General Formula

1 Agents

Have:
- Abilities
- Goals/Preferences
- Prior Knowledge
- Stimuli
- Past Experiences
- Actions

Belief State: Internal belief about the world
Knowledge: Information used to solve tasks
Representation: Data structure to encode knowledge
Knowledge Base: Representation of all knowledge possessed
Model: How KB relates to world

Dimensions of Complexity

1. Modularity
   - Flat: No modularity in computation
   - Modular: Each component is separateand siloed
   - Hierarchical: Modular components are broken into a hierarchical manner of subproblems
2. Planning Horizon:
   - Non-Planning: World doesn't change as a result (Ex: Protein Folding)
   - Finite: Reason ahead fixed number of steps
   - Indefinite: Reason ahead finite number (but undetermined) of steps
   - Infinite: Reason forever (focus on process)
3. Representation:
   - States: State describes how world exists
   - Features: An attribute of the world
   - Individuals and relations: How features relate to one another (Eg: child.failing() relates to child.grade)
4. Computational limits:
   - Perfect rationality: Agent always picks best action (Eg: Tic-Tac-Toe)
   - Bounded rationality: Agent picks best action given limited computation (Eg: Chess)
5. Learning:
   - Given knowledge (Eg Road laws)
   - Learned knowledge (Eg How car steers in rain)
6. Uncertainty:
   - Fully observable: Agent knows full state of world from observations (Eg: Chess)
   - Partially observable: Many states can lead to same representation (Eg: Battleship)
   - Deterministic: Action has predictable effect
   - Stochastic: Uncertainty exists over effect of action to state
7. Preference:
   - Achievement Goal: Goal to reach (binary)
   - Maintenance Goal: State to maintain
   - Complex Preferences: Complex tradeoffs between criteria and ordinality (can't please everyone)
8. Num Agents:
   - Single agent
   - Adversarial
   - Multiagent
9. Interactivity:
   - Offline: Compute its set of actions before agent has to act, so no computations required
   - Online: Computation is done between observing and acting

2 Search

b is branching factor (max num children of any node)
m is max depth of search-tree
d is depth of shallowest goal node

Search Problem:
- Set of states
- Initial state
- Goal function
- Successor function
- (optional) cost

Frontier: Ends of paths from start node that have been explored

Graph Search Algorithm

```
frontier is just start node
while frontier isn't empty
    select and remove path $\langle n_0, ..., n_k \rangle$ from frontier
    if goal($n_k$) then
        return $\langle n_0, ..., n_k \rangle$
    for each neighbor $n$ of $n_k$ do
        add $\langle n_0, ..., n_k, n \rangle$
```

Uninformed Search

Depth-first-search

Use frontier as stack, always select last element
Cycle Check: Check if current node exists within path you are Checking
Space Complexity: $O(bm)$
Time Complexity: $O(b^m)$
Completeness: No
Optimal: No

Use:
- Restricted space
- Many solutions with long paths

Don't use:
- Infinite paths exist
- Optimal solutions are shallow
- Multiple paths to node

Breadth-first-search

Use frontier as queue, always select first element
Multiple-Path Pruning: Check if current node has been visited by any previous path by maintaining explored set.
Space Complexity: $O(b^d)$
Time Complexity: $O(b^d)$
Completeness: Yes
Optimal: No (only finds shallowest goal node)

Use:
- Space isn't restricted
- Want shallowest arc

Don't use:
- All solutions are deep
- Problem is large and graph is dynamically generated

Iterative-Deepening

For every depth-limit, perform depth-first-search, this marries BFS and DFS by "doing BFS" but without space-concerns. However, we end up revisiting nodes.

Space Complexity: $O(bd)$
Time Complexity: $O(b^d)$, $b^d \sum_{n=1}^{d} n\left(\frac{1}{b}\right)^{n-1} = b^d \left(\frac{b}{1-b}\right)^2$ We visit level i $d-i$ times, and level i has b^i nodes, so that's the sum, then extending to infinity, and use geometric series.
Completeness: Yes
Optimal: No (only finds shallowest goal node)

Use:
- Space isn't restricted
- Want shallowest arc

Don't use:
- All solutions are deep
- Problem is large and graph is dynamically generated

Lowest-Cost-First-Search

Select a path on frontier with lowest cost. Frontier is priority queue ordered by path cost. *Technically uninformed/blind search because it's searching randomly*
Space Complexity: $O(b^d)$
Time Complexity: $O(b^d)$
Completeness and optimality: Yes if branching factor is finite and cost of every edge is strictly positive.
Termination: Only terminate when the goal node is first on the frontier, not if it's in the frontier.

Dijkstra's

Similar to LCFS but keep track of lowest cost to reach each node, if we find lower cost path, update that value and resort the priority queue. Ex: Suppose we have path in frontier $\langle P, Q, R, \rangle$ and the found path to Q is 10 and overall cost is 12, then we find a new path to Q of cost 9, then we should recompute path to get cost 11.

Heuristic Search

$h(n)$ is estimate of cost of shortest path from n to a goal node. Should only use readily obtainable information and be **much** easier than solving the problem.

Greedy Best-First Search

Select path whose end is closest to a goal based on heuristic. Frontier is a priority queue ordered by h.
Space Complexity: $O(b^d)$
Time Complexity: $O(b^d)$
Completeness and optimality: Not guaranteed (could be stuck in a cycle or return sub-optimal path)

Heuristic Depth-First-Search

Do Depth-First-Search, but add paths to stack ordered according to h. Basically, do DFS, but sort the children by h to determine who to check. Same complexity and problems as DFS but used often.

A* search

Use both path cost and heuristic values. Frontier is sorted by $f(p) = \text{cost(p)} + h(p)$. Always selects node with lowest estimated distance.

Space Complexity: $O(b^d)$
Time Complexity: $O(b^d)$
Completeness and optimality: Only with admissable heuristic, finite branching factor, and bounded arc-costs (there is a minimum positive arc-cost). A* always expands the fewest nodes for all optimal algorithms and use the same heuristic. No algorithm with same info can do better. This is because if an algorithm does not expand all nodes with $f(n) < \text{cost}(s, g)$ they might not find the optimal solution.

Admissable Heuristic

Never overestimates the shortest path from n to goal node.

Procedure for construction:
1. Define relaxed problem by simplifying or removing constraints
2. Solve relaxed problem without search
3. Cost of optimal solution to relaxed problem is admissable heuristic for original problem.

Dominating Heuristic

Given two heuristics, $h_2(n)$ dominates $h_1(n)$ if $\forall n h_2(n) \geq h_1(n)$ and $\exists n h_2(n) > h_1(n)$ We prefer dominating heuristics because it reduces the nodes we have to expand (they're bigger, so we don't care)

Monotone Restriction

A* guarantees finding optimal goal, but not necessarily shortest path. In order to do that, we would want our estimate path $f(p)$ to indeed allow us to remove longer paths, but what if one path has shorter cost, but heuristic sums make the shorter path have larger $f(p)$? We can avoid that, by inducing monotonic restriction. $h(n') - h(n) \leq \text{cost}(n', n)$. This guarantees heuristic estimate is always less than actual cost and if we ever find a shorter estimate, that estimate will actually be shorter, so we can prune it.

Further, monotonic restriction with multi-path pruning always finds shortest path to goal, not just optimal goal itself.

Note that admissability guarantees heuristic is never bigger than shortest path to goal, monotonicity ensures heuristic is never bigger than shortest path to any other node.

Summary

Strategy	Frontier Selection	Halt?	Space	Time
Depth-first	Last node added	No	Linear	Exp
Breadth-first	First node added	Yes	Exp	Exp
Heuristic Depth-first	Local min $h(n)$	No	Linear	Exp
Best-first	Global min $h(n)$	No	Exp	Exp
Lowest-cost-first	min cost(n)	Yes	Exp	Exp
A*	min $f(n)$	Yes	Exp	Exp

Adversarial Search (Minimax)

For one node look to maximize the heuristic, for the other node look to minimize it (to simulate the adversarial search)

- Alpha-beta pruning can ignore portions of search tree without losing optimality. Useful in application but doesn't change asymptotics
- Can stop early by evaluating non-leafs via heuristics (doesn't guarantee optimal play)

Higher-level strategies

Bidirectional Search: Search from backward and forward simultaneously taking $2b^{\frac{k}{2}}$ vs b^k and try to find where frontiers match
Island-driven Search: Find set of islands between s and g as mini problems. With m islands, you get $mb^{\frac{k}{m}}$ vs b^k but it's harder to guarantee optimality.

