}}$$

Approximate PI and the Critic

- We introduced the Critic to approximate the Policy Evaluation step of PI.
- This is done in the typical "supervised learning" fashion:

$$\theta^{(t)} = \arg\min_{\theta} \left\{ \sum_{s=1}^{S} \left(\tilde{J}_{\mu^{(t)}}(i_0^s, \theta) - \beta^s \right)^2 \right\}$$

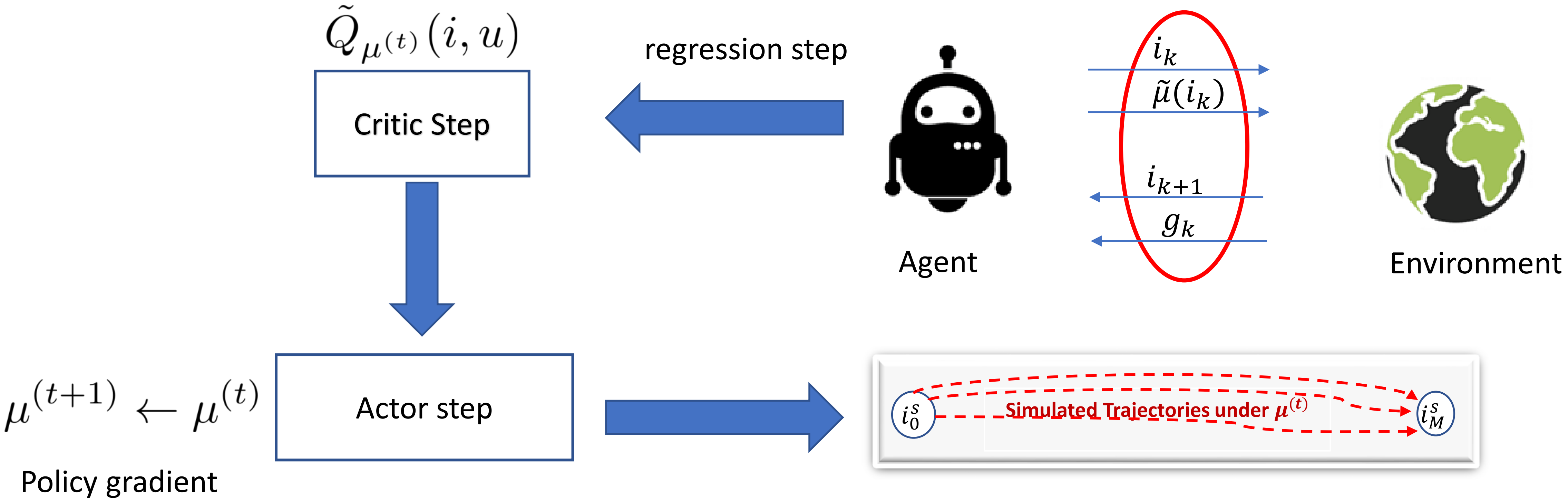


$$\beta^s = \sum_{k=0}^{M-1} \alpha^k g(i_k^s, \mu^{(t)}(i_k^s), i_{k+1}^s) + \alpha^M \hat{J}(i_M^s)$$

