CS 540 Introduction to Artificial Intelligence
Review

University of Wisconsin-Madison
Spring 2025

Announcements

- Homework:
  - **HW10** due on Friday **May 2nd at 11:59 PM**

- Office hours: Monday **May 5th 10:30 AM – 12:30 PM**

- Course evaluation ending **tomorrow May 2nd**
  - **If participation reaches 50%,**
    we'll reveal more information for the final

Final Information

- **Time: May 7th 07:45 AM - 09:45 AM**
- Location (by section**):
  - **Lecture 003 (Instructor Blerina Gkotse): B10 Ingraham Hall**

- **All alternate and McBurney exams have been set up, you should have received an email with the time and location.**

- **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 + past exams**: https://piazza.com/class/m5zvrf0clyo3sl/post/449

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

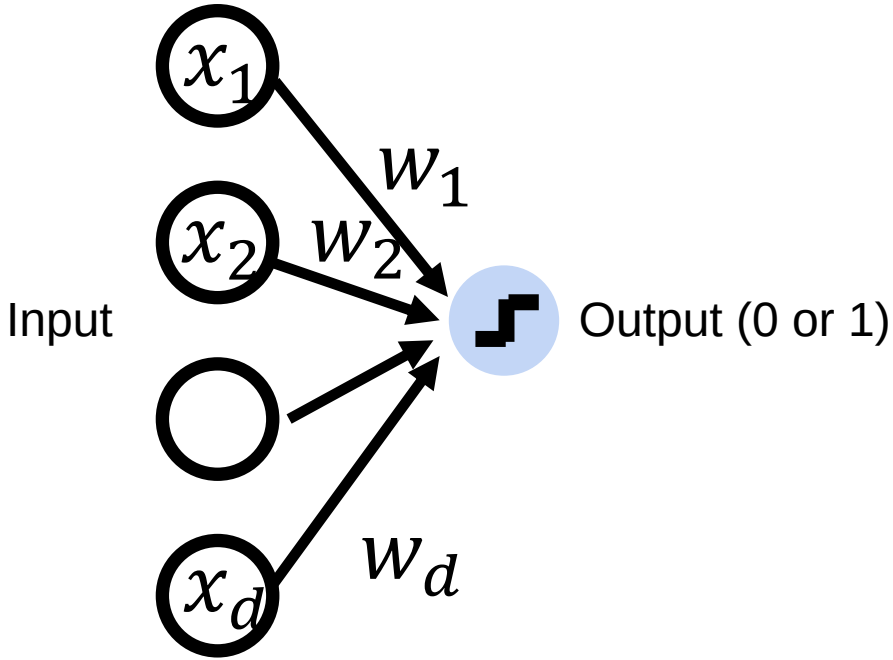
- Given input $\mathbf{x}$, weight $\mathbf{w}$ and bias b , perceptron outputs:

$$o = \sigma(\mathbf{w}^\top \mathbf{x} + b)$$

$$\sigma(x) = \begin{cases} 1 & if\ x > 0 \\ 0 & otherwise \end{cases}$$

Activation function

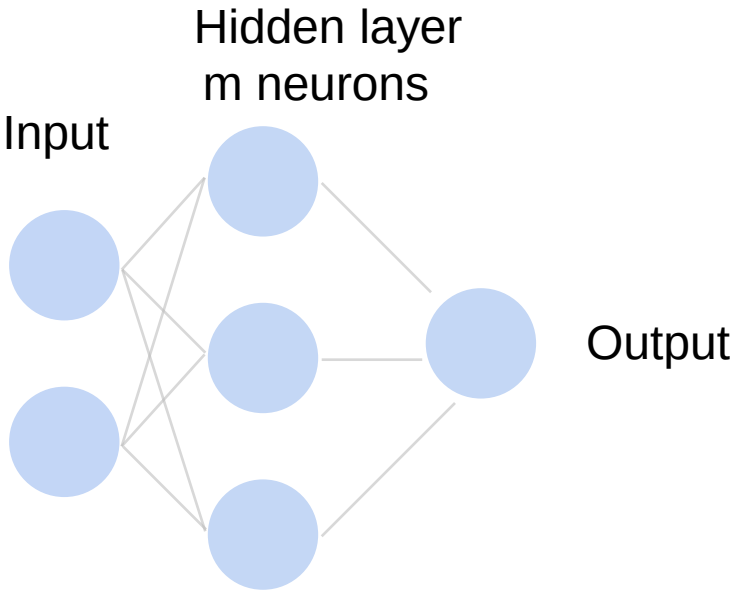
Cats vs. dogs?



Single Hidden Layer

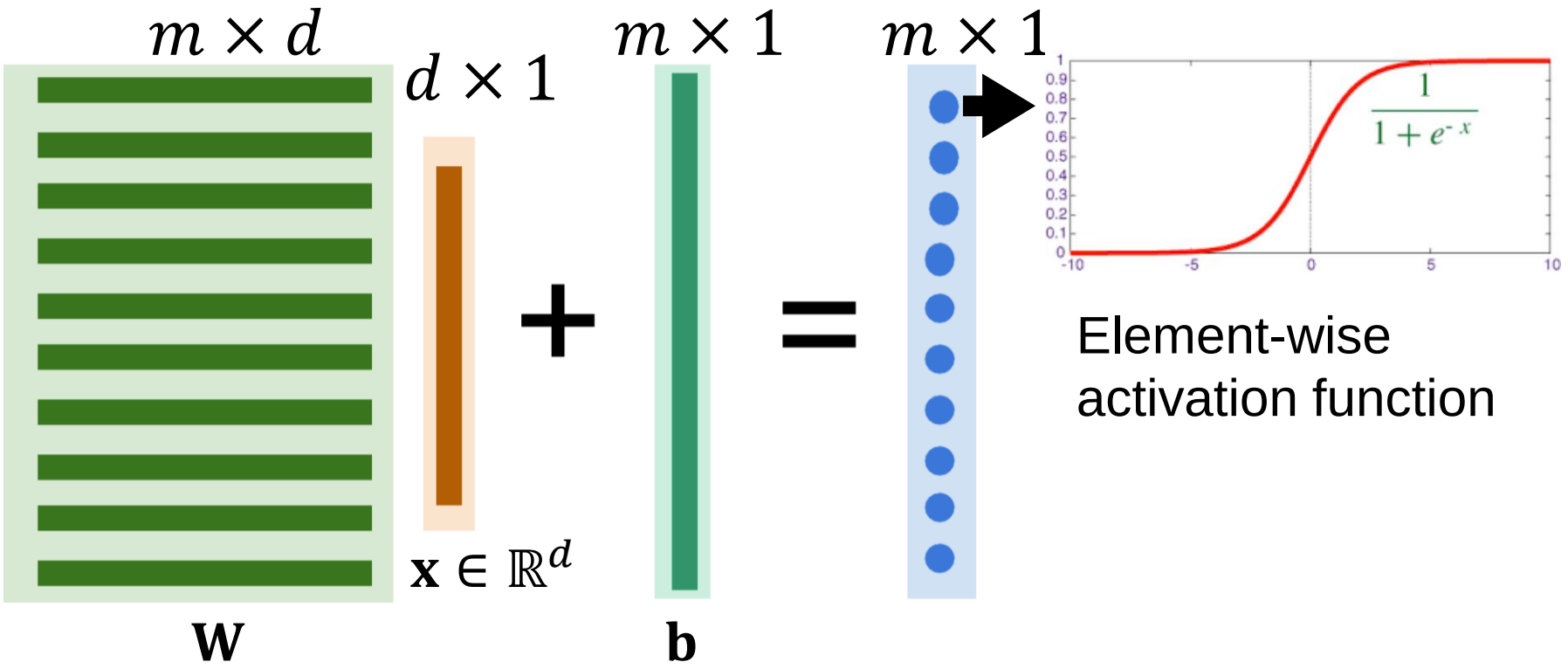
How to classify

Cats vs. dogs?



Neural networks with one hidden layer

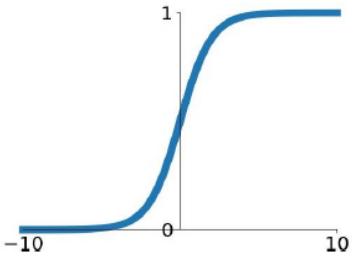
Key elements: linear operations + Nonlinear activations

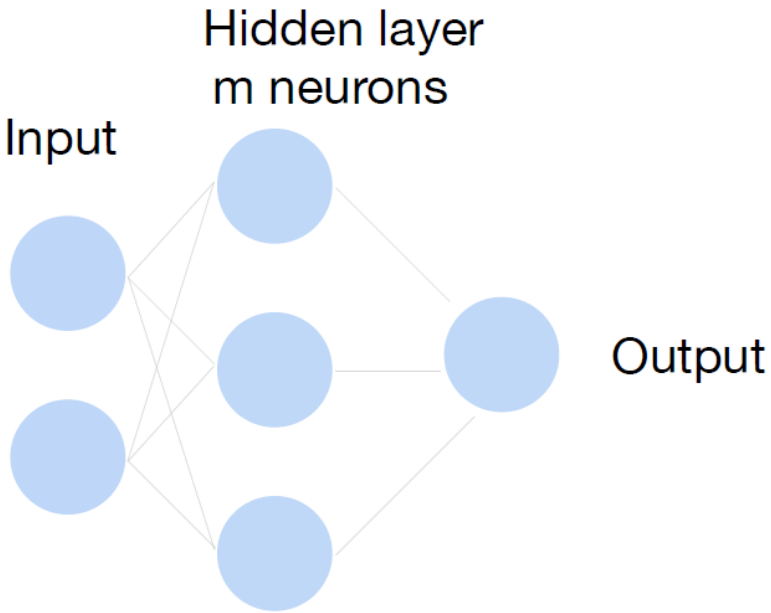


Single Hidden Layer

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

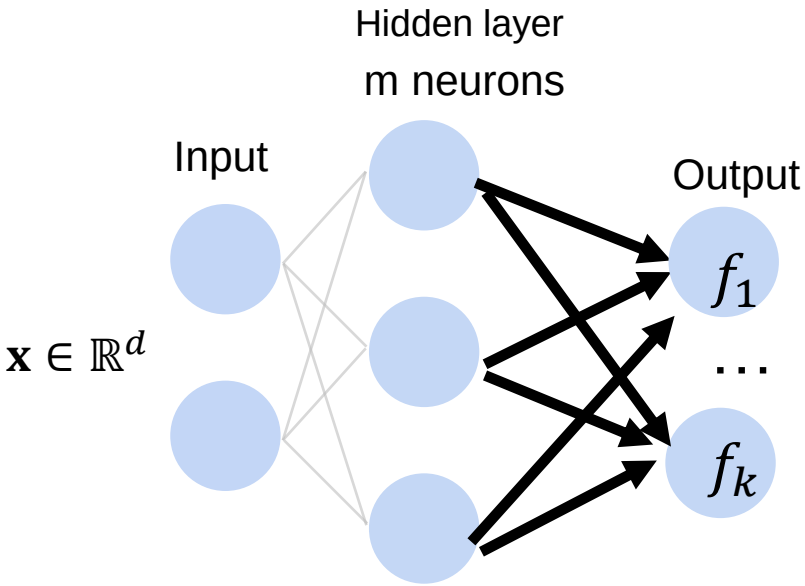
$$p(y = 1 \mid \mathbf{x}) = \frac{1}{1 + e^{-f}}$$





Multi-class classification

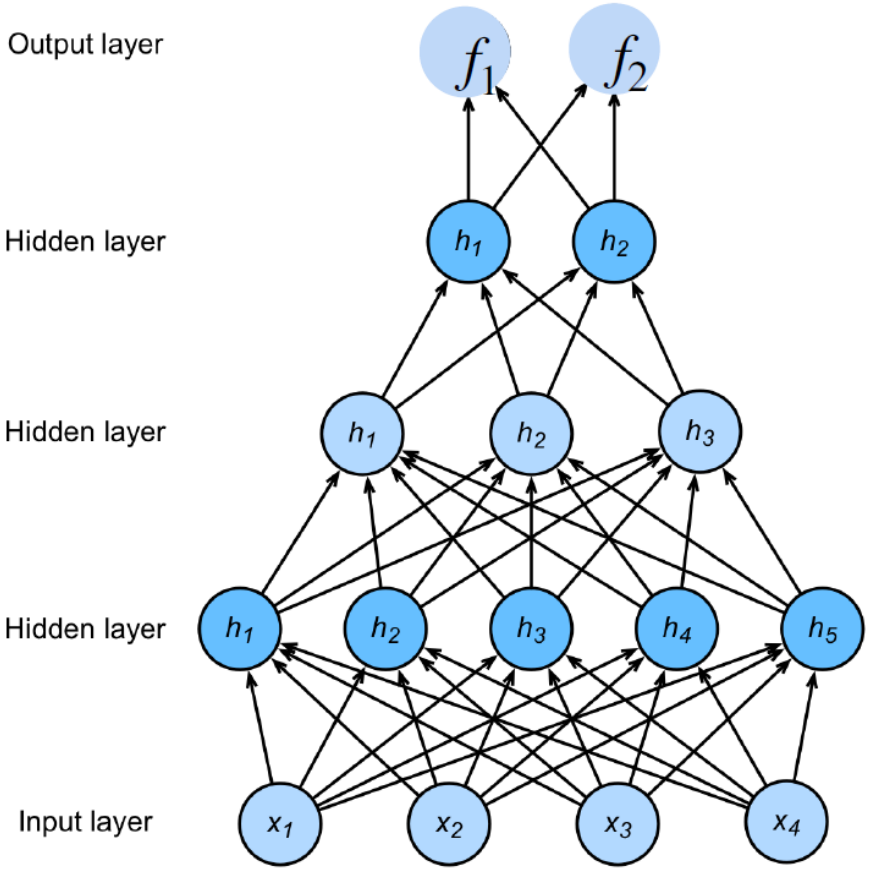
Turns outputs *f* into *k* probabilities (sum up to 1 across *k* classes)



