Reinforcement Learning

Chapter 4: Function Approximation

Ali Bereyhi

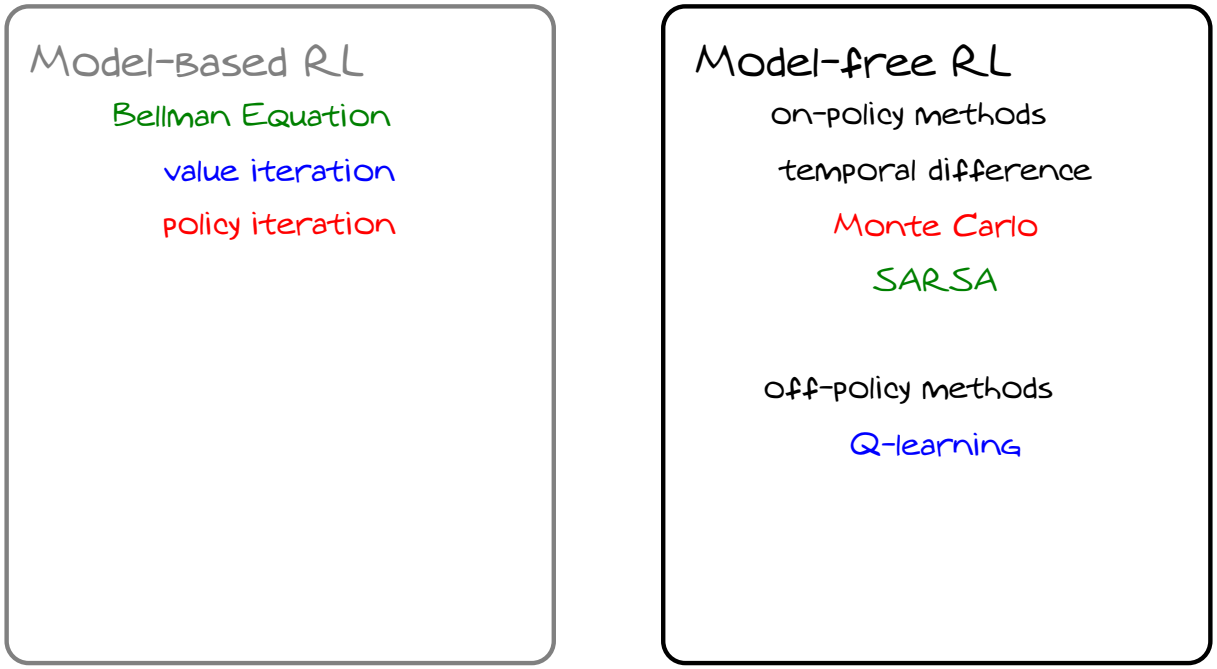
ali.bereyhi@utoronto.ca

Department of Electrical and Computer Engineering

University of Toronto

Fall 2025

Review: *Where Are We Now?*



We need to overcome one last challenge

we need to learn how to deal with large-scale problems

Tabular RL: *What We Had Already*

What we have studied up to now is usually called tabular RL

+ *Why we call it tabular?*
- Because we think of *value* and *action-value* function as a *table*

Let's consider a mode-free control loop with finite number of states and actions

- *We initiate all action-values with zero*
  - ↳ *We initiate a table of zeros*
- *But, we do know that these action-values are some specific values*
  - ↳ *We are dealing with a table of unknowns*
- *We try estimating them from samples*

Tabular RL: *Schematic*

	a^1	a^2	$\dots$	a^M
s^1	$\hat{q}_\pi\left(s^1, a^1\right)$	$\hat{q}_\pi\left(s^1, a^2\right)$		
s^2				
$\vdots$				
s^N				$\hat{q}_\pi\left(s^N, a^M\right)$

Q-table

s^1	$\hat{v}_\pi\left(s^1\right)$
s^2	$\hat{v}_\pi\left(s^2\right)$
$\vdots$	$\vdots$
s^N	$\hat{v}_\pi\left(s^N\right)$

value-table

Computational Complexity of Tabular RL

A tabular approach needs estimation of NM values from samples

- *Say we use Monte-Carlo*
  - ↳ *We need to visit all the state-action pairs enough times*
  - ↳ *Say enough is C for us; then, we need*

