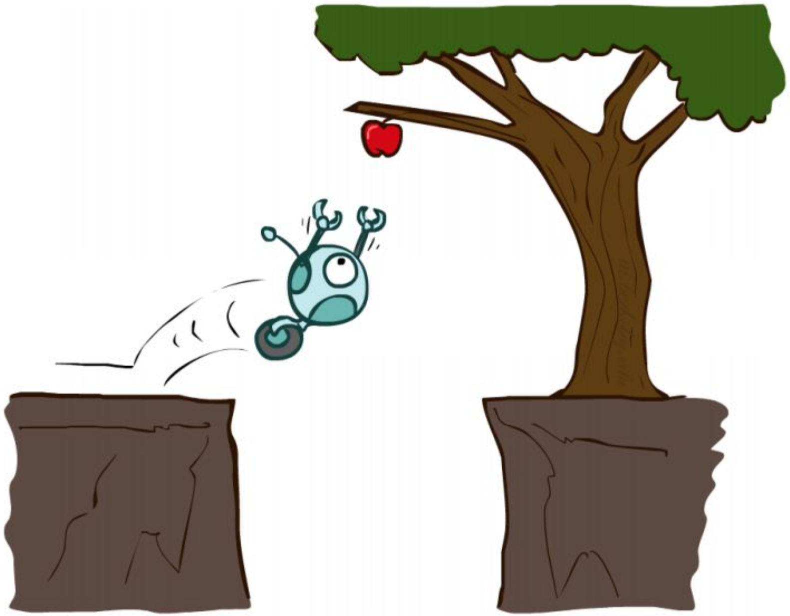
Introduction to Artificial Intelligence

By Adel Setoodehnia and Kevin Chant

graphics and examples from Berkeley CS188 Fa ‘16 and Sp ‘18

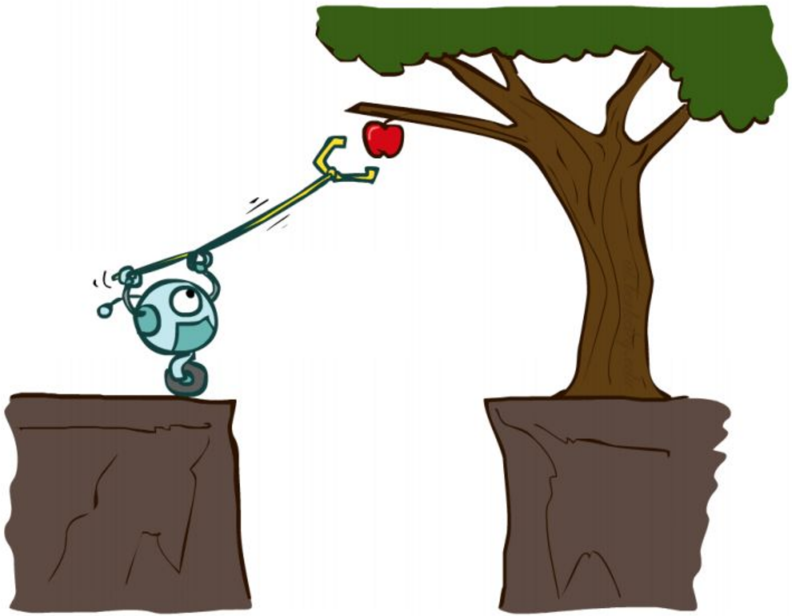
Search Agents

Reflex agents



- Chooses action based **solely** on current state of the world
- Doesn't think about the consequences of its actions
- Typically outperformed by **planning agents**

Planning agents

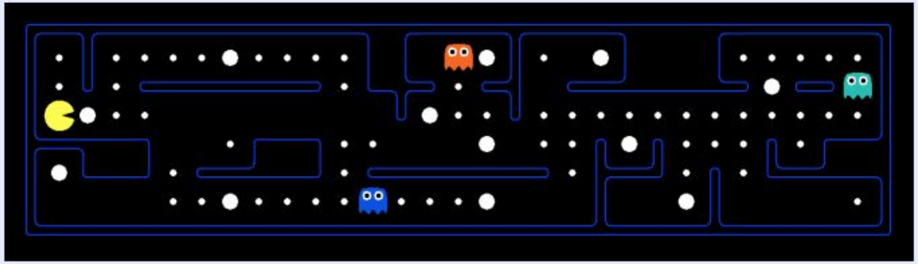


- Asks “what if?”
- Decisions based on (hypothesized) consequences of actions
- Must have a model of how the world evolves in response to actions

State Spaces and Search Problems

- A **search problem** consists of:
  - A **state space**
  - A **successor function**
  - A **start state** and a **goal test**
- A **solution** is a sequence of actions which takes you from a start state to a goal state

The world state includes every last detail of the environment

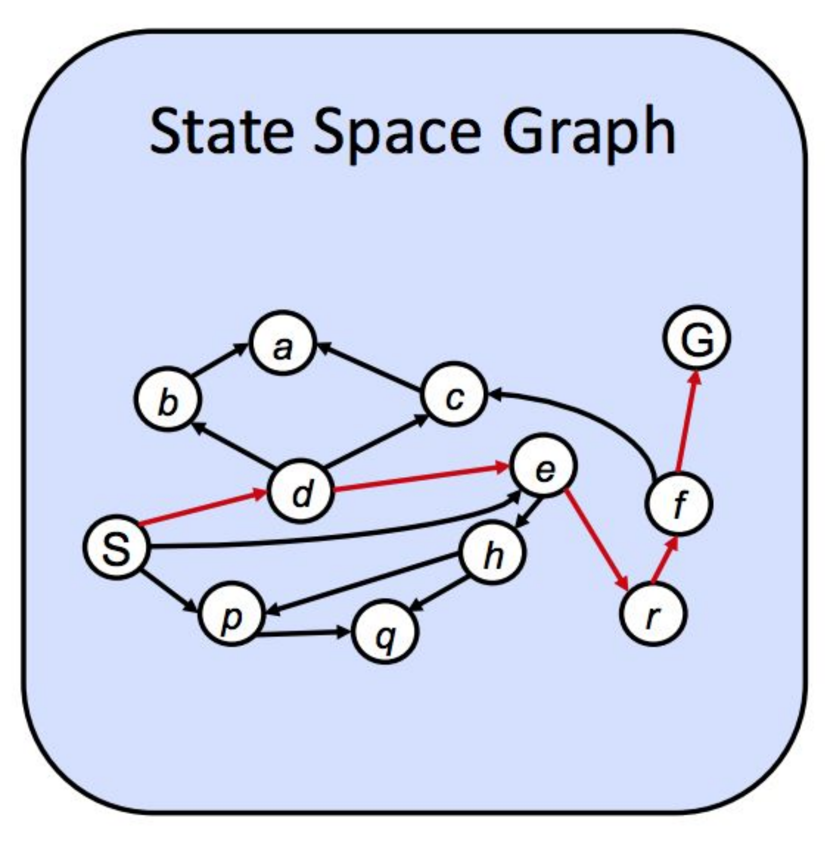


A search state keeps only the details needed for planning (abstraction)

- Problem: Pathing
  - States: (x,y) location
  - Actions: NSEW
  - Successor: update location only
  - Goal test: is (x,y)=END

- Problem: Eat-All-Dots
  - States: {(x,y), dot booleans}
  - Actions: NSEW
  - Successor: update location and possibly a dot boolean
  - Goal test: dots all false

