Reinforcement Learning
Lecture 7: Function approximation

Lecturer: Prof. Dr. Mathias Niepert

Institute for Artificial Intelligence
Machine Learning and Simulation Lab

University of Stuttgart
Germany

imprs-is

June 13, 2024

Outline

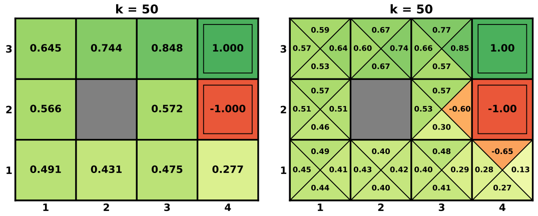
Introduction

Large state spaces

- Backgammon: 10^{20} states
- Go: 10^{170} states
- Continuous spaces:
  - inverted pendulum
  - mountain car
  - helicopter
  - ...

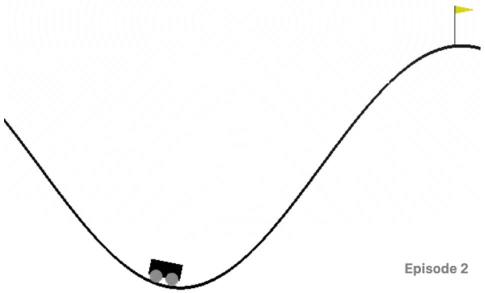
Small domains: tabular representation

- So far V and Q were just *lookup tables*:



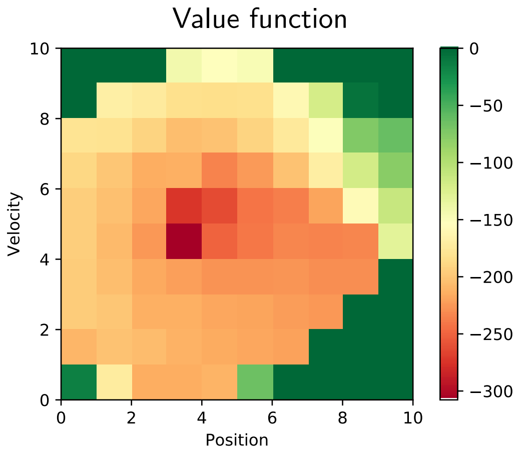
- Problems with large MDPs:
  - too many states to store in memory
  - too slow to learn
- How can we scale the tabular solution methods to arbitrarily large state/action spaces?
  - *generalize* from previous encounters of similar states (or state–action pairs)
  - **function approximation** (supervised learning)

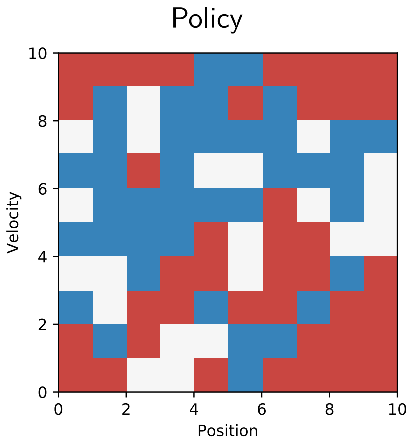
Example: mountain car



$a \in \{-1, 0, 1\}$ is the action

State aggregation in mountain car





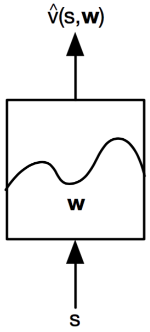
Value function approximation

Idea of value function approximation

- Parameterized functional form, with weights $\boldsymbol{w} \in \mathbb{R}^d$:

