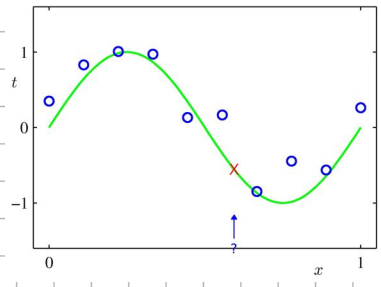
Regression



No Free Lunch Theorem

have to make some assumptions about structure of the data, otherwise generalization is impossible

Model

$$y(x, \vec{w}) = \vec{w}^T \vec{\phi}(\vec{x}), \quad \vec{\phi}(\vec{x}) = (\phi_0(\vec{x}), \dots, \phi_m(\vec{x}))^T$$

Fitting

other approaches are also possible

$$E(\vec{w}) = \frac{1}{N} \sum_n e_n(\vec{w}) = \frac{1}{N} \sum_{n=1}^{N} \frac{1}{2} (y(x_n, \vec{w}) - t_n)^2 \quad (MSE)$$

$$\frac{dE}{d\vec{w}} = \left[\frac{\partial E}{\partial w_0}, \dots, \frac{\partial E}{\partial w_m} \right]$$

$$= \frac{1}{N} \sum_{n=1}^{N} \left(\vec{w}^T \vec{\phi}(\vec{x}_n) \vec{\phi}(\vec{x}_n)^T - t_n \vec{\phi}(\vec{x}_n)^T \right)$$

$$= \frac{1}{N} \left(\vec{w}^T \sum_{n=1}^{N} \vec{\phi}(\vec{x}_n) \vec{\phi}(\vec{x}_n)^T - \sum_{n=1}^{N} t_n \vec{\phi}(\vec{x}_n)^T \right)$$

$$= \frac{1}{N} \left(\vec{w}^T \Phi^T \Phi - \vec{t}^T \Phi \right)$$

$$\stackrel{!}{=} \vec{0}_m^T$$

$$\Rightarrow \vec{w}^T \Phi^T \Phi = \vec{t}^T \Phi$$

nonlinear in the inputs, but linear in parameters $\vec{w}$

$$\vec{w}^T = \vec{t}^T \Phi (\Phi^T \Phi)^{-1}$$

$$\Rightarrow \vec{w}_{min} = \left(\vec{t}^T \Phi (\Phi^T \Phi)^{-1} \right)^T = \left((\Phi^T \Phi)^{-1} \right)^T \Phi^T (\vec{t}^T)^T = \left((\Phi^T \Phi)^T \right)^{-1} \Phi^T \vec{t} = (\Phi^T \Phi)^{-1} \Phi^T \vec{t}$$

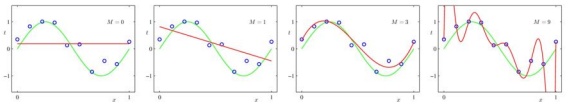
if Φ is square & invertible, it coincides with the regular inverse

Moore-Penrose Pseudo-Inverse of Φ

Design Matrix

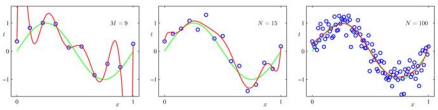
$$\Phi = \begin{bmatrix} — & \vec{\phi}_1^T & — \\ & \vdots & \\ — & \vec{\phi}_N^T & — \end{bmatrix} = \begin{bmatrix} \phi_0(\vec{x}_1) & \cdots & \phi_m(\vec{x}_1) \\ \vdots & & \vdots \\ \phi_0(\vec{x}_N) & \cdots & \phi_m(\vec{x}_N) \end{bmatrix}$$

Overfitting



overfitted polynomial yields bad predictions for new data
tuned to random noise present in data
has very large coefficients

Order	M=1	M=3	M=9
w_0	0.82	0.31	0.35
w_1	-1.27	7.99	232.37
w_2		-25.43	-5321.83
w_3		17.37	48568.31
w_4			-231639.30
w_5			640042.26
w_6			-1061800.52
w_7			1042400.18
w_8			-557682.99
w_9			125201.43



ways to reduce:
- more training data (easiest way)
  - artificially create e.g. injecting random noise to samples, domain-specific transformations
- limiting number of parameters
- constrain parameters by penalizing their absolute value
  - → in case of linear regression e.g. $\tilde{E}(\vec{w}) = E(\vec{w}) + \frac{\lambda}{2} \|\vec{w}\|_2^2$ (L_2 regularization)
  - → closed form solution (Ridge regression): $\vec{w}_{min} = (\lambda \underline{I} + \Phi^T \Phi)^{-1} \Phi^T \vec{t}$

Underfitting

model does not reflect structure of training data

Regularization

changing training criterion of a system to improve generalization

Features

choice of features $\phi_j(x)$ often depends on knowledge of the task
preprocessing data in a way to make relevant information accessible and remove irrelevant content
reduce dimensionality of data

