CS2109S Intro. to AI & Machine Learning

AY 25/26 Sem 1 — github/omgeta

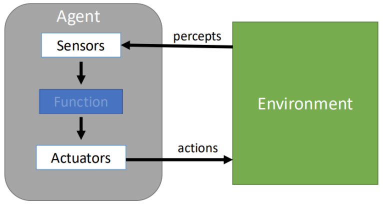
1. Intelligent Agents

Agents receive precepts from the environment through sensors and perform actions through actuators.

PEAS Framework

PEAS defines problems in terms of Performance Measure, Environment, Actuators and Sensors:

i. Rational Agent: chooses actions that maximise performance measure



Properties of Task Environment:

i. Fully Observable (vs. Partially Observable): sensors give access to the complete state of the environment at each point in time.

ii. Single-agent (vs. Multi-agent): agent operates by itself in an environment.

iii. Deterministic (vs. Stochastic): next state of the environment is completely determined by current state and action by agent.
   - Strategic: if environment is dependent on actions of other unpredictable (not "dumb") agents

iv. Episodic (vs. Sequential): agent's experience is divided into atomic "episodes", and choice of action in each episode depends only on the episode itself.

v. Static (vs. dynamic): environment is unchanged while agent deliberates.
   - Semi-dynamic: if environment does not change with time, but agent's performance score does.

vi. Discrete (vs. Continuous): a limited number of distinct percepts and actions.

Agent Structures

Agents are completely specified by the agent function:

i. Agent Function: $f : \mathcal{P} \to \mathcal{A}$ maps from precept histories $\mathcal{P}$ to actions $\mathcal{A}$

ii. Agent Program: implements the agent function

Common Agent Structures:

i. Simple Reflex: chooses action only based on current percept, ignoring precept history (follow if-then rules).

ii. Goal-based: select actions to achieve a given tracked goal.

iii. Utility-based: selects actions to maximise a utility function which assigns a score to any precept sequence. If the utility function aligns with performance measure, the agent is rational.

iv. Learning: improves performance over time with experience.
   - Performance Element: selects actions.
   - Learning Element: updates knowledge from feedback.
   - Critic: provides feedback on performance relative to a fixed performance standard.
   - Problem Generator: suggest exploratory actions

Exploration-Exploitation Dilemma:

i. Explore: learn more about the world.

ii. Exploit: maximize gain from current knowledge.

2. Systematic Search

Systematic search problems are fully-observable, deterministic, static, discrete and defined by:

i. States: representation of problem state

ii. Initial State

iii. Goal State/Test

iv. Actions: possible operations on a state

v. Transition Model: function of action on a state

vi. Action Cost Function

Uninformed Search

Uninformed search explores a problem space without domain-specific knowledge or heuristics to guide searching.

i. Branching factor b: max successors of any node

ii. Optimal solution depth d, Maximum depth m

Breadth-First Search (BFS) expands nodes level-by-level using a queue:

i. Time: $O(b^{d+1})$

ii. Space: $O(b^d)$

iii. Complete: Yes (if b is finite)

iv. Optimal: Yes (if all same step cost)

Uniform-Cost Search (UCS) expands least-cost node using a priority queue, for optimal solution cost C^*, levels C^*/ϵ:

i. Time: $O(b^{C^*/\epsilon})$

ii. Space: $O(b^{C^*/\epsilon})$

iii. Complete: Yes (if step costs $> 0 \land$ finite total cost)

iv. Optimal: Yes (if step costs > 0)

Depth-First Search (DFS) expands deepest unexpanded nodes using a stack:

i. Time: $O(b^m)$

ii. Space: $O(bm)$

iii. Complete: No (if infinite depth $\lor$ cyclic)

iv. Optimal: No

Depth-Limited Search (DLS) limits search depth to ℓ:

i. Time: $O(b^\ell)$

ii. Space: $O(b\ell)$ (with DFS)

iii. Complete: No (if $\ell < d$)

iv. Optimal: No (with DFS)

Iterative Deepening Search (IDS) runs DLS with increasing depth limit until solution found:

i. Time: $O(b^d)$ (with additional overhead)

ii. Space: $O(bd)$ (with DFS)

iii. Complete: Yes (if b finite)

iv. Optimal: Yes (if all same step cost)