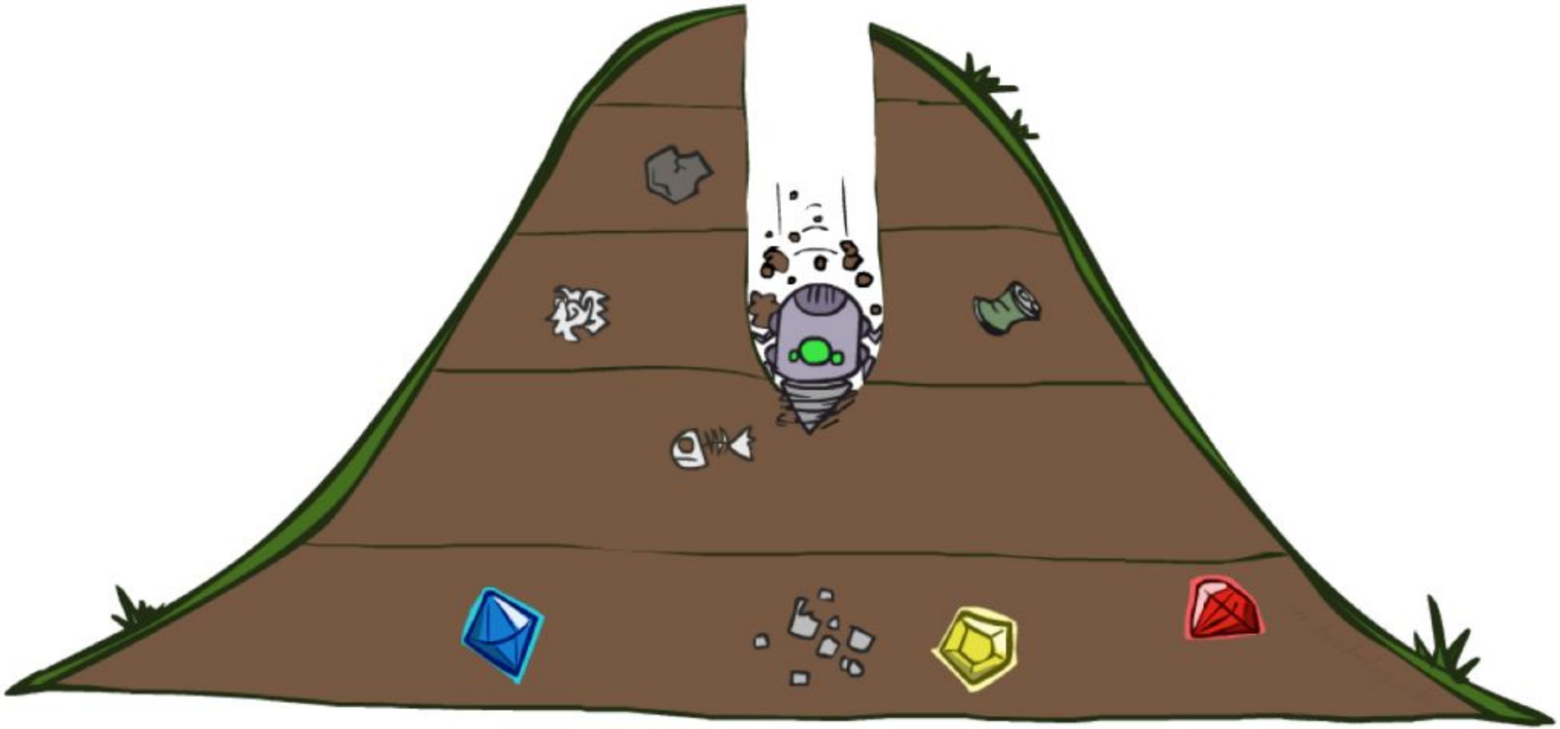
State Space Graph



Graph Search Algorithm

```
function GRAPH-SEARCH(problem, fringe) return a solution, or failure
  closed ← an empty set
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
    if fringe is empty then return failure
    node ← REMOVE-FRONT(fringe)
    if GOAL-TEST(problem, STATE[node]) then return node
    if STATE[node] is not in closed then
      add STATE[node] to closed
      for child-node in EXPAND(STATE[node], problem) do
        fringe ← INSERT(child-node, fringe)
      end
  end
```

Depth-First Search



Depth-First Search

- DFS is a strategy for exploration that selects the *deepest* fringe node from the start node for expansion
- Uses a Stack for the fringe representation

Breadth-First Search

- BFS is a strategy for exploration that selects the *shallowest* fringe node from the start node for expansion
- Uses a Queue for the fringe representation

Uniform-Cost Search

- UCS is a strategy for exploration that selects the *lowest cost* fringe node from the start node for expansion
- Uses a Priority Queue for the fringe representation

A* Search

- A* Search is a strategy for exploration that selects the fringe node with the *lowest estimated total cost* for expansion
- Uses a Priority Queue for the fringe representation just like UCS
- Must have an admissible heuristic for A* Tree Search
- Must have a consistent heuristic for A* Graph Search

Demo Time!

Coding Time!

Game Trees: Minimax and Expectimax

Adversarial Search Problems

- So now... what happens to our standard search problems, like path-finding or Pac-Man when we start to introduce adversaries?
  - Ghosts
  - Random accidents on the freeway blocking your way to work
- We come up with something new: **adversarial search problems**, more commonly known as **games**

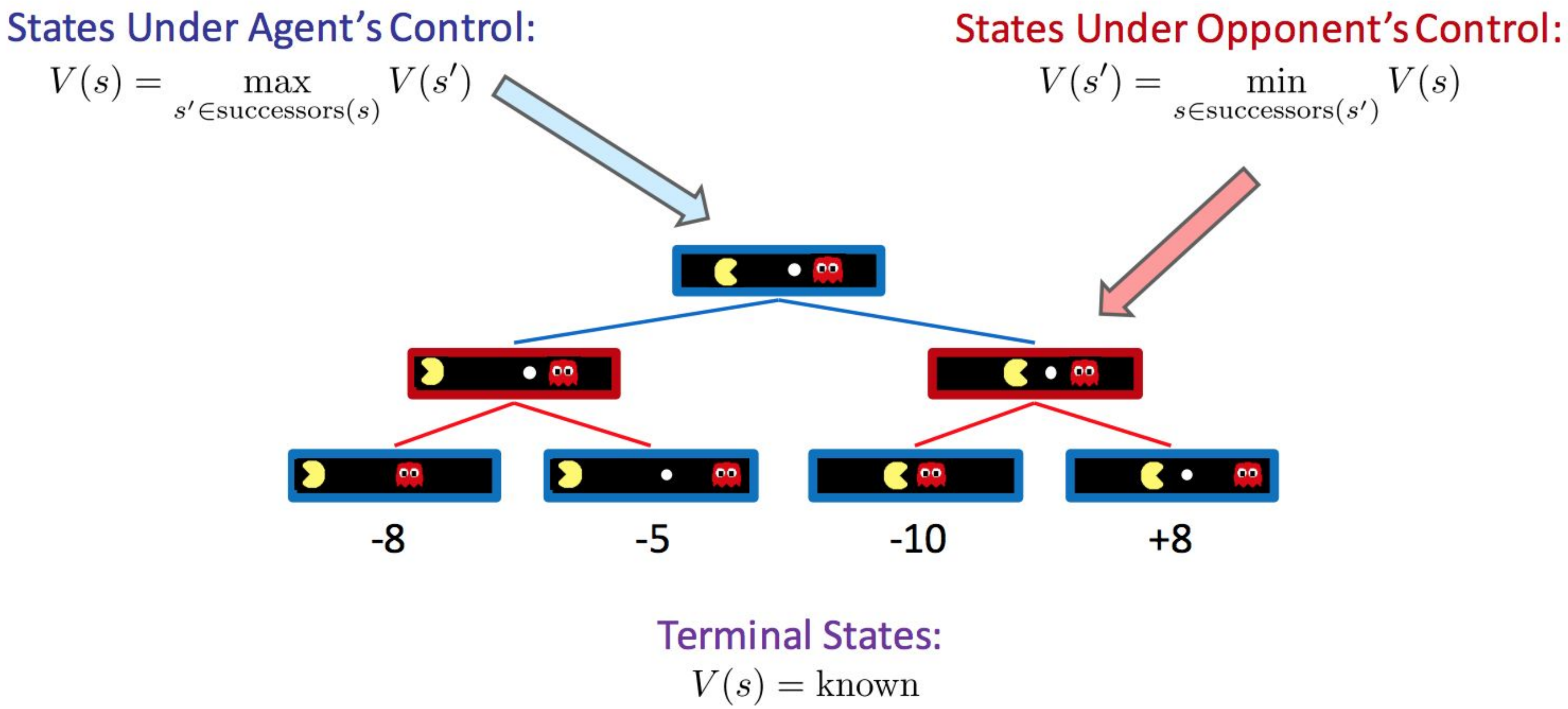
Games

- Chess - 1997, Deep Blue became the first computer agent to defeat human chess champion Gary Kasparaov in a six-game match!
- Go - AlphaGo, developed by Google, historically defeated Go champion Lee Sodol 4 games to 1 in March 2016