$$p(y|\mathbf{x}) = softmax(\mathbf{f})$$

$$= \frac{\exp f_y(x)}{\sum_i^k \exp f_i(x)}$$

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

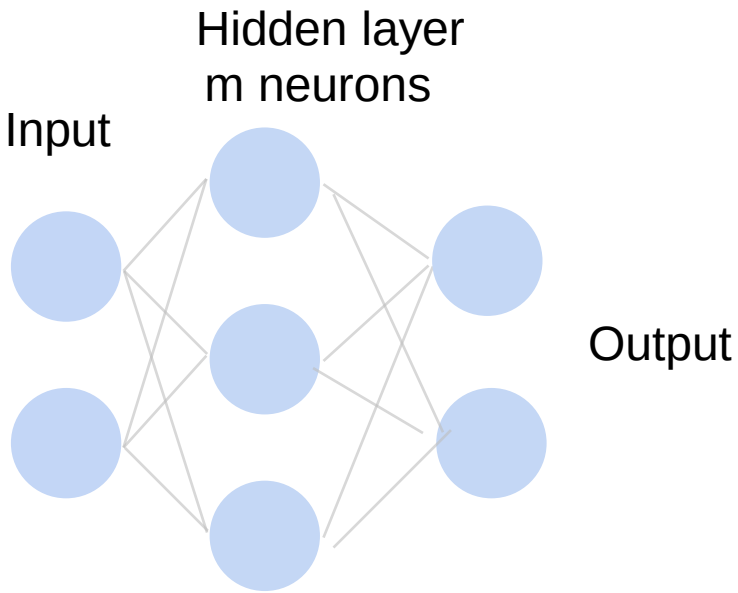
How to train a neural network?

Loss function: $\frac{1}{|D|}\sum_i \ell(\mathbf{x}_i, y_i)$

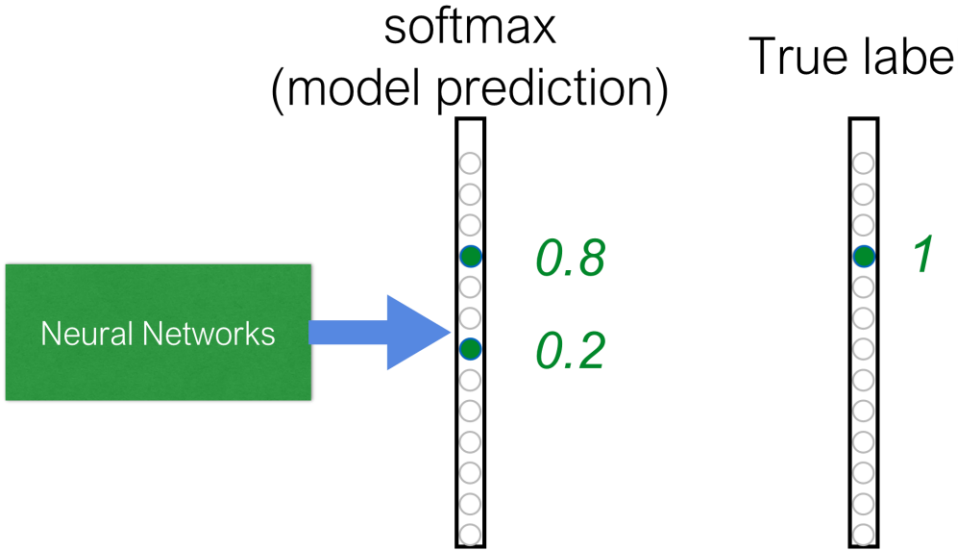
Per-sample loss:

$$\ell(\mathbf{x}, y) = \sum_{j=1}^{K} -y_j \log p_j$$

Also known as cross-entropy loss or softmax loss



Cross-Entropy Loss



$$L_{CE} = \sum_j -y_j \log(p_j)$$

$$= -\log(0.8)$$

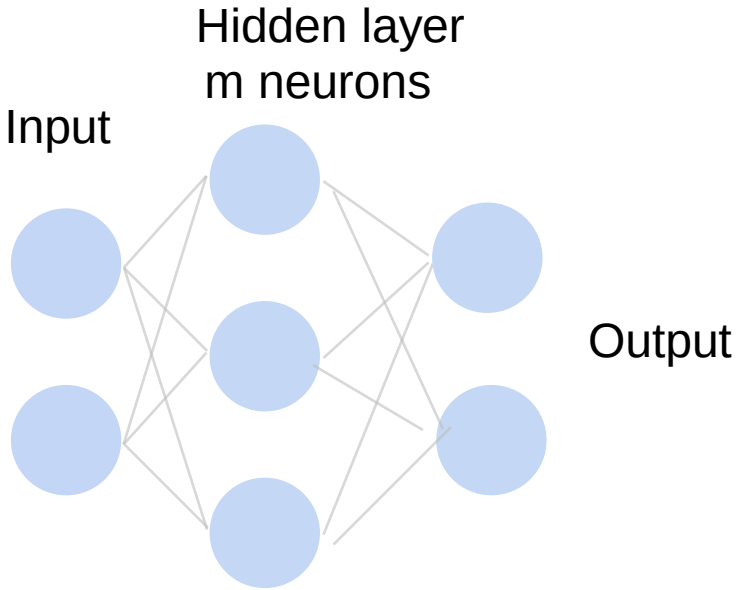
Goal: push ***p*** and ***Y*** to be identical

How to train a neural network?

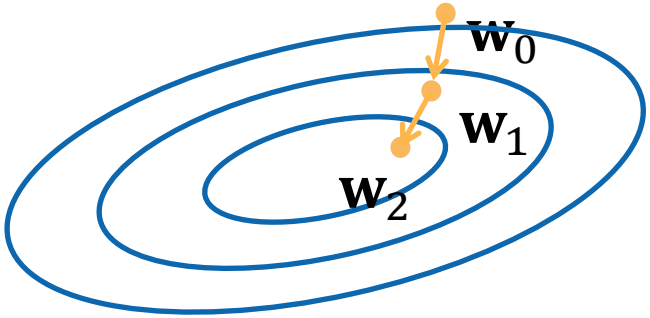
Update the weights W to minimize the loss function

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

Use gradient descent!



Gradient Descent



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

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

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

D can be very large. Expensive per iteration

- 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

Numerical Stability

Gradients for Neural Networks

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

$$\frac{\partial \ell}{\partial \mathbf{W}^t} = \frac{\partial \ell}{\partial \mathbf{h}^d} \underbrace{\frac{\partial \mathbf{h}^d}{\partial \mathbf{h}^{d-1}} \cdots \frac{\partial \mathbf{h}^{t+1}}{\partial \mathbf{h}^t}} \frac{\partial \mathbf{h}^t}{\partial \mathbf{W}^t}$$

Multiplication of *many* matrices

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

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

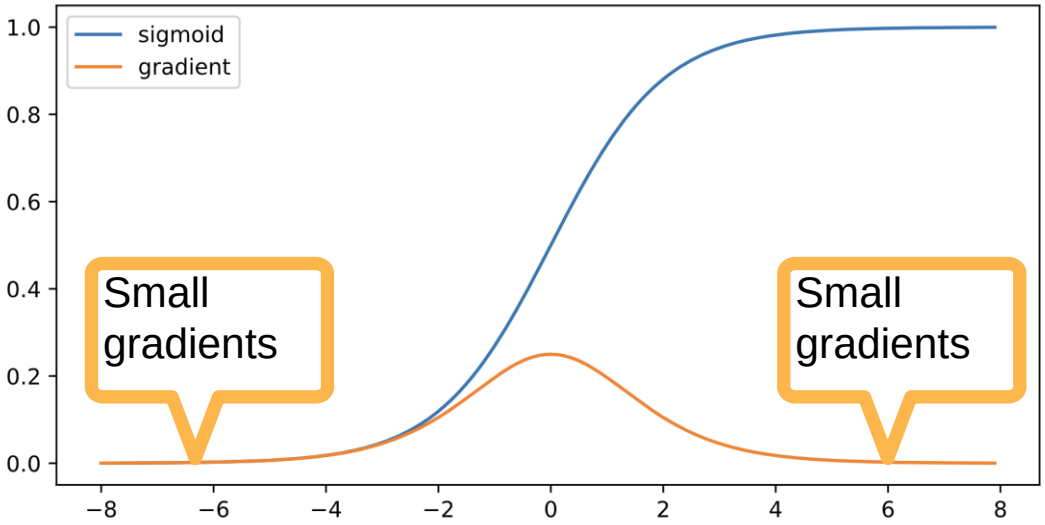
Issues with Gradient Exploding

- 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

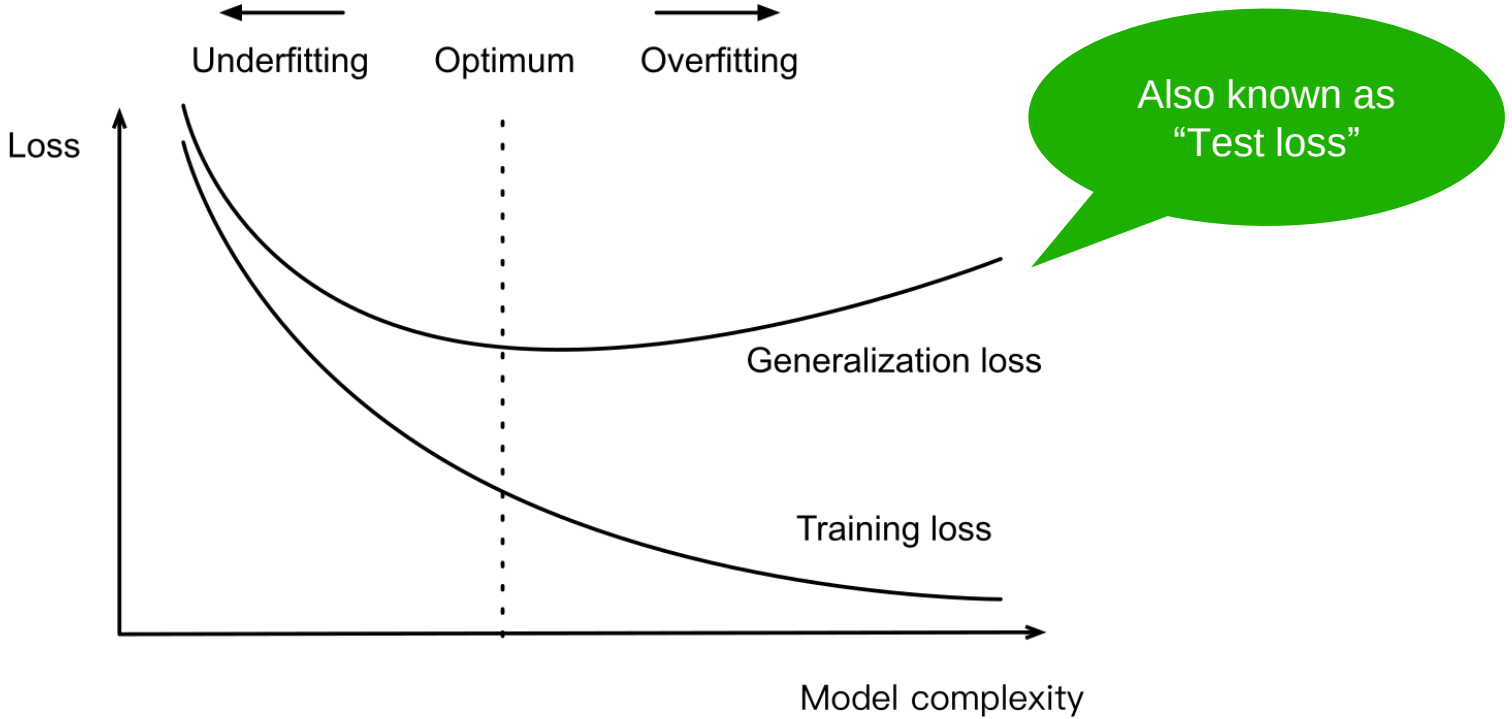
D. The gradient of ReLU is constant for positive input

Generalization & Regularization

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.

Convolutional Neural Networks (CNNs)

How to classify

Cats vs. dogs?

Dual

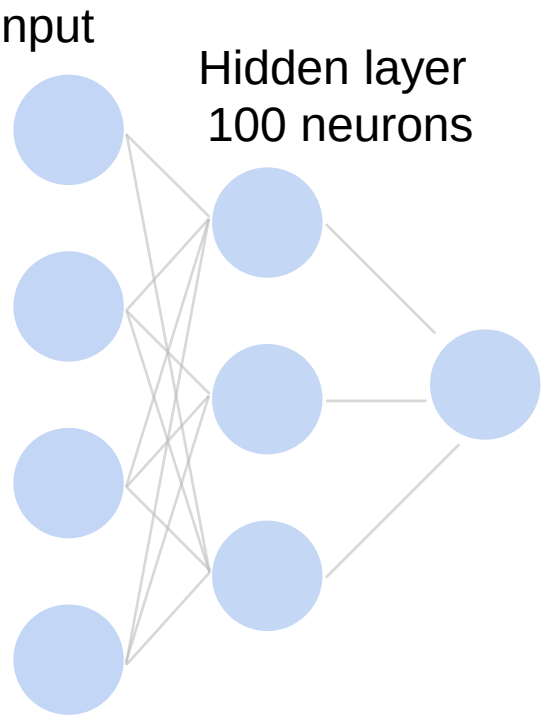
12MP

wide-angle and
telephoto cameras

36M floats in a RGB image!

