1 Intelligent Agents

PEAS: Performance Measure, Environment, Actuator, Sensors

Properties of Task Environment: (1) Fully (Sensors give access to complete state of environment) vs partially observable (2) Deterministic (next state is completely determined by current state and action taken) vs stochastic (3) Strategic: if the environment depends on actions of other agents, unless the other agents are predictable (unintelligent) (4) Episodic vs sequential, (5) Static (environment is unchanging while agent thinks) vs dynamic (environment changes with time but agent's score does not), (6) Discrete (limited number of percepts and actions) vs continuous, (7) Single vs multi-agent

Agent Function: Maps from percept histories P to actions A, $f : P \to A$. An agent program is the implementation of an agent function. An agent is **completely** specified by the agent function.

Agent Structures: 1. *Simple Reflex*: if-else actions, 2. *Goal based*, 3. *Utility Based*, 4. *Learning Agent*

Exploration vs Exploitation: Learn about the world vs maximize gain based on current knowledge

2 Uninformed/Informed Search

Completeness: Search always find a solution. **Optimal**: If the search produces a solution, it is the best one

Uninformed Search: No clue how good a state is i.e. how close to goal

Algo: Create frontier, insert initial state to frontier. While frontier is not empty, pop node from frontier. If node is goal, return solution. Else, for each available action of current state, generate next state using transition model and add that to the frontier. If the frontier becomes empty and the goal is not reached, return failure

BFS: FIFO Queue. **Uniform Cost Search (UCS)**: priority queue. **DFS**: stack.

Search with Visited Memory: Maintain set of visited states. When a node is popped, if it is in visited memory, continue. Else, add it to the visited states.

Depth Limited Search (DLS): Limit depth to l, backtrack when hit. Can be used with BFS/UCS/DFS.

Iterative Deepning Search (IDS): Search with depth $= 0, 1, \ldots,$ return solution if found

Name	Time	Space	Complete	Opt
BFS	Exp	Exp	Y	Y
UCS	Exp	Exp	Y	Y
DFS	Exp	Poly	N	N
DLS	Exp	Poly*	N*	N*
IDS	Exp	Poly*	Y	Y

* If used with DFS

Informed Search: has a clue how good the state is using a heuristic function

Best-first search: Priority queue but using $f(n)$ to evaluate cost

A* search: $f(n) = g(n) + h(n)$ where $g(n)$ is the cost to reach n and $h(n)$ is the estimate distance from n to the goal

Admissible: iff for every node n, $h(n) \leq h^*(n)$ where $h^*(n)$ is the optimal path cost to reach the goal state from n. If $h(n)$ is admissible, A* search without visited memory is optimal.

Consistent: iff for every node n, every successor n' generated by any action a, $h(n) \leq c(n, a, n') + h(n')$ and $h(G) = 0$. If $h(n)$ is consistent, A* search with visited memory is optimal

Dominance: if $h_1(n) \geq h_2(n)$ for all n, then $h_1(n)$ dominantes $h_2(n)$. $h_1(n)$ is better for search if it is admissible.

3 Local/Adversarial Search

Local Search: For problems with too big search space. Search local neighbour states using perturbation or construction. Typically incomplete and suboptimal. **Any-time** property: longer runtime, better solution. Can obtain "good enough" solution

Pertubative search: Search space = complete solutions, search step = modification of solution.

Constructive search: search space = partial solutions, search step = extend solution.