$$\hat{v}_\pi(s, \boldsymbol{w}) \approx v_\pi(s)$$

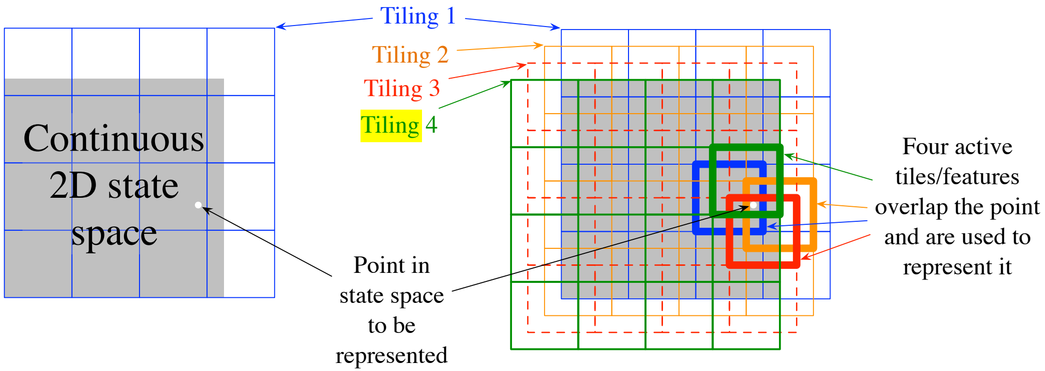
- Generally, much less weights than states $d \ll |\mathcal{S}|$
  - obvious for continuous state spaces
  - changing single weight, changes value estimate of many states
  - when one state is updated, change *generalizes* to many states
- Update $\boldsymbol{w}$ with MC or TD learning



Function approximators

- Linear combinations of features
  - **state aggregation**
  - **tile coding**
  - polynomials
  - radial basis functions (RBFs)
  - Fourier bases
  - ...
- Neural networks
- Decision trees
- Non-parametric approaches

Tile coding


Tiling 1
Tiling 2
Tiling 3
Tiling 4
Continuous 2D state space
Point in state space to be represented
Four active tiles/features overlap the point and are used to represent it

Function approximators for RL

- Differentiable function approximators, e.g.:
  - Linear combination of features
  - Neural networks
- RL specific problems:
  - non-stationary
  - non-iid data
  - bootstrapping
  - delayed targets

Random variables are *independent and identically distributed* (iid) if they each have the same probability distribution and are mutually independent

Stochastic Gradient Descent (SGD)

- Approximate value function $\hat{v}(s, \boldsymbol{w})$
  - differentiable for all $s \in \mathcal{S}$
- Weight vector $\boldsymbol{w} = (w_1, w_2, \ldots, w_d)^\top$
  - $\boldsymbol{w}_t$ weight vector at time $t = 0, 1, 2, \ldots$
- *Gradient* of $f(\boldsymbol{w})$: $\boldsymbol{\nabla} f(\boldsymbol{w}) = \left(\frac{\partial f(\boldsymbol{w})}{\partial w_1}, \frac{\partial f(\boldsymbol{w})}{\partial w_2}, \ldots, \frac{\partial f(\boldsymbol{w})}{\partial w_d}\right)^\top$
- Do gradient descent by sampling additive parts of the full gradient (i.e., each state consecutively)
- We can compute our update over smaller sets of inputs

Stochastic Gradient Descent (SGD) (part 2)

- When we **approximate** the gradient
- For example

$$\mathcal{L}(\boldsymbol{w}) = \sum_{n=1}^{N} \mathcal{L}_n(\boldsymbol{w})$$

where $\boldsymbol{w}$ are weights.

- In machine learning

$$\mathcal{L}(\boldsymbol{w}) = -\sum_{n=1}^{N} \log p(y_n \mid \boldsymbol{x}_n, \boldsymbol{w})$$

where $\boldsymbol{x}_n \in \mathbb{R}^D$ are training *inputs* and y_n are the training *targets*

- The corresponding update

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \alpha_t \sum_{n=1}^{N} (\boldsymbol{\nabla} \mathcal{L}_n)(\boldsymbol{w}_t)$$

- Often the gradient is too difficult to compute (CPU/GPU expensive)
- **Mini-batch**: random subset
  a Large: accurate but costly
  b Small: noisy but cheep

Stochastic Gradient Descent (SGD) (part 3)