Curse of Dimensionality

number of model parameters rises exponentially with input dimension

example: polynomial fitting we have to take into account all possible products between variables, up to desired order:

$$y(\vec{x}, \vec{w}) = w_0 + \sum_i w_i^{(i)} x^{(i)} + \sum_{i_1, i_2 \atop i_1 \leq i_2} w_2^{(i_1, i_2)} x^{(i_1)} x^{(i_2)} + \dots + \sum_{i_1, \dots, i_M \atop i_1 \leq \dots \leq i_M} w_M^{(i_1, \dots, i_M)} x^{(i_1)} \dots x^{(i_M)}$$

we need to estimate many coefficients, which may be inefficient and can cause severe overfitting

→ number of model parameters rises exponentially with input dimension

ways to tackle the problem:
- real data often lies in or subspace of lower dimensionality
- use dimensionality reduction techniques on data
- define suitable set of features: $\phi_k(x_1, \dots, x_L)$ where number of features can be much smaller than L^M
- neural networks can intrinsically deal with high-dimensional input and often do not require sophisticated features

Intermezzo: #Coefficients in Polynomial (= #monomials of L variables and degree $\leq M$)

stars and bars argument:
first L bins correspond to the powers of $x_1, \dots, x_L$ and ignore last bin
#stars in first L bins is $\leq M$
→ this produces all possible monomials of L variables and degree $\leq M$

$$\binom{L+M}{M}$$

Classification

discriminant function: $C = f(\phi)$
probabilistic modelling: $p(C_k|\phi) = f(\phi, C_k)$ — easy

k-NN

non parametric (works directly on data, as opposed to learning parameters of a model)
majority vote determines class assignment
a possible extension: weigh influence of kNN (e.g. by distance)
Problems:
- requires to store all training data and compute distances between test sample and all training samples
- does not discover structure
- sensitive to outliers
- problematic for high-dimensional feature spaces (Curse of Dimensionality: samples do not cover space well)
- difficult to define suitable notion of closeness

Linear Classifier

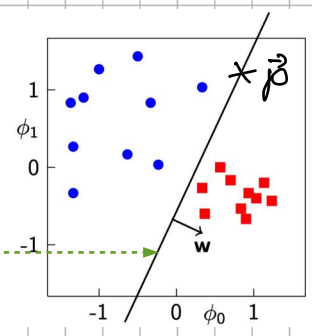
parametric (structure of data gets summarized by a set of parameters, independent of number of training samples)

discriminant function:

$$y(\vec{\phi}) = \vec{w}\,\vec{\phi} + w_0 \quad \text{with} \quad C(\vec{\phi}) = \begin{cases} C_0 & \text{if } y(\vec{\phi}) \geq 0 \\ C_1 & \text{if } y(\vec{\phi}) < 0 \end{cases}$$

$\vec{w}$ is called the weight vector, w_0 the bias

decision boundary: hyperplane with equation $\vec{w}^T\vec{\phi} + w_0 = 0$



$\vec{w}$ is orthogonal to the decision boundary

distance of any point $\vec{\phi}$ to decision boundary:

$$(\vec{\phi} - \vec{p}) \cdot \frac{\vec{w}}{\|\vec{w}\|} = \frac{\vec{w}^T\vec{\phi} - \vec{w}^T\vec{p}}{\|\vec{w}\|} = \frac{\vec{w}^T\vec{\phi} - \vec{w}^T\vec{p} - w_0 + w_0}{\|\vec{w}\|} = \frac{\vec{w}^T\vec{\phi} + w_0}{\|\vec{w}\|} = \frac{y(\vec{\phi})}{\|\vec{w}\|}$$

just like for regression, we can instead define suitable nonlinear features

prone to drastic underfitting

multiple classes:
use K linear functions $y_k(\vec{\phi}) = \vec{w}_k^T\vec{\phi} + w_{k0}$ and assign $\vec{\phi}(\vec{x})$ to class C_k if $y_k(\vec{\phi}(\vec{x})) > y_j(\vec{\phi}(\vec{x})) \; \forall j \neq k$

Logistic Regression

$$\tilde{y}(\vec{\phi}, \vec{w}) = \vec{w}^T\vec{\phi} + w_0$$

$$y(\vec{\phi}, \vec{w}) = \sigma(\tilde{y}(\vec{\phi}, \vec{w})) = \frac{1}{1 + e^{\vec{w}^T\vec{\phi} + w_0}}$$

$$p(C_1|\vec{\phi}) = y(\vec{\phi}), \quad p(C_0|\vec{\phi}) = 1 - y(\vec{\phi})$$

$$E(\vec{w}) = -\log L(\vec{w}) = -\log\left(\prod_{n=1}^{N} y_n^{t_n}(1-y_n)^{1-t_n}\right) = -\sum_{n=1}^{N}(t_n \log y_n + (1-t_n)\log(1-y_n)) \rightarrow \text{minimize}$$

no closed form solution

if C_0 and C_1 are linearly separable → overfitting

Foundations

assumption: data is described by a prob. distr. $P(\vec{x}, \vec{t})$ (→ unknown), $\vec{x} \in \mathbb{R}^M$, $\vec{t} \in \mathbb{R}^N$

let $\vec{y}(\vec{x})$ be a predictor for $\vec{t}$, $L(\vec{t}, \vec{y}(\vec{x}))$ any loss

