CS 540 Introduction to Artificial Intelligence
Review

University of Wisconsin-Madison
Spring 2025

Final Information

- **Time: May 7th 07:45 AM - 09:45 AM**
- **Location (by section**):**
  - **Lecture 001 (Instructor Sharon Li): S429 Chemistry Building**
  - **Lecture 002 (Instructor Fred Sala): 1220 Microbial Sciences**
  - **Lecture 003 (Instructor Blerina Gkotse): B10 Ingraham Hall**

  To find your section go to MyUW->Course Schedule->It will say "LEC 00_". **Do not use canvas to find your section (everyone will see CS540 001 since we merged the canvas site for all three sections).

- **Format**: The final exam will be entirely multiple choice.
- **Cheat Sheet**: You will be allowed a cheat sheet of a single piece of paper (8.5" x 11", front and back). The exam will focus on conceptual and applied AI reasoning.
- **Calculator**: Calculators are allowed if they don't have an internet connection. A calculator will not be necessary though it may be useful to double check simple arithmetic.
- **Detailed topic list + practice**: https://piazza.com/class/m5zvrf0clyo3sl/post/449

Survey!

- **If we hit 50%, we'll narrow down the exam information.**
- Current:

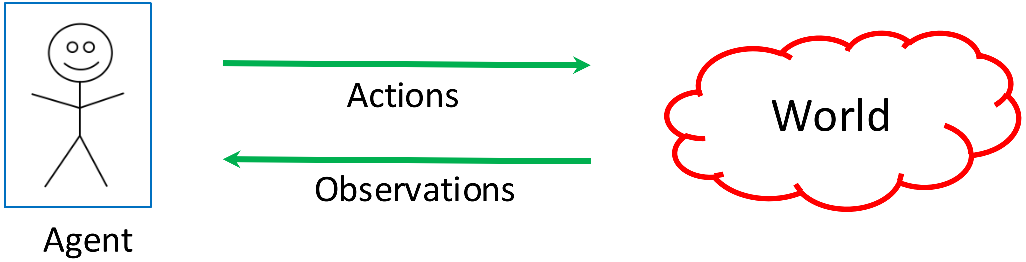
  COMP SCI 540-002
  2025 Spring
  Ends: 2025-05-02 (1 days)
  Results Available: 2025-05-16

- Deadline: **Tomorrow night** (5/2 at midnight).

Reinforcement Learning

Building The Theoretical Model



Basic setup:

- Set of states, S
- Set of actions A
- Information: at time t, observe state $s_t \in$ S. Get reward r_t
- Agent makes choice $a_t \in$ A. State changes to s_{t+1}, continue

Goal: find a map from **states to actions** maximize rewards.

A “policy”

Markov Decision Process (MDP)

The formal mathematical model:

- **State set** S. Initial state s_0. **Action set** A
- **State transition model**: $P(s_{t+1} | s_t, a_t)$
  - Markov assumption: transition probability only depends on s_t and a_t, and not previous actions or states.
- **Reward function:** $r(s_t)$
- **Policy**: $\pi(s) : S \rightarrow A$, action to take at a particular state.

$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \ldots$$

Discounting Rewards

One issue: these are infinite series. **Convergence**?

- Solution

$$U(s_0, s_1 \ldots) = r(s_0) + \gamma r(s_1) + \gamma^2 r(s_2) + \ldots = \sum_{t \geq 0} \gamma^t r(s_t)$$

- Discount factor γ between 0 and 1
  - Set according to how important **present** is VS **future**
  - Note: has to be less than 1 for convergence

Values and Policies

- Now that $V^{\pi}(s_0)$ is defined what *a* should we take?
  - First, set V*(s) to be expected utility for **optimal** policy from s
  - What's the expected utility of an action?
    - Specifically, action a in state s?

$$\sum_{s'} P(s'|s,a)V^*(s')$$

All the states we could go to

Transition probability

Expected rewards

Obtaining the Optimal Policy

Assume, we know the expected utility of an action.

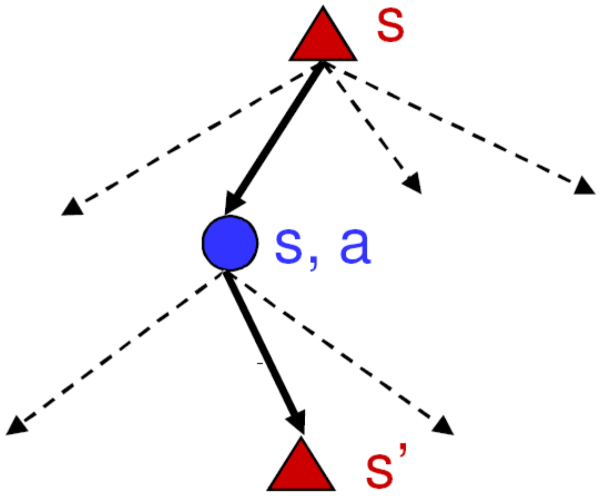
- So, to get the optimal policy, compute

$$\pi^*(s) = \operatorname{argmax}_a \sum_{s'} P(s'|s,a)V^*(s')$$

All the states we could go to

Transition probability

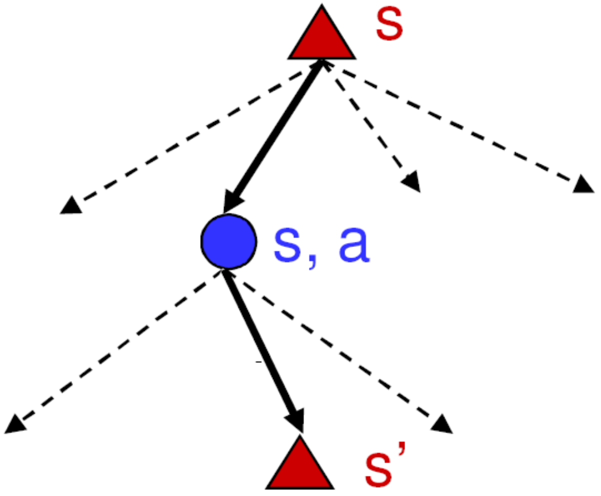
Expected rewards



Credit L. Lazbenik

Bellman Equations

Let’s walk over one step for the value function:



Credit L. Lazbenik

$$V^*(s) = r(s) + \gamma \max_{a} \sum_{s'} P(s'|s,a)V^*(s')$$

Current state reward

Discounted expected future **rewards**

Richard Bellman: Inventor of dynamic programming.

Value Iteration

Q: how do we find $V^*(s)$?

- Why do we want it? Can use it to get the best policy
- Know: reward $r(s)$, transition probability $P(s'|s,a)$
  - Knowing r and P is the “planning” problem. In reality r and P must be estimated from interactions : “reinforcement learning”
- Also know $V^*(s)$ satisfies Bellman equation (recursion above)

A: Use the property. Start with $V_0(s)=0$. Then, update

$$V_{i+1}(s) = r(s) + \gamma \max_{a} \sum_{s'} P(s'|s,a) V_i(s')$$

Q-Learning

- Our **next** reinforcement learning algorithm.
- Does not require knowing *r* or *P*. Learn from data of the form: $\{(s_t, a_t, r_t, s_{t+1})\}$.
- Learns an action-value function *Q**(*s*,*a*) that tells us the expected value of taking *a* in state *s*.
  - Note: $V^*(s) = \max_a Q^*(s,a)$.
- Optimal policy is formed as $\pi^*(s) = \mathrm{arg}\max_a Q^*(s,a)$

Q-Learning Iteration

How do we get Q(*s*,*a*)?

- Iterative procedure

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r(s_t) + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$$

Learning rate

Idea: combine old value and new estimate of future value.

Note: We are using a policy to take actions; based on the estimated Q!

Q-Learning

Estimate Q*(*s*,*a*) from data $\{(s_t, a_t, r_t, s_{t+1})\}$:

1. Initialize Q(.,.) arbitrarily (eg all zeros)
    1. Except terminal states $Q(s_{terminal},.)=0$
2. Iterate over data until Q(.,.) converges:

$$Q(s_t, a_t) \leftarrow (1-\alpha)Q(s_t, a_t) + \alpha(r_t + \gamma \max_b Q(s_{t+1}, b))$$

Learning rate

Exploration Vs. Exploitation

General question!

- **Exploration:** take an action with unknown consequences
  - Pros:
    - Get a more accurate model of the environment
    - Discover higher-reward states than the ones found so far
  - Cons:
    - When exploring, not maximizing your utility
    - Something bad might happen
- **Exploitation:** go with the best strategy found so far
  - Pros:
    - Maximize reward as reflected in the current utility estimates
    - Avoid bad stuff
  - Cons:
    - Might prevent you from discovering the true optimal strategy

Q-Learning: ε-Greedy Behavior Policy

Getting data with both **exploration and exploitation**

- With probability ε, take a random action; else the action with the highest (current) Q(*s*,*a*) value.

$$a = \begin{cases} \operatorname{argmax}_{a \in A} Q(s, a) & \text{uniform}(0,1) > \epsilon \\ \text{random } a \in A & \text{otherwise} \end{cases}$$

Q-learning Algorithm

Input: step size α, exploration probability ϵ

1. set Q(*s*,*a*) = 0 for all *s*, *a*.
2. For each episode:
3. Get initial state *s*.
4. While (*s* not a terminal state):
5. Perform a = ϵ-greedy(Q, *s*), receive *r*, *s*'
6. $Q(s,a) = (1-\alpha)Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a'))$
7. $s \leftarrow s'$
8. End While
9. End For

Explore: take action to see what happens.

Update action-value based on result.

RL Practice Problems

25. Supposed you have the following information about an environment:

    1. The discount factor is 0.8
    2. The reward in s_1 taking action a_1 is 3
    3. The transition probabilities are: $P(s_2|s_1, a_1) = 0.6$ and $P(s_3|s_1, a_1) = 0.4$
    4. Currently, $V(s_2) = 10$ and $V(s_3) = 6$

    Remember the update for value iteration is $V_{t+1}(s) = r(s) + \gamma \max_a \sum_{s'} P(s'|s, a)V(s')$. After a single iteration of value iteration, what is the value for state s_1 (what is $V(s_1)$)? Choose the **closest** option.

    A. 8

    B. 10

    C. 12

    D. 14

    E. None of the above.

RL Practice Problems

24. What is NOT true about Q-learning with ϵ-greedy exploration where $\epsilon = 0.1$?

    A. At every step, the agent selects the action with the highest Q-value with probability $1 - \epsilon$.

    B. The point of a non-zero ϵ is to encourage exploration.

    C. The agent selects a random action with probability ϵ.

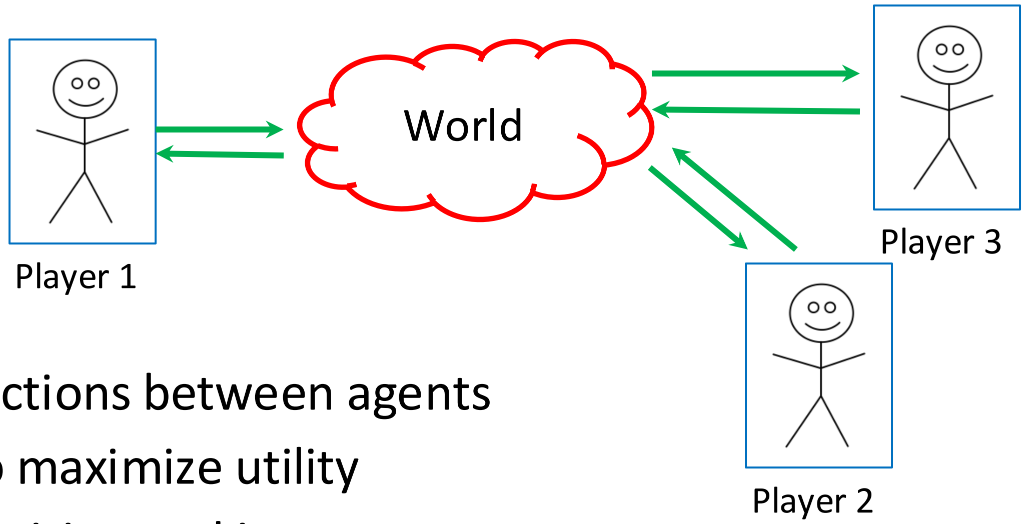
    D. The policy can achieve the maximum possible reward.

    E. None of the above.

Games

Games Setup

Games setup: **multiple** agents



- Now: interactions between agents
- Still want to maximize utility
- **Strategic** decision making.

Normal Form Game

Mathematical description of simultaneous games.

- $\boldsymbol{n}$ players $\{1,2,\ldots,n\}$
- Player i chooses strategy $\boldsymbol{a_i}$ from action space $\boldsymbol{A_i}$.
- Strategy profile: $\boldsymbol{a} = (\boldsymbol{a_1}, \boldsymbol{a_2}, \ldots, \boldsymbol{a_n})$
- Player i gets rewards $\boldsymbol{u_i}(\boldsymbol{a})$
  - **Note**: reward depends on other players!

- We consider the simple case where all reward functions are common knowledge.

Example of Normal Form Game

Ex: Prisoner's Dilemma

Player 2 Player 1	*Stay silent*	*Betray*
Stay silent	−1, −1	−3, 0
Betray	0, −3	−2, −2

- 2 players, 2 actions: yields 2x2 payoff matrix
- Strategy set: {Stay silent, betray}

Strictly Dominant Strategies

Let's analyze such games. Some strategies are better than others!

- Strictly dominant strategy: if a_i strictly better than b *regardless* of what other players do, a_i is **strictly dominant**
- I.e., $u_i(a_i, a_{-i}) > u_i(b, a_{-i}), \forall b \neq a_i, \forall a_{-i}$

All of the other entries of *a* excluding *i*

- Sometimes a dominant strategy does not exist!

Dominant Strategy Equilibrium

a^* is a (strictly) dominant strategy equilibrium (DSE), if every player i has a strictly dominant strategy a_i^*

- Rational players will play at DSE, if one exists.

Player 2 / Player 1	*Stay silent*	*Betray*
Stay silent	−1, −1	−3, 0
Betray	0, −3	−2, −2

Dominant Strategy: Absolute Best Responses

Player *i*'s best response to strategy to $a_{-i} BR(a_{-i}) = \underset{b}{\arg\max}\, u_i(b, a_{-i})$

BR(player2=silent) = betray
BR(player2=betray) = betray

Player 2 Player 1	*Stay silent*	*Betray*
Stay silent	−1, −1	−3, 0
Betray	0, −3	−2, −2

a_i^* is the dominant strategy for player *i*, if

$$a_i^* = BR(a_{-i}), \forall\, a_{-i}$$

Dominant Strategy Equilibrium

Dominant Strategy Equilibrium does not always exist.

Player 2 Player 1	*L*	*R*
T	2, 1	0, 0
B	0, 0	1, 2

Nash Equilibrium

a^* is a Nash equilibrium if no player has an incentive to **unilaterally deviate**

$$u_i(a_i^*, a_{-i}^*) \geq u_i(a_i, a_{-i}^*) \quad \forall a_i \in A_i$$

Player 2 / Player 1	*L*	*R*
T	2, 1	0, 0
B	0, 0	1, 2

Nash Equilibrium: Best Response to Each Other

a^* is a Nash equilibrium:

$$\forall i, \forall b \in A_i: u_i(a_i^*, a_{-i}^*) \geq u_i(b, a_{-i}^*)$$

(no player has an incentive to **unilaterally deviate**)

- Equivalently, for each player i:

$$a_i^* \in BR(a_{-i}^*) = argmax_b\, u_i(b, a_{-i}^*)$$

- Compared to DSE (a DSE is a NE, the other direction is generally not true):

$$a_i^* = BR(a_{-i}), \forall\, a_{-i}$$

Nash Equilibrium: Best Response to Each Other

a^* is a Nash equilibrium:

$$\forall i, \forall b \in A_i: u_i(a_i^*, a_{-i}^*) \geq u_i(b, a_{-i}^*)$$

(no player has an incentive to **unilaterally deviate**)

- Pure Nash equilibrium:
  - A **pure strategy** is a deterministic choice (no randomness).
  - Later: we will consider **mixed** strategies
  - In pure Nash equilibrium, players can only play pure strategies.

Finding (pure) Nash Equilibria by hand

- As player 1: For each column, find the best response, underscore it.

Player 2 Player 1	*L*	*R*
T	2, 1	0, 0
B	0, 0	1, 2

Finding (pure) Nash Equilibria by hand

- As player 2: For each row, find the best response, upper-score it.

Player 2 / Player 1	*L*	*R*
T	2, 1	0, 0
B	0, 0	1, 2

Finding (pure) Nash Equilibria by hand

- Entries with both lower and upper bars are pure NEs.

Player 2 Player 1	*L*	*R*
T	$\overline{\underline{2,\ 1}}$	0, 0
B	0, 0	$\overline{\underline{1,\ 2}}$

Pure Nash Equilibrium may not exist

So far, pure strategy: each player picks a deterministic strategy. But:

Player 2 / Player 1	*rock*	paper	*scissors*
rock	0, 0	-1, 1	1, -1
paper	1, -1	0, 0	-1, 1
scissors	-1, 1	1, -1	0, 0

Mixed Strategies

Can also randomize actions: "**mixed**"

- Player *i* assigns probabilities x_i to each action

$$x_i(a_i), \text{ where } \sum_{a_i \in A_i} x_i(a_i) = 1, x_i(a_i) \geq 0$$

- Now consider **expected rewards**

$$u_i(x_i, x_{-i}) = E_{a_i \sim x_i, a_{-i} \sim x_{-i}} u_i(a_i, a_{-i}) = \sum_{a_i} \sum_{a_{-i}} x_i(a_i) x_{-i}(a_{-i}) u_i(a_i, a_{-i})$$

Mixed Strategy Nash Equilibrium

Example: $x_1^*(\cdot) = x_2^*(\cdot) = \left(\frac{1}{3}, \frac{1}{3}, \frac{1}{3}\right)$

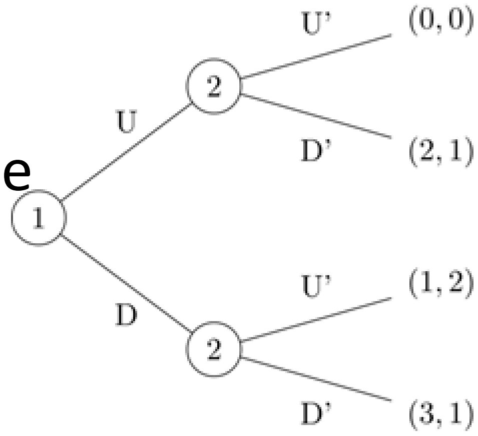
Player 2 Player 1	*rock*	paper	*scissors*
rock	0, 0	-1, 1	1, -1
paper	1, -1	0, 0	-1, 1
scissors	-1, 1	1, -1	0, 0

Sequential-Move Games

More complex games with multiple moves

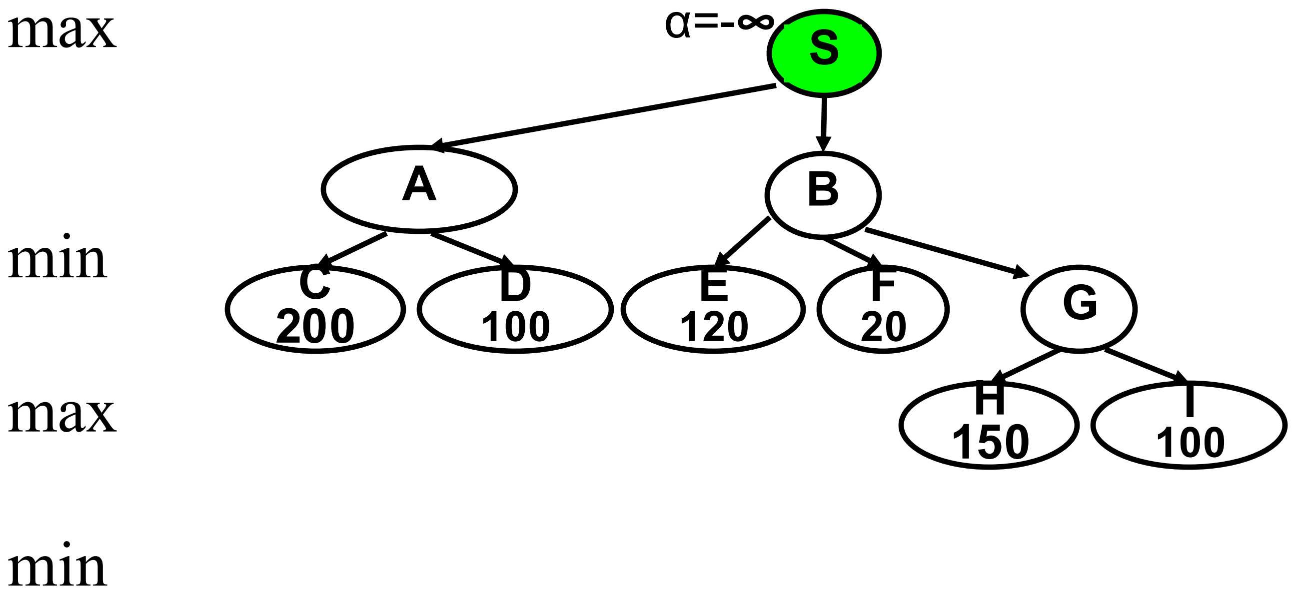
- Instead of normal form, **extensive form**
- Represent with a **tree**
- **Rewards at leaves**
- Find strategies: perform search over the tree

