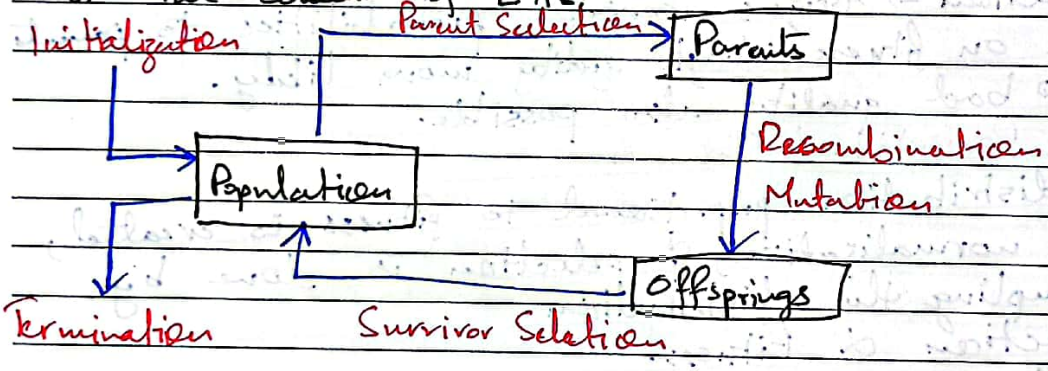
Computational Intelligence Notes

Nature Inspired Computing → inspired by nature
↳ to simulate / emulate nature
↳ bring properties to our systems such as:
↳ robustness, fault tolerance, self-repair.

Combinatorial Optimization → finding optimal object from a set of finite objects. → Exhaustive search not tractable.

Evolutionary Computing (EC)
→ Evolution → change in inherited traits of a population of organisms through successive generations.
→ EC is the study of comp sys. inspired by natural evolution.
↳ an EA is a generic population based meta-heuristic optimization algorithm.
↳ uses mechanisms such as:
reproduction, Mutation, Recombination, Selection

General Scheme of EAs:



Variation & Selection → Reduces diversity
↳ some alleles survive.
↳ Builds up genetic Diversity
↳ produces new alleles.

Exploitation vs. Exploration.
↳ a good alg must find a good tradeoff b/w the Two.
↳ Exploration investigates new & unknown areas in the search space
↳ Exploitation makes use of knowledge found at points previously visited to find better points.

Typical EA Cycle:

1-> Initialize random samples.

2-> While no convergence (or terminating condition)

(a)-> evaluate fitness

(b)-> Perform crossover

(i) Select two parents

(ii) Produce offsprings

(c) perform mutation (select one individual & mutate it)

(d) select the new generation

(e) evolve the next generation.

-> Crossover (merging two parents) is explorative since it makes a jump b/w two parent areas.

-> Mutation is exploitative. since it creats small diversities thereby staying near the parent.

* Only crossover can combine info
* only mutation can introduce new info.

Selection Schemes:

Parent Selection -> Assigns variable probabilities to parents based on fitness -> high quality more likely.
b bad quality also possible.

1. Fitness Proportional Scheme:
   - b A distribution proportional to fitness is created, by normalization & selection is done by sampling the distribution.
   - b Selection α Fitness.
   - b High selective pressure -> strong individuals dominate.
2. Rank Based Selection
   - b Rank population acc. to fitness & base selection probabilities on rank: where worst has 1, best has μ.
   - b Selection indep. of actual fitness
     - b best won't dominate.
   - b preserves constant selection pressure.

3. Tournant Selection
- ↳ If population v. large, might be too time consuming for FP or R, or if universal fitness not dif.
- ↳ In TS, pick n_{ts} numbers at random & then select the best of those.
  - ↳ inherits adv of rank.
- ↳ For crossover, TS done twice (once for each parent).
- ↳ If n_{ts} not too large, prevents the best from dominating → lower selective pressure.
- ↳ If n_{ts} too small, bad chances increase.
- ↳ Pressure α n_{ts}. (at $n_{ts}=1$ → random, $n_{ts}=s$ → best done)

TS most widely used.

4. Random Selection.
- ↳ simplest sol. $P(x) = 1/n$ for each indiv.

Replacement:

Age-Based → remove certain indivs that have not been replaced in a set time.