- *Mean squared Value Error*: $\mathcal{L}(\boldsymbol{w}) = \sum_{s \in S} \mu(s) \left[v_\pi(s) - \hat{v}(s, \boldsymbol{w})\right]^2$
- Adjust $\boldsymbol{w}$ to reduce the error on sample $S_t \mapsto v_\pi(S_t)$:

$$\begin{aligned} \boldsymbol{w}_{t+1} &= \boldsymbol{w}_t - \frac{1}{2}\alpha_t(\boldsymbol{\nabla}\mathcal{L}_t)(\boldsymbol{w}_t) \\ &= \boldsymbol{w}_t - \frac{1}{2}\alpha_t \boldsymbol{\nabla} \underbrace{\left[v_\pi(S_t) - \hat{v}(S_t, \boldsymbol{w}_t)\right]^2}_{\textbf{squared sample error}} \\ &= \boldsymbol{w}_t + \alpha_t[v_\pi(S_t) - \hat{v}(S_t, \boldsymbol{w}_t)]\nabla\hat{v}(S_t, \boldsymbol{w}) \end{aligned}$$

- α_t is a step size parameter
- Why not use $\alpha = 1$, thus eliminating full error on sample?

Linear methods

- Special case where $\hat{v}(\cdot, \boldsymbol{w})$ is *linear* in the weights
- **Feature vector** $\boldsymbol{x}(s)$ represents state s:

$$\boldsymbol{x}(s) = \left[x_1(s), \quad x_2(s), \quad \dots, \quad x_d(s)\right]^\top$$

- Each component of $\boldsymbol{x}$ is a feature, examples:
  - distance of robot to landmarks
  - piece on a specific location on a chess board
- Value function is represented as a linear combination of features $\boldsymbol{x}(s)$:

$$\hat{v}(s, \boldsymbol{w}) = \boldsymbol{w}^\top \boldsymbol{x}(s) = \sum_{i=1}^{d} w_i x_i(s)$$

- Gradient is simply $\nabla \hat{v}(s, \boldsymbol{w}) = \boldsymbol{x}(s)$

Prediction

Prediction with function approximation

- We assumed the true value function $v_\pi(S_t)$ is known
- Substitute target U_t for $v_\pi(s)$

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha[U_t - \hat{v}(S_t, \boldsymbol{w}_t)]\nabla\hat{v}(S_t, \boldsymbol{w}_t)$$

- U_t might be a noisy or bootstrapped approximation of the true value
- **Monte Carlo:** $U_t = G_t$
- **TD(0):** $U_t = R_{t+1} + \gamma\hat{v}(S_{t+1}, \boldsymbol{w}_t)$
- **TD(λ):** $U_t = G_t^\lambda$

Monte–Carlo with function approximation

- Target is unbiased by definition:

$$\mathbb{E}[U_t|S_t = s] = \mathbb{E}[G_t|S_t = s] = v_\pi(S_t)$$

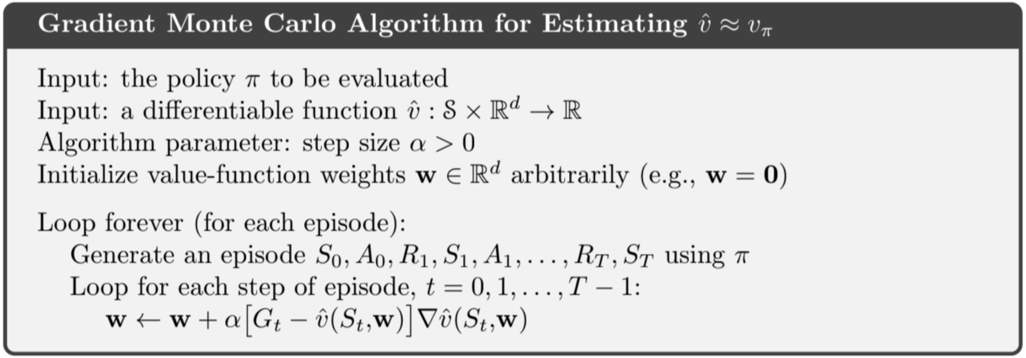
- Training data:

$$\mathcal{D} = \left\{(S_1, G_1),\ (S_2, G_2),\ \ldots,\ (S_{T-1}, G_{T-1}),\ (S_T, 0)\right\}$$