Terminal cost function approximation

The Actor-Critic Algorithm

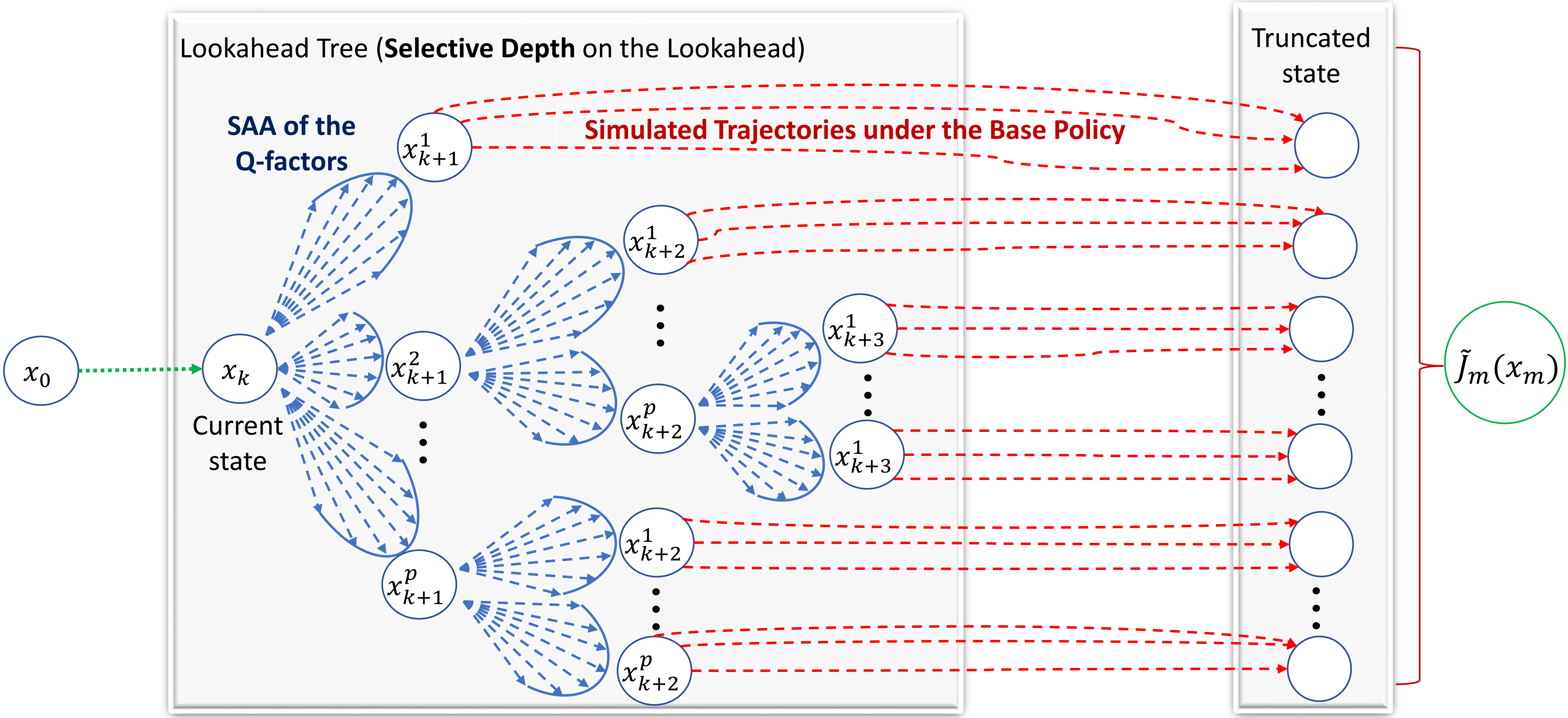
- We finished our study of model-free methods with the Actor-Critic Method, which implements Approximate Policy Iteration, for both evaluation and improvements step:



The Monte-Carlo Tree Search

- The model-free methods studied require a lot of **simulation** and **sampling.**
- Moreover than can be combined with Multi-step Lookahead and Rollout Algorithm.
- The framework which we can combine them is the Monte-Carlo Tree Search (MCTS).
- The MCTS provide **selective-depth** lookahead with dynamic pruning of the search-tree.
- Many success cases (we studied in depth AlphaGo) implement model-free methods combined with MCTS.

Monte-Carlo Tree Search (MCTS)



Model-based Methods

- The cornerstone algorithm of Model-based methods is the Model Predictive Control Algorithm (MPC).

