Course plan

Search problems

Markov decision processes

Adversarial games

Constraint satisfaction problems

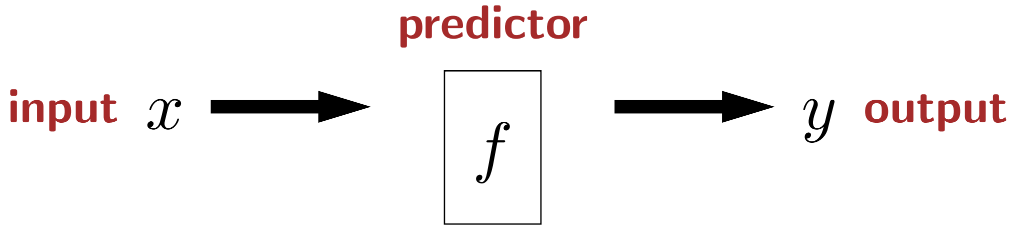
Bayesian networks

Reflex **States** **Variables** **Logic**

"Low-level intelligence" "High-level intelligence"

Machine learning

Reflex-based models



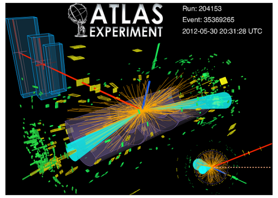
Binary classification

classifier

$x \longrightarrow \boxed{f} \longrightarrow y \in \{+1, -1\}$ label

Fraud detection: credit card transaction $\rightarrow$ fraud or no fraud

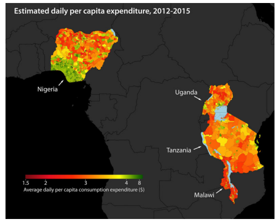
Toxic comments: online comment $\rightarrow$ toxic or not toxic



Higgs boson: measurements of event $\rightarrow$ decay event or background

Regression

$$x \longrightarrow \boxed{f} \longrightarrow y \in \mathbb{R}$$ **response**



Poverty mapping: satellite image $\rightarrow$ asset wealth index

Housing: information about house $\rightarrow$ price

Arrival times: destination, weather, time $\rightarrow$ time of arrival

Structured prediction

$x \longrightarrow \boxed{f} \longrightarrow y$ is a complex object

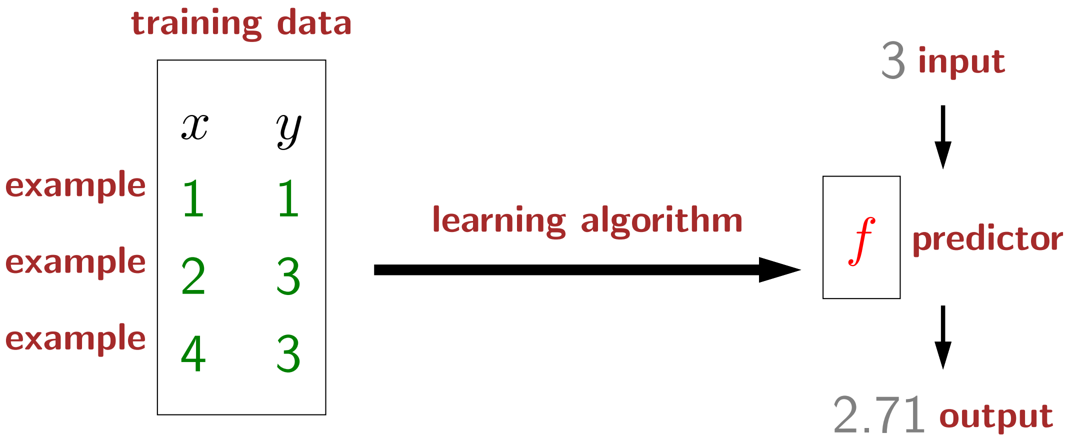
Machine translation: English sentence $\rightarrow$ Japanese sentence

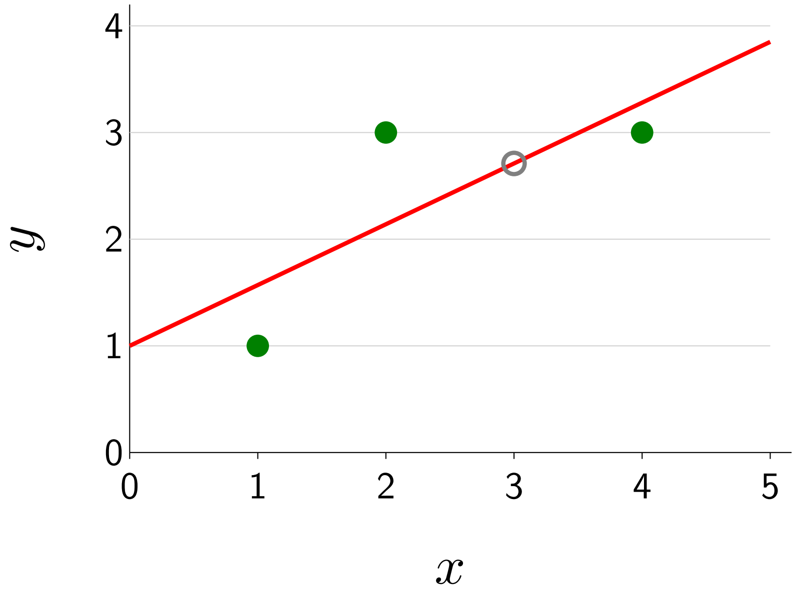
Dialogue: conversational history $\rightarrow$ next utterance

Image captioning: image $\rightarrow$ sentence describing image

Image segmentation: image $\rightarrow$ segmentation

Linear regression framework





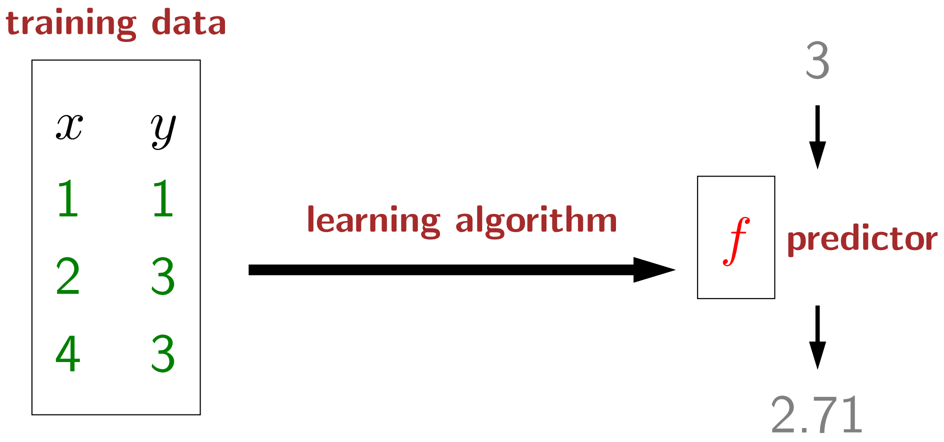
Design decisions:

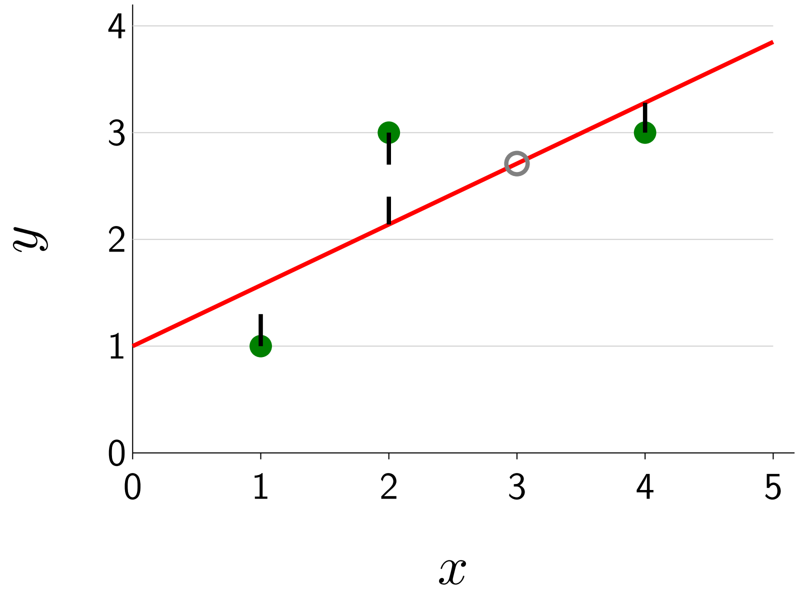
Which predictors are possible? **hypothesis class**

How good is a predictor? **loss function**

How do we compute the best predictor? **optimization algorithm**

Summary





Which predictors are possible?

Hypothesis class

Linear functions

$\mathcal{F} = \{f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x)\}, \phi(x) = [1, x]$

How good is a predictor?

Loss function

Squared loss

$\text{Loss}(x, y, \mathbf{w}) = (f_{\mathbf{w}}(x) - y)^2$

How to compute best predictor?

Optimization algorithm

Gradient descent

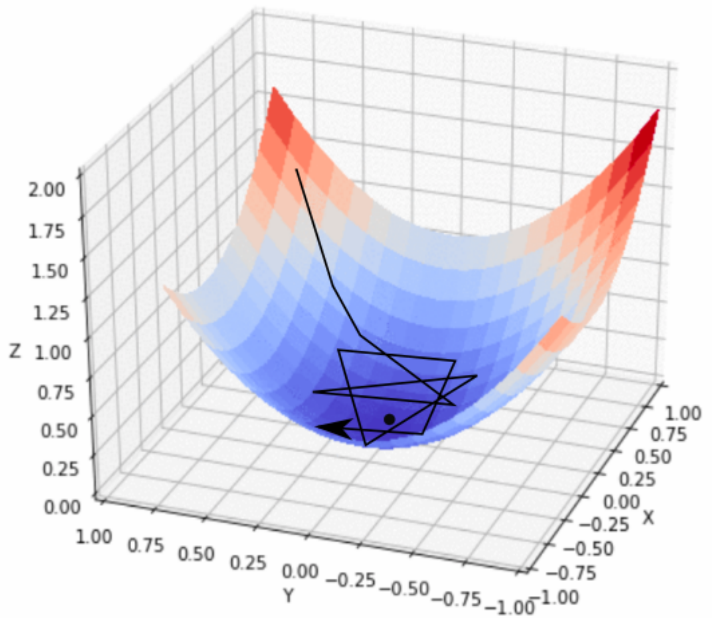
$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla \textbf{TrainLoss}(\mathbf{w})$

Optimization algorithm: how to compute best?

Goal: $\min_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})$

Definition: gradient

The gradient $\nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})$ is the direction that increases the training loss the most.

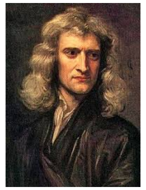


Algorithm: gradient descent

Initialize $\mathbf{w} = [0, \dots, 0]$

For $t = 1, \dots, T$: **epochs**

$$\mathbf{w} \leftarrow \mathbf{w} - \underbrace{\eta}_{\text{step size}} \underbrace{\nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})}_{\text{gradient}}$$



Computing the gradient

Objective function:

$$\text{TrainLoss}(\mathbf{w}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y)\in\mathcal{D}_{\text{train}}} (\mathbf{w}\cdot\phi(x) - y)^2$$

Gradient (use chain rule):

$$\nabla_{\mathbf{w}}\text{TrainLoss}(\mathbf{w}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y)\in\mathcal{D}_{\text{train}}} 2(\underbrace{\mathbf{w}\cdot\phi(x) - y}_{\text{prediction}-\text{target}})\phi(x)$$

Linear classification framework

training data

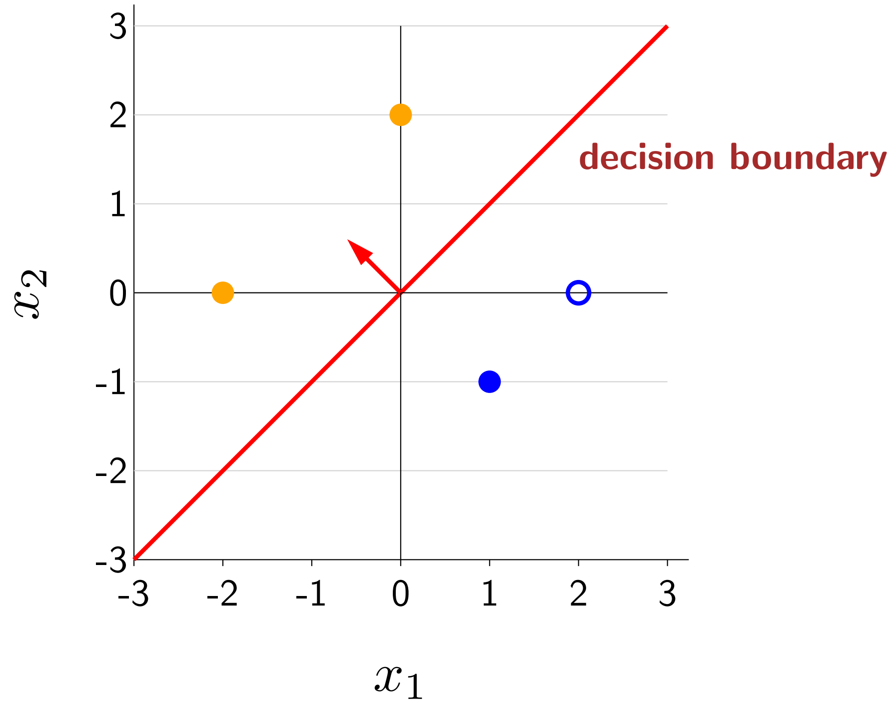
	x_1	x_2	y
example	0	2	1
example	-2	0	1
example	1	-1	-1

learning algorithm

[2, 0] input

f classifier

-1 label



Design decisions:

Which classifiers are possible? **hypothesis class**

How good is a classifier? **loss function**

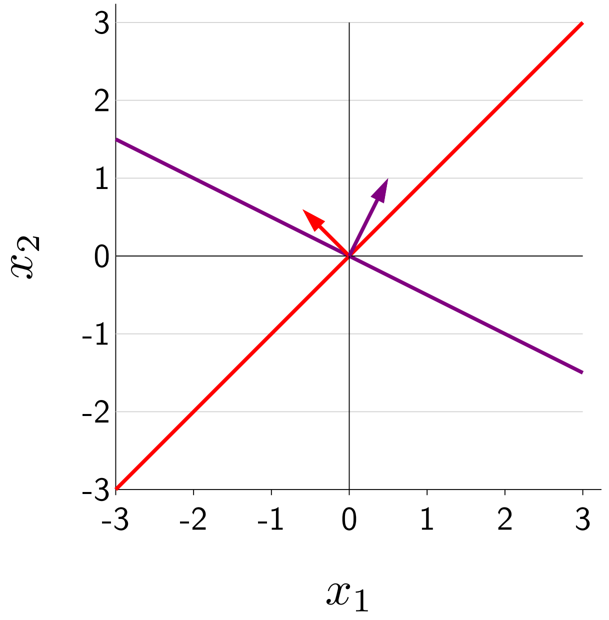
How do we compute the best classifier? **optimization algorithm**

Hypothesis class: which classifiers?

