Function Approximation: Machine Learning and Gradient Descent

Joseph Le Roux

10/07/26

Outline

1
Introduction

1 $V(s)$ and $Q(s,a)$: Table vs. Function

Why get rid of tables?

Tables

1. impose a discretization of states;
2. use memory space at least linear in the number of states;
3. cannot easily indicate that *close* states tend to have *similar* values

Why use functions?

Functions

1. can replace tables (functions by *if-then-else* cases), usually with less space
2. can manage dense states
3. can easily be constrained to give close states similar values
4. function definition *explains* how decisions are made

1 Function Approximation

Parametrized Functions

Let us assume that values v and q can be approximated by functions:

- $\hat{v}(s; \mathbf{w}) \approx v^{\pi}(s)$
- $\hat{q}(s, a; \mathbf{w}) \approx q^{\pi}(s, a)$

where $\mathbf{w}$ is a vector of parameters in $\mathbb{R}^d$.

What parameters?

In practice, restrict to known *families* of functions (linear, convex, polynomial, implementable by neural networks,...) to find optimal parameters $\mathbf{w}$ for our agent.
The quality of the function approximation can be measured by an approximation error function.

1 Learning via Approximation Error

Let $\mathcal{T}$ be a set of trajectories, and π a policy. The family of $\hat{v}$ is fixed.

Approximation Mean Squared Error

- always positive, or zero when no error
- for state values:

$$MSE_v(\mathbf{w}; \mathcal{T}) = \frac{1}{\sum_{\tau \in \mathcal{T}} |\tau|} \sum_{\tau \in \mathcal{T}} \sum_{t=0}^{|\tau|-1} \frac{1}{2} \Big(v^\pi(s_t) - \hat{v}(s_t; \mathbf{w}) \Big)^2$$

- for state-action values, replace equality between q^π and table Q

$$MSE_q(\mathbf{w}; \mathcal{T}) = \frac{1}{\sum_{\tau \in \mathcal{T}} |\tau|} \sum_{\tau \in \mathcal{T}} \sum_{t=0}^{|\tau|-1} \frac{1}{2} \Big(q^\pi(s_t, a_t) - \hat{q}(s_t, a_t; \mathbf{w}) \Big)^2$$

Remark that :

1. MSE is a function of $\mathbb{R}^d \longrightarrow \mathbb{R}$ (from parameters to scalars)
2. MSE is a convex function in $\hat{v}$, and if $\hat{v}$ linear in $\mathbf{w}$, convex in $\mathbf{w}$.
3. In reality we don't know v^π (otherwise, why learn it?) but let us pretend it is known for now

A good approximation is one that has low MSE

- since the family of $\hat{v}$ is fixed, the only difference between approximations is $\mathbf{w}$.
- 🤯 can we find $\mathbf{w}$ which minimizes MSE for given $\mathcal{T}$, π??

1 Learning via Approximation Error: Stochastic Gradient Descent (1)

Analytic Method

The $\mathbf{w}$ that minimises MSE has null gradient: $\nabla_{\mathbf{w}} MSE(\mathbf{w}; \mathscr{T}) = 0$

Problem we don't always know how to solve a non-linear equations system, or intractable in practice (sum over $\mathscr{T}$)

Iterative Method: stochastic gradient descent

make small updates after each example (action/reward) instead of one big update after all trajectories have been collected

- error for 1 example: $MSE_v(\mathbf{w}; \tau, t) = \frac{1}{2}\left(v^{\pi}(s_t) - \hat{v}(s_t; \mathbf{w})\right)^2$
- $MSE_v(\mathbf{w}; \mathscr{T}) = \frac{1}{\sum_{\tau \in \mathscr{T}} |\tau|} \sum_{\tau \in \mathscr{T}} \sum_{t=0}^{|\tau|-1} MSE_v(\mathbf{w}; \tau, t)$
- $\nabla MSE_v(\mathbf{w}; \mathscr{T}) = \frac{1}{\sum_{\tau \in \mathscr{T}} |\tau|} \sum_{\tau \in \mathscr{T}} \sum_{t=0}^{|\tau|} \nabla MSE_v(\mathbf{w}; \tau, t) = \mathbb{E}[\nabla MSE_v(\mathbf{w}; \tau, t)]$ where $p(\tau, t) = \frac{1}{\sum_{\tau \in \mathscr{T}} |\tau|}$.