True Risk

$$R(\vec{y}) := \mathbb{E}_{P(\vec{x},\vec{t})} L(\vec{t}, \vec{y}(\vec{x})) = \int L(\vec{t}, \vec{y}(\vec{x}))\, dP(\vec{x}, \vec{t})$$

goal of ML is to find predictor that minimizes the risk:

$$\vec{y}^* = \underset{\vec{y}: \mathbb{R}^M \to \mathbb{R}^N}{\operatorname{argmin}} R(\vec{y})$$

Irreducible Error

stems from fact that input does not completely determine the target

Empirical Risk

$$R(\vec{y}) \approx \frac{1}{N} \sum_{n=1}^{N} L(\vec{t}_n, \vec{y}(\vec{x}_n)) =: R_{emp}(\vec{y}) \quad \text{with dataset } D = \{(\vec{x}_n, \vec{t}_n)\}_{n=1\dots N}$$

for pathological functions $\vec{y}$, the empirical risk does not reflect true risk

↳ restrict predictors we take into account to a subset of functions from $\mathbb{R}^M$ to $\mathbb{R}^N$:

Hypothesis Class H

we hope this helps us to obtain a predictor whose R_{emp} (on D_{Train}) reflects R_{true} (on D_{Test}):

$$\vec{y}_H^* = \underset{\vec{y} \in H}{\operatorname{argmin}} R(\vec{y}), \quad \vec{y}_H^{emp} = \underset{\vec{y} \in H}{\operatorname{argmin}} R_{emp}(\vec{y})$$

in almost all cases, exactly computing the optimal predictor in H is intractable
→ needs to be approximated

all ML algorithms aim at optimally defining the hypothesis class H, such that efficient optimization over a wide range of functions is possible, while retaining generalization ability

Difference between R and R_{emp}

many bounds have been proposed, typically of the form

$$\forall \vec{y} \in H \quad R(\vec{y}) - R_{emp}(\vec{y}) < O\left(\sqrt{\frac{cap(H)}{N}}\right)$$

Kernel Methods

Linear methods in a feature space
→ yet they offer a different view of both linear models, and the concept of features

Consider Linear Regression with L_2 regularization:

$$E(\vec{w}) = \sum_{n=1}^{N} \frac{1}{2}(\vec{w}^T\vec{\phi}_n - t_n)^2 + \frac{\lambda}{2}\|\vec{w}\|_2^2 \quad \text{where} \quad \vec{\phi}_n = \vec{\phi}(\vec{x}_n)$$

Gradient w.r.t. $\vec{w}$ is $\frac{\partial}{\partial \vec{w}} E = \sum_{n=1}^{N} (\vec{w}^T\vec{\phi}_n - t_n)\vec{\phi}_n + \lambda\vec{w}$ and setting it to 0 yields

$$\vec{w} = \sum_{n=1}^{N} \underbrace{-\frac{1}{\lambda}(\vec{w}^T\vec{\phi}_n - t_n)}_{a_n}\vec{\phi}_n = \underline{\Phi}^T\vec{a}$$

$$\underline{\Phi} = \begin{bmatrix} — \vec{\phi}_1^T — \\ \vdots \\ — \vec{\phi}_N^T — \end{bmatrix} = \begin{bmatrix} \phi_0(\vec{x}_1) & \cdots & \phi_m(\vec{x}_1) \\ \vdots & & \vdots \\ \phi_0(\vec{x}_N) & \cdots & \phi_m(\vec{x}_N) \end{bmatrix}$$

We see that not only the model is linear, but also that $\vec{w}$ is a linear combination of the (features of the) training data