3 Constraints

Constraint Satisfaction Problems

- A set of variables
- Domain for each variable
- Two kinds of problems:
  - Satisfiability problems: Assignment satisfying hard constraints
  - Optimizatoin: Find assignment optimizing evaluation function (soft constraints)
- Solution is assignment to variables satisfying all constraints
- Solution is model of constraints

CSPs as graphs

Search spaces can be very large, path isn't important, only goal, and no set starting nodes make this bad idea
Complete Assignment: Nodes: Assignment of value to all variables
Neighbors: Change one variable value
Partial Assignment: Nodes: Assignment to first $k-1$ variables
Neighbors: Assignment to k^{th} variable

Constraints

- Can be **N-ary** (over sets of N variables) (Ex: A + B = C involves is 3-ary for 3 vars)

Generate and Test

Exhaust every possible assignment of vars and test validity

Backtracking

Order all variables and evaluate constraints in order as soon as they are fixed. (Ex: $A = 1 \wedge B = 1$ is inconsistent with $A \neq B$ so go to last assigned variable and change its value)

Consistency

Represent constraints as network to determine how all variables are related.
Domain Constraint: Unary constraint on values in domain written $\langle X, c(X) \rangle$ (Eg: $B, B \neq 3$)
Domain Consistent: A node is domain consistent if no domain value violates any domain constraint, and a network is domain consistent if all nodes are domain consistent.
Arc: Arc $\langle X, c(X, Y) \rangle$ is a constraint on X
Arc Consistent: Arc $\langle X, c(X, Y) \rangle$ is arc consistent if for every valid x there is a valid y such that constraint is satisfied.
Path Consistent: A set of variables is path consistent if all arcs and domains are consistent.

AC-3

Make Consistency network arc consistent
- To-Do Arcs Queue contains all inconsistent arcs
- Make all domains domain consistent
- Put all arcs in TDA
- Repeat until TDA is empty:
  - Select and remove an arc from TDA
  - Remove all values of domain of X that don't have value in domain of Y that satisfy constraint
  - If any were removed, add all arcs to TDA

Termination:
- If every domain is empty, no solution
- If every domain has a single value, solution
- If some domain has more than one value, split in two run AC-3 recursively on two halves
- Guaranteed to terminate
- Takes $O(cd^3)$ time, with n variables, c binary constraints, and max domain size is d because each arc $\langle X_k, X_i \rangle$ can be added to queue at most d times because we can delete at most d values from X_i. Checking consistency takes $O(d^2)$ time.

Variable Elimination

- Eliminate variables one-by-one passing constraints to neighbours.
- When single variable remains, if no values exist then network was inconsistent.
- Variables are eliminated according to elimination ordering.

Pseudocode:
- If only one variable, return intersection of unary constraints referencing it
- Select variable X
  - Join constraints affecting X, forming constraint R
  - Project R onto its variables other than X, calling this R2
  - Place new constraint between all variables that were connected to X
  - Remove X
  - Recursively solve simplified problem
  - Return R joined with recursive solution

Local Search

- Maintain assignment of value to each variable
- At each step, select neighbor of current assignment
- Stop when satisfying assignment found or return best assignment found
- Heuristic function to be minimized: Number of conflicts

- Goal is an assignment with zero conflicts

Greedy Descent

Select some variable (through some method) and then select the value that minimizes the number of conflicts. THe problem is that we could be stuck in a local minimum, without reaching the proper global minimum.

Stochastic Local Search

Do Greedy descent, but allow some steps to be random, and the potential to restart randomly, to minimize potential for being stuck in local minimum.

Problem: in high dimensions often consist of long, nearly flat “canyons” so it’s hard to optimize using local search.

Simulated Annealing

Pick variable at random, if it improves, adopt it. If it doesn’t improve, then accept it with a probability through the temperature parameter, which can get slowly reduced.

Tabu Lists

Variant of Greedy Satisfiability, where to prevent cycling and getting stuck in local optimum, we maintain a “tabu list” of the k last assignments, and don’t allow assignment that has already existed.

Parallel Search

- Total assignment is called individual
- Maintain population of k individuals
- At each stage, update each individual in population
- Whenever individual is a solution, it can be reported
- Similar to k restarts, but uses k times minimum number of steps

Beam Search

- Like parallel search, with k individuals, but choose the k best out of all the neighbors. The value of k can limit space and induce parallelism

Stochastic Beam Search

- Like beam search, but probabilistically choose k individualls at next generation. Probability of selecting neighbor is proportional to heuristic: $e^{-\frac{h(n)}{T}}$. This maintains diversity among the individuals, because it’s similar to simulated annealing.

Genetic Algorithms

- Like stochastic beam search, but pairs of individuals are combined to create offspring.
- For each generation, randomly choose pairs where fittest individuals are more likely selected
- For each pair, do cross-over (form two offspring as mutants of parents)
- Mutate some values
- Stop when solution is found

Comparing Algorithms

Since some algorithms are super fast some of the time and super slow other times, and others are mediocre all of the time, how do you compare? You use runtime distribution plots to see the proportion of runs that are solved within a specific runtime.

4 Inference and planning

Problem Solving

Procedural solving: Devise algorithm, program, execute
Declarative solving: Identify required knowledge, encode knowledge in representation, use logical consequences to solve.

Logic

Syntax: What is an acceptable sentence
Semantics: What do the sentences and symbols mean?
Proof procedure: How to construct valid proofs?
Proof: Sequence of sentences derivable using an inference recursively

Statements/Premises: $\{X\}$ is a set of statements or premises, made up of propositions.
Interpretation: Set of truth assignments to propositions in $\{X\}$
Model: Interpretation that makes statements true
Inconsistent statements: No model exists
Logical Consequence: If for every model of $\{X\}$, A is true, then A is a logical consequence of $\{X\}$

Argument Validity is satisfied if any of the identical statements are true:
- Conclusions are a logical consequence of premises
- Conclusions are true in every model of premises
- No situation in which the premises are all true but the conclusions are false.
- Arguments $\rightarrow$ conclusions is a **tautology** (always true)

Proof

Knowledge Base: Set of axioms
Derivation: KB $\vdash g$ can be found from KB using proof procedure
Theorem: If KB $\vdash g$, then g is a theorem
Sound: Proof procedure is sound if KB $\vdash$ g then KB $\vDash g$ (anything that can be proven must be true (sound reasoning))
Complete: Proof procedure is sound if KB $\vDash$ g then KB $\vdash g$ (anything that is true can be proven (complete proof system))

Complete Knowledge: Assume a closed world where **negation** implies failure since we can’t prove it, if it’s open there are true things we don’t know, so if we can’t prove something, we can’t decide if it’s true or false.

Bottom-up Proof (aka forward chaining)

Start from facts and use rules to generate all possible derivable propositions

To prove: $F \leftarrow A \wedge E$ $A \leftarrow B \wedge C$ $A \leftarrow D \wedge C$ $E \leftarrow C$ D C

Steps of proof: $\{D, C\} \rightarrow \{D, C, E\} \rightarrow \{D, C, E, A\} \rightarrow \{D, C, E, A, F\}$ Therefore, if g is an atom, KB $\vdash$ g, if g $\in C$ at the end of the procedure, where C is the consequence set.

Top-Down

Start from query and work backwards yes $\leftarrow F$ yes $\leftarrow A \wedge E$ yes $\leftarrow D \wedge C \wedge E$ yes $\leftarrow D \wedge C \wedge C$ yes $\leftarrow D \wedge C$ yes $\leftarrow D$ yes $\leftarrow$

Individuals and Relations

KB can contain **relations**: part_of(C, A) is true if C is a “part of” A
KB can contain **quantification**: part_of(C, A) holds $\forall C, A$ Proofs are the same with extra bits for handling relations & quantification.

Planning

Decide sequence of actions to solve goal based on abilities, goal, state of the world Assumptions:
- Single agent
- Deterministic
- No exogenous events
- Fully-observable state
- Time progresses discretely from one state to another
- Goals are predicates of states to achieve or maintain (no complex goals)

Action: Partial function from state to state
Partial Function: Some actions are not possible in some states, preconditions specify when action is valid, and effect determines next state

State Representations

Feature-based representation of actions: For each action, there is a precondition (proposition) that specifies when action is valid and a set of consequences for features after action.
State-based representation: For each possible assignment of features, define a state. Then for each action define the starting and ending state for the state-based graph.

Causal Rule: When feature gets a new value
Frame Rule: When feature keeps its value *Features are capitalized, but values aren’t If X is a feature, X’ is feature after an action*

Forward Planning: Search in state-space graph, where nodes are states, arcs are actions, and a plan is a path representing initial state to goal state.
Regression Planning: Search backwards from goal, nodes correspond to subgoals and arcs to actions. Nodes are propositions (formula made of assignment of values to features), arcs are actions that can achieve one of the goals. Neighbors of node N associated with arc specify what must be true immediately before A so that N is true immediately after. Start node is goal to be achieved. Goal(N) is true if N is a proposition true of initial state.

5 Learning

Learning: Ability to improve behaviour based on experience. Either improve range (more abilities), accuracy, or speed.

