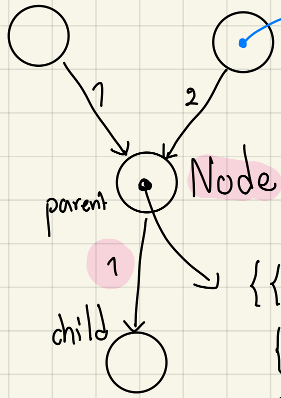
↳ 1. Problem Souling Agent

↳ ไปหาอ่านในไฟล์ case study เอาเขียนเองแล้วเรื่อย

↳ searching → เพื่อค้นหาคำตอบของปัญหาเรา

↳ infrastructure for search algorithms



ในการค้นหาคำตอบแต่ละครั้ง → เราจะเก็บข้อมูลไว้ใน node

Node → มี action เก็บเอาไว้ : เมื่อทำ action A จาก parent จะได้ node นี้

state

{{1, 2, 3},
{5, 0, 6}
{7, 9, 8}} → รับข้อมูลแล้วแต่เราออกแบบ

↓

เป็น 2dim หรือ text ก็ได้

123506798

↳ Measuring problem-solving performance

↳ สามารถวัดได้ 4 ข้อ → ① Completeness : ถ้ามีคำตอบต้องหาคำตอบเจอ

② Optimality : คำตอบที่หามาได้คุณภาพดีไหม → (ex) cost น้อย

③ Time Complexity : เวลาในการค้นหานานแค่ไหน

④ Space Complexity : หน่วยความจำที่ใช้

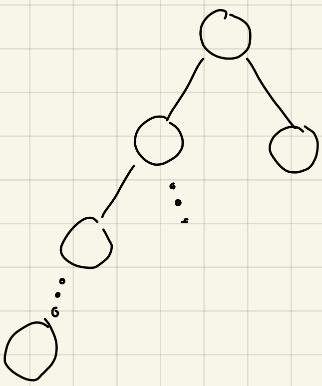
↳ ใน data structure :> |Edges|+|Vertices|

↳ ใน AI :> b (branching factor) แตกได้เท่าไหร่

d (depth) ลงไปกี่ชั้น

m (maximum length) ลึกสุดเท่าไหร่

↗ ถ้าไปเที่ยวทุก node

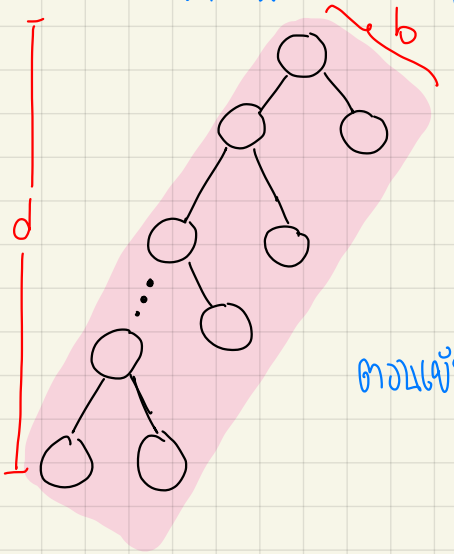


time complexity คือ $b^0 + b^1 + b^2 + b^3 + \dots = O(b^d)$

space complexity คือ $b^0 + b^1 + b^2 + b^3 + \dots = O(b^d)$

↳ node ที่เก็บไว้

↳ การค้นหาแบบลึกก่อน (dept-first search)



เหมือน ◇ ∴ space complexity: $O(bd)$ เก็บทุก node ที่อยู่ในเส้นทางที่ยาวที่สุด

time complexity: $O(b^d)$ ไปเที่ยวทุก node

ตอนเขียน code: ① check ว่าเป็น node นั้นไหม
② ถ้าถูกก็หยุด
③ ไปทางซ้าย
④ ไปทางขวา

1. Uninformed Search

intro

- มีหลักเหมือน 6 แบบก่อนหน้านี้
- แบ่งการ search เป็น 2 รูปแบบใหญ่ๆ ① Uninformed Search (Blind Search) ← ไม่มีข้อมูลเพิ่มเติมในการค้นหา
  ② Informed Search (Heuristic Search)
  - มีข้อมูลเพิ่มในการค้นหา

Uninformed Search Strategies

① Breath first Search (แบบกว้าง)

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the shallowest node in frontier */
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
        frontier ← INSERT(child, frontier)
```

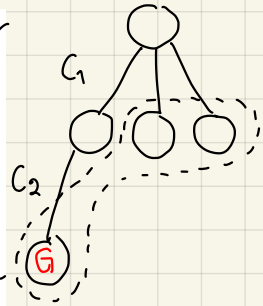
- evaluate
  - completeness : yes
  - optimality : yes (in term of step) → ถ้า path cost ไม่เท่ากัน → อาจจะเจอ cost เยอะ
  - time complexity : $O(b^d)$ → ไปเกี่ยวทุก node
  - space complexity : $O(b^d)$

  } เมื่อความลึกเพิ่มขึ้นเรื่อยๆ จะโตแบบ expo

② Uniform cost search

- ก้าวมาเพื่อแก้ปัญหาของ BFS → ที่ cost ไม่เท่า BFS หาสุดเอาเลย → เจอที่ถี่สุด

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        frontier ← INSERT(child, frontier)
      else if child.STATE is in frontier with higher PATH-COST then
        replace that frontier node with child
```



} frontier ต้องไปทั่ว frontier เลย

- evaluate
  - Completeness : yes
  - Optimality : yes เลือกทางที่ cost ต่ำที่สุด
  - Time complexity : $O(b^{1+\lfloor C^*/\epsilon \rfloor})$; C^*/ϵ = จำนวนชั้นที่ลงไป ← โดย ϵ ชั้นเท่ากันเท่ากัน
  - Space complexity : $O(b^{1+\lfloor C^*/\epsilon \rfloor})$

↳ ③ Depth First Search

↳ ไป node ที่ลึกที่สุดก่อน, ใช้ stack ในการ implement

↳ เจอเร็วมากเมื่อเทียบกับวิธีอื่น

↳ evaluate

- ↳ Completeness : No → หลงวนในกรณี state ซ้ำ

ไหลยาว

- ↳ Optimality : No → ไม่ได้เจอ cost น้อยที่สุด
- ↳ Time Complexity : $O(b^d)$
- ↳ Space Complexity : $O(bd)$