- Nash equilibrium still well-defined
  – Backward induction

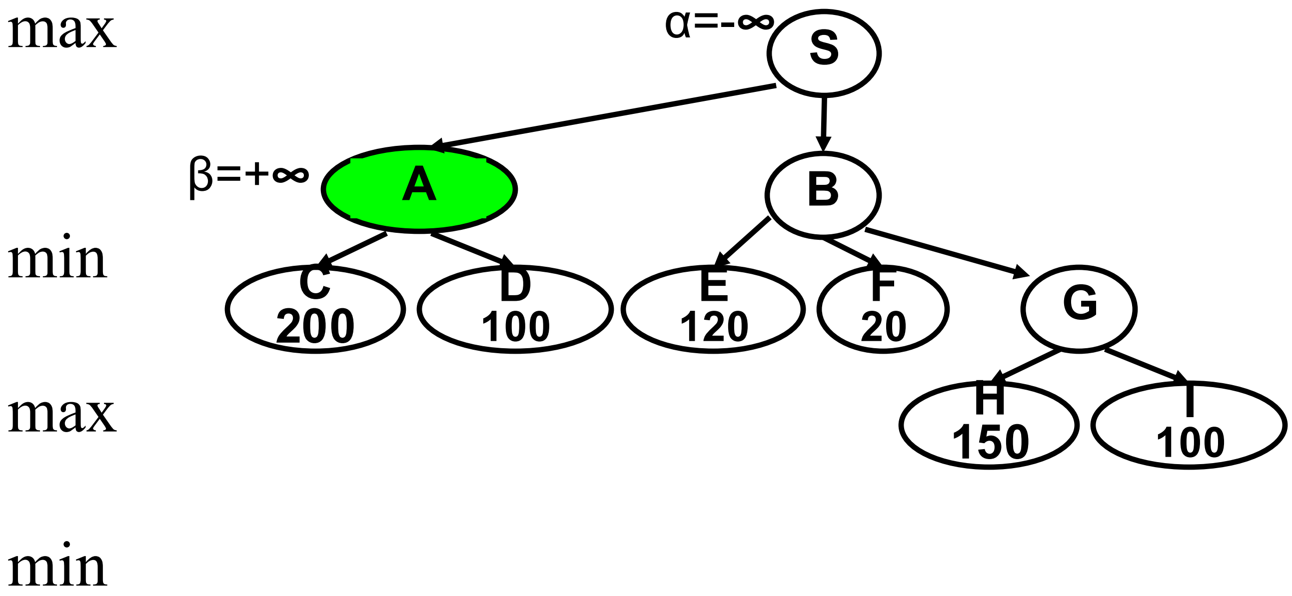


Wiki

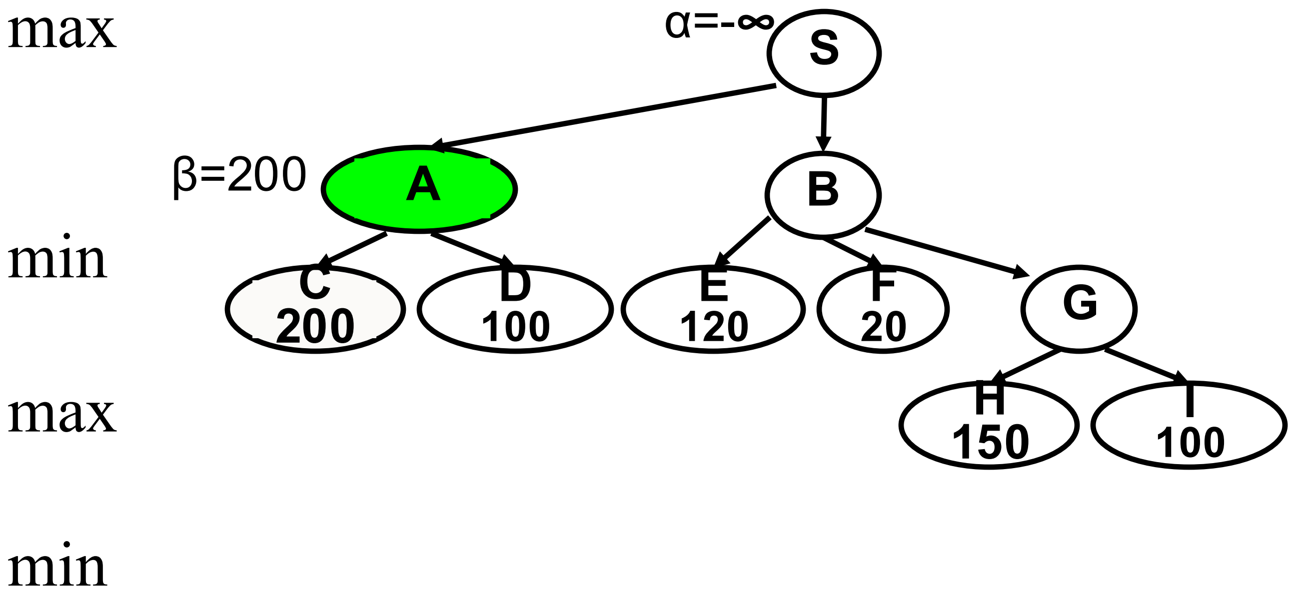
Minimax algorithm in execution



Minimax algorithm in execution

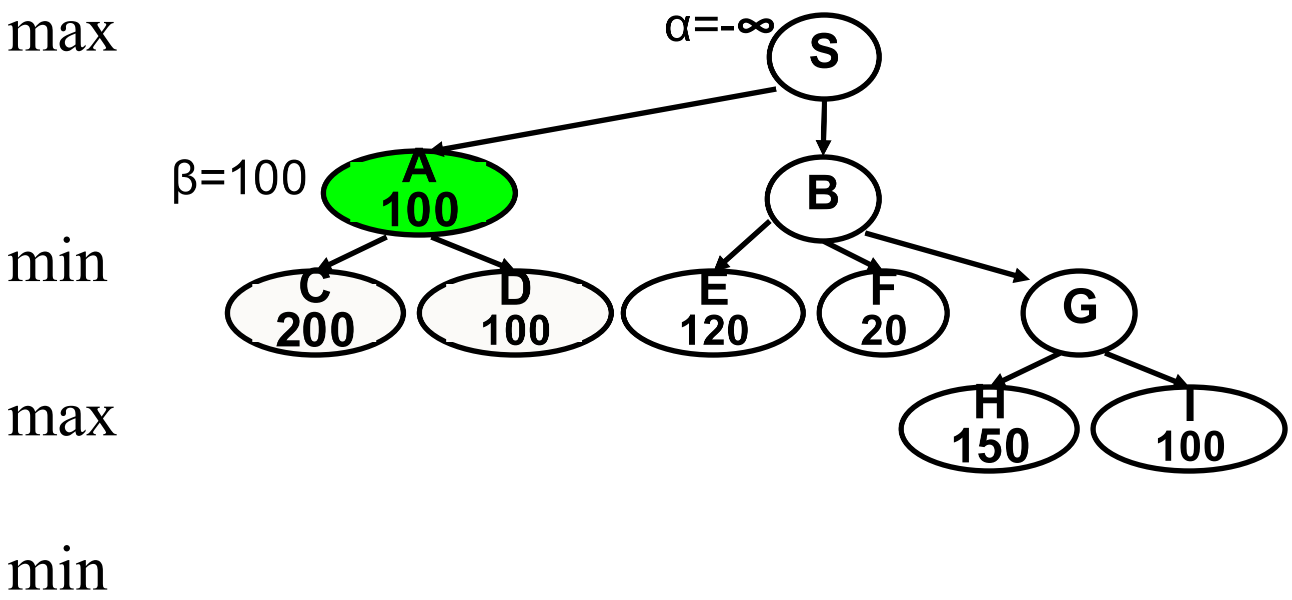


Minimax algorithm in execution

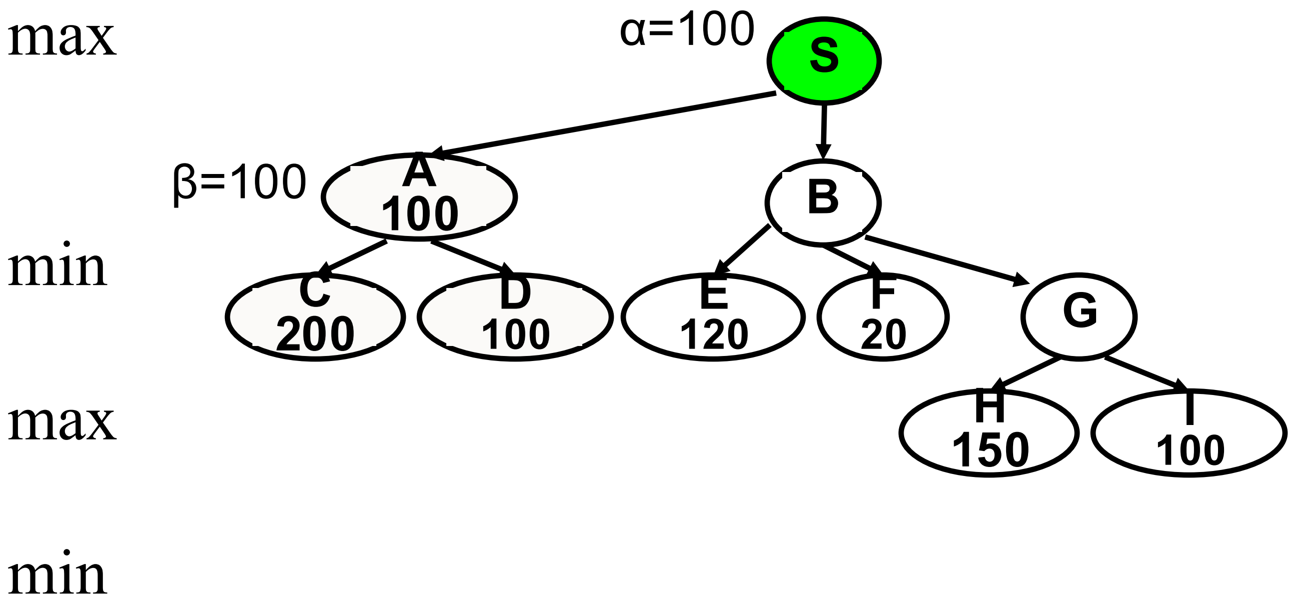


The execution on the terminal nodes is omitted.