$\phi(x) = [x_1, x_2]$

$f(x) = \text{sign}([-0.6, 0.6] \cdot \phi(x))$

$f(x) = \text{sign}([0.5, 1] \cdot \phi(x))$



General binary classifier:

$$f_{\mathbf{w}}(x) = \text{sign}(\mathbf{w} \cdot \phi(x))$$

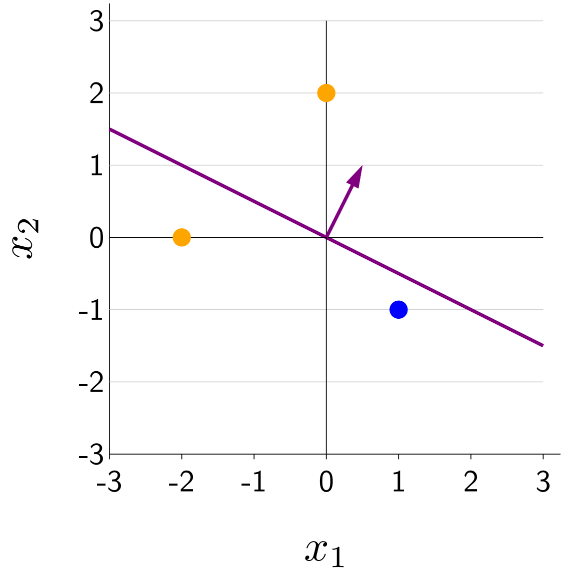
Hypothesis class:

$$\mathcal{F} = \{f_{\mathbf{w}} : \mathbf{w} \in \mathbb{R}^2\}$$

Score and margin

Predicted label: $f_{\mathbf{w}}(x) = \text{sign}(\mathbf{w} \cdot \phi(x))$

Target label: y



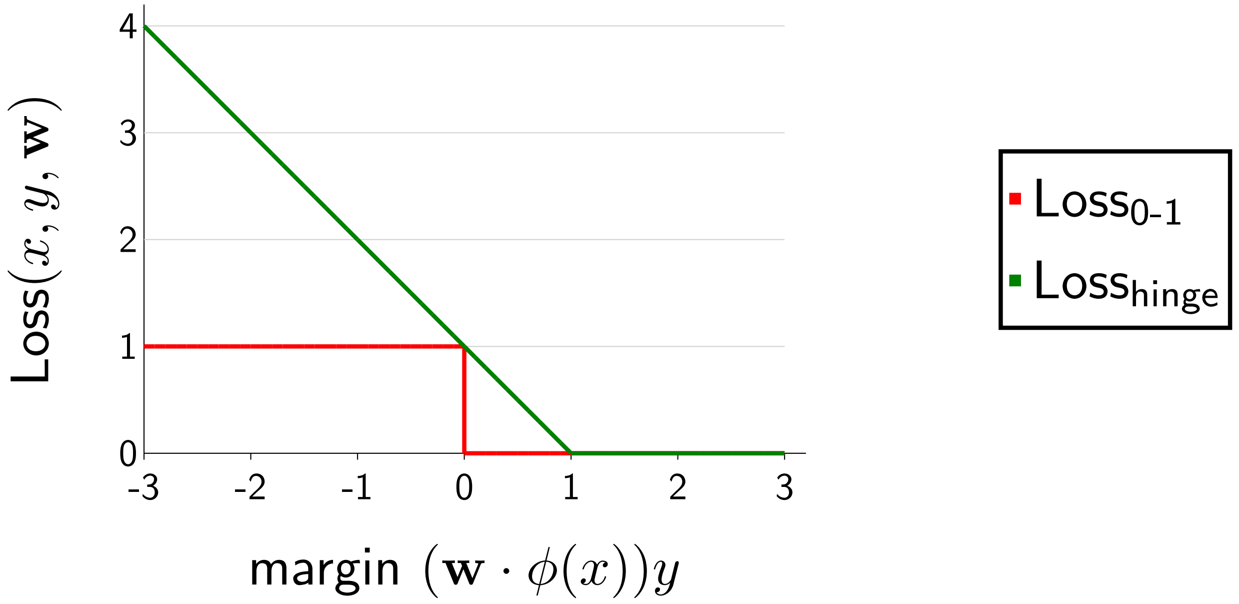
Definition: score

The score on an example (x, y) is $\mathbf{w} \cdot \phi(x)$, how **confident** we are in predicting $+1$.

Definition: margin

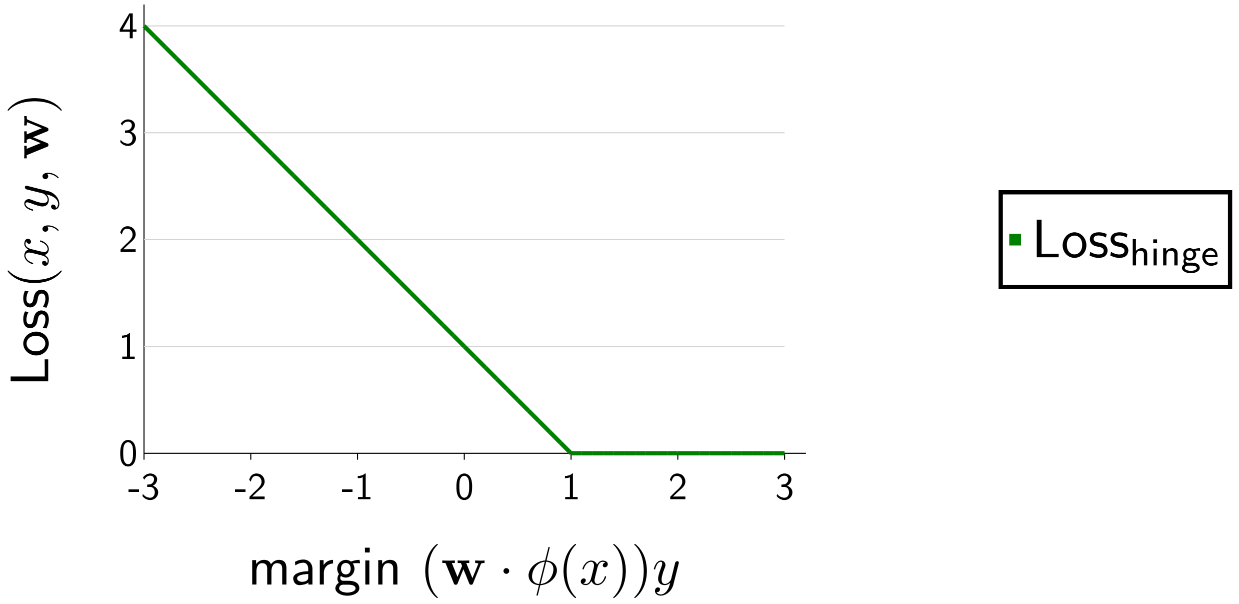
The margin on an example (x, y) is $(\mathbf{w} \cdot \phi(x))y$, how **correct** we are.

Hinge loss



$$\text{Loss}_{\text{hinge}}(x, y, \mathbf{w}) = \max\{1 - (\mathbf{w} \cdot \phi(x))y, 0\}$$

Gradient of the hinge loss



$$\text{Loss}_{\text{hinge}}(x, y, \mathbf{w}) = \max\{1 - (\mathbf{w} \cdot \phi(x))y, 0\}$$

$$\nabla \text{Loss}_{\text{hinge}}(x, y, \mathbf{w}) = \begin{cases} -\phi(x)y & \text{if } \{1 - (\mathbf{w} \cdot \phi(x))y\} > \{0\} \\ 0 & \text{otherwise} \end{cases}$$

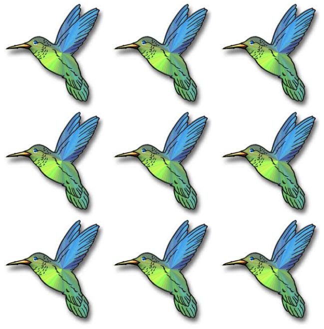
Summary so far

$$\underbrace{\mathbf{w} \cdot \phi(x)}_{\text{score}}$$

	Regression	Classification
Prediction $f_{\mathbf{w}}(x)$	score	sign(score)
Relate to target y	residual (score $- y$)	margin (score y)
Loss functions	squared absolute deviation	zero-one hinge logistic
Algorithm	gradient descent	gradient descent

Stochastic gradient descent

$$\text{TrainLoss}(\mathbf{w}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y)\in\mathcal{D}_{\text{train}}} \text{Loss}(x, y, \mathbf{w})$$



Algorithm: stochastic gradient descent

Initialize $\mathbf{w} = [0, \dots, 0]$

For $t = 1, \dots, T$:

  For $(x, y) \in \mathcal{D}_{\text{train}}$:

    $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \text{Loss}(x, y, \mathbf{w})$

Per-group loss

x	y	g
1	4	A
2	8	A
5	5	B
6	6	B
7	7	B
8	8	B

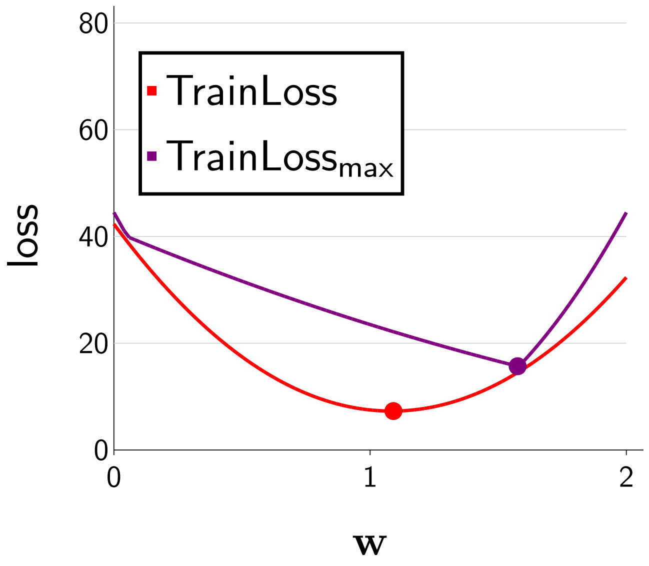
$$\text{TrainLoss}_g(\mathbf{w}) = \frac{1}{|\mathcal{D}_{\text{train}}(g)|} \sum_{(x,y)\in\mathcal{D}_{\text{train}}(g)} \text{Loss}(x, y, \mathbf{w})$$

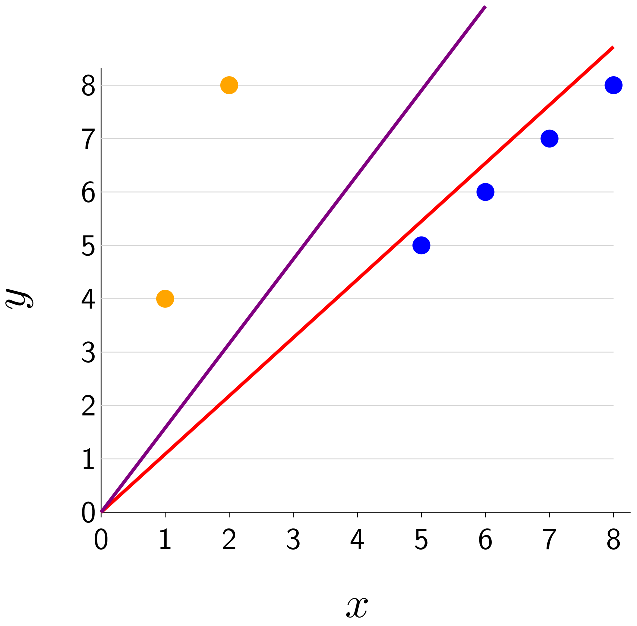
$$\text{TrainLoss}_{\text{A}}(1) = \tfrac{1}{2}((1-4)^2 + (2-8)^2) = 22.5$$

$$\text{TrainLoss}_{\text{B}}(1) = \tfrac{1}{4}((5-5)^2 + (6-6)^2 + (7-7)^2 + (8-8)^2) = 0$$

Summary

x	y	g
1	4	A
2	8	A
5	5	B
6	6	B
7	7	B
8	8	B





- Maximum group loss $\neq$ average loss
- Group DRO: minimize the maximum group loss
- Many more nuances: intersectionality? don't know groups? overfitting?

Quadratic predictors

$$\phi(x) = [1, x, x^2]$$

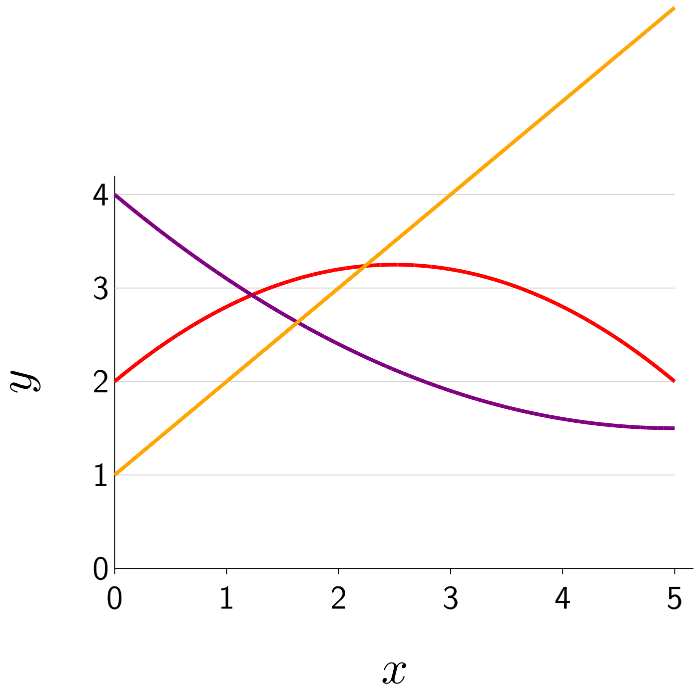
Example: $\phi(3) = [1, 3, 9]$

$$f(x) = [2, 1, -0.2] \cdot \phi(x)$$

$$f(x) = [4, -1, 0.1] \cdot \phi(x)$$

$$f(x) = [1, 1, 0] \cdot \phi(x)$$

$$\mathcal{F} = \{f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x) : \mathbf{w} \in \mathbb{R}^3\}$$



Non-linear predictors just by changing ϕ

Predictors with periodicity structure

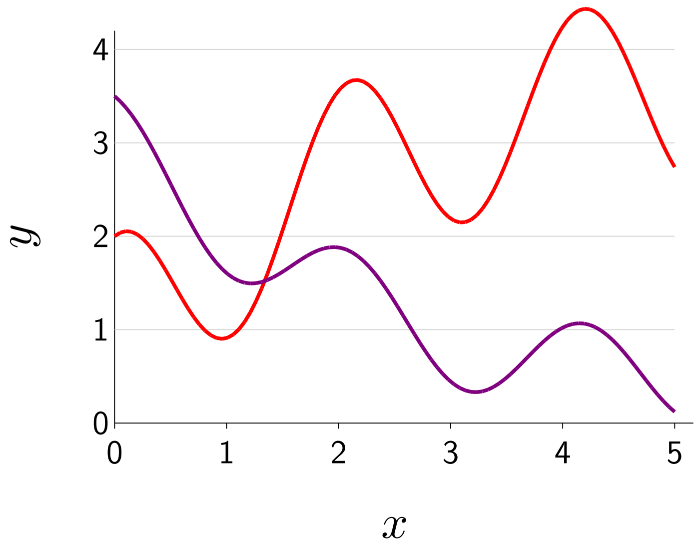
$$\phi(x) = [1, x, x^2, \cos(3x)]$$

Example: $\phi(2) = [1, 2, 4, 0.96]$

$$f(x) = [1, 1, -0.1, 1] \cdot \phi(x)$$

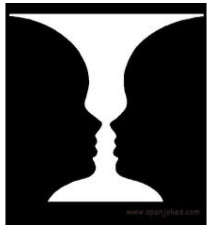
$$f(x) = [3, -1, 0.1, 0.5] \cdot \phi(x)$$

$$\mathcal{F} = \{f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x) : \mathbf{w} \in \mathbb{R}^4\}$$



Just throw in any features you want

Linear in what?