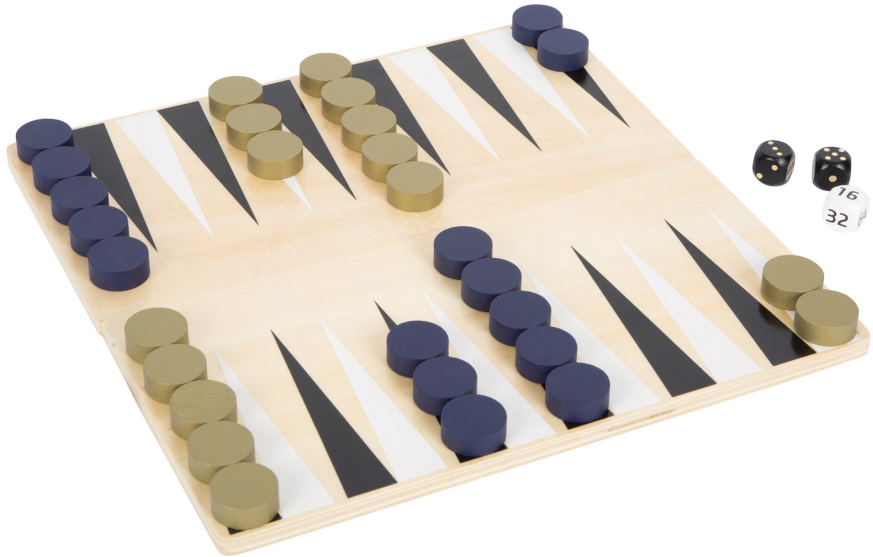
$$\#\text{ sample pairs in all episodes} \approx CNM$$

- *Same thing with temporal difference*

Moral of Story

In tabular RL methods, the number of required sample interactions with the environment scales with number of states and number of actions

Complexity Examples: *Backgammon*

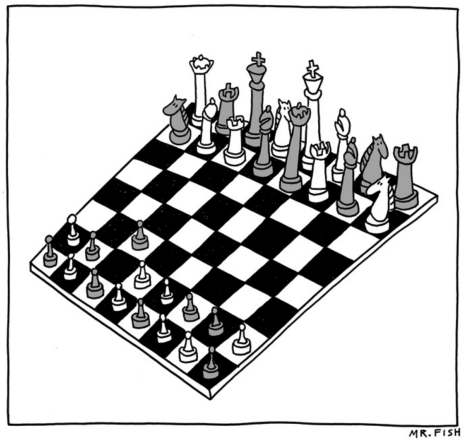


There are roughly 10^{20} possible states for backgammon

- *Say we can get over with each state-action pair and update in only 1 Picosec = 10^{-12} sec; then, we need*

$$\#\,\text{time} \approx 10^{8} CM \text{ sec} \approx 3.2 CM \text{ years}$$

Complexity Examples: *Chess*



In the fun part of Assignment 1, you saw that Shannon found out about roughly 10^{120} *possible states for chess, and later on some people came out with some approximations for legal positions: let's take Tromp's number, i.e.,* $\approx 4 \times 10^{44}$

- *Say again we need only 1 Picosec per step; then, we need*

$$\#\,\text{time} \approx 4 \times 10^{32} CM \text{ sec} \approx 12 \times 10^{24} CM \text{ years}$$

Milky Way is about 13.6 billion years old!

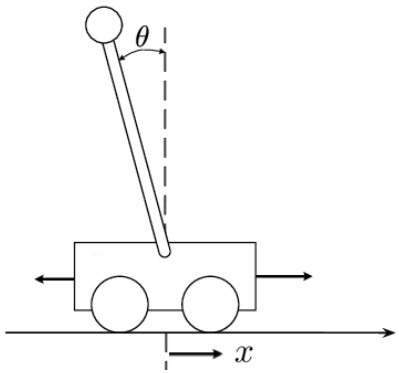
Complexity Examples: *Game Go*

There are roughly 10^{170} possible states for board game Go

- *Say again we need only 1 picosec*

$\#\,\text{time} \approx$ *do we really need computing it?* ☺

Computational Complexity: *Continuous State Space*



We saw the Cart-Pole problem: *if we want to put it into a tabular form*

- *We need to put it into a table*
  - ↳ *We have to make a discrete grid of states*
  - ↳ *Say we have L state parameters and we grid each one with B bins*
    - ↳ *We have B^L grid points in total*
- *This grows exponentially large in number of state parameters!*

