CS 4300/6300: Artificial Intelligence

Midterm Review

Midterm Logistics

- In our classroom during normal class time
  - Thursday 12:25-1:45
- 1 sheet of notes (front and back)
- Calculator allowed but math will be simple and easy to do by hand

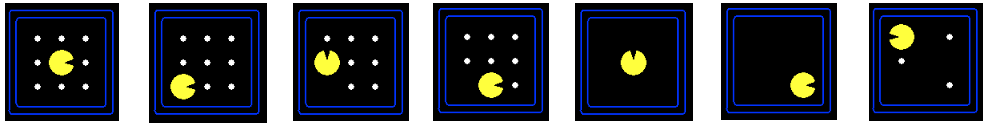
Topics you'll need to know

- A*
- Consistent/admissible heuristics
- Min-Max search
- Alpha-Beta pruning
- Expectimax
- Probability
  - conditional prob
  - Independence
  - Bayes' rule
  - chain rule

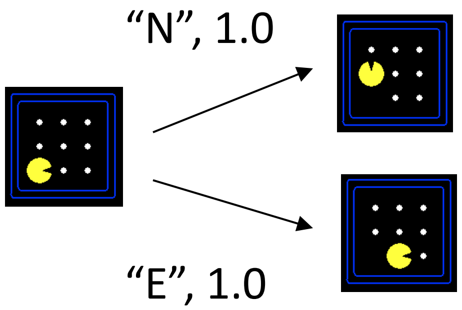
- MDPs
  - Value Iteration
  - Policy Iteration
  - Monte Carlo estimation
- Machine Learning
  - Perceptron
  - Classification
  - Regression

Search Problems

- A search problem consists of:
  - A state space



  - A successor function (with actions, costs)



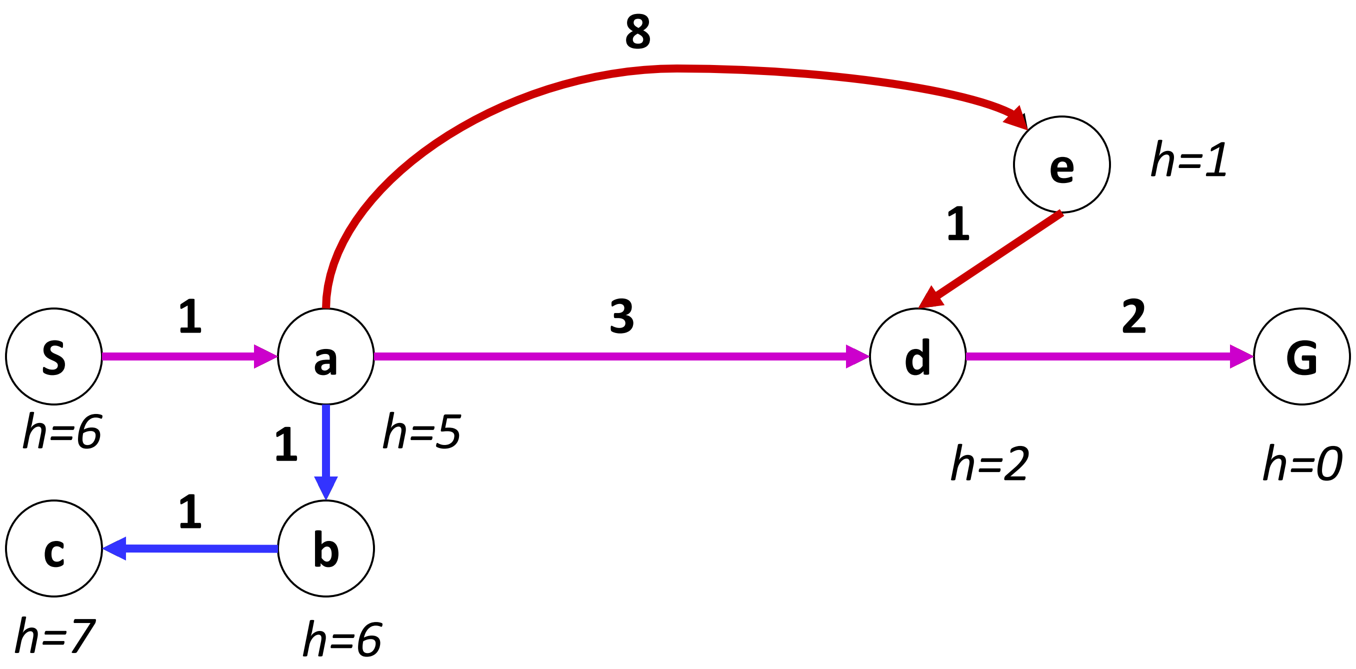
  - A start state and a goal test
- A solution is a sequence of actions (a plan) which transforms the start state to a goal state

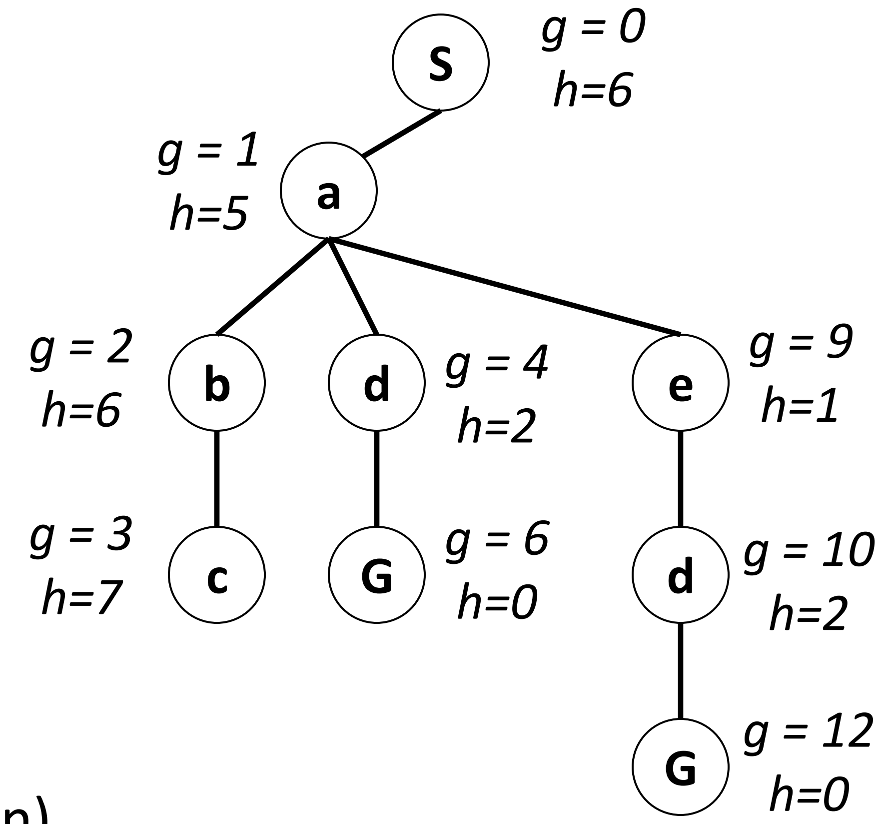
Graph Search Pseudo-Code

```
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
                if STATE[child-node] is not in closed then   fringe ← INSERT(child-node, fringe)
            end
    end
```

A-star: Combining UCS and Greedy

- Uniform-cost orders by path cost, or *backward cost* g(n)
- Greedy orders by goal proximity, or *forward cost* h(n)





- A* Search orders by the sum: f(n) = g(n) + h(n)

Example: Teg Grenager

Admissible Heuristics

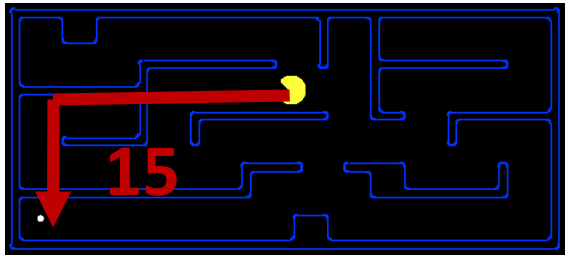
required for tree A* search to be opt.

- A heuristic h is ***admissible*** (optimistic) if:

$$0 \leq h(n) \leq h^*(n)$$

where $h^*(n)$ is the true cost to a nearest goal

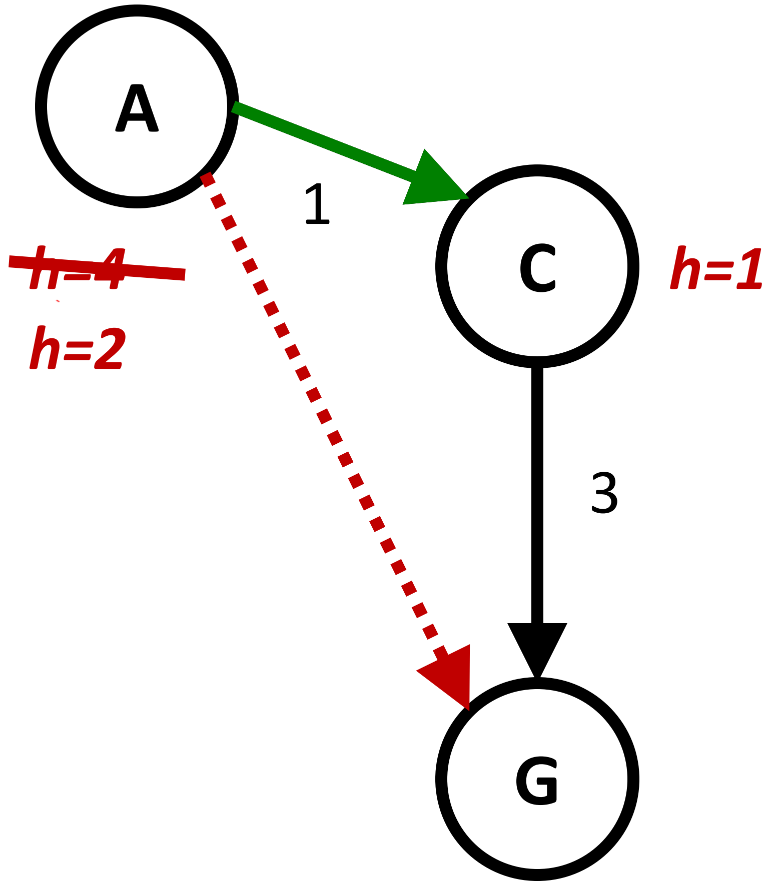
- Examples:



- Coming up with admissible heuristics is most of what's involved in using A* in practice.