↳ ④ Depth Limited Search

↳ มาเพื่อแก้ปัญหา DFS, ใช้ได้ดีเมื่อรู้ maximum solution depth

↳ ทำการกำหนด cutoff → เมื่อมันยาวเกินไป

↳ Evaluate

- ↳ Completeness : Yes → if $l \geq$ maximum dept of solution (ต้องเลือก l)
- ↳ Optimality : No
- ↳ Time Complexity : $O(b^l)$ → l ความยาวที่ไหลไปลึก คูณกับจำนวนกิ่งที่แตก
- ↳ Space Complexity : $O(bl)$

↳ ⑤ Iterative Deepening Search

↳ มาเพื่อแก้ปัญหา DLS, เปลี่ยนค่า l เป็น 0, 1, 2, ... จนกว่าจะถึง goal, l is vary

↳ เปรียบเสมือนทำ BFS ได้คุณสมบัติมาด้วย

↳ evaluate

- ↳ Completeness : yes → สืบทอด BFS
- ↳ Optimality : yes → ได้ BFS (เจอตื้น)
- ↳ Time Complexity : $(d)b + (d-1)b^2 + (d-2)b^3 + \dots + b^d = O(b^d)$ (d คือรอบที่ทำซ้ำตามความลึก)
- ↳ Space Complexity : $O(bd)$ → สืบทอดข้อดีมาจาก DFS

เมื่อ $b=10, d=5$

$N(IDS) \approx N(BFS)$

↳ note : ทุกที่มันเริ่มใหม่ทุกรอบ → เหมือนจะเปลืองแต่ ?ไม่เปลือง

↳ node ข้างบนจะมีจำนวนที่น้อยเมื่อเทียบกับตอนปลาย

↳ ⑥ Bidirectional Search

↳ 2 การรันคู่ขนานกันโดยจะหาจาก init → goal อีกฝั่งจะหา goal → init ต้องเผลอจะชน เลยต้องรู้ state ปัจจุบันของเลยกับ goal

↳ หยุดวิ่งเมื่อจะเจอกันสักทางหนึ่ง ถ้า set(init) ∩ set(goal) ไม่ใช่ {∅} → เจอคำตอบแล้ว

↳ เราสามารถใช้ breath-first search หรือ iterative deepening search ได้ เพื่อลด space complexity

↳ evaluate

↳ Completeness : Yes

Optimality : yes → ถ้า cost เท่ากัน, หรือเพิ่มแบบ function เชิงเดี่ยว

Time Complexity : $O(b^{\frac{d}{2}})$ → มาจากการไปเที่ยว 2 ฝั่ง $O(b^{\frac{d}{2}} + b^{\frac{d}{2}})$

Space Complexity : $O(b^{\frac{d}{2}})$ [BFS], $O(b\frac{d}{2})$ IDS } แล้วแต่ algo

Comparison of uninformed search strategies

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

Figure 3.21 Evaluation of tree-search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: [a] complete if b is finite; [b] complete if step costs $\geq \epsilon$ for positive ϵ; [c] optimal if step costs are all identical; [d] if both directions use breadth-first search.

2. Informed Search

↳ Informed Search Strategies

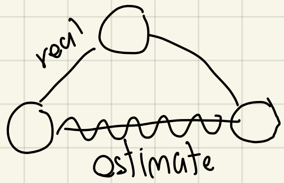
↳ เป็นการค้นหาที่อยู่บนพื้นฐานของ best first search → เลือกโหนดที่เราคิดว่าดีที่สุด จาก evaluation f f(n)

↳ f(n) = h(n) + g(n) ข้ามอ่านไปก่อน

↳ โดย h(n) คือ heuristic function เป็นเหมือน func(n) ที่มีข้อมูลเพิ่มเติมว่าไปหา goal ได้เร็วขึ้น

↳ เป็นค่า cost โดยประมาณจาก state ที่ n ใดๆ ไป goal → มันจะเดินทางไปน้อยที่สุด

↳ แต่จะต้องรู้ node ใน frontier เพิ่มด้วยเหมือน uniform cost search



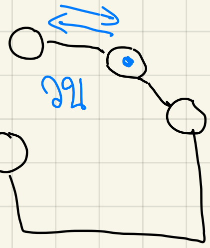
↳ ① Greedy best-first search

↳ f(n) = g(n) → การตัดสินใจผ่าน heuristic function อย่างเดียว

↳ g(n) ที่ดีจะไม่ให้ค่า cost ที่สูงเกินความเป็นจริง "overestimate"

↳ มันคือการที่เราเชื่อว่าดี → ไม่ได้หมายความว่าเป็นดีที่สุด!

↳ ไม่ optimal, incomplete อาจจะติด loop



↳ evaluation → completeness: no → ติด loop ค่า cost

optimality: no → แค่เราเชื่อว่าเป็นคือการที่ดีที่สุด

Time Complexity: $O(b^d)$

Space Complexity: $O(b^d)$

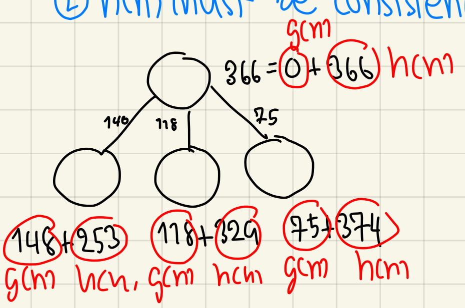
↳ ② A* search

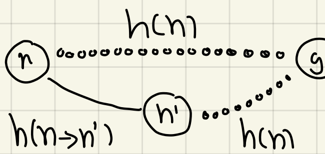
↳ ใช้กันอย่างแพร่หลาย $f(n) = h(n) + g(n)$; g(n) คือ actual cost

↳ เป็น algorithm ที่เอาไปใช้ประยุกต์ในหลายด้านด้วย → ทำให้ optimal & complete

↳ เงื่อนไขของ a* → ① h(n) must be admissible heuristic → ต้องไม่ over estimate ขึ้นเอง