Prediction:

$$f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x)$$

Linear in $\mathbf{w}$? Yes

Linear in $\phi(x)$? Yes

Linear in x? No!

Key idea: non-linearity

- Expressivity: score $\mathbf{w} \cdot \phi(x)$ can be a **non-linear** function of x
- Efficiency: score $\mathbf{w} \cdot \phi(x)$ always a **linear** function of $\mathbf{w}$

Feature templates

Definition: feature template

A **feature template** is a group of features all computed in a similar way.

abc@gmail.com → last three characters equals ___ →

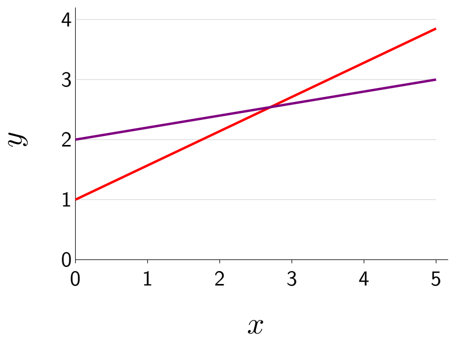
endsWith_aaa : 0
endsWith_aab : 0
endsWith_aac : 0
...
endsWith_com : 1
...
endsWith_zzz : 0

Define types of pattern to look for, not particular patterns

Non-linear predictors

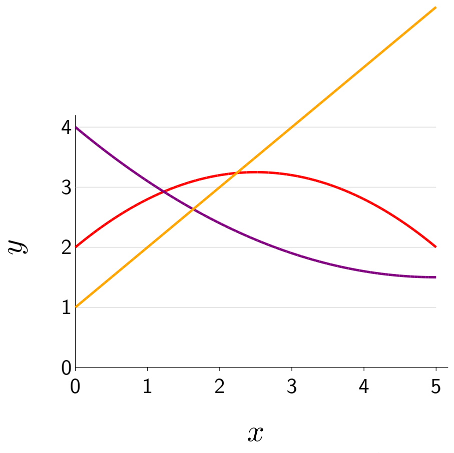
Linear predictors:

$$f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x),\ \phi(x) = [1, x]$$



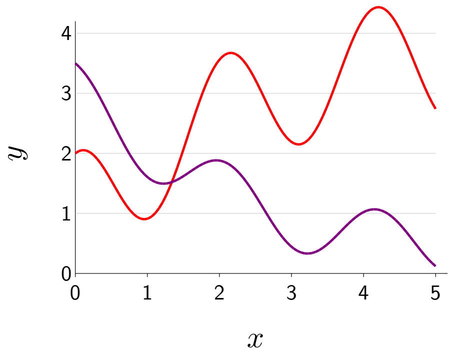
Non-linear (quadratic) predictors:

$$f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x),\ \phi(x) = [1, x, x^2]$$



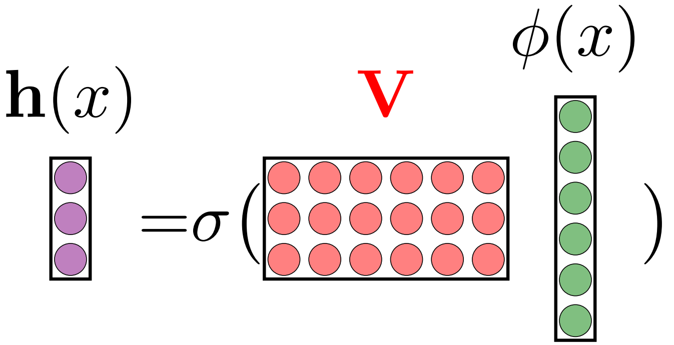
Non-linear neural networks:

$$f_{\mathbf{w}}(x) = \mathbf{w} \cdot \sigma(\mathbf{V}\phi(x)),\ \phi(x) = [1, x]$$

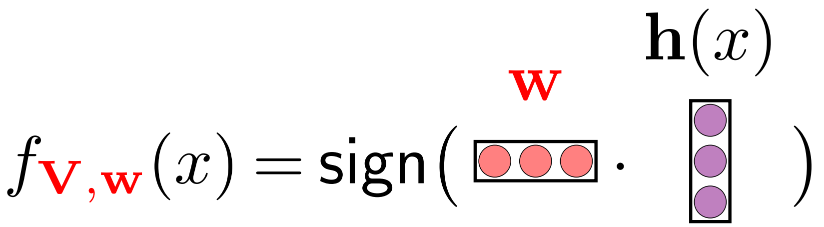


Two-layer neural networks

Intermediate subproblems:



Predictor (classification):



Interpret $\mathbf{h}(x)$ as a learned feature representation!

Hypothesis class:

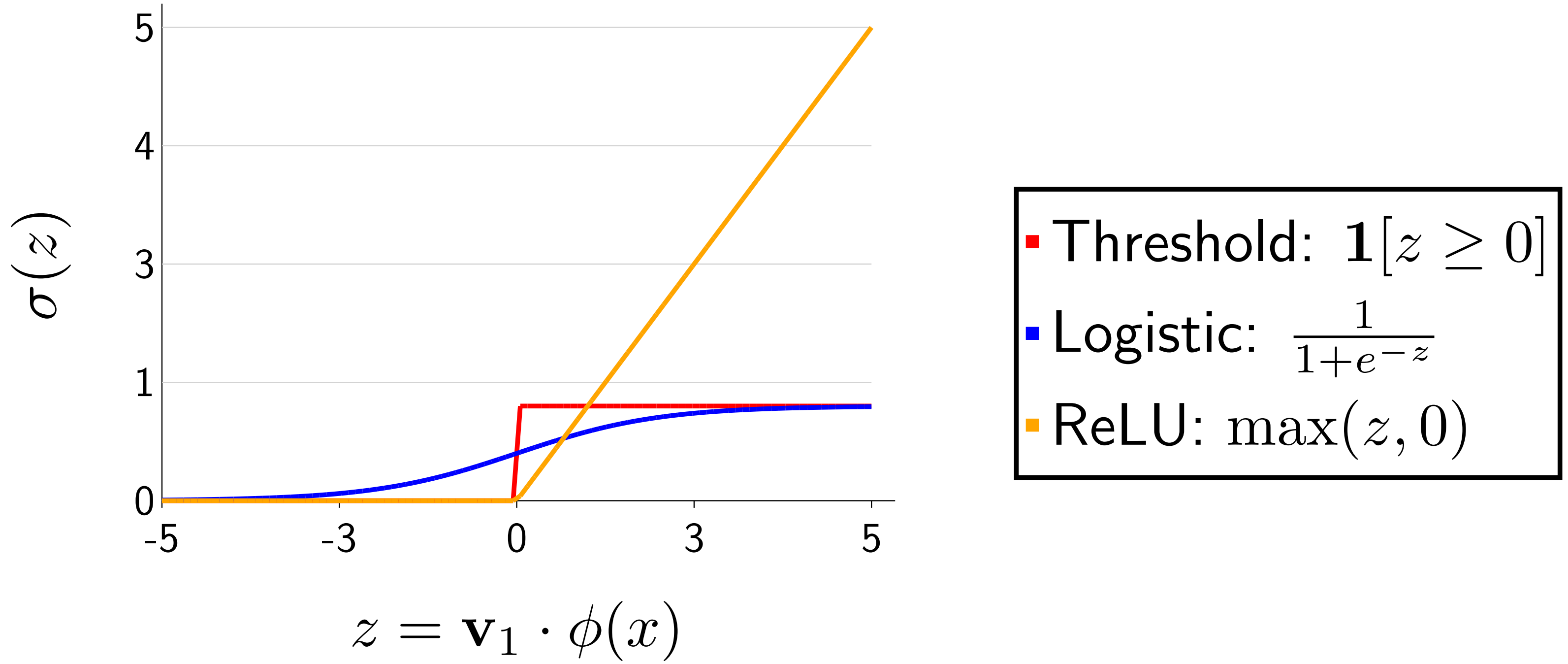
$$\mathcal{F} = \{f_{\mathbf{V},\mathbf{w}} : \mathbf{V} \in \mathbb{R}^{k\times d}, \mathbf{w} \in \mathbb{R}^{k}\}$$

Avoid zero gradients

Problem: gradient of $h_1(x)$ with respect to $\mathbf{v}_1$ is 0

$$h_1(x) = \mathbf{1}[\mathbf{v}_1 \cdot \phi(x) \geq 0]$$

Solution: replace with an **activation function** σ with non-zero gradients



$$h_1(x) = \sigma(\mathbf{v}_1 \cdot \phi(x))$$

Summary

$$\text{score} = \mathbf{w} \cdot \sigma(\mathbf{V}\,\phi(x))$$

- Intuition: decompose problem into intermediate parallel subproblems
- Deep networks iterate this decomposition multiple times
- Hypothesis class contains predictors ranging over weights for all layers
- Next up: learning neural networks

Motivation: regression with four-layer neural networks

Loss on one example:

$$\text{Loss}(x, y, \mathbf{V}_1, \mathbf{V}_2, \mathbf{V}_3, \mathbf{w}) = (\mathbf{w} \cdot \sigma(\mathbf{V}_3 \sigma(\mathbf{V}_2 \sigma(\mathbf{V}_1 \phi(x)))) - y)^2$$

Stochastic gradient descent:

$$\mathbf{V}_1 \leftarrow \mathbf{V}_1 - \eta \nabla_{\mathbf{V}_1} \text{Loss}(x, y, \mathbf{V}_1, \mathbf{V}_2, \mathbf{V}_3, \mathbf{w})$$

$$\mathbf{V}_2 \leftarrow \mathbf{V}_2 - \eta \nabla_{\mathbf{V}_2} \text{Loss}(x, y, \mathbf{V}_1, \mathbf{V}_2, \mathbf{V}_3, \mathbf{w})$$

$$\mathbf{V}_3 \leftarrow \mathbf{V}_3 - \eta \nabla_{\mathbf{V}_3} \text{Loss}(x, y, \mathbf{V}_1, \mathbf{V}_2, \mathbf{V}_3, \mathbf{w})$$

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \text{Loss}(x, y, \mathbf{V}_1, \mathbf{V}_2, \mathbf{V}_3, \mathbf{w})$$

How to get the gradient without doing manual work?

Computation graphs

$$\text{Loss}(x, y, \mathbf{V}_1, \mathbf{V}_2, \mathbf{V}_3, \mathbf{w}) = (\mathbf{w} \cdot \sigma(\mathbf{V}_3 \sigma(\mathbf{V}_2 \sigma(\mathbf{V}_1 \phi(x)))) - y)^2$$

Definition: computation graph

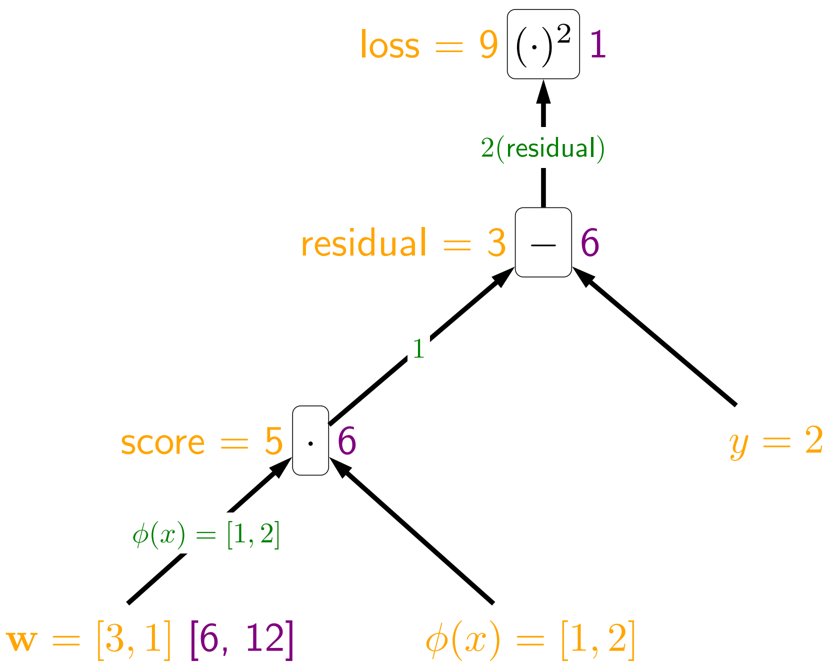
A directed acyclic graph whose root node represents the final mathematical expression and each node represents intermediate subexpressions.

Upshot: compute gradients via general **backpropagation** algorithm

Purposes:

- Automatically compute gradients (how TensorFlow and PyTorch work)
- Gain insight into modular structure of gradient computations

Backpropagation



$$\text{Loss}(x, y, \mathbf{w}) = (\mathbf{w} \cdot \phi(x) - y)^2$$

$$\mathbf{w} = [3, 1], \phi(x) = [1, 2], y = 2$$

backpropagation

$$\nabla_{\mathbf{w}} \text{Loss}(x, y, \mathbf{w}) = [6, 12]$$

Definition: Forward/backward values

Forward: f_i is value for subexpression rooted at i

Backward: $g_i = \frac{\partial \text{loss}}{\partial f_i}$ is how f_i influences loss

Algorithm: backpropagation algorithm

Forward pass: compute each f_i (from leaves to root)

Backward pass: compute each g_i (from root to leaves)

Summary

Not the real objective: training loss

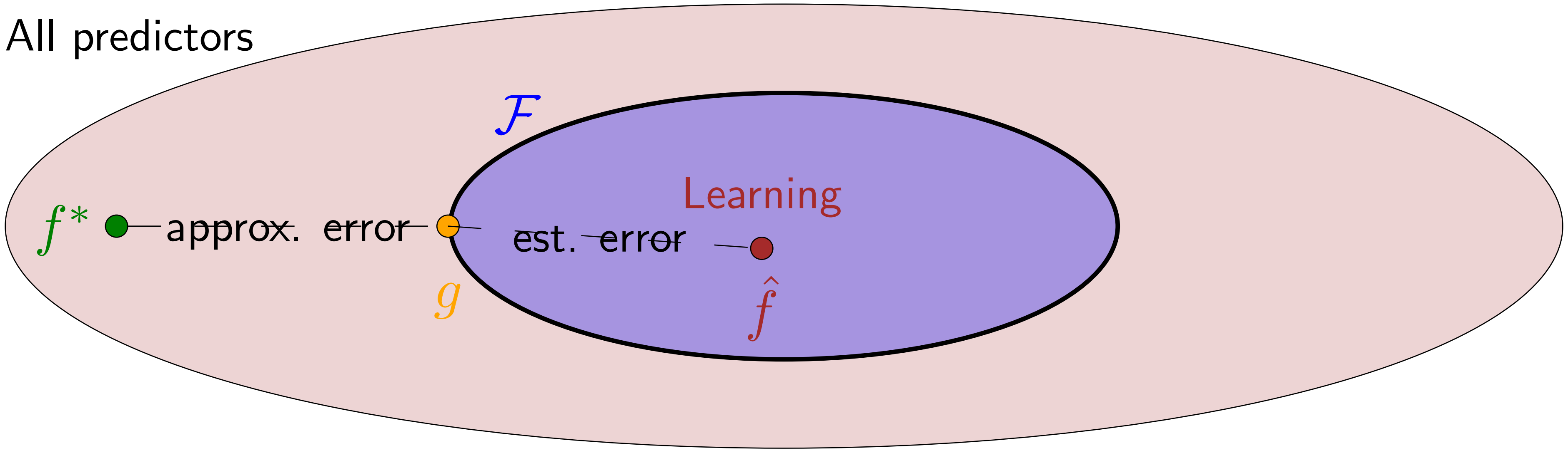
Real objective: loss on unseen future examples

Semi-real objective: test loss

Key idea: keep it simple

Try to minimize training error, but keep the hypothesis class small.