Informed Search

Informed search uses heuristics to guide searching, where heuristic $h(n)$ estimates optimal cost from a state to goal:

i. All heuristics must be non-negative $\wedge$ $h(goal) = 0$

Usefulness Properties:

i. Admissible: $\forall$ node n, $h(n) \leq h^*(n)$ where $h^*(n)$ is optimal path cost to reach goal state from n ($h(n)$ never over-estimates true cost to goal)
   - Theorem: if $h(n)$ is admissible, A^* search (without visited memory) is optimal

ii. Consistent: $\forall$ node n, $h(n) \leq c(n, a, n') + h(n')$ (triangle inequality)
   - Theorem: if $h(n)$ is consistent, A^* search (with visited memory) is optimal
   - Theorem: if $h(n)$ is consistent, $h(n)$ is admissible

iii. Dominance: $\forall$ nodes n, $h_1(n) \geq h_2(n)$ implies that h_1 dominates h_2 (more informed)
   - Theorem: if h_1 is admissible, h_1 is better for search, expanding no more nodes than h_2

Admissible heuristics can be found using optimal cost of a relaxed problem (fewer restrictions on actions):

Ex. h_{SLD} straight-line distance if agent can fly

Ex. $h = 0$ if agent can teleport

Best-First Search expands nodes with evaluation $f(n) = h(n)$ using a priority queue:

i. Time: $O(b^m)$

ii. Space: $O(b^m)$

iii. Complete: No (may get stuck in loops).

iv. Optimal: No (heuristic-only may mislead).

A* expands nodes with evaluation $f(n) = g(n) + h(n)$, where $g(n)$ is step cost, using a priority queue:

i. Time: $O(b^d)$ (good heuristic can improve)

ii. Space: $O(b^d)$ (keeps all nodes in memory).

iii. Complete: Yes (if step costs $> 0 \wedge$ finite b).

iv. Optimal: Depends (if h admissible in tree search, or consistent in graph search).

3. Local Search

Local search problems are defined by:

i. States: representation of candidate solution, may not map to actual problem state

ii. Initial State

iii. Goal State/Test (optional)

iv. Successor Function: generate neighbour states by applying modifications to current state

Local search explores large, otherwise intractable problem spaces by considering only locally reachable states, guided by an evaluation function:

i. Evaluation Function: assesses quality of a state; we either minimize or maximise this function

State Space Landscape:

i. Global Maximum: optimal solution

ii. Local Maximum: local optimum solution

iii. Shoulder: region with flat evaluation function, difficult for algorithm to move past

Hill-Climbing searches for local optimum, generating successors from current state and picking the best using heuristic:

i. Any-time: More time gives better solutions

ii. Space: $O(b)$

iii. Complete: No

iv. Optimal: Not guaranteed

v. Variants:
   - Simulated Annealing: allow some bad moves, gradually decreasing frequency
   - Beam Search: perform parallel k hill-climbs
   - Genetic Algorithm: successor generated by combining 2 parent states
   - Random-Restart: escape local optimum by restarting search

4. Adversarial Search

Adversarial search problems are fully-observable, deterministic-strategic, static, discrete and defined by:

i. States: representation of candidate solution, may not map to actual problem state

ii. Initial State

iii. Terminal States: where game ends (e.g. win)

iv. Actions: possible operations on a state

v. Transition Model: function of action on a state

vi. Utility Function: output value of state from the perspective of our agent

Minimax assumes both players play optimally, expanding game tree in DFS with alternating MAX and MIN levels:

i. Time: $O(b^m)$

ii. Space: $O(bm)$

iii. Complete: Yes (for finite games)

iv. Optimal: Yes (for both, if both play optimally)
   - Theorem: if opponent plays sub-optimally, utility obtained by agent is never less than utility obtained against an optimal opponent