Zero-Sum Games

- Agents have opposite utilities
- Lets us think of a single value that one maximizes and the other minimizes
- Other General Games can be more complicated and less clear-cut
    - Cooperation, indifference, etc.

Minimax Algorithm



Minimax Algorithm Pseudocode

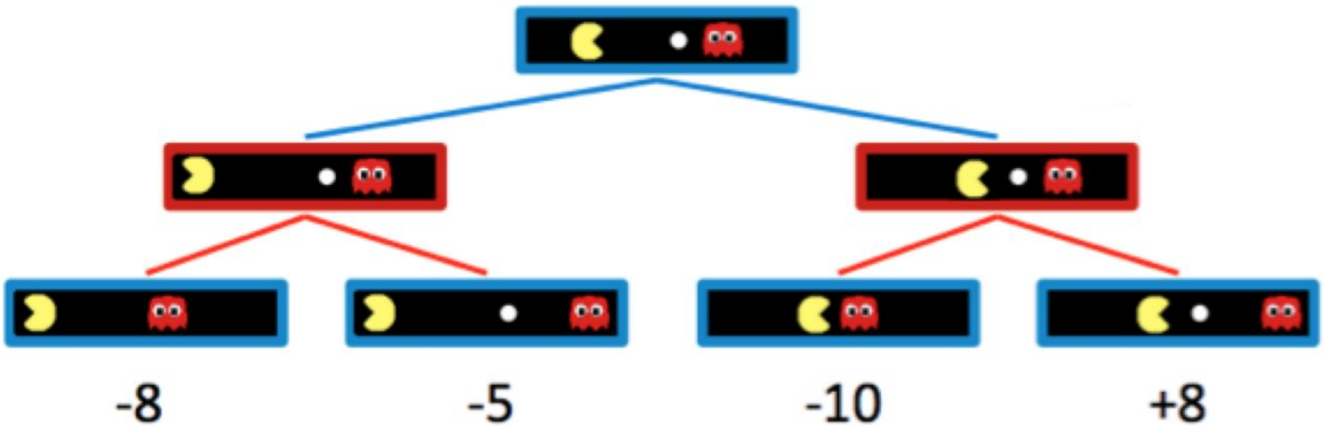
```
def value(state):
    if the state is a terminal state: return the state's utility
    if the next agent is MAX: return max-value(state)
    if the next agent is MIN: return min-value(state)
```

```
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v
```

```
def min-value(state):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor))
    return v
```

Expectimax Algorithm

- Now imagine that our adversaries do not always behave as optimally as they ought to… can we do better given we know this?



Expectimax Algorithm Pseudocode

```
def value(state):
    if the state is a terminal state: return the state's utility
    if the next agent is MAX: return max-value(state)
    if the next agent is EXP: return exp-value(state)
```

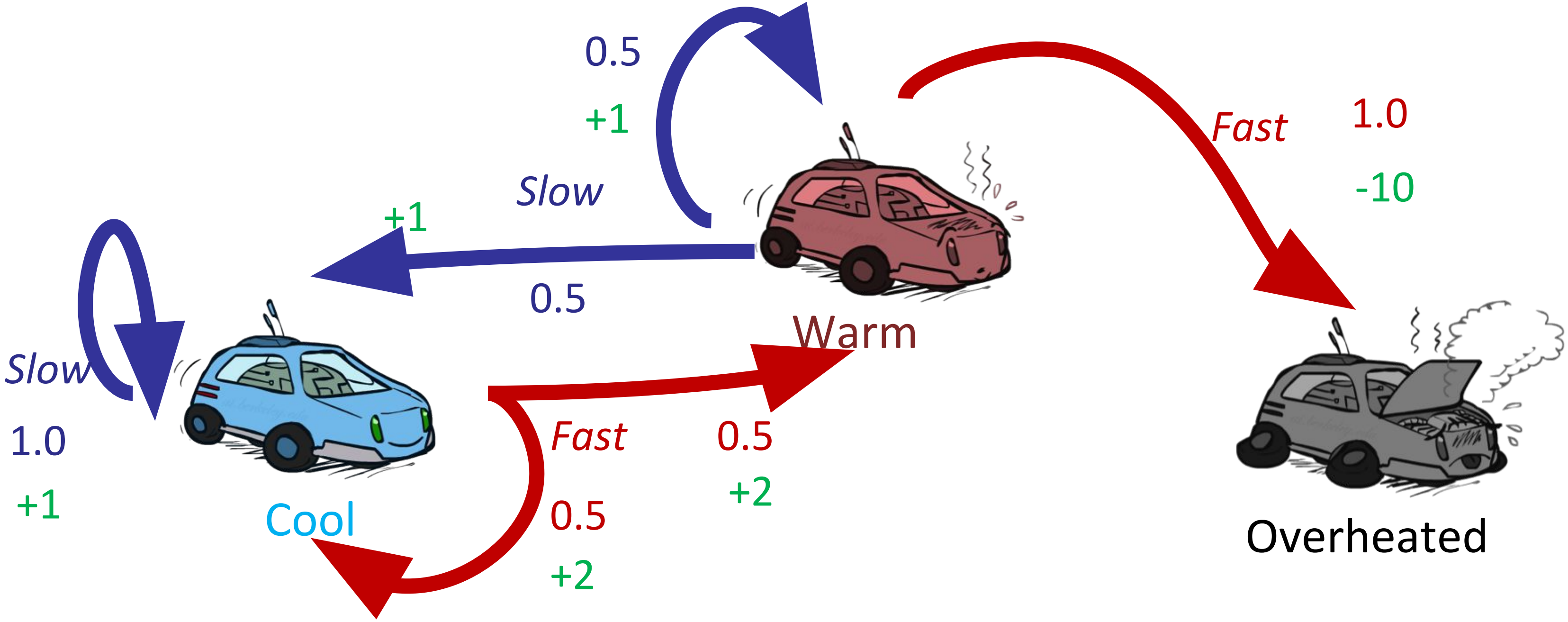
```
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v
```

```
def exp-value(state):
    initialize v = 0
    for each successor of state:
        p = probability(successor)
        v += p * value(successor)
    return v
```

Coding Time!