Approximation and estimation error



- Approximation error: how good is the hypothesis class?
- Estimation error: how good is the learned predictor **relative to** the potential of the hypothesis class?

$$\text{Err}(\hat{f}) - \text{Err}(f^*) = \underbrace{\text{Err}(\hat{f}) - \text{Err}(g)}_{\text{estimation}} + \underbrace{\text{Err}(g) - \text{Err}(f^*)}_{\text{approximation}}$$

Controlling the norm

Regularized objective:

$$\min_{\mathbf{w}} \text{TrainLoss}(\mathbf{w}) + \frac{\lambda}{2}\|\mathbf{w}\|^2$$

Algorithm: gradient descent

Initialize $\mathbf{w} = [0, \dots, 0]$

For $t = 1, \dots, T$:

$\quad \mathbf{w} \leftarrow \mathbf{w} - \eta(\nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w}) + \lambda \mathbf{w})$

Same as gradient descent, except shrink the weights towards zero by λ.

Hyperparameters

Definition: hyperparameters

Design decisions (hypothesis class, training objective, optimization algorithm) that need to be made before running the learning algorithm.

How do we choose hyperparameters?

Choose hyperparameters to minimize $\mathcal{D}_{\text{train}}$ error?

No - optimum would be to include all features, no regularization, train forever

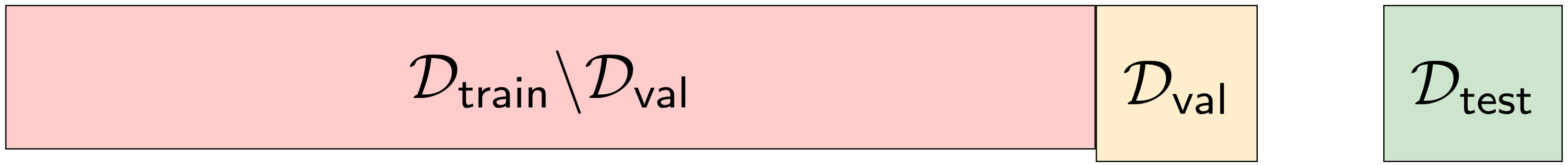
Choose hyperparameters to minimize $\mathcal{D}_{\text{test}}$ error?

No - choosing based on $\mathcal{D}_{\text{test}}$ makes it an unreliable estimate of error!

Validation set

Definition: validation set

A **validation set** is taken out of the training set and used to optimize hyperparameters.

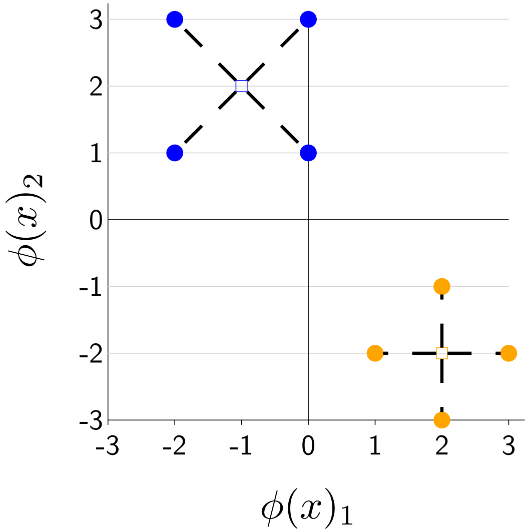


For each setting of hyperparameters, train on $\mathcal{D}_{\text{train}} \backslash \mathcal{D}_{\text{val}}$, evaluate on $\mathcal{D}_{\text{val}}$

Summary

Clustering: discover structure in unlabeled data

K-means objective:



K-means algorithm:

assignments $\mathbf{z}$  centroids $\boldsymbol{\mu}$

Unsupervised learning use cases:

- Data exploration and discovery
- Providing representations to downstream supervised learning

K-means algorithm

Algorithm: K-means

Initialize $\boldsymbol{\mu} = [\mu_1, \dots, \mu_K]$ randomly.

For $t = 1, \dots, T$:

Step 1: set assignments $\mathbf{z}$ given $\boldsymbol{\mu}$

For each point $i = 1, \dots, n$:

$$z_i \leftarrow \arg \min_{k=1,\dots,K} \|\phi(x_i) - \mu_k\|^2$$

Step 2: set centroids $\boldsymbol{\mu}$ given $\mathbf{z}$

For each cluster $k = 1, \dots, K$:

$$\mu_k \leftarrow \frac{1}{|\{i : z_i = k\}|} \sum_{i : z_i = k} \phi(x_i)$$

Course plan

Search problems

Markov decision processes

Adversarial games

Constraint satisfaction problems

Bayesian networks

Reflex **States** **Variables** **Logic**

"Low-level intelligence" "High-level intelligence"

Machine learning

Beyond reflex

Classifier (reflex-based models):

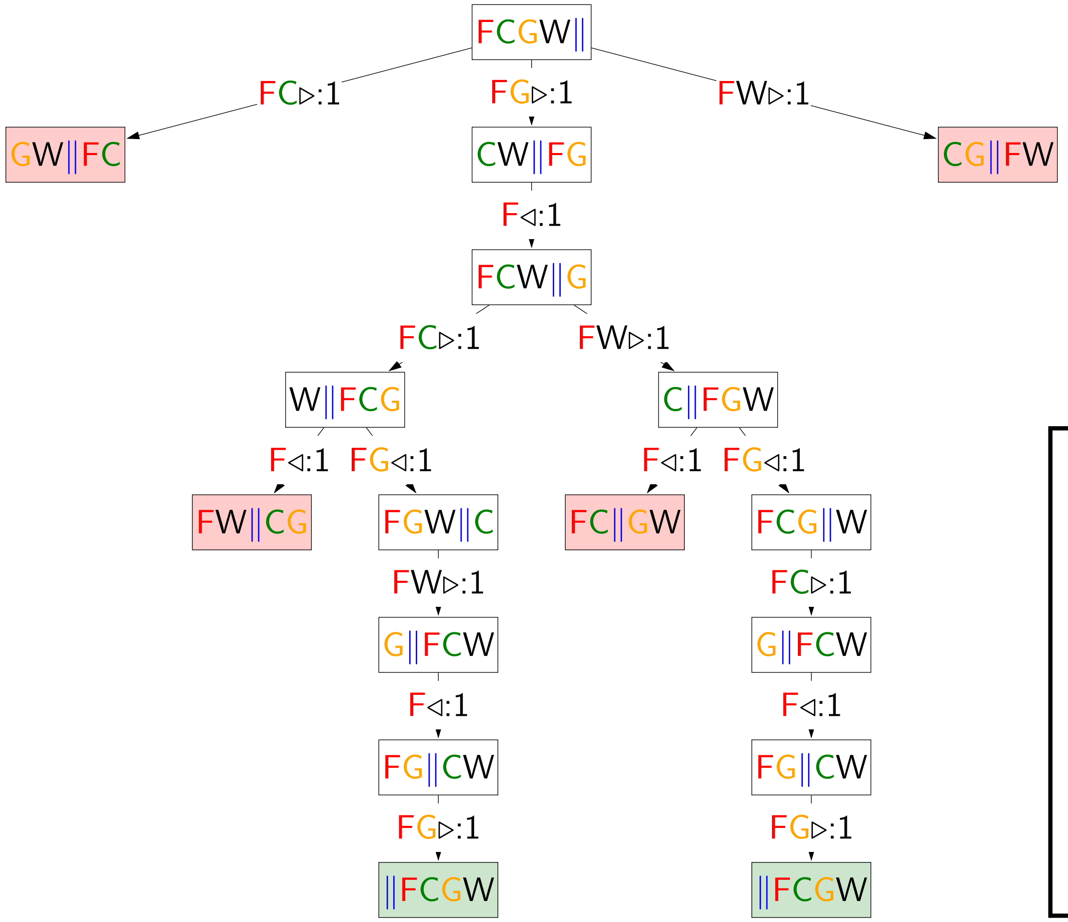
$x \longrightarrow \boxed{f} \longrightarrow$ single action $y \in \{-1, +1\}$

Search problem (state-based models):

$x \longrightarrow \boxed{f} \longrightarrow$ **action sequence** $(a_1, a_2, a_3, a_4, \dots)$

Key: need to consider future consequences of an action!

Search problem



Definition: search problem

- s_{start}: starting state
- $\text{Actions}(s)$: possible actions
- $\text{Cost}(s, a)$: action cost
- $\text{Succ}(s, a)$: successor
- $\text{IsEnd}(s)$: reached end state?

Dynamic Programming Review

state s

$\text{Cost}(s, a)$

state s'

$\text{FutureCost}(s')$

end state

$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if IsEnd}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s, a) + \text{FutureCost}(\text{Succ}(s, a))] & \text{otherwise} \end{cases}$$

Key idea: state

A **state** is a summary of all the past actions sufficient to choose future actions **optimally**.

Dynamic programming

Algorithm: dynamic programming

def DynamicProgramming(s):

  If already computed for s, return cached answer.

  If IsEnd(s): return solution

  For each action $a \in$ Actions(s): ...

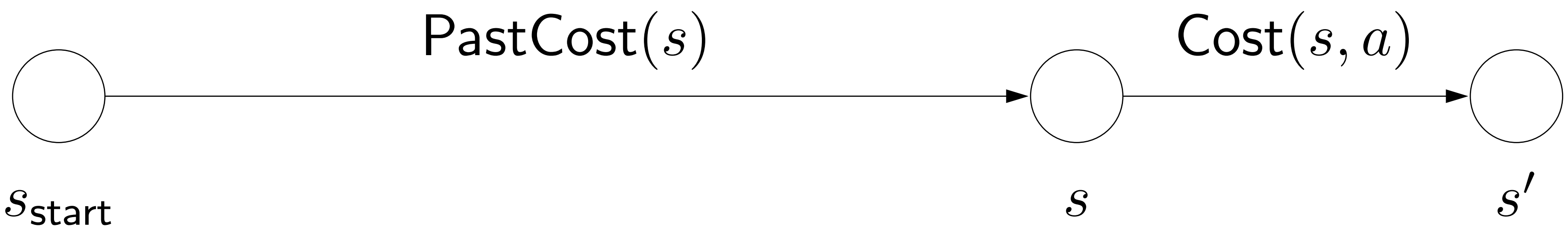
[semi-live solution: `Dynamic Programming`]

Assumption: acyclicity

The state graph defined by Actions(s) and Succ(s, a) is acyclic.

Ordering the states

Observation: prefixes of optimal path are optimal



Key: if graph is acyclic, dynamic programming makes sure we compute $\text{PastCost}(s)$ before $\text{PastCost}(s')$

If graph is cyclic, then we need another mechanism to order states...

Uniform cost search (UCS)

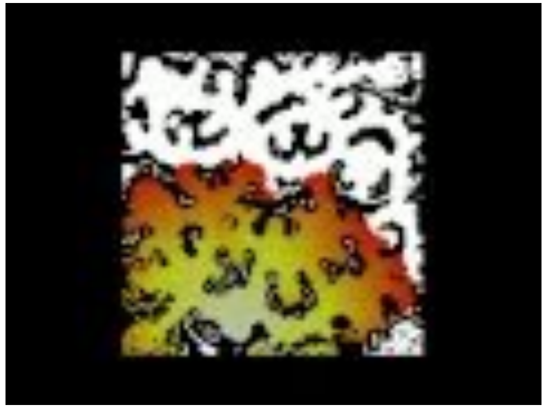
Key idea: state ordering

UCS enumerates states in order of increasing past cost.

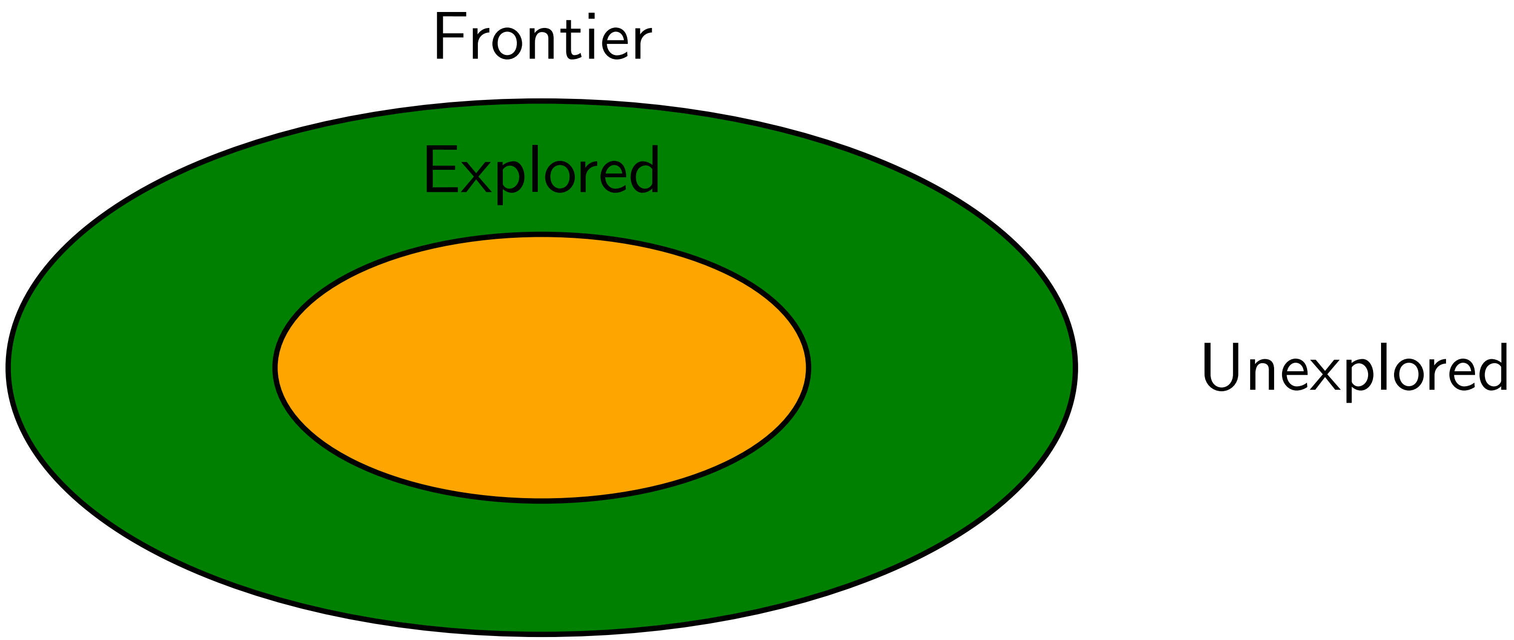
Assumption: non-negativity

All action costs are non-negative: $\text{Cost}(s, a) \geq 0$.