Alpha-Beta Pruning reduces nodes evaluated without altering minimax value and optimal move for root node:

i. Maintains α = best value , β = worst value and pruning subtree if $\alpha \geq \beta$

ii. Time: $O(b^{m/2})$ (with perfect ordering)

iii. Space: $O(bm)$ (same as minimax)

iv. Complete & Optimal: same as minimax

Cutoff Strategy halts search in the middle and estimates value of mid-game states with an evaluation function:

i. Evaluation Function: if terminal, use utility function, else use a heuristic

ii. Heuristic Function: estimates utility of a state
   - A heuristic must be between max and min utility
   - Admissibility and consistency are not needed

iii. Theorem: returns a move that is optimal with respect to the evaluation function at the cutoff; may be suboptimal with respect to true minimax

5. Machine Learning

Machine learning develops learning agents with data.

i. Data, $D = \{1 \leq i \leq n : (x^{(i)}, y^{(i)})\}$ of n samples, for $x^{(i)} \in \mathbb{R}^d$ feature vector, $y^{(i)}$ label of i^{th} data point.
ii. Model/Hypothesis, $h : X \to Y$ is a predictor of outputs which is learned by a learning algorithm.
iii. Performance Measure evaluates h map $x^{(i)} \to y^{(i)}$.
    - $MAE = \frac{1}{n}\sum_{i=1}^{n} \left|\hat{y}^{(i)} - y^{(i)}\right|$ (outlier robust)
    - $Accuracy = \frac{1}{n}\sum_{i=1}^{n} \mathbb{1}_{\hat{y}^{(i)}=y^{(i)}}$ (classification)
iv. Loss Function measures $\hat{y}^{(i)}$ error for optimization
    - $MSE = \frac{1}{n}\sum_{i=1}^{n} \left(\hat{y}^{(i)} - y^{(i)}\right)^2$ (outlier sensitive)

Binary Metrics:

i. Accuracy $= \frac{\text{TP + TN}}{\text{TP + FN + FP + TN}}$
ii. Precision, $P = \frac{\text{TP}}{\text{TP + FP}}$ (if FP are costly)
iii. Recall, $R = \frac{\text{TP}}{\text{TP + FN}}$ (if FN are costly)
iv. F1 Score, $F1 = \frac{2}{\frac{1}{P}+\frac{1}{R}}$ (balance precision & recall)

Decision Trees

Decision trees split data using features to predict outputs.

i. Representational Completeness: any decision tree can fit any finite consistent labelled data exactly

Entropy: $H(Y) = -\sum_i P(y_i)\log_2 P(y_i)$

Cond. Entropy: $H(Y \mid X) = \sum_x P(X = x)H(Y \mid X = x)$

Information Gain: $IG(Y;X) = H(Y) - H(Y \mid X)$

DTL grows tree top-down, greedily choosing feature X with highest IG, and recurses. At leaves:

i. If no more data, return default
ii. If data has same classification, return classification
iii. If no more features, return majority class

Pruning reduces overfitting, giving simpler hypothesis by limiting representational capacity, removing outliers:

i. Max-depth: limit max depth of decision tree.
ii. Min-sample leaves: set min samples for leaf nodes.

6. Supervised Learning

Supervised Learning learns from labelled data to learn a mapping from inputs to outputs.

Linear Regression

Linear regression creates a linear model to predict $y \in \mathbb{R}$:

$$h_w(x) = w^T x = w_0 + w_1 x_1 + \cdots + w_d x_d$$

where $w_0, \cdots, w_d$ are weights, using MSE loss function:

$$J_{MSE}(w) = \frac{1}{2n}\sum_{i=1}^{n}(h_w(x^{(i)}) - y^{(i)})^2$$

Learning uses $\frac{\partial J(w)}{\partial w_j} = \frac{1}{n}\sum_{i=1}^{n}(h_w(x^{(i)}) - y^{(i)})x_j^{(i)}$ via:

i. Normal Equation: $w = (X^\top X)^{-1}X^\top Y$
    - Closed form, assumes invertible, $O(d^3)$ inversion
ii. Gradient Descent: $w_j \leftarrow w_j - \gamma\frac{\partial J(w)}{\partial w_j}$ repeated
    - Converge on global min., for appropriate $\gamma > 0$
    - If features linearly indep., global min is unique
    - May need feature scaling/ different γ_j for weights
    - Stochastic/ Mini-batch faster by using less data