Dual Representation

the observation that $\vec{w}$ and features $\underline{\Phi}$ have such a simple relationship leads us to the idea of expressing the ridge regression problem in terms of the finitely many data points
→ such a reformulation, called dual representation, is possible for a variety of linear models

substituting $\vec{w} = \underline{\Phi}^T\vec{a}$ into $E(\vec{w})$ yields:

$$\begin{aligned}
E(\vec{a}) &= \frac{1}{2}\sum_{n=1}^{N}(\vec{\phi}_n^T\vec{w} - t_n)^2 + \frac{\lambda}{2}\|\vec{w}\|_2^2 \\
&= \frac{1}{2}\begin{bmatrix} \vec{\phi}_1^T\vec{w} - t_1 \\ \vdots \\ \vec{\phi}_N^T\vec{w} - t_N \end{bmatrix}^T \begin{bmatrix} \vec{\phi}_1^T\vec{w} - t_1 \\ \vdots \\ \vec{\phi}_N^T\vec{w} - t_N \end{bmatrix} + \frac{\lambda}{2}\vec{w}^T\vec{w} \\
&= \frac{1}{2}(\underline{\Phi}\vec{w} - \vec{t})^T(\underline{\Phi}\vec{w} - \vec{t}) + \frac{\lambda}{2}\vec{w}^T\vec{w} \\
&= \frac{1}{2}(\vec{w}^T\underline{\Phi}^T - \vec{t}^T)(\underline{\Phi}\vec{w} - \vec{t}) + \frac{\lambda}{2}\vec{w}^T\vec{w} \\
&= \frac{1}{2}(\vec{w}^T\underline{\Phi}^T\underline{\Phi}\vec{w} - \vec{w}^T\underline{\Phi}^T\vec{t} - \vec{t}^T\underline{\Phi}\vec{w} + \vec{t}^T\vec{t}) + \frac{\lambda}{2}\vec{w}^T\vec{w} \\
&= \frac{1}{2}(\vec{a}^T\underline{\Phi}\underline{\Phi}^T\underline{\Phi}\underline{\Phi}^T\vec{a} - \vec{a}\underline{\Phi}\underline{\Phi}^T\vec{t} - \vec{t}^T\underline{\Phi}\underline{\Phi}^T\vec{a} + \vec{t}^T\vec{t}) + \frac{\lambda}{2}\vec{a}^T\underline{\Phi}\underline{\Phi}^T\vec{a} \\
&= \frac{1}{2}\vec{a}^T\underline{\Phi}\underline{\Phi}^T\underline{\Phi}\underline{\Phi}^T\vec{a} - \vec{a}\underline{\Phi}\underline{\Phi}^T\vec{t} + \frac{1}{2}\vec{t}^T\vec{t} + \frac{\lambda}{2}\vec{a}^T\underline{\Phi}\underline{\Phi}^T\vec{a}
\end{aligned}$$

Kernel function

$$k(\vec{x}_n, \vec{x}_m) := \vec{\phi}(\vec{x}_n)^T\vec{\phi}(\vec{x}_m) = \langle\vec{\phi}(\vec{x}_n), \vec{\phi}(\vec{x}_m)\rangle$$

Gram matrix

$$\underline{K} := \underline{\Phi}\underline{\Phi}^T = (k(\vec{x}_n, \vec{x}_m))_{n,m=1,\ldots,N} = \begin{bmatrix} — \vec{\phi}_1^T — \\ \vdots \\ — \vec{\phi}_N^T — \end{bmatrix}\begin{bmatrix} | & & | \\ \vec{\phi}_1 & \cdots & \vec{\phi}_N \\ | & & | \end{bmatrix} \quad \text{s.t.}$$

→ allows us to express the error in the dual representation:

$$E(\vec{a}) = \frac{1}{2}\vec{a}^T\underline{K}\underline{K}\vec{a} - \vec{a}\underline{K}\vec{t} + \frac{1}{2}\vec{t}^T\vec{t} + \frac{\lambda}{2}\vec{a}^T\underline{K}\vec{a}$$

Solving in Dual Representation

setting the gradient of $E(\vec{a})$ wrt. $\vec{a}$ to zero, one obtains the following minimum of the error function:

$$\vec{a} = (\underline{K} + \lambda \underline{I}_N)^{-1} \vec{t} \qquad \rightarrow \text{inverting an } N \times N \text{ matrix}$$

plugging this solution into the original model, the prediction function is given by

$$y(\vec{x}) = \vec{k}(\vec{x})^T (\underline{K} + \lambda \underline{I}_N)^{-1} \vec{t},$$

with $\vec{k}(\vec{x}) = (k(\vec{x}_n, \vec{x}))_{n=1,\dots,N}$,

The Role of the Kernel

the solution is found by inverting a $N \times N$ matrix, whereas in the original formulation, we have inverted an $M \times M$ matrix

advantage of dual formulation: we never use the feature vector ϕ directly

input data appears only inside the kernel function $k(\vec{x}, \vec{y})$

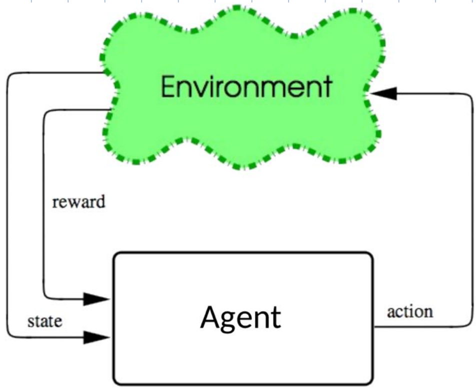
→ allows us to implicitly use very high-dimensional (possibly infinite-dimensional) feature spaces!