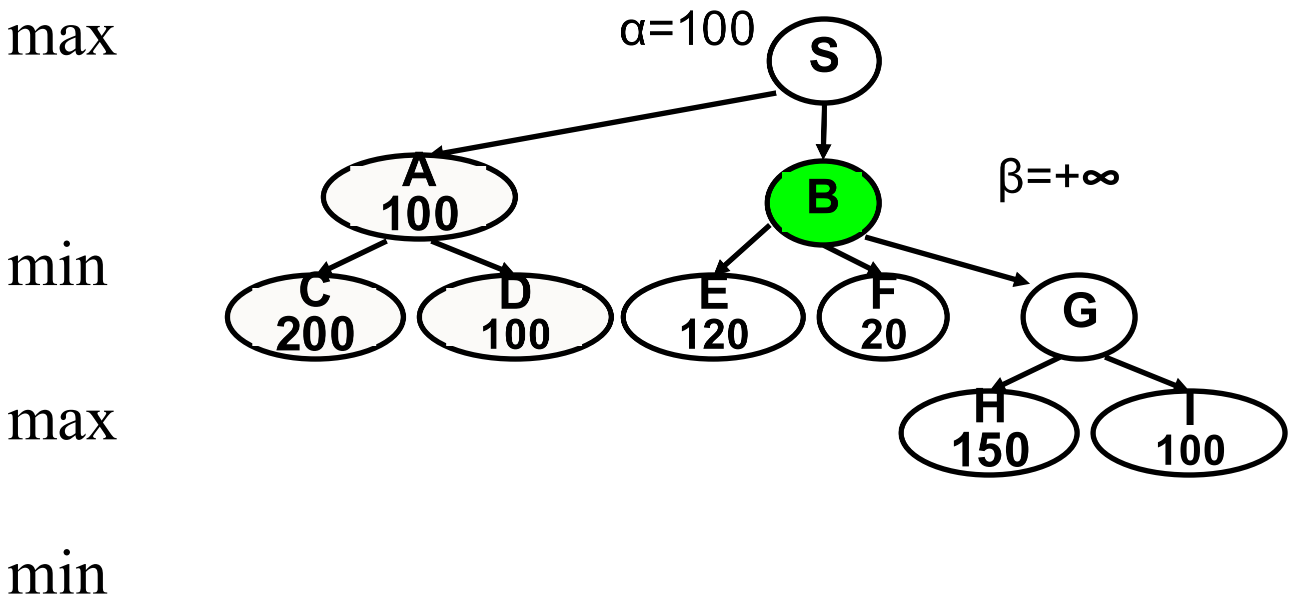
Minimax algorithm in execution



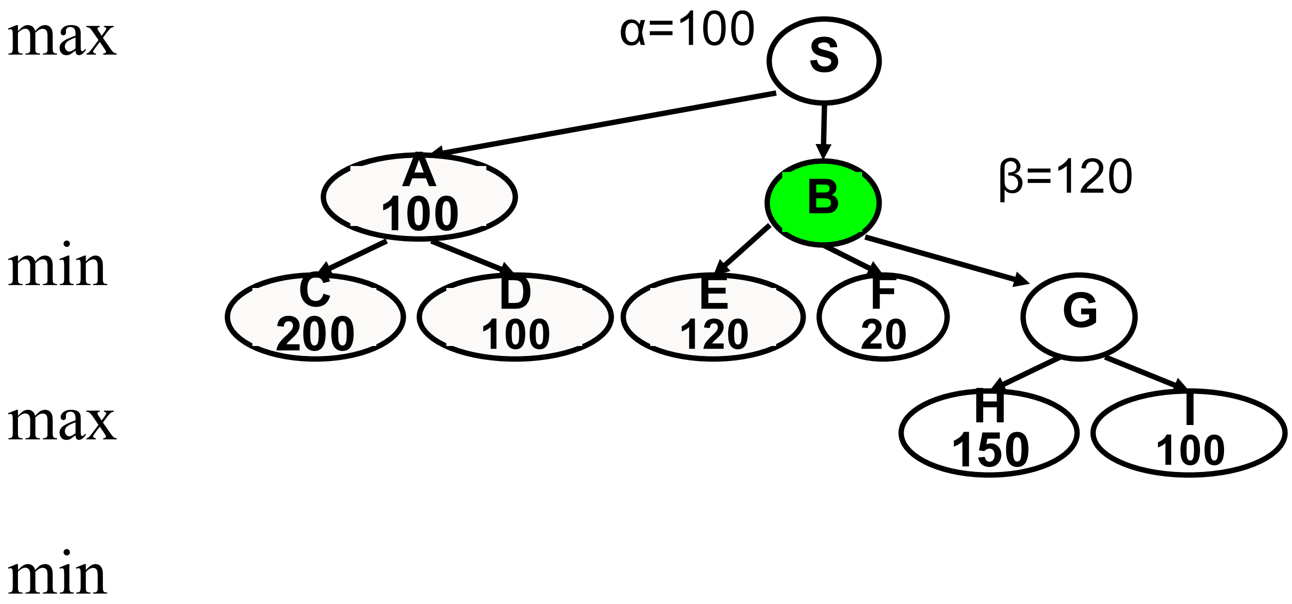
Minimax algorithm in execution



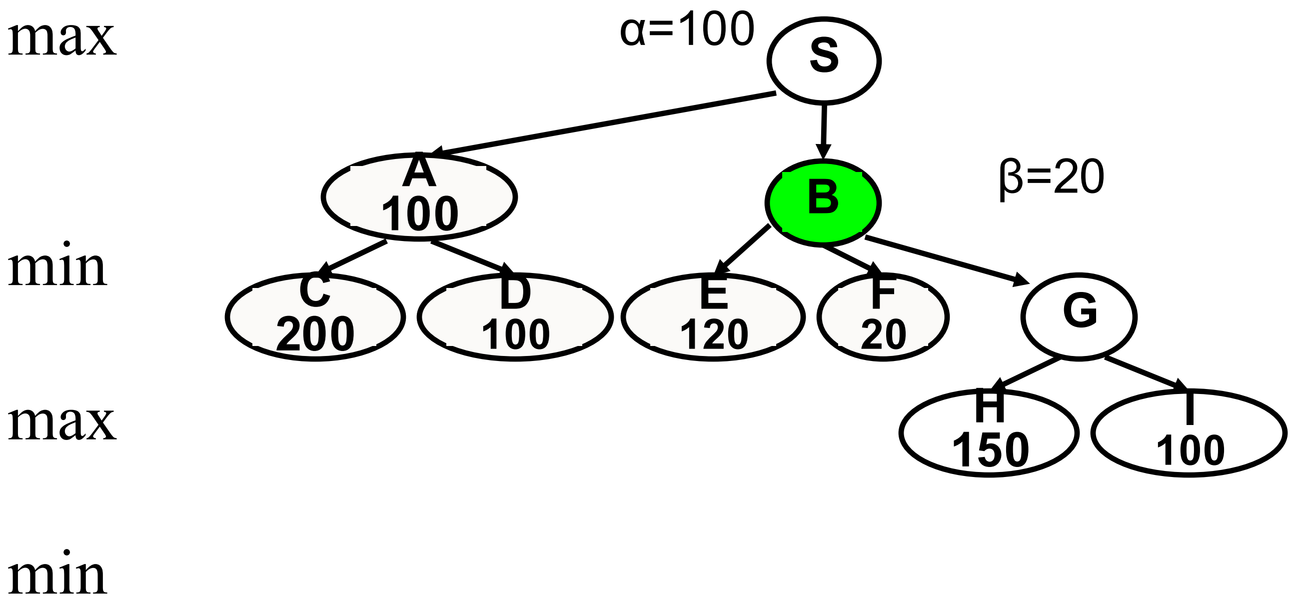
Minimax algorithm in execution



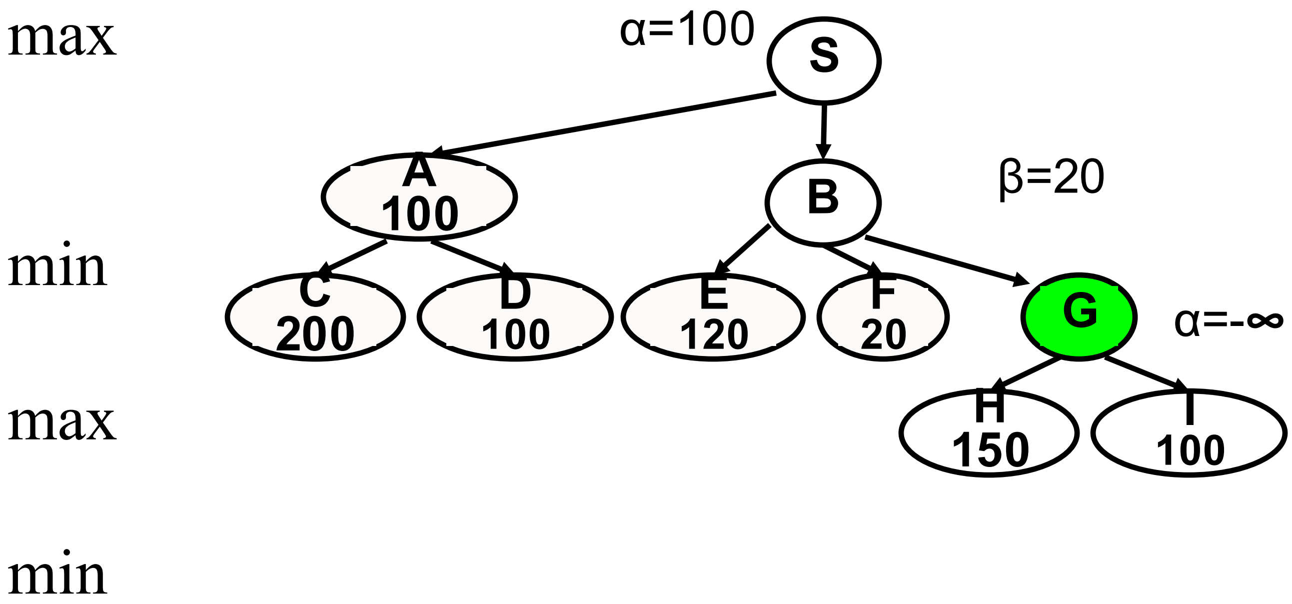
Minimax algorithm in execution



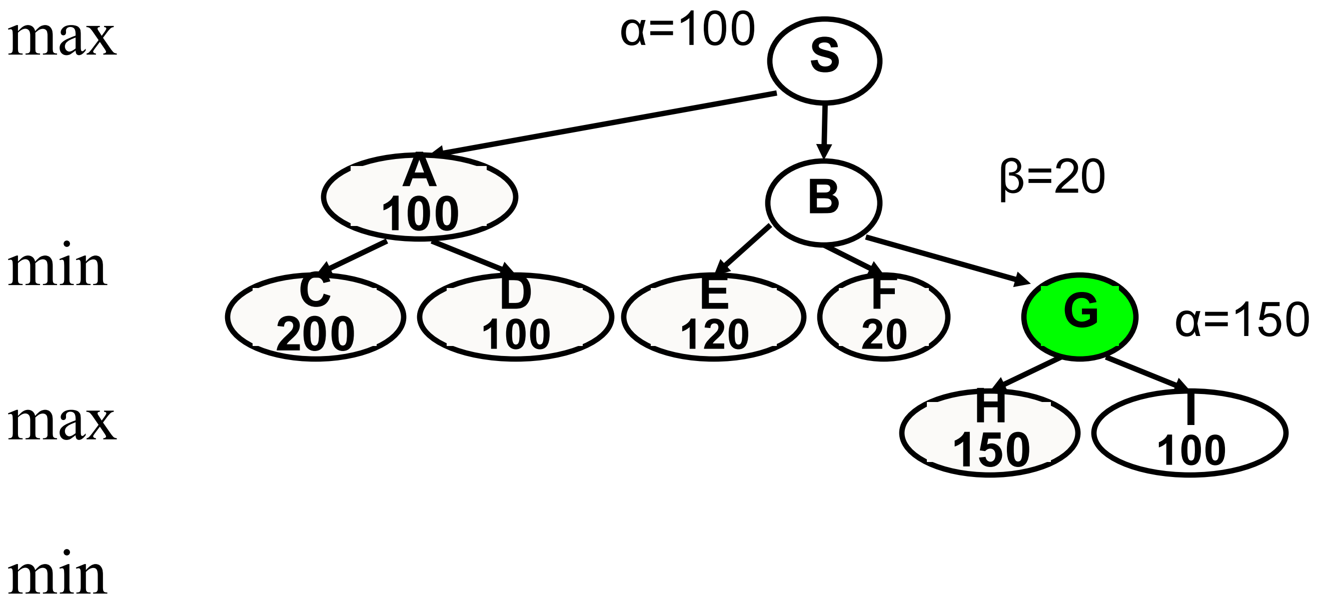
Minimax algorithm in execution



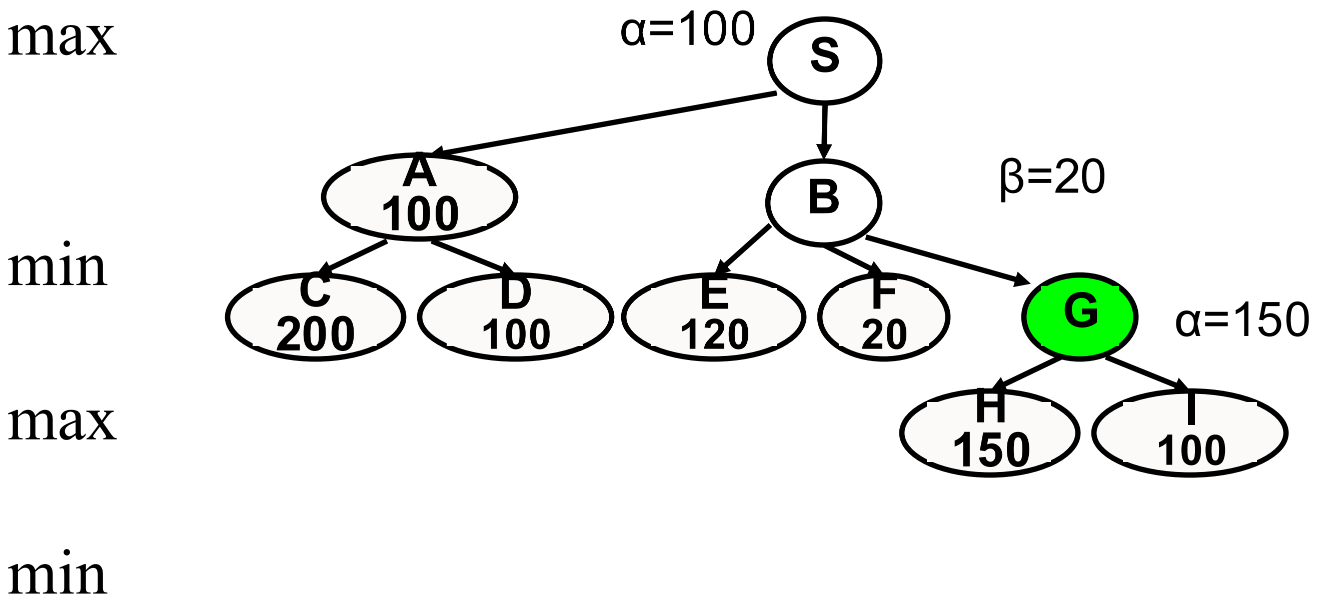
Minimax algorithm in execution



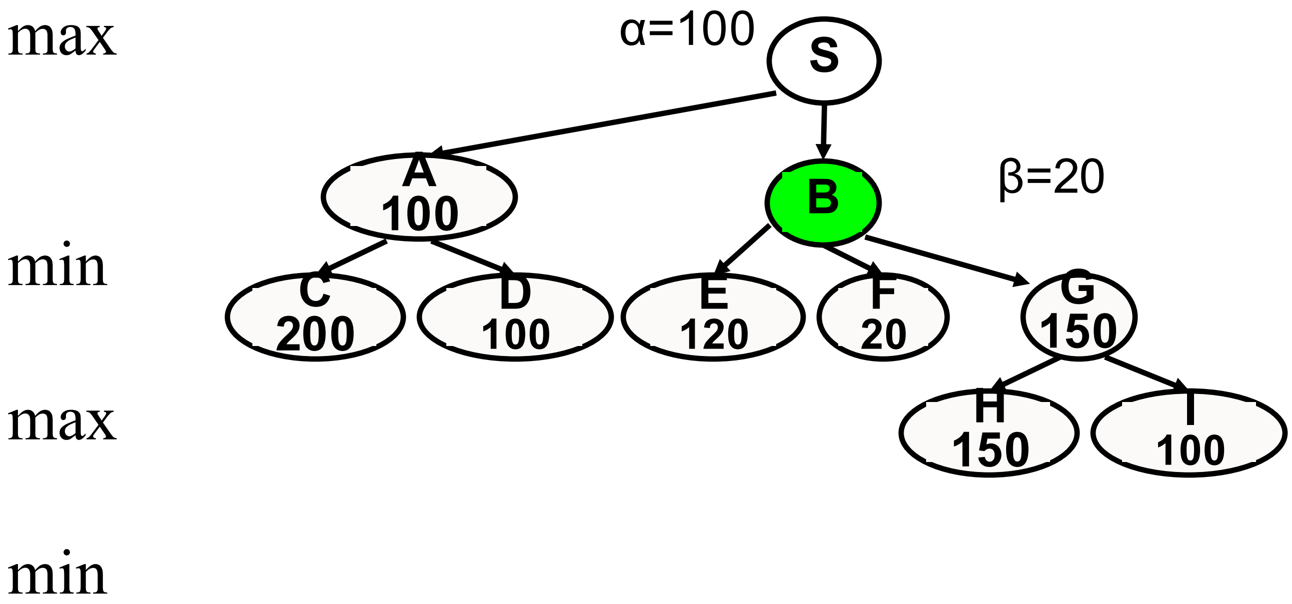
Minimax algorithm in execution



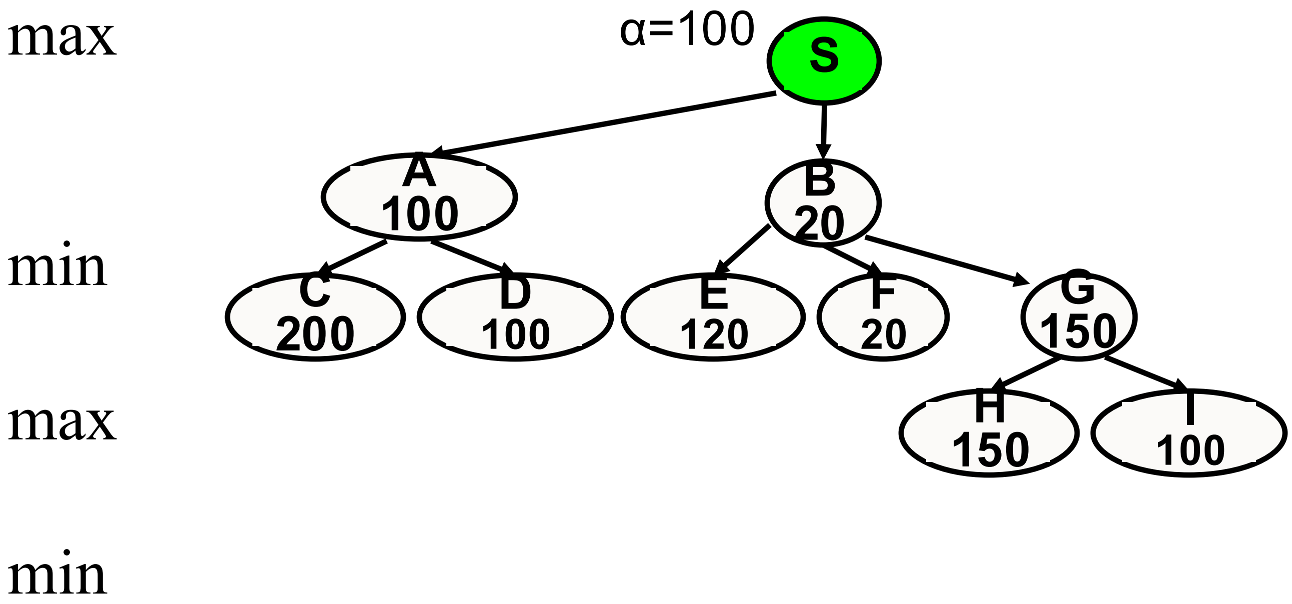
Minimax algorithm in execution



Minimax algorithm in execution



Minimax algorithm in execution



Our Approach So Far

We find the minimax value/strategy bottom up

- Minimax value: score of terminal node when both players play optimally
  - **Max's** turn, take max of children
  - **Min's** turn, take min of children

- Can implement this as depth-first search: **minimax algorithm**

Minimax Algorithm

```
function Max-Value(s)
inputs:
      s: current state in game, Max about to play
output: best-score (for Max) available from s

      if ( s is a terminal state )
      then return ( terminal value of s )
      else
            α := – infinity
            for each s’ in Succ(s)
                α := max( α , Min-value(s’))
      return α
```

```
function Min-Value(s)
output: best-score (for Min) available from s

      if ( s is a terminal state )
      then return ( terminal value of s)
      else
            β := infinity
            for each s’ in Succs(s)
                β := min( β , Max-value(s’))
      return β
```

Time complexity?

- $O(b^m)$

Space complexity?

- $O(bm)$

Game Theory Practice

17. Two firms, A and B, are deciding whether to launch a new product. Each firm can either launch or not launch. Their profits depend on their choices, and the payoff matrix is as follows:

	B: Launch	B: Not Launch
A: Launch	$(20, 20)$	$(40, 10)$
A: Not Launch	$(10, 40)$	$(30, 30)$

What is the strictly dominant strategy for each firm?

A. A's dominant strategy is to launch, and B's dominant strategy is not to launch.

B. A's dominant strategy is to launch, and B's dominant strategy is to launch.

C. A's dominant strategy is not to launch, and B's dominant strategy is to launch.

D. A's dominant strategy is not to launch, and B's dominant strategy is not to launch.

E. None of the above.

Game Theory Practice

18. Which of the following statements is true about Nash equilibrium?

    A. The mixed strategy equilibrium for the game Rock-Paper-Scissors is when both players play the mixed strategy that puts equal probabilities on all three actions.

    B. In a pure strategy Nash equilibrium, players can deviate from their strategies to increase their payoff, while in a mixed strategy Nash equilibrium, any deviation would result in a lower payoff.

    C. A mixed strategy Nash equilibrium occurs when all players choose a specific action with certainty, while a pure strategy Nash equilibrium occurs when all players choose their actions deterministically.

    D. Both pure and mixed strategy Nash equilibria involve certain level of uncertainty in the choice of strategies.

    E. None of the above.

Game Theory Practice

19. Which of the following statements is true about the minimax algorithm?

    A. The minimax algorithm can be used to find the optimal strategy for a single-

    B. The minimax algorithm finds the optimal strategy for each player in a two-player cooperative game, aiming to maximize the combined score of both players.

    C. The time complexity of a minimax algorithm is $O(bm)$.

    D. The minimax algorithm is a technique used to minimize the maximum possible loss in a two-player zero-sum game, where each player tries to maximize their score while minimizing their opponent's score.

    E. None of the above.

Game Theory Practice

20. In a two-player game, each player has three possible moves in a single round. One round is considered finished after both players have played. The game lasted for 4 rounds in total. Considering this game scenario, how many leaf nodes are there in the corresponding minimax tree?

    A. 3^4

    B. 3^8

    C. $\sum_{k=0}^{8} 3^k$

    D. $3^2 \times 4$

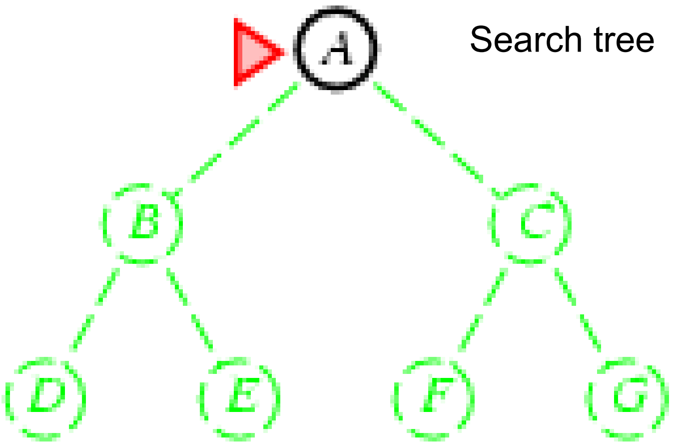
    E. None of the above.

Uninformed Search

Breadth-first search (BFS)

Use a queue (First-in First-out)

1. en_queue(Initial states)
2. While (queue not empty)
3. s = de_queue()
4. if (s==goal) success!
5. T = succs(s)
6. en_queue(T)
7. endWhile



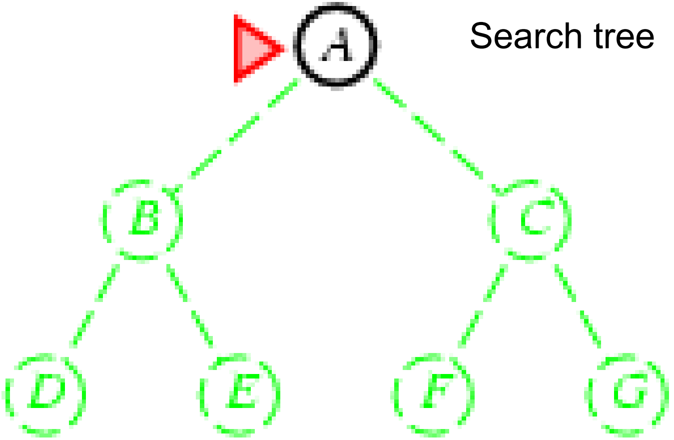
Search tree

Initial state: **A**
Goal state: **G**

Breadth-first search (BFS)

Use a queue (First-in First-out)

1. en_queue(Initial states)
2. While (queue not empty)
3. s = de_queue()
4. if (s==goal) success!
5. T = succs(s)
6. en_queue(T)
7. endWhile



Search tree

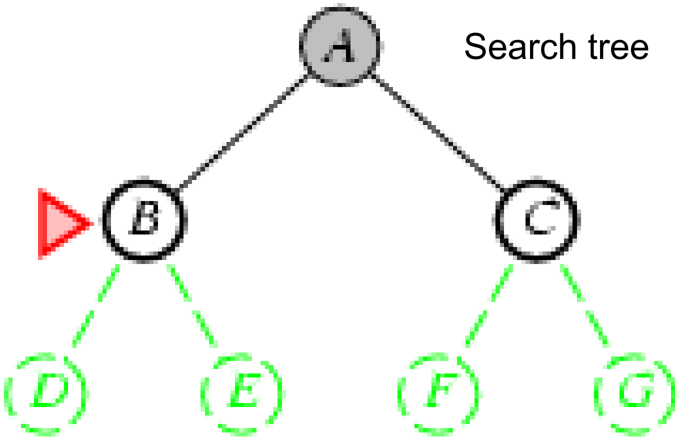
queue (fringe, OPEN)
→ [A] →

Initial state: **A**
Goal state: **G**

Breadth-first search (BFS)

Use a queue (First-in First-out)

1. en_queue(Initial states)
2. While (queue not empty)
3.   s = de_queue()
4.   if (s==goal) success!
5.   T = succs(s)
6.   en_queue(T)
7. endWhile



Search tree

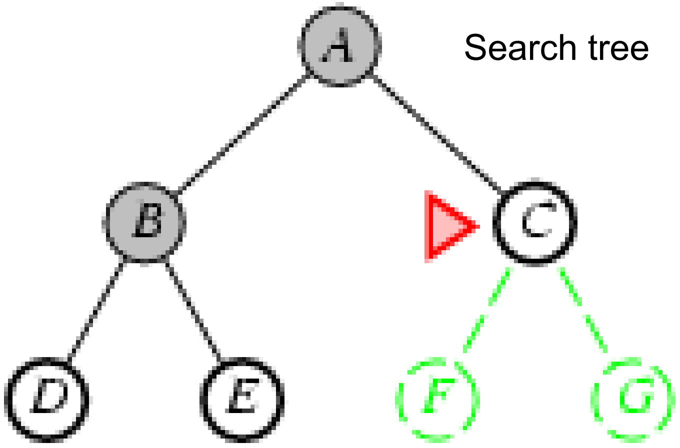
queue (fringe, OPEN)
→ [CB] → A

Initial state: **A**
Goal state: **G**

Breadth-first search (BFS)

Use a queue (First-in First-out)

1. en_queue(Initial states)
2. While (queue not empty)
3. s = de_queue()
4. if (s==goal) success!
5. T = succs(s)
6. en_queue(T)
7. endWhile



Search tree

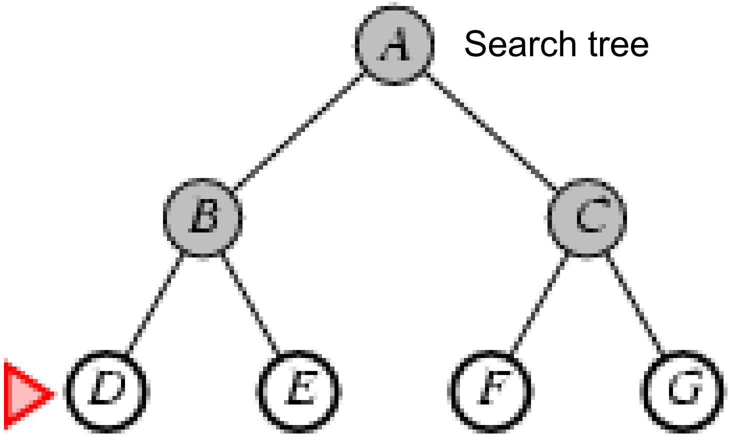
queue (fringe, OPEN)
→ [EDC] → B

Initial state: **A**
Goal state: **G**

Breadth-first search (BFS)

Use a queue (First-in First-out)

1. en_queue(Initial states)
2. While (queue not empty)
3. s = de_queue()
4. if (s==goal) success!
5. T = succs(s)
6. en_queue(T)
7. endWhile



queue (fringe, OPEN)
□[GFED] → C

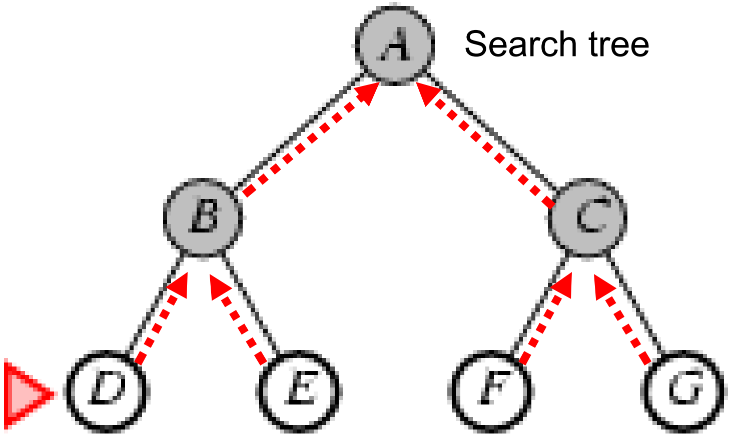
If G is a goal, we've seen it, but we don't stop!