Fitness-Based → On fitness.
- ↳ Replace Worst / Truncation.
  - ↳ worst λ numbers are replaced.
  - ↳ Rapid improvements in the mean
    - ↳ can lead to premature convergences.
- ↳ Elitism → ensuring that the best survive.
  - ↳ best copied without mutation.
  - ↳ less diversity
  - ↳ ~~Immunity~~ guarantees convergence (risk of local maxima)
- ↳ Hall of Fame. → best individual of each generation is selected to be inserted into H.o.F.
  - ↳ parent pool for crossover.

Variants of EA

EC vs Classical Optimization [CO]

* CO uses deterministic rules to move from one point to another. EC uses prob.
* CO is more successful in linear, quadratic, strongly convex, unimodal & other specialized problems. EAs more efficient in discontinuous & multimodal probs.
* EC starts from diverse set of points, CO starts from one point.
* CO uses derivative info, ECs use fitness value.

Swarm Intelligence

Ant Colony Optimization (ACO):

Swarm Behaviours in Animals:
- Food foraging
- clustering
- flocking / schooling
- Natural Construction

Key Principles - social insects work without supervision
- individuals largely homogeneous
- ↳ act asynchronously in parallel.
- self-organization
- flexibility → colony can adapt to changing environment
- Robustness → group can perform even if an individual fails.
- Communication → stigmergy.

Stigmergy: * Communication via signs or cues placed by one entity which affects the behaviour of other entities who encounter them
* Stigmergy is the indirect communication that seems to underpin cooperation among social insects.
↳ form of self organization → produces complex, seemingly intelligent structures, without planning control & or direct communication b/w agents.
↳ Supports efficient adaptation.
→ Ants lay down pheromones on their way back. when they find food.
Termites example → deposit mudballs, positive feedback

① Ant Colony Optimization (ACO)
↳ food foraging behaviour of ants
② Particle Swarm Optimization (PSO)
↳ flocking & fish schooling

① Ant Colony Optimisation (ACO):

- While initially random, once a food source is located, activity patterns become more organized.
- This behaviour emerges from individuals following 2 simple rules: - lay pheromones
  - follow the trails of others.

* Working of AC: two ants leave at the same time, taking separate paths to the food source, marking their trail with pheromone.
* The ant taking the shorter path returns first, thus this trail is marked by twice as much pheromone.
* Since shorter path has more concentration of pheromones, next ants attracted to that.
* As more ants take that route, they further lay their pheromones, further amplifying their attractiveness.
* As time passes, pheromones also evaporate, so longer paths vanish.
  - Also prevents convergence to a sub optimal solution.
* Without evaporation, exploration is restricted.

ACO Algorithm:

~~* Let $B_i(t)$ be number of ants~~

* Let $B_i(t)$, $(i=1,\dots,n)$, be the nber of ants in town i at time t,
* Let $m = \sum B_i(t)$ the total number of ants.
* Each ant is a simple agent with the following characteristics:
  - chooses the town to go to with a probability that is a function of the town distance & of the amount of trail present on the connecting edge.
  - To force legal tours, transitions to already visited towns are disallowed until a tour is completed.
  - When a trail is completed, a "trail" is layed on each edge (i,j) visited.
* Let $\tau_{ij}(t)$ be intensity of trail on edge (i,j) at time t.
* Each ant at time t chooses the next town $(t+1)$.

* In n iterations, each ant completes a tour.
* Trail intensity updated:

$$\tau_{ij}(t+n) = \rho \cdot \tau_{ij}(t) + \Delta\tau_{ij}$$

∴ ρ is a coefficient such that $(1-\rho)$ is the evaporation of trail between time t & t+n.

$$\therefore \Delta\tau_{ij} = \sum_{k=1}^{m} \Delta\tau_{ij}^{k}$$

$$\Delta\tau_{ij}^{k} = \begin{cases} Q/L_k & \text{if k-th ant uses edge } (i,j) \text{ in its tour.} \\ 0 & \text{otherwise.} \end{cases}$$

Q is constant, L_k is tour length of k-th ant.

* Tabu List Data Structure associated with each ant.
  ↳ saves already visited towns upto time 't'

* ACO has an advantage over evolutionary algorithms when the graph may change dynamically.
  ↳ Can adapt in real time
  ↳ Interest in network routing & urban transportation systems.

ACO has been applied to various Combinatorial Optimization Problems (COPs)

- Assignment Problem
- Closure Problem
- Constraint Satisfaction Problem
- Cutting Stock Problem
- Integer Programming
- Knapsack Problem
- Minimum Spanning Tree
- Nurse Scheduling Problem
- Vehicle Routing Problem

Date:

Particle Systems & Particle Swarm Optimisation (PSO)

PSO → inspired by bird flocking & fish schooling
→ population based stochastic optimisation technique
→ no evolution operators
→ initially a random solution, searches for optima by updating generations.