Example: Quadratic Features

$$\phi(\vec{x})^T = [1, \sqrt{2}x_1, \dots, \sqrt{2}x_K, x_1^2, \dots, x_K^2, \sqrt{2}x_1x_2, \dots, \sqrt{2}x_1x_K, \sqrt{2}x_2x_3, \dots, \sqrt{2}x_2x_K, \dots, \sqrt{2}x_{K-1}x_K]$$

$$\begin{aligned}
k(\vec{x}, \vec{y}) &= \vec{\phi}(\vec{x})^T \vec{\phi}(\vec{y}) \\
&= 1 + 2x_1y_1 + \dots + 2x_Ky_K + x_1^2y_1^2 + \dots + x_K^2y_K^2 + 2x_1x_2y_1y_2 + \dots + 2x_{K-1}x_K y_{K-1}y_K \\
&= 1 + 2\sum_{i=1}^{K} x_iy_i + \sum_{i=1}^{K} x_i^2y_i^2 + 2\sum_{1 \le i < j \le K} x_ix_jy_iy_j \\
&= 1 + 2\vec{x}^T\vec{y} + \sum_{i=1}^{K} x_iy_ix_iy_i + 2\sum_{1 \le i < j \le K} x_iy_i\, x_jy_j \\
&= 1 + 2\vec{x}^T\vec{y} + \left(\sum_{i=1}^{K} x_iy_i\right)^2 \\
&= 1 + 2\vec{x}^T\vec{y} + \vec{x}^T\vec{y}^{\,2} \\
&= (1 + \vec{x}^T\vec{y})^2
\end{aligned}$$

Reinforcement Learning



agent takes **actions** in some environment and collects scalar **rewards**
goal of **agent**: maximize **return** (cumulative reward)

differs from SL:
- → learn actions instead of predictions
- → no fixed dataset, data is generated by interacting with environment
- → no explicit error correction by reward
- → rewards are asynchronous!!!, not clear which action led to which reward
- → inherently sequential, focus on online capabilities

often only partial information available (full state of environment is unknown)

Definitions:

S_t — state of environment at time t
A_t — action taken at the time t, after seeing S_t
R_t — reward after taking A_t

better to give reward only for winning game, because figure

$$\ldots \xrightarrow{A_{t-1}} S_t \xrightarrow{A_t} S_{t+1} \xrightarrow{A_{t+1}} S_{t+2} \xrightarrow{A_{t+2}} \ldots$$
(with R_{t-1} above S_t, R_t above S_{t+1}, R_{t+1} above S_{t+2})

chess ex:
- state: board with figures on it
- action: single move of figure
- reward: usually no intermediate rewards, because it would incentivize agent to the wrong thing, usually only +1 for winning and −1 for losing

$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{H} \gamma^k R_{t+k+1}$ — cumulative future reward (**return**)

γ — **discount factor**, $0 < \gamma \leq 1$
- → $\gamma = 1$: agent is **farsighted** (all rewards matter equally)
- → $\gamma < 1$: agent is **shortsighted**

which one makes more sense? depends on environment!
- in chess probably makes more sense to set $\gamma = 1$, because doesn't matter when game is won, you just have to win
- robot in labyrinth makes a lot of sense to set $\gamma < 1$, because otherwise agent may just aimlessly wander around endlessly, because it got no incentive to reach the goal quickly

H — **horizon**, may be finite or infinite
→ how far agent looks into the future

we assume $H = \infty$, finite horizon is approximated by setting $\gamma < 1$

π — **policy**, tells agent for any state what action to take
→ defines behaviour of agent
- → **stochastic**: $\pi(s, a) \xrightarrow{\text{we can calculate conditional probability}} \pi(s \mid a)$
- → **deterministic**: $a = \pi(s)$

both possible, both used
→ could also be one in training and other when employing

↳ we can reformulate objective of RL: learn policy which maximizes (expected*) return!
→ we learn a function

expected value
→ in practice we run agent in some way, collect samples and then average over samples

* environment often stochastic (i.e. same action may lead to different successor states and rewards)
policy may also be stochastic
→ return may also be stochastic and we are interested in expected value

choosing an action may lead to (pos or neg) reward only much later in the future
→ **credit assignment problem**: how much credit should each action in the sequence of actions get for the outcome?
→ no explicit error correction: reward evaluates actions, but does not give instructions

excurs: how RL can be used to simulate SL / how RL is a superset of SL
sequence could contain just 1 step (episodes of 1 timestep)
state: image, action: pick class
wether this is a good idea is a different question

model
- set of states
- ways in which actions lead to state transitions → stochastic / deterministic
- rewards which are obtained in each state, or state-action pair

model-based reinforcement learning
- only possible if above information is perfectly known and set of states and actions is finite
  - → ex robot: state only partially observable → does not know what is behind
  - ↳ i.e. set of states (and actions) finite and observable
- create full map* of optimal actions in each state
- plan actions which lead to high return
- → table of state and actions and then plan how to do best