Components of learning problem:
- Task: Behaviour to improve (Ex: Classification)
- Data: Experiences used to improve performance
- Improvement metric

Common Learning Tasks:
- Supervised classification: Given pre-classified training examples, classify new instance
- Unsupervised learning: Find natural classes for examples
- Reinforcement learning: Determine what to do based on rewards and punishments
- Transfer Learning: Learn from expert
- Active Learning: Learner actively seeks to learn
- Inductive logic programming: Build richer models in terms of logic programs

Learning by feedback:
- Supervised learning: What to be learned is specified for each example
- Unsupervised learning: No classifications given, learner has to discover categories from data
- Reinforcement learning: Feedback occurs after a sequence of actions

Metrics: Measure success by how well agent performs for new examples, not training.
P agent: Consider this agent that claims negative training data are the only negative instances, and gives positive otherwise (100% training data, bad test)
N agent: Same as N agent, but considers only positive training data. They both have 100% on training data, but disagree on everything else.

Bias: Tendency to prefer one hypothesis over another. Necessary to make predictions on unseen data. You can’t evaluate hypotheses by the training data, it depends on what works best in practice (unseen data).

Learning as search: Given representation and bias, learning is basically a search through all possible representations looking for representation that best fits the data, given the bias. However, systematic search is infeasible, so we typically use a **search space**, **evaluation function**, and a, **search method**

Interpolation: Inferring results in between training examples (Ex: Train on 1,2,4,5. Test on 3)
Extrapolation: Inferring results beyond training examples (Ex: Train on 1, 2, 4, 5. Test on 1000)

Supervised Learning

Given input features, target features, training examples and test examples we want to predict the values for the target features for the test examples.
Classification: When target features are discerete.
Regression: When target features are continuous.

Noise: Sometimes features are assigned wrong value, or inadequate for predicting classification, or missing features.
Overfitting: Distinction appears in data that doesn’t exist in unseen examples (through random correlations)

Evaluating Predictions

$Y(e)$ is value of feature Y for example e $\hat{Y}(e)$ is predicted value of feature Y for example e from the agent.
Error: Prediction of how close $\hat{Y}(e)$ is to $Y(e)$

Types of Features

Real-Valued Features: Values are totally-ordered. Ex: Height, Grades, Shirt-size. (The finitenss of the domain of values is irrelevant)

Categorical Features: Domain is fixed finite set. (Total-ordering is irrelevant). Point estimates are either definitive predictions or probabilistic predictions for each category.

Measures of Error

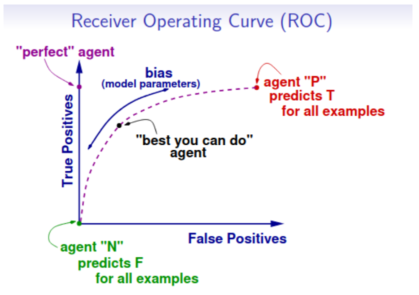
Absolute error: $\sum_{e \in E} \sum_{Y \in T} \left|Y(e) - \hat{Y}(e)\right|$
Sum of squares: $\sum_{e \in E} \sum_{Y \in T} \left(Y(e) - \hat{Y}(e)\right)^2$
Worst-case: $\max_{e \in E} \max_{Y \in T} \left|Y(e) - \hat{Y}(e)\right|$
- Cost-based error takes into account costs of various errors, so some are more costly than others.

Mean Log Loss: For probabilistic predictions, calculate the mean log loss (logloss$(p, a) = -\log(p[a])$) for all predictions (p = pred, a = acc). Mean log loss is a transformation of log-likelihood, so minimizing mean log loss finds highest likelihood.
Binary Log Loss: logloss$(p, a) = -a \log p - (1 - a) \log(1 - p)$ is a variant for boolean features.

Precision and Recall

Recall: Percentage of positive statements that are accurately predicted
Specificity: Percentage of negative statements that are accurately predicted
Precision: Percentage of predicted truths that are correct.

Receiver Operating Curve: A graph between True positives and False Positives



Basic Models: Decision Trees, Linear classifiers (Generalize to Neural Networks), Bayesian Classifiers

6 Decision Trees

- Representation: Decision tree
- Bias: Tendency towards a simple decision tree

Nodes: Input attributes/features
Branches: Labeled with input feature values (can have multiple feature values)
Leaves: Predictions for target features (point estimates) Search through the space of decision trees from simple to complex decision trees

Learning:
- Incrementally split training data
- Recursively solve sub-problems
- Try to get small decision tree with good classification (low training error) and generalization (low test error)

Learn Pseudocode:

```
X is input features, Y output features, E is training examples
Output is decision tree (Either Point estimate of Y or form $\langle X_i, T_1, ..., T_N \rangle$ where $X_i$ is input feature and $T_1, ..., T_N$ are decision trees.
procedure DecisionTreeLearner(X, Y, E):
    if stopping criteria is met then
        return pointEstimate(Y, E)
    select feature $X_i \in X$
        for each value $x_i \in X_i$ do
            $E_i$ = all examples in E where $X_i = x_i$
                                                 $X$
            $T_i$ = DecisionTreeLearner($\{X_i\}, Y, E_i$)
        return $\langle X_i, T_1, ..., T_N \rangle$
```

Classify Pseudocode:

```
X: input features, Y: output features, e: test example, DT: Decision Tree
procedure ClassifyExample(e, X, Y, DT)
    S ← DT
    while S is internal node of form $\langle X_i, T_1, ..., T_N \rangle$
        $j \leftarrow X_{i(e)}$
        $S \leftarrow T_j$
    return S
```

Stopping Criteria:
- Stopping criteria is related to final return value
- Depends on what we will need to do
- Possible: Stop when no more features, or when performance is good enough

Feature Selection: We want to choose sequence of features resulting in smallest tree. Actually we should myopically split, as if only allowed one split, which feature would give best performance.

Heuristics:
- Most even split
- Maximum information gain
- GINI index

Information Theory

Bit is binary digit, n bits classify 2^n items, straight up. But if we use probabilities we can make the average bit usage lower. Eg: P(a) = 0.5, P(b) = 0.25, P(c) = P(d) = 0.125 If we encode a:00, b: 01, c: 10, d: 11, it uses on average 2 bits. If we encode a:0, b: 10, c: 110, d: 111, it uses on average 1.75 bits.

In general we need $-\log_2 P(x)$ bits to encode x, each symbol needs on average $-P(x)\log_2 P(x)$ bits and to transmit an entire sequence distributes according to $P(x)$ we need on average $\sum_x -P(x)\log_2 P(x)$ bits (entropy/information content)

Information Gain

Given a set E of N training examples, if number of examples with output feature $Y = y_i$ is N_i then $P(Y = y_i) = P(y_i) = \frac{N_i}{N}$ (Point estimate)

Total information content for set E is $I(E) = -\sum_{y_i \in Y} P(y_i)\log_2 P(y_i)$

So after splitting E into E_1 and E_2 with size N_1, N_2 based on input attribute X_i, the information content is $I(E_{\text{split}}) = \frac{N_1}{N} I(E_1) + \frac{N_2}{N} I(E_2)$ and we want the X_i that maximizes information gain $I(E) - I(E_{\text{split}})$

Note:

$$-\frac{N_1}{N}\left(\sum_{y\in Y} P(y \mid x=1)\log P(y \mid x=1)\right) - \frac{N_2}{N}\left(\sum_{y\in Y} P(y \mid x=2)\log P(y \mid x=2)\right)$$

$$-\sum_{y\in Y} P(x=1)P(y \mid x=1)\log P(y \mid x=1) - \sum_{y\in Y} P(x=2)P(y \mid x=2)\log P(y \mid x=2)$$

$$-\sum_{x\in X, y\in Y} P(x=x)P(y \mid x=x)\log P(y \mid x=x)$$

$$-\sum_{x\in X, y\in Y} P(x,y)\log\frac{P(x,y)}{P(x)}$$

Final Return Value: **Point estimate** of Y (output features) over all examples. This is just a prediction of target features. (Eg: Mean, Median, most likely classification, $P(Y = y_i) = \frac{N_i}{N}$)

Priority Queue

Basic version grows all branches for a node in decision tree, but it's more efficient to sort leaves via priority queue ranked by how much information can be gained with best feature at that leaf and always expand leaf at top of queue.

Overfitting: When decision tree is too good at classifying training it doesn't generalize well, occurs with insufficient data. Methods to fix:
- Regularization: Prefer small decision trees over big ones (complexity penalty)
- Pseudocounts: Add some data based on prior knowledge
- Cross validation: Partition into training and validation set. Use validation set as test set and optimize decision maker to perform well on validation set, not training set.
- Errors can be caused by:
  - Bias:
    - Represntation bias: Model is too simple
    - Search bias: Not enough search
  - Variance: Error due to lack of data
  - Noise: Error from data depending on features not modeled, or because process generating data is stochastic.
  - Bias-variance trade-off:
    - Complicated model, not enough data (low bias, high variance)
    - Simple model, lots of data (high bias, low variance)
  - Capacity: Ability to fit wide variety of functions (inverse of bias)

Linear Regression

The goal is to fit a linear function a set of real-valued training data. The linear function would be a combination of weights, which we can optimize over by taking the partial derivative of the loss for each weight, setting to 0, and solving the linear system (ex. a la RANSAC)

Squashed Linear Functions

Linear functions don't make sense for **binary classification** because we shouldn't extrapolate beyond the domain $[0,1]$ which linear functions allow us to do. The solution is to use a function with domain $[-\infty,\infty]$ and range $[0,1]$ to transform a linear function onto sensical binary domain.