Remark

Since we will always compute derivatives w.r.t , we drop from gradient notations: $\nabla MSE(; \mathscr{T}) = \nabla MSE(; \mathscr{T})$

1 Learning via Approximation Error: Stochastic Gradient Descent (2)

Algorithm:

- input : sequence of step sizes $(\alpha_i)_{i\geq 0}$
  - positive, small, decreasing
- init : $_0$ initialised arbitrarily
- loop:
  - draw randomly (τ, t) with $\tau \in \mathcal{T}, t \in 0 \dots |\tau|$
  - $\mathbf{w}_{n+1} \leftarrow \mathbf{w}_n - \alpha_n \nabla MSE_v(\mathbf{w}_n; \tau, t)$
- return final $\mathbf{w}$

1 Learning via Approximation Error: Stochastic Gradient Descent (2)

Algorithm:

- input : sequence of step sizes $(\alpha_i)_{i\geq 0}$
  - positive, small, decreasing
- init : $_0$ initialised arbitrarily
- loop:
  - draw randomly (τ, t) with $\tau \in \mathcal{T}, t \in 0 \dots |\tau|$
  - $\mathbf{w}_{n+1} \leftarrow \mathbf{w}_n - \alpha_n \nabla MSE_v(\mathbf{w}_n; \tau, t)$
- return final $\mathbf{w}$

Why does it work? (sketch for GD)

Gives a decreasing sequence ... $\geq MSE(\mathbf{w}_n; \tau, t) \geq MSE(\mathbf{w}_{n+1}; \tau, t) \geq$ ...

$$
\begin{aligned}
MSE(_n) - MSE(_{n+1}) &= MSE(_n) - MSE\Big(_n - \alpha_n \nabla MSE(_n)\Big) \\
&= MSE(_n) - [MSE(_n) - \alpha_n ||\nabla MSE(_n)||_2^2 + o(\alpha_t^2 \nabla MSE(_n)^2)] \\
&= \alpha_n ||\nabla MSE(_n)||_2^2 + o(\alpha_n^2 \nabla MSE(_n^2)) \\
&\approx \alpha_n ||\nabla MSE(_n)||_2^2 \geq 0
\end{aligned}
$$

- convergence depends on α_n
- for a more rigorous proof, we need more assumptions (convexity...)

1 Learning via Approximation Error: Stochastic Gradient Descent (2)

Algorithm:

- input : sequence of step sizes $(\alpha_i)_{i\geq 0}$
  - positive, small, decreasing
- init : $_0$ initialised arbitrarily
- loop:
  - draw randomly (τ, t) with $\tau \in \mathcal{T}, t \in 0 \dots |\tau|$
  - $\mathbf{w}_{n+1} \leftarrow \mathbf{w}_n - \alpha_n \nabla MSE_v(\mathbf{w}_n; \tau, t)$
- return final $\mathbf{w}$

Why does it work? (sketch for GD)

Gives a decreasing sequence $\dots \geq MSE(\mathbf{w}_n; \tau, t) \geq MSE(\mathbf{w}_{n+1}; \tau, t) \geq \dots$

$$\begin{aligned} MSE(_n) - MSE(_{n+1}) &= MSE(_n) - MSE\Big(_n - \alpha_n \nabla MSE(_n)\Big) \\ &= MSE(_n) - [MSE(_n) - \alpha_n ||\nabla MSE(_n)||_2^2 + o(\alpha_t^2 \nabla MSE(_n)^2)] \\ &= \alpha_n ||\nabla MSE(_n)||_2^2 + o(\alpha_n^2 \nabla MSE(_n^2)) \\ &\approx \alpha_n ||\nabla MSE(_n)||_2^2 \geq 0 \end{aligned}$$

- convergence depends on α_n
- for a more rigorous proof, we need more assumptions (convexity…)

Why does it work? (sketch for SGD)

- replace $\nabla MSE(_n)$ (an average) with a sampled $\nabla MSE_v(\mathbf{w}_n; \tau, t)$ from $p(\tau, t)$
- this works because using samples give an unbiased estimator (cf. definition by expectation previous slide)

2 Application to Policy Iteration

2 Application to $\hat{v}$

SGD update for MSE_v

$$\begin{aligned}
\mathbf{w_{n+1}} &\leftarrow \mathbf{w_n} - \alpha_n \nabla MSE_v(\mathbf{w}_n; \tau, t) \\
&\leftarrow \mathbf{w}_n - \alpha_n \nabla \frac{1}{2}\Big(v^\pi(s_t) - \hat{v}(s_t; \mathbf{w})\Big)^2 \\
&\leftarrow \mathbf{w}_n - \alpha_n \nabla \frac{1}{2}\{\Big(v^\pi(s_t)\Big)^2 - 2v^\pi(s_t)\hat{v}(s_t; \mathbf{w}_n) + \Big(\hat{v}(s_t; \mathbf{w}_n)\Big)^2\} \\
&\leftarrow \mathbf{w}_n + \alpha_n \Big\{v^\pi(s_t) - \hat{v}(s_t; \mathbf{w}_n)\Big\}\nabla \hat{v}(s_t; \mathbf{w}_n)
\end{aligned}$$