*:

s	a	
1	1	good
	2	good
	3	bad
	4	⋮
2	1	
	2	
	3	
	4	

in chess this would be finite but the number of states is already very large (would never be able to put in memory)

however, often:
- number of states very large (e.g. chess) or unlimited
- state cannot be completely observed (e.g. robot)
- even if we know states, but we do not know about expected rewards, i.e. we don't understand the reward function

→ no full map from states to actions → model-free RL

model-free reinforcement learning
- focus directly on learning good action from partial observation of the state
- generalize to unseen states / observations of states

Model-based

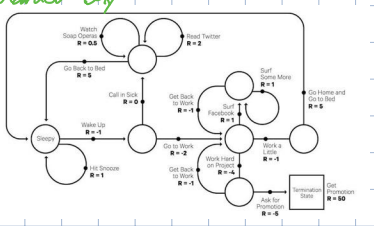
finite number of states $\{s_1, \dots, s_N\}$ and we can exactly observe the current state s_n at time step t: $S_t = s_n$

finite set of actions $\{a_1, \dots, a_m\}$
→ each action causes a change of state, which may be probabilistic or deterministic

we know the distribution of rewards in a particular state, or for a particular state-action pair
→ also rewards may be probabilistic!

in particular, the whole model has a finite description

Markov Day



formalized as a Markov Decision Process (MDP)
→ state transitions have the Markov property: $P(S_{t+1} = s' \mid S_t, S_{t-1}, \dots, S_1) = P(S_{t+1} = s' \mid S_t)$
→ also reward only depends on current state and current action

formally, an MDP is a tuple $(S, A, P^a_{ss'}, R^a_{ss'})$, where

MDP

$S = \{s_1, \dots, s_N\}$
$A = \{a_1, \dots, a_m\}$
$P^a_{ss'} = P(S_{t+1} \mid S_t = s, A_t = a)$ are the state transition probabilities
$R^a_{ss'} = E[R_{t+1} \mid S_{t+1} = s', S_t = s, A_t = a]$ is the reward function

we also include the discount factor γ in this definition

value function

$$V^\pi(s) = E[G_t \mid S_t = s] \quad \text{where} \quad G_t = \sum_{k=0}^{H} \gamma^k R_{t+k+1}$$

for $H = \infty$: $G_t = R_{t+1} + \gamma G_{t+1}$

reward the agent can expect in future if it continues with its policy from state s
→ indicates how "good" it is to be in a given state, given a policy π
→ we assume a stationary environment here

depends on state and policy
→ in a chess a state depends on what you are going to do

action-value function

$$Q^\pi(s, a) = E[G_t \mid S_t = s, A_t = a]$$

expected return if agent chooses action a in state s, and then continues to follow policy π

for a given policy, we can just average over all possible actions that might be taken to calculate $V^\pi(s)$ from $Q^\pi(s, a)$

$$V^\pi(s) = \sum_{a \in A} \pi(a \mid s)\, Q^\pi(s, a)$$

my guess: $V^\pi(s) = \sum_{i=1}^{m} \pi(a_i \mid s)\, Q^\pi(s, a_i)$?

likewise, by summing over all possible successor states s' that can be reached by taken action a, we obtain

guess 1: $Q^\pi(s,a) = \sum_{s' \in S} P^a_{ss'} V^\pi(s')$ summing reward from current state and...

$$\begin{aligned}
Q^\pi(s, a) &= E(G_t \mid S_t = s, A_t = a) \\
&= \sum_{s' \in S} P^a_{ss'} E[G_t \mid S_{t+1} = s', S_t = s, A_t = a] \\
&= \sum_{s' \in S} P^a_{ss'} E\left[R_{t+1} + \sum_{k=1}^{H} \gamma^k R_{t+k+1} \mid S_{t+1} = s', S_t = s, A_t = a\right] \\
&= \sum_{s' \in S} P^a_{ss'} \left(R^a_{ss'} + E\left[\sum_{k=0}^{H-1} \gamma^{k+1} R_{t+k+2} \mid S_{t+1} = s'\right]\right) \\
&= \sum_{s' \in S} P^a_{ss'} \left(R^a_{ss'} + \gamma E\left[\sum_{k=0}^{H-1} \gamma^k R_{(t+1)+k+1} \mid S_{t+1} = s'\right]\right) \\
&\overset{H=\infty}{=} \sum_{s' \in S} P^a_{ss'} \left(R^a_{ss'} + \gamma E[G_{t+1} \mid S_{t+1} = s']\right) \\
&= \sum_{s' \in S} P^a_{ss'} \left(R^a_{ss'} + \gamma V^\pi(s')\right)
\end{aligned}$$