Activation Function: Funciotn that transforms from real $(-\infty,\infty)$ to subset $[0,1]$

To determine weights for a squashed linear function, you need to minimize the **log loss**.

Log Loss: $\text{LL}(E,w) = -\frac{1}{|E|} \times \sigma_{e\in E}\left(Y(e) \times \log \hat{Y}(e) + (1 - Y(e)) \times \log\left(1 - \hat{Y}(e)\right)\right)$ ($\hat{Y}(e)$) is the predicted value after transformation. To optimize for this function, take partial derivatives for each weight and find their minimal.

Stochastic Gradient Descent

The goal is to find a local minimum of weights according to some error function, based on some initial value, learning rate, and partial derivative of weights to error.

The definition of gradient descent requires all training data to be interetpreted. Stochastic gradient descent uses a random sample (batch) before updating its weights.

Batch: Set of b examples used in each update
Epoch: $\left\lceil \frac{|Es|}{b} \right\rceil$ batches, which is one pass through all data (on average)

By definition of P(y | x)

Linear Learner:

```
Inputs:
    Xs: set of input features
    Y: Target Feature
    Es: Set of training examples
    η: learning rate
    b: batch size
Output: Funciton to make prediction on examples
Algorithm:
    Initialize weights randomly
    Define squashed linear regression prediction function
    initialize derivative values to 0
    Select batch B of size b from Es
    for each example e in B
        Predict target and find error
        for each weight, add the partial derivative of error to
          derivative values
    for each weight take gradient descent step
Return: Prediction algorithm based on weights
```

Smaller batch sizes learn faster, but may not converge to local optimum, because you're more influenced by the randomness of your selection process.

Incremental Gradient Descent: Select batches of size 1 to update. Used for streaming data where each example is used once and then discarded.
Catastrophic forgetting: Discarding older data (and not using it again) means you fit later data better but forget earlier examples.

Linear Separability

Each input feature can be viewed as a dimension, and a hyperplane can separate an m-dimensional space into two spaces.
Linearly separable: A dataset is linearly separable if there exists a hyperplane where the classification is true on one side and false on other side.

Categorical Target Features

Indicator Variables: Technique to transform a categorical feature into a set of binary variables (If you have a categorical feature or not)

One issue with indicator variables is that, if we can only use one value, then the predicted probabilities should add to 1. One way to ensure this is to only learn for all but one variables and make the remaining variable's value the difference to 1, but this doesn't work because errors for other values accumulate but the non-trained value's errors don't.

One solution is to find a linear function for each value, exponentiate, and normalize. For instance, take the softmax function:

$\text{softmax}((\alpha_1, ..., \alpha_k))_i = \left(\frac{\exp(\alpha_i)}{\sum_{j=1}^k \exp(\alpha_j)}\right)$

This guarantees all values are positive and sum to 1, so are proper analogs for probability distribution.

Multinomial Logistic Regression: Given the categorical functions with k values, we can generalize our linear function into $u_{j(e)} = w_{0,j} + X_1(e) \times w_{1,j} + ... + X_{m(e)} \times w_{m,j}$ Note we have one linear function for each output value, so our weights are w_{ij} for input i, output j.

Categorical Log Loss: Take the log loss of the softmax function.

$$\begin{aligned}\frac{\partial}{\partial w_{ij}} \quad & logloss(softmax((u_1(e), ..., u_k(e))), v_q) \\ &= \frac{\partial}{\partial w_{ij}} - \log\left(\frac{exp(u_q(e))}{\sum_j exp(u_j(e))}\right) \\ &= \frac{\partial}{\partial w_{ij}} (\log(\sum_j exp(u_j(e))) - u_q(e)) \\ &= ((\hat{Y}(e))_j - \mathbf{1}(j = q)) * X_i\end{aligned}$$

Here we only compare the predicted target value against the test target feature.

One-hot encoding: Given complete set of values where only one value is 1 and rest are 0

Overfitting

A complex model (with many degrees of freedom) has a better chance of fitting to the training data.

Bias: The error due to the algorithm finding an imperfect model. This is the deviation between the model found and the ground truth model.

Representation Bias: Representation does not contain a model close to the ground truth
Search Bias: Algorithm hasn't searched enough of the search space to find a better model.

Variance: Error from lack of data.

Bias-Variance Trade-Off: Complicated models can be accurate, but without sufficient data it can't estimate it properly (low bias, high variance). Simple models cannot be accurate, but can estimate model well given data (high bias, low variance)

Noise: Inherent error due to data depending on features not modeled or because data collection is inherently stochastic.

Overconfidence: Learner is more confident in prediction than data warrants.

Pseudocounts

Optimal prediction is usually the mean, however, this is not a good estimate for new cases, so we minimize the weight of the mean through pseudo-examples:
$\hat{v} = \frac{c \times a_0 + \sum_i v_i}{c+n}$, this value is a "weighted" mean, which minimizes the relative influence of the actual mean. a_0 is some defined set value.

Regularization

In addition to trying to fit a model to data, also include a term that rewards simplicity and penalizes complexity.

Ridge Regression: A linear regression function with an L2 regularizer
Lasso (Least Absolute Shrinkage and Selection Operator): Loss function + L1 regularizer

Feature Selection: The use of specific features for calculation. L1 regularization does this by making many weights zero.

Cross Validation

Instead of separate training and test sets, we go one step further. We have one training set, one validation set, and one test set (each completely separate).

Validation Set: This is an emulation of the test set, used to optimize the hyperparameters of the model.

Note, we want to train on as many examples as possible to get better models, so partitioning our data into three is not ideal. We use **k-fold cross validation**.

k-fold cross validation:
- Partition non-test examples randomly into k sets of approximately equal size, called folds.
- For each parameter, train k times for that parameter setting, using one of the folds as the validation set and remaining folds for training.
- Optimize parameter settings based on validation errors
- Return model with selected parameters trained on all data.

Composite Models

Although linear functions and decision trees can't be used to represent many functions, combining linear functions with non-linear inputs is a way to use simple linear models but make them more accurate.

Kernel Function: Function applied to input features to create new features. (Ex. x^2, x^3, xy)

Regression Tree: Decision Tree with constant function at each leaf (piecewise constant function)

Piecewise Linear Function: Decision Tree with linear function at leaves.

Ensemble Training: Using multiple learners, combine outputs via some function to create an ensemble prediction.

Base-level algorithms: Algorithms used as inputs for ensemble learners.

Random Forest: Using multiple decision trees make predictions through each tree and combine outputs. This works best if trees make diverse predictions: Each tree uses different subset of examples to train on (bagging) or subset of conditions for splitting are used.

Boosting

Boosting: Sequence of learners where each learns from errors of previous ones.

Boosting Algorithm:
- Base learners exist in sequence.
- Each learner is trained to fit examples previous learners didn't fit well.
- Final prediction uses a composite of predictions of each learner.
- Base learners don't have to be good, but have to be better than random.

Functional Gradient Boosting: Given hyperparameter K with K base learners, final prediction as function of inputs is $p_0 + d_1(X) + ... + d_{k(X)}$ where p_0 is initial prediction and d_i is difference from previous prediction (i.e. $p_{i(X)} = p_{i-1}(X) + d_{i(X)}$) Given p_{i-1} is fixed, the learners meant to optimize for $\hat{d}_i$ optimize for: $\sum_e \text{loss}\left(p_{i-1}(e) + \hat{d}_i(e), Y(e)\right) = \sum_e \text{loss}\left(\hat{d}_i(e), Y(e) - p_{i-1}(e)\right)$

Gradient-Boosted Trees

Gradient-Boosted Trees: Linear models where features are decision trees with binary splits, learned using boosting.

The prediction for example (x_e, y_e) is $\hat{y}_e = \sum_{k=1}^K f_{k(x_e)}$

Each f_k is a decision tree, each leaf has a corresponding output w_j (we call this the weight because of how it weighs to the larger sum). We also have a function q which corresponds to the if-then-else structure of the tree. For simplicity, we represent w as a single vector of weights, so instead of the tree hierarchy, we use a vector to store all the weights for quick compute, and q maps onto this vector (just so simplify tree traversal).

The loss function is: $\mathcal{L} = \left(\sum_e (\hat{y}_e - y_e)^2\right) + \sum_{k=1}^K \Omega(f_k)$

$\Omega(f) = \gamma \cdot |W| + \frac{1}{2}\lambda \cdot \sum_j w_j^2$, this is the regularization term to minimize the weights. $|W|$ minimizes the number of leaves. The lambda product minimizes the values of the parameters. γ and λ are non-negative learnable parameters.

Choosing Leaf Values

The model is learnt using boosting, so each tree is learned sequentially. Consider building the tth tree, where previous are fixed. Assume (for simplicity) that the tree structure (q) is fixed. Lets optimize for the weight of a single leaf w_j. Let $I_j = \{e \mid q(x_e) = j\}$ be the set of training examples that map to the jth leaf.