Initial state: **A**
Goal state: **G**

Breadth-first search (BFS)

Use a queue (First-in First-out)

1. en_queue(Initial states)
2. While (queue not empty)
3. s = de_queue()
4. if (s==goal) success!
5. T = succs(s)
6. en_queue(T)
7. endWhile



queue
□[] →G

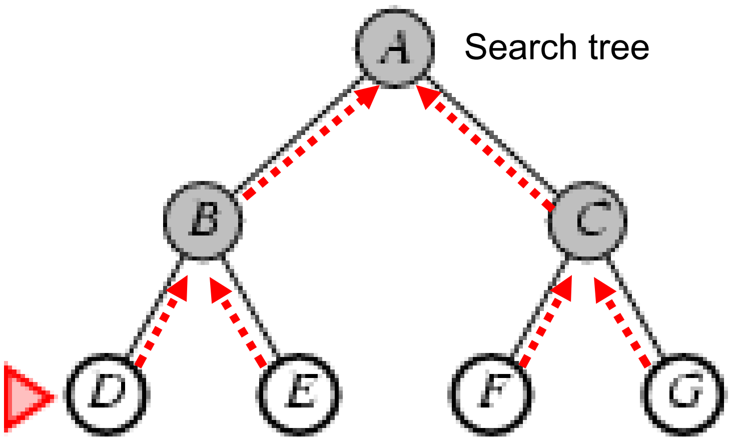
... until much later we pop G.

Looking foolish? Indeed. But let's be consistent…

Breadth-first search (BFS)

Use a queue (First-in First-out)

1. en_queue(Initial states)
2. While (queue not empty)
3. s = de_queue()
4. if (s==goal) success!
5. T = succs(s)
6. en_queue(T)
7. endWhile



queue
□[] →G

... until much later we pop G.

We need back pointers to recover the solution path.

Looking foolish? Indeed. But let's be consistent…

Performance of search algorithms on trees

b: branching factor (assume finite)

d: goal depth

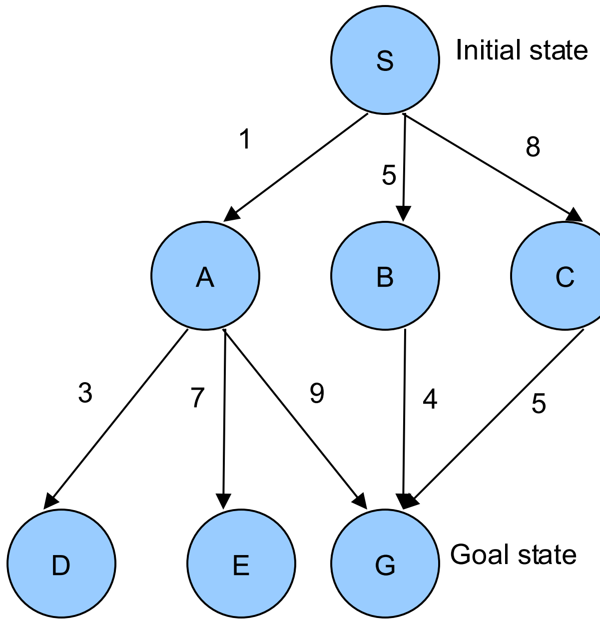
	Complete	optimal	time	space
Breadth-first search	Y	Y, if [1]	$O(b^d)$	$O(b^d)$

1. Edge cost constant, or positive non-decreasing in depth

Uniform-cost search

- Find the least-cost goal
- Each node has a path cost from start (= sum of edge costs along the path).
- Expand the least cost node first.
- Use a priority queue instead of a normal queue
  - Always take out the least cost item

Example



1: (S,0), [(A,1), (B,5), (C,8)]
2: (A,1), [(B,5), (C,8), (D,4), (E,8), (G,10)]
3: (D,4), [(B,5), (C,8), (E,8), (G,10)]
4: (B,5), [(C,8), (E,8), (G,9)]
5: (C,8), [(E,8), (G,9)]
6: (E,8), [(G,9)]
7: (G,9), []: Success!

(All edges are directed, pointing downwards)

Performance of search algorithms on trees

b: branching factor (assume finite)   d: goal depth

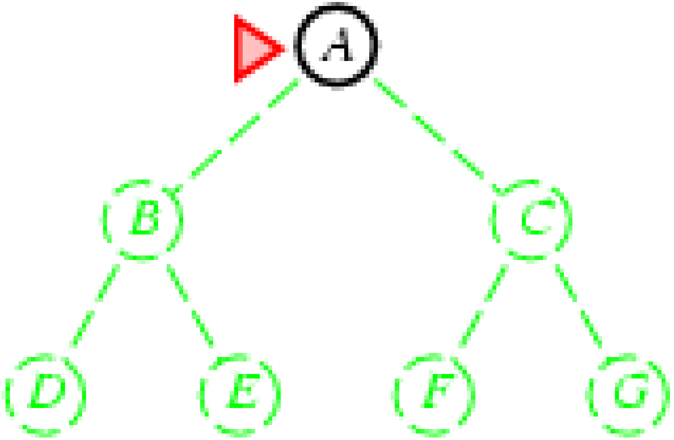
	Complete	optimal	time	space
Breadth-first search	Y	Y, if [1]	$O(b^d)$	$O(b^d)$
Uniform-cost search[2]	Y	Y	$O(b^{C^*/\varepsilon})$	$O(b^{C^*/\varepsilon})$

1. edge cost constant, or positive non-decreasing in depth
2. edge costs $\geq \varepsilon > 0$. C^* is the best goal path cost.

Depth-first search (DFS)

Use a stack (First-in Last-out)

1. push(Initial states)
2. While (stack not empty)
3. s = pop()
4. if (s==goal) success!
5. T = succs(s)
6. push(T)
7. endWhile



stack (fringe)

1. A, [B, C]
2. B, [D, E, C]
3. D, [E, C]
4. E, [C]
5. C, [F, G]
6. F, [G]
7. G

Performance of search algorithms on trees

b: branching factor (assume finite) d: goal depth m: graph depth

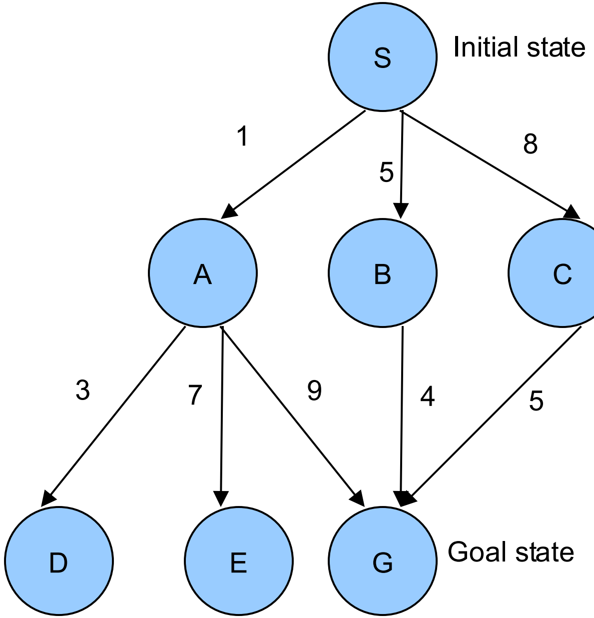
	Complete	optimal	time	space
Breadth-first search	Y	Y, if [1]	$O(b^d)$	$O(b^d)$
Uniform-cost search[2]	Y	Y	$O(b^{C^*/\varepsilon})$	$O(b^{C^*/\varepsilon})$
Depth-first search	N	N	$O(b^m)$	$O(bm)$

1. edge cost constant, or positive non-decreasing in depth
2. edge costs $\geq \varepsilon > 0$. C^* is the best goal path cost.

Iterative deepening

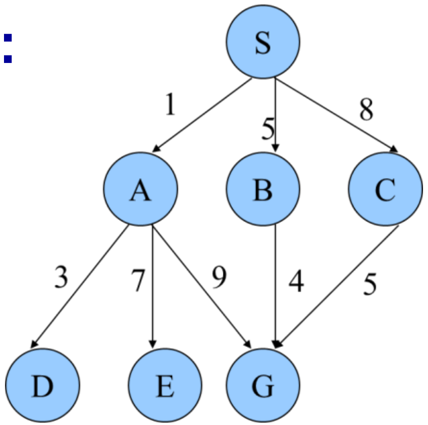
- Search proceeds like BFS, but fringe is like DFS
  - Complete, optimal like BFS
  - Small space complexity like DFS
  - Time complexity like BFS
- Preferred uninformed search method

Example



(All edges are directed, pointing downwards)

Nodes expanded by:



- Breadth-First Search: S A B C D E G

  Solution found: S A G

- Uniform-Cost Search: S A D B C E G

  Solution found: S B G (This is the only uninformed search that worries about costs.)

- Depth-First Search: S A D E G

  Solution found: S A G

- Iterative-Deepening Search: S A B C S A D E G

  Solution found: S A G

Performance of search algorithms on trees

b: branching factor (assume finite) d: goal depth m: graph depth

	Complete	optimal	time	space
Breadth-first search	Y	Y, if [1]	$O(b^d)$	$O(b^d)$
Uniform-cost search[2]	Y	Y	$O(b^{C^*/\varepsilon})$	$O(b^{C^*/\varepsilon})$
Depth-first search	N	N	$O(b^m)$	$O(bm)$
Iterative deepening	Y	Y, if [1]	$O(b^d)$	$O(bd)$

1. edge cost constant, or positive non-decreasing in depth
2. edge costs $\geq \varepsilon > 0$. C^* is the best goal path cost.

Informed Search

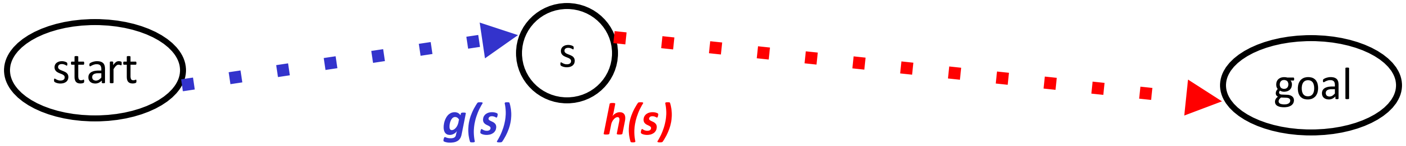
Uninformed vs Informed Search

Uninformed search (all of what we saw). Know:

- Path cost ***g***(*s*) from start to node *s*
- Successors.

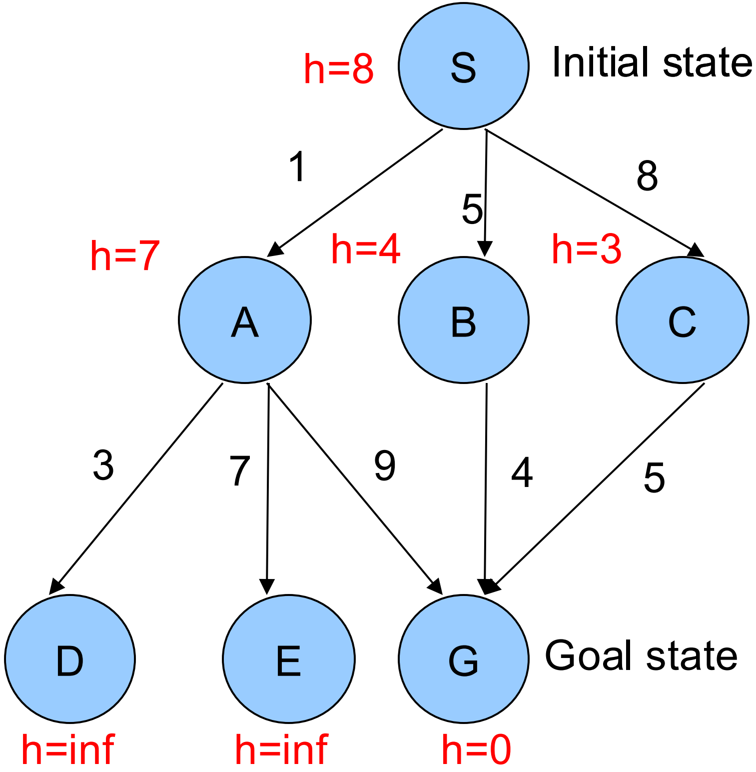
Informed search. Know:

- All uninformed search properties, plus
- Heuristic **h(s)** from s to goal (recall game heuristic)



Recap and Examples

Example for A*:

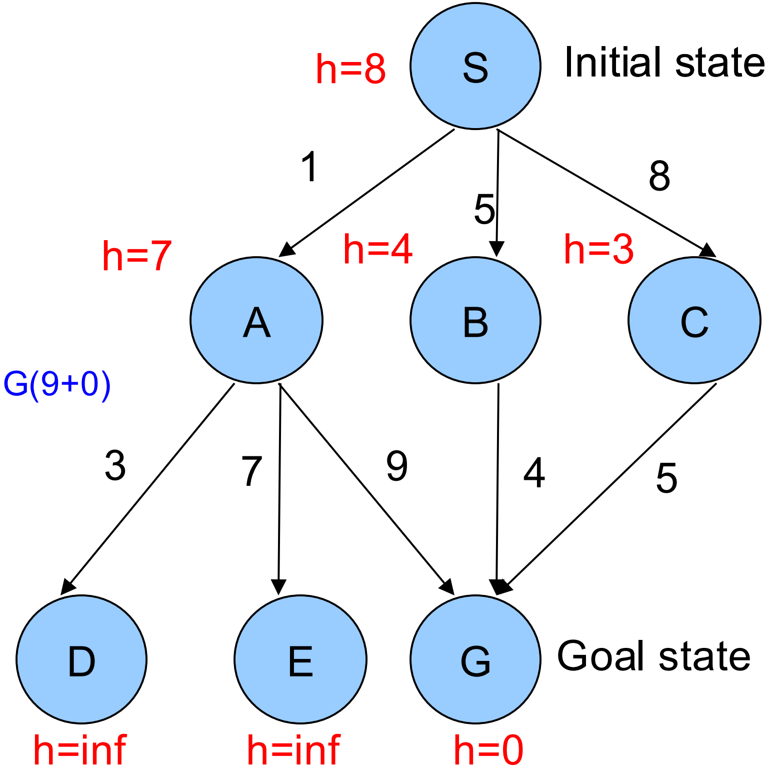


Recap and Examples

Example for A*:

OPEN	CLOSED
S(0+8)	-
A(1+7) B(5+4) C(8+3)	S(0+8)
B(5+4) C(8+3) D(4+inf) E(8+inf) G(10+0)	S(0+8) A(1+7)
C(8+3) D(4+inf) E(8+inf) G(9+0)	S(0+8) A(1+7) B(5+4)
C(8+3) D(4+inf) E(8+inf)	S(0+8) A(1+7) B(5+4) G(9+0)

$G \rightarrow B \rightarrow S$



Neural Networks

How to classify

Cats vs. dogs?

Single-layer Perceptron

↓

Multi-layer Perceptron

↓

Training of neural networks

↓

Convolutional neural networks

Neural networks can also be used for regression.

- Typically, no activation on outputs, mean squared error loss function.

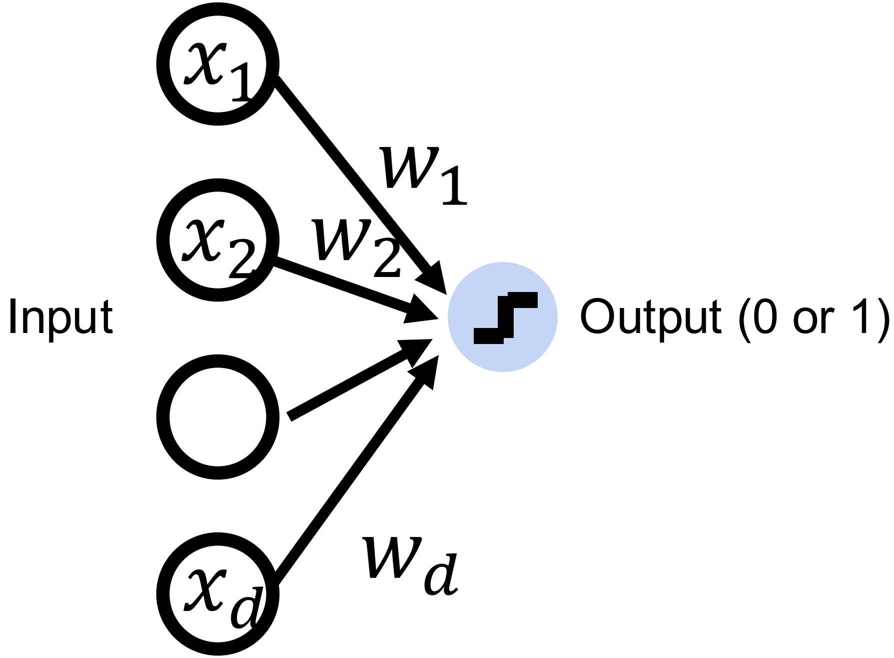
Perceptron

- Given input $\mathbf{x}$, weight $\mathbf{w}$ and bias b, perceptron outputs:

$$o = \sigma(\mathbf{w}^\top \mathbf{x} + b) \qquad \sigma(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

Activation function

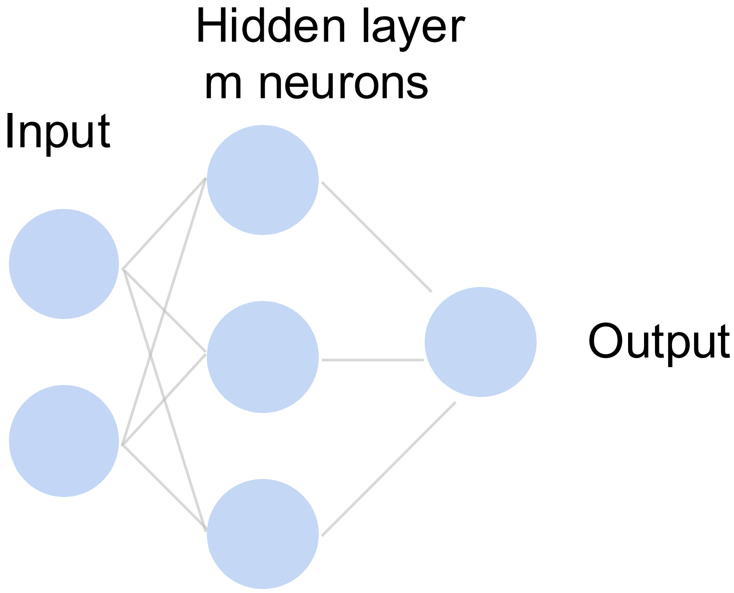
Cats vs. dogs?