Problem Formulation: States (may not be actual states), Initial State, Goal Test (optional), Successor function (generate neighbours)

Evaluation function: calculate quality of current state

Hill Climbing: Generate neighbours of current state and pick the one with highest evaluation score. If all neighbours score are lower than current state's score, return current state. Else, update current state to highest score neighbour. Repeat until current state has the highest score

Adversarial Search – Multi-agent problems: Cooperative (agents work towards same goal), Competitive (agents have opposing goals), Mixed-motive (some cooperation and competition)

Adversarial Games: Players compete against each other. Two player zero-sum games: only one winner

Problem Formulation: States, Initial State, Terminal State (win/lose/draw), Actions, Transition, Utility Function (value of a state from one agent's perspective)

Minimax: Algorithm for two-player zero sum game. Player A: max player (try to maximize score). B: min player

```
def minimax(state):
  v = max_value(state)
  return action in expand(state)
    with value v
def max_value(state, a, b):
  if is_terminal(state):
    return utility(state) v = -inf
  for next_state in expand(state):
    v = max(v, min_value(next_state))
    a = max(a, v)
    if v >= b:
      return v
  return v
def min_value(state, a, b):
  if is_terminal(state):
    return utility(state) v = inf
  for next_state in expand(state):
    v = min(v, max_value(next_state))
    b = min(b, v)
    if v <= a:
      return v
  return v
```

Time: Exponential $O(b^m)$, branching factor b, max depth m. Space: polynomial. Complete: Yes, if tree is finite. Optimal: Yes, against optimal opponent. If B is not optimal, A's action never has an utility lower than the utility against an optimal opponent, but a faster path can be taken to win, hence A's actions might be suboptimal

AB Pruning: α = largest value found so far for MAX, β = smallest value found so far for MIN. Pruning doesn't affect the final result, but brings the TC down to $O(b^{m/2})$.

4 Machine Learning

Supervised Learning: Agent learns from labelled data (input-output pairs) and aims to learn mapping from inputs to outputs.

Unsupervised Learning: Agent learns from unlabelled data (input only) and aims to find patterns or structure.

Regression Error: The difference between the predicted ($\hat{y}$) and true value (y), where $\hat{y}^{(i)} = h(x^{(i)})$

Mean Squared Error:
MSE $= \frac{1}{N} \sum_{i=1}^{N} (\hat{y}^{(i)} - y^{(i)})^2$

Mean Absolute Error:
MAE $= \frac{1}{N} \sum_{i=1}^{N} |\hat{y}^{(i)} - y^{(i)}|$

Classification Correctness/Accuracy: Correct when $\hat{y} = y$. Accuracy = average correctness $= \frac{1}{N} \sum_{i=1}^{N} \mathbb{1}_{\hat{y}^{(i)} = y^{(i)}}$ (1 if predicted = true value)

Entropy: Given outcomes $v_1, \ldots, v_k$ and their probabilities $P(v_1), \ldots, P(v_k)$, $I(P(v_1), \ldots, P(v_k)) = -\sum_{i=1}^{k} P(v_i) \log_2 P(v_i)$
For p positive and n negatives, $P(pos) = \frac{p}{p+n}$, $P(neg) = \frac{n}{p+n}$, $I(P(pos), P(neg)) = -\frac{p}{p+n} \log_2 \frac{p}{p+n} - \frac{n}{p+n} \log_2 \frac{n}{p+n}$

Information Gain: If attribute A divides the training set into $E_1, \ldots, E_v$ subsets, $IG(A) = I(\frac{p}{p+n}, \frac{n}{p+n}) - remainder(A)$ where $remainder(A) = \sum_{i=1}^{v} \frac{p_i + n_i}{p+n} I(\frac{p_i}{p_i+n_i}, \frac{n_i}{p_i+n_i})$

Decision Tree: Basically, nested if-else. Used for both classification and regression. Greedily pick attribute with maximal IG at each node. **Overfitting**: DT captures

noise, performs perfectly on training data but worse on test data. **Occam's Razor**: prefer short/simple hypothesis (more likely) over long/complex hypothesis. **Pruning**: Min-sample leaf (min. number of samples to be a leaf node), Max-depth (limit depth of DT)

Data Preprocessing: Continuous values (partition values into intervals), Missing values: assign most common values, assign probabilities to values, drop rows/attributes

5 Linear Regression

Regression: Given a data point x and **no** target, find a function that **predicts** the target y for that x.