Since the tth tree is learnt, for regression, the loss for the tth tree is: $\mathcal{L}^{(t)} = \frac{1}{2}\lambda * \sum_j w_j^2 + \sum_e \left(y_e - \sum_{k=1}^t f_{k(x_e)}\right)^2 + \text{constant} = \frac{1}{2}\lambda * \sum_j w_j^2 + \sum_e \left(y_e - \sum_{k=1}^{t-1} f_{k(x_e)} - w_j\right)^2 + \text{constant}$ where constant is regularization values for previous trees and size of tree.

Therefore, the minimum value for w_j is $w_j = \frac{\sum_{e\in I_j}\left(y_e - \widehat{y_e}^{(t-1)}\right)}{|I_j| + \lambda}$

For classification, when you take the derivative, it's difficult to solve analytically, so instead an approximation is used in the opposite step of the gradient.

Choosing Splits

Proceed greedily. Start with single leaf, find best leaf to expand to minimize loss. For small datasets look through every split, for larger splits, take subsamples or pre-computed percentiles.

No-Free-Lunch Theorem

No matter the training set, for any two definitive predictors A and B, there are as many functions from the input feature sto target features consistent with evidence that A is better than B on off-training set (examples not in training) as when B is better than A on off-training set.

Consider m-Boolean input features, then there are 2^m assignments of input features, and 2^{2^m} functions from assignments onto {0, 1}. If we assume uniform distribution over functions, we can use 2^m bits to represent a function (one bit for each assignment, basically a lookup), and memorize the training data. Then for a training set of size n, we'll set n of these bits, but the remaining bits are free to be assigned in any way, as in there is a wide class of functions that it could be.

7 Reasoning Under Uncertainty

Probability

Prior Probability: Belief of agent before it observes anything
Posterior Probability: Updated belief (after discovering information - observation)

Bayesian Probability: Probability as a measure of belief (Different agents can have different information, so different beliefs)

Epistemology: A value based on one's belief
Ontology: A veritable fact of the world

The belief in a proposition α is measured in between 0 and 1, where 0 means it's **believed** to be definitely false (no evidence will shift that) and 1 means it's believed to be definitely true. Note the bounds are arbitrary. A value in between 0 and 1 doesn't reflect the variability of the truth, but rather strength of agent's belief.

Worlds: Semantics are defined in terms of "possible worlds", each of which is one way the world could be. There is only one true world, but real agents aren't omniscient, and can't tell.

Random Variable: Function on worlds—given world, it returns a value. Set of values it returns is the domain of the variable.

Primitive Proposition: Assignment of a value to a variable, or inequality between variable and value, or between variables.

Proposition: A connection between primitive propositions using logical connectives, that is either true or false in a world.

Probability Measure: Function μ from sets of world, into nonnegative real numbers that satisfies two constraints: If Ω_1 and Ω_2 are disjoint sets of worlds, then $\mu(\Omega_1 \cup \Omega_2) = \mu(\Omega_1) + \mu(\Omega_2)$ and $\mu(\Omega) = 1$ where Ω is the set of all possible worlds.

Probability of Proposition: Probability of proposition α, written $P(\alpha)$ is measure of set of possible worlds where α is true, so $P(\alpha) = \mu(\{\omega : \alpha \text{ is true in } \ \omega\})$

Probability Distribution: If X is random variable, $P(X)$ is function from domain of X onto real numbers such that given $x \in$ domain(X), $P(X)$ is probability of proposition $X = x$

Joint Probability Distribution: If $X_1, ..., X_n$ are all of the random variables, then $P(X_1, ..., X_n)$ is the distribution over all worlds, and an assignment to the random variables corresponds to a world.

Infinite Worlds

There can be infinitely many worlds if domain of variable is infinite, infinitely many variables (ex. variable for location of robot for every second from now into the future)

The probability of $X = v$ can be zero for a variable with continuous domain, but between a range of values, $v_0 < X < v_i$ it can have real-values (think of probability function), therefore a **probability density function** is used: $P(a \leq X \leq) = \int_a^b p(X)dX$

Parametric Distribution: Density function is defined by formula with free parameters.

Nonparametric Distribution: Probability function where number of parameters is not fixed, such as in decision tree.

Discretization: Convert continuous variables into discrete values (like heights that are converted into separate regions and then capped)

Conditional probability

Evidence: Proposition e representing conjunction of all of agent's observations

Posterior Probability: $P(h \mid e)$, given evidence e, belief of h
Prior Probability: $P(h)$, without any evidence, what is initial assumption of h

Formal Definition of prior probability: $\mu_{e(S)} = \begin{cases} c \times \mu(S) \text{ if e is true in } \ \omega \text{ for all } \ \omega \in S \\ 0 \text{ if e is false in } \ \omega \text{ for all } \ \omega \in S \end{cases}$

Then, for μ_e to be probability distribution:

$$\begin{aligned} 1 &= \mu_{e(\Omega)} \\ &= \mu_{e(\{w : e \text{ true in } \ w\})} + \mu_{e(\{w : e \text{ false in } \ w\})} \\ &= c \times \mu(\{w : e \text{ true in } w\}) + 0 \\ &= c \times P(e) \end{aligned}$$

So $c = \frac{1}{P(e)}$

Formal definition of posterior probability:

$$\begin{aligned} P(h \mid e) &= \mu_{e(\{\omega : h \text{ true in } \ \omega\})} \\ &= \mu_e(\{\omega : h \wedge e \text{ true in } \omega\}) + \mu_{e(\{\omega : h \wedge \neg e \text{ true in } \omega\})} \\ &= \frac{1}{P(e)} \mu(\{\omega : h \wedge e \text{ true in } \ \omega\}) + 0 \\ &= \frac{P(h \wedge e)}{P(e)} \end{aligned}$$

Chain Rule:

$$\begin{aligned} P(a_1 \wedge ... \wedge a_n) &= P(a_n \mid a_1 \wedge ... \wedge a_{n-1}) \times P(a_1 \wedge ... \wedge a_{n-1}) \\ &= P(a_n \mid a_1 \wedge ... \wedge a_{n-1}) \times ... \times P(a_2 \mid a_1) \times P(a_1) \\ &= \prod_{i=1}^{n} P(a_i \mid a_{i-1} \wedge ... \wedge a_1) \end{aligned}$$

Bayes Rule

Given current belief in proposition h based on evidence k, given new evidence e we update the belief as follows:

$P(h \mid e \wedge k) = (P(e \mid h \wedge k)) \times P(h \mid k) \frac{)}{P(e \mid k)}$
(assuming $P(e \mid k)) \neq 0$

Simplifying by keeping k implicit, we get:

$P(h \mid e) = \frac{P(e \mid h) \times P(h)}{P(e)}$

Expected Value ($\xi_{P(X)} = \sum_{v \in \text{ domain}(X)} v \times P(X = v)$) if finite/countable, integral if continuous.

Independence

Conditional independence: If $P(X \mid Y, Z) = P(X \mid Z)$, then X is conditionally independent of Y

Unconditional Independence: If $P(X, Y) = P(X)P(Y)$, so they are conditionally independent given no observations. Note this doesn't imply they are conditionally independent.

Context-specific independence: Variables X and Y are independent with respect to context $Z = v$ if $P(X \mid Y, Z = v) = P(X \mid Z = z)$ that is it is conditionally independent for one specific value of Z

Belief Networks

Markov Blanket: Set of locally affecting variables that directly affect X's value.

Belief Network: Directed Acyclic Graph representing conditional dependence among a set of random variables. Nodes are random variables. Edges are direct dependence. Conditional independence is determined by an ordering of the variables; each variable is independent of its predecessors in total ordering given a subset of the predecessors called the parents. Independence is indicated by missing edges.

8 Probabilistic Learning

Bayes Rule: $P(m \mid E) = \frac{P(E \mid m) \times P(m)}{P(E)}$

Prior Probability: Agent's beliefs before any observations. $P(h)$
Posterior Probability: Updated beliefs after observing. $P(h \mid e)$

Bayesian Probability: View of probability as measure of belief (Epistimological-Pertain to knowledge)
Frequentist Probability: Probability is dependent on observations (Ontological - How world is)

Joint Probability: P(X, Y) is probability X and Y are both true

Proposition: Assignment of value to a variable

Axioms of Probability:
- $P(x) > 0$
- $\sum P(x) = 1$
- $P(A \vee B) = 1$ if $P(A)$ and $P(B)$ are impossible at same time.

Note: 1 is simply convention—could be anything else.

$P(h \mid e) = P(h \mid e = \text{True}) = \frac{P(h \wedge e)}{P(e)}$

Chain Rule: $P(a_1 \wedge a_2 \wedge ... \wedge a_n) = P(a_n \mid a_{n-1} \wedge ... \wedge a_1) \times ... \times P(a_2 \mid a_1) \times P(a_1)$

Marginal Distribution: Given Joint distribution, marginalize out some of the variables to isolate for a subset.