* **Particle** → current position
→ velocity

Velocity Update: → current velocity
→ personal best [PB]
→ global best [GB]
} influence velocity update.

* At each step, particles are displaced from their current position by a vector gradient
* Magnitude & direction of their velocity at each step is influenced by their velocity in the previous iteration, simulated momentum, location of the particle relative to the PB & GB.
* Particle updates its velocity & position.

```
v[] = φ*v[] + c1*rand()*(pbest[] - present[])
            + c2*rand()*(gbest[] - present[])
present = present[] + v[].
```

c1 & c2 are learning factors, (usually 2)
φ is inertia weight.

* PSO: → Swarm, a set of particles (S)
→ Particle, a potential solution
• position: $x_i = (x_{i,1}, x_{i,2}, \ldots, x_{i,n}) \in \mathbb{R}^n$
• Velocity: $v_i = (v_{i,1}, v_{i,2}, \ldots, v_{i,n}) \in \mathbb{R}^n$.
→ each particle maintains its PB
→ Swarm maintains GB.

Date:

PSO Algorithm: 1 → Randomly initialize the swarm to form the solution space.
2 → evaluate the fitness of each particle
3 → Update individual [PB] & global best.
4 → update velocity & position of each particle.
5 → Repeat from step 2 until termination condition.

* Initially pbest is the starting location & gbest is the best out of all particles [best pbest for all particles].
* In each generation, a particle is accelerated towards its previous best position & towards a global best position, forcing particles to continually search in the most-promising regions.

Date:

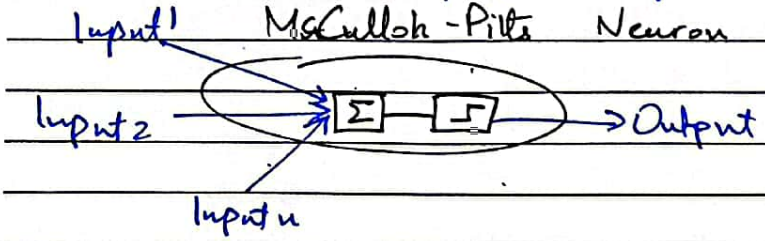
Artificial Neural Networks (ANNs)

→ Neural Networks modelled after our understanding of how the brain works.
→ Because of massive number of neurons & interconnections, how the brain works remains a mystery.

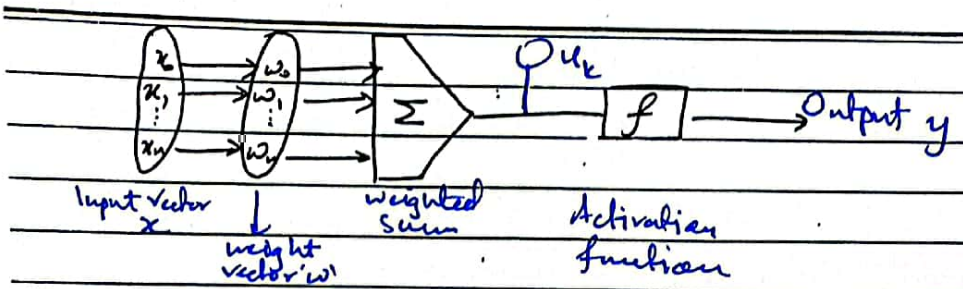
Neural Networks: → Made up of highly connected network of individual computing elements (mimicking neurons)
→ Collectively used to solve interesting & difficult problems.

Neurons: - responsive cells in the nervous system that process & transmit information by on chemical signals within the neuron.
- number of different neurons exist.
- respond to stimulus & communicate the presence of that stimuli to the central nervous system which processes that info & sends responses to the other parts of the body

Artificial Neural Network:
→ highly connected network of individual computing elements (mimicking neurons) that collectively can be used to solve interesting & difficult problems.
→ An "artificial neuron" is an information processing unit that is fundamental to the operation of an ANN.
→ Artificial Neuron → Set of input links
↳ to Output
↳ An adder for Σ input signals weighted by respective synaptic strengths.
↳ Activation function for limiting the amplitude of the output of a neuron.



Date:



* The 'n' dimensional input vector 'x' is mapped onto variable 'y' by means of the scalar product & a nonlinear function mapping.