Reinforcement Learning

Markov Decision Processes

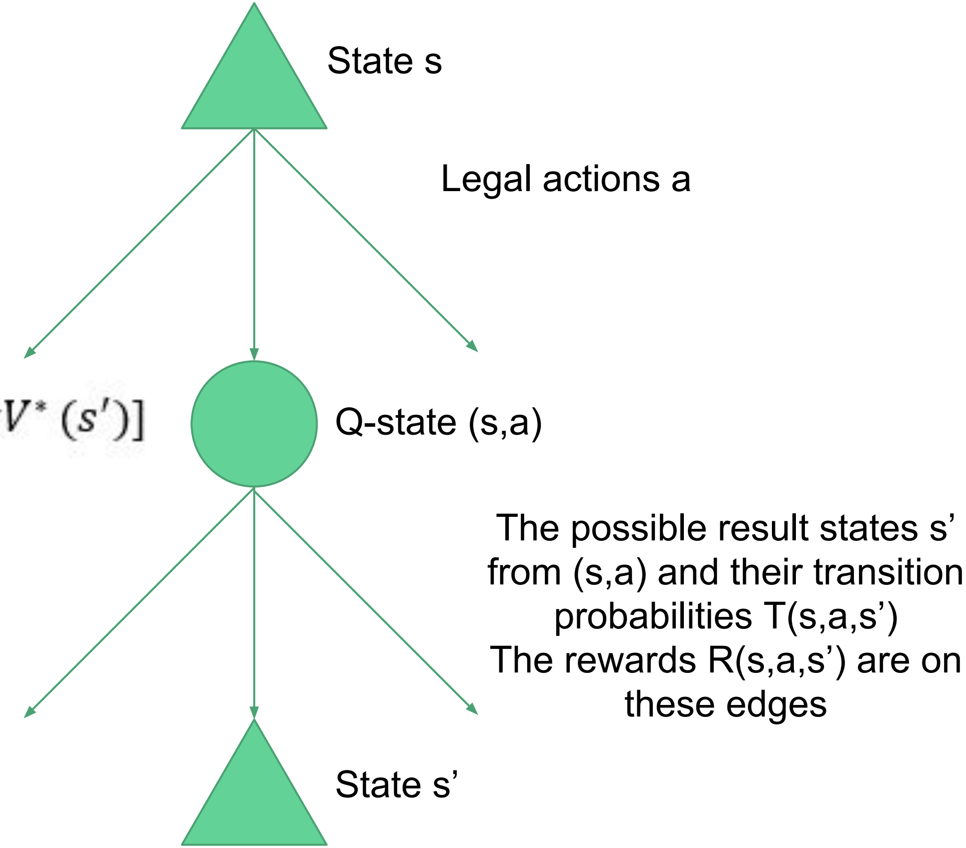


The math behind an MDP

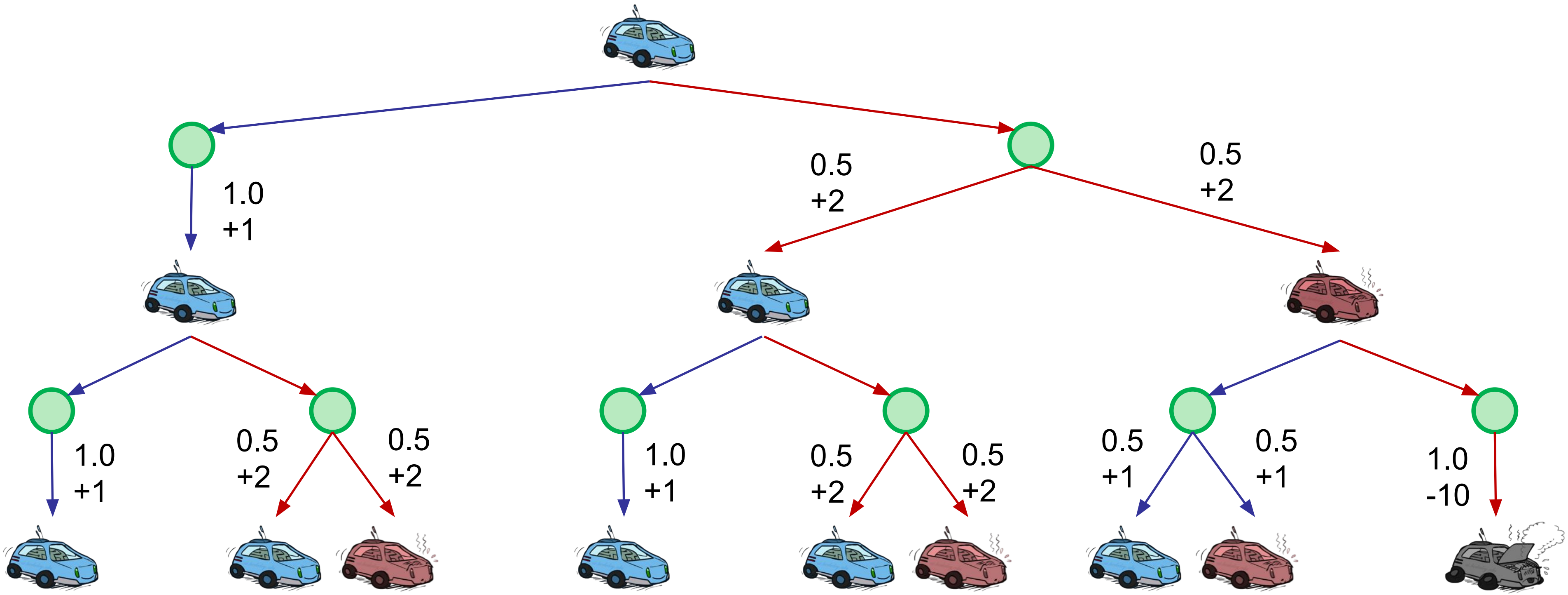
- Definitions
  - Q: The expected utility gained after taking an action and continuing to completion (or infinity)
  - V: The expected utility gained after arriving in a state and continuing to completion (or infinity)
  - π: A policy determining what action to take given the current state
    - Dictionary mapping {state : action}
    - Has no history, only uses current state
  - γ: The discount rate - a multiplier that reduces the utility gained later in the process
    - E.g. γ=.9, R(s1,a1) = 1, R(s2,a2)= 1, R(s3,a3)= 10, Total = 1 + 1*γ + 10*γ*γ = 10
- How to "solve" an MDP - iteration
  - Bellman Equations
  - $Q^*(s,a) = \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$
  - $V^*(s) = \max_a Q^*(s,a)$
- Approximate Q learning