required for graph search

Consistency of Heuristics



- Main idea: estimated heuristic costs ≤ actual costs
  - Admissibility: heuristic cost ≤ actual cost to goal

    h(A) ≤ actual cost from A to G
  - Consistency: heuristic "arc" cost ≤ actual cost for each arc

    h(A) – h(C) ≤ cost(A to C)
- Consequences of consistency:
  - The f value along a path never decreases

    h(A) ≤ cost(A to C) + h(C)
  - A* graph search is optimal

Adversarial Search

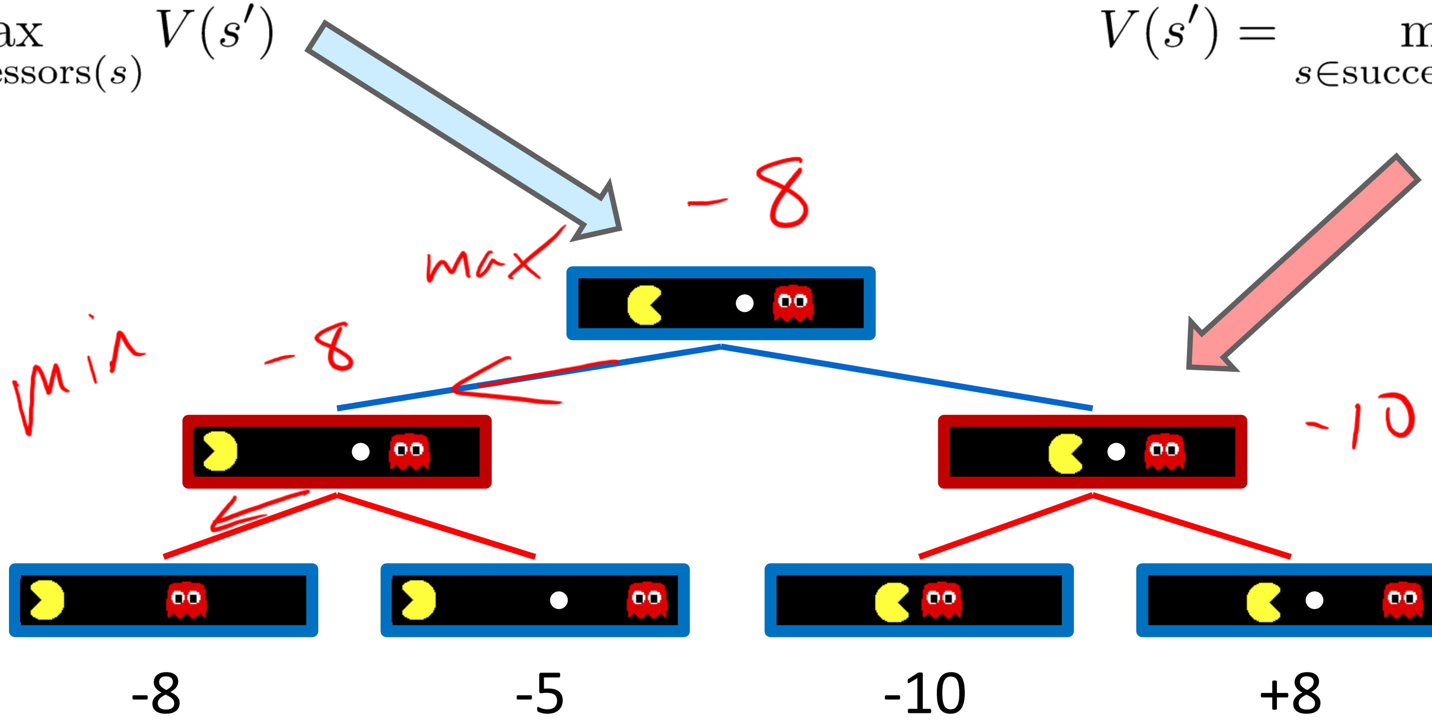
Minimax Values

States Under Agent's Control:

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

States Under Opponent's Control:

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$



Terminal States:

$$V(s) = \text{known}$$

Minimax Implementation

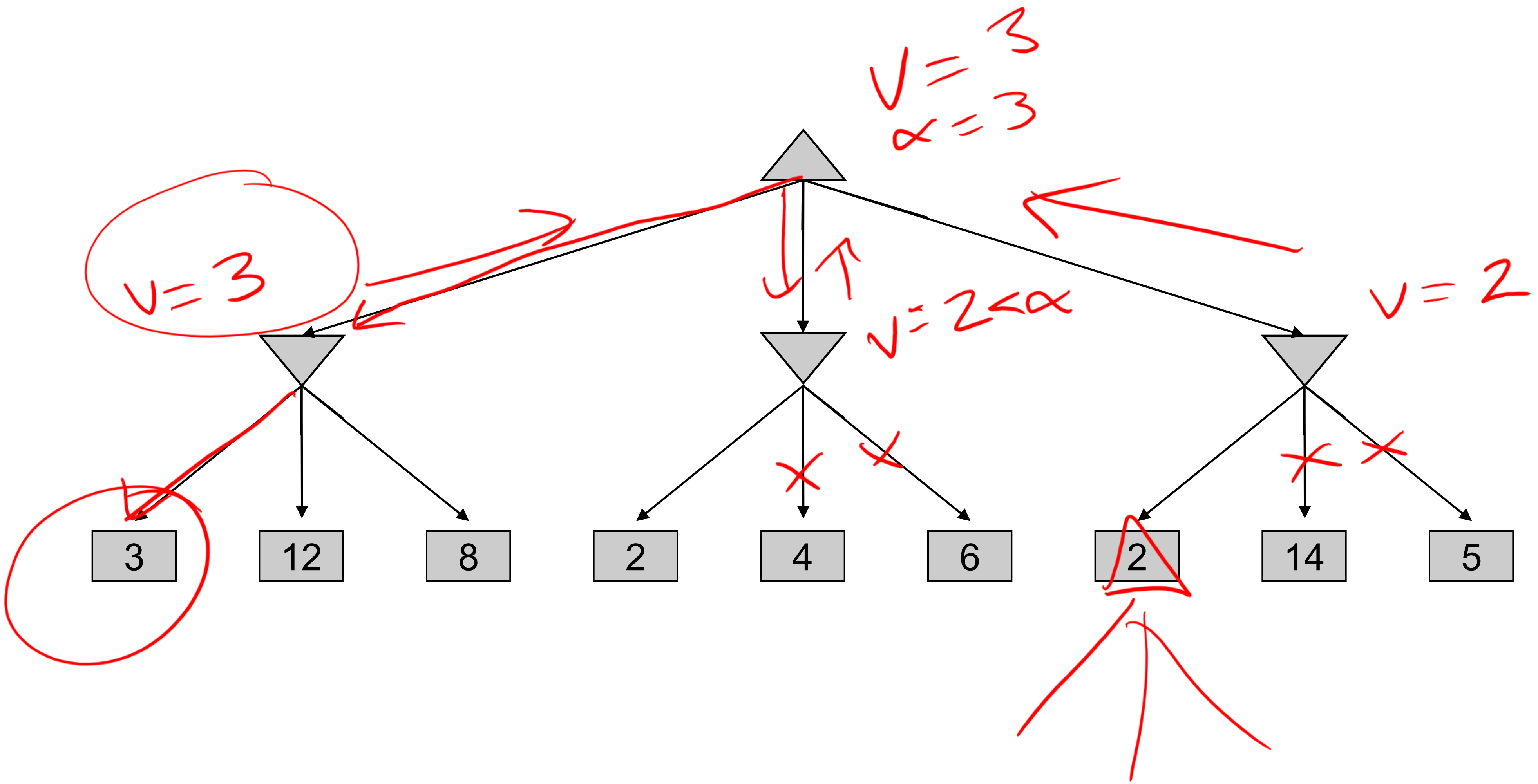
```
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

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

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

Minimax Example



Alpha-Beta Implementation

α: MAX's best option on path to root
β: MIN's best option on path to root

At root you should initialize $\alpha = -\infty$ and $\beta = +\infty$

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

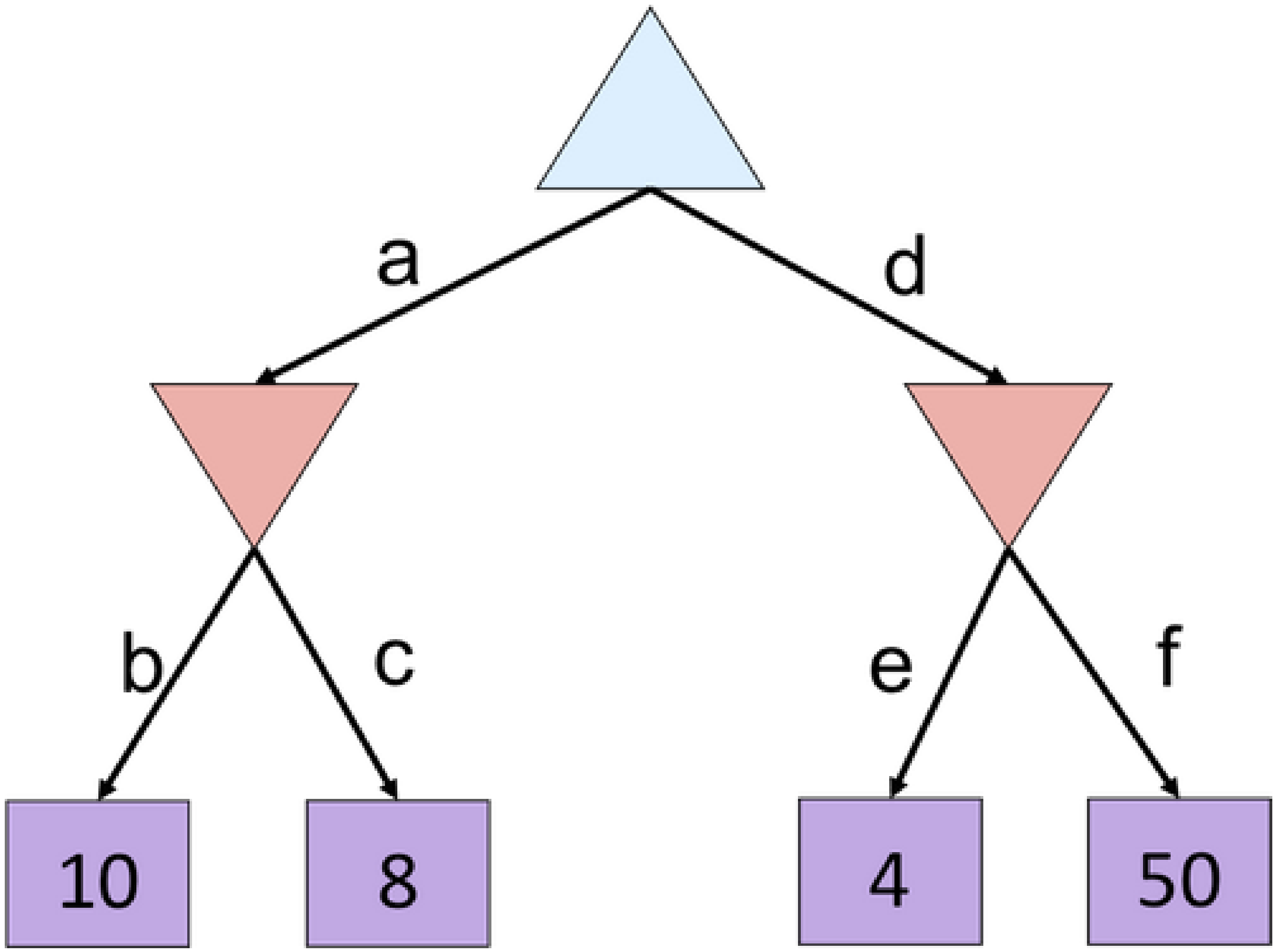
```
def min-value(state , α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor, α, β))
        if v ≤ α return v
        β = min(β, v)
    return v
```

Alpha-Beta Quiz

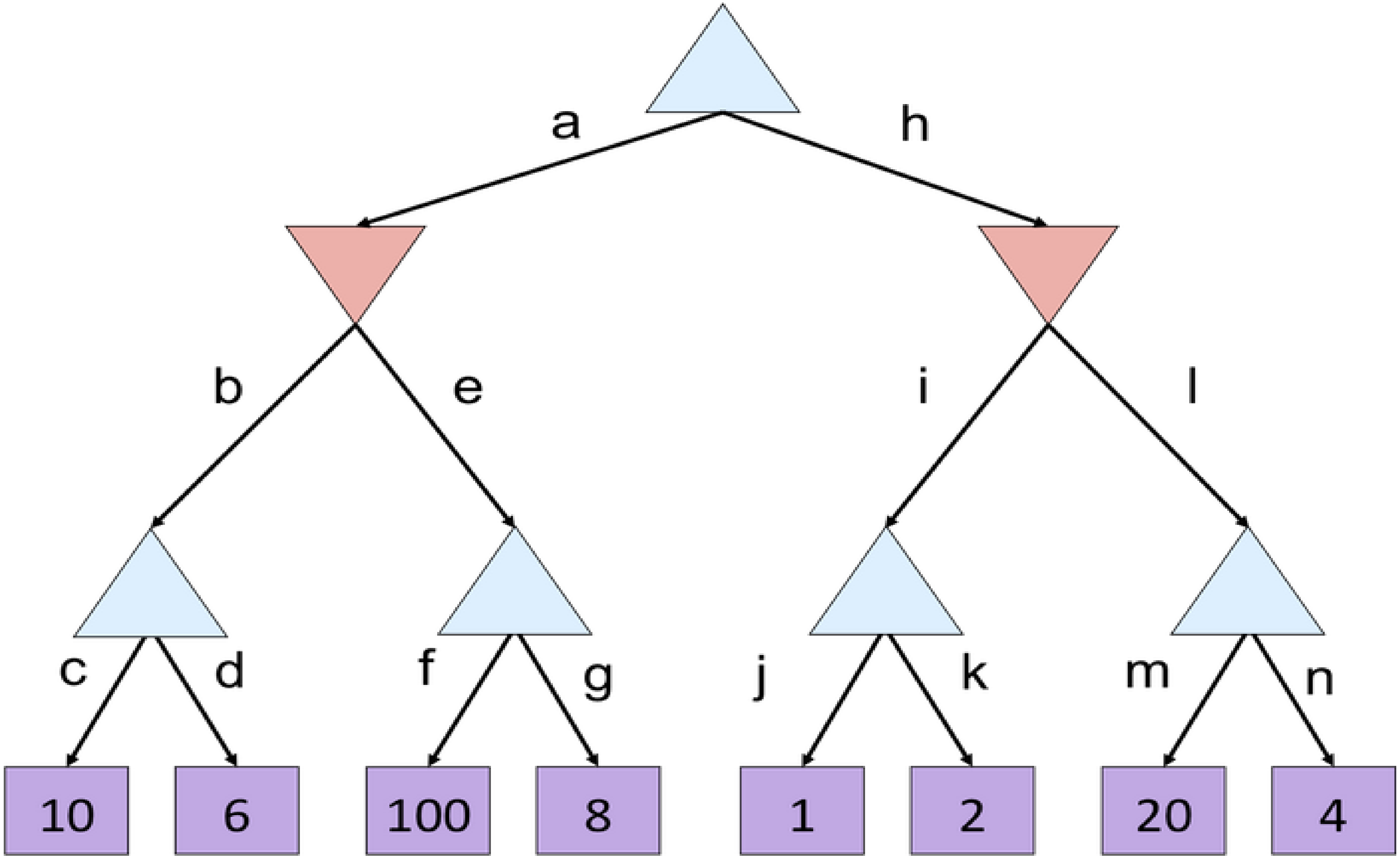
α: MAX's best option on path to root
β: MIN's best option on path to root

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

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



Alpha-Beta Example 2



Uncertain Search

Expectimax 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)
```