② h(n) must be consistency



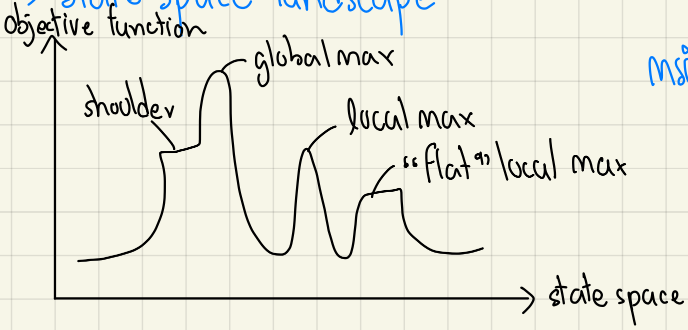


h(n) ต้อง < h(n→n') + h(n')

↳ Local Search and Optimization Problems

- ↳ เป็นการค้นหาบริเวณรอบๆ
- ↳ เกิดขึ้นเมื่อเราไม่สนใจ path ในการค้นหาขั้นๆ สนแค่คำตอบที่ได้
- ↳ มักจะใช้การขยับไปข้างๆ เรื่อยๆ
- ↳ ซึ่งได้เปรียบของวิธีนี้ (1) ใช้ memory น้อยมากๆ → ใช้การขยับไปด้านข้างเรื่อยๆ
  (2) ได้รับคำตอบที่สมเหตุสมผลยอมรับได้ (not the best)
- ↳ ใช้เมื่อต้องการหา state ที่ดีที่สุดตาม objective function
  - ↳ optimization problems → พวกโจทย์ปัญหาลงท้ายด้วยคำว่า "ที่สุด"

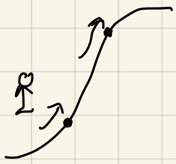
↳ State-space landscape



การค้นหาที่ดีคือการหา global max

↳ Hill-climbing Search

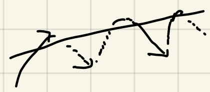
↳ return state ที่เป็น local maximum → อาจจะไม่เจอ global max



```
current ← make-node(problem.initial-state)
loop do:
    neighbor ← highest-valued successor of current
    if neighbor.value ≤ current.value then return current.state
    current ← neighbor
```

(highest-valued successor of current → หา successor (ลูก))
(current ← neighbor → ไปที่สูงกว่า only ↳ ถ้าต่ำกว่า state ปัจจุบัน จะไม่ไป)

- ↳ มันจะ random เลือกหากมี best successor มากกว่า 1
- ↳ บางครั้งถูกเรียกว่า "greedy local search"
- ↳ แต่อย่างไรก็ตามจะมีข้อเสียอยู่
  - ↳ (1) local maxima: มีที่ max หลายที่ อาจจะไม่เจอ global max
    (2) ridges (สันเขา): พยายามยากมากเพื่อเลือกที่ดีขึ้นนิดเดียว



(3) plateaux (พื้นที่ราบ): หา max ไม่เจอจริงๆ

↳ Simulated Annealing

↳ การจำลองการอบอ่อน → เอาโลหะมาทำให้เป็นของเหลวด้วย T มากๆ แล้วค่อยๆ ลด T ลงทีละนิด

↳ ใช้ algorithm นี้ในการ search

(problem → ปัญหา, schedule → map(t : T))

```
func simulated-annealing(problem, schedule) return a solution state
current ← make-node(problem.initial-state)
for t=1 to ∞ do
    T ← schedule(t)
    if T = 0 then return current
    next ← a randomly selected successor of current      (random เลือก)
    ΔE ← next.value − current.value
    if ΔE > 0 then current ← next       (สูตรหาไปเลย)
    else current ← next only with probability e^(ΔE/T)
```

↳ prob ก็คือที่เรากำหนดเช่น (เอา / ไม่เอา, $p=0.4$)

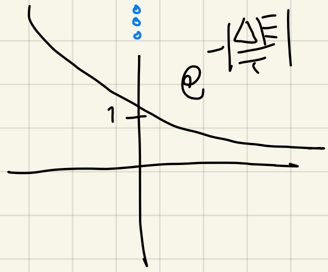
schedule

1 : T_max

2 : $T\_max(0.98)^1$

3 : $T\_max(0.98)^2$

⋮



$e^{-|\frac{\Delta E}{T}|}$

เมื่อ T เยอะ $e^{-0} = 1$

เมื่อ T น้อย $e^{-\infty} \approx 0$

↳ หลังๆโอกาสที่ไปจุดต่ำกว่าน้อยลง

↳ Local Beam Search

↳ เป็นการทำงานแบบใช้ agent ทั้งหมด k ตัว แทนที่ปกติเราใช้ 1 ตัว

↳ ถ้าเจอ goal ใช้ตัวที่ดีที่สุด

↳ Local Search In Continuous Spaces

↳ ที่ผ่านมาทั้งหมด → discrete space → branching factor มีจำกัด

↳ {hill climbing, simulated annealing} → ไม่รองรับ continuous space, action space

↳ มี infinite branching factor

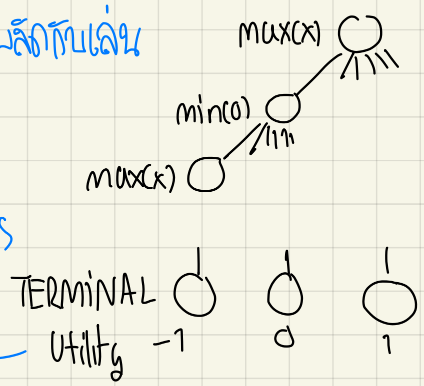
↳ ใช้หลักการของ calculus → หา gradient → optimizer with learning rate