Fully Connected Networks

Cats vs. dogs?



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

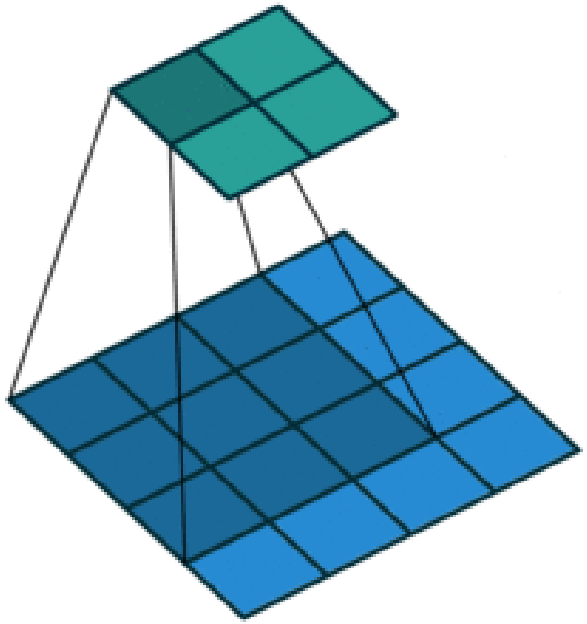
Why Convolution?

- Translation Invariance
- Locality

2-D Convolution

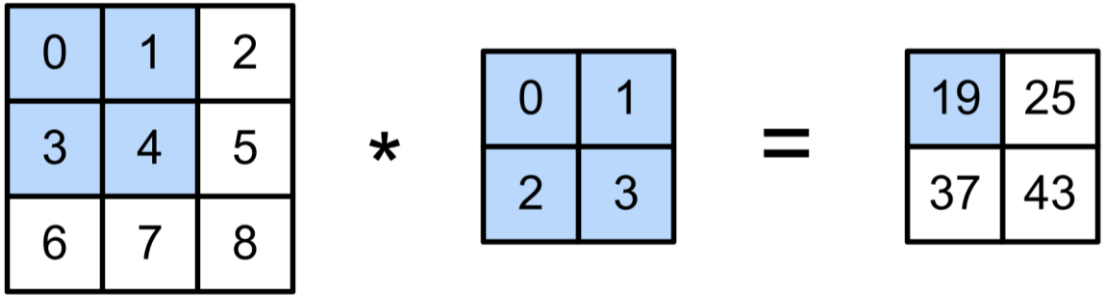
Input				Kernel			Output	
0	1	2	*	0	1	=	19	25
3	4	5		2	3		37	43
6	7	8						

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



(vdumoulin@ Github)

2-D Convolution Layer



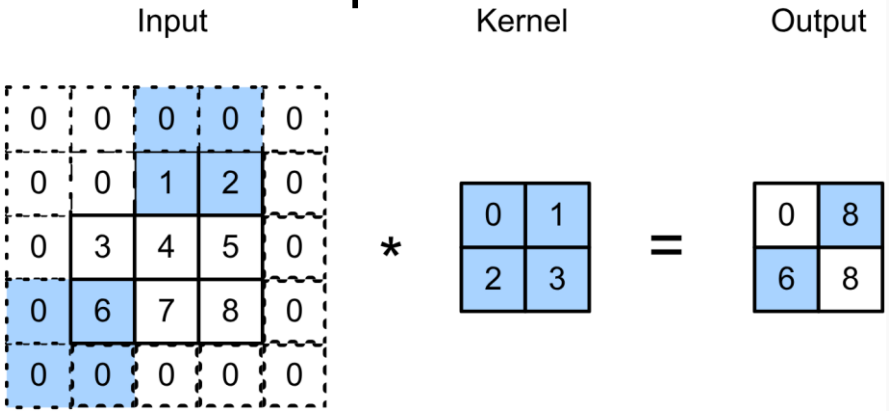
- $\mathbf{X}$: $n_h \times n_w$ input matrix
- $\mathbf{W}$: $k_h \times k_w$ kernel matrix
- b: scalar bias
- $\mathbf{Y}$: $(n_h - k_h + 1) \times (n_w - k_w + 1)$ output matrix

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

- **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



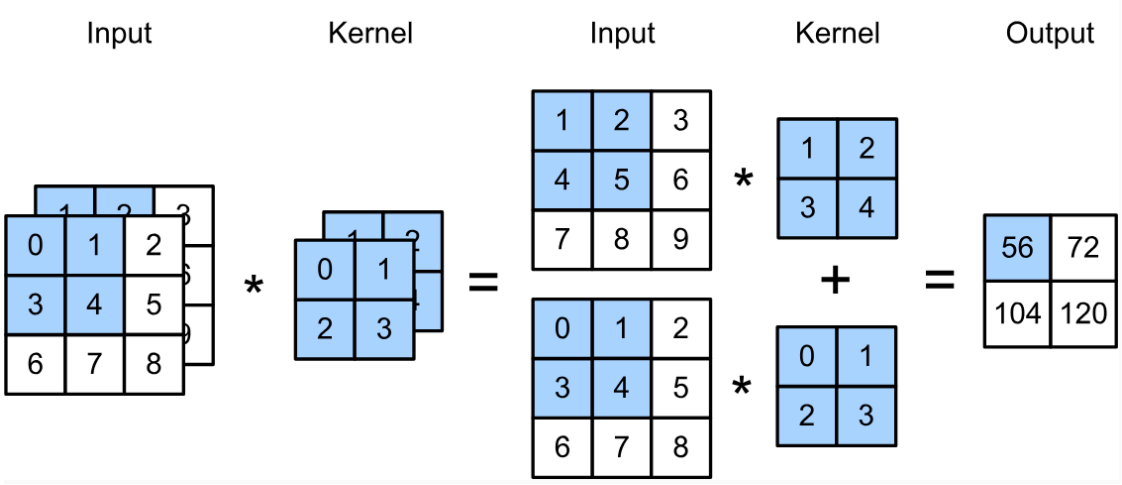
Kernel/filter size

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

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

Multiple Input Channels

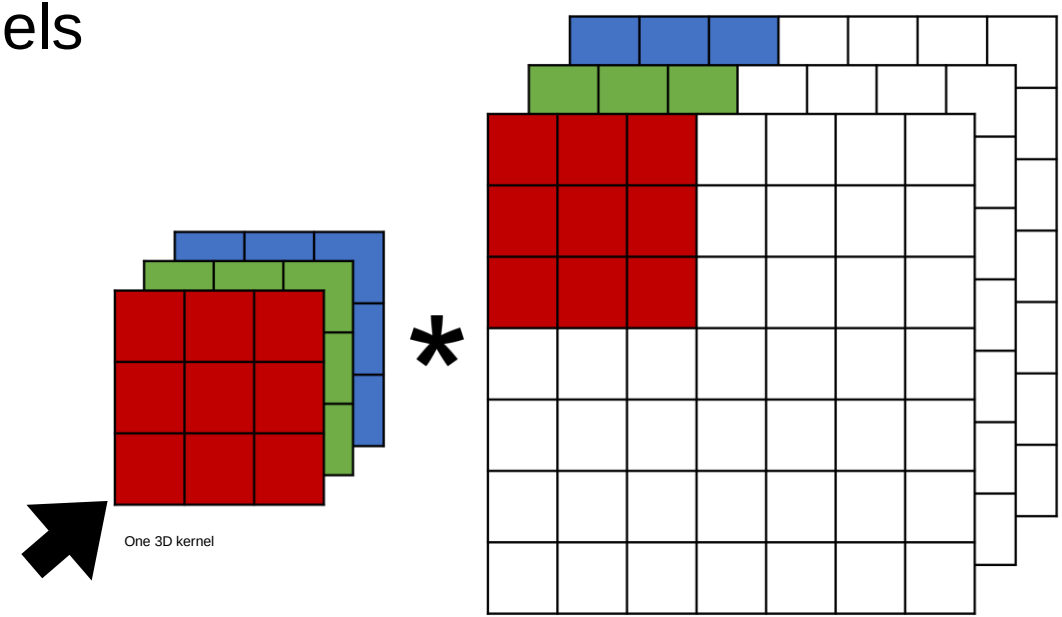
- Input and kernel can be 3D, e.g., an RGB image have 3 channels
- Have a kernel for each channel, and then sum results over channels



$$(1 \times 1 + 2 \times 2 + 4 \times 3 + 5 \times 4) + (0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3) = 56$$

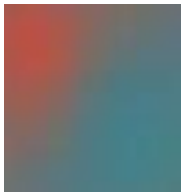
Multiple Input Channels

- 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 over channels



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** output channel (due to summation over channels)



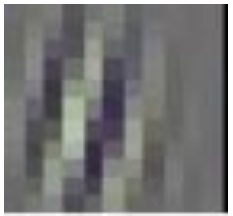
One filter
(3 channels)

*

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 filter (3D kernel) produces one 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 $\mathbf{W}$: $c_o \times c_i \times k_h \times k_w$
- Output $\mathbf{Y}$: $c_o \times m_h \times m_w$

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

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

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?

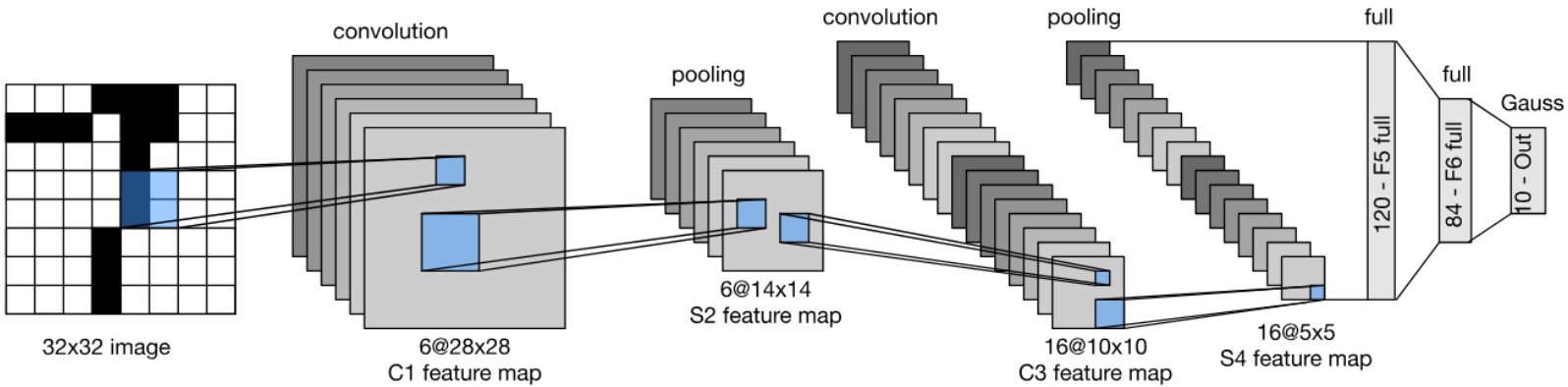
- 11x11x16
- 6x6x16
- 7x7x16
- 5x5x16

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
- 6x6x16
- 7x7x16
- 5x5x16

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

Convolutional Neural Network Architecture



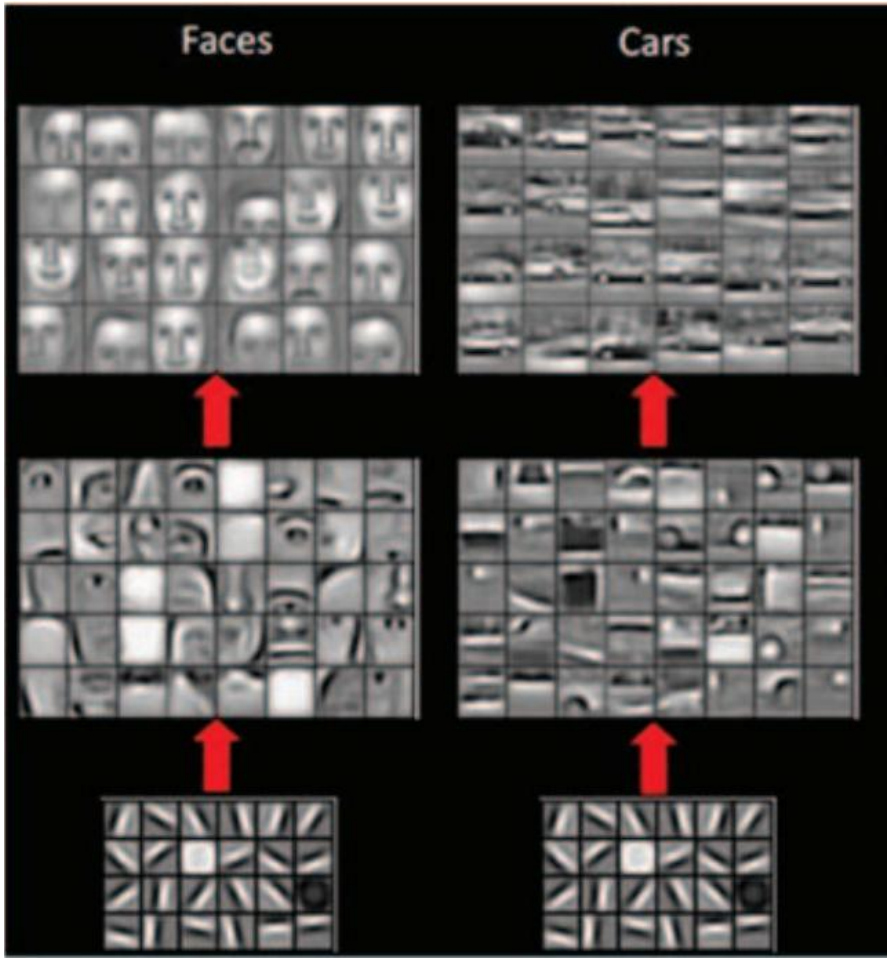
Gradient-based learning applied to document recognition, by Y. LeCun, L. Bottou, Y. Bengio and P. Haffner