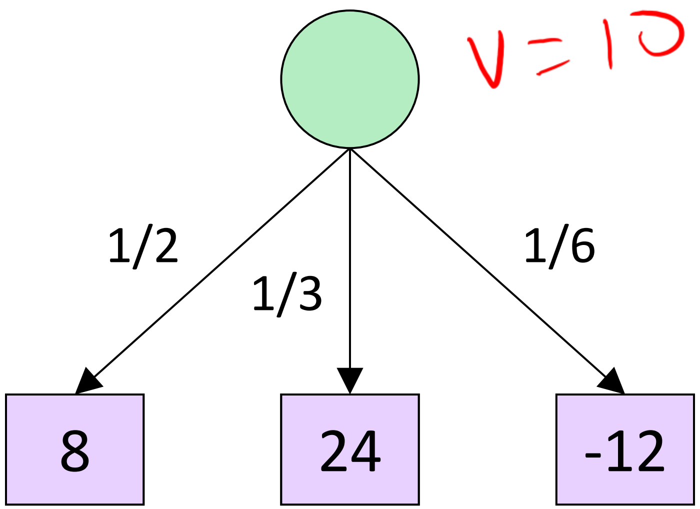
$\sum_{s \in succ} p(s) V(s)$

```
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
```

Expectimax Pseudocode

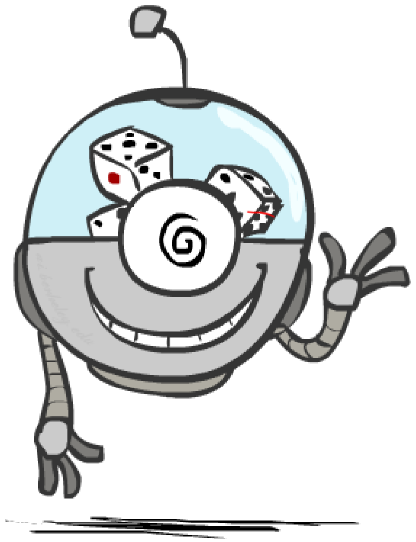
```
def exp-value(state):
    initialize v = 0
    for each successor of state:
        p = probability(successor)
        v += p * value(successor)
    return v
```

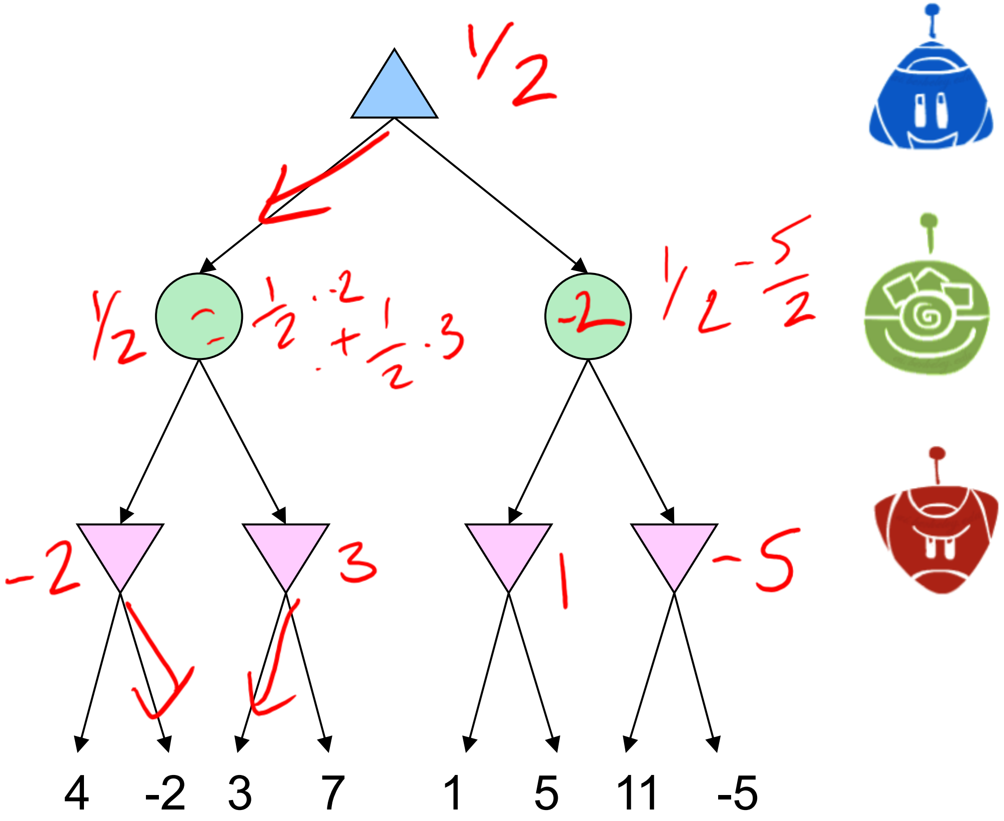


v = (1/2) (8) + (1/3) (24) + (1/6) (-12) = 10

Mixed Layer Types

- E.g. Backgammon
- Expectiminimax
  - Environment is an extra “random agent” player that moves after each min/max agent
  - Each node computes the appropriate combination of its children





Probability

Probability Distributions

- Unobserved random variables have distributions

$P(T)$

T	P
hot	0.5
cold	0.5

$P(W)$

W	P
sun	0.6
rain	0.1
fog	0.3
meteor	0.0

Shorthand notation:

$$P(hot) = P(T = hot),$$
$$P(cold) = P(T = cold),$$
$$P(rain) = P(W = rain),$$
$$\ldots$$

OK *if* all domain entries are unique

- A distribution is a TABLE of probabilities of values

- A probability (lower case value) is a single number

$$P(W = rain) = 0.1$$

- Must have: $\forall x \;\; P(X = x) \geq 0$ and $\sum_x P(X = x) = 1$

Joint Distributions

- A *joint distribution* over a set of random variables: $X_1, X_2, \ldots X_n$ specifies a real number for each assignment (or *outcome*):

$$P(X_1 = x_1, X_2 = x_2, \ldots X_n = x_n)$$

$$P(x_1, x_2, \ldots x_n)$$

  - Must obey: $$P(x_1, x_2, \ldots x_n) \geq 0$$

$$\sum_{(x_1, x_2, \ldots x_n)} P(x_1, x_2, \ldots x_n) = 1$$

$P(T, W)$

T	W	P
hot	sun	0.4
hot	rain	0.1
cold	sun	0.2
cold	rain	0.3

- Size of distribution if n variables with domain sizes d?
  - For all but the smallest distributions, impractical to write out!

Quiz: Events

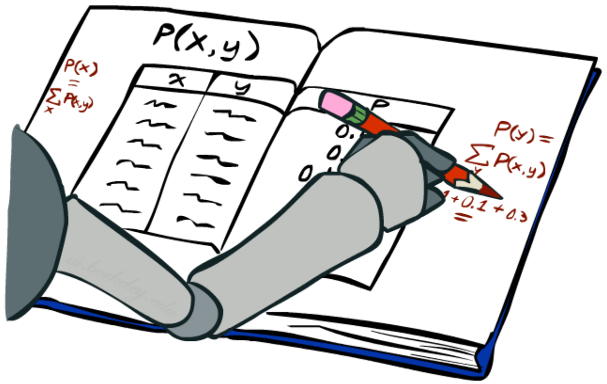
- P(+x, +y) ?
- P(+x) ?
- P(-y OR +x) ?

$$P(X, Y)$$

X	Y	P
+x	+y	0.2
+x	-y	0.3
-x	+y	0.4
-x	-y	0.1

Marginal Distributions

$\sum_W P(T,W) = P(T)$



- Marginal distributions are sub-tables which eliminate variables
- Marginalization (summing out): Combine collapsed rows by adding

$P(T,W)$

T	W	P
hot	sun	0.4
hot	rain	0.1
cold	sun	0.2
cold	rain	0.3

$$P(t) = \sum_s P(t,s)$$

$P(T)$

T	P
hot	0.5
cold	0.5

$$P(s) = \sum_t P(t,s)$$

$P(W)$

W	P
sun	0.6
rain	0.4

$P(W=s) = \sum_t P(t, W=s)$

$$P(X_1 = x_1) = \sum_{x_2} P(X_1 = x_1, X_2 = x_2)$$

Quiz: Marginal Distributions

$\sum_y P(X=+x, y) = P(+x,+y) + P(+x,-y)$
$= 0.2 + 0.3$

$P(X,Y)$

X	Y	P
+x	+y	0.2
+x	-y	0.3
-x	+y	0.4
-x	-y	0.1

$$P(x) = \sum_y P(x,y)$$

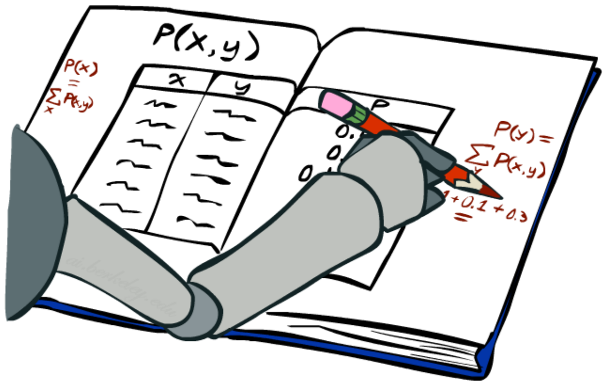
$$P(y) = \sum_x P(x,y)$$

$P(X)$

X	P
+x	0.5
-x	

$P(Y)$

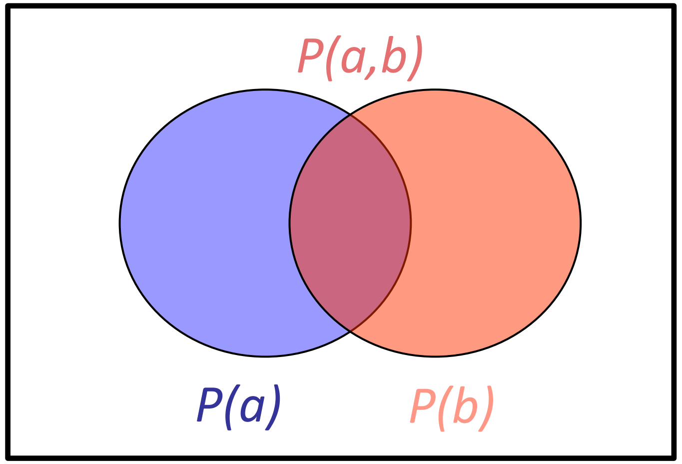
Y	P
+y	
-y	



Conditional Probabilities

- A simple relation between joint and conditional probabilities
  - In fact, this is taken as the *definition* of a conditional probability

$$P(a|b) = \frac{P(a,b)}{P(b)}$$



$P(T, W)$

T	W	P
hot	sun	0.4
hot	rain	0.1
cold	sun	0.2
cold	rain	0.3

$$P(W = s|T = c) = \frac{P(W = s, T = c)}{P(T = c)} = \frac{0.2}{0.5} = 0.4$$

$$= P(W = s, T = c) + P(W = r, T = c)$$
$$= 0.2 + 0.3 = 0.5$$

$P(T)$

c	0.5
h	0.5

Quiz: Conditional Probabilities

$P(X, Y)$

X	Y	P
+x	+y	0.2
+x	-y	0.3
-x	+y	0.4
-x	-y	0.1

- P(+x | +y) ? $= \dfrac{P(+x, +y)}{P(+y)}$
- P(-x | +y) ?
- P(-y | +x) ?