↳ MINMAX Game

↳ เป็นการเล่นเกมส์ของคน 2 คนที่ต้องการเอาชนะกัน

↳ เช่น MIN ก็พยายามทำให้ค่าของตัวเองน้อยที่สุด, MAX ก็ตรงกันข้าม

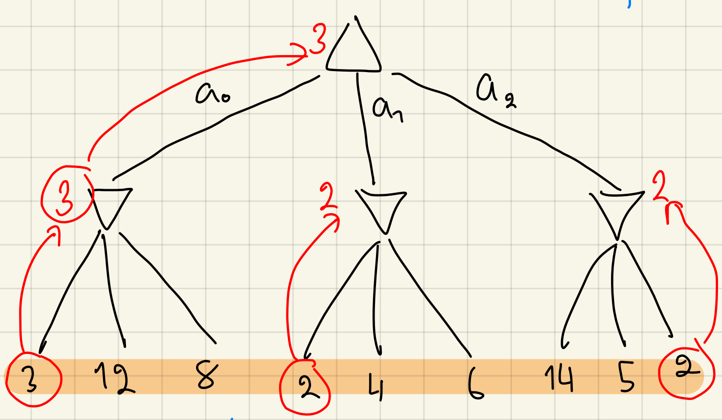
↳ เกมส์ XO ที่ผลัดกันเล่น



เรากำหนดเอง

คะแนนของเกมส์ ← Utility

ถ้าเรามี tree ที่สมบูรณ์ → ใช้วิธีการแบบ "Optimal Decision in Game"



ดูค่าสถานะ แต่ละ state เลือกค่าต่ำสุด/สูงสุด ของตัวเอง

เขียนเป็นสูตรได้ minimax(s) for each s step

$$\text{minimax}(s) = \begin{cases} \text{utility}(s) & \text{if terminal-test}(s) \\ \max_{a \in Actions} \text{minimax}(\text{result}(s,a)) & \text{if player}(s) = \max \\ \min_{a \in Actions} \text{minimax}(\text{result}(s,a)) & \text{if player}(s) = \min \end{cases}$$

จากตัวอย่างจะเป็น $\text{minimax(root)} = \max(\min(3,12,8), \min(2,4,6), \min(14,5,2))$

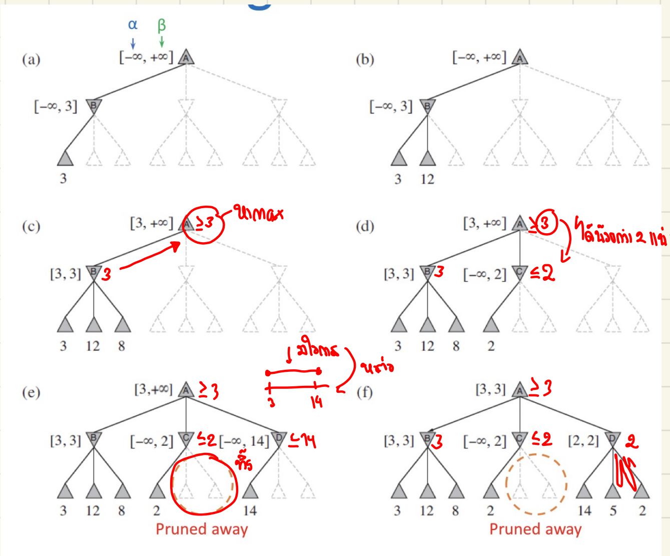
$= \max(3,2,2)$

$= 3$

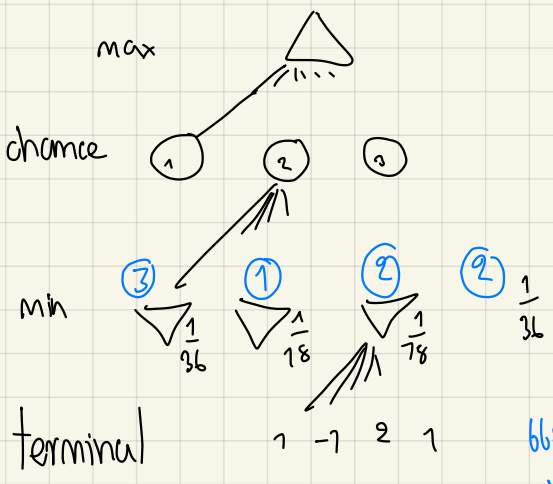
- time complexity $O(b^m)$
  space complexity $O(bm)$

- optimal decision in multiplayer games
  - ถ้ามีมากกว่า 2 คน ก็เปลี่ยนจาก min max ก็จะมีตามผู้เล่น utility → (1,2,3) เป็นคะแนนทั้ง n คนเลย

- Alpha-Beta Pruning
  - ใช้กับ dfs only
  - alpha → ค่าที่ดีที่สุดที่เคยเจอสำหรับ maximizer, beta → ค่าที่ดีที่สุดสำหรับ minimizer



↳ ถ้าเป็นเกมส์ที่มีการสุ่มเกี่ยวข้องด้วยนะ



Expectiminimax(s) =

$$\begin{cases} \text{utility}(s) & ;\text{if terminal-test}(s) \\ \max_a \text{expectiminimax}(\text{result}(s,a)) & ;\text{if player}(s)=\max \\ \min_a \text{expectiminimax}(\text{result}(s,a)) & ;\text{if player}(s)=\min \\ \sum_r P(r)\,\text{expectiminimax}(\text{result}(s,r)) & ;\text{if player}(s)=\text{chance} \end{cases}$$

แล้วค่าของค่าของ chance อยู่ไง

เช่น

ค่าของ chance ที่ min เลือกได้ =

$$② = 3\frac{1}{36} + 1\frac{1}{18} + 2\frac{1}{18} + 2\frac{1}{36}$$

3. Logic agent

↳ knowledge-based agents

↳ เป็นที่เก็บข้อมูลแบบไหนก็ได้ or text file

↳ มีการเก็บกฎของปัญหานั้นเอาไว้

↳ มีการเก็บ fact ไว้ (เพิ่มขึ้นมาเรื่อยๆ จากการทำงานไปเรื่อยๆ) ⟶ sentence vs fact

sentence

↳ การเพิ่ม sentence เข้า KB

(sentence → เครื่องจักรเข้าใจ ← convert ← คนเข้าใจ ← fact)