- The MPC Algorithm is tailored to solve DP problems with continuous state space (e.g.: real-valued vector spaces):

$$J_N(x_n) = g_N(x_N)$$

$$J_i(x_i) = \min_{u_i \in U_i(x_i)} \left\{ \mathbb{E}_{w_i} \left[g_i(x_i, u_i, w_i) + J_{i+1}(f_i(x_i, u_i, w_i)) \right] \right\}, \forall i \in \{0, ..., N-1\}$$

- And the constraints sets are given by:
  - Polyhedrons
  - Convex sets
  - Etc

LQR and Linear MPC

- The simplest case is the Linear Quadradic Regulator (LQR):

$$J_N(x_N) = x_N^\top Q_N x_N$$

$$J_k(x_k) = \min_{u_k} \left\{ \mathbb{E}_{w_k} \left[x_k^\top Q_k x_k + u_k^\top R_k u_k + J_{k+1}(A_k x_k + B u_k + w_k) \right] \right\}$$

- Where the optimal policy is linear:

$$\mu_k^*(x_k) = K_k x_k$$

Riccati Recursion

- And the optimal value function (cost-to-go) is quadratic:

$$J_0^*(x_0) = x_0^\top P_0 x_0 + \sum_{k=0}^{N-1} \mathbb{E}_{w_k} \left[w_k^\top P_{k+1} w_k \right]$$

Linear MPC (deterministic case)

- Solving this problem is hard as it has infinite horizon and constraints. The goal of MPC is to solve, instead, the following N-step lookahead problem:

Terminal Cost Approximation

$$J_0(\bar{x}_0) = \min_{X,U} \hat{J}_N(x_N) + \sum_{i=0}^{N-1} x_i^\top Q x_i + u_i^\top R u_i$$

$$\text{s.t. } x_{i+1} = Ax_i + Bu_i, \quad \forall i \in \{0,1,2,\ldots\}$$

Polytopes
$$\left[\begin{array}{ll} x_i \in \mathcal{X}, & \forall i \in \{0,1,2,\ldots\} \\ u_i \in \mathcal{U}, & \forall i \in \{0,1,2,\ldots\} \end{array}\right.$$

$$x_0 = \bar{x}_0$$

$$x_N \in \mathcal{X}_f$$

Terminal set constraint

MPC Properties

- The MPC Algorithm have two essential properties:
  - **(1) Recursive Feasibility**
  - **(2) Asymptotic Stability**

- Which we saw that hold under the following set of assumptions:

- Stage costs are positive definite: strictly positive and only zero at the origin

- The terminal set $\mathcal{X}_f$ is a **invariant set** under some local control policy $v(x_k)$:

$$x_{k+1} = f(x_k, v(x_k)) \in \mathcal{X}_f, \, \forall x_k \in \mathcal{X}_f$$

$$\mathcal{X}_f \subseteq \mathcal{X}, \, v(x_k) \in \mathcal{U}, \, \forall x_k \in \mathcal{X}_f$$

MPC Properties

- And the terminal cost approximation is **a Lyapunov Function** in the terminal set $\mathcal{X}_f$

$$\hat{J}_k(x_{k+1}) - \hat{J}_k(x_k) \leq -g(x_k, v(x_k)), \; \forall x_k \in \mathcal{X}_f$$

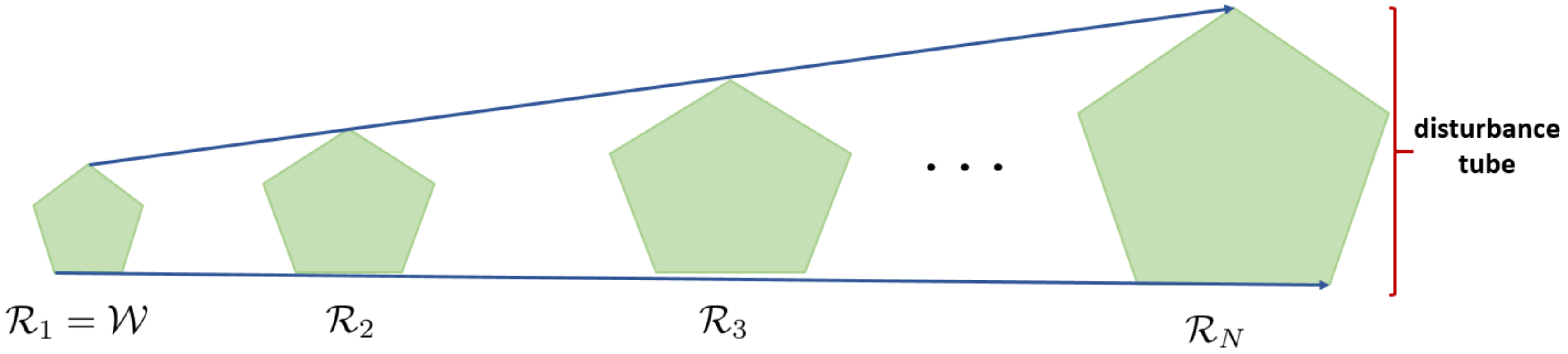
- Then, the MPC policy, which applied the first-stage control and discards the rest:

$$\mu_{\text{MPC}}(\bar{x}_0) = u_0^*$$

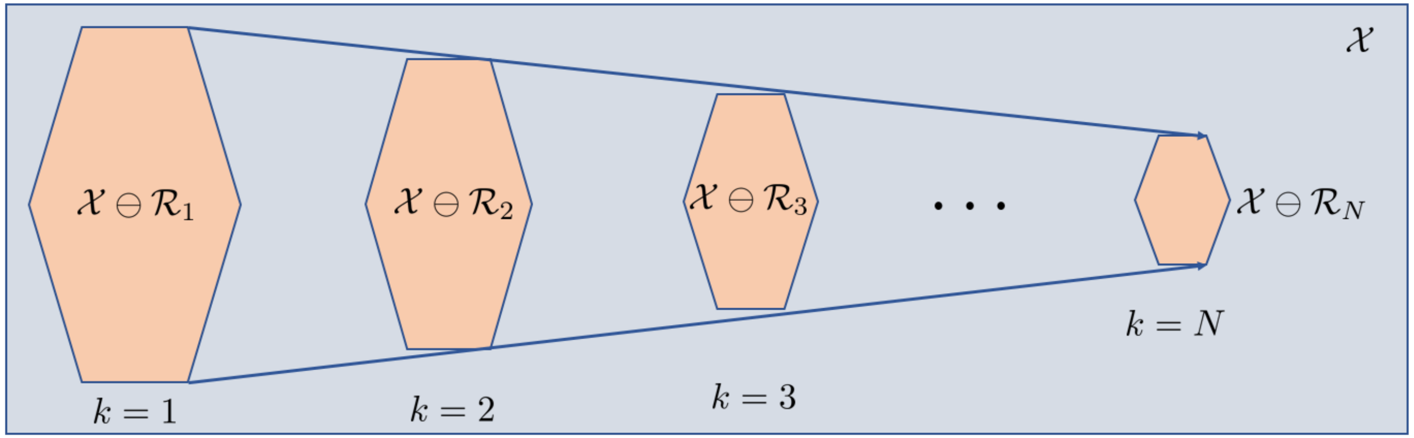
- Is **Recursive Feasible** and **Asymptotically Stable** with initial feasible set $\mathcal{X}_0$

Robust Linear MPC

- In order to handle disturbances (uncertainties) we study the concepts of **disturbances tubes**:



- And of **robust feasible region**:



Robust MPC

- We the Robust Optimal Control problem face by the MPC Algorithm is:

$$J_0(\bar{x}_0) = \min_{X,U,Z} \quad \hat{J}_N(\bar{x}_N) + \sum_{i=0}^{N-1} \bar{x}_i^\top Q \bar{x}_i + u_i^\top R u_i$$

Nominal stage cost

$$\text{s.t.} \quad \bar{x}_{i+1} = A\bar{x}_i + Bu_i, \qquad \forall i \in \{0, 1, 2, \ldots\}$$

Nominal dynamics

$$\bar{x}_i \in \mathcal{X} \ominus \mathcal{R}_i, \qquad \forall i \in \{1, 2, \ldots\}$$

$$u_i = K\bar{x}_i + z_i, \qquad \forall i \in \{0, 1, 2, \ldots\}$$

z_i is effectively the control decision

$$u_i \in \mathcal{U} \ominus (K \circ \mathcal{R}_k), \qquad \forall i \in \{0, 1, 2, \ldots\}$$

$$x_0 = \bar{x}_0$$

$$x_N \in \mathcal{X}_f \ominus \mathcal{R}_N$$

Robust Invariant Set

Constraint Robustification

Learning-Based Model Predictive Control (LBMPC)

- Lastly, we added a learning component to our MPC formulation:

Nominal Model

$$\bar{x}_{k+1} = A\bar{x}_k + Bu_k$$

- Enforce robust constraints.