?

if environment is deterministic, there would only be one possible successor state s' if we take action a in state s

note that this is only exact for infinite horizon ($H = \infty$)

→ $H < \infty$ is rarely used (see right)
normally $H = \infty$ with discount factor, which means don't have to look at 1000 steps into the future, because with e.g. $\gamma = 0.9$, the possible rewards after 20-30 steps are so small we can ignore them

Bellman equation

depends on policy!

plugging equations into each other yields recursive relation

→ foundation for a lot of algorithms in RL

→ memorize !!!!!

$$V^\pi(s) = \sum_{a \in A} \pi(a|s) \sum_{s' \in S} P^a_{ss'} \left(R^a_{ss'} + \gamma V^\pi(s') \right)$$

Bellman equation for value function

$$Q^\pi(s,a) = \sum_{s' \in S} P^a_{ss'} \left(R^a_{ss'} + \gamma \sum_{a' \in A} \pi(a|s') Q^\pi(s',a') \right)$$

→ value of state s is fundamentally a sum of the value of all the successor states plus the reward for taking the step $s \to s'$ taking into account the probabilities of the actions and successor states

if π, γ, R, P are fixed, it can be shown, that the true value function of the states is the only solution for the Bellman equation
↳ across all states, i.e. if we have N states, we have N equations to solve

agent can now quantify properties of environment, and the quality of its policy

→ importantly, agent can also use V and/or Q to improve the policy!
which is the goal of RL

greedy policy

assume Q (or V) are given (and finite, i.e. we have a table), then we can derive a policy as follows:

$$\pi(s) = \arg\max_a Q(s,a) \overset{?}{=} \arg\max_a \sum P^a_{ss'} V(s')$$ nc, because $R^a_{ss'}$ is missing

more generally
- ϵ-greedy policy: select greedy action $1-\epsilon$ of the time and some other random action ϵ of the time
- stochastic policy: select actions probabilistically (e.g. following the relative values of Q)

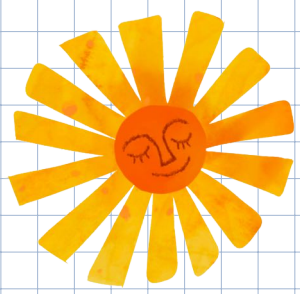
→ interesting for learning (exploration)
↳ exploitation, on the other hand, would mean always taking best action, as far as I've learned them, so I am exploiting my knowledge

this does not solve the RL objective, because V and Q depend on the policy !!!

optimal (action-)value function

$$V^*(s) = \max_\pi V^\pi(s)$$
$$Q^*(s,a) = \max_\pi Q^\pi(s,a)$$

maximize for each state



V^* is the value function of a policy, so it follows of course the Bellman equation
↳ tilde since it is the value function of the best policy (i.e. I always choose the best? action), I can also write it like this:

$$V^*(s) = \max_{a \in A} \sum_{s' \in S} P^a_{ss'} \left(R^a_{ss'} + \gamma V^*(s') \right)$$

what is beautiful here: no policy mentioned, only depends on model

→ if we find way to compute this optimal value function, we can derive a policy from it and as it turns out, that will be the optimal policy

A+

if we can compute V^*, it is only to be worthwhile → of course, we still need to approximate V^*

in a nutshell: optimal value function is value function of optimal policy, but can be written in a form that depends only on model

Dynamic Programming

→ can be used if we have a model with finite set of states, finite set of actions, we know the probabilities of state transitions, we know the reward function

Policy Iteration

start with random policy, then iteratively repeat 2 steps → Policy Evaluation: compute value function of current policy
→ Policy Improvement: improve policy wrt current value function
→ converges to optimal policy

Policy Evaluation

compute V iteratively for given π using Bellman equation as update rule: $$V_k(s) \leftarrow \sum_{a \in A} \pi(a|s) \sum_{s' \in S} P^a_{ss'} \left(R^a_{ss'} + \gamma V_{k-1}(s')\right)$$

fixed point iteration
converges to true value function of the policy, because it is the only fixed point to the set of equations

Policy Improvement Theorem

let π and π' be deterministic policies such that $Q^\pi(s, \pi'(s)) \geq V^\pi(s)$ $\forall s \in S$ ⇒ then π' is better than π and $V^{\pi'}(s) \geq V^\pi(s)$ $\forall s \in S$
furthermore if inequality on left is strict, the second one is strict as well

Policy Improvement

derive new policy using information from old policy and model: $$\pi'(s) = \underset{a \in A}{\operatorname{argmax}} \sum_{s' \in S} P^a_{ss'} \left(R^a_{ss'} + \gamma V^\pi(s')\right)$$

at each state, select action which appears best
due to PIT, new policy is strictly better than old one, unless optimal policy has already been found

in finite MDP only finitely many deterministic policies → algorithm guaranteed to converge (but combersome for large state spaces)