Logistic Regression

Logistic regression creates a model to classify $y \in \{0, 1\}$:

$$h_w(x) = \sigma(w^T x), \quad \sigma(z) = \frac{1}{1 + e^{-z}}$$

where w is the weight vector and $\sigma(z)$ is sigmoid function with $\sigma'(z) = \sigma(z)(1 - \sigma(z))$, using BCE loss function:

$$J_{BCE} = \frac{1}{n}\sum_{i=1}^{n} BCE(y^{(i)}, h_w(x^{(i)}))$$

$$BCE(y, p) = -y\log(p) - (1 - y)\log(1 - p)$$

Model $h_w(x)$ can be interpreted as $P(y = 1 \mid x)$, where we classify $y = 1$ if $h_w(x) \geq \tau$, decision boundary $h_w(x) = \tau$

Learning only via Gradient Descent with same properties.

Feature Transformation $x \in \mathbb{R}^d \to \phi(x) \in \mathbb{R}^M$ allows linear and logistic regression on non-linear relationships.

Multi-class Classification ($y \in \{1, \cdots, K\}$)

One-vs-One classifies every pair of classes:

i. $\binom{K}{2}$ classifiers choosing between class I and J
ii. Each classifier votes for a class, most votes wins

One-vs-Rest classifies per class vs. all others:

i. K classifiers choosing between class I and not-I
ii. Classifier with highest probability determins class

Multi-label Classification ($y \in \{0, 1\}^K$)

Binary Relevance uses independent label binary classifiers:

i. Ignores label correlations and constraints (e.g. mutual exclusive)

Classifier Chains feeds predictions of previous binary labels classifiers as features for subsequent labels:

i. Captures label dependencies; sensitive to chain order

Label Powerset treats each unique label-set as a single multi-class label:

i. Preserves correlations; can explode in classes and be data-sparse

Generalization

Generalization is the model's performance on unseen data.

Dataset Factors:

i. Relevance: features should be relevant to problem
ii. Noise: outliers can hinder learning, generalization
iii. Balance (for classification): ensures all classes are fairly represented for generalization over classes

Model Complexity (optimal complexity minimises error):

i. Low Complexity: underfits for complex data, high bias; low variance on retrains across different training sets
ii. High Complexity: overfits for scarce data; low bias with enough data; high variance across retrains on different training sets

Support Vector Machines

7. Neural Networks

8. Unsupervised Learning

Unsupervised Learning learns from unlabelled data to find patterns or structure.

K-Means Clustering

Hierarchical Clustering

Dimensionality Reduction

9. Ethics

```
def alpha_beta_search(state):
    v = max_value(state, -∞, ∞)
    return action in successors(state) with value v

def max_value(state, α, β):
    if is_terminal(state): return utility(state)
    v = -∞
    for next_state in expand(state):
        v = max(v, min_value(next_state, α, β))
        α = max(α, v)
        if v >= β: return v
    return v

def min_value(state , α, β):
    if is_terminal(state): return utility(state)
    v = ∞
    for next_state in expand(state):
        v = min(v, max_value(next_state, α, β))
        β = min(β, v)
        if v <= α: return v
    return v
```

a-B pruning Steps:

i. Start at root $\alpha = -\infty, \beta = \infty$
ii. When going down, pass down the values of α, β
iii. When going up, pass up the value of α if MAX, or β if MIN
iv. When taking a value up, store $\alpha = \max(\alpha, value)$ if MAX, or $\beta = \min(\beta, value)$ if MIN
v. At any point, if there is $\alpha \geq \beta$ prune all other children

Worst-case

Name	Time Complexity[1]	Space Complexity[1]	Complete?	Optimal?
Breadth-first Search	Exponential	Exponential	Yes	Yes
Uniform-cost Search	Exponential	Exponential	Yes[3]	Yes[3]
Depth-first Search	Exponential	Polynomial	No | Yes[4]	No
Depth-limited Search	Exponential	Polynomial[2]	No	No[2]
Iterative Deepening Search	Exponential	Polynomial[2]	Yes	Yes
A* Search	Exponential	Exponential	Yes[3]	Yes if admissible
A* Search with visited memory	Exponential	Exponential	Yes[3]	Yes if consistent

1) In terms of some notion of depth/tier
2) If used with DFS
3) If edge costs are positive
4) With visited memory