- So, for any possible mismatch, the true system remains feasible:

$$x_{k+1} = A\bar{x}_k + Bu_k + w_k \in \mathcal{X}$$

Learned Model

$$\tilde{x}_{k+1} = A\tilde{x}_k + Bu_k + h(\tilde{x}_k, u_k)$$

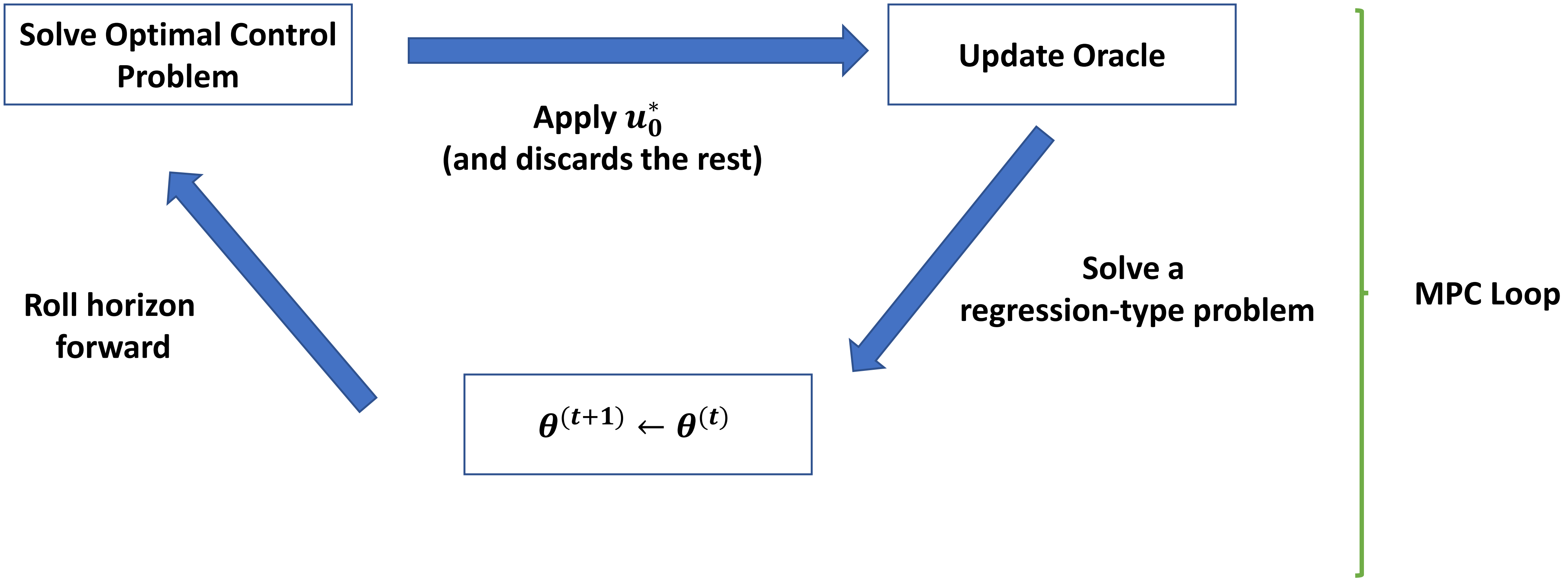
- No constraints enforced on the learned model.