Parameter update proportional to the difference between the true value and the current approximation, multiplied by the gradient of the current approximation. $\Longrightarrow$ looks like MC update, but dimensions *rescaled* by derivatives

Problem

as mentionned before we don't know v^π, let alone v_*

Solution : Use an Estimator

$$\mathbf{w}_{n+1} \leftarrow \mathbf{w}_n + \alpha_n \Big\{U(s_t) - \hat{v}(s_t; \mathbf{w}_n)\Big\}\nabla \hat{v}(s_t; \mathbf{w}_n)$$

1. via sampling (Monte-Carlo style) : $U(s_t) = G_t$
2. via Bellman (*bootstrap* style) : $U(s_t) = \mathrm{sg}(r_{t+1} + \gamma \hat{v}(s_{t+1}; \mathbf{w}_n))$
   - where sg means *stop gradient*: we treat $U(s_t)$ as a constant although it's computation here depends on
   - we depart from the real SGD algorithm: this is a semi-gradient algorithm (will not converge in general)

2 Application to $\hat{q}$

SGD update for MSE_q

$$
\begin{aligned}
\mathbf{w}_{n+1} &\leftarrow \mathbf{w}_n - \alpha_n \nabla MSE_q(\mathbf{w}_n; \tau, t) \\
&\leftarrow \mathbf{w}_n - \alpha_n \nabla \frac{1}{2}\Big(q^\pi(s_t, a_t) - \hat{q}(s_t, a_t; \mathbf{w}_n)\Big)^2 \\
&\leftarrow \mathbf{w}_n - \alpha_n \nabla \frac{1}{2}\{\Big(q^\pi(s_t, a_t)\Big)^2 - 2q^\pi(s_t, a_t)\hat{q}(s_t, a_t; \mathbf{w}_n) + \Big(\hat{q}(s_t, a_t; \mathbf{w}_n)\Big)^2\} \\
&\leftarrow \mathbf{w}_n + \alpha_n \Big\{q^\pi(s_t, a_t) - \hat{q}(s_t, a_t; \mathbf{w}_n)\Big\} \nabla \hat{q}(s_t, a_t; \mathbf{w}_n)
\end{aligned}
$$

Parameter update proportional to the difference between the true value and the current approximation, multiplied by the gradient of the current approximation. $\Longrightarrow$ looks like MC update, but dimensions *rescaled* by derivatives

Problem

as mentionned before we we don't know q^π, nor q_* of course

Solution : use an estimator

$$
\mathbf{w}_{n+1} \leftarrow \mathbf{w}_n + \alpha_n \Big\{U(s_t, a_t) - \hat{q}(s_t, a_t; \mathbf{w}_n)\Big\} \nabla \hat{q}(s_t, a_t; \mathbf{w}_n)
$$

1. Via sampling (Monte-Carlo style) : $U(s_t, a_t) = G_t$
2. Via Bellman (*bootstrap* style) : $U(s_t, a_t) = \mathrm{sg}(r_{t+1} + \gamma \hat{q}(s_{t+1}, a_{t+1}; \mathbf{w}_n))$
   - where sg means *stop gradient*: we treat $U(s_t)$ as a constant although it's computation here depends on
   - we depart from the real SGD algorithm: this is a semi-gradient algorithm (will not converge in general)

2 SARS'A' Algorithm with function

- Input: sequence of step size $(\alpha)_t > 0$ and $\epsilon > 0$ for policy
- Init:
  - parameters $\mathbf{w}$ initialised arbitrarily
  - $t \leftarrow 0$

Infinite Loop:

- $s \leftarrow S_0$ (initialise initial state)
- Choose a from $\hat{q}$ and s (for instance, via ϵ-greedy)
- While s is not terminal
  - from s, a get r, s' (via the environment)
  - Choose a' from $\hat{q}$, policy and s'
  - if s' is not terminal
    - $\leftarrow + \alpha_t\{r + \gamma\hat{q}(s', a'; \mathbf{w}) - \hat{q}(s, a; \mathbf{w})\}\nabla\hat{q}(s, a; \mathbf{w})$ (UPDATE1)
  - otherwise
    - $\leftarrow + \alpha_t\{r - \hat{q}(s, a; \mathbf{w})\}\nabla\hat{q}(s, a; \mathbf{w})$ (UPDATE2)
  - $s \leftarrow s'$; $a \leftarrow a'$; $t \leftarrow t + 1$