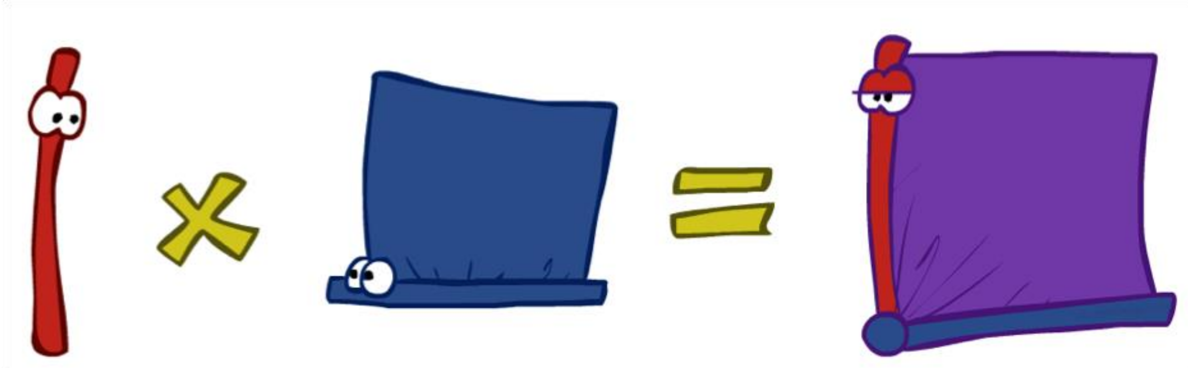
$P(Y)$

$P(X)$

The Product Rule

- Sometimes have conditional distributions but want the joint

$$P(y)P(x|y) = P(x,y) \quad \Longleftrightarrow \quad P(x|y) = \frac{P(x,y)}{P(y)}$$



The Product Rule

$$P(y)P(x|y) = P(x,y)$$

- Example:

$P(W)$

R	P
sun	0.8
rain	0.2

$P(D|W)$

D	W	P
wet	sun	0.1
dry	sun	0.9
wet	rain	0.7
dry	rain	0.3

$P(D,W)$

D	W	P
wet	sun	
dry	sun	
wet	rain	
dry	rain	

The Chain Rule

- More generally, can always write any joint distribution as an incremental product of conditional distributions

$$P(x_1, x_2, x_3) = P(x_1)P(x_2|x_1)P(x_3|x_1, x_2)$$

$$P(x_1, x_2, \ldots x_n) = \prod_i P(x_i|x_1 \ldots x_{i-1})$$

- You can pick any order.

- Why is the Chain Rule always true?

Bayes' Rule

- Two ways to factor a joint distribution over two variables:

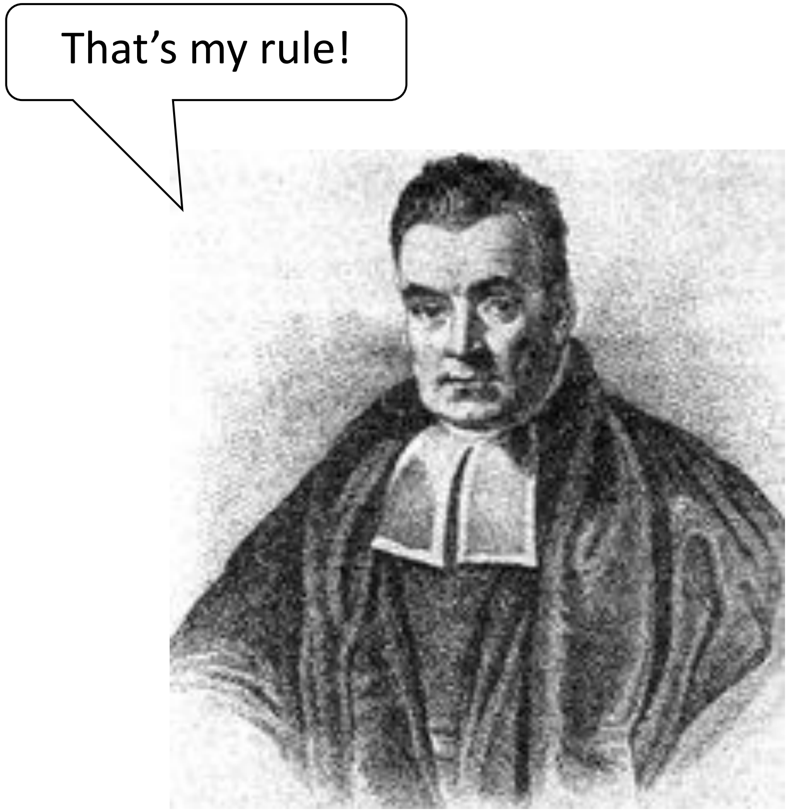
$$P(x,y) = P(x|y)P(y) = P(y|x)P(x)$$

- Dividing, we get:

$$P(x|y) = \frac{P(y|x)}{P(y)}P(x)$$

- Why is this at all helpful?
  - Lets us build one conditional from its reverse
  - Often one conditional is tricky but the other one is simple
  - Foundation of many systems (e.g. ASR, MT, IRL)

- In the running for most important AI equation!



Independence

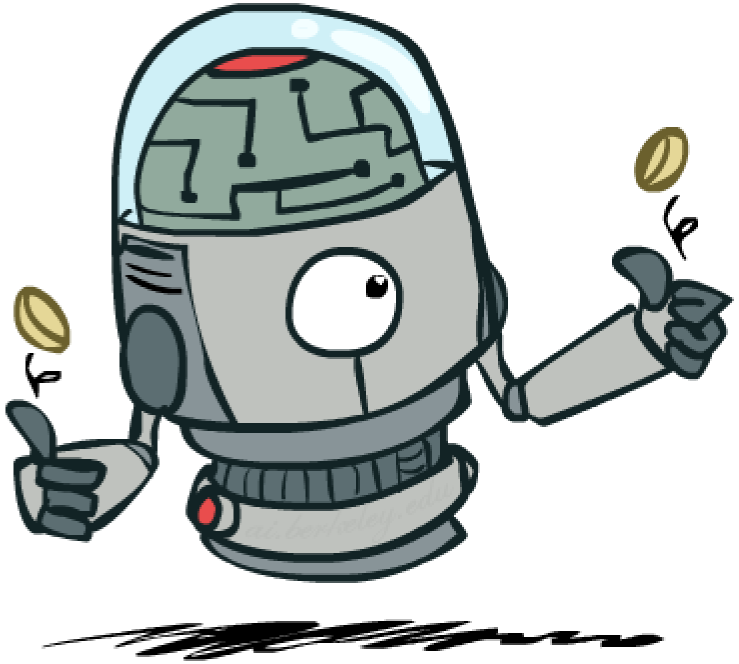
- Two variables are *independent* in a joint distribution if:

$$P(X,Y) = P(X)P(Y)$$

$$X \perp\!\!\!\perp Y$$

$$\forall x, y \; P(x,y) = P(x)P(y)$$

  - Says the joint distribution *factors* into a product of two simple ones
  - Usually variables aren't independent!



- Can use independence as a *modeling assumption*
  - Independence can be a simplifying assumption
  - *Empirical* joint distributions: at best "close" to independent
  - What could we assume for {Weather, Traffic, Cavity}?
- Independence is like something from CSPs: what?

Example: Independence?

$P(T)$

T	P
hot	0.5
cold	0.5

$P_1(T, W)$

T	W	P
hot	sun	0.4
hot	rain	0.1
cold	sun	0.2
cold	rain	0.3

$P(W)$

W	P
sun	0.6
rain	0.4

$P_2(T, W) = P(T)P(W)$

T	W	P
hot	sun	0.3
hot	rain	0.2
cold	sun	0.3
cold	rain	0.2

Conditional Independence

- Unconditional (absolute) independence very rare (why?)
- *Conditional independence* is our most basic and robust form of knowledge about uncertain environments.
- X is conditionally independent of Y given Z $X \perp\!\!\!\perp Y | Z$

if and only if:

$$\forall x, y, z : P(x, y|z) = P(x|z)P(y|z)$$

or, equivalently, if and only if

$$\forall x, y, z : P(x|z, y) = P(x|z)$$

You should feel comfortable with equations

Assume $P(x,y|z) = P(x|z)\,P(y|z)$

Prove

$\Rightarrow P(x|z,y) = P(x|z)$

$$P(x|z,y) \overset{\text{def}}{=} \frac{P(x,y,z)}{P(y,z)} \overset{\text{chain}}{=} \frac{P(x,y|z)\,P(z)}{P(y,z)} \overset{\text{plug in assump}}{=} \frac{P(x|z)\,P(y|z)\,P(z)}{P(y,z)} = P(x|z)$$

prod: $\frac{P(y,z)}{}$

Forwards and backwards

Probability Recap

- Conditional probability $$P(x|y) = \frac{P(x,y)}{P(y)}$$

Bayes' Rule

$$P(x|y) = \frac{P(y|x)P(x)}{P(y)}$$

- Product rule $$P(x,y) = P(x|y)P(y)$$

- Chain rule
$$\begin{aligned} P(X_1, X_2, \ldots X_n) &= P(X_1)P(X_2|X_1)P(X_3|X_1, X_2)\ldots \\ &= \prod_{i=1}^{n} P(X_i|X_1, \ldots, X_{i-1}) \end{aligned}$$