- The objective function is based on the learned model:

$$\hat{J}_N(\tilde{x}_N) + \sum_{i=0}^{N-1} \tilde{x}_i^\top Q \tilde{x}_i + u_i^\top R u_i$$

Learning-Based Model Predictive Control (LBMPC)

- We can represent the LBMPC in the following scheme:



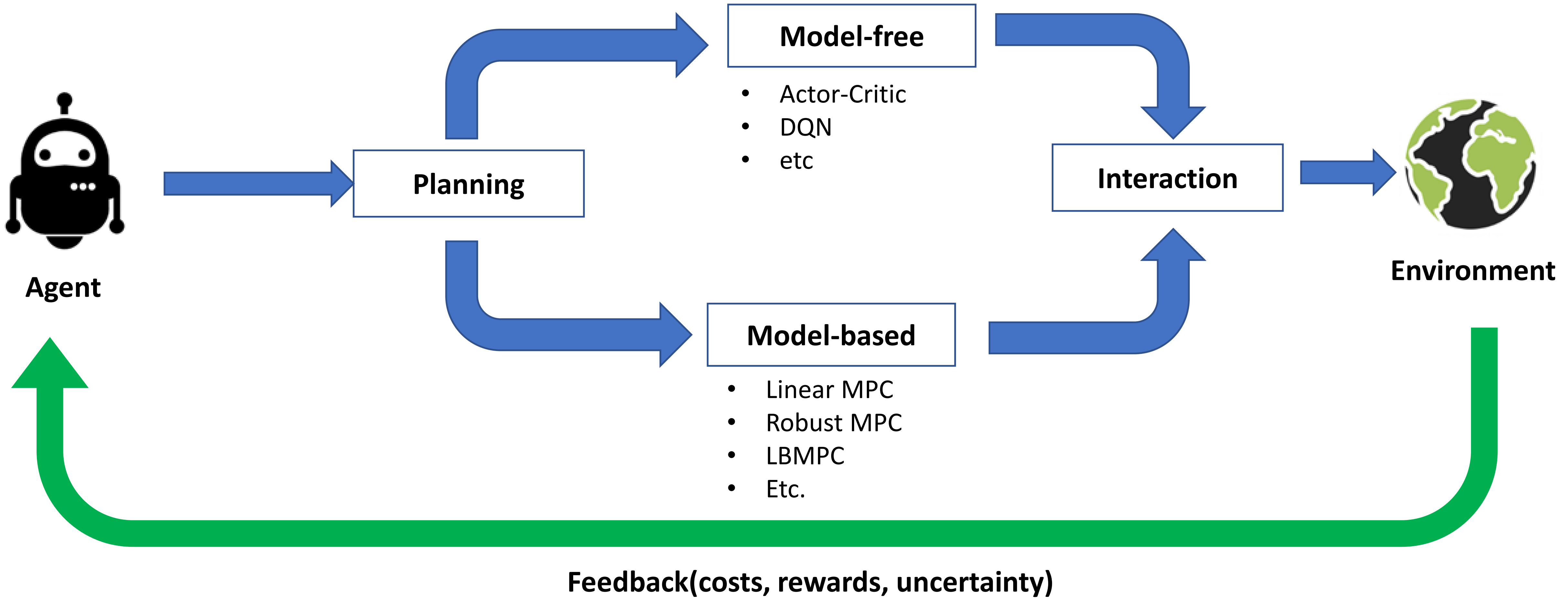
Learning-Based Model Predictive Control (LBMPC)

- And the LBMPC optimal control problem is given by:

$$
\begin{aligned}
J_0(\bar{x}_0, \theta^{(0)}) = \min_{X, U, Z} \quad & \hat{J}_N(\tilde{x}_N) + \sum_{i=0}^{N-1} \tilde{x}_i^\top Q \tilde{x}_i + u_i^\top R u_i \\
\text{s.t.} \quad & \tilde{x}_{i+1} = A\tilde{x}_i + Bu_i + h(\tilde{x}_i, u_i; \theta^{(0)}), \quad \forall i \in \{0, 1, 2, ...\} \\
& \bar{x}_{i+1} = A\bar{x}_i + Bu_i, \quad \forall i \in \{0, 1, 2, ...\} \\
& u_i = K\bar{x}_i + z_i, \quad \forall i \in \{0, 1, 2, ...\} \\
& \bar{x}_i \in \mathcal{X} \ominus \mathcal{R}_i, \; u_i \in \mathcal{U} \ominus (K \circ \mathcal{R}_k), \quad \forall i \in \{0, 1, 2, ...\} \\
& x_0 = \bar{x}_0 \\
& \bar{x}_N \in \mathcal{X}_f \ominus \mathcal{R}_N
\end{aligned}
$$