Single Hidden Layer

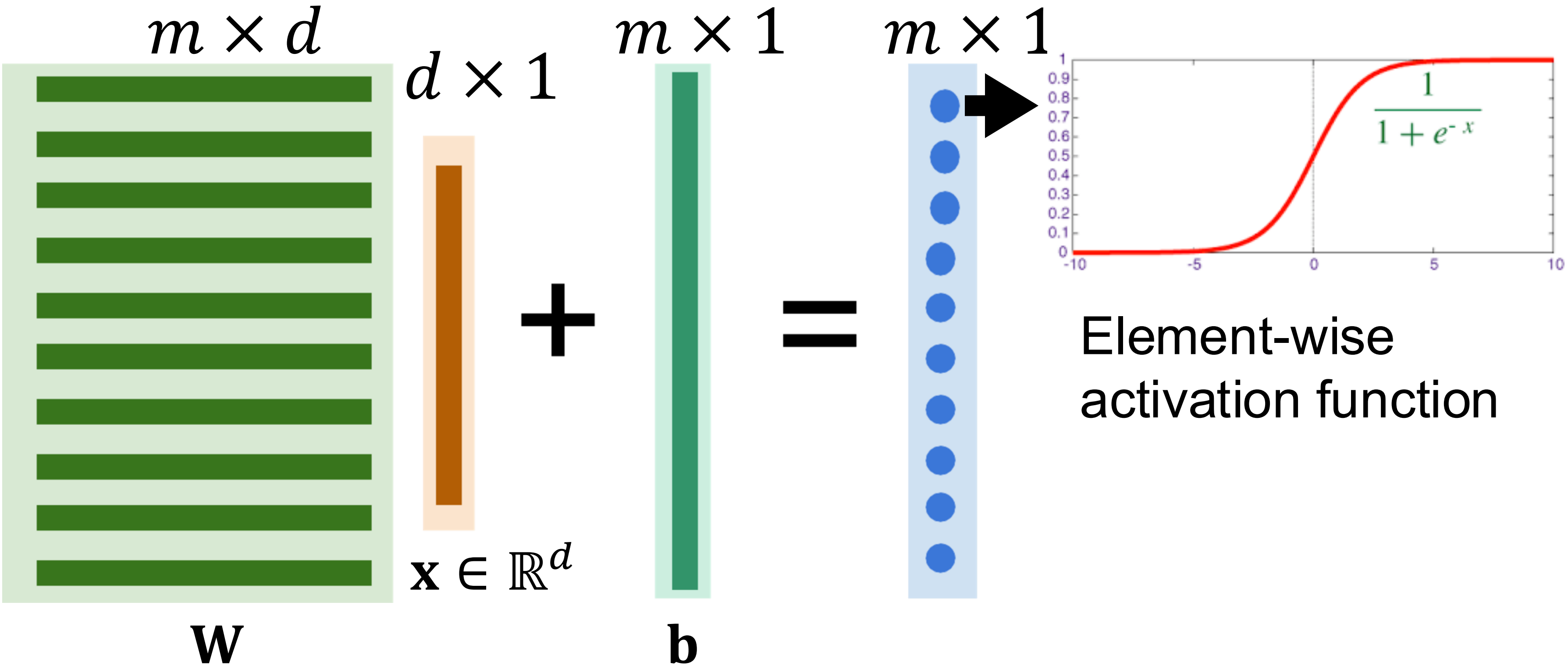
How to classify

Cats vs. dogs?



Neural networks with one hidden layer

Key elements: linear operations + Nonlinear activations

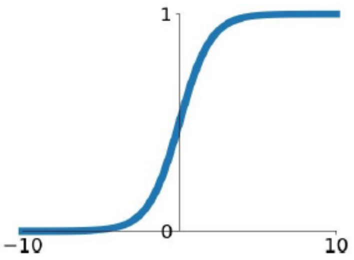


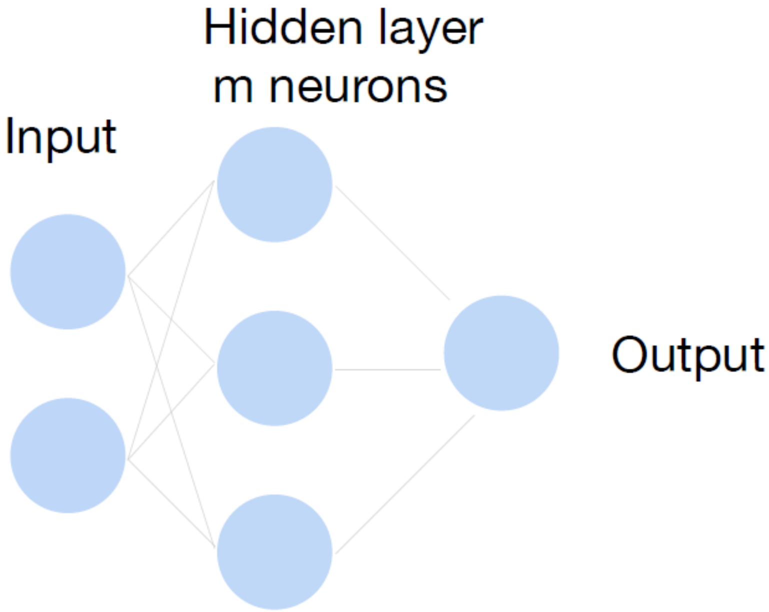
Single Hidden Layer

- Output $f = \mathbf{w}_2^\top \mathbf{h} + b_2$
- Normalize the output into probability using sigmoid

$$p(y = 1 \mid \mathbf{x}) = \frac{1}{1 + e^{-f}}$$

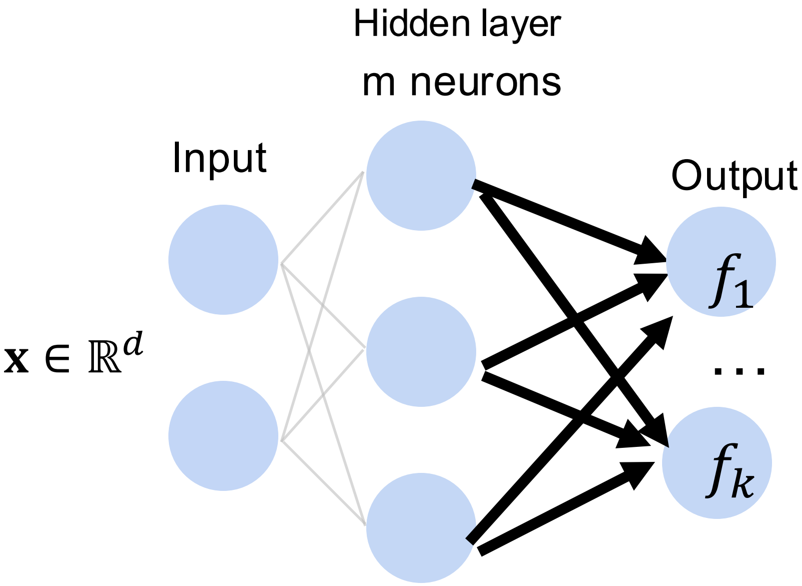
Sigmoid





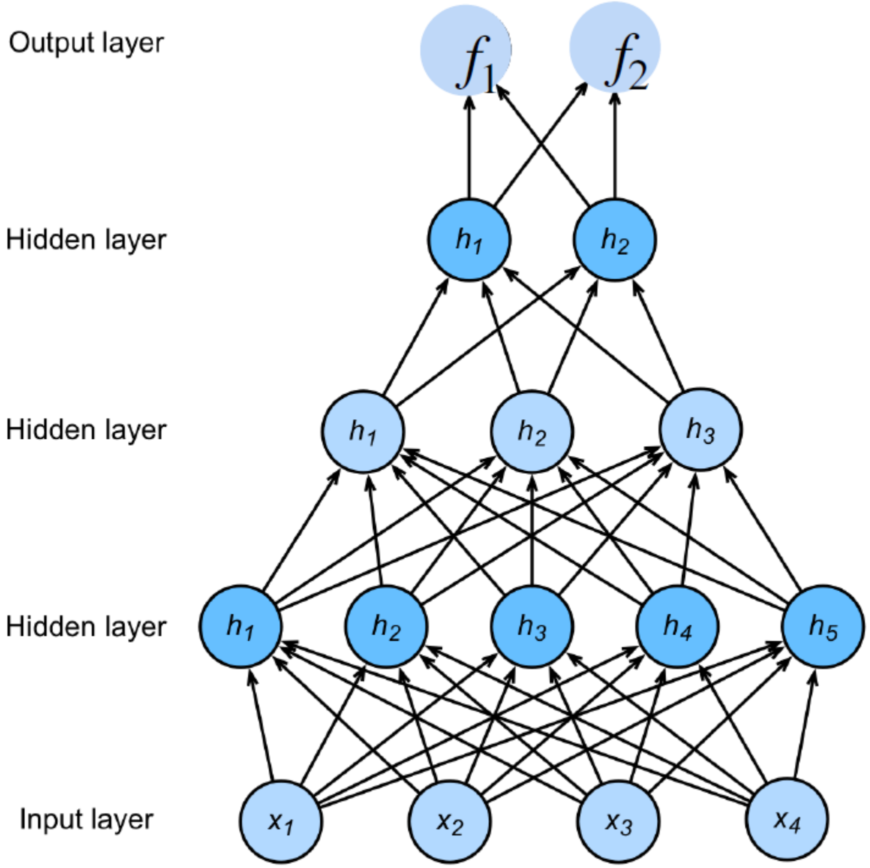
Multi-class classification

Turns outputs *f* into *k* probabilities (sum up to 1 across *k* classes)



$$p(y|\mathbf{x}) = softmax(\mathbf{f})$$
$$= \frac{\exp f_y(x)}{\sum_i^k \exp f_i(x)}$$

Deep neural networks (DNNs)



$$\mathbf{h}_1 = \sigma(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$$
$$\mathbf{h}_2 = \sigma(\mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2)$$
$$\mathbf{h}_3 = \sigma(\mathbf{W}_3\mathbf{h}_2 + \mathbf{b}_3)$$
$$\mathbf{f} = \mathbf{W}_4\mathbf{h}_3 + \mathbf{b}_4$$
$$\mathbf{y} = \text{softmax}(\mathbf{f})$$

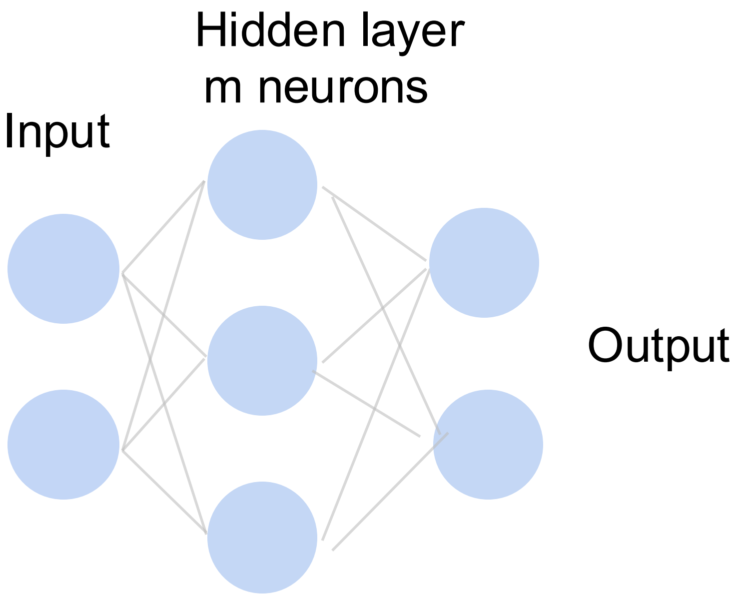
NNs are composition of nonlinear functions

How to train a neural network?

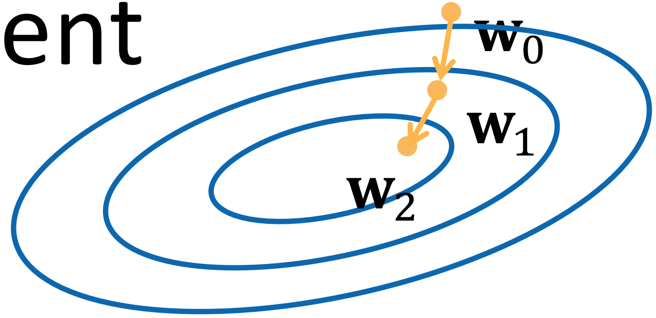
Update the weights W to minimize the loss function

$$L = \frac{1}{|D|} \sum_i \ell(\mathbf{x}_i, y_i)$$

Use gradient descent!



Gradient Descent



- Choose a learning rate $\alpha > 0$
- Initialize the model parameters w_0
- For t =1, 2, …
  - Update parameters:

$$\mathbf{w}_t = \mathbf{w}_{t-1} - \alpha \frac{\partial L}{\partial \mathbf{w}_{t-1}}$$

D can be very large. Expensive per iteration

$$= \mathbf{w}_{t-1} - \alpha \frac{1}{|D|} \sum_{\mathbf{x} \in D} \frac{\partial \ell(\mathbf{x}_i, y_i)}{\partial \mathbf{w}_{t-1}}$$

- Repeat until converges

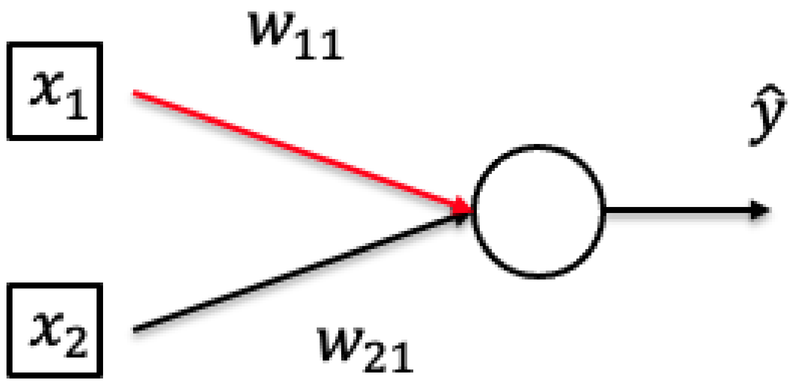
Minibatch Stochastic Gradient Descent

- Choose a learning rate $\alpha > 0$
- Initialize the model parameters w_0
- For *t* =1, 2, …
  - **Randomly sample a subset (mini-batch)** $B \subset D$
    Update parameters:

$$\mathbf{w}_t = \mathbf{w}_{t-1} - \alpha \frac{1}{|B|} \sum_{\mathbf{x} \in B} \frac{\partial \ell(\mathbf{x}_i, y_i)}{\partial \mathbf{w}_{t-1}}$$

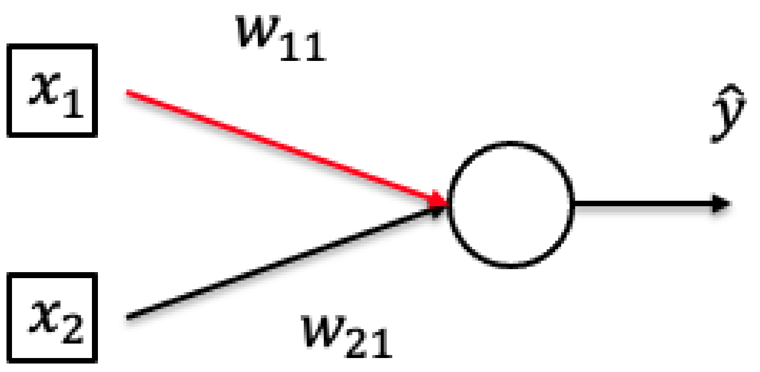
- Repeat

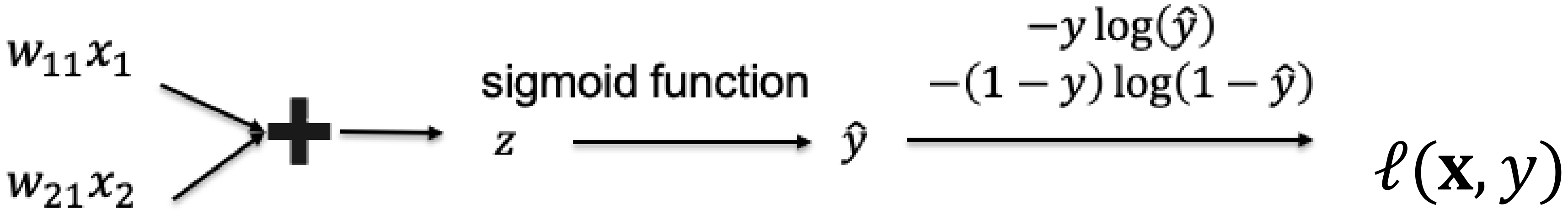
Calculate Gradient (on one data point)



- Want to compute $\frac{\partial \ell(\mathbf{x},y)}{\partial w_{11}}$
- Data point: $((x_1, x_2), y)$

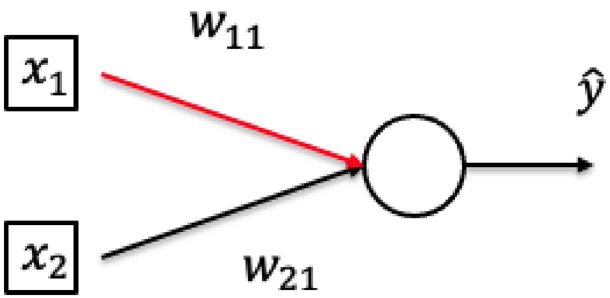
Calculate Gradient (on one data point)

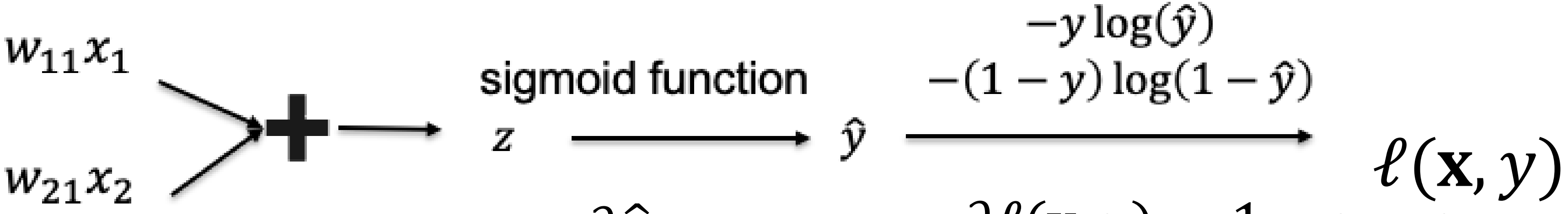




Use chain rule!

Calculate Gradient (on one data point)

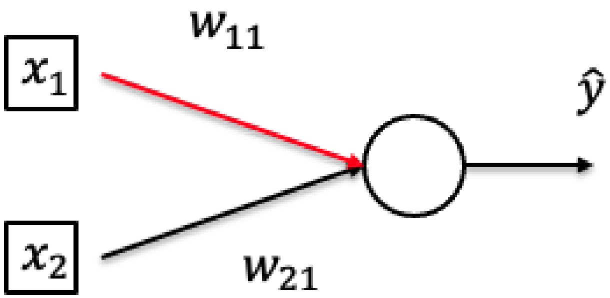




$$\frac{\partial \hat{y}}{\partial z} = \sigma'(z) \qquad \frac{\partial \ell(\mathbf{x}, y)}{\partial \hat{y}} = \frac{1-y}{1-\hat{y}} - \frac{y}{\hat{y}}$$

- By chain rule: $\dfrac{\partial l}{\partial w_{11}} = \dfrac{\partial l}{\partial \hat{y}} \dfrac{\partial \hat{y}}{\partial z} \dfrac{\partial z}{\partial w_{11}}$

Calculate Gradient (on one data point)

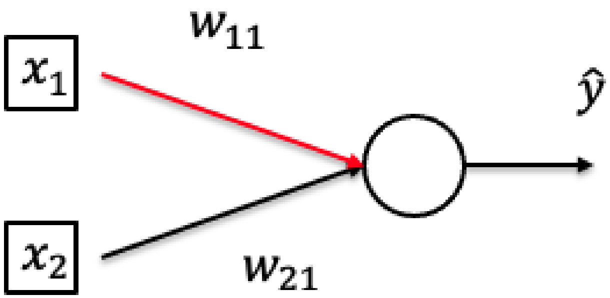


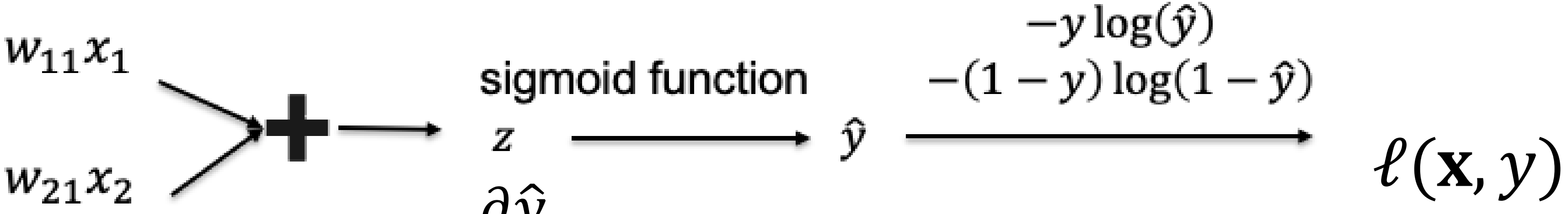
$$w_{11}x_1,\ w_{21}x_2 \longrightarrow + \longrightarrow z \xrightarrow{\text{sigmoid function}} \hat{y} \xrightarrow{-y\log(\hat{y})-(1-y)\log(1-\hat{y})} \ell(\mathbf{x}, y)$$

$$\frac{\partial \hat{y}}{\partial z} = \sigma'(z) \qquad \frac{\partial \ell(\mathbf{x}, y)}{\partial \hat{y}} = \frac{1-y}{1-\hat{y}} - \frac{y}{\hat{y}}$$