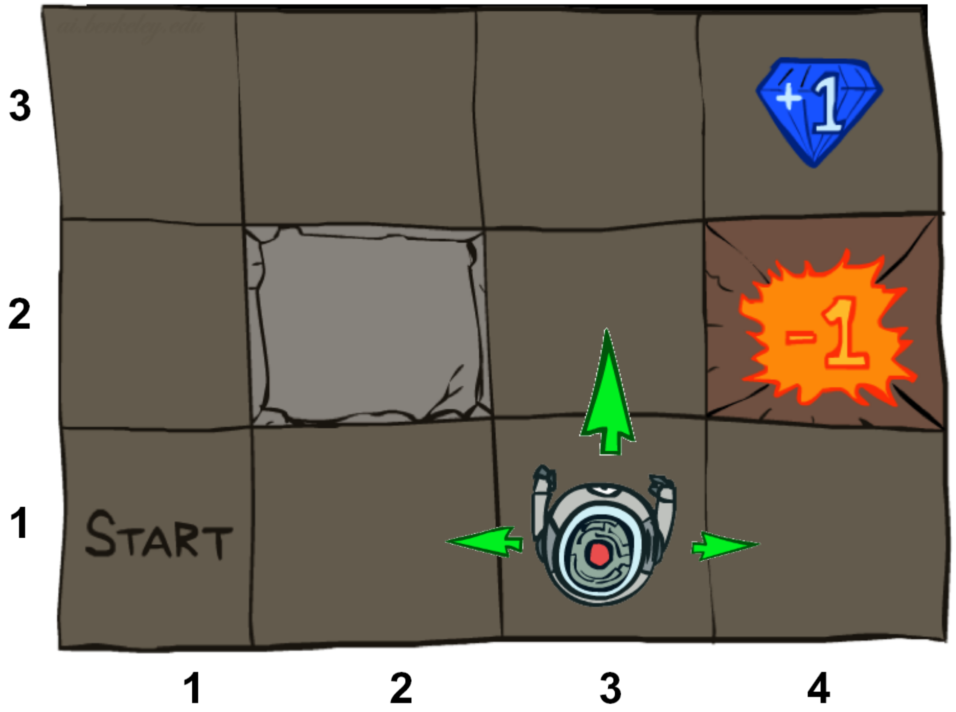
- X, Y independent if and only if: $\forall x, y : P(x,y) = P(x)P(y)$

- X and Y are conditionally independent given Z if and only if: $X \perp\!\!\!\perp Y | Z$

$$\forall x, y, z : P(x, y|z) = P(x|z)P(y|z)$$

Markov Decision Processes

- An MDP is defined by:
  - A set of states s ∈ S
  - A set of actions a ∈ A
  - A transition function T(s, a, s')
    - Probability that a from s leads to s', i.e., P(s' | s, a)
    - Also called the model or the dynamics
  - A reward function R(s, a, s')
    - Sometimes just R(s) or R(s')
  - A start state
  - Maybe a terminal state
  - Discount factor γ



- MDPs are non-deterministic search problems
  - One way to solve them is with expectimax search
  - Policy Iteration and Value Iteration

[Demo – gridworld manual intro (L8D1)]

What is Markov about MDPs?

- "Markov" generally means that given the present state, the future and the past are independent
- For Markov decision processes, "Markov" means action outcomes depend only on the current state

$$P(S_{t+1} = s' | S_t = s_t, A_t = a_t, S_{t-1} = s_{t-1}, A_{t-1}, \ldots S_0 = s_0)$$
$$= \quad P(S_{t+1} = s' | S_t = s_t, A_t = a_t)$$

- This is just like search, where the successor function could only depend on the current state (not the history)

Andrey Markov
(1856-1922)

Important Quantities

- The value (utility) of a state s:

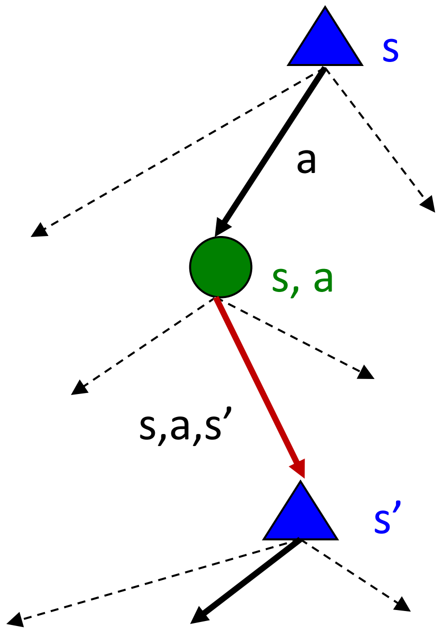
  $V^*(s)$ = expected utility starting in s and acting optimally

- The value (utility) of a q-state (s,a):

  $Q^*(s,a)$ = expected utility starting out having taken action a from state s and (thereafter) acting optimally

- The optimal policy:

  $\pi^*(s)$ = optimal action from state s



s is a *state*

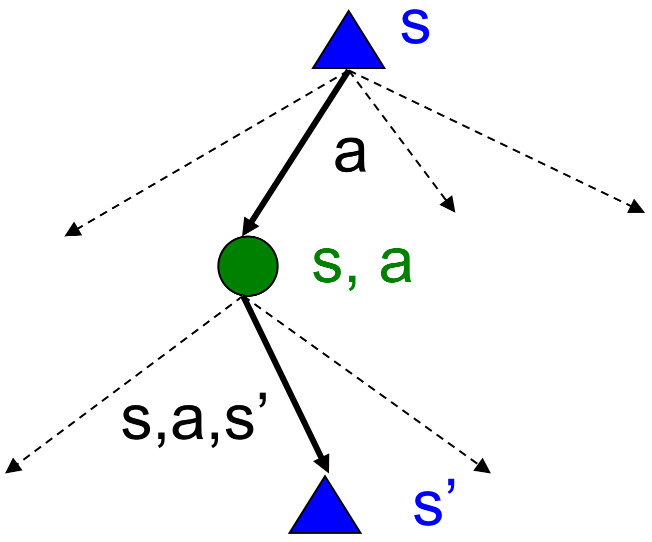
(s, a) is a *q-state*

(s,a,s') is a *transition*

[Demo – gridworld values (L8D4)]

Bellman Equations

- Fundamental operation: compute the (expectimax) value of a state
  - Expected utility under optimal action
  - Average sum of (discounted) rewards
  - This is just what expectimax computed!



- Recursive definition of value:

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

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

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

Value Iteration

- Start with $V_0(s) = 0$: no time steps left means an expected reward sum of zero
- Given vector of $V_k(s)$ values, do one ply of expectimax from each state:

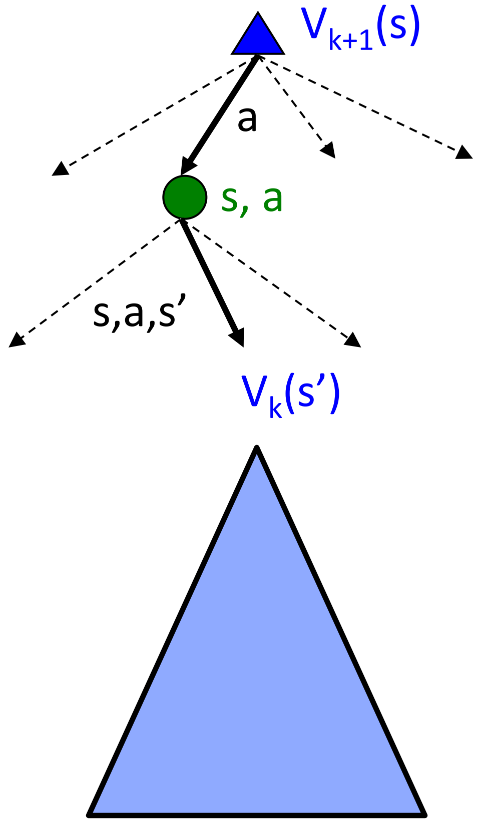
$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_k(s')\right]$$

Bellman Update Equation

- Repeat until convergence

- Complexity of each iteration: $O(S^2A)$

- Theorem: will converge to unique optimal values
  - Basic idea: approximations get refined towards optimal values
  - Policy may converge long before values do



Policy Iteration

- Alternative approach for optimal values:
  - Step 1: Policy evaluation: calculate utilities for some fixed policy (not optimal utilities!) until convergence
  - Step 2: Policy improvement: update policy using one-step look-ahead with resulting converged (but not optimal!) utilities as future values
  - Repeat steps until policy converges

- This is policy iteration
  - It's still optimal!
  - Can converge (much) faster under some conditions

Policy Iteration

- Evaluation: For fixed current policy π, find values with policy evaluation:
  - Iterate until values converge:

$$V_{k+1}^{\pi_i}(s) \leftarrow \sum_{s'} T(s, \pi_i(s), s') \left[R(s, \pi_i(s), s') + \gamma V_k^{\pi_i}(s') \right]$$

- Improvement: For fixed values, get a better policy using policy extraction
  - One-step look-ahead:

$$\pi_{i+1}(s) = \arg\max_a \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma V^{\pi_i}(s') \right]$$

  - What about Q-values?

Monte Carlo Value Estimation

- Use actual *experience* of interactions with the environment.
  - Environment could be the real world or a simulation.
  - Building a simulator is often easier than fully specifying T(s,a,s')
  - Works with continuous states and actions

$e_1 = (s_0, a_0, r_0, s_1, a_1, r_1, s_2, a_2 \ldots\ldots)$, with $s_1 = s'$

$$V(s') = \frac{1}{N} \sum_{i=1}^{N} \sum r_i$$

Initialize:

$\pi \leftarrow$ policy to be evaluated

$V \leftarrow$ an arbitrary state-value function

$Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Repeat forever:

(a) Generate an episode using π (rollout)

(b) For each state s appearing in the episode:

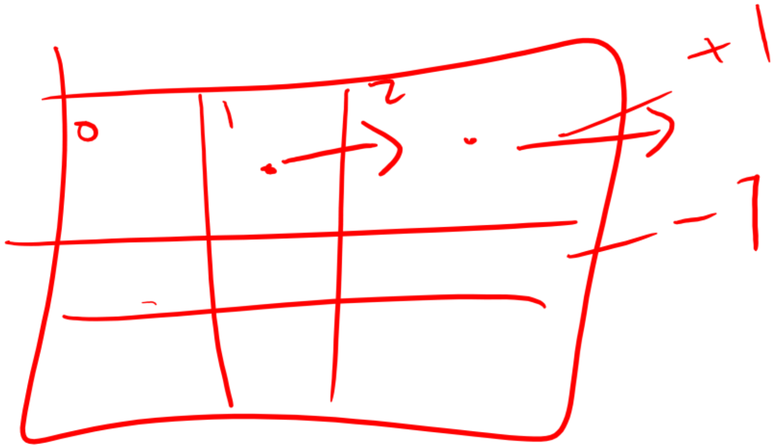
$R \leftarrow$ return following the first occurrence of s

Append R to $Returns(s)$

$V(s) \leftarrow \text{average}(Returns(s))$

Example

- Estimate the value of a state V(s) given a policy π without complete knowledge of the transition function T



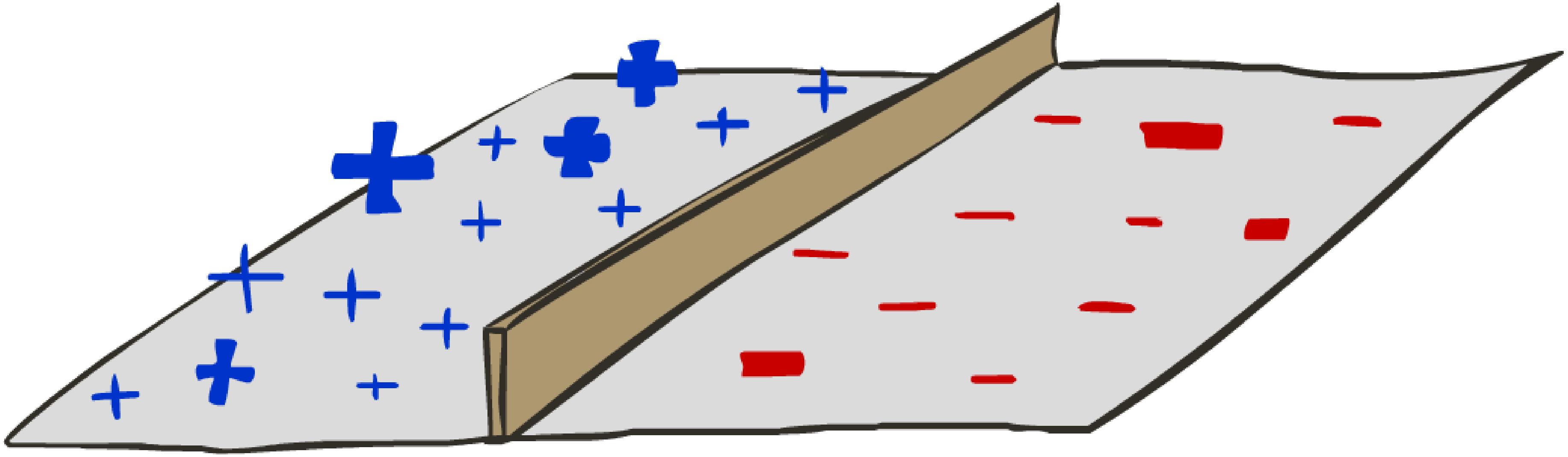
$$(s_1, \rightarrow, 0, s_2, \rightarrow, +1, s_{done})$$