UCS in action:



High-level strategy



- Explored: states we've found the optimal path to
- Frontier: states we've seen, still figuring out how to get there cheaply
- Unexplored: states we haven't seen

Uniform cost search (UCS)

Algorithm: uniform cost search [Dijkstra, 1956]

Add s_{start} to **frontier** (priority queue)

Repeat until frontier is empty:

  Remove s with smallest priority p from frontier

  If IsEnd(s): return solution

  Add s to **explored**

  For each action $a \in$ Actions(s):

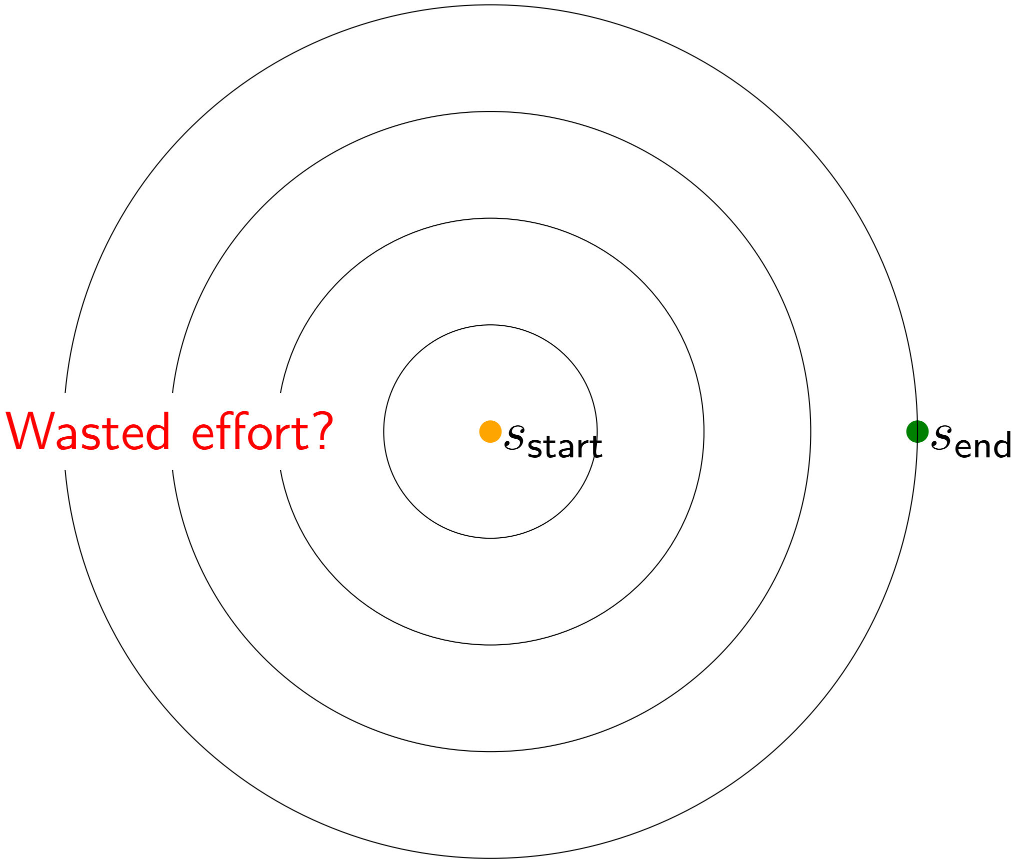
    Get successor $s' \leftarrow$ Succ(s, a)

    If s' already in explored: continue

    Update **frontier** with s' and priority $p +$ Cost(s, a)

[semi-live solution: `Uniform Cost Search`]

Can uniform cost search be improved?

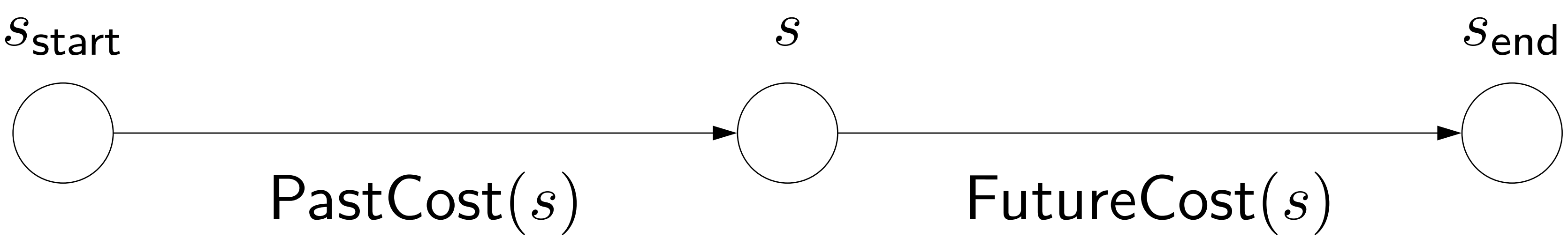


Problem: UCS orders states by cost from s_{start} to s

Goal: take into account cost from s to s_{end}

Exploring states

UCS: explore states in order of PastCost(s)



Ideal: explore in order of PastCost(s) + FutureCost(s)

A*: explore in order of PastCost(s) + $h(s)$

Definition: Heuristic function

A heuristic $h(s)$ is any estimate of FutureCost(s).

A* search

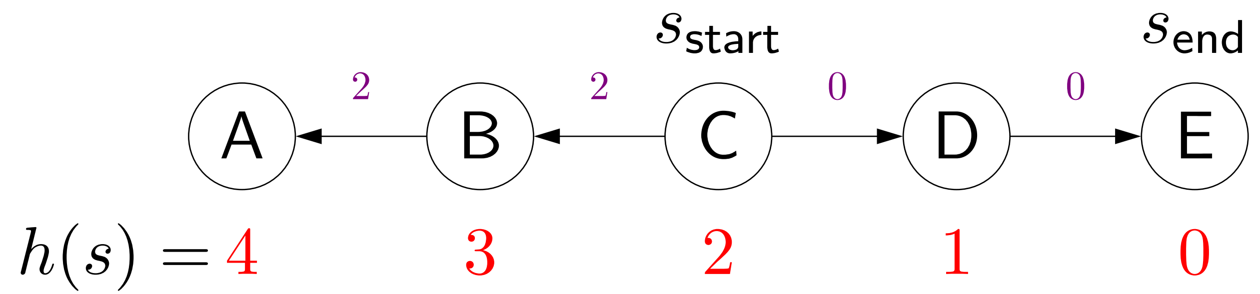
Algorithm: A* search [Hart/Nilsson/Raphael, 1968]

Run uniform cost search with **modified edge costs**:

$$\text{Cost}'(s,a) = \text{Cost}(s,a) + h(\text{Succ}(s,a)) - h(s)$$

Intuition: add a penalty for how much action a takes us away from the end state

Example:



$$\text{Cost}'(C,B) = \text{Cost}(C,B) + h(B) - h(C) = 1 + (3-2) = 2$$

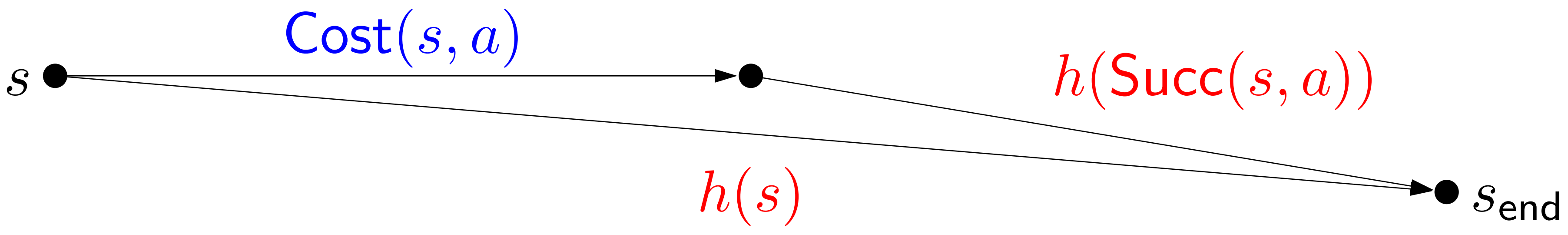
Consistent heuristics

Definition: consistency

A heuristic h is **consistent** if

- $\text{Cost}'(s, a) = \text{Cost}(s, a) + h(\text{Succ}(s, a)) - h(s) \geq 0$
- $h(s_{\text{end}}) = 0$.

Condition 1: needed for UCS to work (triangle inequality).



Condition 2: $\text{FutureCost}(s_{\text{end}}) = 0$ so match it.

Correctness of A*

Proposition: correctness

If h is consistent, A* returns the minimum cost path.

Efficiency of A*

Theorem: efficiency of A*

A* explores all states s satisfying
$\text{PastCost}(s) \leq \text{PastCost}(s_{\text{end}}) - h(s)$

Interpretation: the larger $h(s)$, the better

Proof: A* explores all s such that

$$\text{PastCost}(s) + h(s)$$

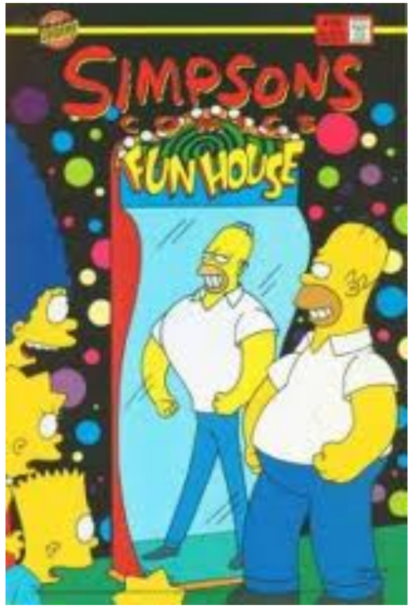
$$\leq$$

$$\text{PastCost}(s_{\text{end}})$$

A* search

Key idea: distortion

A* distorts edge costs to favor end states.



Admissibility

Definition: admissibility

A heuristic $h(s)$ is admissible if

$$h(s) \leq \text{FutureCost}(s)$$

Intuition: admissible heuristics are optimistic

Theorem: consistency implies admissibility

If a heuristic $h(s)$ is **consistent**, then $h(s)$ is **admissible**.

Proof: use induction on $\text{FutureCost}(s)$

Relaxation

Intuition: ideally, use $h(s) = \text{FutureCost}(s)$, but that's as hard as solving the original problem.

Key idea: relaxation

Constraints make life hard. Get rid of them.

But this is just for the heuristic!

General framework

Definition: relaxed search problem

A **relaxation** P_{rel} of a search problem P has costs that satisfy:

$$\text{Cost}_{\text{rel}}(s, a) \leq \text{Cost}(s, a).$$

Definition: relaxed heuristic

Given a relaxed search problem P_{rel}, define the **relaxed heuristic** $h(s) = \text{FutureCost}_{\text{rel}}(s)$, the minimum cost from s to an end state using $\text{Cost}_{\text{rel}}(s, a)$.

Max of two heuristics

How do we combine two heuristics?

Proposition: max heuristic

Suppose $h_1(s)$ and $h_2(s)$ are consistent.

Then $h(s) = \max\{h_1(s), h_2(s)\}$ is consistent.

Proof: exercise

Course plan

Search problems

$\boxed{\text{Markov decision processes}}$

Adversarial games

Constraint satisfaction problems

Bayesian networks

Reflex **States** **Variables** **Logic**

"Low-level intelligence" "High-level intelligence"

Machine learning

Uncertainty in the real world

state s, action a $\xrightarrow{\textbf{random}}$ state s'_1 / state s'_2

Markov decision process

Definition: Markov decision process

States: the set of states

$s_{\text{start}} \in$ States: starting state

Actions(s): possible actions from state s

$T(s, a, s')$: probability of s' if take action a in state s

Reward(s, a, s'): reward for the transition (s, a, s')

IsEnd(s): whether at end of game

$0 \leq \gamma \leq 1$: discount factor (default: 1)

What is a solution?

Search problem: path (sequence of actions)

MDP:

Definition: policy

A **policy** π is a mapping from each state $s \in$ States to an action $a \in$ Actions(s).

Example: volcano crossing

s	$\pi(s)$
(1,1)	S
(2,1)	E
(3,1)	N
...	...

Evaluating a policy

Definition: utility

Following a policy yields a **random path**.

The **utility** of a policy is the (discounted) sum of the rewards on the path (this is a random variable).

Path	Utility
[in; stay, 4, end]	4
[in; stay, 4, in; stay, 4, in; stay, 4, end]	12
[in; stay, 4, in; stay, 4, end]	8
[in; stay, 4, in; stay, 4, in; stay, 4, in; stay, 4, end]	16
...	...

Definition: value (expected utility)

The **value** of a policy at a state is the **expected** utility.

Value: 12

Discounting

Definition: utility

Path: $s_0, a_1 r_1 s_1, a_2 r_2 s_2, \ldots$ (action, reward, new state).

The **utility** with discount γ is

$$u_1 = r_1 + \gamma r_2 + \gamma^2 r_3 + \gamma^3 r_4 + \cdots$$

Discount $\gamma = 1$ (save for the future):

[stay, stay, stay, stay]: $4 + 4 + 4 + 4 = 16$

Discount $\gamma = 0$ (live in the moment):

[stay, stay, stay, stay]: $4 + 0 \cdot (4 + \cdots) = 4$

Discount $\gamma = 0.5$ (balanced life):

[stay, stay, stay, stay]: $4 + \frac{1}{2} \cdot 4 + \frac{1}{4} \cdot 4 + \frac{1}{8} \cdot 4 = 7.5$

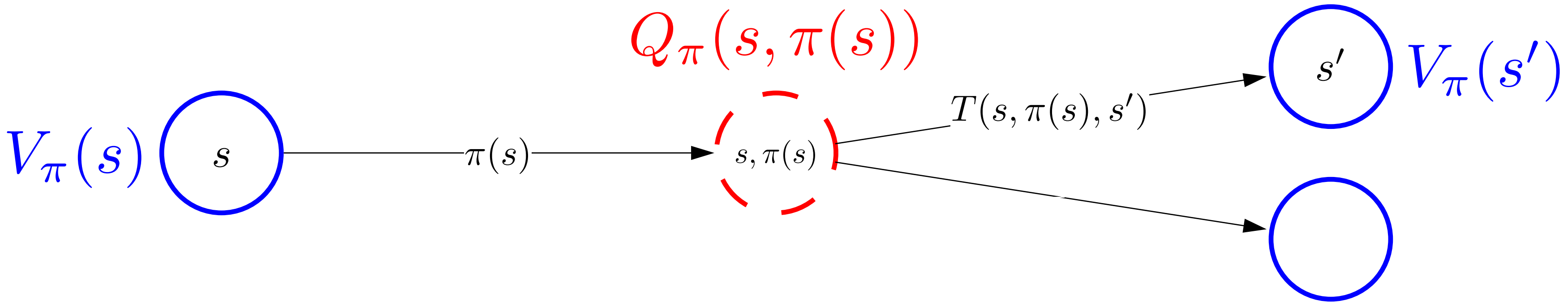
Policy evaluation

Definition: value of a policy

Let $V_\pi(s)$ be the expected utility received by following policy π from state s.

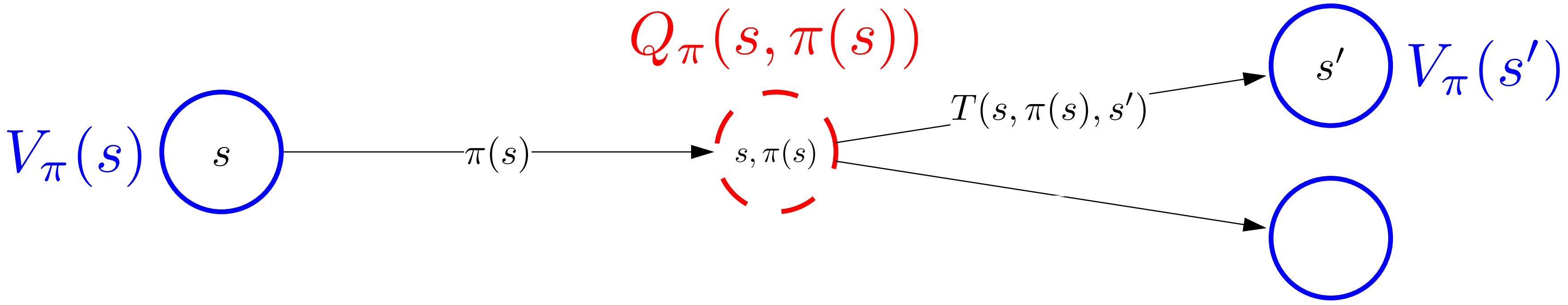
Definition: Q-value of a policy

Let $Q_\pi(s, a)$ be the expected utility of taking action a from state s, and then following policy π.