Overview of Approximate Dynamic Programming

- We can summarize all methods as algorithms that perform **planning** and **policy** execution:



Inverse Decision Making

- So far, the focus has been on planning:
  - Making good decisions
  - Planning ahead
  - Creating strong policies

- Let’s focus now on the agent’s themselves. In particular we are interested in the following questions:
  - What do the agents want? What are they trying to accomplish in the environment?
  - Do they even have preferences?
  - What are preferences?
  - Can we “learn” their preferences? How?

- These are the question we will try to answer on the remainder portion of our lectures!

Preferences and Utility Functions

- Let's first tackle the notion of preferences.

- Abstractly, agents (humans and machines) have preferences:
  - Humans naturally prefer certain things over others
  - Machines have preferences over outcomes, due to their programming

- Formally, a preference is a **partial order.**
  - like positive semi-definiteness for matrices (it is a partial order)

- Given two elements x_1, x_2 belonging to some set X, we say that if an agent (weakly) prefers x_1 over x_2 then we write:

$$x_1 \succeq x_2$$

- And if we are indifferent between x_1 and x_2 we write: $x_1 \sim x_2$

Preferences and Utility Functions

- In order for the partial order to be useful we need to "quantify" the ordering somehow.

- That is the role of the Utility Function.

- Consider again the two elements x_1 and x_2. We will say that an agent (weakly) prefers x_1 over x_2 if and only if there is an Utility function $\mathrm{U}(x)$ such that:

$$U(x_1) \geq U(x_2) \Leftrightarrow x_1 \succeq x_2$$

- Hence the Utility function enforces an **ordering** among elements:
  - It does not really matter that the utility is 1 or 10
  - All it matters is the order (whether it is bigger or smaller than something else)

Preferences and Utility Functions

- One good question to make is whether it even makes sense to represent an agent's preferences by an Utility Function.

- It turns out that if the preferences (i.e.: the partial order) satisfies some conditions we can, in fact, represent the agent's preferences by an Utility Function.

- This the **Von Neumann–Morgenstern Utility Theorem.**

- In particular, the following conditions are essential:

$$x_1 \succ x_2, \quad x_1 \prec x_2, \text{or} \quad x_1 \sim x_2 \quad , \forall x_1, x_2 \in X$$

Complete Preferences

$$\text{If } x_1 \succeq x_2 \text{ and } x_2 \succeq x_3, \text{ then } \quad x_1 \succeq x_3 \quad , \forall x_1, x_2, x_3 \in X$$

Transitive Preferences

Preferences and Utility Functions

- Preferences that are both complete and transitive are called **Rational.**

- Hence, in this definition, a **rational agent** is the one that have complete and transitive preferences.

- And any rational agent can have their preferences represented by an Utility Function.

- We will work with Utility Functions, so we will assume rationality for any agent we consider.

Preferences and Choices

- Preferences cannot be observed:
  - They are “inside” a person’s mind

- What we can observe are the person’s choices (or actions).

- We can then use the information from those choices/actions to infer the agent’s preferences.

- Of course, we may think: “What if we just ask the agent directly what are their preferences?”
  - (1) The agents can lie (they may not tell the truth)
  - (2) The agents may not be able to answer due to uncertainty in the environment

Example: Contract Selection

- One of the earliest methods of interfering preferences is by leveraging the concept of "self-selection".

- We provide the agent a set of possible choices and ask the agent to select one of them.

- If we craft the choice set carefully, we can induce the agent to reveal to us their preference, even if they are not willing.

- This is the "self-selection" phenomenon: the agent's select an option that reveals their underlying preferences.

Example: Contract Selection

- Suppose we own a company and we would like to hire new employees.

- Our goal is to maximize our profit and our profit function is given as follows:

$$P(q, t) = 10\sqrt{q} - t$$

- Where:
  - q is the amount of “work” done by the employees.
  - t is their salary

- Now, assume, there are two types of workers:
  - (1) a worker who is a perfect fit for the job (called the “efficient” worker).
  - (2) a worker who is not (called the “inefficient” worker).