$$V^{\pi_\rightarrow}(s_1) = 0 + \gamma \cdot 1 = \gamma 1$$

Types of Machine Learning

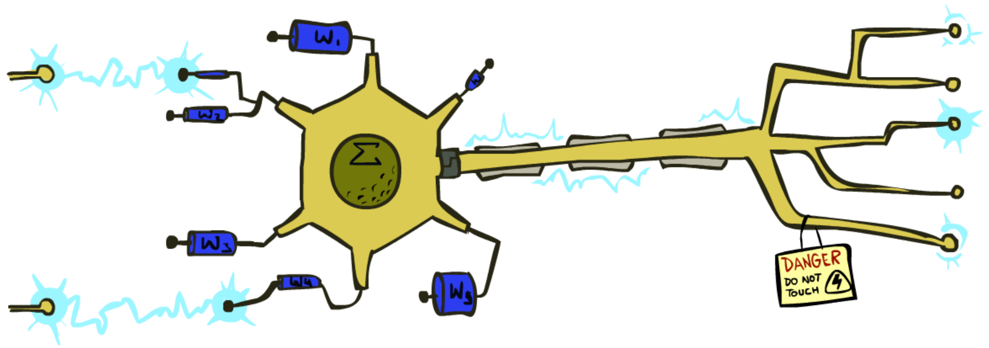
- Supervised Learning
  - Classification
  - Regression

Decision Rules



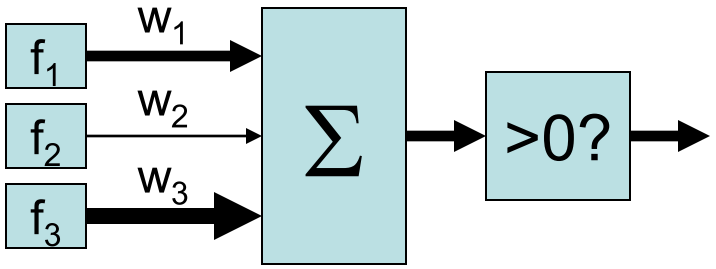
Linear Classifiers

- Inputs are feature values
- Each feature has a weight
- Sum is the activation



$$\text{activation}_w(x) = \sum_i w_i \cdot f_i(x) = w \cdot f(x)$$

- If the activation is:
  - Positive, output +1
  - Negative, output -1



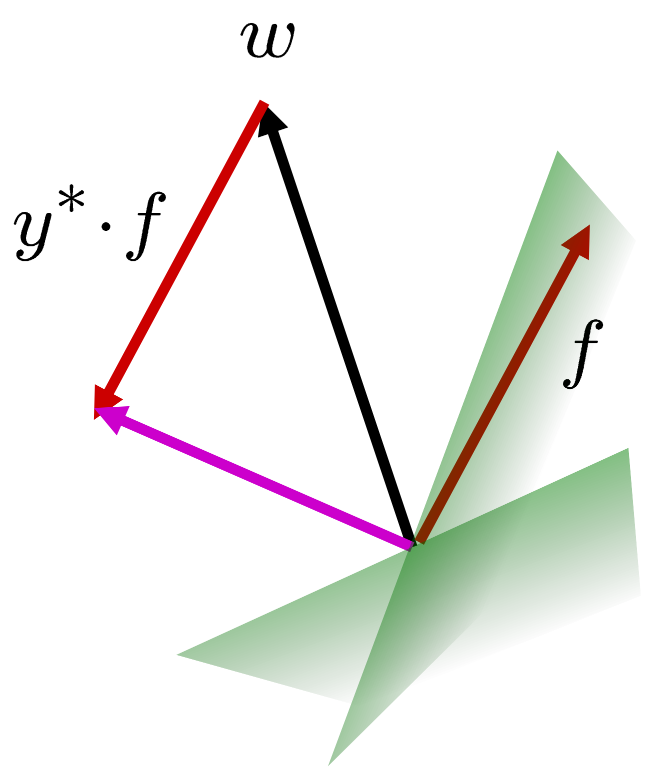
Learning: Binary Perceptron

- Start with weights = 0
- For each training instance:
  - Classify with current weights

$$y = \begin{cases} +1 & \text{if } \ w \cdot f(x) \geq 0 \\ -1 & \text{if } \ w \cdot f(x) < 0 \end{cases}$$

  - If correct (i.e., y=y*), no change!
  - If wrong: adjust the weight vector by adding or subtracting the feature vector. Subtract if y* is -1.

$$w = w + y^* \cdot f$$



Before update $w^T f(x) > 0$ and $y^* = -1$

After update
$(w - f(x))^T f(x) = w^T f(x) - f(x)^T f(x) < w^T f(x)$

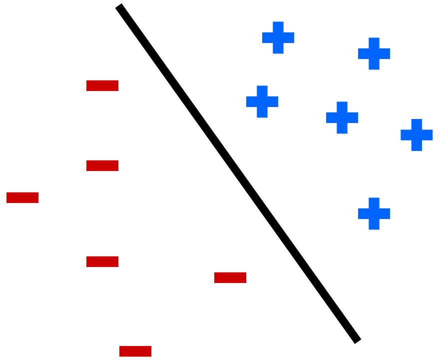
Properties of Perceptrons

- Separability: true if some parameters get the training set perfectly correct

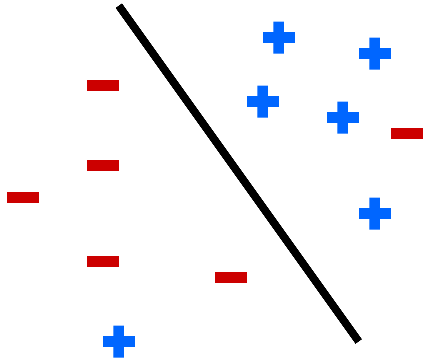
- Convergence: if the training is separable, perceptron will eventually converge (binary case)

- Mistake Bound: the maximum number of mistakes (binary case) related to the *margin* or degree of separability

Separable



Non-Separable

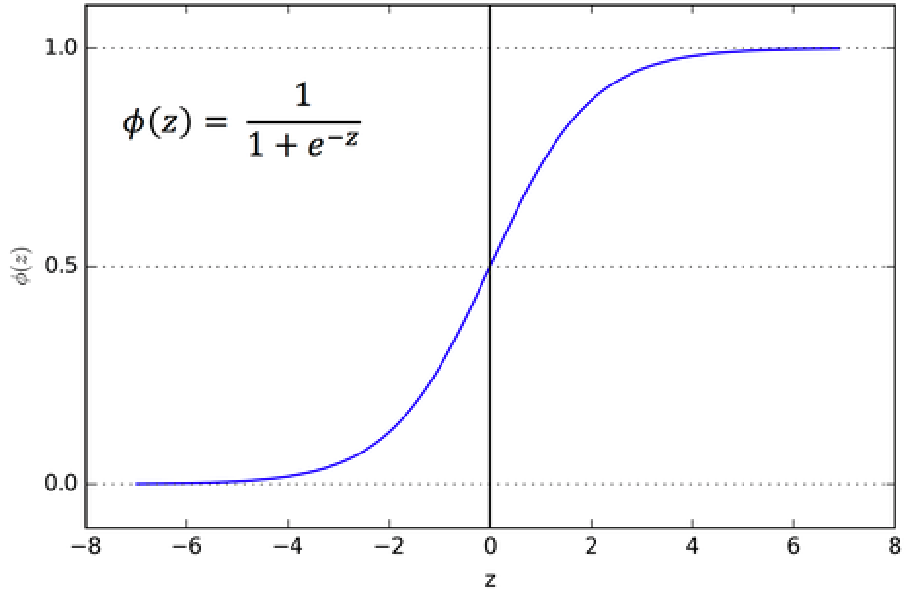


How to get probabilistic decisions?

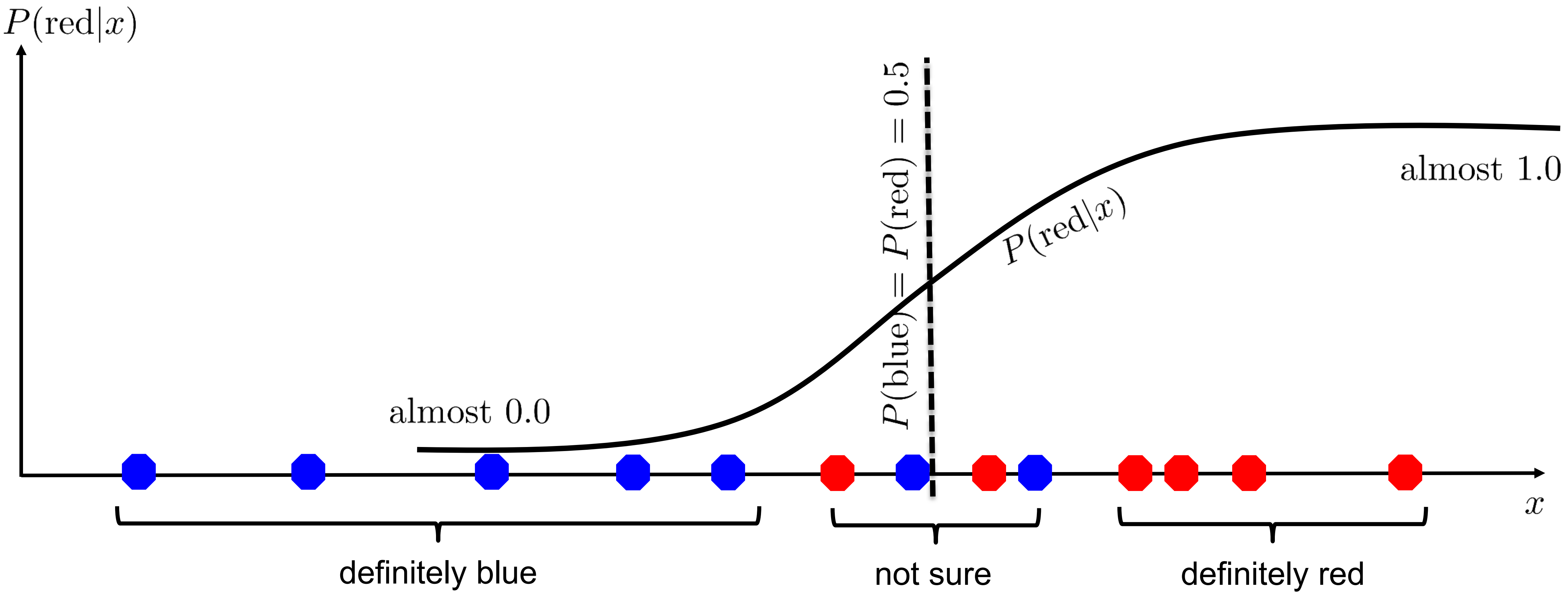
- Perceptron scoring: $z = w \cdot f(x)$
- If $z = w \cdot f(x)$ very positive → want probability going to 1
- If $z = w \cdot f(x)$ very negative → want probability going to 0