Linear Model: The *hypothesis class* of linear models is defined as the set of functions $h_w(x) = w_0x_0 + w_1x_1 + \cdots + w_dx_d$, where $w_0, \ldots, w_d$ are **parameters/weights** and $x_0 = 1$ is a dummy variable. Shorthand: $h_w(x) = w^Tx$. To measure the 'fit' of a hypothesis, use a loss function to calculate the difference in predicted vs actual value. We want to find a w that minimizes h.

Normal Equation: Gives the best weights w that minimizes J_{MSE}, assuming X^TX is invertible i.e. of full rank. If not invertible $\rightarrow$ 1. regularize, or 2. compute approximation using Moore-Penrose (least squares) inverse

Problems with Normal Equation: Cost of interving matrix is high (d^3): does not scale well for large matrices + Will not work for non-linear models + Assumes invertibility.

Partial Derivative of Linear Model: $\frac{\partial}{\partial w_j} h_w(x^{(i)}) = \frac{\partial}{\partial w_j}(w^Tx^{(i)}) = x_j^{(i)}$

Partial Derivative of MSE: $\frac{\partial J_{MSE}(w)}{\partial w_j} = \frac{1}{2N}\frac{\partial}{\partial w_j}\sum_{i=1}^{N}(h_w(x^{(i)}) - y^{(i)})^2 = \frac{1}{2N}\sum_{i=1}^{N}\frac{\partial}{\partial w_j}(h_w(x^{(i)}) - y^{(i)})^2 = \frac{1}{N}(h_w(x^{(i)}) - y^{(i)})x_j^{(i)}$. Minimum when $\frac{\partial J_{MSE}(w)}{\partial w_j} = 0$, i.e. $w = (X^TX)^{-1}X^TY$. (Note: X has a bias column)

Gradient Descent: Move in the opposite direction of gradient i.e. towards lower loss.

Theorem: MSE loss function is **convex** for linear regression (and polynomial regression, with feature transformations).

Features of Different Scales: Normalize/Standardize - $x_j \leftarrow \frac{x_j - \mu_j}{\sigma_j}$, where σ_j is the std deviation of the feature j across the training data. Alternatives: min-max scaling, robust scaling, different learning rates γ_j for each parameter.

Variants of GD: *Mini-batch*: use a subset of whole training data per epoch, *Stochastic*: use a random point per epoch. Faster, more randomness, may escape local minima.

GD vs NE: GD - Need to choose learning rate γ, many iterations, performs well for large d, may need feature scaling. NE - slow for large d, X^TX must be invertible, no iterations or γ, no feature scaling needed.

6 Logistic Regression

Classification: Given a new data point $x \in \mathbb{R}^d$ and **no** target, based on existing dataset, find a function that predicts the target $y \in \{0, 1\}$ for that x

Squashing Models - Sigmoid Function Maps a real number to a value between 0 and 1. $\sigma(x) = \frac{1}{1+e^{-x}}$. Derivative: $\sigma'(x) = \sigma(x)(1 - \sigma(x))$ a.k.a the **logistic function**

The curve of $\sigma(c + x)$ where $c > 0$ is a shift left

Logistic Regression Model: $h_w(x) = \sigma(w_0x_0 + w_1x_1 + \cdots + w_dx_d) = \sigma(w^Tx)$ where $w_0, \ldots, w_d$ are weights, and $x_0 = 1$ is a dummy variable.

The output of this model can be treated as the **probability of an input to be of class 1**. To decide whether an input belongs to a certain class, compare the probability to a **decision threshold**, e.g. 0.5.

Decision Boundary: the surface (or line, hyperplane) that separates different classes in the feature space

Non-linearly Separable Data: when a single linear decision boundary e.g. straight line cannot effectively separate the classes

Theorem: MSE loss function is **non-convex** for logistic regression, so it can't be used.

Binary Cross Entropy (BCE): Given the probability value $y \in [0, 1]$ and $\hat{y} \in [0, 1]$, $BCE(y, \hat{y}) = -y\log(\hat{y}) - (1 - y)log(1 - \hat{y})$

A.k.a logistic loss / log loss. If y and $\hat{y}$ are close, BCE loss is small. Else it is large.