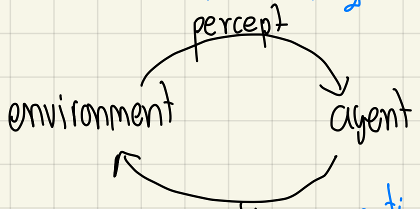
↳ เหมือนกัน

↳ method ที่สำคัญ tell, ask

↳ การอ่านความรู้จาก KB

↳ การเรียนรู้ เหมือนเด็กที่เราต้องทำการบอกกฎก่อน → (tell₁) เด็กเริ่มเรียนรู้เองจาก env ด้วยความรู้ที่มีอยู่ → รู้ว่าทำไร (ask₁ จาก KB) → จำ (tell₂)

action



→ action might change env or percept

↳ เคส study: wumpus world (ไปอ่านเอาในสไลด์ ขก อธิบายอีก)

↳ คือถ้าเป็นคน คนจะมี logic การหาที่ตั้ง wumpus, หลุม ว่าอยู่ตรงไหน

↳ จะทำอย่างไรให้ คอมพ์ เข้าใจได้แบบนี้

↳ กระบวนการ reasoning

↳ การที่เราจะถอด logic ต่อไป

↳ คือ การสร้าง sentence ใหม่จาก sentence เดิมที่มีอยู่ โดยของใหม่ที่ขึ้นต้องไม่ขัดแย้งกับของเก่า

Representation	$Sentence_{old}$	$\xrightarrow{\text{entail}}$	$Sentence_{new}$
	⇵ semantic		⇵ semantic
World	Aspect of Real $World_{old}$	$\xrightarrow{\text{follow}}$	Aspect of Real $World_{new}$

↳ Propositional logic: Very Simple Logic

↳ syntax: กฎต่างๆที่กำหนดมาเพื่อใช้เขียน sentence

↳ atomic sentence

↳ แบ่ง sentence ย่อยออกไปไม่ได้แล้ว

↳ เช่น symbol → $W_{1,3}$: wumpus อยู่ตำแหน่ง (1,3) → แยกเป็น W_1 ไร้ความหมาย

↳ complex sentence

↳ การนำ atomic sentence มาเชื่อมกันด้วย logic connectives $\{\neg, \wedge, \vee, \rightarrow, \leftrightarrow\}$ (สูง — ลำดับ — ต่ำ →)

↳ Sematics

↳ การอธิบายค่าความหมายของ syntax → may be in term of truth table

↳ กฎ 7 ข้อช่วยสรุปได้ง่ายขึ้น $\frac{\text{ที่เรารู้}}{\text{สรุป}}$

↳ $\dfrac{\alpha \to \beta, \alpha}{\beta}$ $\dfrac{\alpha_1 \land \alpha_2 \land \dots \land \alpha_n}{\alpha_i}$ $\dfrac{\alpha_1, \alpha_2, \dots \alpha_n}{\alpha_1 \land \alpha_2 \land \alpha_3 \dots}$ $\dfrac{\alpha_i}{\alpha_1 \lor \alpha_2 \lor \dots}$ $\dfrac{\sim\sim\alpha}{\alpha}$

$\dfrac{\alpha \lor \beta, \sim\beta}{\alpha}$ $\dfrac{\alpha \lor F}{\alpha}$ $\dfrac{\alpha \lor \beta, \sim\beta \lor \gamma}{\alpha \lor \gamma}$ $\sim\alpha \to \beta,\ \beta \to \gamma$

$\sim\alpha \to \gamma$

$\alpha \lor \gamma$

↳ การทำงานคร่าวๆ env $\xrightarrow{\text{percept}}$ maker sentence $\xrightarrow{\text{sentence}}$ KB

↳ การที่เราจะได้ sentence ใหม่ → มาการ query เอาค่าความจริง + ใช้กฎด้านบนมาแบบเข้มๆ

↳ พอได้ sentence ใหม่มาปุ๊บ → เช็คของเก่าด้วย → ถ้าขัดก็ลบทิ้ง

↳ กฎการตัดสินใจไปไหนอยู่ตรงไหน แล้วเราจะ take action อย่างไร?

↳ ต้องมีกฎสำหรับ action ด้วยขึ้นเอง

↳ ปัญหาของ Propositional Logic

↳ ① ต้องสร้างกฎขึ้นมารองรับทุกสถานการณ์ ถ้าเป็น wumpus world เมื่อกี้ $\frac{16}{\text{ช่อง}} \times \frac{4}{\text{ทิศ}}$ (↑ แต่ของจริง wumpus อาจย้ายไปข้างหน้า)

② ใช้กฎเดิมเมื่อเวลาผ่านไปไม่ได้ → env สามารถเปลี่ยนแปลงหรือ กลับมาที่เดิมด้วยเวลาที่เปลี่ยนไป

4 First Order Logic

- First Order Logic
  - propositional logic กระจัดกระจายในการแทน fact แบบกระชับ → ต้อง define กฎเยอะเยอะ
  - fol อธิบายกฎได้ดีกว่า → อธิบายคุณสมบัติของ object ได้ (การรวมกฎย่อย → กฎหลัก) [แทนที่เราจะไปทำทีละตัว
- FOL Syntax and Semantics
  - Symbol
    - Constant symbol : object
    - Predicate symbol : relation
    - Function symbol : function that return object
  - Term : logical expression that refer to object (เอา symbol มารวมกัน)
  - Atomic sentence : represent a relationship between object — Broter(x, y) (pre) (con) (con)
  - Complex sentence : logic connectives of atomic sentence — king(o) → king(เอ๋อ)
  - โดยหลักๆ ก็เหมือนกันเลย แค่จะมีเพิ่ม quantifier
- Quantifier
  - ตัวที่ express the properties of entire collection of objects
  - ① Universal Quantification (∀)

    $\forall_x \, Cat(x) \rightarrow mammal(x)$ [∧] → ถ้าไม่มีแมวเลย F → ▨ เป็นจริงเสมอ เหมือนเดิม

    [T → mamal T] ∧ [F → ▨] = T — ทุกตัวตามชื่อ for all

    สำหรับ object x ทุกตัวที่อยู่ใน domain, ถ้า x เป็นแมว แล้ว x จะเป็น mammal

    - ถ้าเราเขียนผิดแล้วจะ "แมวทุกตัวคือ mammal" $\forall_x Cat(x) \wedge Mammal(x)$ → [Cat(King) ∧ ...] ∧ [Cat(Meow) ∧ ...] ∧ จะได้ T ∧ F ≈ F เสมอ ↓ เป็นเท็จเสมอ
      - apply ทุก obj ใน domain