Computational Complexity of Tabular RL

In most interesting problems, we are dealing with scaling problem

- *We have an exponentially large number of states*
- *We cannot even visit a subset of them!*
- *Our tabular RL algorithms will not give us any meaningful results!*

+ *What then?! Should we always play 4×4 Frozen Lake with RL?!*
– No! We try to approximate our table from our limited observations

Approximating Value Function

Let's start with a simple case: *say we want to evaluate a policy, i.e.,*

> *We are given with a policy π and want to find some estimate $\hat{v}_\pi(\cdot)$*

In the tabular RL, we assume that

$$v_\pi\left(s^1\right) = v^1 \qquad \ldots \qquad v_\pi\left(s^N\right) = v^N$$

for some unknown $v^1, \ldots, v^N$ and try to estimate them

Approximating Value Function

Let us now approach the problem differently: *we assume that*

$$v_\pi(s) = f(s, \mathbf{w})$$

for some weights in $\mathbf{w}$ *and a known function* $f()$

- $f()$ *is a approximation model that we assume for the value function*
- $\mathbf{w}$ *contains a set of learnable parameters*

+ *How on earth we know such a thing?!*
– *We don't really know it! We just assume it; however, sometimes it really makes sense*

Approximating Value Function

If we assume an approximation model: *we use our observations to find weight vector* $\mathbf{w}^\star$ *that fits this approximator best to our observations. We then set*

$$\hat{v}_\pi(s) = f(s, \mathbf{w}^\star)$$

+ *How is it better than tabular then?!*
– Well! For lots of reasons

1. *We can use every single sample to update the estimate for all states*
   ↳ *We use samples to fit* $\mathbf{w}$*: this impacts on the whole value function*
2. *We get some updating estimates for states that have not been visited*
   ↳ *If we get the right* $\mathbf{w}$*: we get the estimator for all states*
3. *We can capture the impact of one state on the others*
   ↳ *If we update* $\mathbf{w}$ *after seeing* S_t*: we change estimated values of all states*

Approximating Value Function

+ *But how can we find such approximators?!*
- There are various types of them

- *Linear function approximators*
- *Deep neural networks (DNNs)*
- *Eigen-transforms, e.g., Fourier or Wavelets*
- . . .

In this course, we are focusing only on parametric approximators which include

- *Linear function approximators*
- *DNNs*

Because they are differentiable: we will see why this is important!

But before using them, let's look at them and understand how they work

Function Approximation: *Formulation*

In function approximation, we have

- *A set of sample input-outputs of a function that we do not know*

$$\mathbb{D} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i) : i = 1, \ldots, I\}$$

  - ↳ $\boldsymbol{x}_i$ *is the input and* $\boldsymbol{y}_i$ *is the output*
  - ↳ *Let's denote the unknown function with* $g(\cdot)$, *i.e.,* $\boldsymbol{y}_i = g(\boldsymbol{x}_i)$