- By chain rule: $\dfrac{\partial l}{\partial w_{11}} = \dfrac{\partial l}{\partial \hat{y}} \dfrac{\partial \hat{y}}{\partial z}\ x_1$

Calculate Gradient (on one data point)

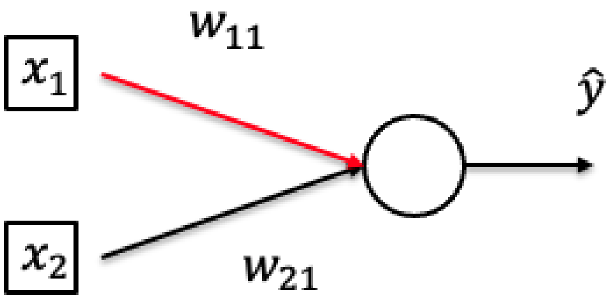


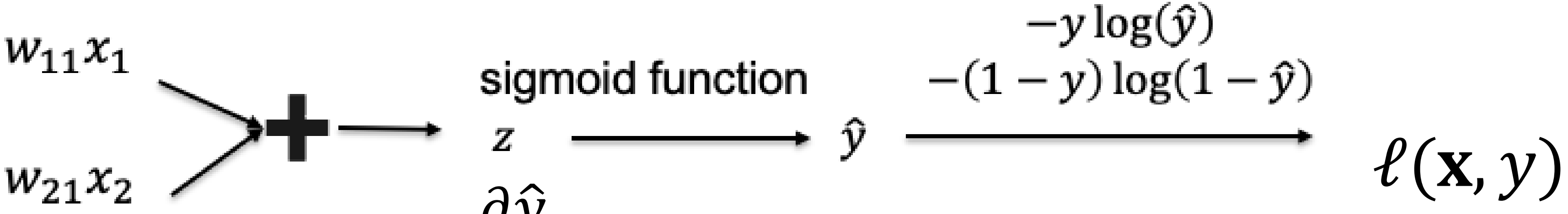


$$\frac{\partial \hat{y}}{\partial z} = \sigma'(z) = \sigma(z)(1-\sigma(z))$$

- By chain rule: $\dfrac{\partial l}{\partial w_{11}} = \dfrac{\partial l}{\partial \hat{y}} \;\; \hat{y}(1-\hat{y})x_1$

Calculate Gradient (on one data point)

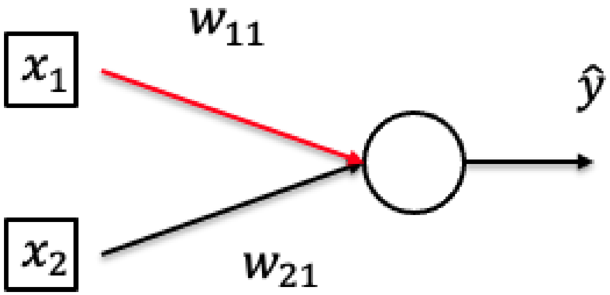


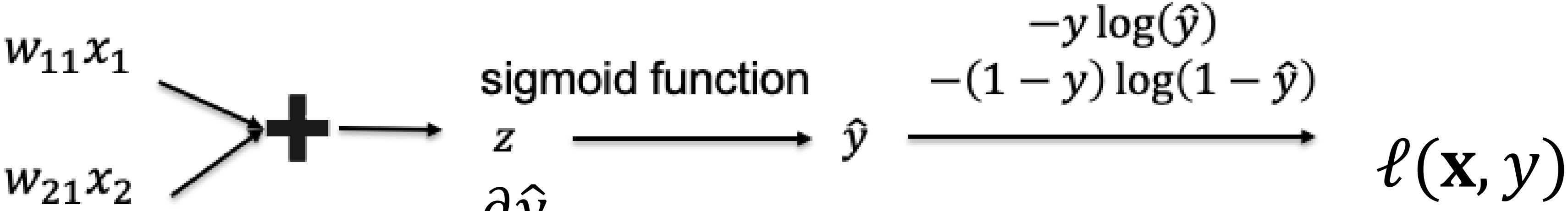


$$\frac{\partial \hat{y}}{\partial z} = \sigma'(z) = \sigma(z)(1 - \sigma(z))$$

- By chain rule: $\dfrac{\partial l}{\partial w_{11}} = \left(\dfrac{1-y}{1-\hat{y}} - \dfrac{y}{\hat{y}}\right)\hat{y}(1-\hat{y})x_1$

Calculate Gradient (on one data point)

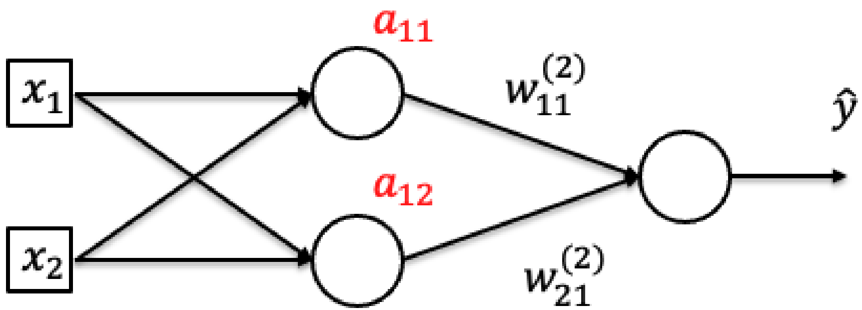




$$\frac{\partial \hat{y}}{\partial z} = \sigma'(z) = \sigma(z)(1-\sigma(z))$$

- By chain rule: $\dfrac{\partial l}{\partial w_{11}} = (\hat{y} - y)x_1$

Calculate Gradient (on one data point)



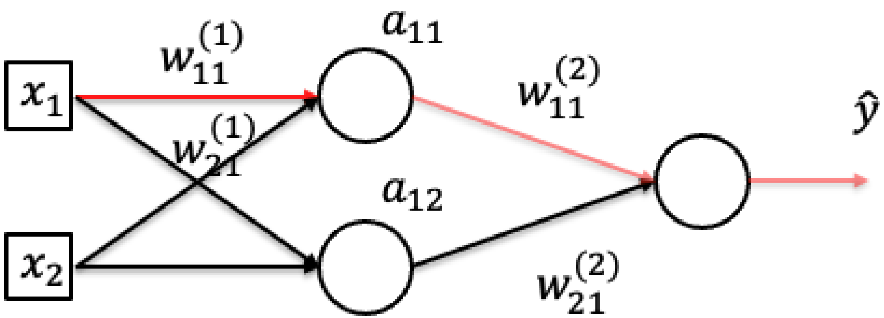
Make it deeper

$$w_{11}^{(2)} a_{11} + w_{21}^{(2)} a_{12} \longrightarrow z \xrightarrow{\text{sigmoid function}} \hat{y} \xrightarrow{-y\log(\hat{y}) - (1-y)\log(1-\hat{y})} \ell(\mathbf{x}, y)$$

$$\frac{\partial \hat{y}}{\partial z} = \sigma'(z) = \sigma(z)(1-\sigma(z))$$

- By chain rule $\frac{\partial l}{\partial a_{11}} = (\hat{y} - y) w_{11}^{(2)}, \; \frac{\partial l}{\partial a_{12}} = (\hat{y} - y) w_{21}^{(2)}$

Calculate Gradient (on one data point)

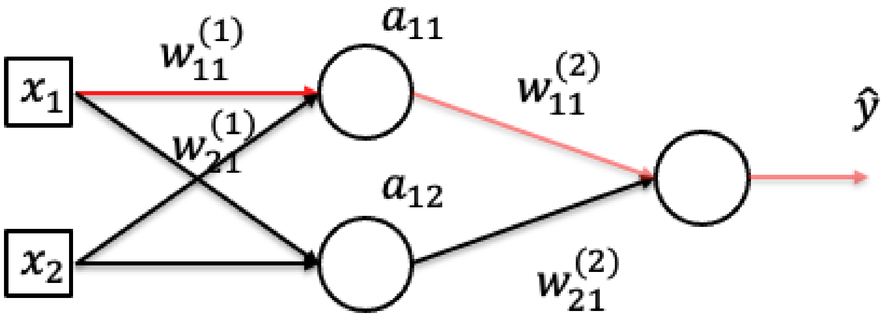


$$\left.\begin{matrix} w_{11}^{(1)} x_1 \\ w_{21}^{(1)} x_2 \end{matrix}\right\} \to + \longrightarrow z_{11} \xrightarrow{\sigma(z_{11})} a_{11} \longrightarrow l(\boldsymbol{x}, y)$$

$$\frac{\partial a_{11}}{\partial z_{11}} = \sigma'(z_{11}) \qquad \frac{\partial l}{\partial a_{11}} = (\hat{y} - y) w_{11}^{(2)}$$

- By chain rule: $\dfrac{\partial l}{\partial w_{11}^{(1)}} = \dfrac{\partial l}{\partial a_{11}} \dfrac{\partial a_{11}}{\partial w_{11}^{(1)}} = (\hat{y} - y) w_{11}^{(2)} \dfrac{\partial a_{11}}{\partial w_{11}^{(1)}}$

Calculate Gradient (on one data point)



$$w_{11}^{(1)}x_1 + w_{21}^{(1)}x_2 \longrightarrow z_{11} \xrightarrow{\sigma(z_{11})} a_{11} \longrightarrow l(\boldsymbol{x}, y)$$

$$\frac{\partial a_{11}}{\partial z_{11}} = \sigma'(z_{11}) \qquad \frac{\partial l}{\partial a_{11}} = (\hat{y} - y)w_{11}^{(2)}$$

- By chain rule: $\dfrac{\partial l}{\partial w_{11}^{(1)}} = \dfrac{\partial l}{\partial a_{11}} \dfrac{\partial a_{11}}{\partial w_{11}^{(1)}} = (\hat{y} - y)w_{11}^{(2)} a_{11}(1 - a_{11})x_1$

Numerical Stability

Gradients for Neural Networks

- Compute the gradient of the loss ℓ $\mathbf{w}_t$ w.

$$\frac{\partial \ell}{\partial \mathbf{W}^t} = \frac{\partial \ell}{\partial \mathbf{h}^d} \underbrace{\frac{\partial \mathbf{h}^d}{\partial \mathbf{h}^{d-1}} \cdots \frac{\partial \mathbf{h}^{t+1}}{\partial \mathbf{h}^t}}_{\text{Multiplication of } \textit{many} \text{ matrices}} \frac{\partial \mathbf{h}^t}{\partial \mathbf{W}^t}$$

Wikipedia

Two Issues for Deep Neural Networks

$$\prod_{i=t}^{d-1} \frac{\partial \mathbf{h}^{i+1}}{\partial \mathbf{h}^{i}}$$

Gradient Exploding

$1.5^{100} \approx 4 \times 10^{17}$

Gradient Vanishing

$0.8^{100} \approx 2 \times 10^{-10}$

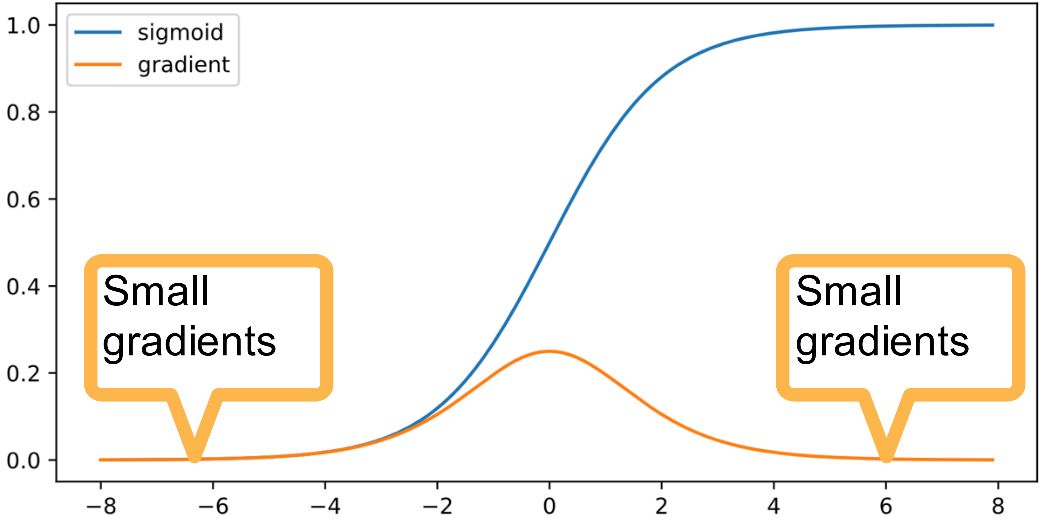
Issues with Gradient Exploding

- Value out of range: infinity value (NaN)
- Sensitive to learning rate (LR)
  - Not small enough LR → larger gradients
  - Too small LR → No progress
  - May need to change LR dramatically during training

Gradient Vanishing

- Use sigmoid as the activation function

$$\sigma(x) = \frac{1}{1+e^{-x}} \quad \sigma'(x) = \sigma(x)(1-\sigma(x))$$



Issues with Gradient Vanishing

- Gradients with value 0
- No progress in training
    - No matter how to choose learning rate
- Severe with bottom layers (those near the input)
    - Only top layers (near output) are well trained
    - No benefit to make networks deeper

How to stabilize training?

Stabilize Training: Practical Considerations

- Goal: make sure gradient values are in a proper range
  - E.g. in [1e-6, 1e3]
- Multiplication → plus
  - Architecture change (e.g., ResNet)
- Normalize
  - Batch Normalization, Gradient clipping
- Proper activation functions

Quiz. Which of the following are TRUE about the vanishing gradient problem in neural networks? Multiple answers are possible.

A. Deeper neural networks tend to be more susceptible to vanishing gradients.

B. Using the ReLU function can reduce this problem.

C. If a network has the vanishing gradient problem for one training point due to the sigmoid function, it will also have a vanishing gradient for every other training point.

D. Networks with sigmoid functions don’t suffer from the vanishing gradient problem if trained with the cross-entropy loss.

Quiz. Which of the following are TRUE about the vanishing gradient problem in neural networks? Multiple answers are possible?

A. Deeper neural networks tend to be more susceptible to vanishing gradients.

B. Using the ReLU function can reduce this problem.

C. If a network has the vanishing gradient problem for one training point due to the sigmoid function, it will also have a vanishing gradient for every other training point.

D. Networks with sigmoid functions don’t suffer from the vanishing gradient problem if trained with the cross-entropy loss.

Quiz. Let’s compare sigmoid with rectified linear unit (ReLU). Which of the following statement is NOT true?

A. Sigmoid function is more expensive to compute

B. ReLU has non-zero gradient everywhere

C. The gradient of Sigmoid is always less than 0.3

D. The gradient of ReLU is constant for positive input

Quiz. Let’s compare sigmoid with rectified linear unit (ReLU). Which of the following statement is NOT true?

A. Sigmoid function is more expensive to compute

B. ReLU has non-zero gradient everywhere

C. The gradient of Sigmoid is always less than 0.3

D. The gradient of ReLU is constant for positive input

Q5. A Leaky ReLU is defined as $f(x)=max(0.1x, x)$. Let f’(0)=1. Does it have non-zero gradient everywhere??

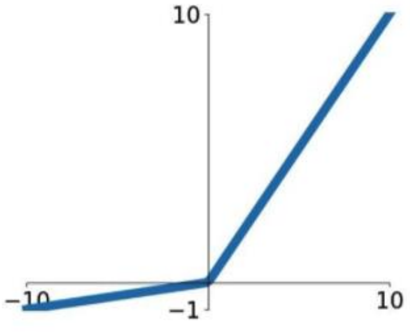
A. Yes

B. No

Q5. A Leaky ReLU is defined as *f(x)=max(0.1x, x)*. Let f'(0)=1. Does it have non-zero gradient everywhere??

A. Yes

B. No



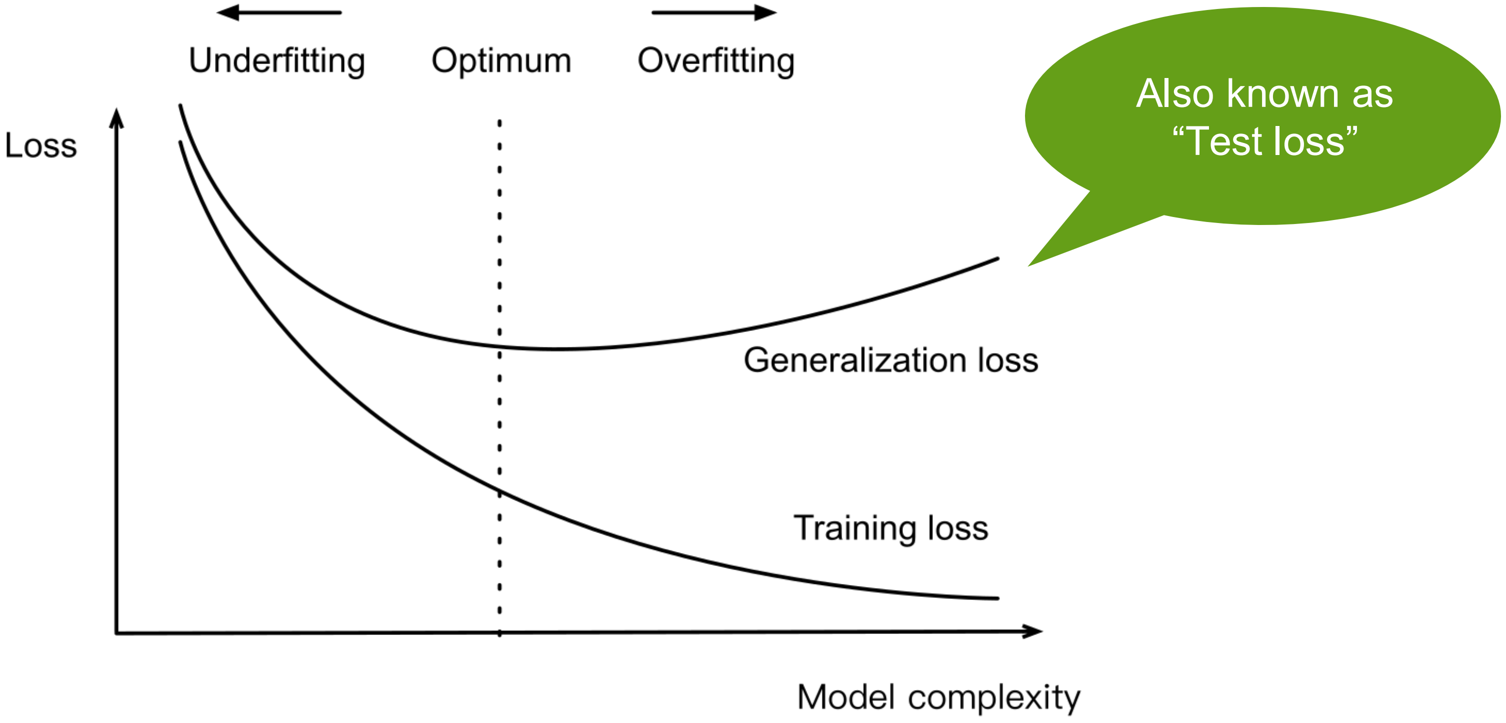
Generalization & Regularization

How good are the models?

Training Error and Generalization Error

- Training error: model error on the training data
- **Generalization error**: model error on new data
- Example: practice a future exam with past exams
  - Doing well on past exams (training error) doesn’t guarantee a good score on the future exam (generalization error)

Influence of Model Complexity



* Recent research has challenged this view for some types of models.

Quiz Break: When training a neural network, which one below indicates that the network has overfit the training data?

A. Training loss is low and generalization loss is high.
B. Training loss is low and generalization loss is low.
C. Training loss is high and generalization loss is high.
D. Training loss is high and generalization loss is low.
E. None of these.

Quiz Break: When training a neural network, which one below indicates that the network has overfit the training data?

A. Training loss is low and generalization loss is high.
B. Training loss is low and generalization loss is low.
C. Training loss is high and generalization loss is high.
D. Training loss is high and generalization loss is low.
E. None of these.

Quiz Break: Adding more layers to a multi-layer perceptron may cause _______.