Multi-Layer Feed Forward Networks

→ one of most imp & popular classes of ANNs
→ referred to as multilayer perceptrons.

Multi Layer Perceptron (MLP)

(1) Network contains one or more layer of hidden neurons not part of input/out. These hidden nodes enable the network to learn complex & nonlinear tasks by extracting progressively more meaningful features from input patterns.
(2) Model of each neuron usually includes a non-linear activation function, sigmoid or hyperbolic.
(3) High degree of connectivity from one layer to the next.

How a multi-layer- neural network works?

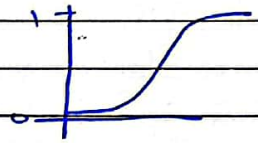
→ Inputs correspond to attributes for each training tuple.
→ Inputs fed simultaneously to the input layer.
→ Weighted & fed simultaneously to (a) hidden layer(s).
→ Weighted outputs are inputs to the output layer which emits the networks predictions.
→ Non Linear Regression is performed

Date:

Specification of an ANN:

→ Number of input attributes found within the individual instances determines the number of input layer nodes.
→ The user specifies the number of hidden layers as well as the number of nodes within a specified hidden layer.

* Input Format. → Numeric ∈ [0, 1]
  → Way to numerically represent categorical data
    - Attribute Color: {R, G, B, Y}
  → Conversion method for numerical data falling outside the [0, 1] range E.g. (100, 200, 420).
* Output Format: [0, 1].
* Activation Function:
  ⇒ Sigmoid Function



  (1) → Must output ∈ [0, 1]
    ↳ Criteria 1
  (2) → Value output should be close to 1 when function is sufficiently excited.

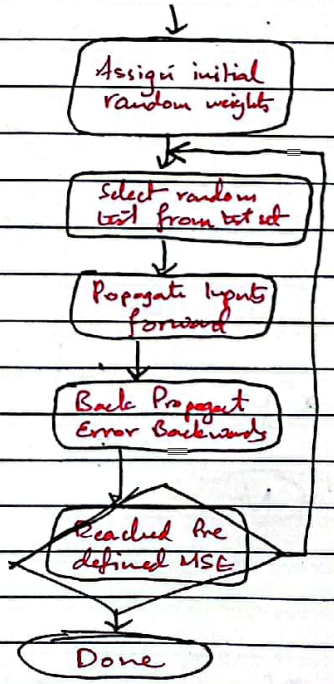
Sigmoid meets the above 2 criteria:

$$f(x) = \frac{1}{1+e^{-x}} = \text{Sigmoid}(x)$$

$$\tanh\left(\frac{x}{2}\right) = \frac{1-e^{-x}}{1+e^{-x}}$$

Learning a Neural Network

* Learning is accomplished by modifying network connections weights while a set of input instances is repeatedly passed through the network
* Once trained, an unknown instance is passed through the model/network & classified acc to the values at the output layer.

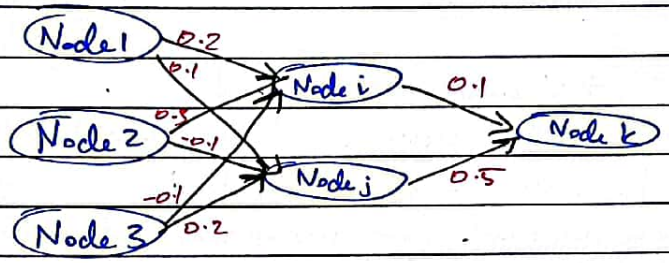


Working of an ANN

Date: .....

Feed-Forwarding Data:

* Input = {1.0, 0.4, 0.7}
* Input to node i = 0.2(1.0) + 0.3(0.4) − 0.1(0.7)
* Now apply the sigmoid function: = 0.562



Gradient Descent

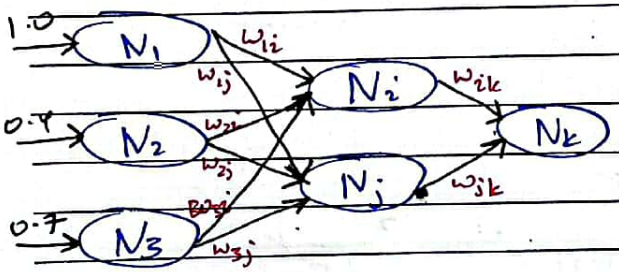
* Alg that minimizes function
* Starts with an initial set of parameter values & iteratively moves towards a set of parameter values that minimize the function.
* Achieved using Calculus → taking steps in the negative direction of the function gradient.