Feature Learning

Later layers recognize complete objects

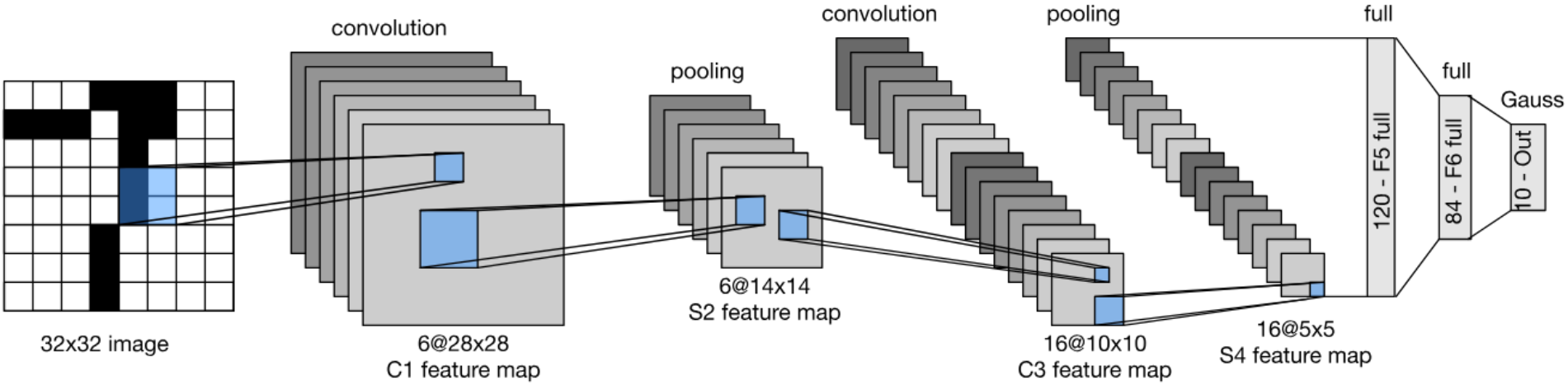
Middle layers recognize parts of objects

Early layers recognize simple patterns



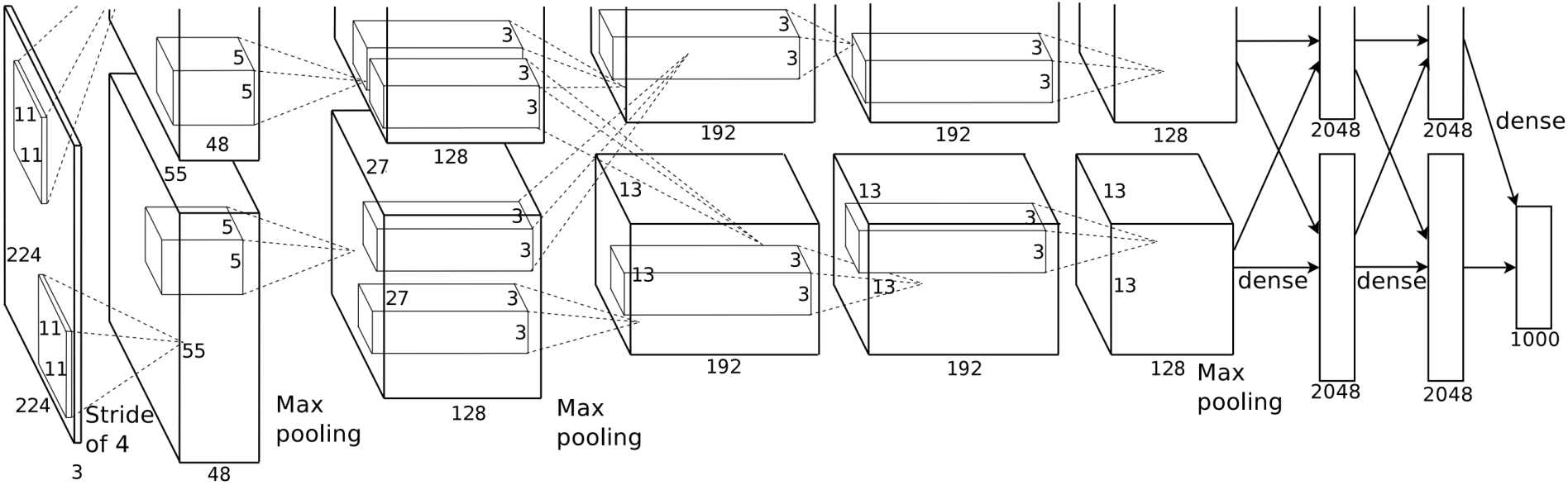
Adaptive Neuron Apoptosis for Accelerating Deep Learning on Large Scale Systems. Seigel et al. 2016.

LeNet Architecture
(first convolutional neural net; 1989)



Gradient-based learning applied to document recognition,
by Y. LeCun, L. Bottou, Y. Bengio and P. Haffner

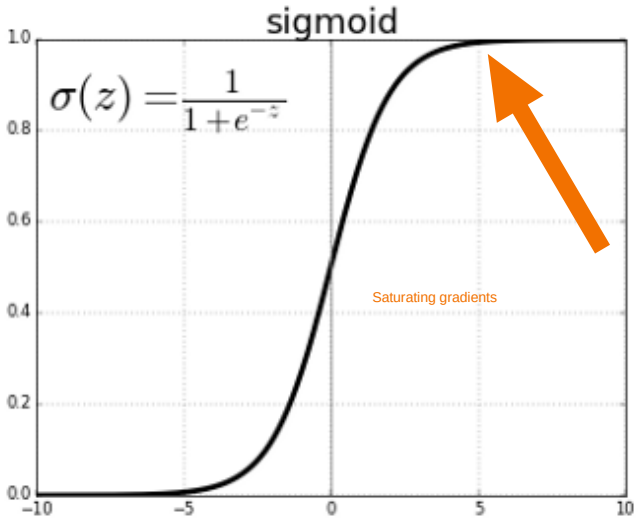
AlexNet



Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*.

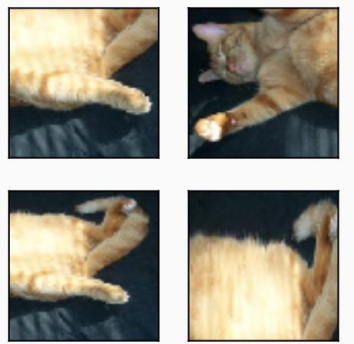
More Differences…

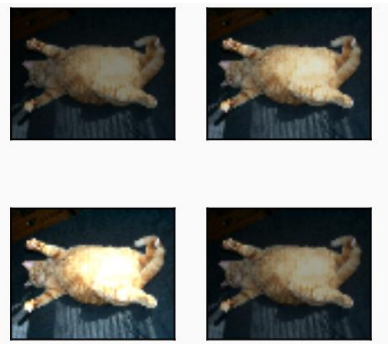
- Change activation function from sigmoid to ReLu (no more vanishing gradient)

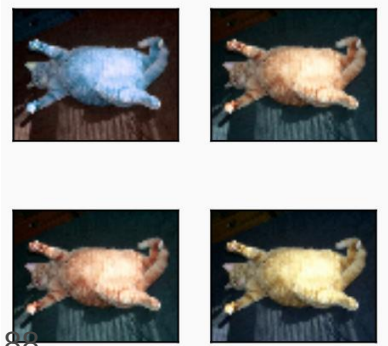


More Differences…

- Change activation function from sigmoid to ReLu (no more vanishing gradient)
- Data augmentation





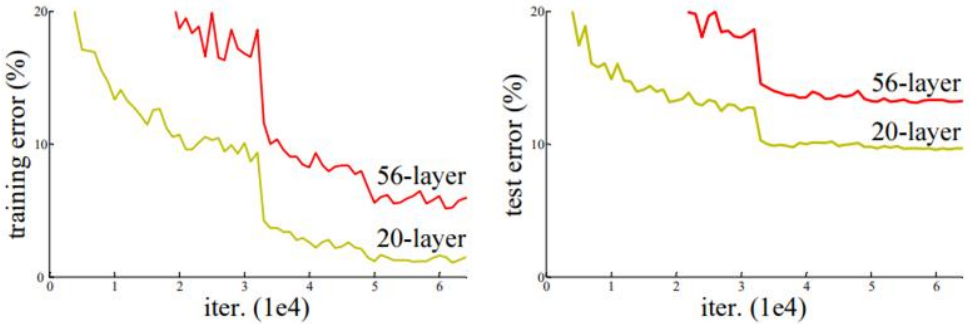


Simple Idea: Add More Layers

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:

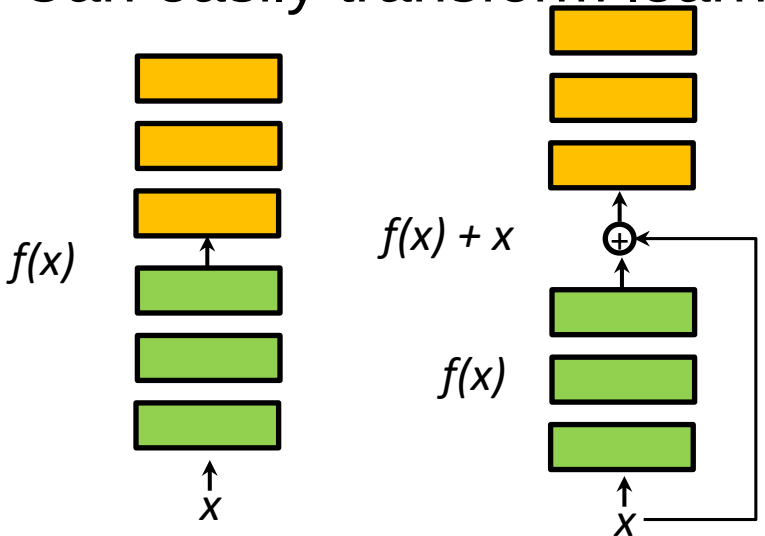


He et al: “Deep Residual Learning for Image Recognition”

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

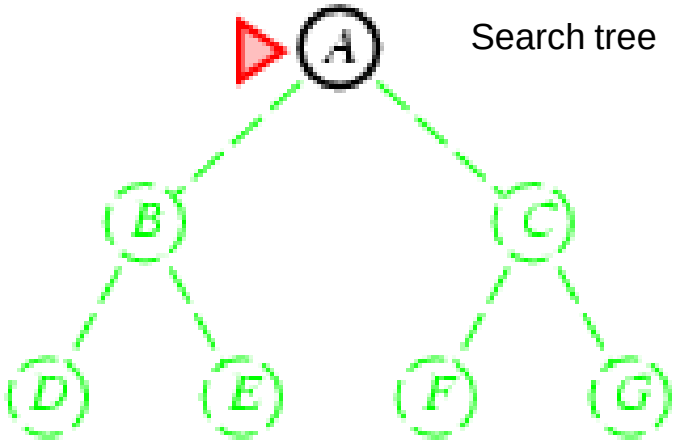
To learn identity *f(x) = x*, layers now need to learn *f(x) = 0* ➔ easier