- Sigmoid function

$$\phi(z) = \frac{1}{1 + e^{-z}}$$



A 1D Example

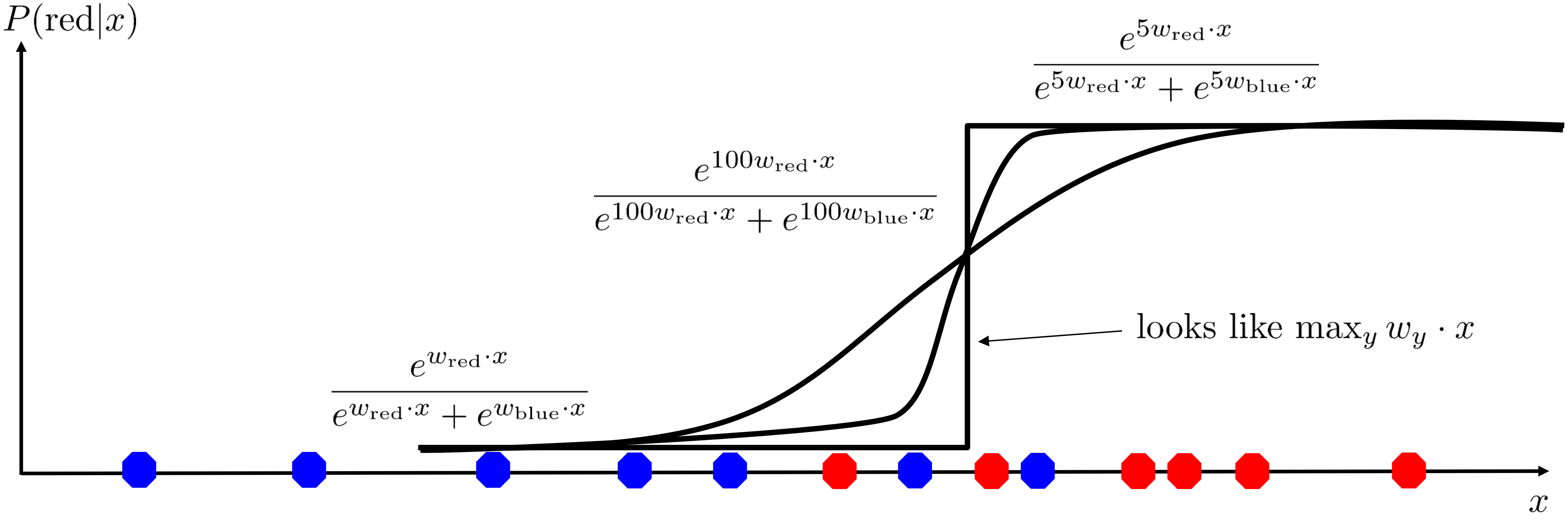


$$P(\text{red}|x) = \frac{e^{w_{\text{red}} \cdot x}}{e^{w_{\text{red}} \cdot x} + e^{w_{\text{blue}} \cdot x}}$$

probability increases exponentially as we move away from boundary (numerator)

normalizer (denominator)

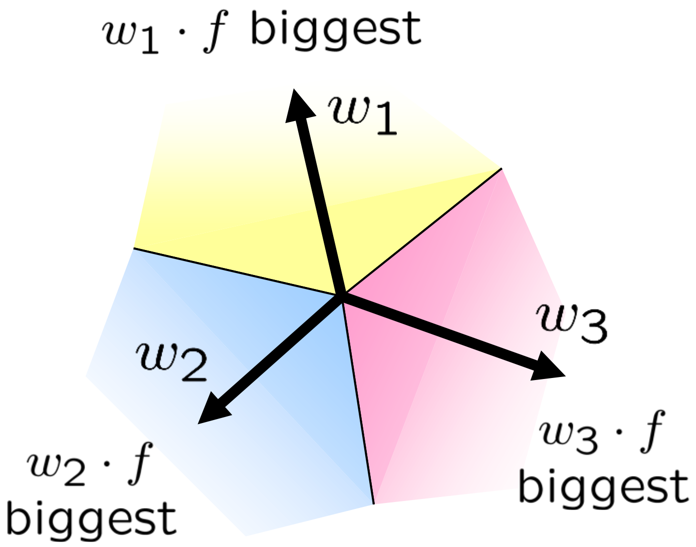
The *Soft* Max



$$P(\text{red}|x) = \frac{e^{w_{\text{red}}\cdot x}}{e^{w_{\text{red}}\cdot x}+e^{w_{\text{blue}}\cdot x}}$$

Multiclass Logistic Regression

- Recall Perceptron:
  - A weight vector for each class: w_y
  - Score (activation) of a class y: $w_y \cdot f(x)$
  - Prediction highest score wins $y = \arg\max_y \; w_y \cdot f(x)$



- How to make the scores into probabilities?

$$\underbrace{z_1, z_2, z_3}_{\text{original activations}} \rightarrow \underbrace{\frac{e^{z_1}}{e^{z_1} + e^{z_2} + e^{z_3}}, \frac{e^{z_2}}{e^{z_1} + e^{z_2} + e^{z_3}}, \frac{e^{z_3}}{e^{z_1} + e^{z_2} + e^{z_3}}}_{\text{softmax activations}}$$

Best w?

- Maximum likelihood estimation:

$$\max_w \quad ll(w) = \max_w \quad \sum_i \log P(y^{(i)}|x^{(i)}; w)$$

with:
$$P(y^{(i)}|x^{(i)}; w) = \frac{e^{w_{y^{(i)}} \cdot f(x^{(i)})}}{\sum_y e^{w_y \cdot f(x^{(i)})}}$$

= Multi-Class Logistic Regression

How do we learn in this setting?

- Optimization
  - i.e., how do we solve:

$$\max_w \quad ll(w) = \max_w \quad \sum_i \log P(y^{(i)}|x^{(i)}; w)$$

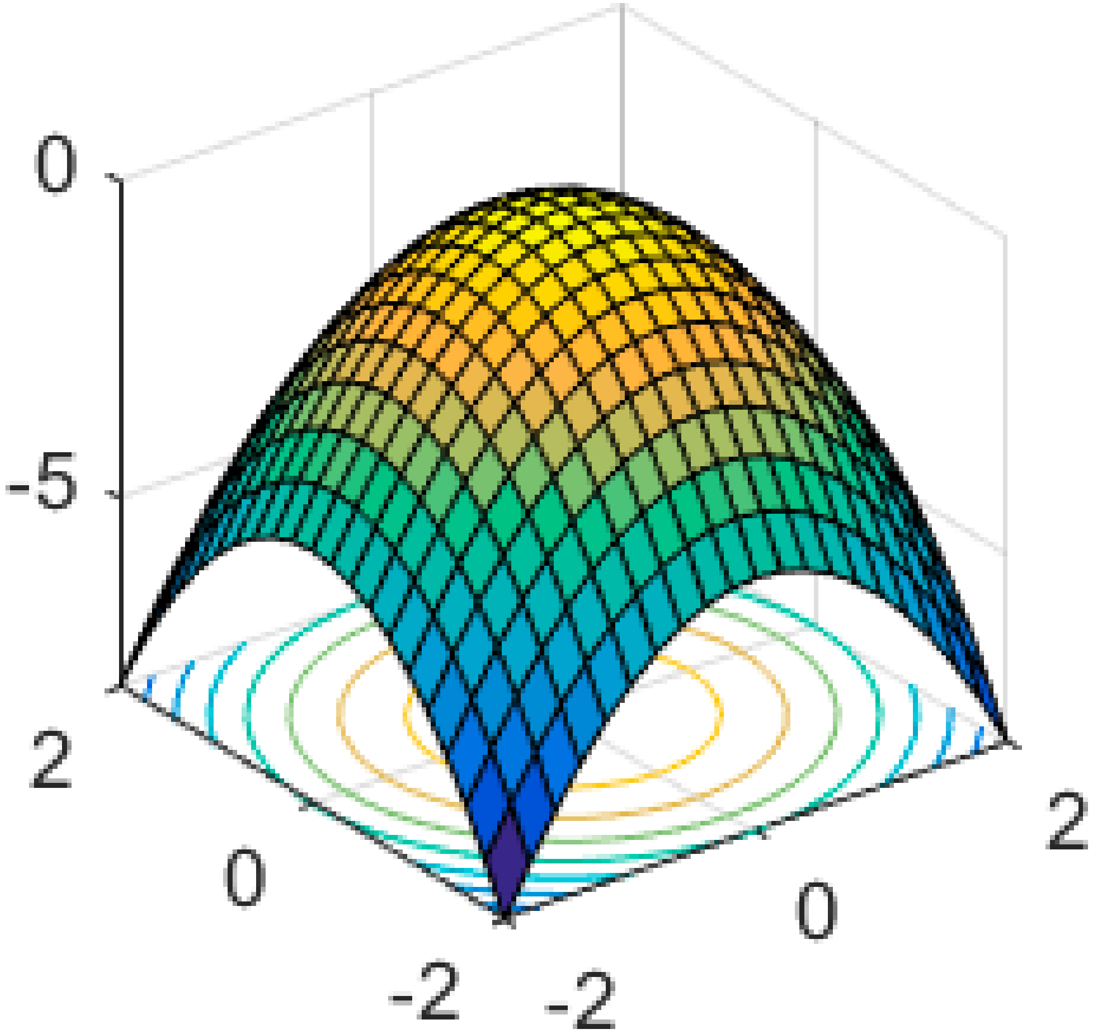
Linear Regression

- How can we measure how good a set of weights w are?
  - Mean Squared Error

$$\mathcal{L}_{MSE}(w) = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{2}(y_i - \hat{y}_i)^2 = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{2}\left(y_i - w^T f(x)\right)^2$$

$$\frac{\partial \mathcal{L}_{MSE}}{\partial w_i} = \frac{1}{N}\sum_{i=1}^{N}(w^T f(x) - y) f_i(x)$$

Optimization via Hill Climbing



Source: offconvex.org

Mini-Batch Gradient Ascent on the Log Likelihood Objective

$$\max_w \quad ll(w) = \max_w \quad \sum_i \log P(y^{(i)}|x^{(i)};w)$$

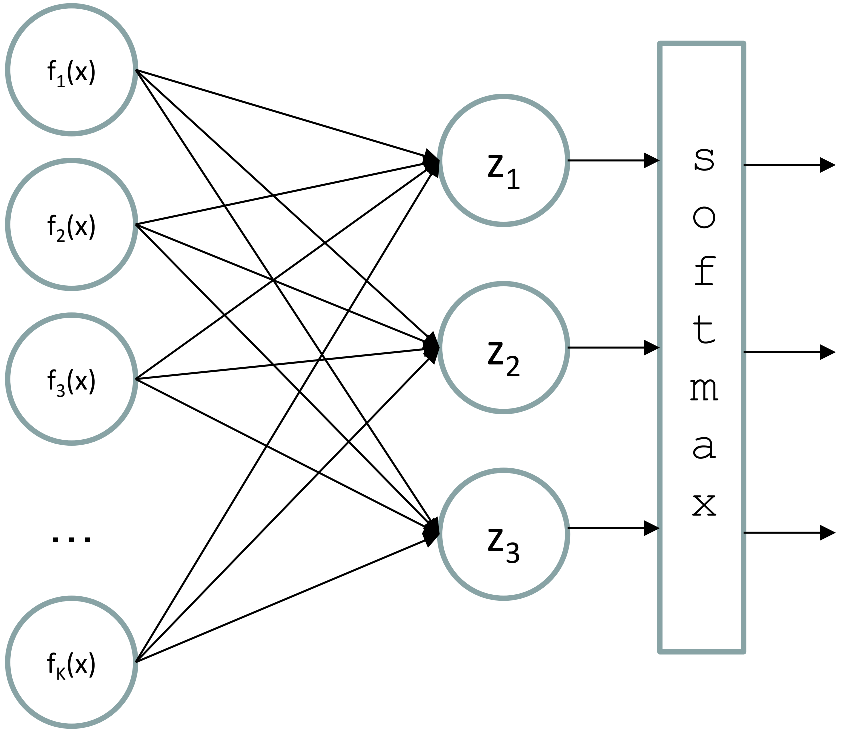
Observation: gradient over small set of training examples (=mini-batch) can be computed in parallel, might as well do that instead of a single one

```
init w
for iter = 1, 2, …
   pick random subset of training examples J
```