Example: Contract Selection

- Suppose the workers have preferences over pairs of "work" and salaries. So if the worker (weakly) prefers the work-salary pair (q_1, t_1) over (q_2, t_2) we say:

$$(q_1, t_1) \succeq (q_2, t_2)$$

- Under our Rationality assumption, suppose that the workers utility functions are given by:

$$U_i(q,t) = t - \theta_i q - F \quad \forall i \in \{E, I\}$$

- Where θ_i is a scalar that represents the agent's **private information (type)**
  - We let θ_E denote the type of the efficient worker and θ_I the type of the inefficient worker.
  - And $\theta_I > \theta_E$
  - F is the utility of staying at home(!)

Example: Contract Selection

- Now the problem is to design a hiring offer where we are able to hire the efficient worker.

- Suppose $F = 20$ and $\theta_E = 0.25$ and $\theta_I = 0.30$

- Let's suppose that our hiring process is to just "ask" the worker what is their type. And then we offer a contract of work-salary (q, t) to them.

- First suppose they tell the truth (or we have perfect information about their types)

- Then, in order to hire the workers we need to make sure we make then an offer such that:

$$U_I(q_I, t_I) = t_I - 0.30q_I \geq 20$$
$$U_E(q_E, t_E) = t_E - 0.25q_E \geq 20$$

Participation Constraints

Example: Contract Selection

- As the company, it is enough if we offer a contract such that each agent's is indifferent between coming to work or staying at home.

- That is we would offer a contract (q, t) such that:

$$t_i - \theta_i q_i = 20 \quad , \forall i \in \{E, I\}$$

- Then we wish to maximize:

$$P(q_i, t_i) = 10\sqrt{q_i} - t_i = 10\sqrt{q_i} - \theta_i q_i - 20$$

- Taking the derivative and setting it to zero, gives the following set of contracts:

$$(q_E, t_E) = (400, 120) \qquad (q_I, t_I) = (277, 103.1)$$

Example: Contract Selection

- Now, let's see what happens if we **do not know** the workers type and we offer both choices of contracts:

$$(q_E, t_E) = (400, 120) \qquad\qquad (q_I, t_I) = (277, 103.1)$$

- Note that the inefficient worker will still select the contract intended to them (self-select):

$$t_E - \theta_I q_E - F = -20 \qquad\qquad t_I - \theta_I q_I - F = 0$$

- But the efficient worker will **not** prefer the contract intended to them:

$$t_I - \theta_E q_I - F = 13.85 \qquad\qquad t_E - \theta_E q_E - F = 0$$

- So our set of choices, is not good: we are not able to infer the workers type by their contract choices

Example: Contract Selection

- Let's design a new set of contracts.

- Since the types are unknown, suppose there is 50% chance of a worker that we offer a contract to be efficient.

- In order to **induce** self-selection (or in order words, **truth-telling**) we need make sure:

$$t_I - \theta_I q_I - F \geq t_E - \theta_I q_E - F$$

$$t_E - \theta_E q_E - F \geq t_I - \theta_E q_I - F$$

Incentive Compatibility Constraints

- Those enforce self-selection when offering the set of contract choices.

- Let's put everything together into an optimization problem.

Example: Contract Selection

- The company finds the set of contract choices by solving the following optimization problem.

$$\max_{(q_I,t_I),(q_E,t_E)} \frac{1}{2}(10\sqrt{q_I} - t_I) + \frac{1}{2}(10\sqrt{q_E} - t_E)$$

$$\text{s.t.:}$$

Incentive Compatibility Constraints

$$\begin{cases} t_I - \theta_I q_I - F \geq t_E - \theta_I q_E - F \\ t_E - \theta_E q_E - F \geq t_I - \theta_E q_I - F \end{cases}$$

Participation Constraints

$$\begin{cases} t_I - \theta_I q_I - 20 \geq 0 \\ t_E - \theta_E q_E - 20 \geq 0 \end{cases}$$

- We will continue next time, presenting a framework in which to solve this problem and more inverse-decision making problems.