Policy evaluation

Plan: define recurrences relating value and Q-value



$$V_\pi(s) = \begin{cases} 0 & \text{if IsEnd}(s) \\ Q_\pi(s, \pi(s)) & \text{otherwise.} \end{cases}$$

$$Q_\pi(s, a) = \sum_{s'} T(s'|s, a)[\text{Reward}(s, a, s') + \gamma V_\pi(s')]$$

Policy evaluation

Key idea: iterative algorithm

Start with arbitrary policy values and repeatedly apply recurrences to converge to true values.

Algorithm: policy evaluation

Initialize $V_\pi^{(0)}(s) \leftarrow 0$ for all states s.

For iteration $t = 1, \dots, t_{\text{PE}}$:

For each state s:

$$V_\pi^{(t)}(s) \leftarrow \underbrace{\sum_{s'} T(s'|s,\pi(s))[\text{Reward}(s,\pi(s),s') + \gamma V_\pi^{(t-1)}(s')]}_{Q^{(t-1)}(s,\pi(s))}$$

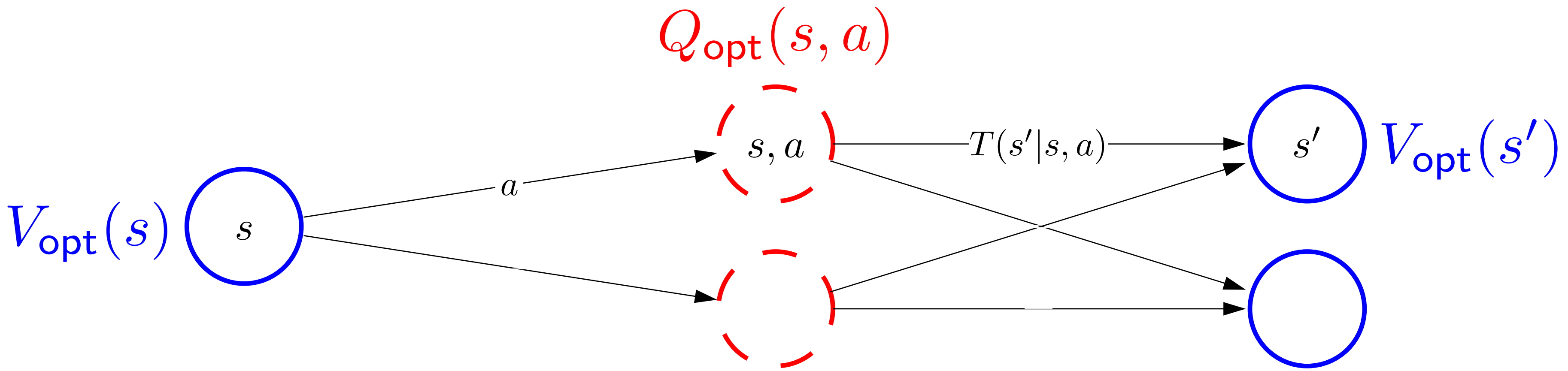
Optimal value and policy

Goal: try to get directly at maximum expected utility

Definition: optimal value

The **optimal value** $V_{\text{opt}}(s)$ is the maximum value attained by any policy.

Optimal values and Q-values



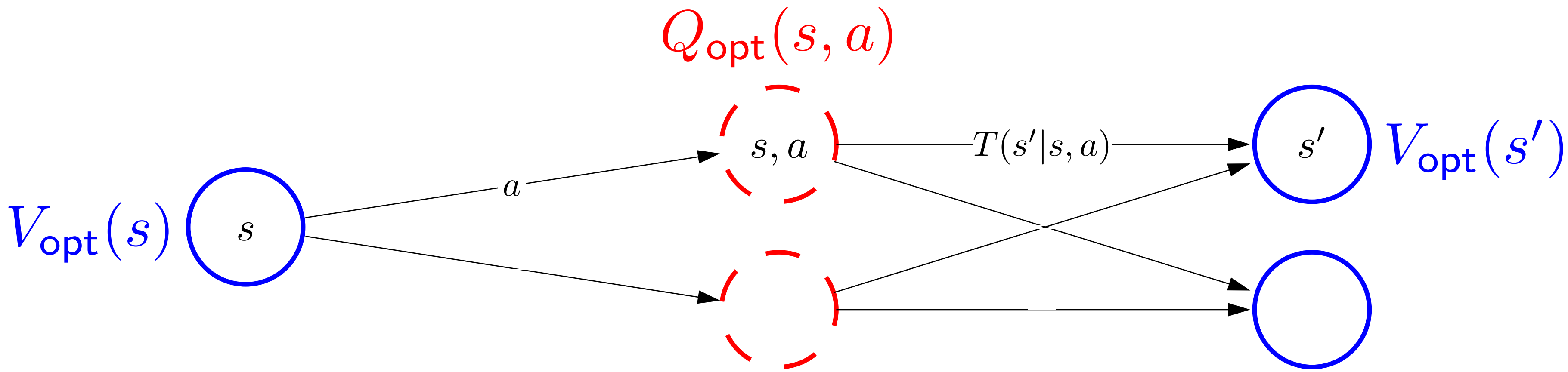
Optimal value if take action a in state s:

$$Q_{\text{opt}}(s,a) = \sum_{s'} T(s,a,s')[\text{Reward}(s,a,s') + \gamma V_{\text{opt}}(s')].$$

Optimal value from state s:

$$V_{\text{opt}}(s) = \begin{cases} 0 & \text{if IsEnd}(s) \\ \max_{a \in \text{Actions}(s)} Q_{\text{opt}}(s,a) & \text{otherwise.} \end{cases}$$

Optimal policies



Given Q_{opt}, read off the optimal policy:

$$\pi_{\text{opt}}(s) = \arg\max_{a \in \text{Actions}(s)} Q_{\text{opt}}(s, a)$$

Value iteration

Algorithm: value iteration [Bellman, 1957]

Initialize $V_{\text{opt}}^{(0)}(s) \leftarrow 0$ for all states s.

For iteration $t = 1, \dots, t_{\text{VI}}$:

For each state s:

$$V_{\text{opt}}^{(t)}(s) \leftarrow \max_{a \in \text{Actions}(s)} \underbrace{\sum_{s'} T(s,a,s')[\text{Reward}(s,a,s') + \gamma V_{\text{opt}}^{(t-1)}(s')]}_{Q_{\text{opt}}^{(t-1)}(s,a)}$$

Time: $O(t_{\text{VI}} SAS')$

[semi-live solution]

Convergence

Theorem: convergence

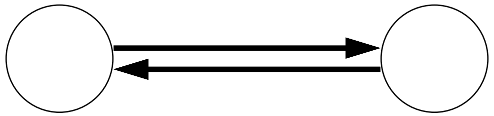
Suppose either

- discount $\gamma < 1$, or
- MDP graph is acyclic.

Then value iteration converges to the correct answer.

Example: non-convergence

discount $\gamma = 1$, zero rewards



Unknown transitions and rewards

Definition: Markov decision process

States: the set of states

$s_{\text{start}} \in$ States: starting state

Actions(s): possible actions from state s

IsEnd(s): whether at end of game

$0 \leq \gamma \leq 1$: discount factor (default: 1)

reinforcement learning!

From MDPs to reinforcement learning

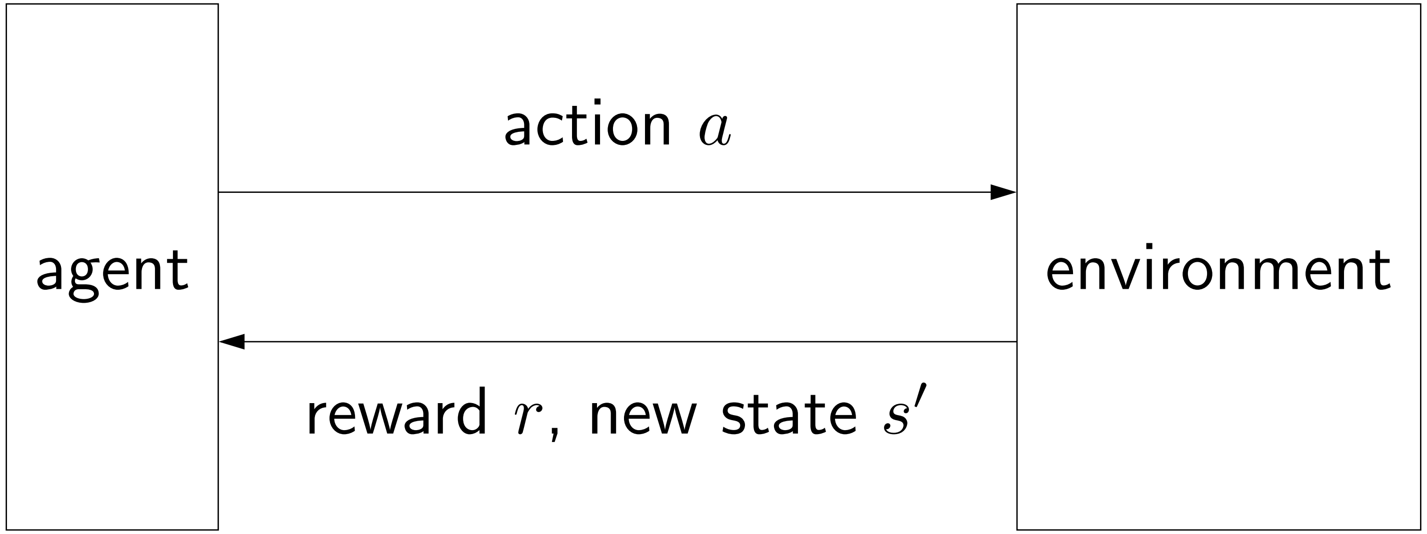
Markov decision process (offline)

- Have mental model of how the world works.
- Find policy to collect maximum rewards.

Reinforcement learning (online)

- Don't know how the world works.
- Perform actions in the world to find out and collect rewards.

Reinforcement learning framework



Algorithm: reinforcement learning template

For $t = 1, 2, 3, \ldots$

Choose action $a_t = \pi_{\text{act}}(s_{t-1})$ (**how?**)

Receive reward r_t and observe new state s_t

Update parameters (**how?**)

Model-Based Value Iteration

Data: $s_0; a_1, r_1, s_1; a_2, r_2, s_2; a_3, r_3, s_3; \ldots; a_n, r_n, s_n$

Key idea: model-based learning

Estimate the MDP: $T(s, a, s')$ and $\text{Reward}(s, a, s')$

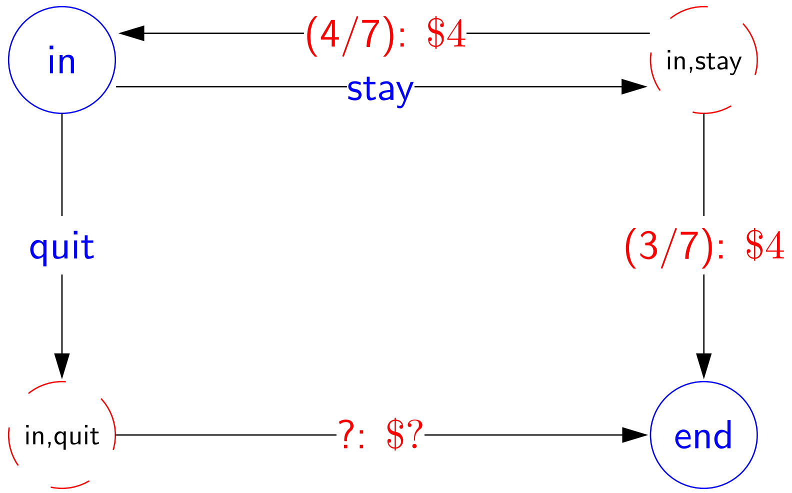
Transitions:

$$\hat{T}(s, a, s') = \frac{\#\text{ times } (s, a, s') \text{ occurs}}{\#\text{ times } (s, a) \text{ occurs}}$$

Rewards:

$$\widehat{\text{Reward}}(s, a, s') = r \text{ in } (s, a, r, s')$$

Problem



Problem: won't even see (s, a) if $a \neq \pi(s)$ ($a =$ quit)

Key idea: exploration

To do reinforcement learning, need to explore the state space.

Solution: need π to **explore** explicitly (more on this later)

From model-based to model-free

$$\hat{Q}_{\text{opt}}(s,a) = \sum_{s'} \hat{T}(s,a,s')[\widehat{\text{Reward}}(s,a,s') + \gamma \hat{V}_{\text{opt}}(s')]$$

All that matters for prediction is (estimate of) $Q_{\text{opt}}(s,a)$.

Key idea: model-free learning

Try to estimate $Q_{\text{opt}}(s,a)$ directly.

Model-free Monte Carlo

Data (following policy π):

$$s_0; a_1, r_1, s_1; a_2, r_2, s_2; a_3, r_3, s_3; \ldots; a_n, r_n, s_n$$

Recall:

$Q_\pi(s, a)$ is expected utility starting at s, first taking action a, and then following policy π

Utility:

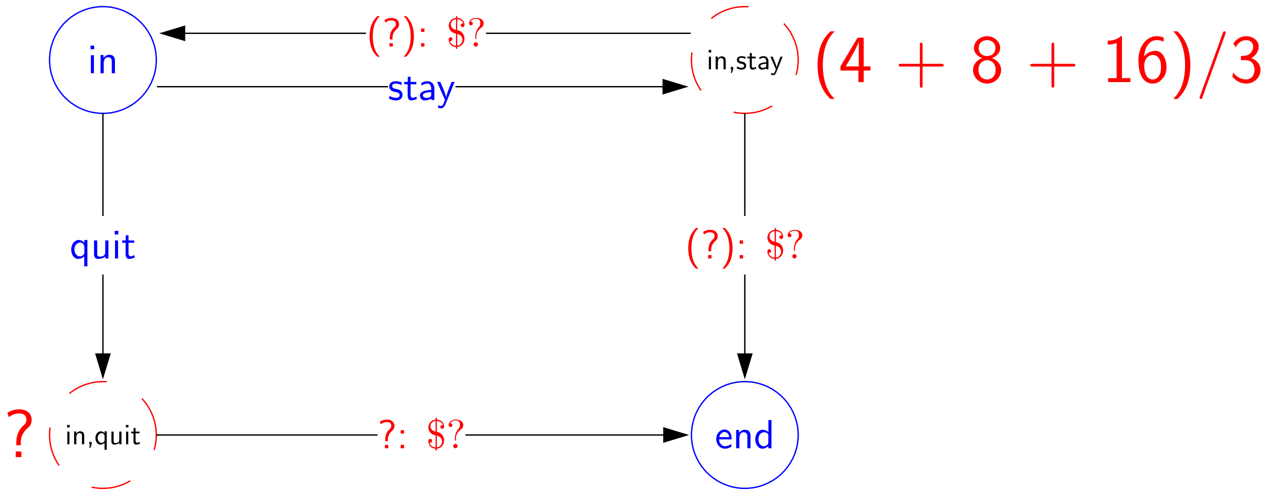
$$u_t = r_t + \gamma \cdot r_{t+1} + \gamma^2 \cdot r_{t+2} + \cdots$$

Estimate:

$\hat{Q}_\pi(s, a) =$ average of u_t where $s_{t-1} = s, a_t = a$

(and s, a doesn't occur in $s_0, \cdots, s_{t-2}$)

Model-free Monte Carlo



Data (following policy $\pi(s) =$ stay):

[in; stay, 4, in; stay, 4, in; stay, 4, in; stay, 4, end]

Note: we are estimating Q_π now, not Q_{opt}

Definition: on-policy versus off-policy

On-policy: estimate the value of data-generating policy

Off-policy: estimate the value of another policy

Q-learning

Bellman optimality equation:

$$Q_{\text{opt}}(s,a) = \sum_{s'} T(s,a,s')[\text{Reward}(s,a,s') + \gamma V_{\text{opt}}(s')]$$

Algorithm: Q-learning [Watkins/Dayan, 1992]

On each (s,a,r,s'):

$$\hat{Q}_{\text{opt}}(s,a) \leftarrow (1-\eta)\underbrace{\hat{Q}_{\text{opt}}(s,a)}_{\text{prediction}} + \eta\underbrace{(r + \gamma \hat{V}_{\text{opt}}(s'))}_{\text{target}}$$

Recall: $\hat{V}_{\text{opt}}(s') = \max_{a' \in \text{Actions}(s')} \hat{Q}_{\text{opt}}(s',a')$

Exploration/exploitation tradeoff

Key idea: balance

Need to balance **exploration** and **exploitation**.