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



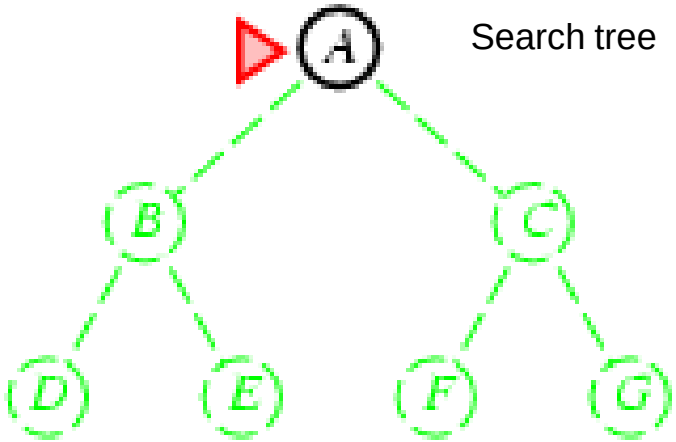
Search tree

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

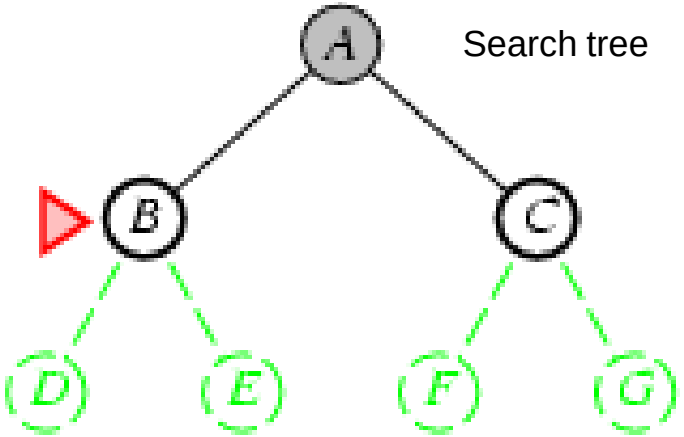
queue (fringe, OPEN)
→ [A] →

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

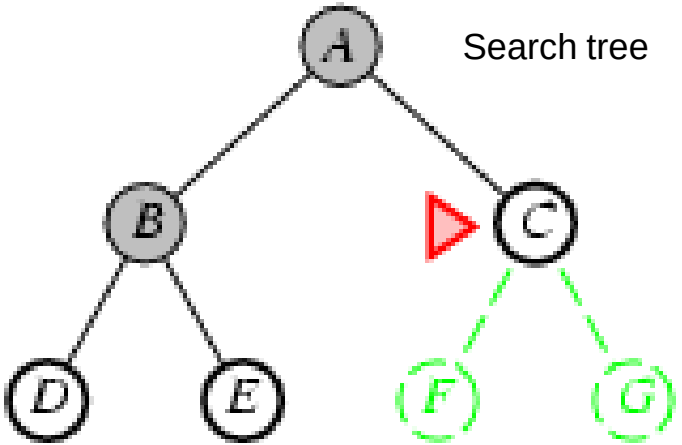
queue (fringe, OPEN)
→ [CB] → A

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

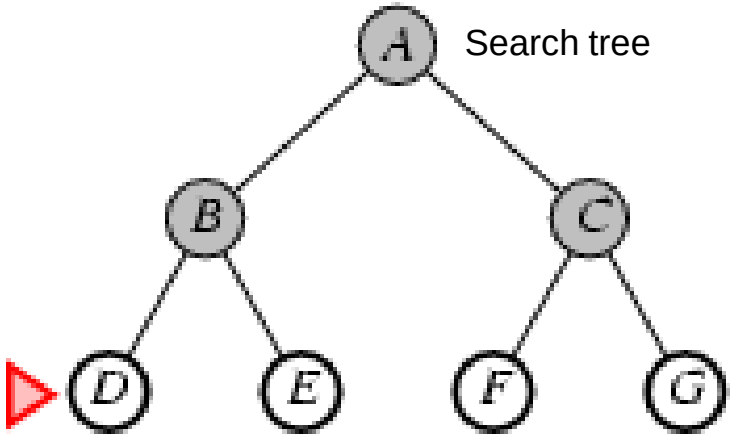
queue (fringe, OPEN)
→ [EDC] → B

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



Search tree

queue (fringe, OPEN)
[GFED] → C

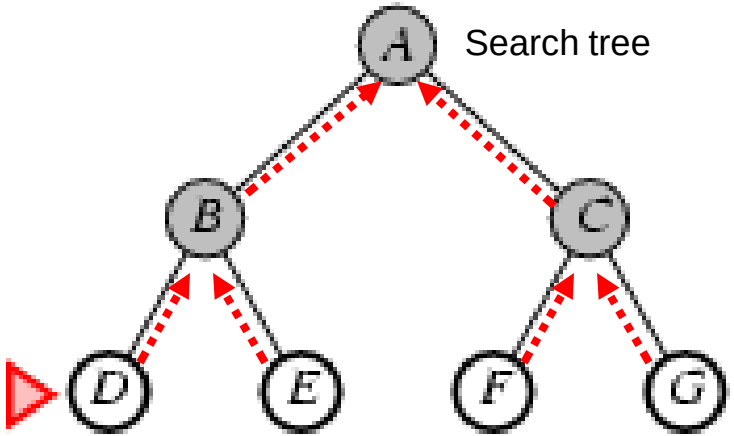
If G is a goal, we've seen it, but we don't stop!

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



Search tree

queue
□[] →G

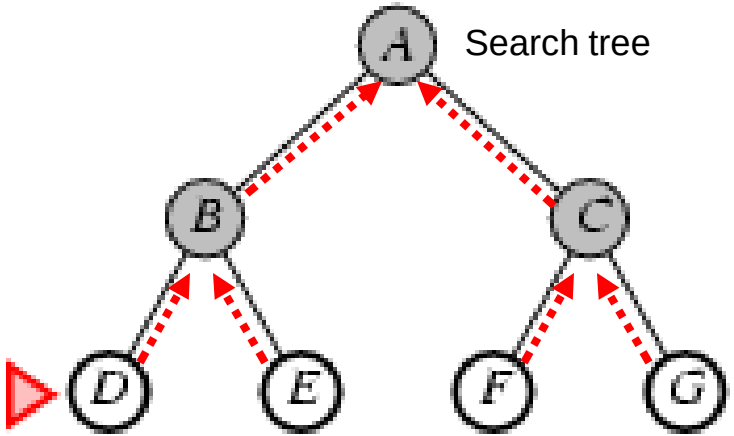
... until much later we pop G.

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



Search tree

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

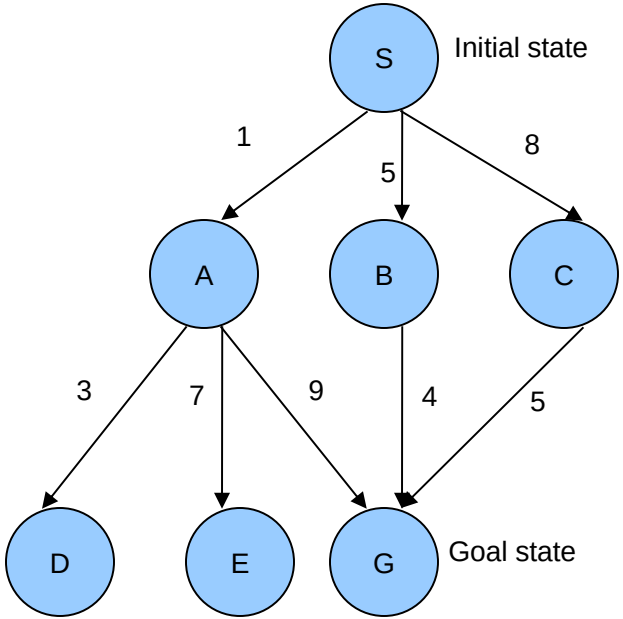
	Complete	optimal	time	space
Breadth-first search	Y	Y, if [1]	$O(b^d)$	$O(b^d)$

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

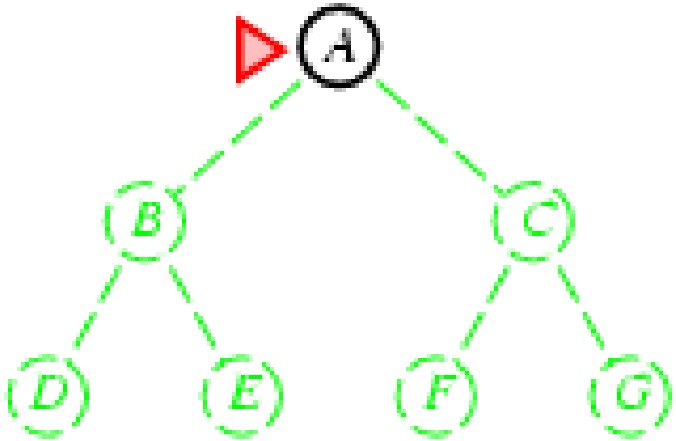
	Complete	optimal	time	space
Breadth-first search	Y	Y, if [1]	$O(b^d)$	$O(b^d)$
Uniform-cost search[2]	Y	Y	$O(b^{C^*/\varepsilon})$	$O(b^{C^*/\varepsilon})$

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

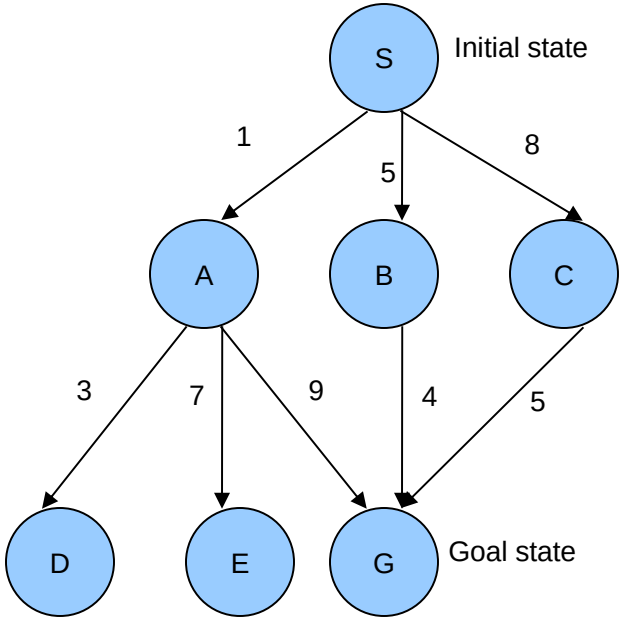
	Complete	optimal	time	space
Breadth-first search	Y	Y, if [1]	$O(b^d)$	$O(b^d)$
Uniform-cost search[2]	Y	Y	$O(b^{C^*/\varepsilon})$	$O(b^{C^*/\varepsilon})$
Depth-first search	N	N	$O(b^m)$	$O(bm)$

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

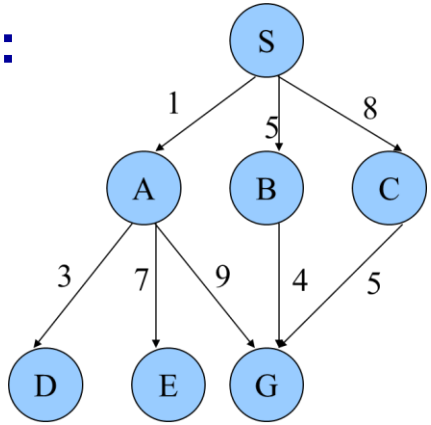
- Search proceeds like BFS, but fringe is like DFS
  - Complete, optimal like BFS
  - Small space complexity like DFS
  - Time complexity like BFS
- Preferred uninformed search method

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

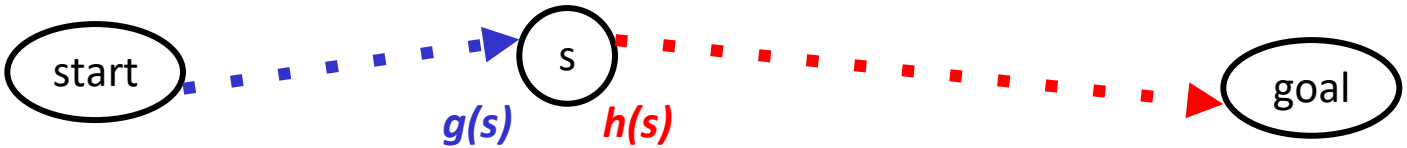
Uninformed vs Informed Search

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

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

Informed search. Know:

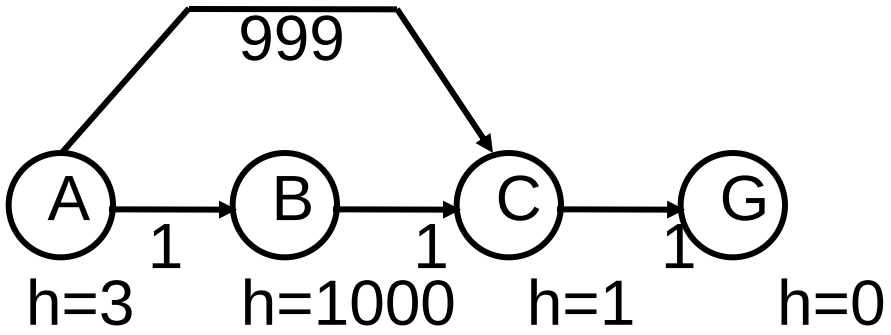
- All uninformed search properties, plus
- Heuristic **h(s)** from s to goal (recall game heuristic)



Attempt 2: **A Search**

Next approach: use both ***g*(*s*)** + ***h*(*s*)**

- Specifically, expand state with smallest ***g*(*s*)** + ***h*(*s*)**
- Again, use a priority queue
- Called **"A" search**



- **Still not optimal!**

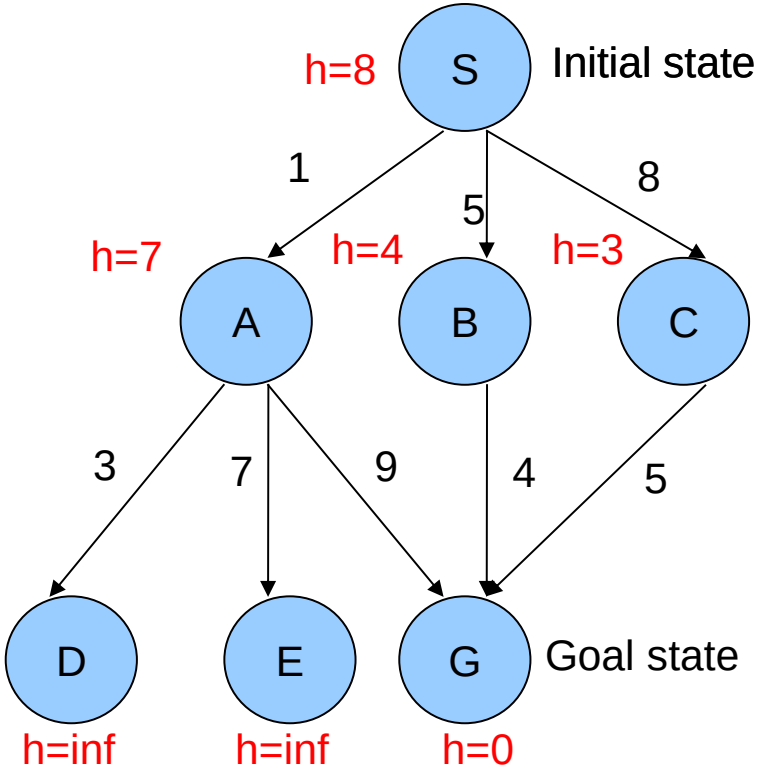
Attempt 3: **A* Search**

Same idea, use $g(s) + h(s)$, with one **requirement**

- Demand that $h(s) \leq h^*(s)$ where $h^*(s)$ is true cost from s to goal.
- If heuristic has this property, it is called "admissible"
  - Optimistic! Never over-estimates
- Still need $h(s) \geq 0$
  - Negative heuristics can lead to strange behavior
- This is **A* search**