2 SARSA Off-policy Variants

Like their tabular counterparts, SARSA with function approximation can be modified to have

- a behaviour policy
- a target policy

and give birth to *off-policy* TD algorithms

Expected SARSA

(UPDATE1) : $\longleftarrow + \alpha\{r + \gamma \sum_b \pi(b|s')\hat{q}(s', b; \mathbf{w}) - \hat{q}(s, a; \mathbf{w})\}\nabla\hat{q}(s, a; \mathbf{w})$

Q-Learning

(UPDATE1) : $\longleftarrow + \alpha\{r + \gamma \max_b \hat{q}(s', b; \mathbf{w}) - \hat{q}(s, a; \mathbf{w})\}\nabla\hat{q}(s, a; \mathbf{w})$

Beware of the deadly triad

Function approximation + bootstrapping + off-policy learning is what Sutton & Barto call the deadly triad

- these three combined can cause Q-Learning with function approximation to diverge.
- Not a theoretical curiosity: it may happen in lab session (although with linear function we are safe)

3 Approximation by Linear Functions

3 How to compute SARSA update's gradient?

It depends on functions $\hat{v}$ and $\hat{q}$

Assume they are linear (or affine)

this means:

- $\hat{v} : \mathbb{R}^d \longrightarrow \mathbb{R}$; $\hat{v}(s; \mathbf{w}) = \sum_{i=1}^{d} \mathbf{w}_i \times s_i$ $\quad$ ($+\mathbf{w}_0$ if affine)
- $\hat{q} : \mathbb{R}^{d \times |A|} \longrightarrow \mathbb{R}$; $\hat{q}(s, a; \mathbf{w}) = \sum_{i=1}^{d} \mathbf{w}_i^a \times s_i$ $\quad$ ($+\mathbf{w}_0^a$ if affine)
- in the remainder, we only detail $\hat{v}$ (simpler to write), as exercise write for $\hat{q}$

Partial derivatives

$\frac{\partial \hat{v}(s;\mathbf{w})}{\partial w_i} = s_i$ if $i > 0$, and 1 if $i = 0$

Gradient

Affine case:

$$\nabla v(s; \mathbf{w}) = [1; s_1; \ldots; s_n]^\top$$

For the linear case it is simply s !

3 Conclusion for gradient descent

Update $\hat{v}$

$$\mathbf{w_{t+1}} \leftarrow \mathbf{w_t} + \alpha_t \Big\{ r_t + \gamma \hat{v}(s'; \mathbf{w}) - \hat{v}(s; \mathbf{w}) \Big\} \times x(s)$$

Update $\hat{q}$ in SARSA and variants

$$\mathbf{w_{t+1}} \leftarrow \mathbf{w_t} + \alpha_t \Big\{ r_t + \gamma \hat{q}(s', a'; \mathbf{w}) - \hat{q}(s, a; \mathbf{w}) \Big\} \times x(s)$$

3 Beyond Linearity: Neural Networks

Why Non-Linearity

Accurately predicting v/q sometimes requires non-linear relations (*and*, *max*, *cos*, powers, logarithms...) between inputs

What are neural networks?

A way to describe non-linear functions as an alternation of:

- linear transformations
- non-linearity operators (such as sigmoid, hyperbolic tangent...)

Thanks to this architecture, gradient can be automatically computed, by the *backpropagation* algorithm (cf INFO3, Data Mining Lectures)

RL and NN

Modern Reinforcement learning (alphago, LLMs, autonomous vehicles...) is based on function approximation powered by neural networks to extract rich information from complex input (go board, complex document and user queries, videos...)

4
Encoding State / Action Input

4 State Representation

Generally $S \neq \mathbb{R}^d$

- in order to use gradient descent, we must *convert* a state into a real vector
- we use *feature extractor*, a function $x : S \longrightarrow \mathbb{R}^d$ where d is fixed a priori
- Examples:
  - for Moutain Car, $S \subset \mathbb{R}^2$, we can simply take $x(s) = s$, or $x(s) = s - b$ to have positive values only (is this what we did in TP?)
  - for a game like Tetris, we will extract features from the current situation in the game:
    1. height of each column
    2. differences between adjacent columns
    3. maximum column height
    4. number of *holes* in each column
    5. ...