  ② Existential Quantification (∃)

  $\exists_x \, sister(x, spot) \wedge cat(x)$ [∨] → [T ∧ mamml] ∨ ... = T — ขอแค่มีตามชื่อ for some

  มีพี่สาวของ spot และเป็นแมว → ถ้าไม่มีเลย [F ∧ ▨] ∨ [F ∧ ▨] ... ก็เป็นเท็จแล้ว

  - ถ้าเราเขียนผิด $\exists_x \, sister(x, spot) \rightarrow cat(x)$ → [T → □] ∨ [F → ▨] ∨ ไปจริง → เป็นจริงเสมอ

  ③ Nested Quantification → รวมมาก ∀ ∃ เข้าด้วยกัน
  - ถ้า $\forall_x$ จะประมาณว่า "สำหรับทุกๆ x, ..."

    ถ้า $\exists_x$ จะประมาณว่า "มันจะมี x โดยที่ x ..."

    กำหนด domain เป็นคนทั้งหมด

- Everybody loves somebody
  - $\forall x \exists y\ love(x,y)$
- There is someone who is loved by everybody
  - $\exists x \forall y\ love(y,x)$
- Everybody is loved by someone
  - $\forall x \exists y\ love(y,x)$
- Someone loves everybody
  - $\exists x \forall y\ love(x,y)$
- Nobody love everyone
  = Everybody does not love somebody
  $\forall x \exists y \sim love(x,y)$

ถ้าโจทย์มีข้อย่อยให้ดูเครื่องหมาย $\forall x\ [\rightarrow]$
$\exists x\ [\wedge]$

- Connection between $\forall$ and $\exists$
  - De Morgan
    $\forall x \sim P \equiv \sim \exists_x P$
    $\sim \forall_x P \equiv \exists_x P$
    $\forall_x P \equiv \sim \exists_x \sim P$
    $\exists_x P \equiv \sim \forall_x \sim P$

    $\sim (P \vee Q) \equiv \sim P \wedge \sim Q$
    $\sim (P \wedge Q) \equiv \sim P \vee \sim Q$
    $P \wedge Q \equiv \sim(\sim P \vee \sim Q)$
    $P \vee Q \equiv \sim(\sim P \wedge \sim Q)$

    ต้องจดจำไว้

  - ปลดนิเสธค่า $\forall_x \equiv \sim \exists_x$ อย่าหาทำ
  - Equity → การใช้เครื่องหมายเท่ากับ
    - ex. Father(John) = Henry
    - แล้วใช้กับการนับเลขอย่างไร → "มองความจริงของมัน"
      - ex. ประยุทธ์มีพี่สาว 2 คน
        - ความจริง: พี่สาว 2 คน → มี 2 คนจริงๆ จากใน domain

          $\exists_{x,y}\ sister(ประยุทธ์, x) \wedge sister(ประยุทธ์, y) \wedge \sim(x=y)$ ← 2 คนนั้นต้องไม่ใช่คนเดียวกัน
  - Kinship domain
    - แสดงความสัมพันธ์ระหว่าง object
      - เช่น domain ของ mother $\forall_{m,c}\ Mother(c) = m \Leftrightarrow femal(m) \wedge parent(m,c)$
        - $\forall_{m,c}\ Mother(c) = m$: domain ของ mother(c)=m
        - $femal(m) \wedge parent(m,c)$: description (บอกรายละเอียดว่าคือตัวเช่น F, P จริงๆ)
  - Assertion and queries in first-order logic (การtell / การask)
    - บอกสิ่งที่บอก(tell) KB ได้เช่น tell(kb, king(Pon)), tell(kb, person(o)), tell(kb, $\forall x$ king(x) → person(x)) ← เหมือนการกำหนดกฎ (แบบเก่าเลย)
    - ถามสิ่งที่ถาม(ask) kb ได้เช่น ask(kb, king(Pon)) → return True, ask(kb, person(Pon)) → True
- First-order logic for reflex agent
  - ① cascual rule: $\forall_{s,b,u,c,t}$ percept([s, b, glitter, u, c], t) → action(grab, t) หรือ Atgold(t) ← เขียนแบบ predictive ก็ได้ (เอาไปใช้ต่อ)
  - ② Diagnostic rule: $\forall_t$ Atgold(t) → Action(Grab, t) → เพื่อมองไม่ออกลองนึกภาพ logic ง่ายๆ $P_1 \wedge P_2 \wedge P_3 \equiv (P_4)$ → เอา P_4 ไปใช้แทนเวลาเขียน $P_{1,2,3}$ ทุกรอบ

5. Uncertainty and Naive Bayes Classifier

↳ Uncertainty

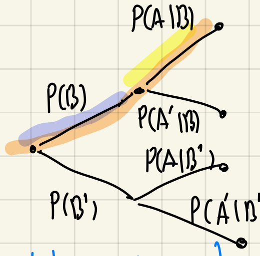
  ↳ ความไม่แน่นอน → อาจจะเกิดหรือไม่เกิดก็ได้

Prior probability

↳ ความน่าจะเป็นที่ไม่มีเงื่อนไข ex. $P(A)$

  เราสามารถเขียน $P(A \wedge B)$ ได้ด้วย $P(A, B)$

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$



$$P(A|B) = \frac{P(A \cap B)}{P(B)} = \frac{P(B|A)P(A)}{P(B)}$$

Conditional probability

↳ มีความรู้ก่อนหน้ามาเกี่ยว ex. $P(A|B)$ #B เกิดก่อน

  ↳ ถ้าเกี่ยวกับ C ด้วยก็จะเป็น $P(A|B \wedge C)$

Joint probability distribution (ความน่าจะเป็นร่วม (มีอย่างน้อย 2 เหตุการณ์เกิดขึ้น))