$$Q^*(s,a) = \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$

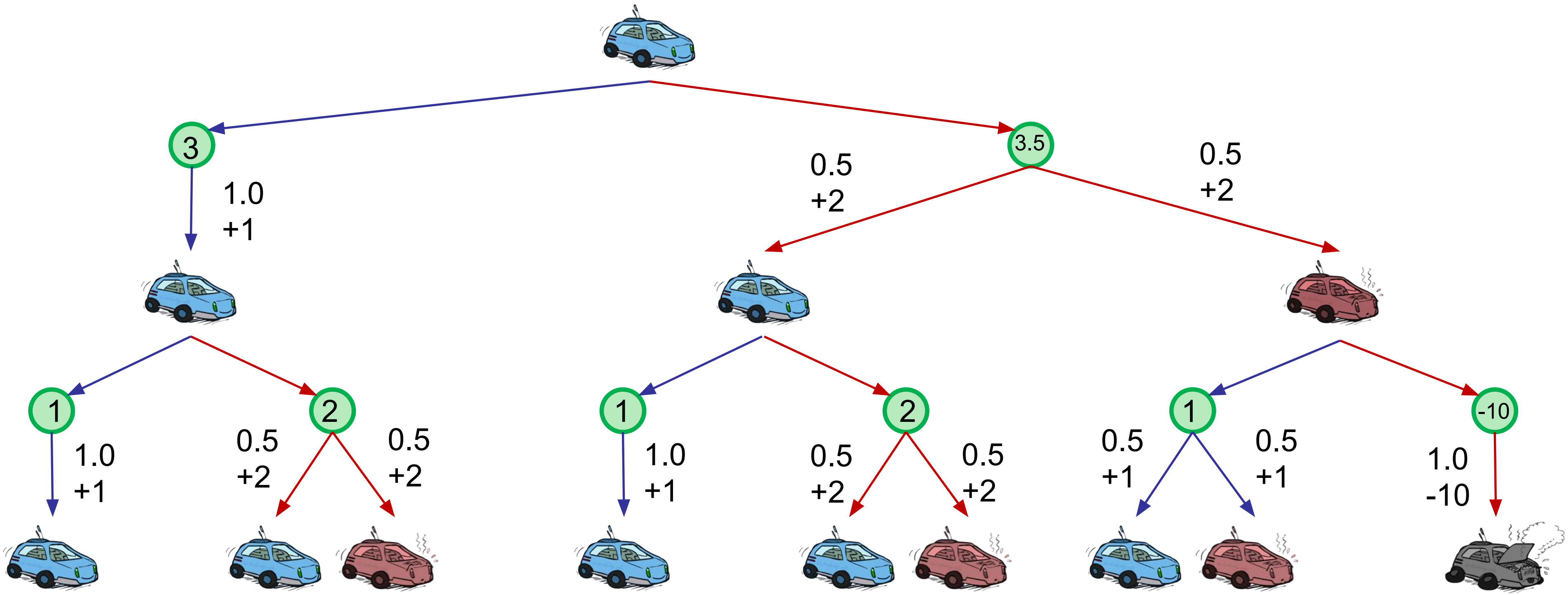
$$V^*(s) = \max_a Q^*(s,a)$$



Example Search Tree

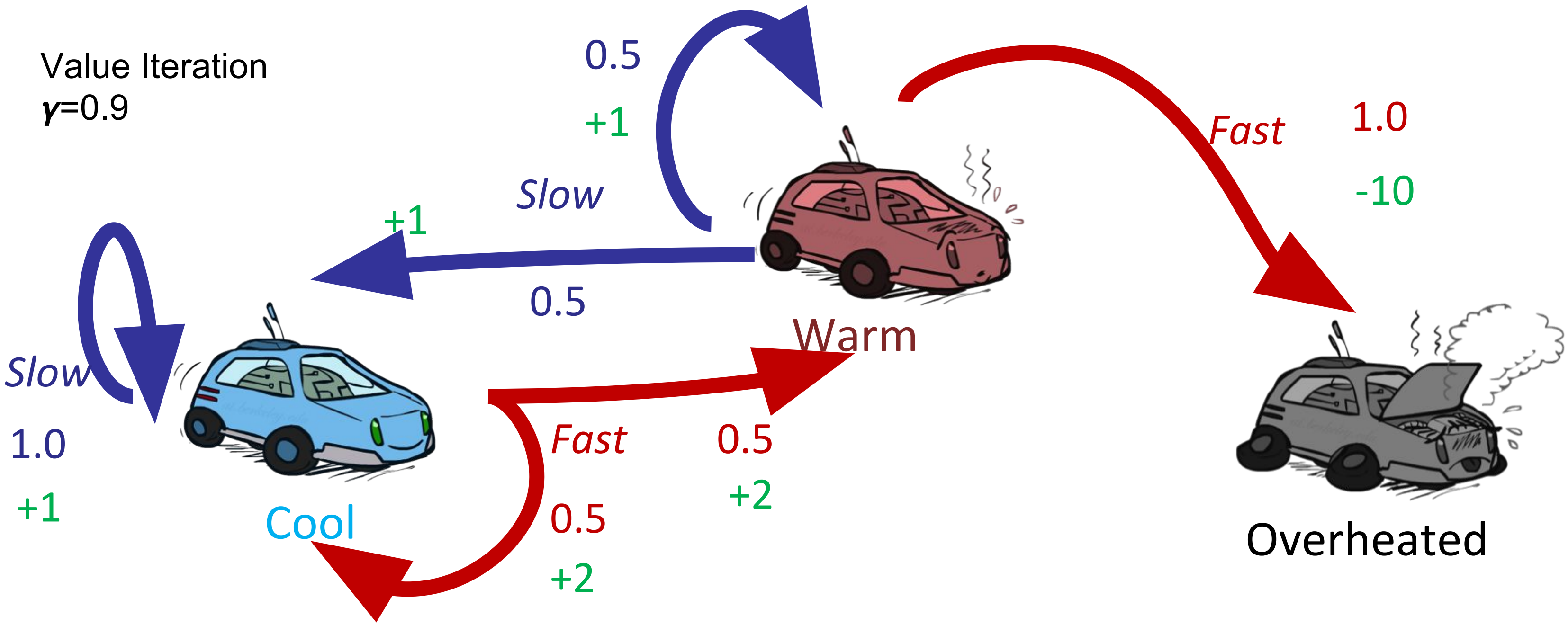


Example Search Tree



Demo

Value Iteration
γ=0.9



One timestep of value iteration

Initial Values

State s	Cool	Warm	Overheated
$V_0(s)$	0	0	0

One timestep of value iteration

Initial Values

State s	Cool	Warm	Overheated
$V_0(s)$	0	0	0

i=1

State s	Cool	Warm	Overheated
$V_0(s)$	0	0	0
$V_1(s)$	$\max(1+\gamma V_0(cool), \frac{(2+\gamma V_0(cool))+(2+\gamma V_0(warm))}{2})$	$\max(\frac{(1+\gamma V_0(cool))+(1+\gamma V_0(warm))}{2}, -10+\gamma V_0(overheated))$	0

One timestep of value iteration

Initial Values

State s	Cool	Warm	Overheated
$V_0(s)$	0	0	0

i=1

State s	Cool	Warm	Overheated
$V_0(s)$	0	0	0
$V_1(s)$	max(1,2)	max(1,-10)	0

One timestep of value iteration

Initial Values

State s	Cool	Warm	Overheated
$V_0(s)$	0	0	0

i=1

State s	Cool	Warm	Overheated
$V_0(s)$	0	0	0
$V_1(s)$	2	1	0

One timestep of value iteration

i=2

State s	Cool	Warm	Overheated
$V_0(s)$	0	0	0
$V_1(s)$	2	1	0
$V_2(s)$	max(1+.9*2, 2+.9*1)	max(1+.9*1.5, -10)	0