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

Independence: $P(X, Y) = P(X)P(Y), P(X) = P(X|Y)$
Conditional Independence: $P(X, Y \mid Z) = P(X|Z)P(Y|Z), P(X|Z) = P(X|Y, Z)$

Expected Value: $\mathbb{E}(X) = \sum_{v \in \text{ domain}(X)} V(X)P(X)$
Bayesian Decision Making: $\max_{\text{decision}} \mathbb{E}(V(\text{decision})) = \max_{\text{decision}} \sum_{\text{outcome}} P(\text{outcome} \mid \text{decision})V(\text{decision})$

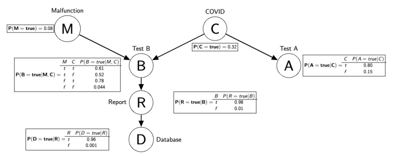
Complete independence means we can represent a probability distribution by the distributions over each individual variable instead of joint connections, reducing space and calculation complexity from $O(2^n)$ to $O(n)$

Belief Networks

Directed acyclic graph with dependencies for each event. *Bayesian networks don't imply causality, just correlation. Having causality makes reasoning with network easier*

We can say A is independent of B given C if learning C makes B irrelevant information. However, without C, it's possible learning B is valuable.

Conditional Probability Table: Stores probability distribution on X given parents. $P(X_i \mid \text{parents}(X_i))$ for each X_i



Updating Beliefs

Given prior belief $P(h)$, and evidence e with likelihood given hypothesis $P(e \mid h)$, posterior belief after observing evidence is $P(h \mid e)$. You can calculate this posterior with Bayes' Rule.

$$P(h \mid e) = \frac{P(e \mid h)P(h)}{P(e)} = \frac{P(e \mid h)P(h)}{\sum_h P(e|h)P(h)}$$

Bayes' Theorem allows you to determine evidential reasoning from causal knowledge.

Forward Inference Rules:

Marginalization: $P(B) = \sum_{m,c} P(M = m, C = c, B)$
Chain Rule: $P(B) = \sum_{m,c} P(B \mid M, c)P(m \mid c)P(c)$
Independence: $P(B) = \sum_{m,c} P(B \mid m, c)P(m)P(c)$
Distribution of product over sum: $P(B) = \sum_m P(m) \sum_c P(c)P(B \mid m, c)$

Backward Inference Rules: When evidence is downstream of query, we reason "backwards", using Bayes' rule.

Variable Elimination

Generalization that applies sum-out rule repeatedly to determine conditional probabilities.

Factors

Factor: Representation of a function from tuple of random variables into a number.
Restricting a factor: Assigning some or all of the variables of a factor.

Product of Factor: $f_1(X, Y)$ and $f_2(X, Y)$ where Y are the common variables is factor $(f_1 \times f_2)(X, Y, Z) = f_1(X, Y)f_2(Y, Z)$

Sum out: X_1 with domain $\{v_1, ..., v_k\}$ from factor $f(X_1, ..., X_j)$ resulting in factor on $X_2, ..., X_j$ as $\sum_{X_1} f(X_2, ..., x_j) = f(X_1 = v_1, ..., X_j) + ... + f(X_1 = v_k, ..., X_j)$

Evidence

To find posterior probability of Z given evidence $Y_1 = v_1 \wedge ... \wedge Y_j = v_j$ $P(Z|Y_1 = v_1, ...Y_j = v_j) = \frac{P(Z, Y_1 = v_1, ..., Y_j = v_j)}{P(Y_1 = v_1, ..., Y_j = v_j)} = \frac{P(Z, Y_1 = v_1, ..., Y_j = v_j)}{\sum_Z P(Z, Y_1 = v_1, ..., Y_j = v_j)}$

Thus the posterior is the joint probability of query and evidence, normalized at the end.

Conjunction Probability

Suppose query Z and evidence Y, then we can achieve remaining variables $M_1, ..., M_k = \{X_1, ..., X_n\} - \{Z\} - \{Y_1, ..., Y_j\}$ and sort into elimination ordering. Then $P(Z, Y = v_1, ..., Y_j = v_j) = \sum_{Z_k} ... \sum_{Z_1} P(X_1, ..., X_n)_{Y_1 = v_1, ..., Y_j = v_j} = \sum_{Z_k} ... \sum_{Z_1} \prod_{i=1}^n P(X_i \mid \text{parents}(X_i))_{Y_1 = v_1, ..., Y_j = v_j}$

Variable Elimination Algorithm

1. Construct factor for each conditional probability
2. Restrict observed vars to observed vals
3. Sum out non-relevant variables in some elimination ordering: For each Z_i starting from $i = 1$
   1. Collect factors containing Z_i
   2. Multiply together and sum out Z_i
   3. Add resulting new factor back to pool
4. Multiply remaining factors
5. Normalize by dividing resulting factor $f(Z)$ by $\sum_Z f(Z)$

Summing out Variable

To sum out variable Z_j from product $f_1, ..., f_k$ of factors, partition factors into those containing Z_j and those that don't, resulting in: $\sum_{Z_j} f_1 \times ... f_k = f_1 \times ... f_i \times \left(\sum_{Z_j} f_{i+1} \times ... \times f_k\right)$

Then create an explicit representation of rightmost factor $\sum_{Z_j} f_{i+1} \times ... \times f_k$ and replace factors $f_{i+1}, ..., f_k$ by new factor.

Variable Elimination Considerations

- Complexity is linear in number of variables, and exponential in size of largest factor.
- When creating new factors, sometimes this blows up, depending on elimination ordering
- For polytrees: work outside in, for general Bayesian Networks, this can be hard.
- Finding optimal elimination ordering is NP-hard for general BNs
- Inference in general is NP-hard
- Better to eliminate singly-connected nodes because no factor is ever larger than original CPT
- Some variables have no impact on certain other variables, so need to sum.
- To remove irrelevant variables, ensure query Q is relevant; if any node is relevant, its parents are relevant. If $E \in$ Evidence, is descendent of relevant variable, then E is relevant.

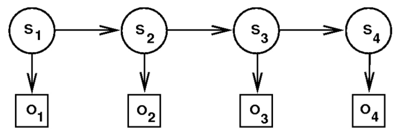
Probability and Time

- Node can repeat over time, into discrete explicit encodings of time.
- Either event-driven time or clock-driven time
- Represented by Markov Chain
- Chain length represents amount of time you want to model

Markov Assumption: $P(S_{t+1}|S_1, ..., S_t) = P(S_{t+1}|S_t)$

Hidden Markov Models

Given observations $O_1, ..., O_t$ we can estimate $P(S_t \mid O_1, ..., O_t)$, or $P(S_k \mid O_1, ..., O_t), k < t$



Dynamic Bayesian Networks

Any Bayesian network can repeat over time as a Dynamic Bayesian Network.

Through sensor fusion, Bayesian probabilities ensures that evidence is integrated proportionally to its precision, with precision weight sensors.

Stochastic Simulation

Monte Carlo Simulation: Stochastic sampling from distribution to estimate probabilities.

To generate samples from a distribution, totally order values of domain of X, generate PCF, select value in range [0, 1], select x such that $f(x) = y$

Forward Sampling in Belief Network:

1. Sample Variables one at a time
2. Sample parents of X before sampling X
3. Given values for parents of X, sample from probability of X given parents
4. For samples s_i, 1…N: $P(X = x_i) \propto \sum_{s_i} \delta(x_i) = N_{X=x_i}$

$P(H = h_i \mid E = e_i) = \frac{P(H=h_i) \land E=e_i}{P(E=e_i)} = \frac{N(h_i,e_i)}{N(e_i)}$

9 Machine Learning

Bayesian Learning

Premise is we have multiple hypotheses without knowing which is correct; we assume all are correct to a degree, and have a distribution over models with the goal of computing expected prediction given average.

Suppose X is input features, and Y is target feature, $d = \{x_1, y_1, ..., x_N, y_N\}$ as evidence and we want to know, summing over all models, $m \in M$: $P(Y \mid x, d) = \sum_{m \in M} P(Y, m \mid x, d) = \sum_{m \in M} P(Y \mid m, x) P(m \mid d)$

Bayesian Learning: Given prior $P(H)$, likelihood $P(d \mid H)$ and evidence d, then $P(H \mid d) \propto P(d \mid H) P(H)$

To predict X, we use $P(X \mid d) = \sum_i P(X \mid d, h_i) P(h_i \mid D) = \sum_i P(X|h_i) P(h_i \mid d)$ where predictions are weighted averages of predictions of individual hypotheses.

Bayesian Learning Properties:
- Optimal: Given prior, no other prediction is correct more often than Bayesian one
- No overfitting: Prior/likelihood both penalize complex hypohteses

Downsides:
- Bayesian learning can be intractable when hypothesis space is large
- Summing over hypotheses space may be intractable.

Maximum A Posteriori

Making prediction based on most probable hypothesis: $h_{MAP} = \arg\max_{h_i} P(h_i \mid d)$, $P(X \mid d) \approx P(X \mid h_{MAP})$, this is in comparison to Bayesian Learning, where all hypotheses are used.