↳ ระวังเรื่องผลรวม prob รวมแล้วต้องได้หนึ่ง

↳ ถ้าเกิดเป็น ∨ ก็เอา prob มา + ถ้าเกิดเป็น ∧ ก็เอามาคูณกัน

  → "marginal probability" (summing out)

$$P(Y|X,E) = \frac{P(Y,X,E)}{P(X,E)} = \frac{P(Y,X,E)}{P(X|E)P(E)} = \frac{P(X|Y,E)P(Y,E)}{P(X|E)P(E)}$$

$$= \frac{P(X|Y,E)P(Y|E)P(E)}{P(X|E)P(E)} = \frac{P(X|Y,E)P(Y|E)}{P(X|E)}$$

Bayes' rule and its use

↳ $P(A|B) = \frac{P(A \cap B)}{P(B)} = \frac{P(B|A)P(A)}{P(B)}$; สามารถหาความน่าจะเป็นย้อนกลับได้

  ↳ ตัวอย่างการใช้งาน $P(\text{เป็นโรค}|\text{อาการ}) = \frac{P(\text{อาการ}|\text{เป็นโรค})\,P(\text{โรค})}{P(\text{อาการ})}$; P(อาการ|เป็นโรค) เอามาจากข้อมูลสถิติ

Probabilistic reasoning (Bayesian Network)

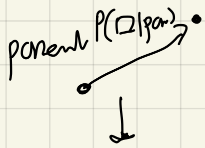
↳ รู้จัก data structure ใหม่ "Bayesian Network" → ลดปัญหาการหา $P(A \cap B \cap C \cap D \cap E)$... คูณกันขึ้นเยอะ

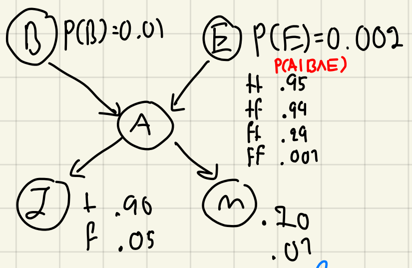
  ↳ เกิดขึ้นมา solved ปัญหา

  ↳ components: ① node : แทนด้วย var

    ② link : แบบ direct และไม่เกิด circle

    ③ แต่ละ node มี conditional probability คือ $P(X_i \mid Parent(X_i))$





ถ้าถาม $P(j, m, a, \sim b, \sim e) = ?$; ทาง sense ข้อง่ายๆ

$$= P(J|A)\,P(m|A)\,P(A|\sim B \wedge \sim E)\,P(\sim B)\,P(\sim E)$$

$$= \prod_{i=1}^{n} P(X_i \mid Parent(X_i))$$

Useful techniques for manipulating probability

	toothache		~toothache	
	catch	~catch	catch	~catch
cavity				
~cavity				

① Marginalization

$$P(cavity) = P(Cav \cap Cat \cap T) + P(Cav \cap \sim Cat \cap T) + P(Cav \cap Cat \cap \sim T) + P(Cav \cap \sim Cat \cap \sim T)$$

$$P(Y) = \sum_{z \in Z} P(Y, z)$$

การที่เราเลือกรวมตัวที่ไม่สนใจ z เลย → all possible combination

② Conditioning

↳ แทนที่เราจะใช้ marginalization เราสามารถใช้ conditional prob เพื่อคำนวณได้

$$P(Y) = \sum_z P(Y|z)P(z)$$

③ Normalization constant

↳ ธรรมชาติของความน่าจะเป็น $P(A|B) + P(\sim A|B) = 1$

$P(Ca|t) = \alpha P(Ca, t)$ (Ca = var, t = value of var)

$= \alpha [P(Ca,t,cat) + P(Ca,t,\sim Cat)]$

$= \alpha [\langle p_1, p_2 \rangle + \langle p_3, p_4 \rangle]$

$= \alpha [\langle p_{1+3}, p_{2+4} \rangle]$

$= \langle \frac{p_{1+3}}{p_{1+3}+p_{2+4}}, \frac{p_{2+4}}{p_{1+3}+p_{2+4}} \rangle$; การ scale ทำให้รวมได้ 1

$\frac{P(A \cap B)}{P(B)}$ $\quad \frac{P(\sim A \cap B)}{P(B)}$

$\therefore P(A|B) = \alpha P(A \cap B)$

P(B) เราไม่ต้องรู้ก็ได้ แทนด้วย α

↳ ประโยชน์ของสมการ "เมื่อเราไม่รู้ P(B)"

↳ (เพราะเราทำการ scale up ขึ้นไป)

④ Using independence of variables to simplify baye's rule

$P(Cavity \mid toothache \wedge catch) = \alpha P(tootache \wedge catch \mid Cavity) P(Cavity)$

↳ เพราะเรามองได้ว่า tootache กับ catch นั้นแต่ไม่ได้ขึ้นตรงเกี่ยวข้องกันเลย

สูตรแปลงเกิดขึ้น $= \alpha P(tootache \mid Cavity) P(catch \mid Cavity) P(Cavity)$

เราสามารถเขียนได้ว่า $P(Cause, E_1, E_2, \ldots, E_n) = P(Cause) \prod P(E_i \mid Cause)$

↳ ไม่มี cause มาสร้าง cause (กรณีนี้)

Naive Bayes Classifier

↳ เป็น algorithm ที่มอง feature ซึ่งๆ ใดๆ ก็ตามมันไม่มีความเกี่ยวข้องกัน

↳ เสร็จแล้วมันจะหา prob ที่มากที่สุดเพื่อ class ไหน $\max(hypo_i)$

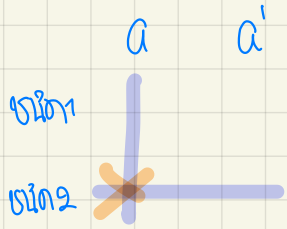
↳ เหมือนเราแค่ feature ที่เป็นคำๆ ของ มารวม → ex spam mail $P(spam) = p(spam \mid word_i) \cdot \ldots$

↳ advantage: ทำงานได้ดีมาก, event ที่ของ (feature) เป็นแบบ category, เป็น classes, event แยกกัน

disadvantage: ไม่เหมาะกับ continuous value, event ที่มีความเกี่ยวข้องกัน → มันจะไม่เหมาะ