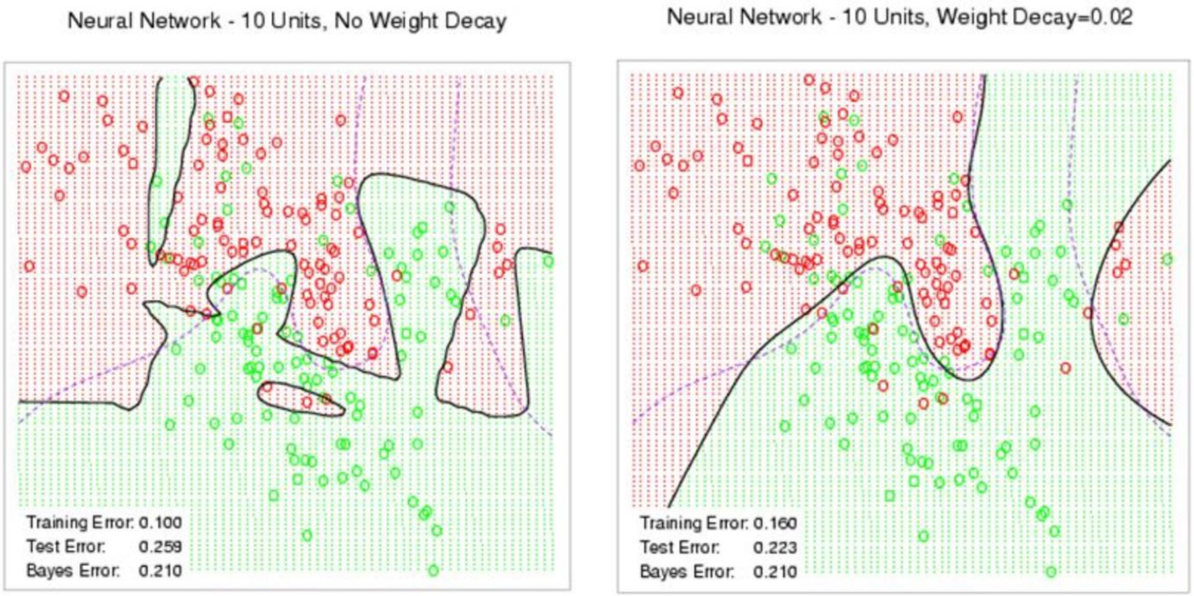
A. Vanishing gradients during back propagation.
B. A more complex decision boundary.
C. Underfitting.
D. Higher test loss.
E. None of these.

Quiz Break: Adding more layers to a multi-layer perceptron may cause _______. (Multiple answers)

A. Vanishing gradients during back propagation.
B. A more complex decision boundary.
C. Underfitting.
D. Higher test loss.
E. None of these.

How to regularize the model for better generalization?

Weight Decay

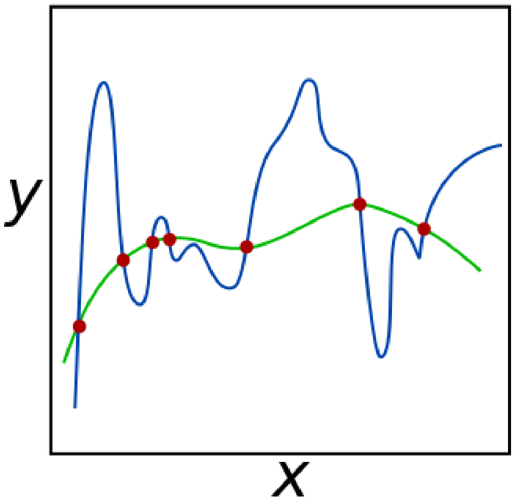


Squared Norm Regularization as Hard Constraint

- Reduce model complexity by limiting value range

$$\min L(\mathbf{w}, b) \text{ subject to } \|\mathbf{w}\|^2 \leq B$$

  – Often do not regularize bias b
- Doing or not doing has little difference in practice
  – A small B means more regularization



Squared Norm Regularization as Soft Constraint

- We can rewrite the hard constraint version as

$$minL(\mathbf{w}, b) + \frac{\lambda}{2} \parallel \mathbf{w} \parallel^2$$

Squared Norm Regularization as Soft Constraint

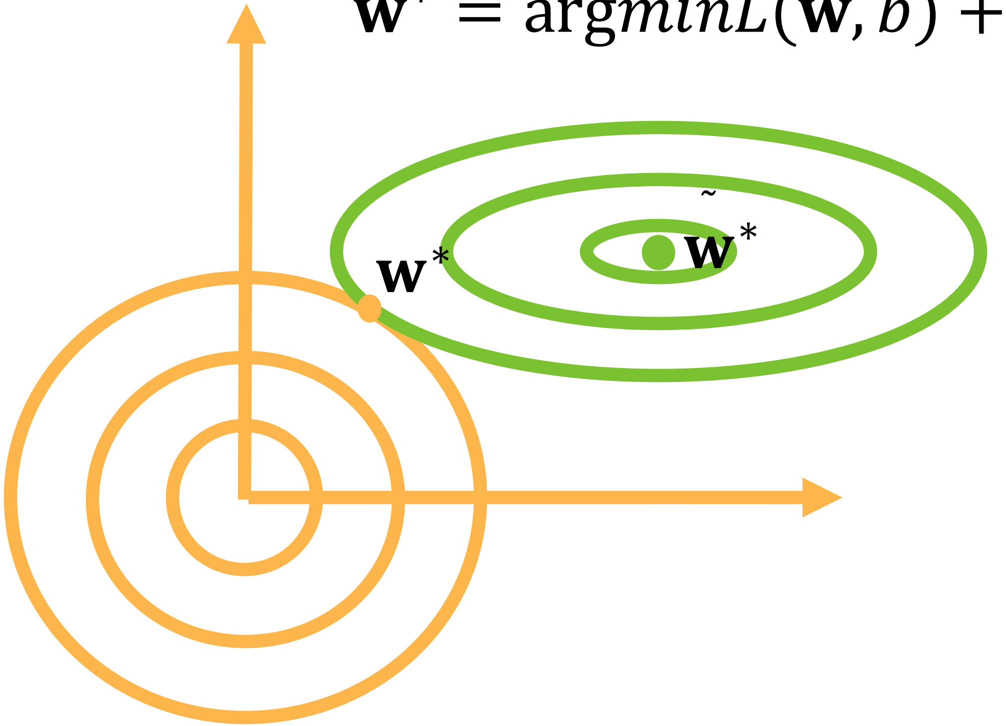
- We can rewrite the hard constraint version as

$$\min L(\mathbf{w}, b) + \frac{\lambda}{2} \| \mathbf{w} \|^2$$

  - Hyper-parameter λ controls regularization importance
  - $\lambda = 0$: no effect
  - $\lambda \to \infty, \mathbf{w}^* \to \mathbf{0}$

Illustrate the Effect on Optimal Solutions

$$\mathbf{w}^* = \mathrm{arg}min L(\mathbf{w}, b) + \frac{\lambda}{2} \| \mathbf{w} \|^2$$



$$\tilde{\mathbf{w}}^* = \mathrm{arg}min L(\mathbf{w}, b)$$

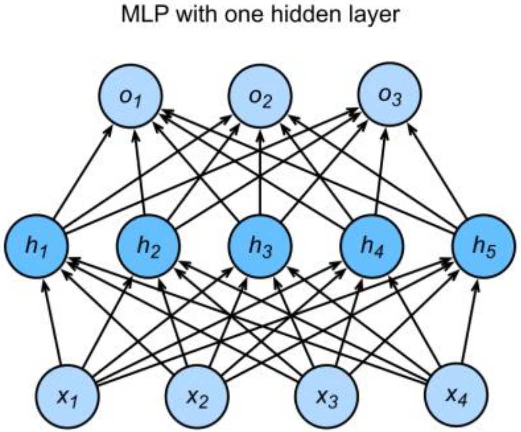
Dropout

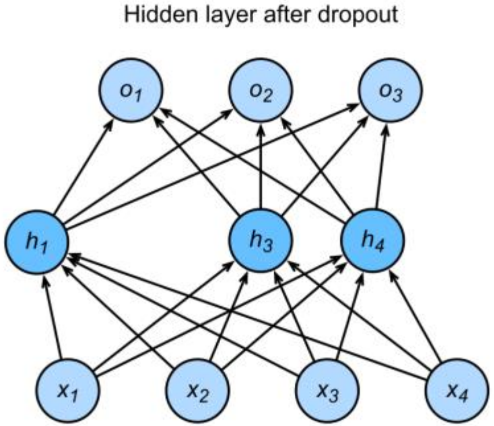
Hinton et al.

Apply Dropout

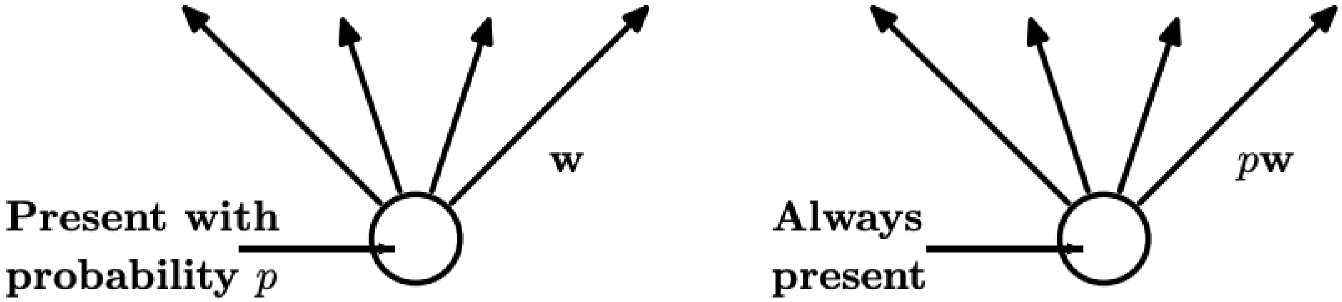
- Often apply dropout on the output of hidden fully-connected layers

$$\mathbf{h} = \sigma(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)})$$
$$\mathbf{h}' = \text{dropout}(\mathbf{h})$$
$$\mathbf{o} = \mathbf{W}^{(2)}\mathbf{h}' + \mathbf{b}^{(2)}$$
$$\mathbf{p} = \text{softmax}(\mathbf{o})$$





Dropout



(a) At training time

(b) At test time

Figure 2: **Left**: A unit at training time that is present with probability p and is connected to units in the next layer with weights $\mathbf{w}$. **Right**: At test time, the unit is always present and the weights are multiplied by p. The output at test time is same as the expected output at training time.

Dropout

Hinton et al.

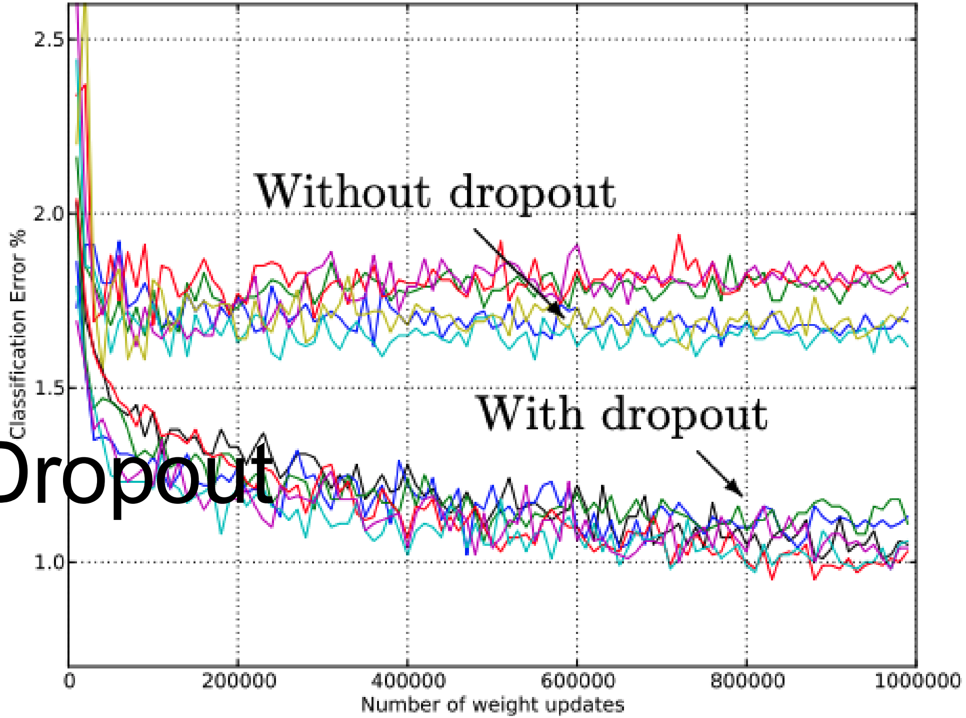
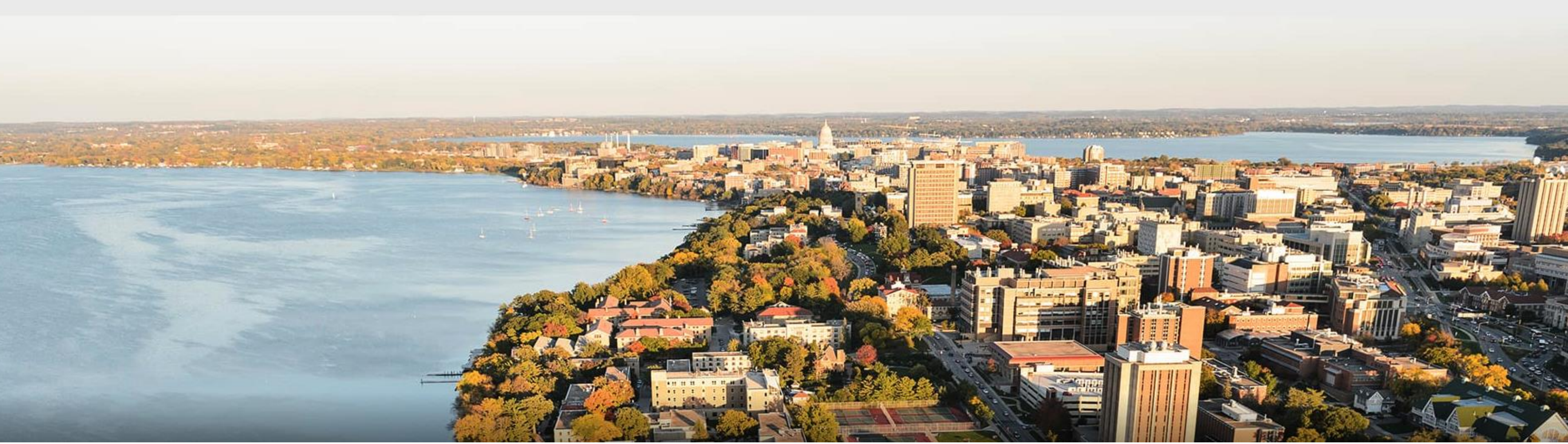


Figure 4: Test error for different architectures with and without dropout. The networks have 2 to 4 hidden layers each with 1024 to 2048 units.



Convolutional Neural Networks (CNNs)

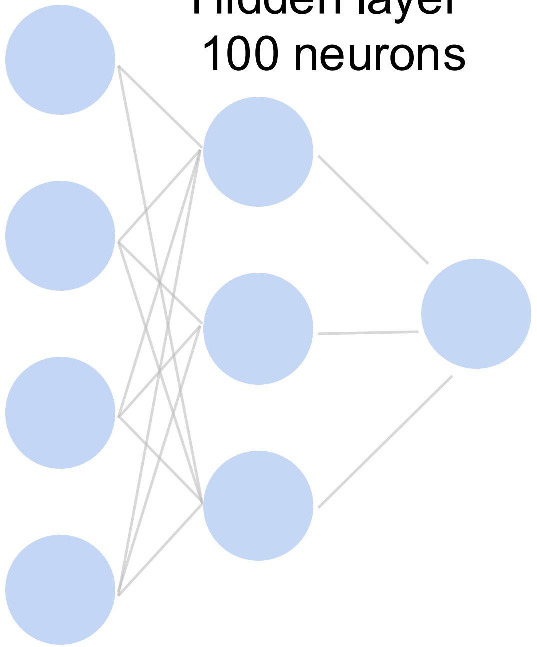
How to classify

Cats vs. dogs?

36M floats in a RGB image!

Fully Connected Networks

Cats vs. dogs?



~ 36M elements x 100 = **~3.6B** parameters!

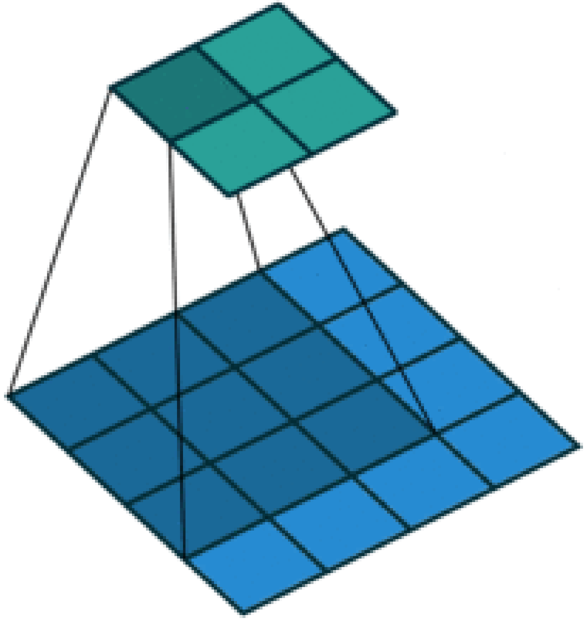
Where is Waldo?

- Translation Invariance
- Locality

2-D Convolution

$$\underset{\text{Input}}{\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}} * \underset{\text{Kernel}}{\begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix}} = \underset{\text{Output}}{\begin{bmatrix} 19 & 25 \\ 37 & 43 \end{bmatrix}}$$

$$0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19,$$
$$1 \times 0 + 2 \times 1 + 4 \times 2 + 5 \times 3 = 25,$$
$$3 \times 0 + 4 \times 1 + 6 \times 2 + 7 \times 3 = 37,$$
$$4 \times 0 + 5 \times 1 + 7 \times 2 + 8 \times 3 = 43.$$



(vdumoulin@ Github)

2-D Convolution Layer

$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix} * \begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix} = \begin{bmatrix} 19 & 25 \\ 37 & 43 \end{bmatrix}$$

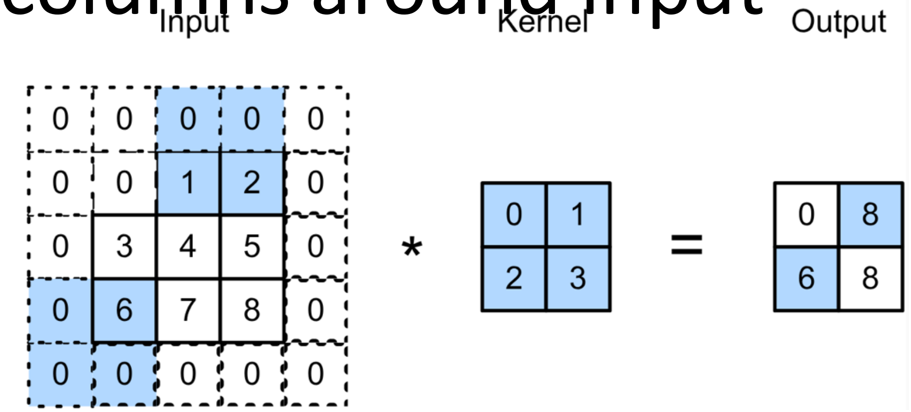
- $\mathbf{X}: n_h \times n_w$ input matrix
- $\mathbf{W}: k_h \times k_w$ kernel matrix
- b: scalar bias
- $\mathbf{Y}: (n_h - k_h + 1) \times (n_w - k_w + 1)$ output matrix

$$\mathbf{Y} = \mathbf{X} \star \mathbf{W} + b$$

- $\mathbf{W}$ and b are learnable parameters

2-D Convolution Layer with Stride and Padding

- Stride is the #rows/#columns per slide
- Padding adds rows/columns around input
- Output shape



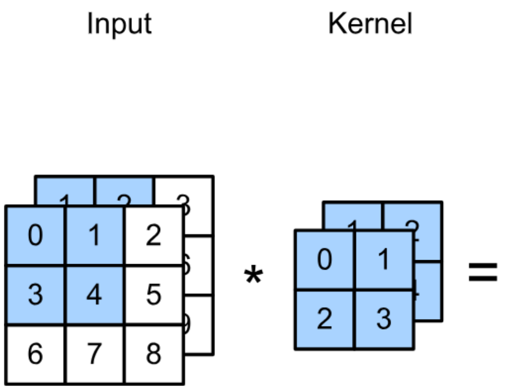
Kernel/filter size

$$\lfloor (n_h - k_h + p_h + s_h)/s_h \rfloor \times \lfloor (n_w - k_w + p_w + s_w)/s_w \rfloor$$