- *We assume an approximating model for this unknown function*

$$\hat{\boldsymbol{y}} = f(\boldsymbol{x}|\mathbf{w})$$

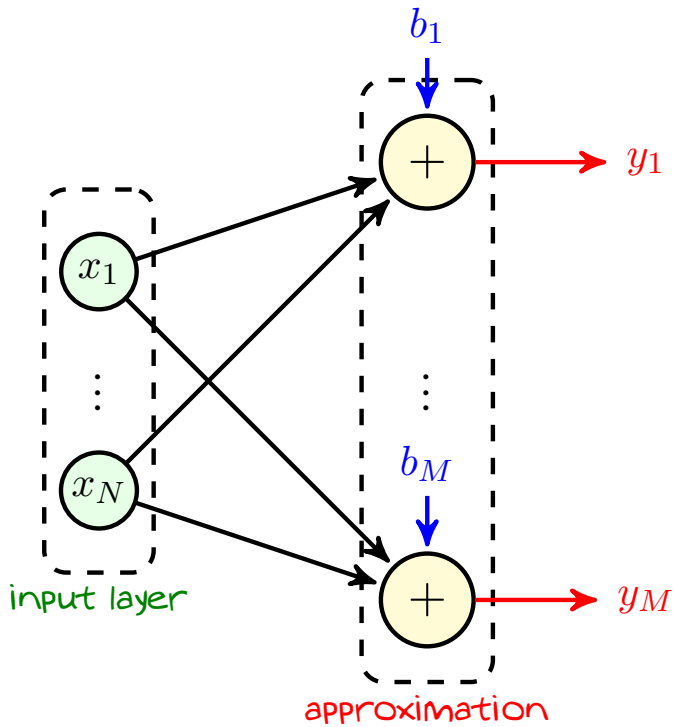
  - ↳ *In this approximator* $\mathbf{w}$ *is learnable, i.e., we are free to tune it as we wish*
  - ↳ *We treat output of this approximator as estimate of function outputs*

Example: *Linear Function Approximation*

A simple example of a function approximator is the linear approximator

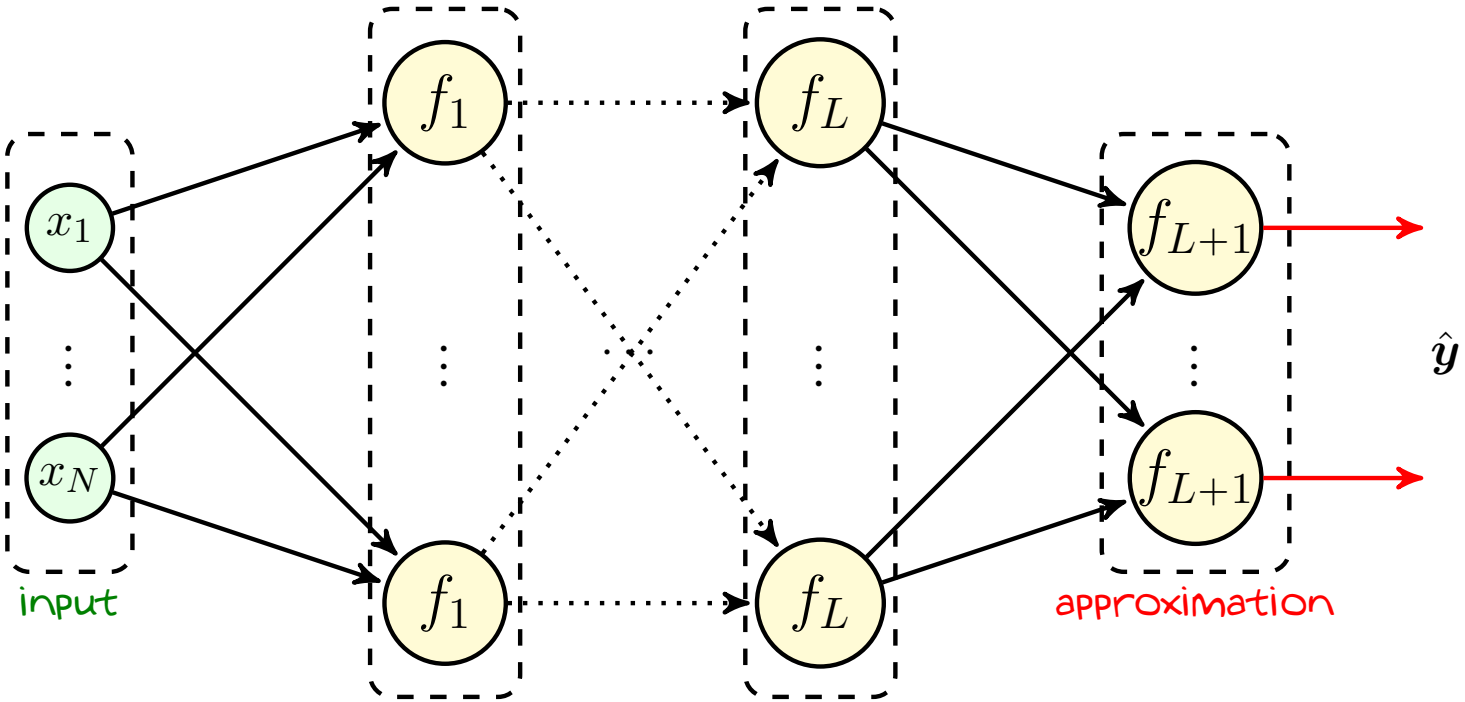
$$\hat{\boldsymbol{y}} = \mathbf{W}\boldsymbol{x} + \mathbf{b}$$