Gradient: → grad of each weight gives an indication on how to modify the weight to get the expected output / reduce error.

→ Each weight has gradient → slope of the error function
- ↳ 0 ⇒ weight not contributing to error
- ↳ -ve ⇒ weight should be increased for lower error
- ↳ +ve ⇒ weight should be decreased for lower error.

Explanation of Back Propagation Algorithm

$w_{1i} = 0.2$, $w_{1j} = 0.1$, $w_{2i} = 0.3$, $w_{2j} = -0.1$, $w_{3i} = -0.1$, $w_{3j} = 0.2$
$w_{ik} = 0.1$, $w_{jk} = 0.5$, $T = 0.65$



* Input = {1.0, 0.4, 0.7}
* Input to N_i = 0.2(0.1) + 0.3(0.4) − 0.1(0.7) = 0.25
* Apply Sigmoid: f(0.25) = 0.562 to N_i
* Input to N_j = 1.0(0.1) + 0.4(−0.1) + 0.7(0.2) = 0.2
  f(0.2) = 0.549
* Input to N_k = 0.582
* Error(k) = (T − O_k) O_k (1 − O_k) = (0.65 − 0.582)(0.582)(1 − 0.582) = 0.016

Date:

* Error(i) = Error(k) w_{ik} O_i $(1-O_i)$
  = 0.418 (0.1)(0.562)(1-0.562) = 0.01
* Error(j) = Error(k) w_{jk} O_j $(1-O_j)$
  = 0.418 (0.5)(0.549)(1-0.549) = 0.0517
* Next step is to update the weights ~~based on~~ associated with individual node connections.
* Adjustments made using delta rule.
  ↳ Minimize Σ square error [error = diff(computed, actual output)]

⇒ * $w_{ik} = w_{ik}(\text{current}) + \Delta w_{ik}$
* ⊗ $\Delta w_{ik} = r \times \text{Error}(k) \times O_i$
  • ↳ r is learning rate $0 < r < 1$
• Compute Δw_{ik}, Δw_{ji}, Δw_{2i}, Δw_{5i}
  w_{ik}(current), r, error, O_i
  * $w_{ik} = 0.5 + (0.8 \times 0.418 \times 0.549)$
  = 0.683

*

Algorithm:

1 * Initialize the network
  i - create topology by choosing the number of nodes for the input, hidden, & output layers.
  ii - initialize weights for all node connections ~~hard-coded~~ to arbitrary values [-1.0, 1.0]
  iii - choose $r \in [0,1]$
  iv - choose terminating condition.
2 * For all training instances:
  i - feed training ~~data~~ instance through network
  ii - determine the output error
  iii - update network weights.
3 * If terminating condition not met, repeat step 2.
4 * Test accuracy of the network on a test data set
  If accuracy is less than optimal, change one or more parameters & start over.

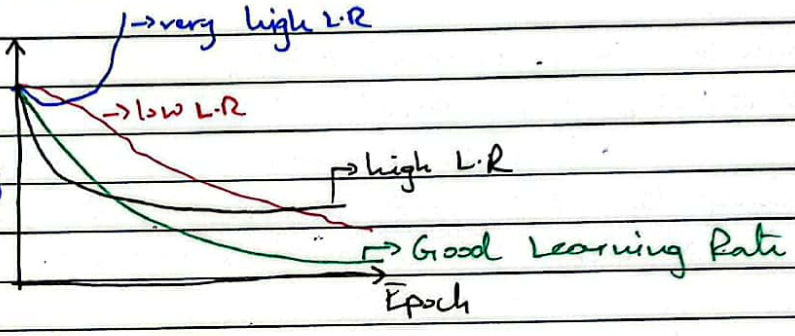
Date:

Training / testing of ANN:

* During training phase, training instances repeatedly passed through network while weights are modified.
* Connection weights changed to minimize training set error rate.
* Training repeat till terminating condition met.
* Terminate (could be any) { → convergence of the network to a minimum value, → time criterion, → Max iterations.

Learning Rate

* Epoch is one complete pass through the training dataset / instance.



* A bias value allows you to shift the activation function to the left or right → may be critical for successful learning.

Weakness of ANNs:

* lack the ability to explain their behaviour
* not guaranteed to converge to an optimal solution
* NNs can be over trained on training data but work poorly on test data.