Input size **Pad** **Stride**

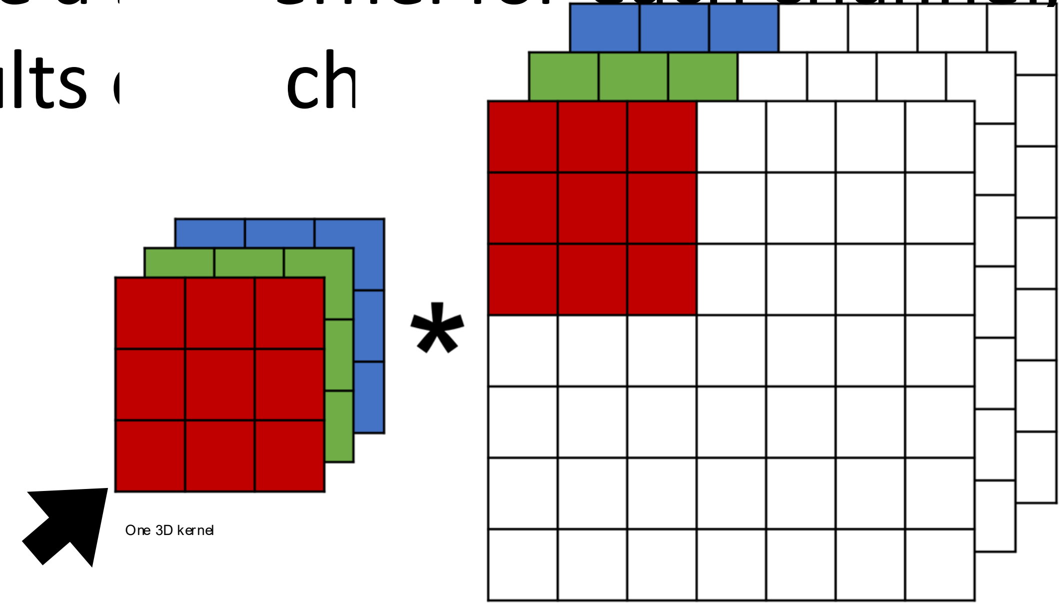
Multiple Input Channels

- Input and kernel can be 3D, e.g., an RGB image have 3 channels
- Have a kernel for each channel,



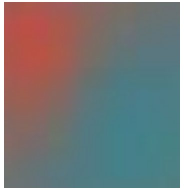
Multiple Input Channels

- Input and kernel can be 3D, e.g., an RGB image have 3 channels
- Have a 2D kernel for each channel, and then sum results [illegible] ch



Multiple Input Channels

- Input and kernel can be 3D, e.g., an RGB image have 3 channels
- Also call each 3D kernel a "**filter**", which produce only **one** outp[illegible] mation over channels)



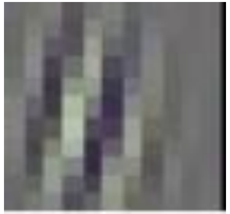
One filter
(3 channels)

*

RGB (3 input channels)

Multiple filters (in one layer)

- Apply multiple filters on the input
- Each filter may learn different features about the input
- Each filt 3D output channel



A different filter

*

RGB (3 input channels)

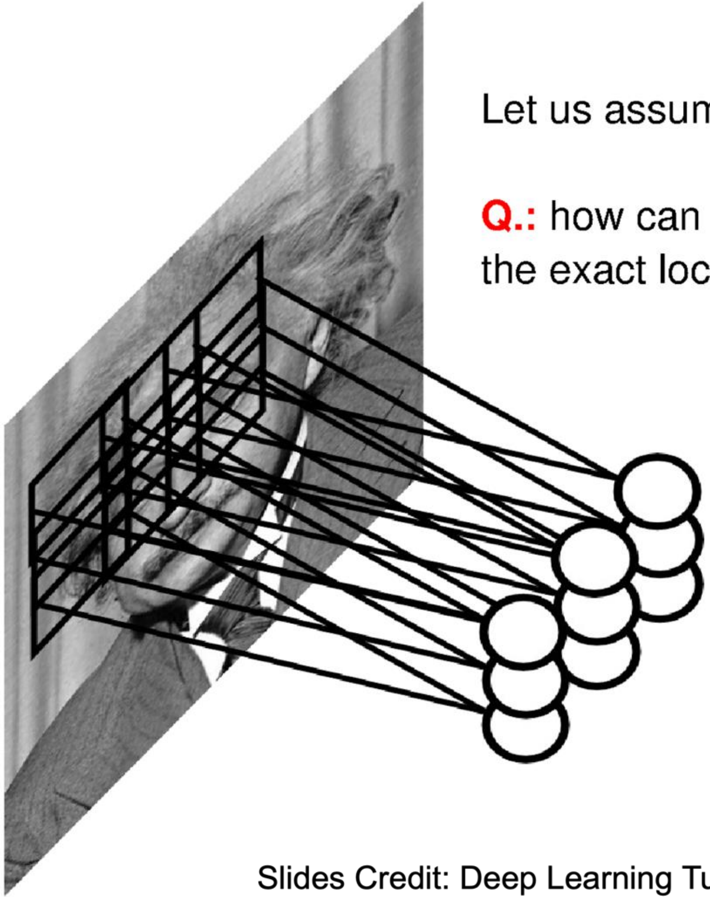
Multiple Output Channels

- The # of output channels = # of filters
- Input $\mathbf{X}: c_i \times n_h \times n_w$
- Kernel $\mathbf{W}: \boxed{c_o} \times c_i \times k_h \times k_w$
- Output $\mathbf{Y}: \boxed{c_o} \times m_h \times m_w$

$$\mathbf{Y}_{i,:,:} = \mathbf{X} \star \mathbf{W}_{i,:,:,:}$$

$$for\, i = 1, \ldots, c_o$$

Pooling

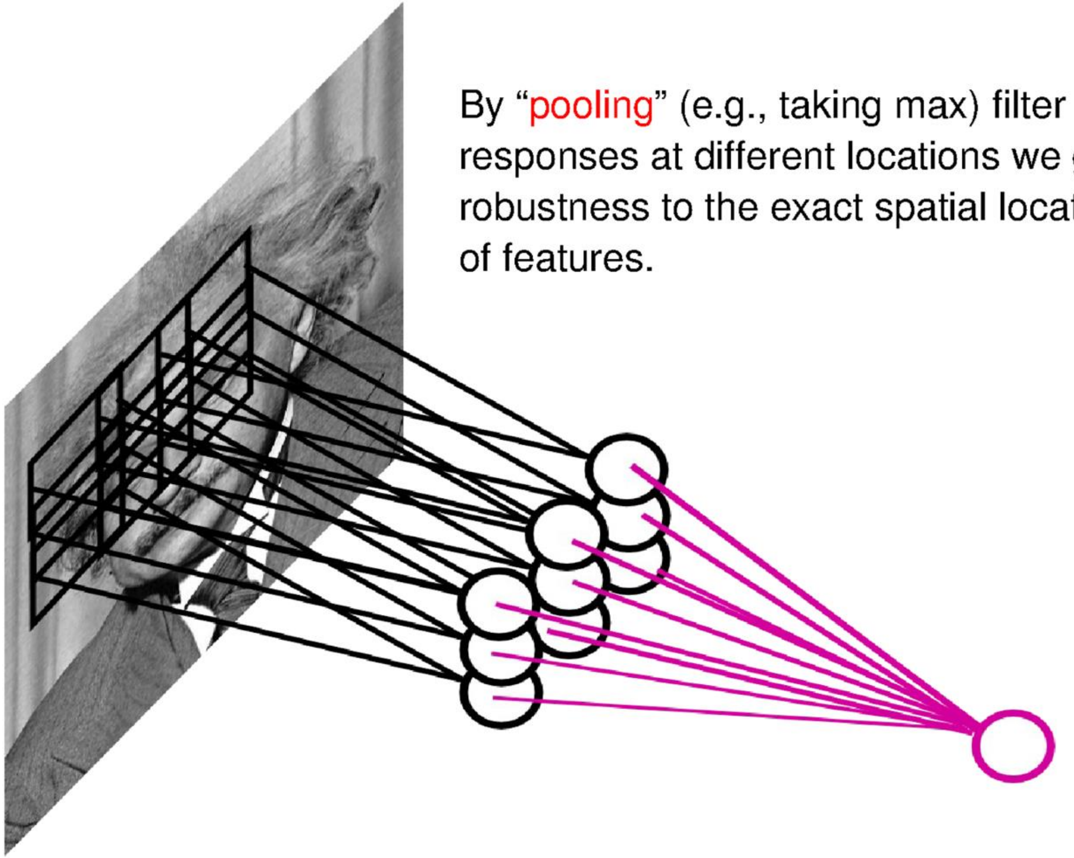


Let us assume filter is an "eye" detector.

Q.: how can we make the detection robust to the exact location of the eye?

Slides Credit: Deep Learning Tutorial by Marc'Aurelio Ranzato

Pooling

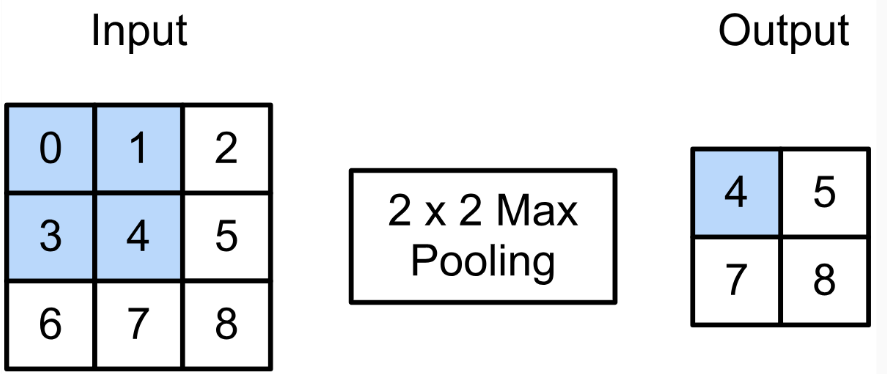


By "pooling" (e.g., taking max) filter responses at different locations we gain robustness to the exact spatial location of features.

Slides Credit: Deep Learning Tutorial by Marc'Aurelio Ranzato

2-D Max Pooling

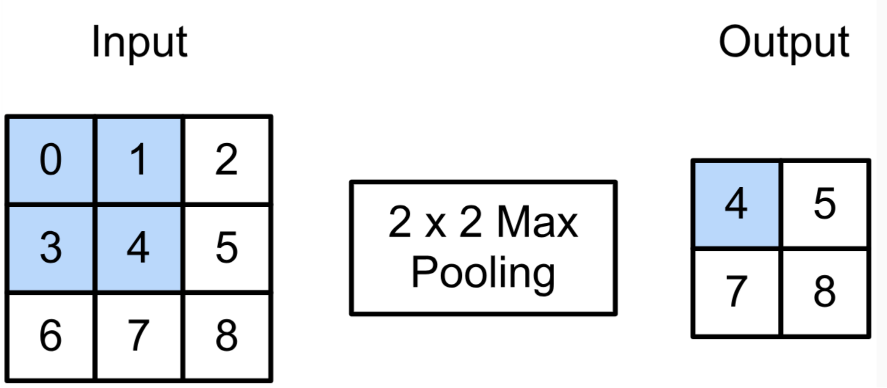
- Returns the maximal value in the sliding window

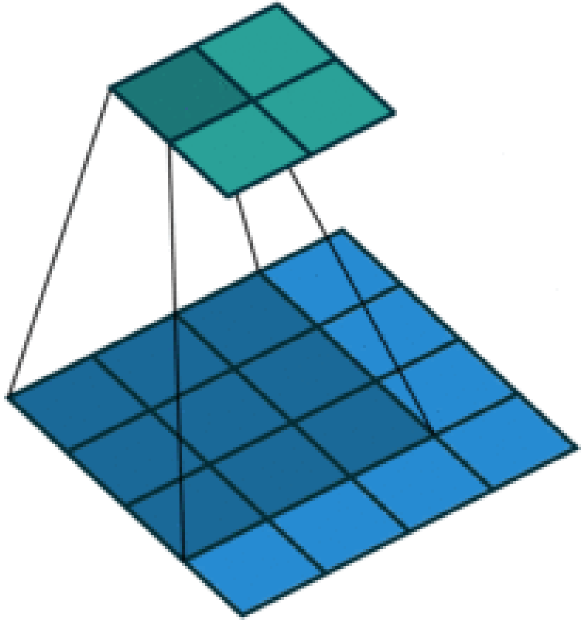


max(0,1,3,4) = 4

2-D Max Pooling

- Returns the maximal value in the sliding window





$$\max(0,1,3,4) = 4$$

Padding, Stride, and Multiple Channels

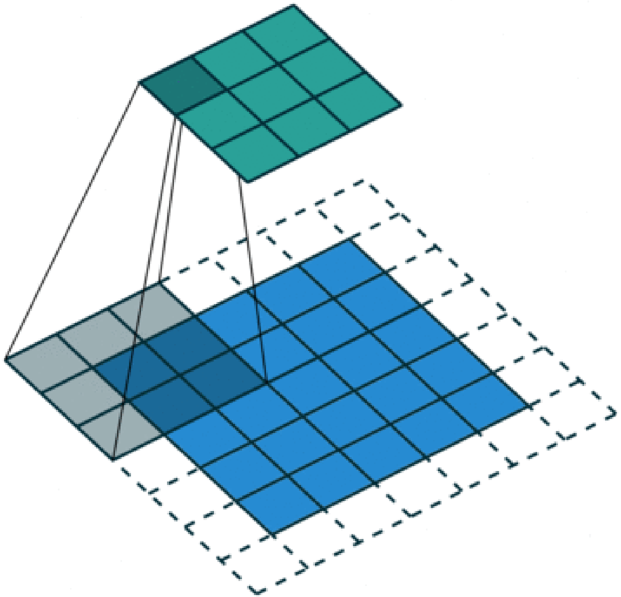
- Pooling layers have similar padding and stride as convolutional layers
- No learnable parameters
- Apply pooling for each input channel to obtain the corresponding output channel

  #output channels = #input channels

Padding, Stride, and Multiple Channels

- Pooling layers have similar padding and stride as convolutional layers
- No learnable parameters
- Apply pooling for each input channel to obtain the corresponding output channel

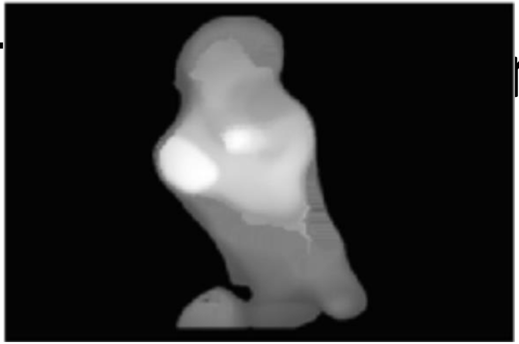
#output channels = #input channels



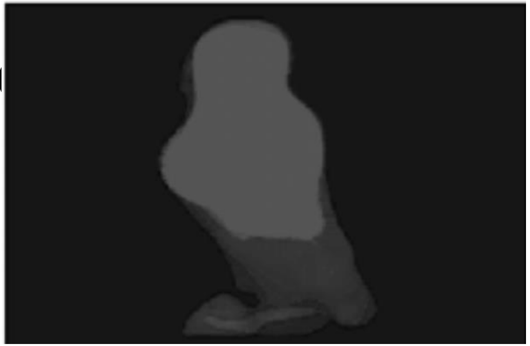
Average Pooling

- Max pooling: the strongest pattern signal in a window
- Average pooling: replace max with mean in max pooling
  - T nal str ow

Max pooling



Average pooling



Consider a convolution layer with 16 filters. Each filter has a size of 11x11x3, a stride of 2x2. Given an input image of size 22x22x3, if we don't allow a filter to fall outside of the input, what is the output size?

- 11x11x16
- 6x6x16
- 7x7x16
- 5x5x16

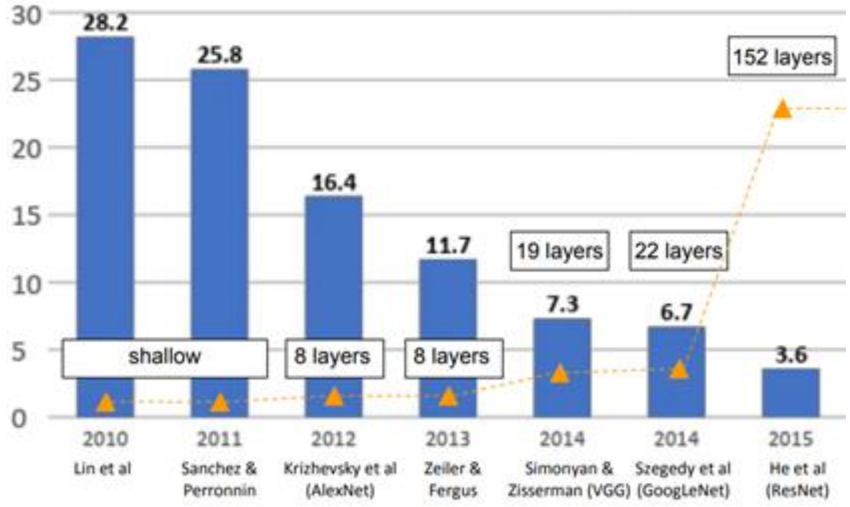
Consider a convolution layer with 16 filters. Each filter has a size of 11x11x3, a stride of 2x2. Given an input image of size 22x22x3, if we don't allow a filter to fall outside of the input, what is the output size?

- 11x11x16
- $\boxed{\text{6x6x16}}$
- 7x7x16
- 5x5x16

$$\lfloor (n_h - k_h + p_h + s_h)/s_h \rfloor \times \lfloor (n_w - k_w + p_w + s_w)/s_w \rfloor$$

Evolution of CNNs

ImageNet competition (error rate)



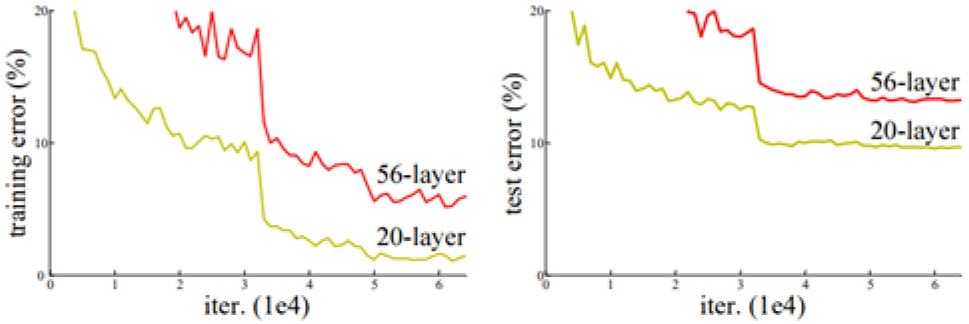
Credit: Stanford CS 231n

Simple Idea: Add More Layers

VGG: 19 layers. ResNet: 152 layers. **Add more layers**… sufficient?

- No! Some problems:
  - i) Vanishing gradients: more layers ➔ more likely
  - ii) Instability: deeper models are harder to optimize

Reflected in training error:

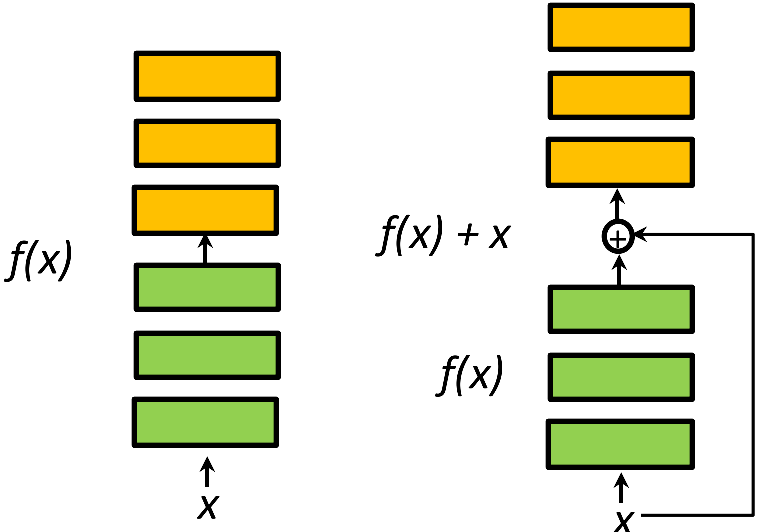


He et al: “Deep Residual Learning for Image Recognition”

Residual Connections

Idea: Identity might be hard to learn, but zero is easy!

- Make all the weights tiny, produces zero for output
- Can easily transform learning identity to learning zero:



Left: Conventional layers block

Right: **Residual** layer block

To learn identity $f(x) = x$, layers now need to learn $f(x) = 0$ ➔ easier

Thank you and good luck!