One timestep of value iteration

i=2

State s	Cool	Warm	Overheated
$V_0(s)$	0	0	0
$V_1(s)$	2	1	0
$V_2(s)$	2.9	2.35	0

Code Practice

Open Q_Learning.ipynb

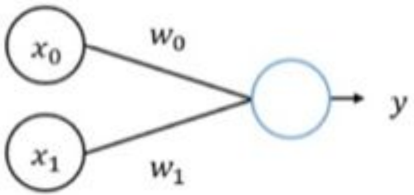
Neural Networks

Introduction

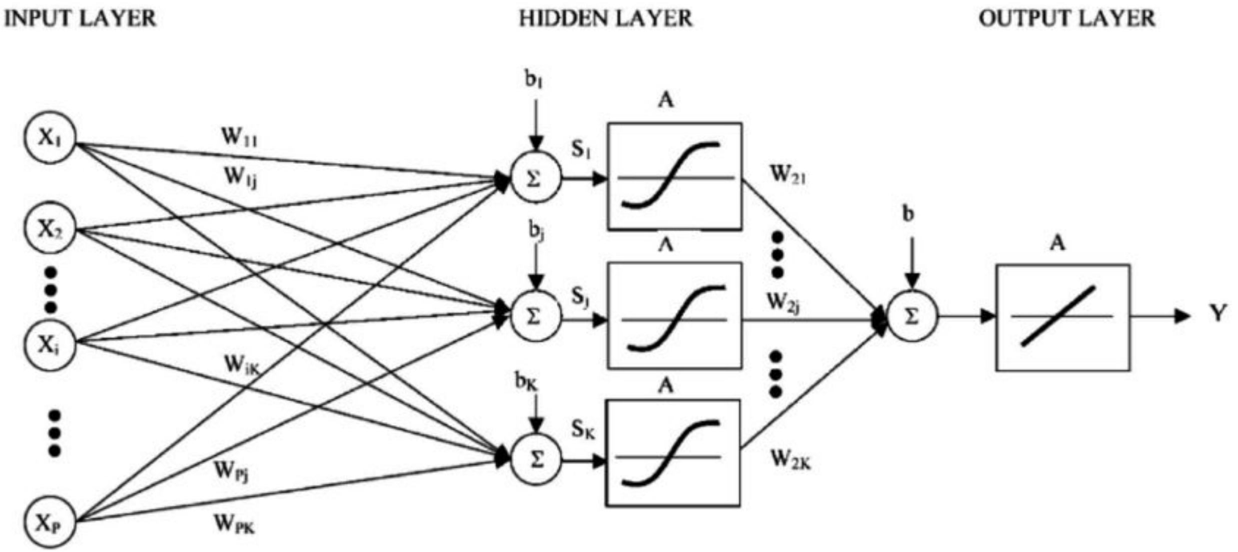
Perceptrons

Activation functions

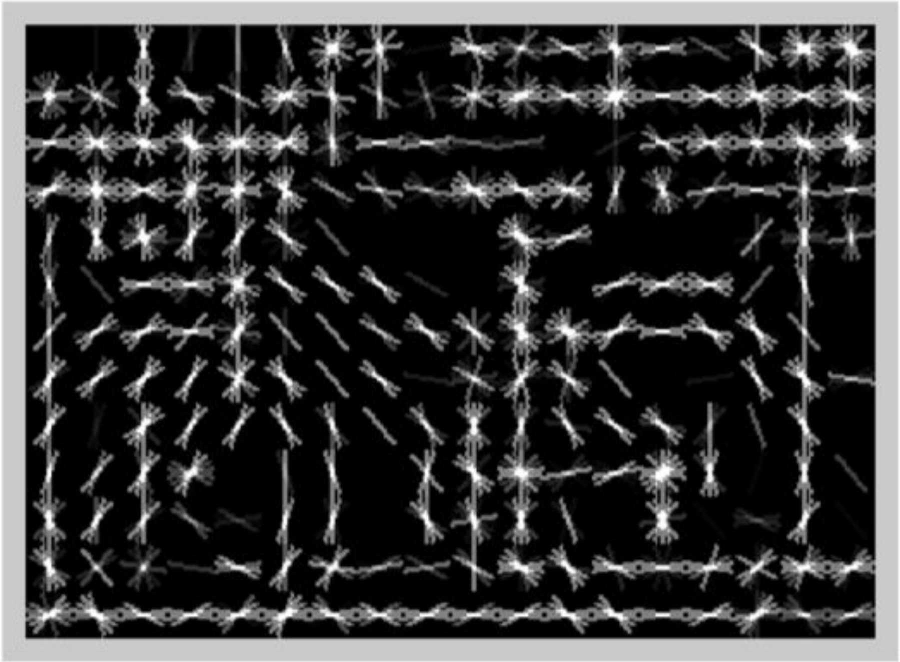
Layered neural networks



$$y = f(x_0 w_0 + x_1 w_1)$$

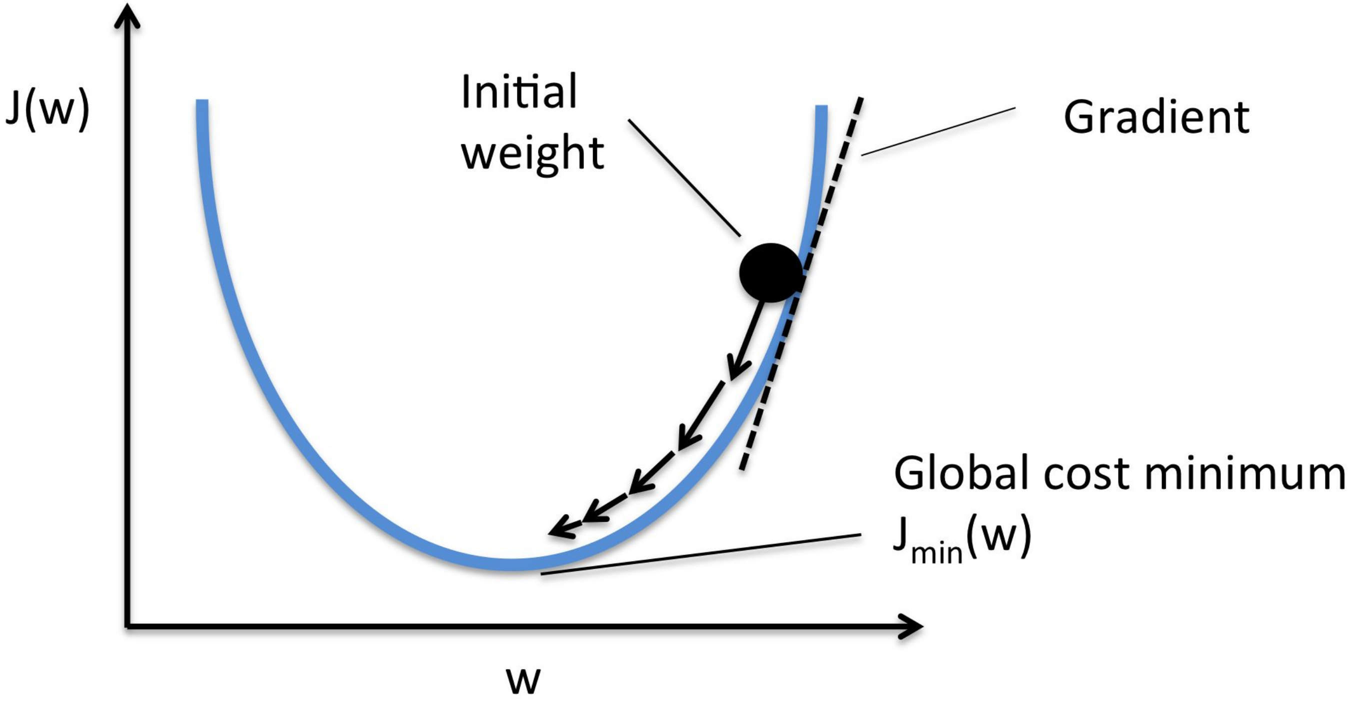


An Example of Manual Design

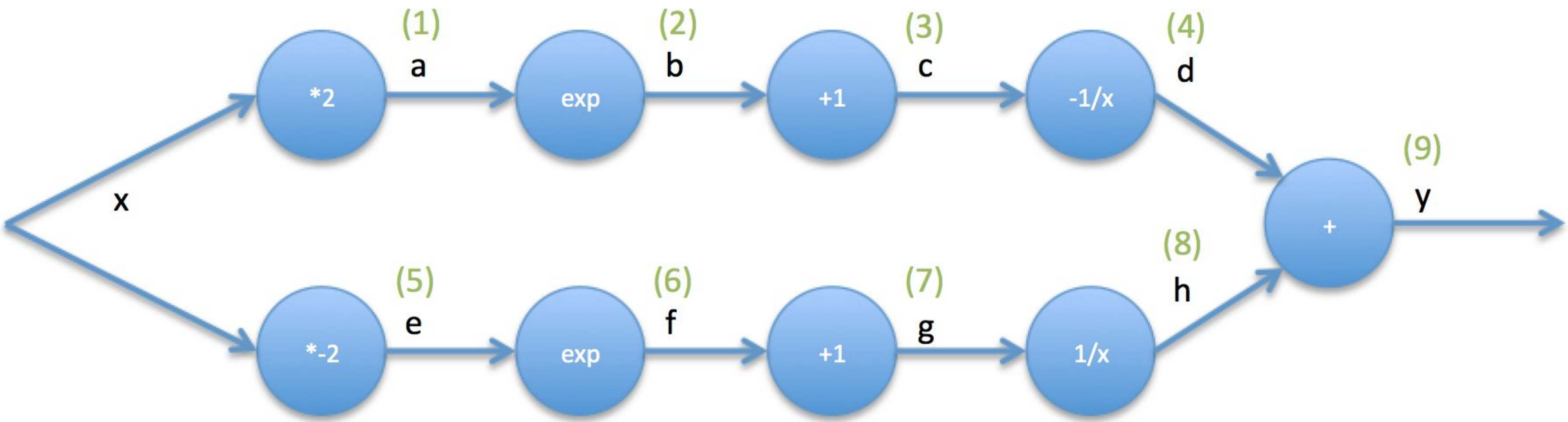


Training a Neural Network

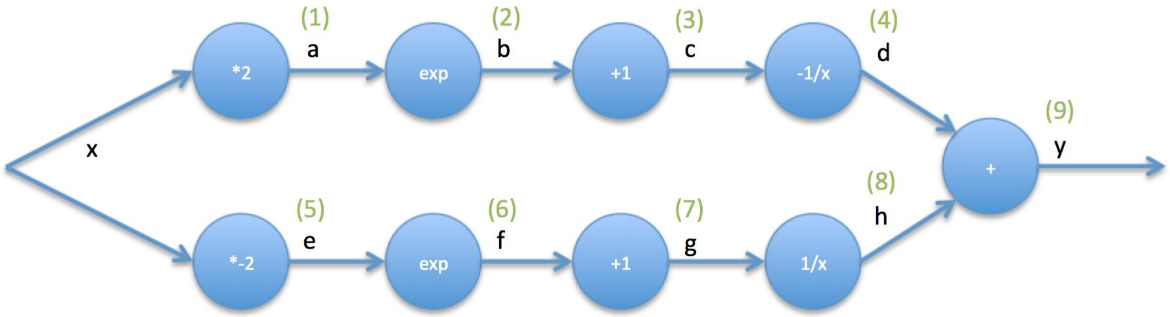
Loss Minimization - Gradient descent



Demo - Gradient Descent on TanH



Forward Propagation



Calculate the value at each point given input (e.g. x= 0.1)

A = .2

B = $e^{.2}$

C = $e^{.2} + 1$

D = $-\dfrac{1}{e^{.2} + 1}$

E = -.2

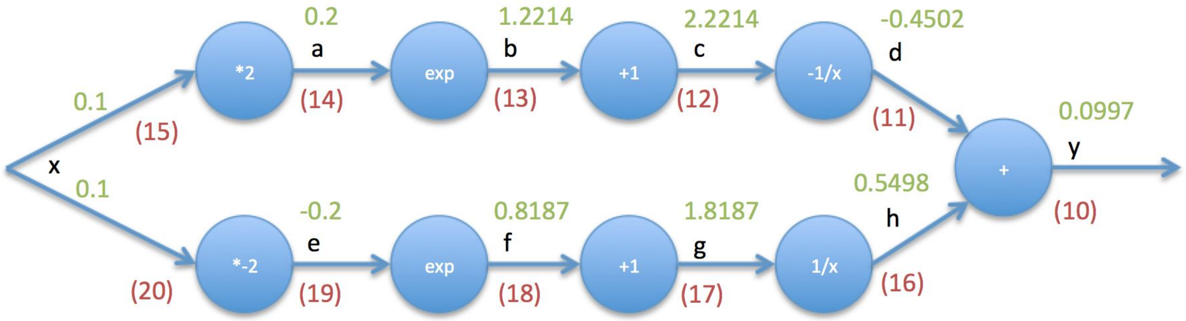
F = $e^{-.2}$

G = $e^{-.2} + 1$

H = $\dfrac{1}{e^{-.2} + 1}$

Y = $\dfrac{1}{e^{-.2} + 1} - \dfrac{1}{e^{.2} + 1}$

Forward Propagation



Calculate the value at each point given input (e.g. x= 0.1)

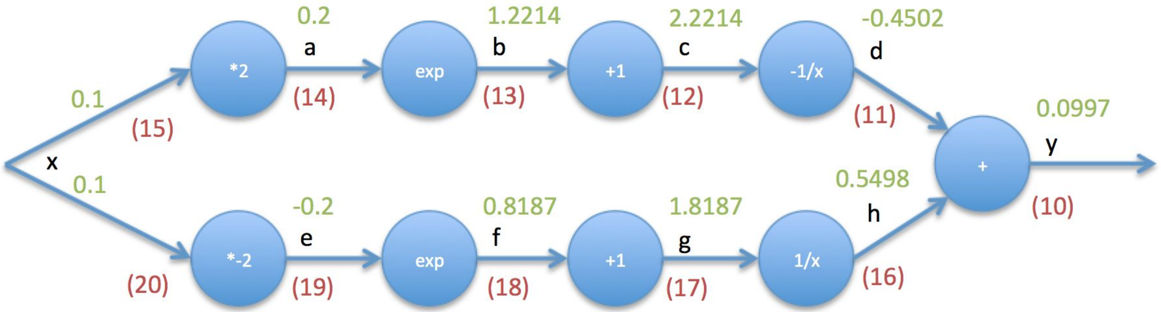