- $\mathbf{W}$ *is the matrix of weights*
- $\mathbf{b}$ *is the vector of biases*



Example: *Deep Neural Networks*

A deep neural network is also a parameterized approximator



Example: *Deep Neural Networks*

+ *Why should we rely on these approximators?*
– *They are known to be very powerful*

Universal Approximation Theorem

Given a family for neural networks: for any function g from its function space, there exists a sequence of configurations that approximates g arbitrarily precise

Function Approximation: *Formulation*

With parametric approximators: *we want to find weight* $\mathbf{w}$ *so that we have*

- *a good approximator: it fits best the dataset, i.e.,*

$$\hat{\boldsymbol{y}}_i = f\left(\boldsymbol{x}_i | \mathbf{w}\right) \approx \boldsymbol{y}_i$$

  ↳ *We need to define a notion for* $\approx$

- *an approximator that generalizes: if we get a new sample input* $\boldsymbol{x}_{\text{new}}$*, we somehow can make sure that*

$$\hat{\boldsymbol{y}}_{\text{new}} = f\left(\boldsymbol{x}_{\text{new}} | \mathbf{w}\right) \approx g\left(\boldsymbol{x}_{\text{new}}\right)$$

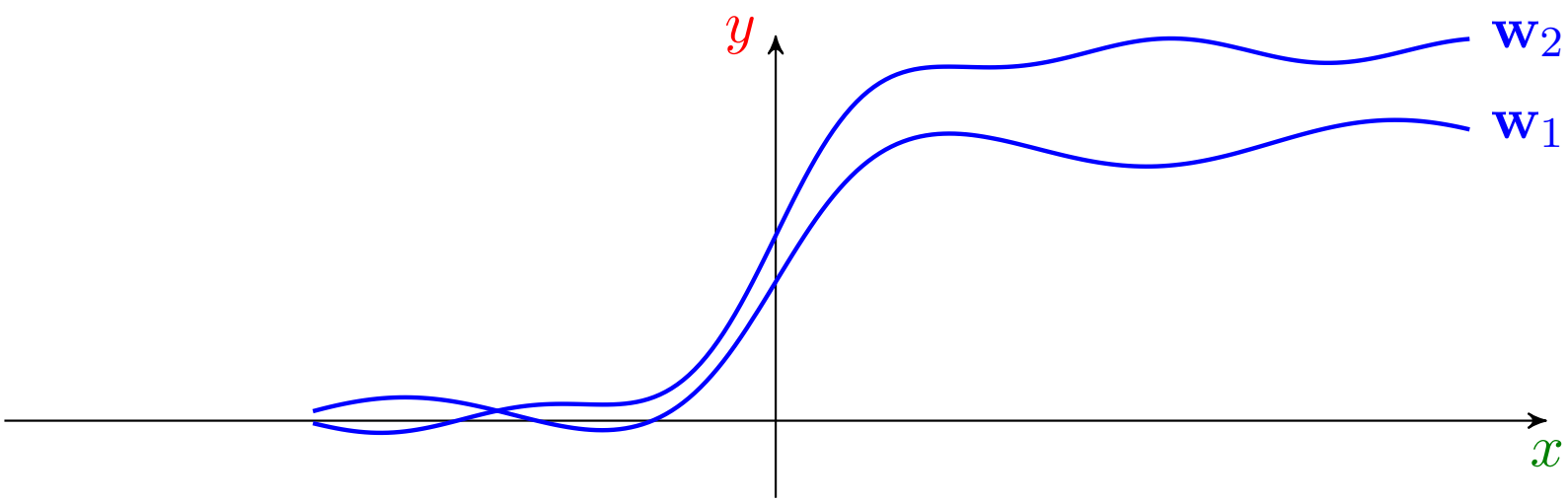
  ↳ *We should find a way to check this*