Maximum a posteriori properties:
- Less accurate than Full Bayesian because of only one hypothesis
- MAP and Bayesian converge as data increases
- No overfitting
- Finding h_{MAP} may be intractable, $h_{MAP} = \arg\max_h P(h \mid D) = \arg\max_h P(h) P(d \mid h) = \arg\max_h P(h) \prod_i P(d_i \mid h)$ which induces non-linear optimization, but we can take log to linearize.

Maximum Likelihood

Simplify MAP by assuming uniform prior, to get $h_{ML} = \arg\max_h P(d \mid h)$, then $P(X \mid d) \approx P(X \mid h_{ML})$

Maximum Likelihood Properties:
- Less accurate than Bayesian or Maximum A Posteriori since it ignores prior and relies on one hypothesis
- Converges with Bayesian as data amount increases
- More susceptible to overfitting due to no prior
- h_{ML} is easier to find than h_{MAP}, $h_{ML} = \arg\max_h \sum_i \log P(d_I \mid h)$

Binomial Distribution

$\binom{n+k}{k} \theta^n (1 - \theta)^k$
Beta distribution: $B(\theta; a, b) \propto \theta^{a-1} (1 - \theta)^{b-1}$

Bayesian Classifier

If you knew classifier you could predict values of features $P(\text{Class} \mid X_1, ..., X_n) \propto P(X_1, ..., X_n \mid \text{Class}) P(\text{Class})$

Naive Bayesian Classifier: X_i are independent of each other given class, requires $P(\text{Class})$ and $P(X_i \mid \text{Class})$ for each X_i, so $P(\text{Class}|X_1, ..., X_n) \propto \prod_i P(X_i \mid \text{Class}) P(\text{Class})$

Laplace Correction

If feature never occurs in training set, but it does in test set, Maximum Likelihood may assign zero probability to high likelihood class. The solution is to add 1 to numerator, and add d (arity of variable) to denominator, like a pseudocount.

Bayesian Network Parameter Learning: For fully observed data, set $\theta_{V,pa(V)=v}$ to relative frequency of values of V given values v of parents of V, where these theta values are the parameters, and essentially calculates probability of value v set to v_i given the set values of the parents of v

Occam's Razor: Simplicity is encouraged in likelihood function; a more complex, high bias function is less preferred, since low-bias can explain more datasets, but with lower probability (higher variance)

Bias-Variance Tradeoff: Simple models have high bias, but low variance. Complex models have low bias, but high variance.

Neural Networks

Linear Regression

Model where output is linear function of input features, $\hat{Y}_{\vec{w}}(e) = \sum_{i=0}^{n} w_i X_i(e)$, with w_0 as bias term.

SSE is $\sum_{e_\in E} \left(Y(e) - \hat{Y}_{\vec{w}}(e)\right)^2$, to minimize.

Deriving Weights:
- Analytically:
  - Take $\vec{y}$ as vector of output features for M examples.
  - X is matrix where j^{th} column is values of input feature for j^{th} example
  - $\vec{w}$ is vector of weights
  - $\vec{y} = \vec{w} X$, so $\vec{y} X^\top (XX^\top)^{-1} = \vec{w} \left(XX^\top\right)^{-1}$ is pseudo-inverse, which we can compute because $XX^\top$ is invertible, since it's square.
- Iteratively (not just linear):
  - Using gradient descent for $w_i \leftarrow w_i - \mu \frac{\partial \text{ Error}}{\partial w_i}$
  - For SSE, update rule is $w_i \leftarrow w_i + \mu \sum_{e \in E} \left(Y(e) - \sum_{i=0}^{n} w_i X_i(e)\right) X_i(e)$

Stochastic Gradient Descent: Examples are chosen randomly
Batched Gradient Descent: Process batch of size n before updating weights. (If n is all data, that's gradient descent, if $n = 1$ that's incremental gradient descent)
Incremental Gradient Descent: Weight udpates are done immediately after example, but it might "undo" work of other weight updates, creating an oscillation

Linear Classifier

Squashed Linear function: $\hat{Y}_{\vec{w}}(e) = f\left(\sum_{i=0}^{n} w_i X_i(e)\right)$, f is activation function. Generally, if activation function is differentiable, you can use gradient descent to update weights.

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

Neural Networks

- Can learn same things decision tree can
- Imposes different learning biases
- Back-propagation learning: errors made are propagated

Neural nets have nodes with weights and a bias, receiving inputs that are either example features or outputs of another layer, and output goes through a non-linear activation function (because otherwise you could represent it all linearly)

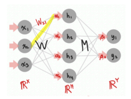
Common Activation Functions:
- Step Function: Not differentiable (1 for x > 0; 0 else)
- Sigmoid:
  - For extreme x, very close to boundaries
  - $f(x) = \frac{1}{1+e^{-kx}}$
  - Can tune sigmoid to step function by changing k
  - Vanishing gradients where $f(x)$ changes little at extreme x
  - Expensive compared to ReLU
- ReLU:
  - $f(x) = \max(0x)$
  - Differentiable
  - Dying ReLU problem, where inputs approaching 0 or negative have 0 gradients, so network can't learn.
- Leaky ReLU: $f(x) = \max(0, x) + k \cdot \min(0, x)$
  - Small positive slope k in negative area, enabling learning for negative input values.

Feedforward Network: Directed acyclic graph with single direction connections; (Function of inputs)
Recurrent Network: Feeds outputs into inputs; Can have short-term memory.

Backpropagation

$\nabla_{z^{(l+1)}} E = \frac{\partial E}{\partial z^{(l+1)}}$ $h^{(l+1)} = \sigma\left(z^{(l+1)}\right) = \sigma\left(W^{(l)} h^{(l)} + b^{(l+1)}\right)$
Basically, h is hidden layer, z is input current, W is weight matrix, b is bias. $\nabla_{z^{(l)}} E = \frac{dh^{(l)}}{dz^{(l)}} \odot \left[\nabla_{z^{(l+1)}E \cdot \left(W^{((l))T}\right)}\right]$ $\odot$ is hadamard product, which does element by element multiplication. We transpose because W_{ij} is connection from ith node in l to jth node in $l + 1$

Note that $\vec{a} = \vec{x} W$ in this diagram:



$\frac{\partial E}{(\partial W_{ij}^{(l)} = \frac{\partial E}{\partial z_j^{(l+1)}} \cdot \frac{\partial z_j^{(l+1)}}{\partial W_{ij}^{(l)}} = \frac{\partial E}{\partial z_j^{(l+1)}} \cdot h_i^{(l)}}$

Finally, $\nabla_{z^{(l)}} E = \sigma'\left(z^{(l)}\right) \odot \left(\nabla_{z^{(l+1)}} E \cdot \left(W^{(l)}\right)^T\right)$ $\nabla_{W^{(l)}} E = \left[h^{(l)}\right]^T \nabla_{z^{(l+1)}} E$

Vectorization

We can generalize this process to take a batch of samples by letting x be a matrix of samples instead of just one sample. Then, note $\nabla_{z^{(l)}} E$ is a matrix with same dimension as $z^{(l)}$ as desired. Further, note that $\nabla_{W^{(l)}} E$ is a gradient vector that sums the weight gradient matrix from each sample.

Improving Optimization

Momentum: Weight changes accumulate over iterations
RMS-Prop: Rolling average of square of gradient
Adam: Combination of Momentum and RMS-Prop
Initialization: Randomly set parameters to start

Improving Generalization: Regularization

- Parameter Norm Penalties added to loss function
- Dataset augmentation
- Early stopping
- Dropout
- Parameter tying
  - Convolutional Neural Nets: Used for images, parameters tied across space
  - Recurrent neural nets: Used for sequences, parameters tied across time

Sequence Modelling

- Word Embeddings: Latent vector spaces representing meaning of words in context
- RNNs: Neural net repeats over time and has inputs from previous time step
- LSTM: RNN with longer-term memory
- Attention: Uses expected embeddings to focus updates on relevant parts of network
- Transformers: Multiple attention mechanisms
- LLMs: Very large transformers for language

Composite Models

- Random Forests
  - Each decision tree in forest is different
  - Different features, splitting criteria, training sets
  - Average of majority vote determines output
- Support Vector Machines
  - Find classificaiton boundary with widest margin
  - Combined with kernel trick
- Ensemble learning: Combination of base-level algorithms
- Boosting:
  - Sequence of learners fitting examples the previous learner did not fit well
  - Learners progressively biased towards higher precision
  - Early learners: Lots of false positives, but reject all clear negatives
  - Later learners: Problem is more difficult, but set of examples is more focused around challenging boundary

Unsupervised Machine Learning

When data is incomplete (missing some variables), or values of some attributes are missing, etc.

Maxmium Likelihood Learning

$\theta_{V=\text{true, parents}(V)=v} = P(V = \text{true} \mid \text{parents}(V) = v)$

ML learning of θ is $\theta_{V=\text{true, parents}(V)=v} = \frac{\text{number with } (V=\text{ true} \land \text{parents}(V)=v)}{\text{number with parents}(V)=v}$

Direct maximum likelihood: $h_{ML} = \arg\max\left[\sum_Z P(e, Z \mid h)\right] = ... = \arg\max_{h\left[\log_{\sum_Z} \pi_{i=1}^{n} P(X_i \mid \text{parents}(X_i), h)_{E=e}\right]}$

Missing Data

You can't ingore missing data unless you know it is missing at random. (If it is, then you can ignore hidden variables, or data with missing variables, but not with true latent variables that are always missing)

Sometimes data is missing because of a correlated variable of interest, which is when you'd need to model why the data is missing.