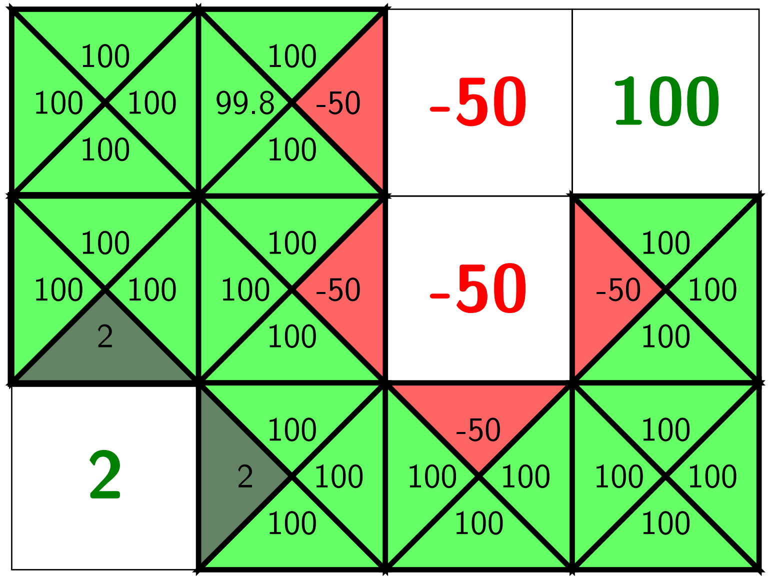
Examples from life: restaurants, routes, research

Epsilon-greedy

Algorithm: epsilon-greedy policy

$$\pi_{\text{act}}(s) = \begin{cases} \arg\max_{a \in \text{Actions}} \hat{Q}_{\text{opt}}(s,a) & \text{probability } 1-\epsilon, \\ \text{random from Actions}(s) & \text{probability } \epsilon. \end{cases}$$

Run (or press ctrl-enter)



Average (lifetime) utility: 31.75

a	r	s
		(2,1)
E	0	(2,2)
S	0	(3,2)
E	0	(3,3)
E	0	(3,4)
N	0	(2,4)
N	100	(1,4)

Function approximation

Key idea: linear regression model

Define **features** $\phi(s,a)$ and **weights** $\mathbf{w}$:

$$\hat{Q}_{\text{opt}}(s,a;\mathbf{w}) = \mathbf{w} \cdot \phi(s,a)$$

Example: features for volcano crossing

$\phi_1(s,a) = \mathbf{1}[a = \text{W}]$

$\phi_2(s,a) = \mathbf{1}[a = \text{E}]$

...

$\phi_7(s,a) = \mathbf{1}[s = (5,*)]$

$\phi_8(s,a) = \mathbf{1}[s = (*,6)]$

...

Function approximation

Algorithm: Q-learning with function approximation

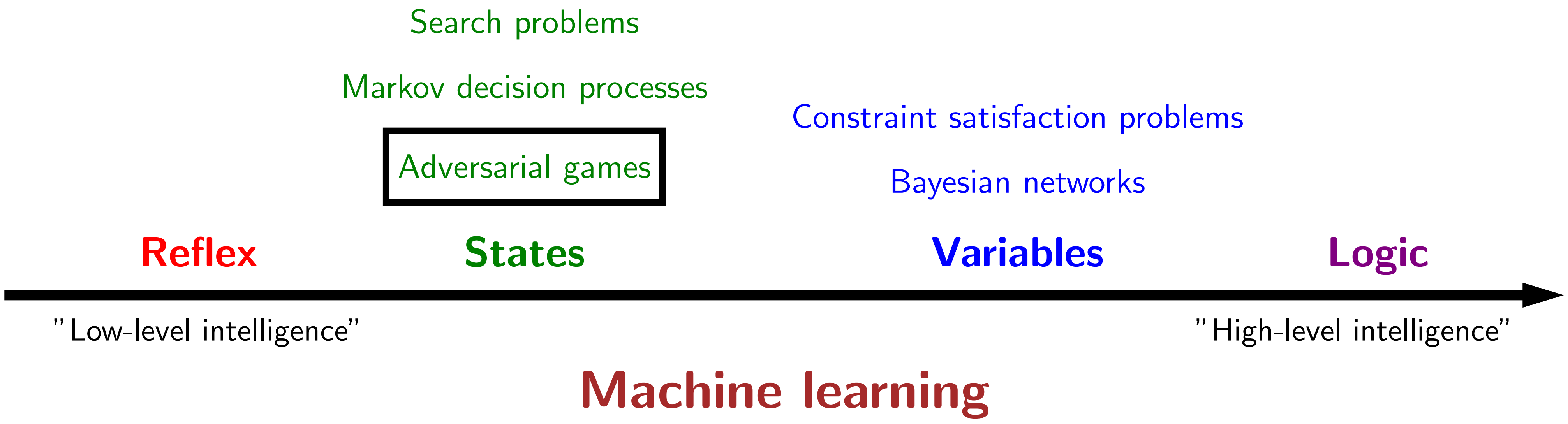
On each (s, a, r, s'):

$$\mathbf{w} \leftarrow \mathbf{w} - \eta[\underbrace{\hat{Q}_{\text{opt}}(s, a; \mathbf{w})}_{\text{prediction}} - \underbrace{(r + \gamma \hat{V}_{\text{opt}}(s'))}_{\text{target}}]\phi(s, a)$$

Implied objective function:

$$(\underbrace{\hat{Q}_{\text{opt}}(s, a; \mathbf{w})}_{\text{prediction}} - \underbrace{(r + \gamma \hat{V}_{\text{opt}}(s'))}_{\text{target}})^2$$

Course plan



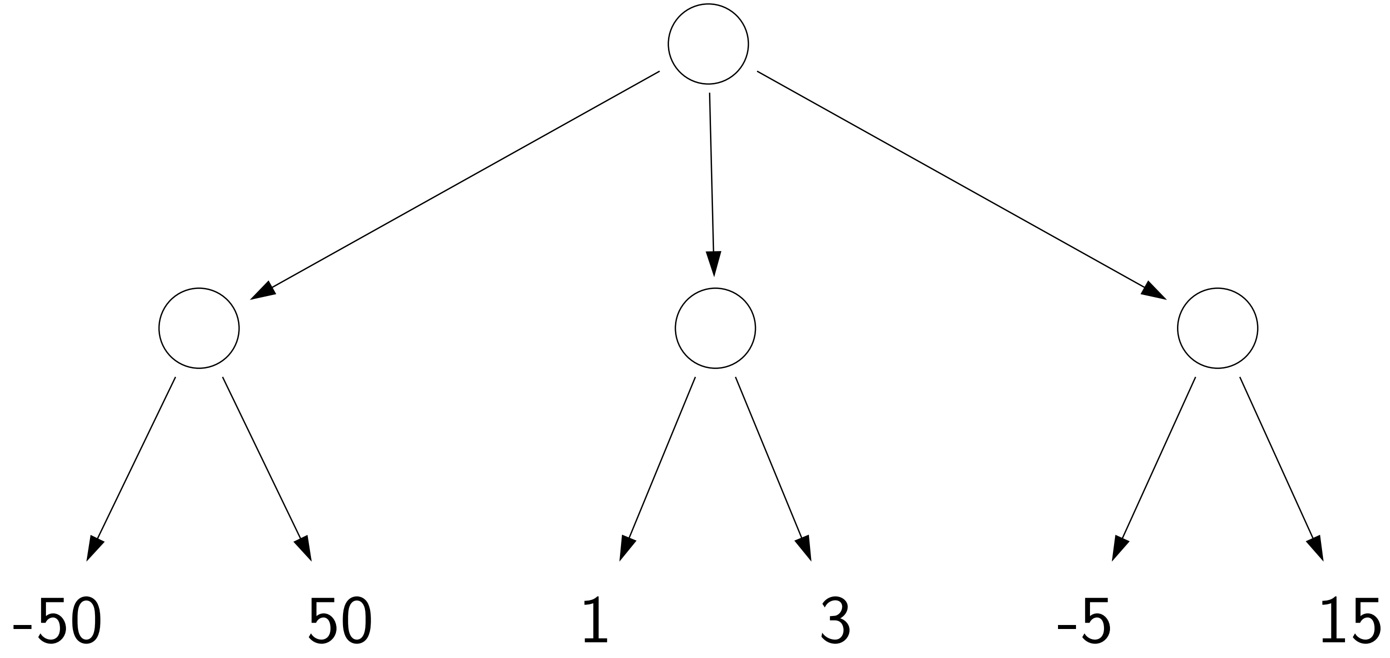
Search problems
Markov decision processes
Adversarial games
Constraint satisfaction problems
Bayesian networks
Reflex
States
Variables
Logic
"Low-level intelligence"
"High-level intelligence"
Machine learning

Game tree

Key idea: game tree

Each node is a decision point for a player.

Each root-to-leaf path is a possible outcome of the game.



Two-player zero-sum games

Players $= \{\text{agent}, \text{opp}\}$

Definition: two-player zero-sum game

s_{start}: starting state

$\text{Actions}(s)$: possible actions from state s

$\text{Succ}(s, a)$: resulting state if choose action a in state s

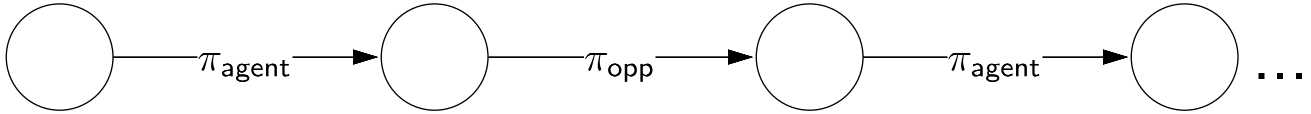
$\text{IsEnd}(s)$: whether s is an end state (game over)

$\text{Utility}(s)$: agent's utility for end state s

$\text{Player}(s) \in \text{Players}$: player who controls state s

Game evaluation recurrence

Analogy: recurrence for policy evaluation in MDPs

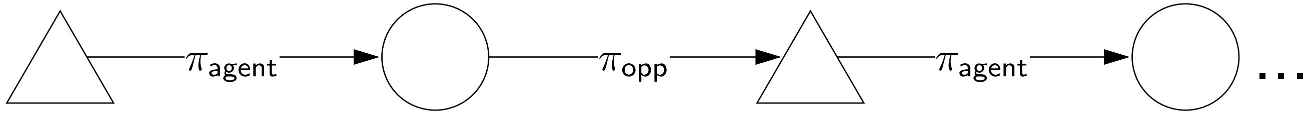


Value of the game:

$$V_{\text{eval}}(s) = \begin{cases} \text{Utility}(s) & \text{IsEnd}(s) \\ \sum_{a \in \text{Actions}(s)} \pi_{\text{agent}}(s,a) V_{\text{eval}}(\text{Succ}(s,a)) & \text{Player}(s) = \text{agent} \\ \sum_{a \in \text{Actions}(s)} \pi_{\text{opp}}(s,a) V_{\text{eval}}(\text{Succ}(s,a)) & \text{Player}(s) = \text{opp} \end{cases}$$

Expectimax recurrence

Analogy: recurrence for value iteration in MDPs

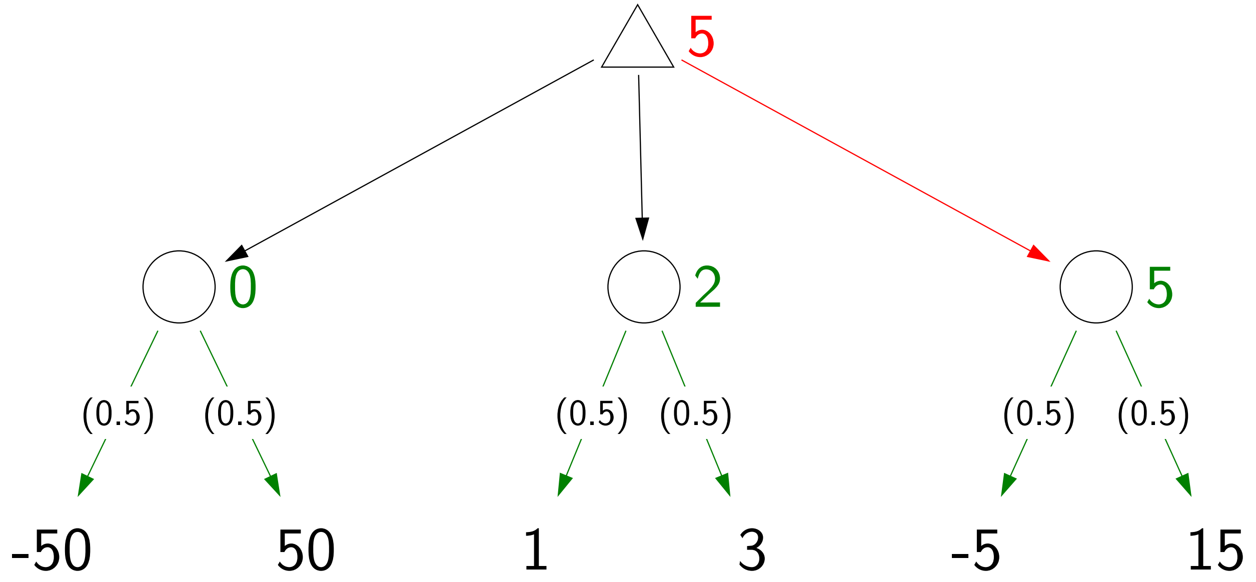


$$V_{\text{exptmax}}(s) = \begin{cases} \text{Utility}(s) & \text{IsEnd}(s) \\ \max_{a \in \text{Actions}(s)} V_{\text{exptmax}}(\text{Succ}(s,a)) & \text{Player}(s) = \text{agent} \\ \sum_{a \in \text{Actions}(s)} \pi_{\text{opp}}(s,a) V_{\text{exptmax}}(\text{Succ}(s,a)) & \text{Player}(s) = \text{opp} \end{cases}$$