Date:

Self Organizing Maps (SOM)

* A SOM falls under the domain of unsupervised learning in Neural Networks.
* Grid of neurons - each denotes one cluster learned during training.

SOM vs ANN:

* SOM is not a series of layers, but a 2D grid of neurons
* Don't error correct rather learn by competitive learning
* Deal with unsupervised machine learning problems
* Competitive Learning for SOMs refers to the fact that when an input is "presented", only one of the neurons will be activated. In a way, the neurons on the grid "compete" for each input.

Clustering in SOM:

* Each neuron has a location.
* Neurons close to each other represent a cluster → similar properties
* Each neuron has a weightage vector
  - is equal to the centroid of its particular cluster.

SOM Setup:

* ~~Vector~~ N-Dimensional Vector $x : x = \langle x_1, x_2, x_3, ---, x_n \rangle$
* N-dimensional weight vector $W : W = \langle w_1, w_2, w_3, ---, w_n \rangle$
* No Lateral connections b/w neurons

Training SOM: → Several Steps over many Iterations.

1. Weights randomly initialized for each node
2. Random vector chosen from training data → given to lattice
3. Every node examined → calculate node's weights most similar to input vector. Winning node commonly known as Best Matching Unit (BMU).
4. Radius of the neighborhood of BMU ~~found~~ calculated. Nodes within this radius deemed inside BMU's neighborhood.
5. Each neighboring node's (from 4.) weights are adjusted to make them like the input vector. Closer to BMU, more the weight gets altered.
6. Repeat from (2.) for N-Iterations.

Date: .....

Neighborhood Function

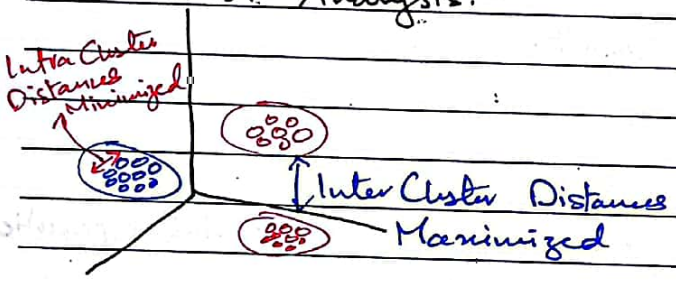
* Each Iteration after BMU, we find nodes within the BMU's neighborhood.
  ↳ All such nodes have their weights altered.
* Kohonen Learning Algorithm → area of the neighborhood shrinks over time by reducing the radius.

$$\therefore \quad W(t+1) = W(t) + \Theta(t)\, L(t)\,[V(t) - W(t)]$$

* L is the Learning Rate
* Θ → Theta is the neighborhood function.

} Time Variant Functions.

* Cluster Analysis:



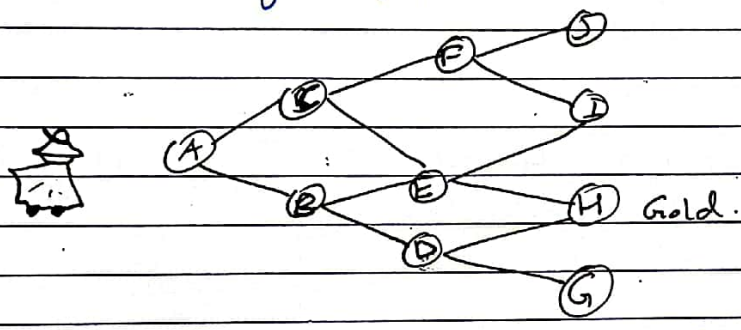
* A Self-Organizing Map (SOM) is a grid of neurons that adapt to the topological shape of the dataset.
  ↳ Allows us to visualize large datasets & identify potential clusters.
* SOM learns the shape of the dataset by repeatedly moving its neurons closer to the data points
  ↳ Different groups of neurons thus reflect underlying clusters

Date:

Reinforcement Learning (RL)

* → RL agent learns by interacting with the environment & observing the results of these interactions.
  → Reward & Loss system
  → Mimics how humans & animals learn

Jackpot Journey Problem
* traveler looking for gold
* white & black stones at each vertex that has a signpost.
* white stone go up^{u}, black stone go down^{d}



* After concluding a journey, traveler back to A,
  - Put a stone with an additional stone if gold found
    - Reward
  - Remove stone if gold not found → loss / Penalty

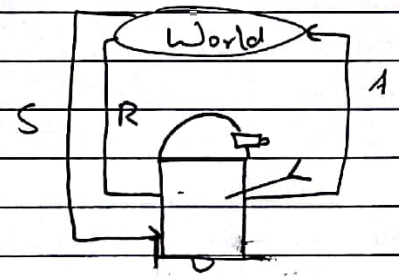
An RL Agent:
→ observes an input state
→ Action determined by a decision making function
→ action performed
→ agent recieves a reward or penalty from environment
→ information about reward is recorded.