$$w \leftarrow w + \alpha * \sum_{j \in J} \nabla \log P(y^{(j)}|x^{(j)};w)$$

Multi-class Logistic Regression

special case of neural network

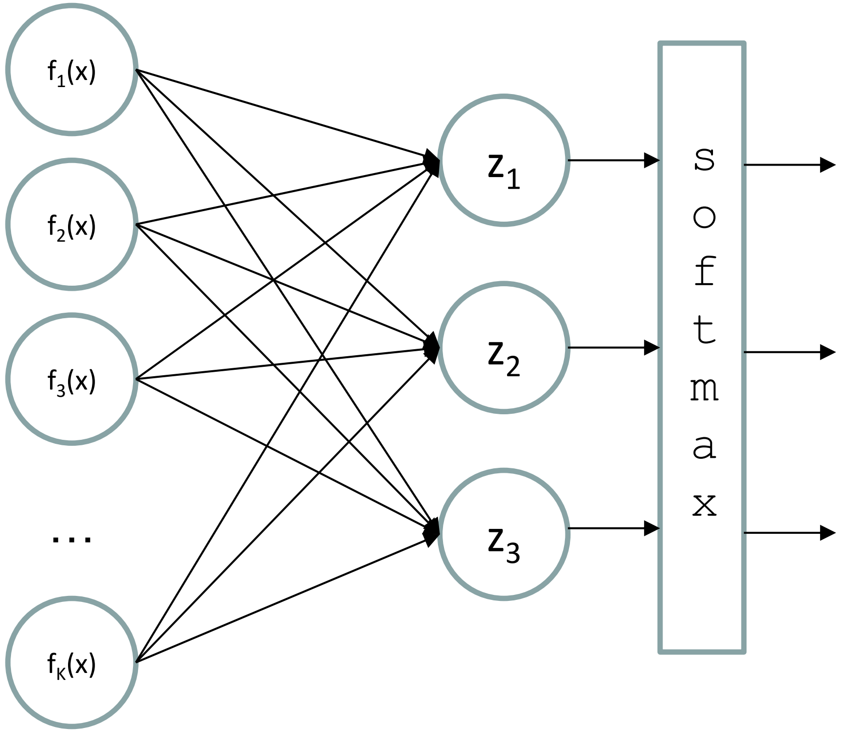


$$P(y_1|x;w) = \frac{e^{z_1}}{e^{z_1} + e^{z_2} + e^{z_3}}$$

$$P(y_2|x;w) = \frac{e^{z_2}}{e^{z_1} + e^{z_2} + e^{z_3}}$$

$$P(y_3|x;w) = \frac{e^{z_3}}{e^{z_1} + e^{z_2} + e^{z_3}}$$

Deep Neural Network = Also learn the features!

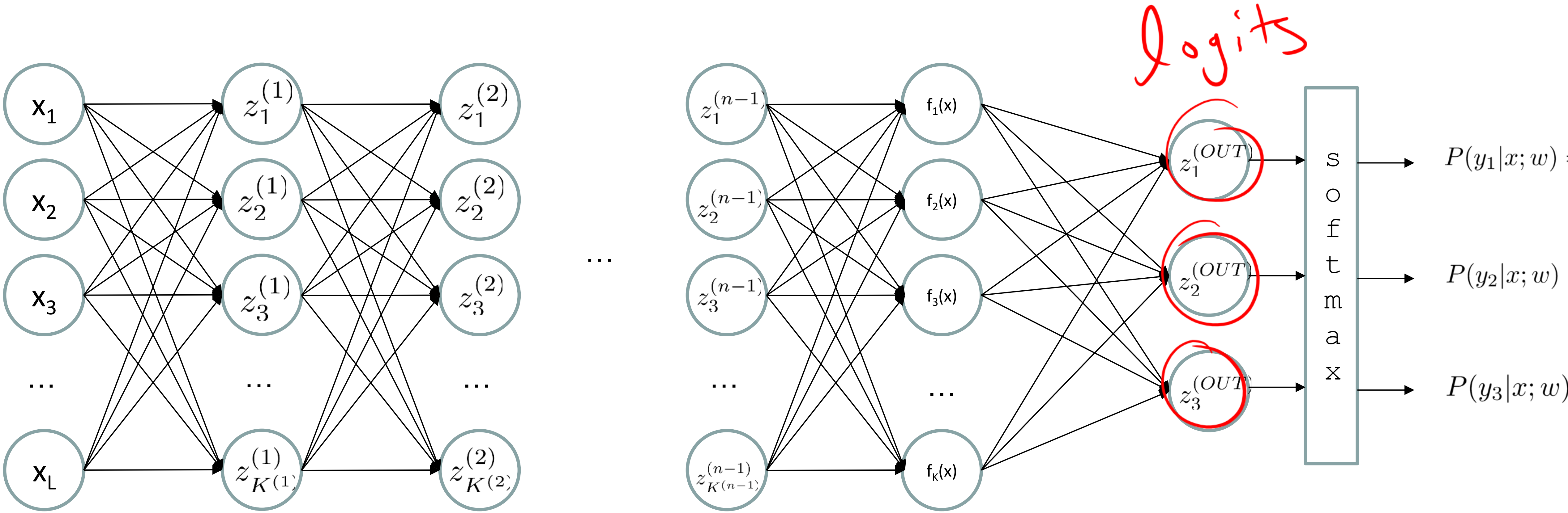


$$P(y_1|x;w) = \frac{e^{z_1}}{e^{z_1} + e^{z_2} + e^{z_3}}$$

$$P(y_2|x;w) = \frac{e^{z_2}}{e^{z_1} + e^{z_2} + e^{z_3}}$$

$$P(y_3|x;w) = \frac{e^{z_3}}{e^{z_1} + e^{z_2} + e^{z_3}}$$

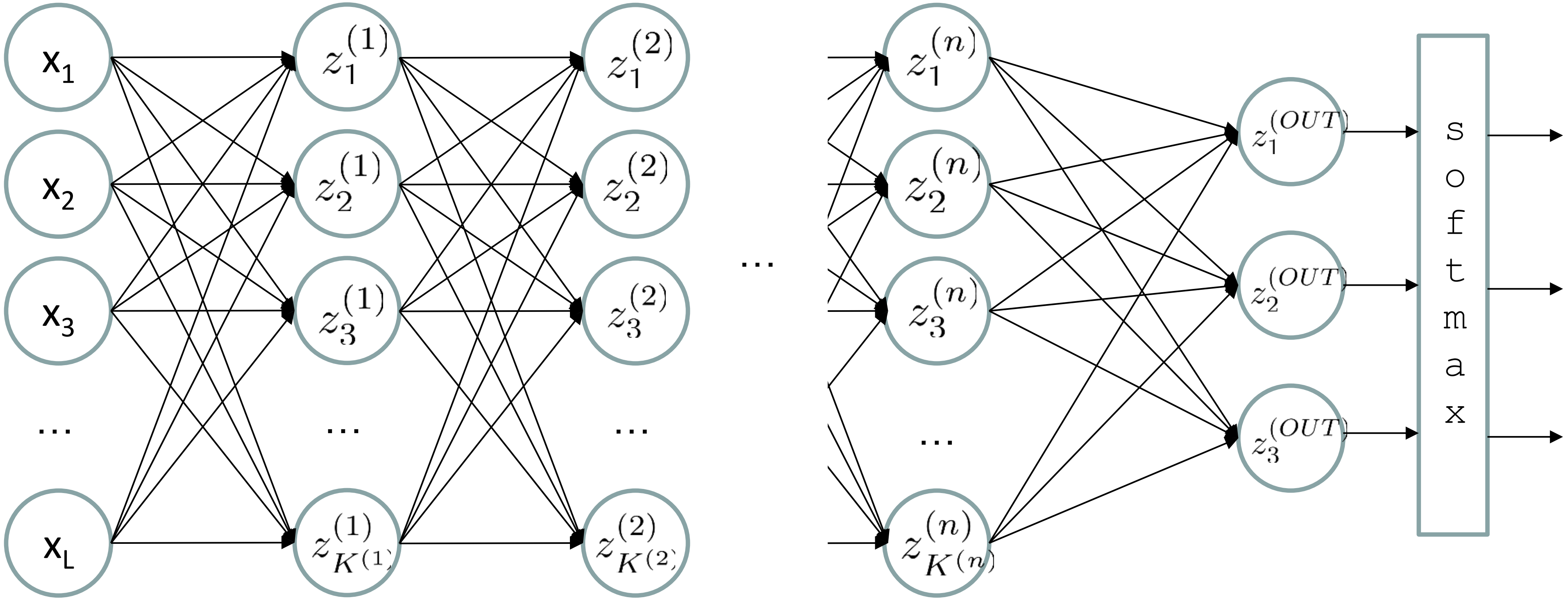
Deep Neural Network = Also learn the features!



$$z_i^{(k)} = g\left(\sum_j W_{i,j}^{(k-1,k)} z_j^{(k-1)}\right)$$

g = nonlinear activation function

Deep Neural Network = Also learn the features!



$$P(y_1|x;w) = \frac{e^{z_1}}{e^{z_1} + e^{z_2} + e^{z_3}}$$

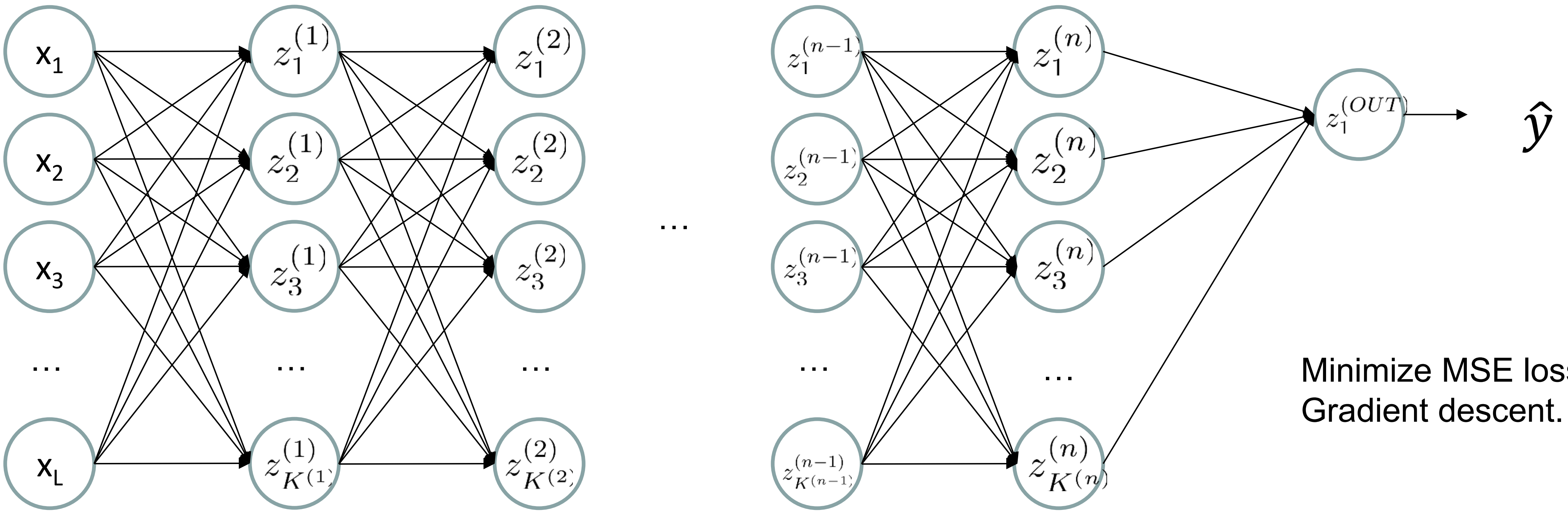
$$P(y_2|x;w) = \frac{e^{z_2}}{e^{z_1} + e^{z_2} + e^{z_3}}$$

$$P(y_3|x;w) = \frac{e^{z_3}}{e^{z_1} + e^{z_2} + e^{z_3}}$$

$$z_i^{(k)} = g(\sum_j W_{i,j}^{(k-1,k)} z_j^{(k-1)})$$

g = nonlinear activation function

Deep Neural Networks for Regression



Minimize MSE loss via Gradient descent.

$$\mathcal{L}_{MSE}(w) = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{2}(y_i - \hat{y}_i)^2 = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{2}\left(y_i - w^T f(x)\right)^2$$

Deep Neural Networks for Regression