- Using SGD, $\boldsymbol{w}$ is guaranteed to converge to a *local optimum*
- MC prediction exhibits local convergence with linear and non-linear function approximation
- SGD update for sample $S_t \mapsto G_t$:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha[G_t - \hat{v}(S_t, \boldsymbol{w})]\nabla\hat{v}(S_t, \boldsymbol{w})$$

Gradient Monte Carlo Algorithm



Gradient Monte Carlo Algorithm for Estimating $\hat{v} \approx v_\pi$

```
Input: the policy $\pi$ to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S} \times \mathbb{R}^d \rightarrow \mathbb{R}$
Algorithm parameter: step size $\alpha > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop forever (for each episode):
    Generate an episode $S_0, A_0, R_1, S_1, A_1, \dots, R_T, S_T$ using $\pi$
    Loop for each step of episode, $t = 0, 1, \dots, T-1$:
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha \big[ G_t - \hat{v}(S_t, \mathbf{w}) \big] \nabla \hat{v}(S_t, \mathbf{w})$
```

TD with function approximation

- TD-target $U_t = R_{t+1} + \gamma \hat{v}(S_{t+1}, \boldsymbol{w})$ is *biased* sample of the true value $v_\pi(S_t)$
- Training data:

$$\mathcal{D} = \left\{(S_1, R_2 + \gamma \hat{v}(S_2, \boldsymbol{w})),\ (S_2, R_3 + \gamma \hat{v}(S_3, \boldsymbol{w})),\ \dots, (S_{T-1}, R_T)\right\}$$

- SGD update for sample $S_t \mapsto G_t$:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha[R_{t+1} + \gamma \hat{v}(S_{t+1}, \boldsymbol{w}) - \hat{v}(S_t, \boldsymbol{w})]\nabla \hat{v}(S_t, \boldsymbol{w})$$

- Linear TD(0):

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha[\underbrace{R_{t+1} + \gamma \boldsymbol{w}^T \boldsymbol{x}(S_{t+1})}_{U_t\text{:TD-Target}} - \underbrace{\boldsymbol{w}^T \boldsymbol{x}(S_t)}_{\hat{v}\text{:value function}}]\boldsymbol{x}(S_t)$$

Semi-Gradient TD(0) Algorithm

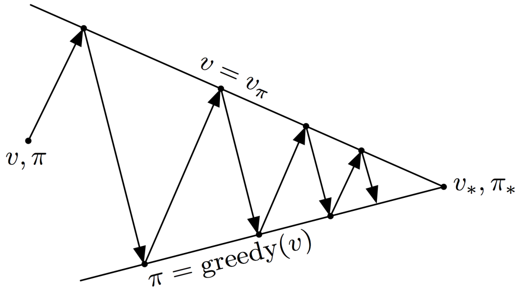
Semi-gradient TD(0) for estimating $\hat{v} \approx v_\pi$

```
Input: the policy $\pi$ to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \to \mathbb{R}$ such that $\hat{v}(\text{terminal},\cdot) = 0$
Algorithm parameter: step size $\alpha > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
    Initialize $S$
    Loop for each step of episode:
        Choose $A \sim \pi(\cdot|S)$
        Take action $A$, observe $R, S'$
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})\big]\nabla\hat{v}(S,\mathbf{w})$
        $S \leftarrow S'$
    until $S'$ is terminal
```