    $x : S \longrightarrow \mathbb{R}^{3c}$ where c is the number of columns

Transparent for gradient descent:

For linear $\hat{v}$:

$$\mathbf{w_{t+1}} \leftarrow \mathbf{w_t} + \alpha_t \Big[r_t + \gamma \hat{v}(s'; \mathbf{w}) - \hat{v}(s; \mathbf{w}) \Big] \times x(s)$$

Another advantage of representation functions, is to keep $\hat{v}$ linear while some elements from state s are used in conjunction.
$\Rightarrow$ Functions $\hat{v}$, $\hat{q}$ are still linear in $\mathbf{w}$

Examples

Mountain Car $x(s) = s = (p, v)^\top$ cannot take into account interactions between position et velocity. We could change to $x(s) = (p, v, pv, p^2, v^2)^\top$

Tetris already non-linear. (Why?)

4 Tabular versions ⊆ Functional approximations

Example: convert the table version of Q-Learning/SARSA, using a special kind of feature extractor, *one-hot vectors*.

Encoding a Q-Learning table Q as parameters

- create a parameter vector $\mathbf{w} \in \mathbb{R}^{|S|}$ and bijection $e : S \longrightarrow 1..|S|$ between states and integers such that:
- the feature vector corresponding to s is the vector $x(s)$ such that:
  - $x(e(s)) = 1$ the entry in bijection with cell s is set to 1
  - $x(e(s')) = 0, \forall s' \neq s$ all other entries are set to 0

Parameter updates = Tables updates

$$\mathbf{w_{t+1}} \leftarrow \mathbf{w_t} + \alpha_t \Big\{ r_t + \gamma \hat{v}(s'; \mathbf{w}) - \hat{v}(s; \mathbf{w}) \Big\} \times x(s)$$

- for the cell s the update is:

$$\mathbf{w_{t+1}}[e(s)] \leftarrow \mathbf{w_t}[e(s)] + \alpha_t \Big\{ r_t + \gamma \hat{v}(s'; \mathbf{w}) - \hat{v}(s; \mathbf{w}) \Big\} \times 1$$

- for the other cells:

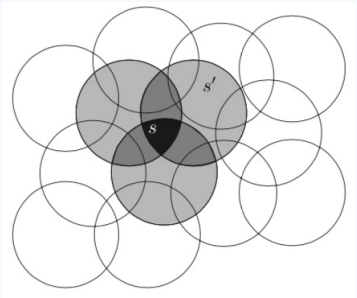
$$\mathbf{w_{t+1}}[e(s')] \leftarrow \mathbf{w_t}[e(s')] + \alpha_t \Big\{ r_t + \gamma \hat{v}(s'; \mathbf{w}) - \hat{v}(s; \mathbf{w}) \Big\} \times 0 = 0$$

We see that exactly one parameter is updated (corresponding the updated cell in tabular MC/SARSA/Q-Learning) with the same update (gradient is 1).

4 Coarse Coding

Idea

Let us assume $s \in \mathbb{R}^d$. We divide space in overlapping *balls*. We represent s by the indices of the balls to which it belongs. Example, in $\mathbb{R}^2$ with euclidean distance:



s will be represented by a vector $(0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0)$ (assuming we have defined an order on circles)

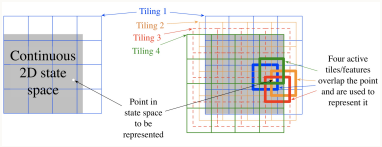
4 *Tile Coding*

A specific instance of coarse coding

tile cube (a kind of ball?) (of hypercube in higher dimension)

tiling partition of the space in cubes

representation a point in space is represented by the tile indexes to which it belongs in several tilings



4 A note on Coarse Encoding for the lab session (*TP*)

Goal: tile coding and linear function approximation for $\hat{q}$

According to what we saw:

$$\hat{q}(s, a; \mathbf{w}) = \sum_{i=1}^{d} \mathbf{w}_i^a \times x(s_i)$$

1. a parameter vector per action ($|A|$ parameters vectors)
2. one feature function x (implemented via tile coding)

In practice

To be compatible with our Tile Coding library, this becomes:

$$\hat{q}(s, a; \mathbf{w}) = \sum_{i=1}^{d} \mathbf{w}_i \times x^a(s_i)$$

1. one parameter vector only
2. a feature function x^a per action (implemented via tile coding)

Gradient becomes: $\nabla q(s, a; \mathbf{w}) = [x^a(s_1); \dots; x^a(s_n)] = x^a$