* By observing the resulting reward, the policy for best action for a state can be fine-tuned.
* Eventually the optimal decision policy will be generated & our agent can perform perfectly in that environment.

Date:

* RL is distinguished from other computational approaches
  ↳ learning by the individual from interactions with the environment. → does not rely on exemplary supervision or complete models.

Basic RL model:
1 → Observe state, s_t
2 → Decide on action, a_t
3 → Perform action
4 → Observe new state, s_{t+1}
5 → Observe reward, r_{t+1}
6 → Learn from experience
7 → Repeat.



Markov Decision Process (MDP) 4-Tuple $(S, A, P(.,.), R(.,.))$
* S is a finite set of states
* A is a finite set of actions
* $P_a(s, s')$
* $R(s')$

Elements of RL → a policy
→ optionally, a model of the environment
→ a reward function
→ a value function

Policy: * Mapping from perceived states of the environment to the actions to be taken when in those states.

Model: * Mimics the behaviour of the environment
* might predict the resultant next state & next reward
* RL models the environment in the form of MDPs

Reward Function: * defines goal in RL problem
* maps each state to a single number
* ↳ indicates intrinsic desirability of the state.

Date:

Value Function:

* value of a state is the amount of reward an agent can expect to accumulate over the future.
* value function specifies what is good in the long-run.

Rewards vs Values: * without rewards, no values exists
* purpose of estimating values is to get more rewards
* We are mostly concerned with value when making & evaluating decisions.
* ↳ we seek actions that bring about states of highest value, not highest reward
  ↳ highest values yield highest rewards over the long run.
* Efficiently calculating/estimating values is the most imp. component of almost all RL algs.

Cumulative Future Reward

$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T$

$G_t = \sum_{k=0} R_{t+k+1}$ * G_t is total future rewards from time step t.

Time value of reward: → Can't add rewards in reality
→ rewards that come sooner are more probable to happen, since they are more predictable than the long term.

↳ **Discounting:** * We define a discount rate 'gamma' $\in [0, 1]$.
* Larger gamma implies smaller discount.
  ↳ learning agent cares more about the reward in long term
* smaller gamma means a bigger discount
  ↳ learning agent cares more about short term reward.

Discounted Cumulative Reward

$R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} \cdots$

$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$ where $\gamma \in [0, 1)$

Date:

Value Iteration:

Path : $s_0 \to s_1 \to s_0 \to s_2 \to \dots$

$$\text{Value}(s_0) = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \gamma^3 R(s_3) + \dots$$

$$\text{Value}(s_0) = R(s_0) + \gamma V(s_1)$$

Since state transitions are stochastic, we have:

$$\text{Value}(s_0) = R(s_0) + \gamma E[V(s_1)]$$

* Value is the prediction of a series of rewards & is calculated as the expected sum of discounted rewards

* The agent will evaluate all the states in the state space, & the value function tells the agent how good each action is in a particular state.
  ↳ Agent chooses the best one.

Bellman Equation:

Expected Reward of executing action 'a' in 's'.

$$V(s) = \max_{a \in A(s)} \sum_{s' \in S} P_a(s'|s) \left[r(s,a,s') + \gamma V(s') \right]$$

- $a \in A(s)$: Best actions from S
- $s' \in S$: For every state
- $r(s,a,s')$: immediate reward
- γ: Discount factor
- $V(s')$: Value of s'

Value Iteration:

Initialize V arbitrarily, eg, $V(s) = 0 \quad \forall s \in S^+$

Repeat

$\Delta \leftarrow 0$

For each $s \in S$:

$v \leftarrow V(s)$

$V(s) \leftarrow \max_a \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V(s') \right]$

$\Delta \leftarrow \max(\Delta, |v - V(s)|)$

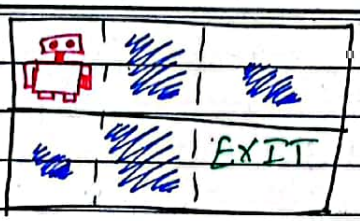
until $\Delta < \theta$ (a small +ve number)

Output a deterministic Policy π s.t.