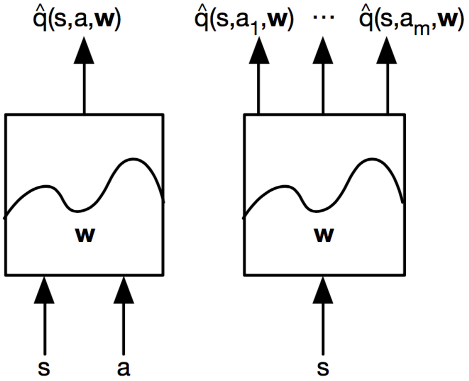
Control

Control with function approximation



- Control via generalized policy iteration (GPI):
  - *policy evaluation:* **approximate** policy evaluation: $\hat{q}(\cdot, \cdot, \boldsymbol{w}) \approx q_\pi$
  - *policy improvement:* ϵ-greedy policy improvement

Types of action–value function approximation



- Action as input: $\hat{q}(s, a, \boldsymbol{w}) \approx q_\pi(s, a)$
- Multiple action–value outputs: $\hat{\boldsymbol{q}}_a(s, \boldsymbol{w}) \approx q_\pi(s, a)$

Action-value function approximation

- Approximate action-value function $\hat{q}(s, a, \boldsymbol{w}) \approx q_\pi(s, a)$
- Linear case:

$$\hat{q}(s, a, \boldsymbol{w}) = \boldsymbol{w}^T \boldsymbol{x}(s, a) = \sum_{i=1}^{d} w_i x_i(s, a)$$

$$\nabla \hat{q}(s, a, \boldsymbol{w}) = \boldsymbol{x}(s, a)$$

- Minimize squared error on samples $S_t, A_t \mapsto q_\pi$: $\left[q_\pi - \hat{q}(S_t, A_t, \boldsymbol{w})\right]^2$
- Use SGD to find local minimum:

$$\begin{aligned} \boldsymbol{w}_{t+1} &= \boldsymbol{w}_t - \frac{1}{2}\alpha \nabla [q_\pi(S_t, A_t) - \hat{q}(S_t, A_t, \boldsymbol{w}_t)]^2 \\ &= \boldsymbol{w}_t + \alpha [q_\pi(S_t, A_t) - \hat{q}(S_t, A_t, \boldsymbol{w}_t)] \nabla \hat{q}(S_t, A_t, \boldsymbol{w}) \end{aligned}$$

Control with function approximation

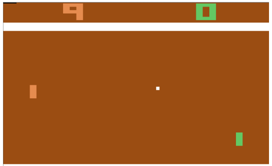
- Again, we must substitute target U_t for true action-value $q_\pi(s, a)$:

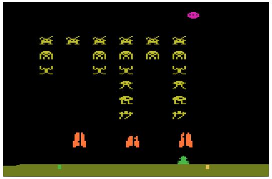
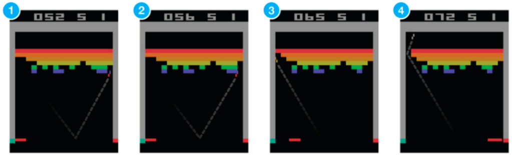
$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha[U_t - \hat{q}(S_t, A_t, \boldsymbol{w}_t)]\nabla\hat{q}(S_t, A_t, \boldsymbol{w})$$

- **Monte Carlo:** $U_t = G_t$
- **One-step Sarsa:** $U_t = R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}, \boldsymbol{w})$

Deep Reinforcement Learning

Recent successes of deep reinforcement learning



Deep (*supervised*) learning

- **Deep representation** is a composition of *many* functions

$$x \xrightarrow[\boldsymbol{w}_1]{} h_1 \xrightarrow[\boldsymbol{w}_2]{} h_2 \xrightarrow[\boldsymbol{w}_3]{} \dots \xrightarrow[\boldsymbol{w}_n]{} h_n \xrightarrow[\boldsymbol{w}_{n+1}]{} y$$

- Linear transformation and non-linear **activation functions** h_k
- Weight sharing
  - **Recurrent neural networks**: across time steps
  - **Convolutional neural networks**: across spatial (or temporal) regions
- Weights $\boldsymbol{w}$ optimized by stochastic gradient descent (SGD)
- Powerful **function approximation** and **representation learning**
  - finds compact low-dimensional representation (*features*)
- State-of-the-art for image, text and audio