A = .2	A = .2
B = $e^{.2}$	B = 1.2214
C = $e^{.2} + 1$	C = 2.2214
D = $-\dfrac{1}{e^{.2}+1}$	D = -0.4502
E = -.2	E = -.2
F = $e^{-.2}$	F = 0.8187
G = $e^{-.2} + 1$	G = 1.8187
H = $\dfrac{1}{e^{-.2}+1}$	H = 0.5498
Y = $\dfrac{1}{e^{-.2}+1} - \dfrac{1}{e^{.2}+1}$	Y = 0.0997

Back Propagation - Detour

Chain rule refresher

$$y = f(a(x), b(x)) \rightarrow \frac{dy}{dx} = \frac{\partial y}{\partial a}\frac{\partial a}{\partial x} + \frac{\partial y}{\partial b}\frac{\partial b}{\partial x}$$

Back Propagation



Calculate the derivative at each point from y back to x through each path

10) $\frac{\partial y}{\partial y} = 1$

11) $\frac{\partial y}{\partial d} = 1$

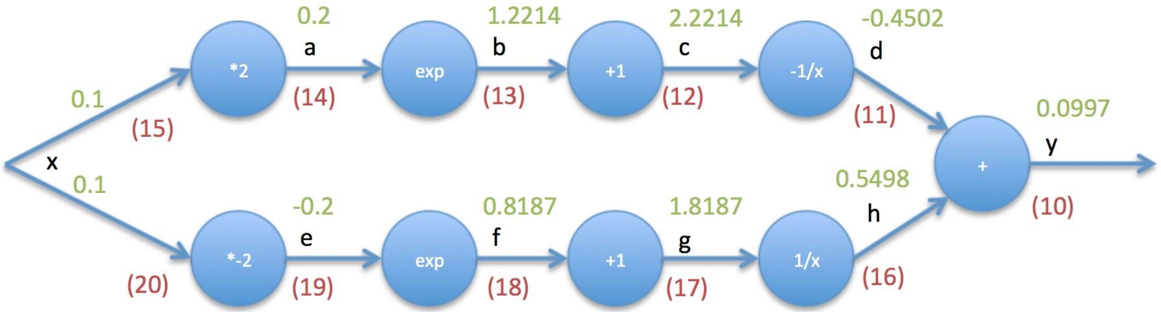
12) $\frac{\partial y}{\partial c} = \frac{\partial y}{\partial d}\frac{\partial d}{\partial c} = 1 * \frac{1}{c^2} \approx .20265$

13) $\frac{\partial y}{\partial b} = \frac{\partial y}{\partial c}\frac{\partial c}{\partial b} = .20265 * 1 = .20265$

14) $\frac{\partial y}{\partial a} = \frac{\partial y}{\partial b}\frac{\partial b}{\partial a} = .20265 * e^a = .24752$