Mean BCE: $J_{BCE}(w) = \frac{1}{N}\sum_{i=1}^{N} BCE(y^{(i)}, h_w(x^{(i)}))$

Theorem: BCE loss function is **convex** for logistic regression

Logistic Regression with Gradient Descent

Hypothesis: $h_w(x) = \sigma(w_0 + w_1x_1 + w_2x_2)$

Loss function: $J_{BCE}(w) = \frac{1}{N}\sum_{i=1}^{N} BCE(y^{(i)}, h_w(x^{(i)}))$

Weight Update: $w_j \leftarrow w_j - \gamma\frac{\partial J_{BCE}(w_0, w_1, \ldots)}{\partial w_j}$

Derivative: $\frac{\partial J_{BCE}(w)}{\partial w_j} = \frac{\partial}{\partial w_j}\frac{1}{N}\sum_{i=1}^{N} BCE(y^{(i)}, h_w(x^{(i)})) = \frac{1}{N}\sum_{i=1}^{N}(h_w(x^{(i)}) - y^{(i)})x_j^{(i)}$ (same as LR)

Multi-class Classification: Given N data points, predict the target $y \in \{1, 2, \ldots, C\}$ where C is the number of classes.

One-vs-One: A separate binary classifier is trained for **every pair of classes**. For C classes, this results in $\frac{C(C-1)}{2}$ classifiers. E.g. $(1, 2), (1, 3), (2, 3), \ldots$ During prediction, each classifier votes for a class; the class with the most votes is selected

One-vs-Rest: A separate binary classifier is trained for each class, **treating all other classes as a single combined class**. For C classes, this results in C classifiers. E.g. 1 and not 1, 2 and not 2. During prediction, the classifier with **highest confidence score** (probability output) determines the class

Generalization: The ability to perform well on **unseen data**. Generalization error: error on unseen data. Two factors affect generalization: 1. *Dataset quality and quantity*, 2. *Model complexity*

Dataset Quality: 1. *Relevance* - data should be relevant for the problem, 2. *Noise* - incorrect data can hinder model's learning, 3. *Balance* – each class should be adequately represented

Dataset Quantity: More data is typically better. **Extreme case**: if the dataset contains every possible data point, the model can simply *memorize* all the data

Model Complexity: Refers to the *size* and *expressiveness* of the hypothesis class. Higher complexity allows for more sophisticated modeling e.g. polynomial vs linear. Low complexity: $h_1(x) = \sigma(w_0 + w_1x)$. Medium complexity: $h_2(x) = \sigma(w_0 + w_1x + w_2x^2)$. High complexity: $h_3(x) = \sigma(w_0 + w_1x + w_2x^2 + w_3x^3 + \ldots)$

Underfitting: Low complexity model cannot capture data

Overfitting: High complexity model may capture noise of the data

Bias: High bias = more data points does not lead to a different model. Low bias = more data points leads to a more complex model

Variance: Low variance = retraining model with different training set based on the same ground truth leads to the same model. High variance = retraining a complex model with a different training set based on the same ground truth leads to a very different model

Model Complexity vs Error: High complexity: Error on unseen data decreases with more data points. Low complexity: error on unseen data remains roughly the same regardless of data quantity. **Overparameterized** (deep neural networks): Error on unseen data decreases initially, peaks once before decreasing again

Hyperparameters: The settings that control the behaviour of the training algorithm and model. They are not learned. They **need to be set before the training process**. E.g. learning rate, feature transformations, batch size and no. of iterations

Parameters: learned **during** training e.g. weights in a linear model

Hyperparameter Tuning: Optimizing the hyperparameters to improve model performance, a.k.a hyperparameter search. Techniques: *Grid (exhaustive) search*, *Random search*, *Local search (hill climbing)*, *Successive halving*, *Bayesian optimization*