Value Iteration

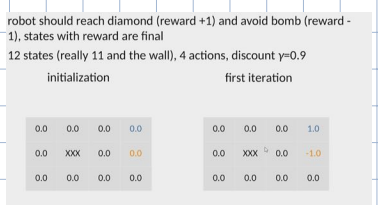
start with random value function, then iteratively improve value function (greedily)
→ converges to optimal value function
→ compute optimal policy from optimal value function

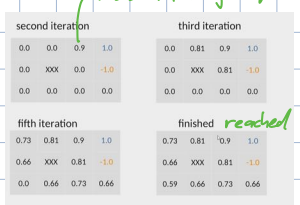
initialize $V_0(s) = 0$ $\forall s$

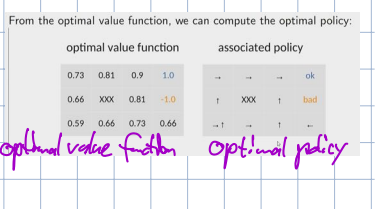
directly optimizes V using Bellman optimality equation as an update rule: $$V_k(s) \leftarrow \max_{a \in A} \sum_{s' \in S} P^a_{ss'} \left(R^a_{ss'} + \gamma V_{k-1}(s')\right)$$

can be seen as a special case of policy iteration, where policy evaluation step is truncated

example:







in more complicated scenarios, state value estimates still get updated after the initial estimate
→ iterate until little change happens

Power of Dynamic Programming:
planning-based → don't need to run the environment / a simulation

as soon as we have more complicated situations, we don't have access to all the information necessary for DP → need to run env. in simulation or whatever and generate actual experience
→ BUT: we still do a similar thing (evaluate policy etc)

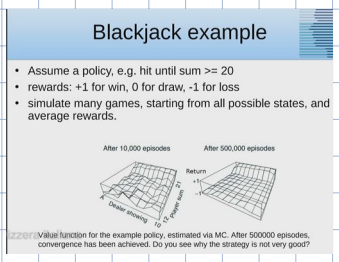
in practical cases model may be intractable / inexistent / only be partially observable / continuous

Tabular Model-free

S	A	Q
1		
2		
3		

state space is known, finite, and observable
→ we can make a table of state-action values and try to estimate/optimize them

Sampling-based Approximations



Bellman equation:

$$V^\pi(s) = \sum_{a\in A}\pi(a|s)\sum_{s'\in S} P^a_{ss'}\left(R^a_{ss'} + \gamma V^\pi(s')\right) = \sum_{a\in A}\sum_{s'\in S}\pi(a|s)P^a_{ss'}\left(R^a_{ss'} + \gamma V^\pi(s')\right)$$

Policy evaluation

$$V_k(s) \leftarrow \sum_{a\in A}\pi(a|s)\sum_{s'\in S} P^a_{ss'}\left(R^a_{ss'} + \gamma V_{k-1}(s')\right)$$

in model-free RL we don't have $P^a_{ss'}$ and $R^a_{ss'}$ → replace sum with expectation

$$V_k(s) \leftarrow E_{a,s'}\left[R^a_{ss'} + \gamma V_{k-1}(s')\right]$$

estimate expectations by sampling:

$$\hat{V}(s) = \frac{1}{N}\sum_{n=1}^{N} G^{(n)}_{smpl}(s) \qquad \text{note } V(s) := E[G_t \mid S_t = s] \text{ where } G_t = \sum_{k=0}^{H}\gamma^k R_{t+k+1}$$

or using a running average:

$$\hat{V}(s) \leftarrow (1-\alpha)\,\hat{V}_{old}(s) + \alpha G_{smpl}(s)$$

what policy is used?

Value Iteration

$$V_k(s) \leftarrow \max_a \sum_{s'} P^a_{ss'}\left(R^a_{ss'} + \gamma V_{k-1}(s')\right)$$

issue: doesn't really work for sampling-based framework, because cannot sample across max operator not really works
→ that's why policy iteration, even though it's more complicated and needs more iterations, is so important

Q function

if we do not have access to state transitions (we don't know which actions lead to which states), V is not enough

$$\hat{Q}(s,a) = \frac{1}{N}\sum_{n=1}^{N} G^{(n)}_{smpl}(s,a)$$

issue: depending on policy, certain state-action pairs never occur
→ stronger requirement than reaching all states

Monte Carlo Methods

idea: collect data from entire episodes
- start anywhere and let agent collect rewards
- assume agent always decides terminal state (episodic task), so that we get an estimate for the return
- estimate state values or action-values by averaging over observed returns!

first-visit: only consider first visit to any state
every-visit: consider all visits

conceptually simple but data hungry

Monte Carlo on Actions

again: if we do not have full knowledge of state transitions, estimating state values is not good enough
→ need to explicitly estimate Q function in order to suggest a policy