15) $\frac{\partial y}{\partial x} = \frac{\partial y}{\partial a}\frac{\partial a}{\partial x} = .24752 * 2 = .49504$

Back Propagation



Calculate the derivative at each point from y back to x through each path

10) $\frac{\partial y}{\partial y} = 1$

11) $\frac{\partial y}{\partial d} = 1$

12) $\frac{\partial y}{\partial c} = \frac{\partial y}{\partial d}\frac{\partial d}{\partial c} = 1 * \frac{1}{c^2} \approx .20265$

13) $\frac{\partial y}{\partial b} = \frac{\partial y}{\partial c}\frac{\partial c}{\partial b} = .20265 * 1 = .20265$

14) $\frac{\partial y}{\partial a} = \frac{\partial y}{\partial b}\frac{\partial b}{\partial a} = .20265 * e^a = .24752$

15) $\frac{\partial y}{\partial x} = \frac{\partial y}{\partial a}\frac{\partial a}{\partial x} = .24752 * 2 = .49504$

16) $\frac{\partial y}{\partial h} = 1$

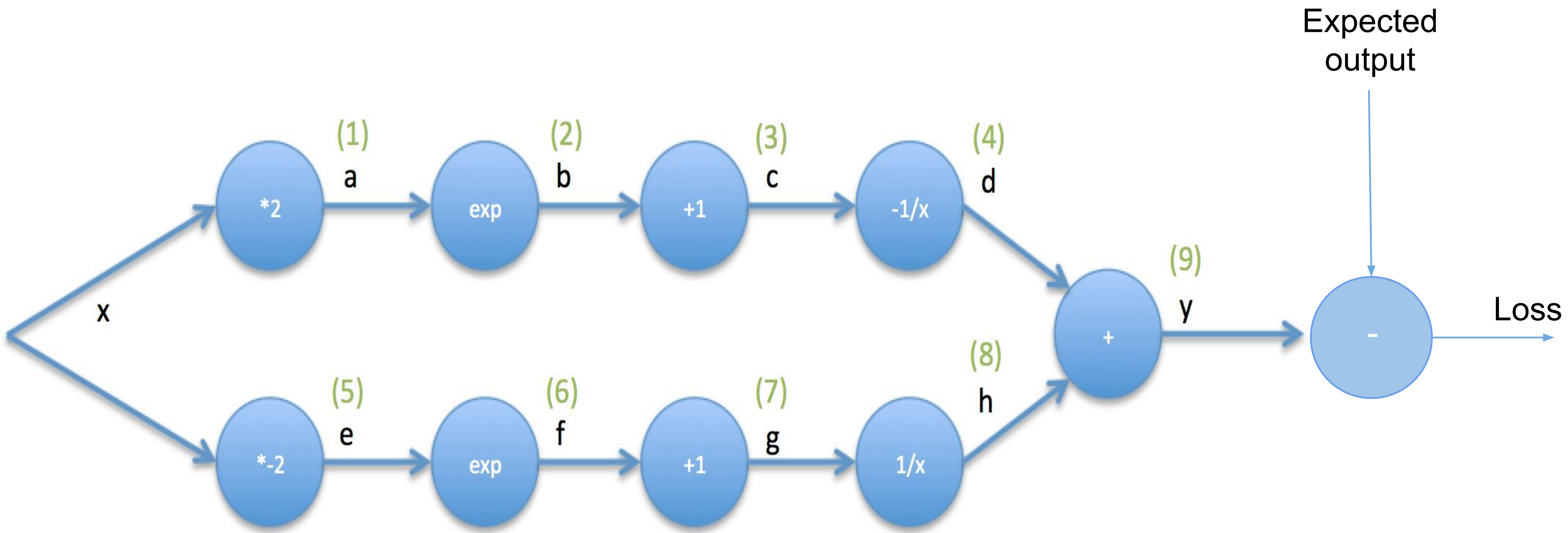
17) $\frac{\partial y}{\partial g} = \frac{\partial y}{\partial h}\frac{\partial h}{\partial g} = 1 * \frac{-1}{g^2} = -.30233$

18) $\frac{\partial y}{\partial f} = \frac{\partial y}{\partial g}\frac{\partial g}{\partial f} = -.30233 * 1 = -.30233$

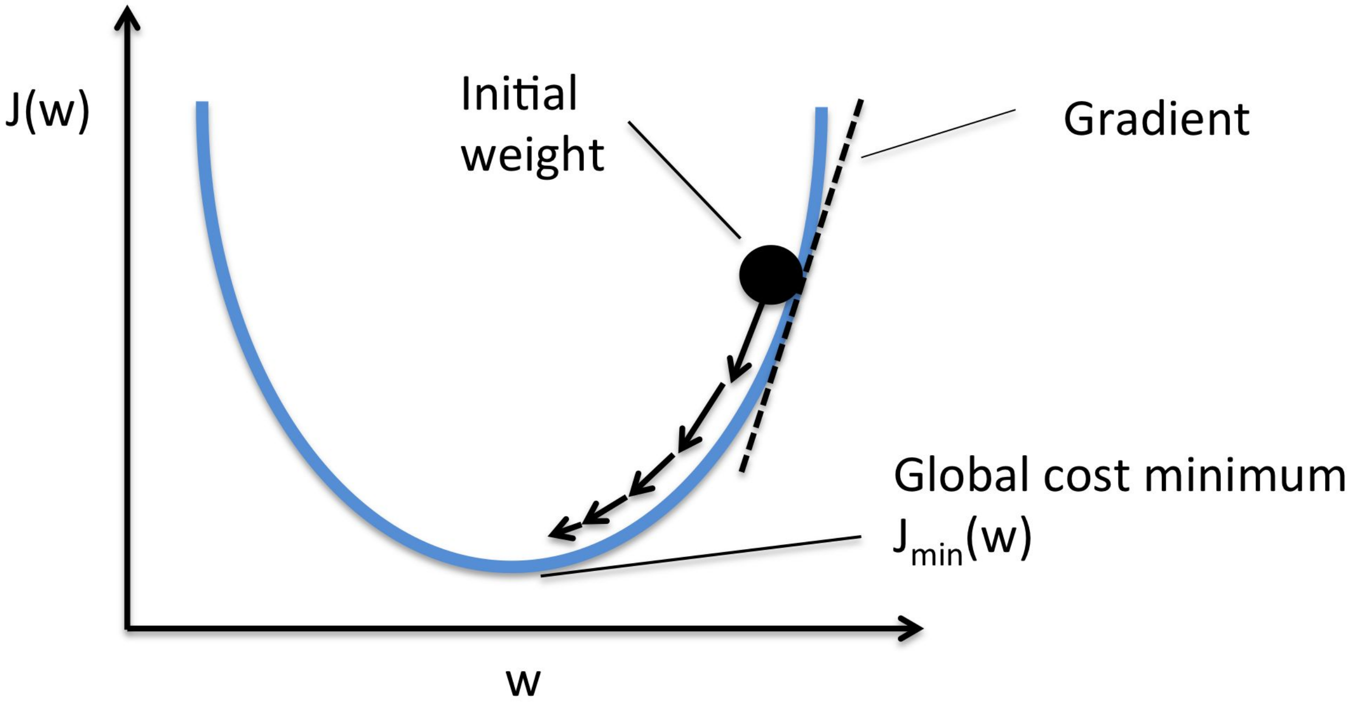
19) $\frac{\partial y}{\partial e} = \frac{\partial y}{\partial f}\frac{\partial f}{\partial e} = -.30233 * .8187 = -.24752$

20) $\frac{\partial y}{\partial x} = \frac{\partial y}{\partial e}\frac{\partial e}{\partial x} = -.247502 * -2 = .49504$

Adding Loss Minimization



Adding Loss Minimization



Code Practice

Open Neural_Networks.ipynb