Recap and Examples

Example for A*:

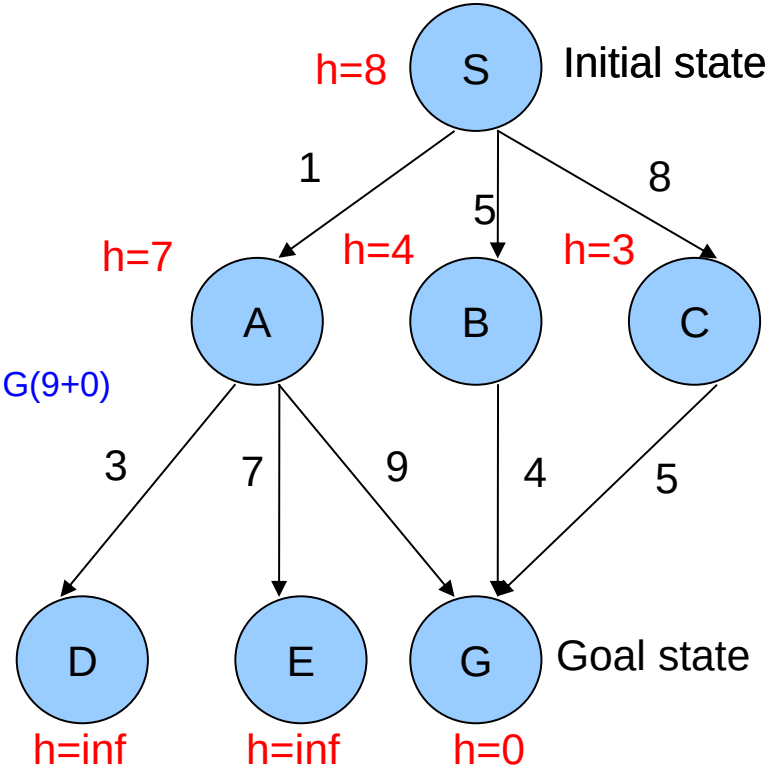


Recap and Examples

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 → B → S



Break & Quiz

Q 1.2: Which of the following are admissible heuristics?

i. $h(s) = h^*(s)$

ii. $h(s) = \max(2, h^*(s))$

iii. $h(s) = \min(2, h^*(s))$

iv. $h(s) = h^*(s)-2$

v. $h(s) = \text{sqrt}(h^*(s))$

- A. All of the above
- B. (i), (iii), (iv)
- C. (i), (iii)
- D. (i), (iii), (v)

Break & Quiz

Q 1.2: Which of the following are admissible heuristics?

i. $h(s) = h^*(s)$

ii. $h(s) = \max(2, h^*(s))$

iii. $h(s) = \min(2, h^*(s))$

iv. $h(s) = h^*(s)-2$

v. $h(s) = \text{sqrt}(h^*(s))$

- A. All of the above
- B. (i), (iii), (iv)
- **C. (i), (iii)**
- D. (i), (iii), (v)

Break & Quiz

Q 1.2: Which of the following are admissible heuristics?

i. $h(s) = h^*(s)$

ii. $h(s) = \max(2, h^*(s))$ No: $h(s)$ might be too big

iii. $h(s) = \min(2, h^*(s))$

iv. $h(s) = h^*(s)-2$ No: $h(s)$ might be negative

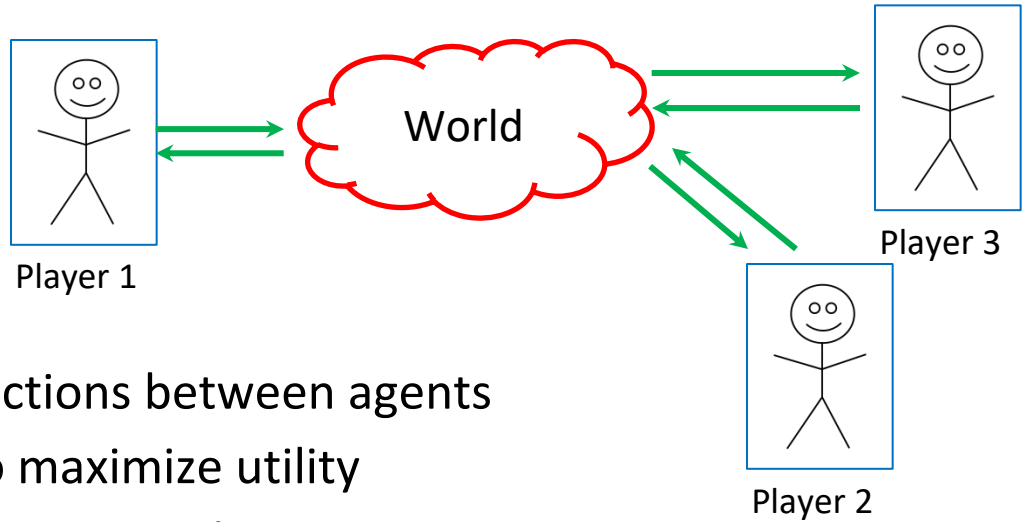
v. $h(s) = \text{sqrt}(h^*(s))$ No: if $h^*(s) < 1$ then $h(s)$ *is bigger*

- A. All of the above
- B. (i), (iii), (iv)
- **C. (i), (iii)**
- D. (i), (iii), (v)

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.

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

Break & Quiz

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: Lauch	(20, 20)	(40, 10)
A: Not Lauch	(10, 40)	(30, 30)

What is the strictly dominant strategy for each firm:

i. A's dominant strategy is to launch, and B's dominant strategy is not to launch.
ii. A's dominant strategy is to launch, and B's dominant strategy is to launch.
iii. A's dominant strategy is not to launch, and B's dominant strategy is to launch.
iv. A's dominant strategy is not to launch, and B's dominant strategy is not to launch.

Break & Quiz

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: Lauch	(20, 20)	(40, 10)
A: Not Lauch	(10, 40)	(30, 30)

What is the strictly dominant strategy for each firm:

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

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

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

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

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**)

- 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.

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

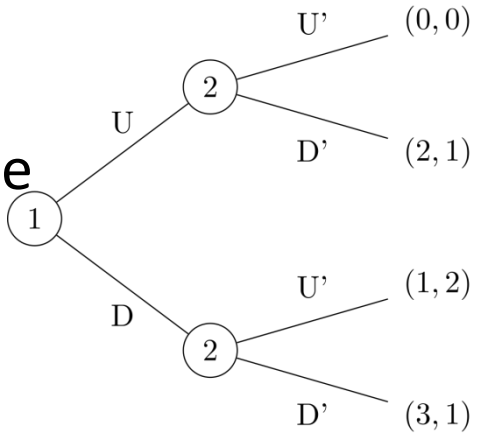
Example: $x_1^*(\cdot) = x_2^*(\cdot) = \left(\frac{1}{3}, \frac{1}{3}, \frac{1}{3}\right)$

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



Wiki

- Nash equilibrium still well-defined
  - Backward induction

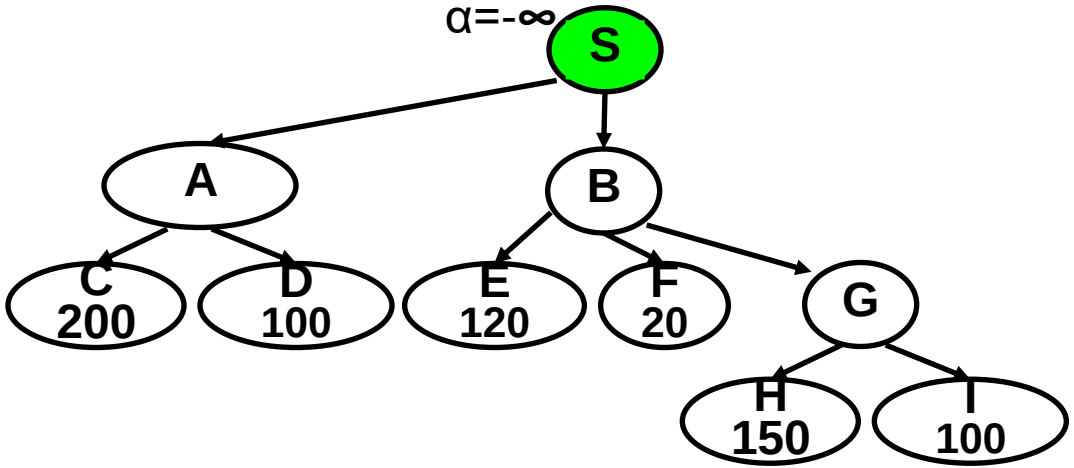
Minimax algorithm in execution

max

min

max

min



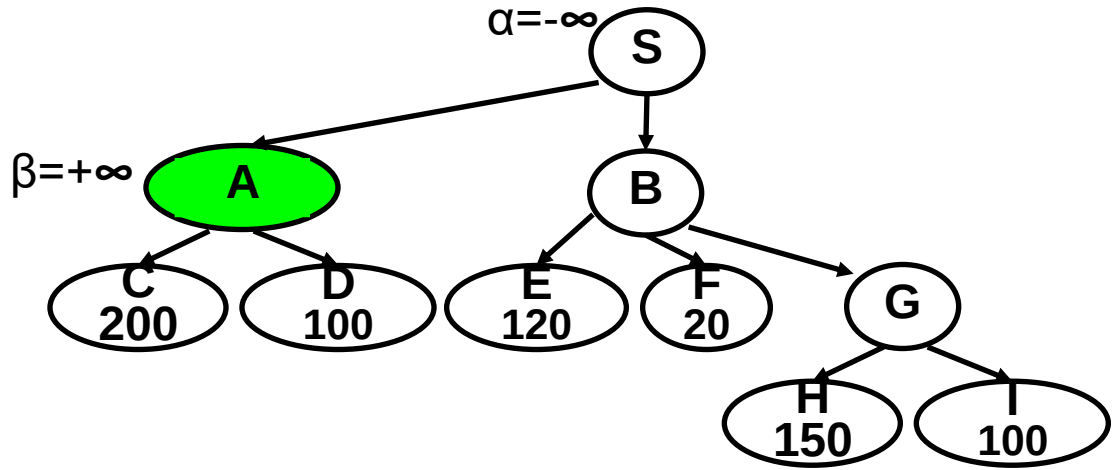
Minimax algorithm in execution

max

min

max

min



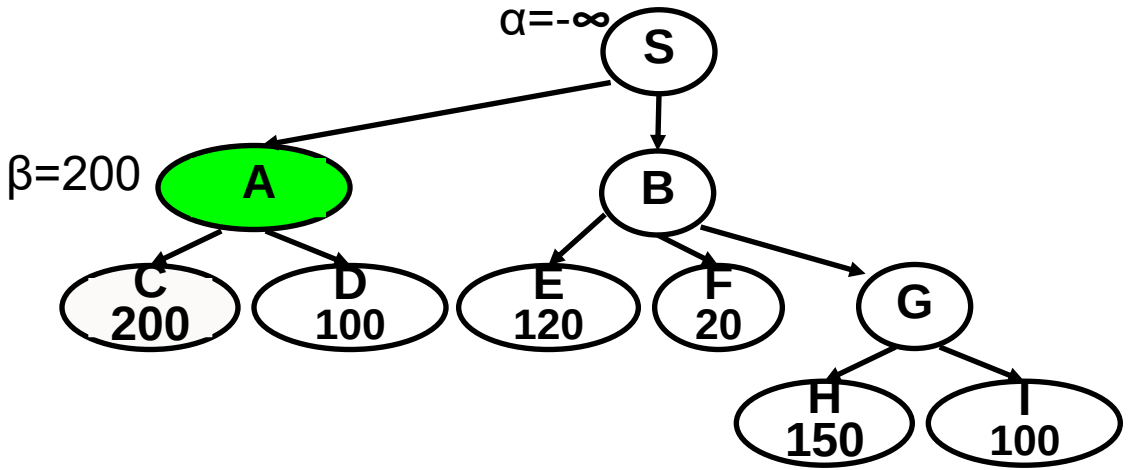
Minimax algorithm in execution

max

min

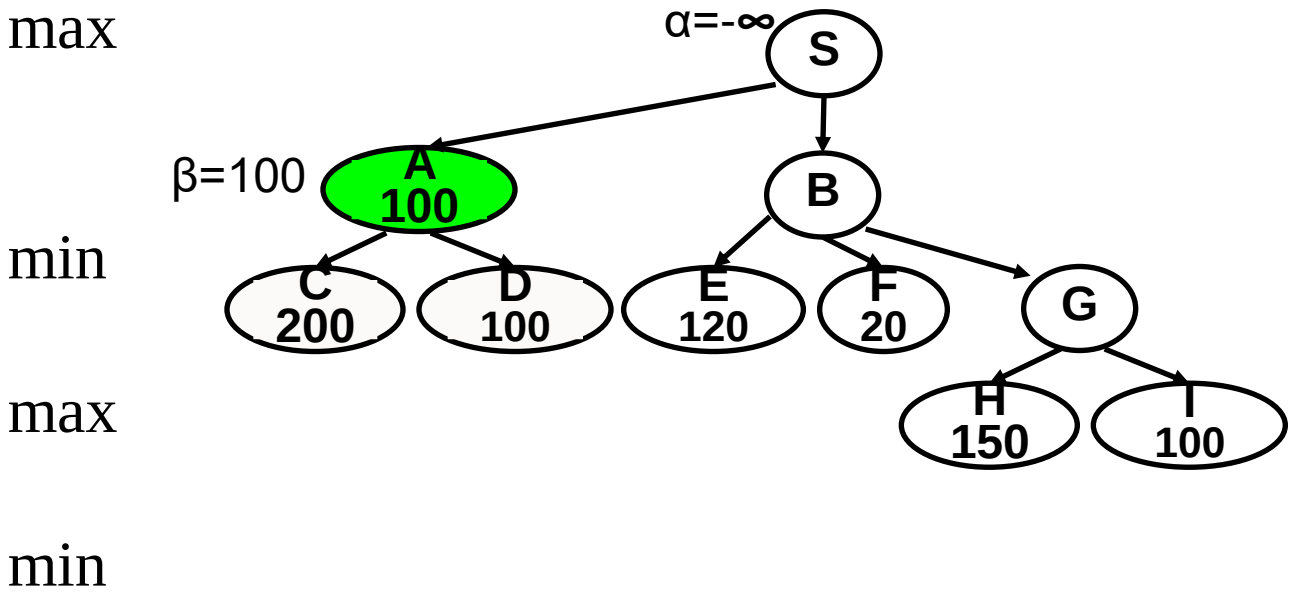
max

min



The execution on the terminal nodes is omitted.