Visualizing Function Approximation

Let's assume one-dimensional inputs and outputs: *this assumption helps us visualize the function approximator*

$$y = f\left(x|\mathbf{w}\right)$$

With scalar input, we can visualize the model as



As learnable parameters change, approximator sketches different functions

Training: *Empirical Risk Minimization*

Empirical Risk

Let $\mathbf{w}$ *includes all learnable parameters, and the dataset be*

$$\mathbb{D} = \{(\boldsymbol{x}_i, y_i) : i = 1, \ldots, I\}$$

for loss function $\mathcal{L}$*, the empirical risk is defined as*

$$\hat{R}(\mathbf{w}) = \frac{1}{I} \sum_{i=1}^{I} \mathcal{L}\left(f\left(\boldsymbol{x}_i | \mathbf{w}\right), \boldsymbol{y}_i\right)$$

The training is performed by minimizing the empirical risk

$$\mathbf{w}^{\star} = \underset{\mathbf{w}}{\operatorname{argmin}}\, \hat{R}(\mathbf{w}) = \underset{\mathbf{w}}{\operatorname{argmin}}\, \frac{1}{I} \sum_{i=1}^{I} \mathcal{L}\left(f\left(\boldsymbol{x}_i | \mathbf{w}\right), \boldsymbol{y}_i\right) \qquad \text{(Training)}$$

Gradient Descent

Let's use gradient descent for function approximation: *we want to minimize*

$$\hat{R}(\mathbf{w}) = \frac{1}{I}\sum_{i=1}^{I}\mathcal{L}\left(f\left(\boldsymbol{x}_i|\mathbf{w}\right), y_i\right)$$

`GradientDescent():`

1: *Initiate with some initial* $\mathbf{w}^{(0)}$ *and set a learning rate* η
2: ***while*** *weights not converged* ***do***
3: ***for*** $i = 1, \ldots, I$ ***do***
4: *Compute gradient for* $\nabla_i = \nabla\mathcal{L}\left(f\left(\boldsymbol{x}_i|\mathbf{w}^{(t-1)}\right), \boldsymbol{y}_i\right)$
5: *Update gradient as* $\nabla\hat{R}(\mathbf{w}^{(t-1)}) \leftarrow \nabla\hat{R}(\mathbf{w}^{(t-1)}) + \nabla_i/I$
6: ***end for***
7: *Update weights as* $\mathbf{w}^{(t)} \leftarrow \mathbf{w}^{(t-1)} - \eta\nabla\hat{R}(\mathbf{w}^{(t-1)})$
8: ***end while***

We call this form of gradient descent full-batch which can be too complex

Stochastic Gradient Descent

We use gradient descent for function approximation: *we want to minimize*

$$\hat{R}(\mathbf{w}) = \frac{1}{I}\sum_{i=1}^{I}\mathcal{L}\left(f\left(\boldsymbol{x}_i|\mathbf{w}\right), \boldsymbol{y}_i\right)$$

SGD():

1: *Initiate with some initial* $\mathbf{w}^{(0)}$ *and set a learning rate* η
2: ***while*** *weights not converged* ***do***
3: ***for*** *a random subset of batch* $b = 1, \ldots, B$ ***do***
4: *Compute gradient for* $\nabla_b = \nabla\mathcal{L}\left(f\left(\boldsymbol{x}_b|\mathbf{w}^{(t-1)}\right), \boldsymbol{y}_b\right)$
5: *Update gradient as* $\nabla\hat{R}(\mathbf{w}^{(t-1)}) \leftarrow \nabla\hat{R}(\mathbf{w}^{(t-1)}) + \nabla_b/I$
6: ***end for***
7: *Update weights as* $\mathbf{w}^{(t)} \leftarrow \mathbf{w}^{(t-1)} - \eta\nabla\hat{R}(\mathbf{w}^{(t-1)})$ ← unbiased estimator
8: ***end while***

This is what we call stochastic mini-batch gradient descent