Expectimax example

Example: expectimax

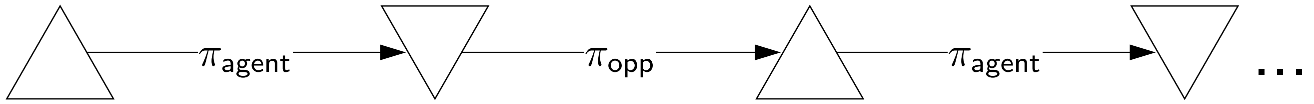
$\pi_{\text{opp}}(s, a) = \frac{1}{2}$ for $a \in \text{Actions}(s)$



$$\boxed{V_{\text{exptmax}}(s_{\text{start}}) = 5}$$

Minimax recurrence

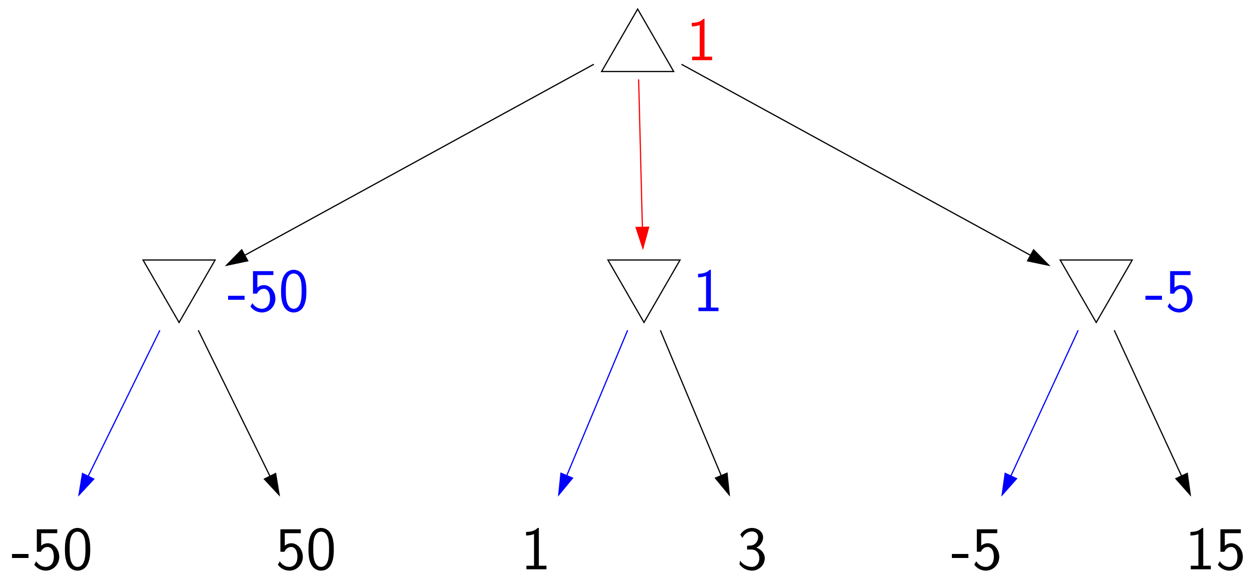
No analogy in MDPs:



$$V_{\text{minmax}}(s) = \begin{cases} \text{Utility}(s) & \text{IsEnd}(s) \\ \max_{a \in \text{Actions}(s)} V_{\text{minmax}}(\text{Succ}(s,a)) & \text{Player}(s) = \text{agent} \\ \min_{a \in \text{Actions}(s)} V_{\text{minmax}}(\text{Succ}(s,a)) & \text{Player}(s) = \text{opp} \end{cases}$$

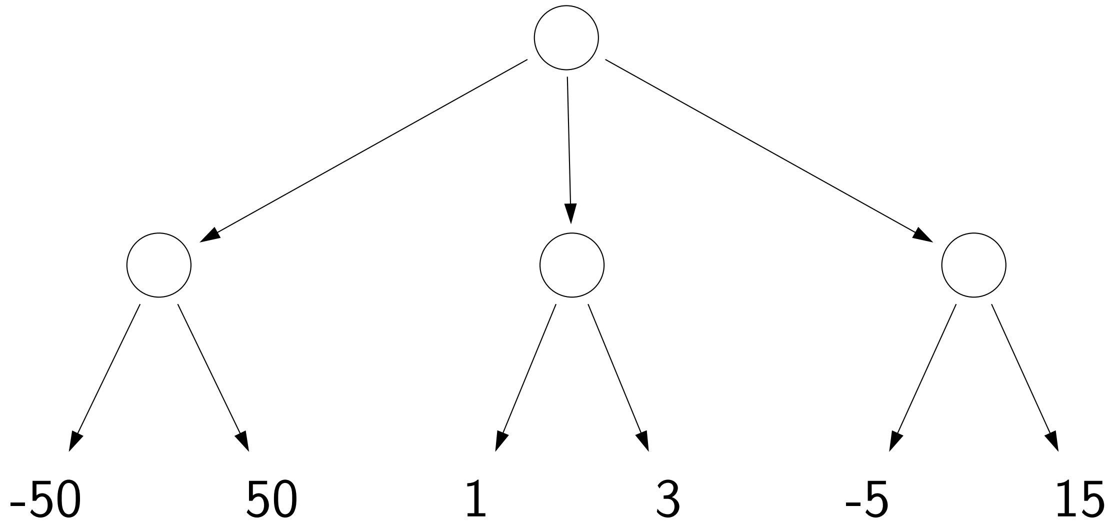
Minimax example

Example: minimax



$$\boxed{V_{\text{minmax}}(s_{\text{start}}) = 1}$$

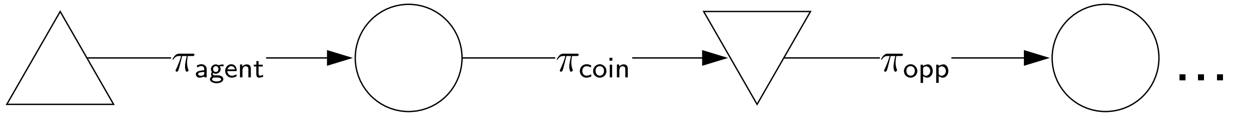
Relationship between game values



	$\pi_{\min}$		π_7
$\pi_{\max}$	$V(\pi_{\max}, \pi_{\min})$ 1	$\leq$	$V(\pi_{\max}, \pi_7)$ 2
	$\geq$		$\leq$
$\pi_{\text{exptmax}(7)}$	$V(\pi_{\text{exptmax}(7)}, \pi_{\min})$ -5		$V(\pi_{\text{exptmax}(7)}, \pi_7)$ 5

Expectiminimax recurrence

Players $= \{\text{agent}, \text{opp}, \text{coin}\}$

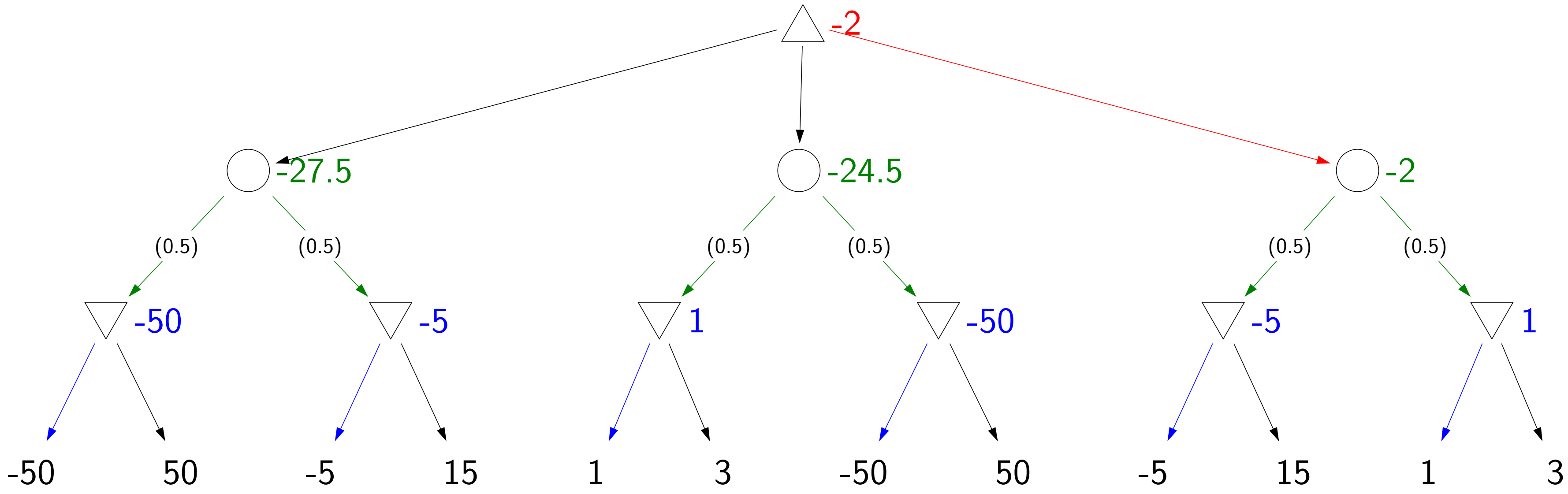


$$V_{\text{exptminmax}}(s) = \begin{cases} \text{Utility}(s) & \text{IsEnd}(s) \\ \max_{a \in \text{Actions}(s)} V_{\text{exptminmax}}(\text{Succ}(s,a)) & \text{Player}(s) = \text{agent} \\ \min_{a \in \text{Actions}(s)} V_{\text{exptminmax}}(\text{Succ}(s,a)) & \text{Player}(s) = \text{opp} \\ \sum_{a \in \text{Actions}(s)} \pi_{\text{coin}}(s,a) V_{\text{exptminmax}}(\text{Succ}(s,a)) & \text{Player}(s) = \text{coin} \end{cases}$$

Expectiminimax example

Example: expectiminimax

$\pi_{\text{coin}}(s, a) = \frac{1}{2}$ for $a \in \{0, 1\}$



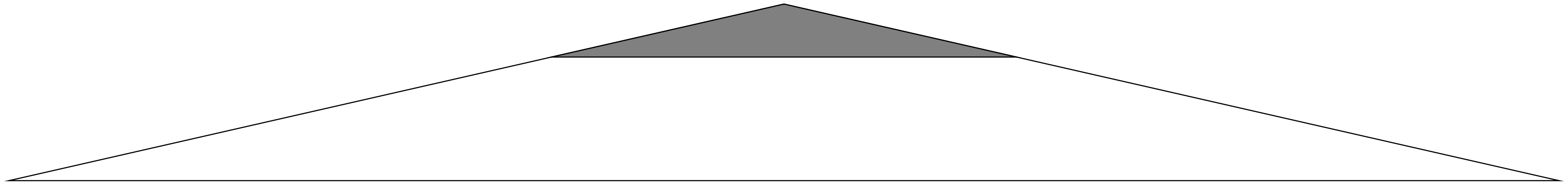
$$\boxed{V_{\text{exptminmax}}(s_{\text{start}}) = -2}$$

Speeding up minimax

- Evaluation functions: use domain-specific knowledge, compute approximate answer

- Alpha-beta pruning: general-purpose, compute exact answer

Depth-limited search



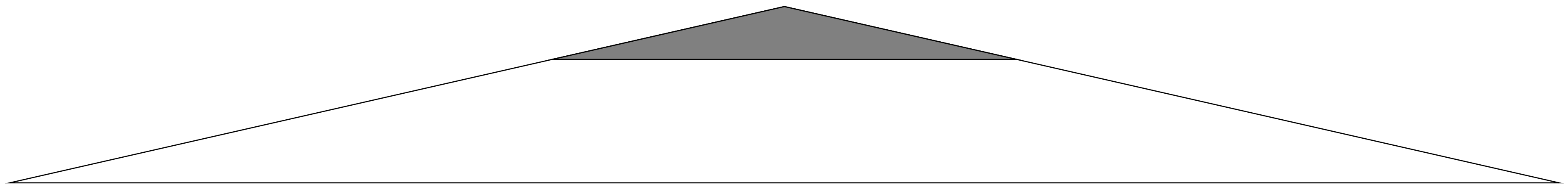
Limited depth tree search (stop at maximum depth d_{max}):

$$V_{\text{minmax}}(s, d) = \begin{cases} \text{Utility}(s) & \text{IsEnd}(s) \\ \text{Eval}(s) & d = 0 \\ \max_{a \in \text{Actions}(s)} V_{\text{minmax}}(\text{Succ}(s, a), d) & \text{Player}(s) = \text{agent} \\ \min_{a \in \text{Actions}(s)} V_{\text{minmax}}(\text{Succ}(s, a), d - 1) & \text{Player}(s) = \text{opp} \end{cases}$$

Use: at state s, call $V_{\text{minmax}}(s, d_{\text{max}})$

Convention: decrement depth at last player's turn

Evaluation functions



Definition: Evaluation function

An evaluation function $\text{Eval}(s)$ is a (possibly very weak) estimate of the value $V_{\text{minmax}}(s)$.

Analogy: $\text{FutureCost}(s)$ in search problems

Pruning principle

Choose A or B with maximum value:

A: [3, **5**]

B: [**5**, 100]

Key idea: branch and bound

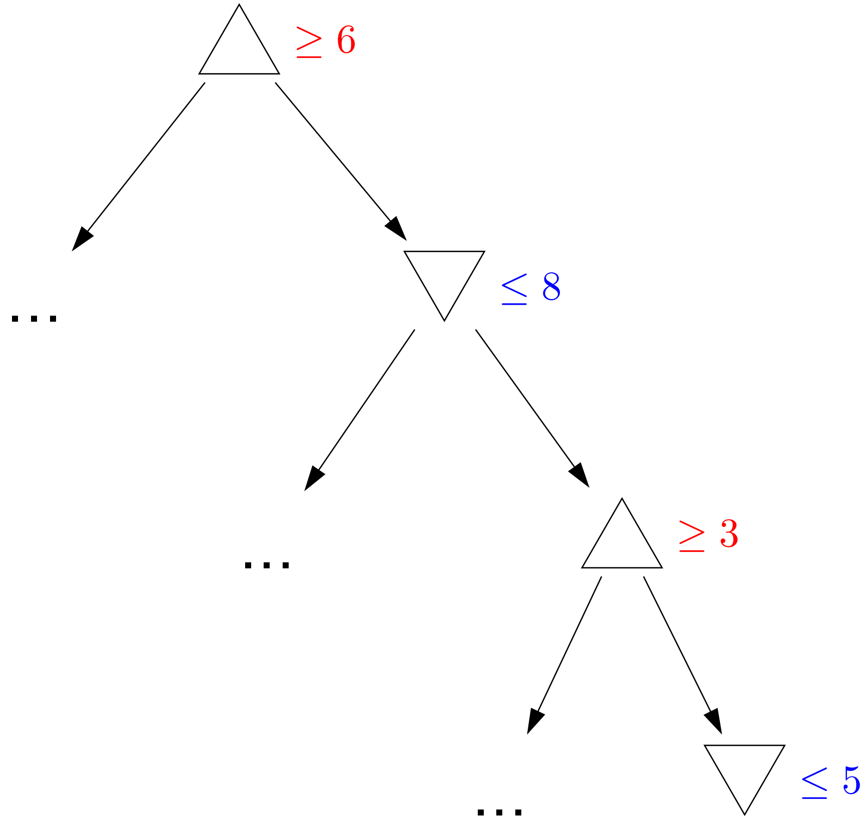
Maintain lower and upper bounds on values.

If intervals don't overlap non-trivially, then can choose optimally without further work.

Alpha-beta pruning

Key idea: optimal path

The optimal path is path that minimax policies take.
Values of all nodes on path are the same.



- a_s: lower bound on value of max node s
- b_s: upper bound on value of min node s
- Prune a node if its interval doesn't have non-trivial overlap with every ancestor (store $\alpha_s = \max_{s' \preceq s} a_{s'}$ and $\beta_s = \min_{s' \preceq s} b_{s'}$)