Minimax algorithm in execution



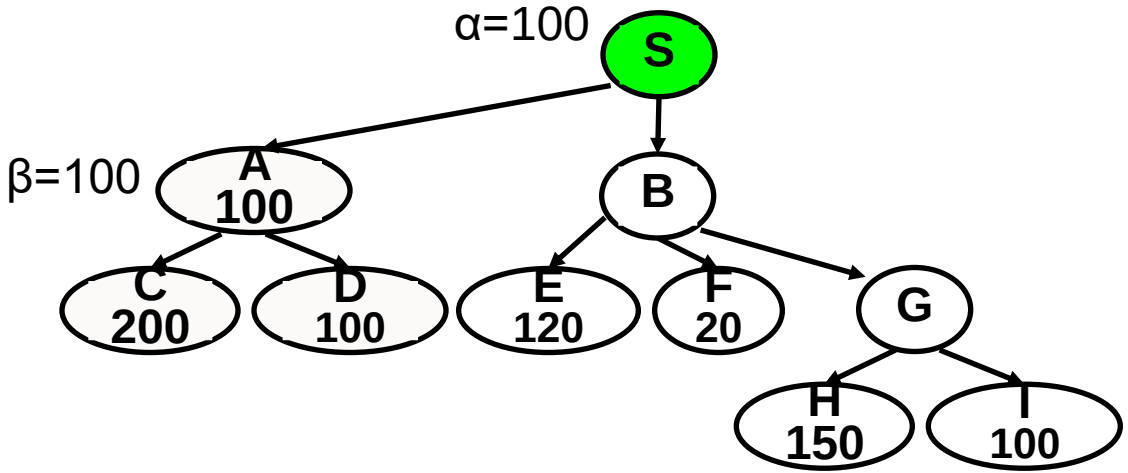
Minimax algorithm in execution

max

min

max

min



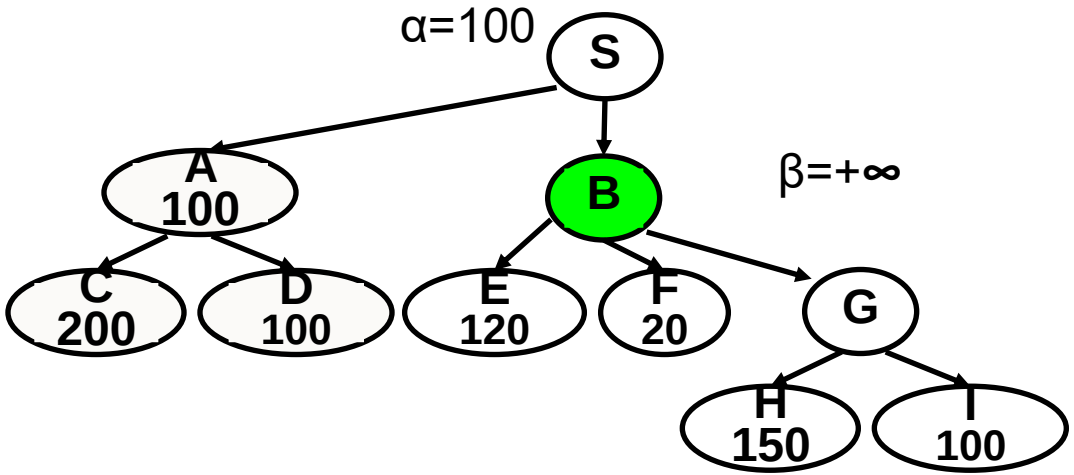
Minimax algorithm in execution

max

min

max

min



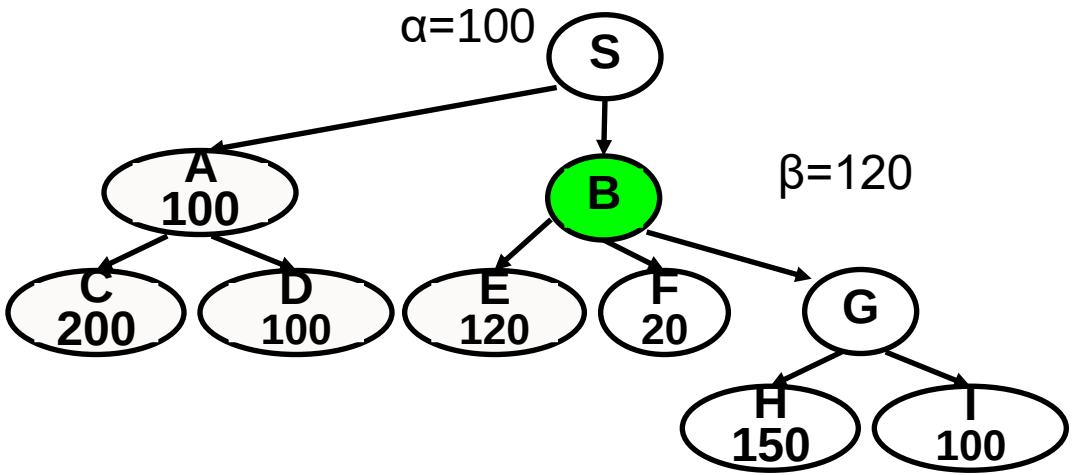
Minimax algorithm in execution

max

min

max

min



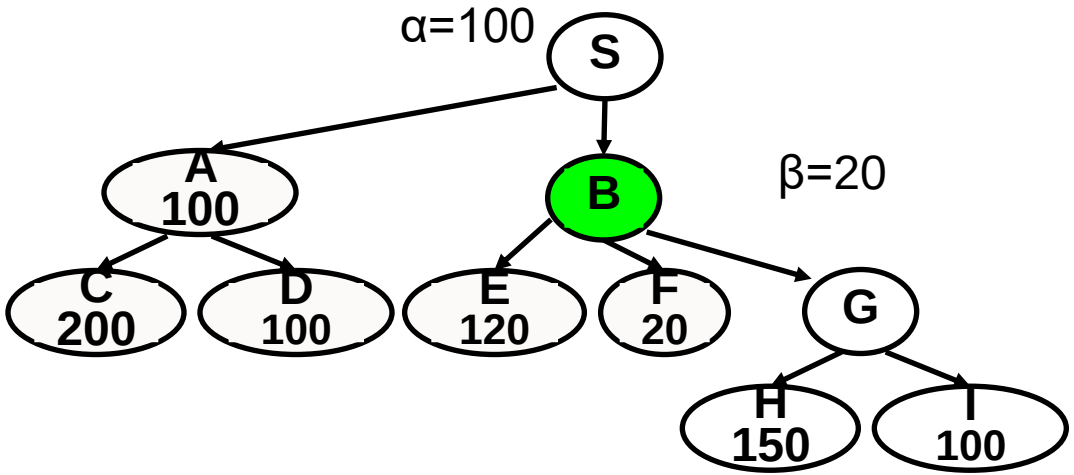
Minimax algorithm in execution

max

min

max

min



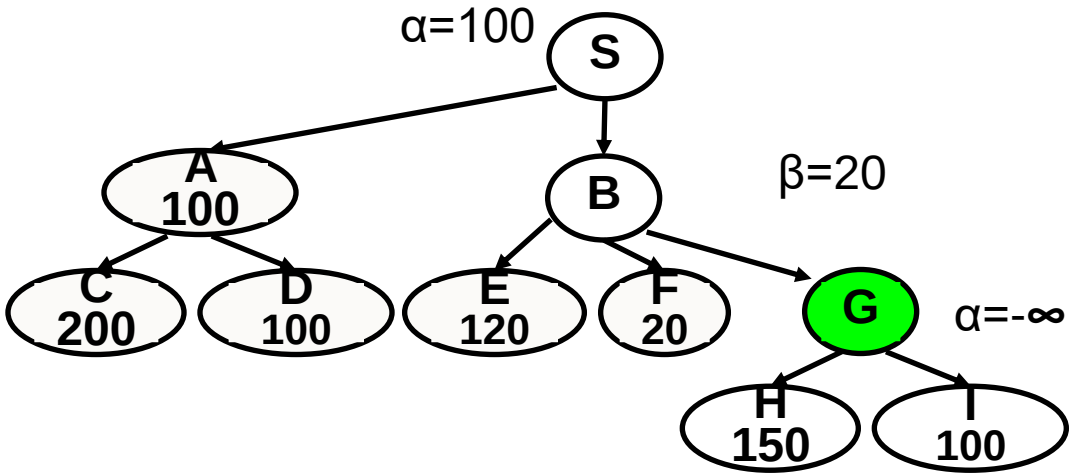
Minimax algorithm in execution

max

min

max

min



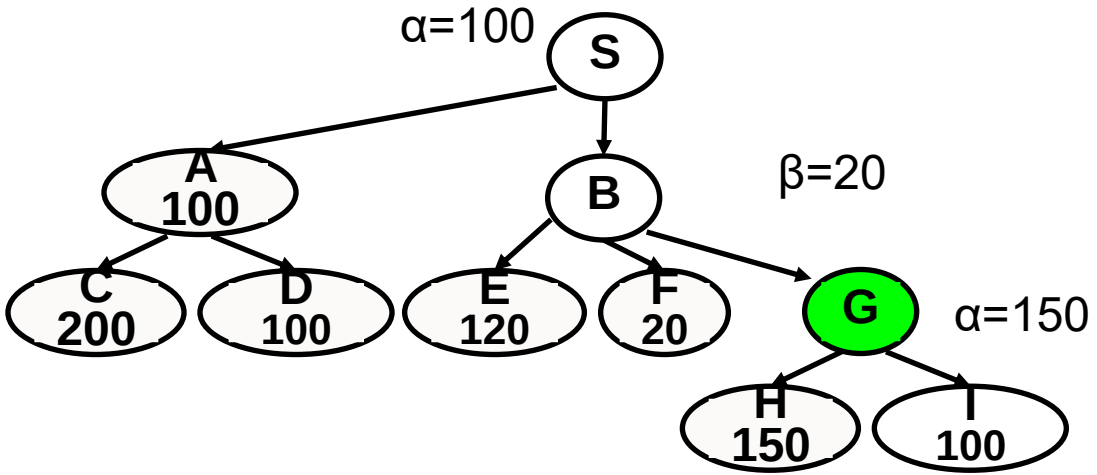
Minimax algorithm in execution

max

min

max

min



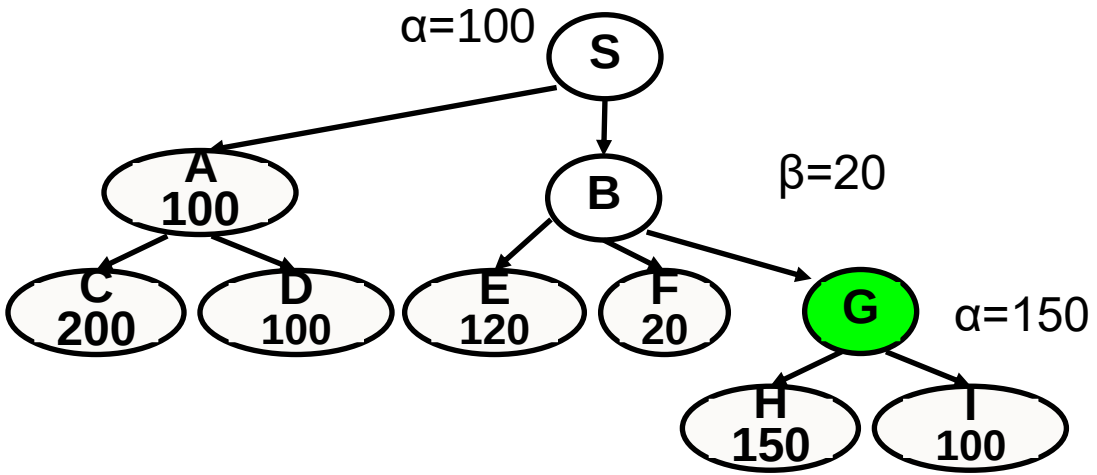
Minimax algorithm in execution

max

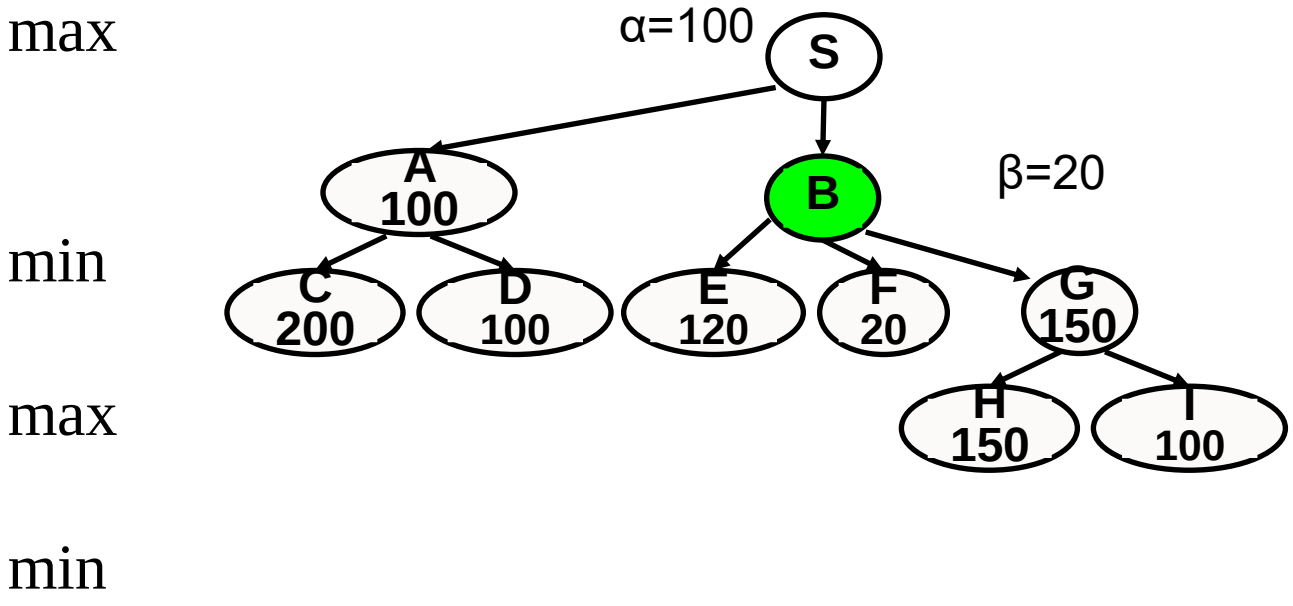
min

max

min



Minimax algorithm in execution



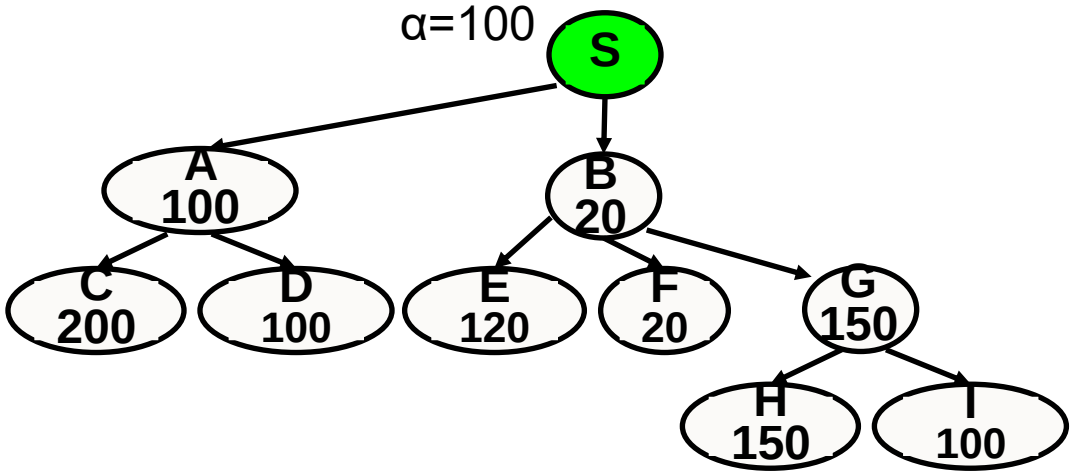
Minimax algorithm in execution

max

min

max

min



Break & Quiz

Q 2.1: We are playing a game where Player A goes first and has 4 moves. Player B goes next and has 3 moves. Player A goes next and has 2 moves. Player B then has one move.

How many nodes are there in the minimax tree, including termination nodes (leaves)?

- A. 23
- B. 65
- C. 41
- D. 2

Break & Quiz

Q 2.1: We are playing a game where Player A goes first and has 4 moves. Player B goes next and has 3 moves. Player A goes next and has 2 moves. Player B then has one move.

How many nodes are there in the minimax tree, including termination nodes (leaves)?

- A. 23
- **B. 65**
- C. 41
- D. 2

Break & Quiz

Q 2.1: We are playing a game where Player A goes first and has 4 moves. Player B goes next and has 3 moves. Player A goes next and has 2 moves. Player B then has one move.

How many nodes are there in the minimax tree, including termination nodes (leaves)?

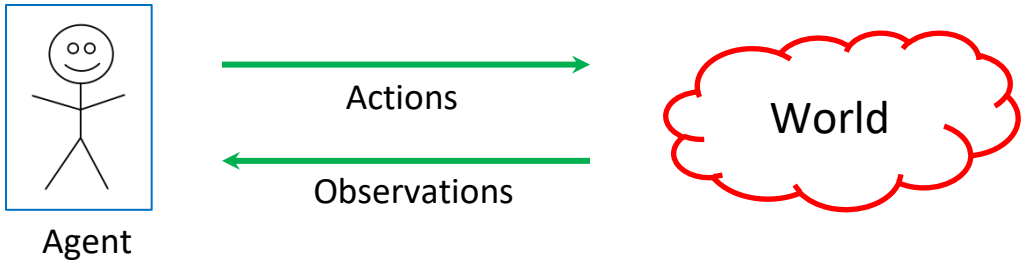
- A. 23
- **B. 65** (1 + 4 + 4*3 + 4*3*2 + 4*3*2 = 65. Note the root and leaf nodes.)
- C. 41
- D. 2

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.

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

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

Obtaining the Optimal Policy

Assume, we know the expected utility of an action.

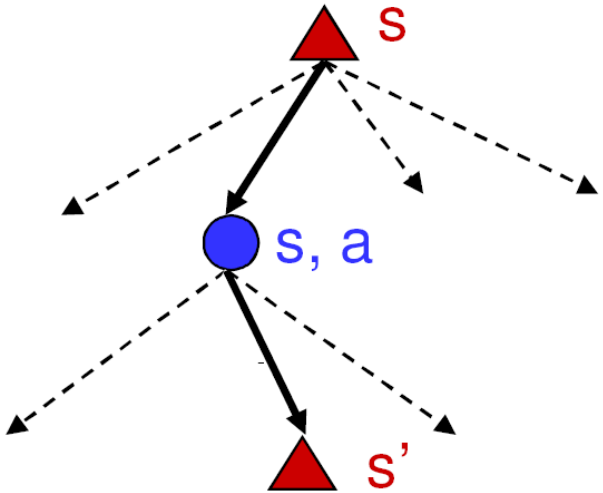
- So, to get the optimal policy, compute

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

All the states we could go to

Transition probability

Expected rewards

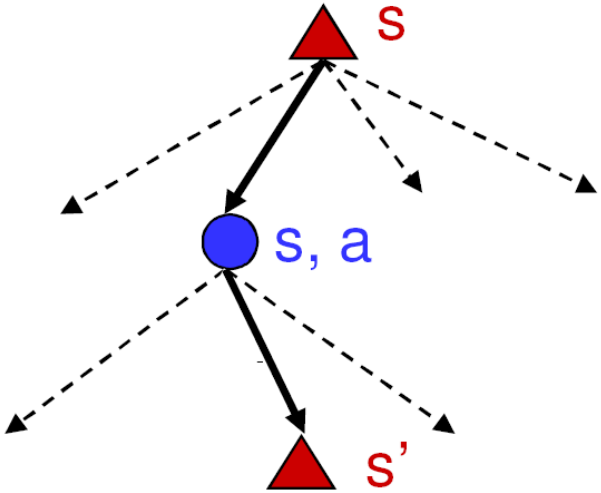


Credit L. Lazbenik

Bellman Equations

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

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



Credit L. Lazbenik

Current state reward

Discounted expected future **rewards**

Richard Bellman: Inventor of dynamic programming.

Break & Quiz