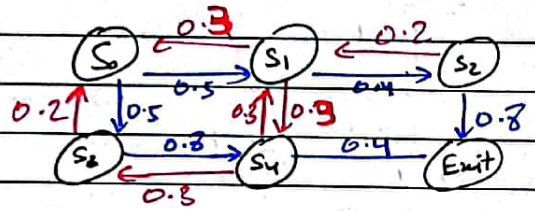
$$\pi(s) = \arg\max_a \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V(s') \right]$$

Date:

Baby Robot



* Small Puddles → skid = 20%
  ↳ not skid = 80%
* Big Puddles → skid = 60%
  ↳ not skid = 40%



Types of Learning

* Episodic ⇒ rewards collected at the end of the episode, & then calculating maximum expected future reward.
* Temporal Difference learning ⇒ Estimate the rewards at each step.

Date:

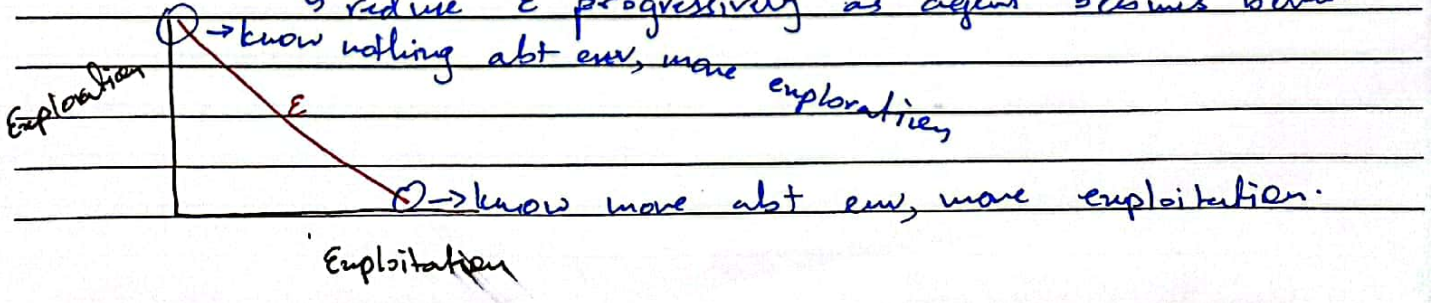
Q-Learning

→ does not need enviroment → can be used online
→ estimating values of state-action pairs
→ Can perform adaptively in a world without understanding
  ↳ sort out good actions from bad ones.
→ Q(s,a) value defined by the expected discount sum of future payoffs obtained by taking action 'a' from state 's' & following the current optimal policy.
  ↳ Once values learned, optimal action from any state is one with the highest value.

Initialize Q(s,a) arbitrarily
Repeat (for each episode)
  Initialize s
  Repeat (for each step of the episode)
    choose a from s using policy derived from Q
    Take 'a', observe r, s'
    $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
    $s \leftarrow s'$;
  until s is terminal.

Choosing an Action:
- function that assigns a probability of being chosen for each action in a given state.
- function should tend to choose actions with higher Q-values. should sometimes select lower Q-value actions
- prob of selecting highest-Q value action should increase over time.

Action Selection Policy:
1. ε-greedy: * P(1-ε) → choose max-value action
  * P(ε) → randomly choose an action from all actions
  → Big ε at the start of Q-function
    ↳ reduce ε progressively as agent becomes better.



Date:

2. Boltzmann Distribution

→ ε can also take worst actions with equal chance even when it is unfavorable.

→ Vary action probs as a graded function

$$P(a'|s) = \frac{e^{Q(s,a)/k}}{\sum_j e^{Q(s,a_j)/k}}$$

→ k → temperature → controls the prob of selecting non-optimal actions. k↑ → all actions selected uniformly
k↓ & Limit k → 0, best always chosen.

→ Begin with large k & decrease it over time.

Calculating Error:

$$\text{New } Q(s,a) = Q(s,a) + \alpha[R(s,a) + \gamma \max\{Q'(s',a')\} - Q(s,a)]$$

- $Q(s,a)$: current Q val.
- α: learning rate
- $R(s,a)$: Reward for 'a' at 's'
- γ: Discount fact
- $\max\{Q'(s',a')\}$: Max expected future reward given new s' & all possible actions at that new state.

$$\text{Loss} = (r + \gamma \max Q(s,a') - Q(s,a))^2$$