Naive deep Q-learning

- Q-learning update rule:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(r + \gamma \max_a Q(s',a) - Q(s,a)\right)$$

- Q is represented by a neural network with weights $\boldsymbol{w}$: $\hat{q}(s,a,\boldsymbol{w})$
- Loss is the mean-squared TD-error:

$$\mathcal{L}(\boldsymbol{w}) = \mathbb{E}\left[\left(r + \gamma \max_a \hat{q}(s',a,\boldsymbol{w}) - \hat{q}(s,a,\boldsymbol{w}])\right)^2\right]$$

- Minimize sample errors with SGD

Stability

Naive Q-learning with neural networks oscillates or diverges:

1. Data is non i.i.d!
   - trajectories
   - samples are correlated (generated by interaction)
2. Policy changes rapidly with slight changes to Q-values
   - policy may oscillate
3. Reward range is unknown
   - gradients can be large
   - instabilities during back-propagation
4. Maximization bias

Deep Q-networks (DQN)

Deep Q-networks (DQN) address instabilities through:

- **Experience replay**
  - store transitions (S_t, A_t, R_t, S_{t+1})
  - sample random mini-batches
  - removes correlation, restores i.i.d. property
- **Target network**
  - second q network (second set of parameters)
  - fixed parameters in target network
  - periodically update target network parameters
- **Reward clipping/normalization**
  - clip rewards to $r \in [-1, 1]$
  - batch normalization

DQN in Atari

"End-to-end" learning:

- *state:* stack of 4 frames, raw pixels
- *action:* joystick commands (18 discrete actions)
- *reward:* change in score

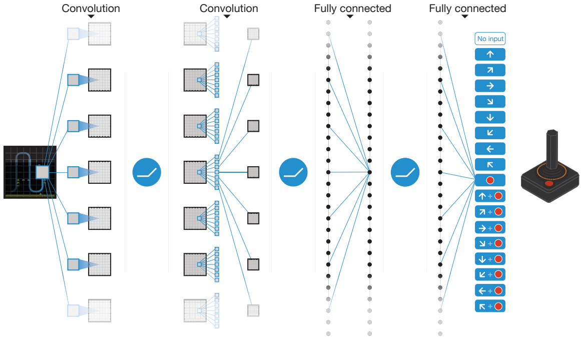


image from *Human-level control through deep reinforcement learning* (Google Deepmind / Nature)

DQN in Atari

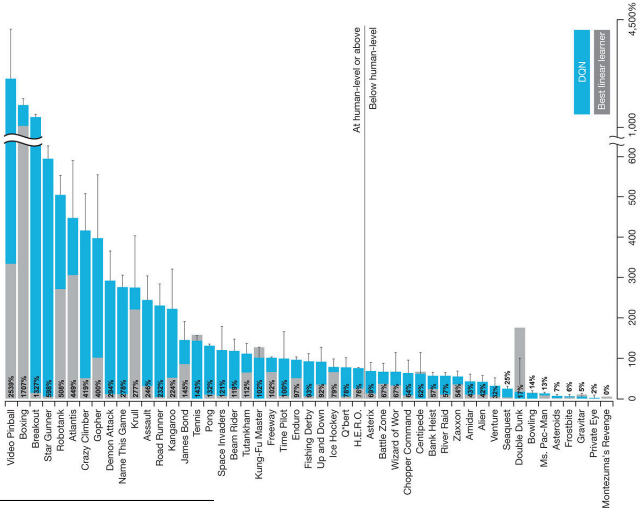


image from *Human-level control through deep reinforcement learning* (Google Deepmind / Nature)

Summary

- When state spaces are **too large** it is not possible to use tables
- Instead we can use **function approximation** to model v_π and q_π
- Many function models exist in the literature, the most popular and general are Neural Networks.
- Stochastic Gradient Descent (SGD) is used to optimize the **Value Function Error**
- Value function approximation is used for prediction
- ϵ-greedy or other covering policies can be used for improvement
- When using a deep-neural network it is important to train in batches using **experience replay** and other techniques to regularize the problem