Q 2.1 Consider an MDP with 2 states {*A, B*} and 2 actions: **"stay"** at current state and "**move**" to other state. Let **r** be the reward function such that **r**(*A*) = 1, **r**(*B*) = 0. Let γ be the discounting factor. Let π: π(*A*) = π(*B*) = **move** (i.e., an "always move" policy). What is the value function $V^{\pi}(A)$?

- A. 0
- B. 1 / (1 -γ)
- C. 1 / (1 -γ^2)
- D. 1

Break & Quiz

Q 2.1 Consider an MDP with 2 states {*A, B*} and 2 actions: **"stay"** at current state and "**move**" to other state. Let **r** be the reward function such that **r**(*A*) = 1, **r**(*B*) = 0. Let γ be the discounting factor. Let π: π(*A*) = π(*B*) = **move** (i.e., an "always move" policy). What is the value function $V^{\pi}(A)$?

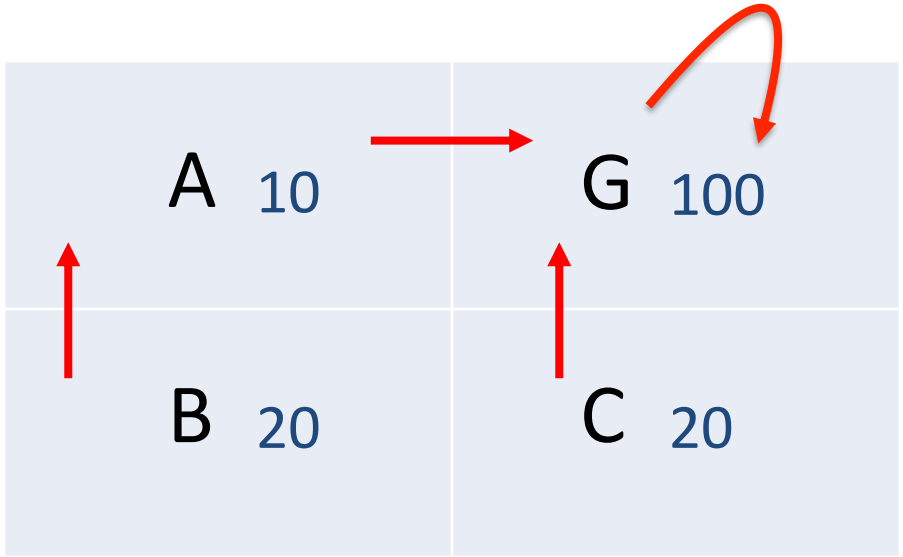
- A. 0
- B. $1/(1-\gamma)$
- **C. $1/(1-\gamma^2)$**
- D. 1

Break & Quiz

Q 2.1 Consider an MDP with 2 states {*A, B*} and 2 actions: **"stay"** at current state and "**move**" to other state. Let **r** be the reward function such that **r**(*A*) = 1, **r**(*B*) = 0. Let γ be the discounting factor. Let π: π(*A*) = π(*B*) = **move** (i.e., an "always move" policy). What is the value function $V^{\pi}(A)$?

- A. 0
- B. 1/(1-γ)
- **C. 1/(1-γ^2)** **(**States: A,B,A,B,… rewards 1,0, γ^2,0, γ^4,0, …)
- D. 1

Example



Deterministic transitions; $\gamma = 0.8$; policy shown with red arrows.

Break & Quiz

Supposed you have the following information about an environment:

1. The discount factor is 0.8
2. The reward in *s1* taking action *α1* is 3
3. The transition probabilities are: *P(s2|s1, α1)* = 0.6 and *P(s3|s1, α1)* = 0.4

Currently, *V(s2)* = 10 and *V(s3)* = 6

Remember the update for value iteration is $V_{i+1}(s) = r(s) + \gamma \max_a \sum_{s'} P(s'|s,a) V_i(s')$

After a single iteration of value iteration, what is the value for state s1 (what is V (s1))?

A. 8

B. 10

C. 12

D. 14

E. None of the above

Break & Quiz

Supposed you have the following information about an environment:

1. The discount factor is 0.8
2. The reward in *s1* taking action *α1* is 3
3. The transition probabilities are: *P(s2|s1, α1)* = 0.6 and *P(s3|s1, α1)* = 0.4

Currently, *V(s2)* = 10 and *V(s3)* = 6

Remember the update for value iteration is $V_{i+1}(s) = r(s) + \gamma \max_a \sum_{s'} P(s'|s,a) V_i(s')$

After a single iteration of value iteration, what is the value for state s1 (what is V(s1))? Choose the closest option.

A. 8

B. 10 **V (s1) = 3 + 0.8(0.6 × 10 + 0.4 × 6) = 9.72 = 10**

C. 12

D. 14

E. None of the above

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: $V^*(s) = \underset{a}{max} Q^*(s, a)$.
- Optimal policy is formed as $\pi^*(s) = \arg\underset{a}{max} Q^*(s, a)$

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

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.

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

Thank you and good luck!