↓ require new technique

probability



$P(a \cup \text{ชนิด2})$ = ผลรวมของเส้นม่วง

$P(a \cap \text{ชนิด2})$ = ✗

↳ เราสามารถหา $P(a \cap b)$ ได้จากตาราง แต่ยากตรง → ถ้าหาไม่ได้ → ขึ้นอยู่กับเหตุการณ์ว่าเกี่ยวกันหรือไม่

↳ เกี่ยวไม่เกี่ยว → ถ้ายกรอบข้างหน้าไป prob ตัวจะเปลี่ยน = เกี่ยว

$$P(A|B) = P(A|\sim B)$$

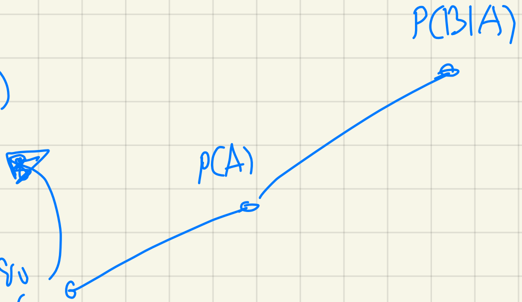
สองเหตุการณ์จะไม่เกี่ยวข้องกันก็ต่อเมื่อ

$$P(A|B) = P(A) \text{ หรือ } P(B|A) = P(B)$$

เลยเป็นที่มา $P(A \cap B) = P(A)\,P(B|A)$

$= P(A) \times P(B)$; ถ้าไม่เกี่ยว

↳ เหมือนพวกการต่อกราฟ



6. Classification I

↳ ทำไมเราไม่เขียนโปรแกรมเองแต่ต้องการใช้แบบ machine learning

- ↳ ① เขียนควบคุมจาก logic ไม่ได้
- ② มีงานแค่ในมิติที่เราคาดไม่ออก → เวลาเปลี่ยน อาจจะต้องเปลี่ยน algorithm
- ③ คิดไม่ออกว่าเขียนอย่างไร

Form of Learning

↳ ① Unsupervised learning → รู้ x ไม่รู้ y

② Reinforcement learning → รับผลจากการกระทำ x_i ตาม reward ที่ได้มา → การเรียนรู้ policy

③ Supervised learning → รู้ x รู้ y

↳ ดีกว่าจะต้องทำนายสิ่งที่ไม่เคยเห็นได้ "generalize" → แบ่ง train ≈ 70-80%

Learning Decision Tree (ID3 algorithm)

↳ การสร้าง tree ที่ optimal ที่สุด → หา node ที่มีความสำคัญมากที่สุด → แยกไปสุด

↳ Information gain entropy

↳ วัดความไม่แน่นอน → entropy ∝ ความไม่แน่นอน

ข้อมูล $label_1 (p)$: +++++++ } entropy ↑ ①

$label_2 (n)$: ----

↓

$Attribute_n$ → Gain($Attribute_n$)

↓

$Attribute_n$

p +++ n | p ++++ n | p n ---- } entropy per group ↓ ②

∴ ที่อยากได้ $Gain(A) = I\left(\frac{p}{p+n}, \frac{n}{p+n}\right) - Remainder(A)$

(① = $I\left(\frac{p}{p+n}, \frac{n}{p+n}\right)$, ② = Remainder(A) → ผลรวมของ branch ที่ possible)

K Nearest Neighbor Algorithm (KNN)

↳ ดูว่าอยู่ใกล้ใครมากสุด k ตัว

↳ อย่าลืมเรื่องการ scale

↳ k-fold เหมือนแบ่ง data ออก → แบ่งส่วนจาก train → val → หา mean ของ accuracy, validation

7. Classification II Neural Network

Perceptron คือแบบของ neural network

↳ error = predicted − target

วิธีการ update ค่าพารามิเตอร์ $w_i(p+1) = w_i(p) + \alpha\, x_i(p) \cdot error$

↳ ข้อเสีย เครื่องไม่รู้เรื่องเลยว่าค่า error rate ที่ยอมรับได้ คือ "coverage" ของข้อมูลเท่าไหร่

Multilayer neural network

↳ เพิ่ม hidden layer เข้าไป

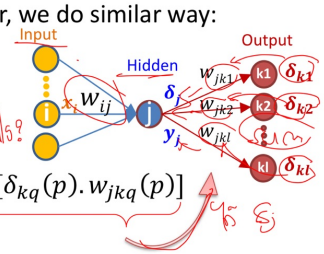
↳ ใช้ backpropagation algorithm ในการปรับ weight

- To update weights in the hidden layer, we do similar way:

$$w_{ij}(p+1) = w_{ij}(p) + \Delta w_{ij}(p),$$

$$\Delta w_{ij}(p) = \alpha . x_i(p) . \delta_j(p),$$

$$\delta_j(p) = y_j(p) . \big(1 - y_j(p)\big) . \sum_{q=1}^{l} [\delta_{kq}(p) . w_{jkq}(p)]$$



Where l is the number of neurons in the output layer.

$$y_j(p) = \frac{1}{1+e^{-X_j(p)}},$$

$$X_j(p) = \sum_{i=1}^{n} [x_i(p) . w_{ij}(p)] - \theta_j$$

Step3: Weight training (Backpropagation)

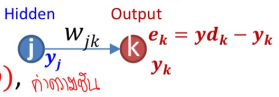
- Calculate the error gradient of neurons in the hidden-output layer.

$$e_k(p) = yd_k(p) - y_k(p),$$

$$\delta_k(p) = y_k(p) . \big(1 - y_k(p)\big) . e_k(p),$$

$$\Delta w_{jk}(p) = \alpha . y_j(p) . \delta_k(p),$$

$$w_{jk}(p+1) = w_{jk}(p) + \Delta w_{jk}(p)$$



- Calculate the error gradient of neurons in the input-hidden layer.

$$\delta_j(p) = y_j(p) . \big(1 - y_j(p)\big) . \sum_{k=1}^{l} \delta_k(p) . w_{jk}(p),$$

$$\Delta w_{ij}(p) = \alpha . x_i(p) . \delta_j(p),$$

$$w_{ij}(p+1) = w_{ij}(p) + \Delta w_{ij}(p)$$