Expectation-Maximization

- Repeat
  - Expectation: based on h_{ML} calculate $P(Z \mid h_{ML}, e)$
  - Maximization: based on expected missing values, compute new estimate of h_{ML}

Simple-version: K-means algorithm: Compute most likely k-means, then maximize based on those classifications.

K-means Algorithm: Pick k means in X, one per class, C. THen assign examples to k classes, and re-estimate k means based on assignment. Repeat until you stop changing.

Expectation Maximization Motivation:
- Approximate maximum likelihood
- Start with guess h_0
- Iteratively compute $h_{i+1} = \arg\max_h \sum_Z P(Z \mid h_i, e) \log P(e, Z \mid h)$
- Expectation: Compute $P(Z \mid h_i, e)$ that fills in missing data (the unknown variables)
- Maximization: Find new h that maximizes $\sum_Z P(Z \mid h_i, e) \log P(e, Z \mid h)$
- Can show that $P(e \mid h_{i+1}) \geq P(e \mid h_i)$

General Bayes Network EM

- Complete Data: Bayes Net Maximum Likelihood

$\theta_{V=\text{ true, parents}(V)=v} = \frac{\text{number } \in e \text{ with } (V=\text{ true} \land \text{parents}(V)=v)}{\text{number } \in e \text{ with parents}(V)=v}$

- Incomplete Data: Bayes Net Expectation Maximization

Observed variables X and missing variables Z. Start with some guess for θ, and then for E step compute weights for each data x_i and latent variable(s) value(s) z_j (using e.g. variable elimination) $w_{ij} = P(z_j \mid \theta, x_i)$. M Step: Update parameters with $\theta_{V=j, \text{ parents}(V)=v} = \frac{\sum_i w_{ij} \mid V=j \land \text{ parents}(V)=v \in \{x_i, z_j\}}{\sum_{ij} w_{ij} \mid \text{parents}(V)=v \in \{x_i, z_j\}}$

Belief Network Structure Learning

$P(\text{model} \mid \text{data}) = \frac{P(\text{data} \mid \text{model}) \times P(\text{model})}{P(\text{data})}$
- Here a model is a belief network.
- Bigger networks can always fit data better.
- $P(\text{model})$ lets us encode a preference for smaller networks (e.g. using description length)
- You can search over network structure looking for most likely model.
- You can do independence tests to determine which features should be the parents.
- Just because features don't give information indivdually doesn't mean they won't give information in combination with out features.
- It is ideal to search over total orderings of variables.

Autoencoders

- Representation Learning algorithm that learns to map examples to lower dimensional representation
- 2 Main components: Encoder that maps x to low-dimensional $\hat{z}$, Decoder that maps $\hat{z}$ to original representation x
- Goal is to minimize SSE of $E = \sum_i (x_i - d(e(x_i)))^2$

Deep Neural Network Autoencoders: Good for complex inputs. e and d are feedforward neural networks, joined in series and trained via backpropagation.

Generative Adversarial Networks

- Generative unsupervised learning algorithm
- Goal is to generate unseen examples that look like training examples
- Contain a pair of networks
  - Generator $g(z)$, where given vector z in latent space, produces example x drawn from distribution taht approximates true distribution of training examples. z usually sampled from Gaussian distribution
  - Discriminator $d(x)$: A classifier that predicts whether x is real or fake
- Trained with minimax error: $E = \mathbb{E}_{x[\log(d(x))]} + \mathbb{E}_{z[\log(1-d(g(z)))]}$
- Discriminator tries to maximize E
  - For x from training set $d(x) \to 1$
  - For x from generator (based on z), $d(x) \to 0$
- Generator tries to minimize E (to fool d)
  - For x from training set, $d(x) \to 0$
  - For x from generator (based on z), $d(x) \to 1$
- After convergence, g should produce realistic outputs, and d should output $\frac{1}{2}$ indicating maximal uncertainty.

10 Planning with Uncertainty

Bayesian Decision Making

$\mathbb{E}(V(\text{decision})) = \sum_{\text{outcome}} P(\text{outcome} \mid \text{decision}) V(\text{outcome})$

You can expand this to include context, so $V(\text{decision, context})$ is value of decision in situation context.

$\mathbb{E}(V(\text{decision, context})) = \sum_{\text{outcome}} P(\text{outcome} \mid \text{decision, context}) V(\text{outcome})$

Preferences

- Actions result in outcomes
- Agents have preferences over outcomes
- (Decision-theoretic) rational agents will do action that has best outcome for them
- Sometimes agents don't know outcomes of the actions, but they still need to compare actions
- Agents have to act (doing nothing is often a meaningful action)

Preferences over outcomes

- If o_1 and o_2 are outcomes
- $o_1 \succeq o_2$ iff o_1 is at least as desirable as o_2 (weak preference)
- $o_1 \sim o_2$ means $o_1 \succeq o_2$ and $o_2 \succeq o_1$ (indifference)
- $o_1 \succ o_2$ means $o_1 \succeq o_2$ and $o_2 \not\succeq o_1$ (strong preference)

Lottery Probability Distribution over outcomes written $[p_1 : o_1, p_2 : o_2, ..., p_k : o_k]$

Preference Properties

- Completeness: Agents have to act, so they must have preferences. $\forall o_1 \forall o_2, o_1 \succeq o_2$ or $o_2 \succeq o_1$
- Transitivity: If $o_1 \succeq o_2$ and $o_2 \succeq o_3$, then $o_1 \succeq o_3$
- Monotonicity: Agent prefers larger chance of getting better outcome than smaller chance. If $o_1 \succ o_2$ and $p > q$, then $[p : o_1, 1-p : o_2] \succ [q : o_1, 1-q : o_2]$
- Continuity: Suppose $o_1 \succ o_2$ and $o_2 \succ o_3$, then there exists $p \in [0,1]$ such that $o_2 \sim [1-p : o_1, p : o_3]$
- Decomposability: Agent is indifferent between lotteries that have same probabilities and outcomes. (Expected value is the only thing that matters, risk is irrelevant)
- Substitutability: If $o_1 \sim o_2$ then agent is indifferent between lotteries that only differ by o_1 and o_2

Utility Function Theorem If preferences follow preceding properties, then preferences can be measured by a function utility : outcomes $\to [0,1]$ such that $o_1 \succeq o_2$ if and only if $\text{utility}(o_1) \geq \text{utility}(o_2)$ and utilities are linear with probabilities so $\text{utility}([p_1 : o_1, ..., p_k : o_k]) = \sum_{i=1}^{k} p_i \times \text{utility}(o_i)$

Rational agents Act to maximize expected utility

Making decisions under uncertainty

- Agents should choose based on their abilities (what options are available), beliefs (way the world could be given agent's knowledge), and preferences (what agent wants and tradeoffs when there are risks)

Single Decisions

Decision Variables Similar to random variables that agent gets to choose value of. In single variable, agent can choose $D = d_i$ for any $d_i \in \text{dom}(D)$

- Expected utility of decision $D = d_i$ leading to outcomes ω for utility function u is $\mathbb{E}(u \mid D = d_i) = \sum P(\omega \mid D = d_i) u(\omega)$

Optimal Single Decision $D = d_{\max}$ whose expected utility is maximal, $\mathbb{E}(u \mid D = d_{\max}) = \max_{d_i \in \text{dom}(D)} \mathbb{E}(u | D = d_i)$

Decision Networks

Decision network Graphical representation of finite sequential decision problem. They extend belief networks to include decision variables and utility. Decision network specifies what information is available when agent has to act and which variables utility depends on.

- Random variables are drawn as ellipses. Edges into node represent probabilistc dependence.
- Decision variables are drawn as rectangles. Edges represent information available when decision is made.
- Utility node is drawn as diamond. Edges into it represent variables that utility depends on.

Finding Optimal Decision

Suppose random variables are $X_1, ..., X_n$, decision variables are D, and utility depends on $X_{i_1}, ..., X_{i_k}$ and D. Then, expected utility is: $\mathbb{E}(u \mid D) = \sum_{X_1,...,X_n} P(X_1, ..., X_n | D) \times u\left(X_{i_1}, ..., X_{i_k}\right), D) = \sum_{X_1,...,X_n} \left[\prod_{j=1}^{n} P\left(X_j \mid \text{parents}\left(X_j\right)\right)\right] \times u\left(X_{i_1}, ..., X_{i_k}, D\right)$

To find optimal decision:

1. Create factor for each conditional probability and for utility.
2. Multiply together and sum out all of the random variables
3. This results in a factor on D that gives expected utility for each D
4. Choose D with maximum value in factor.

Sequential Decisions

An agent typically doesn't do multiple stpes all at once without trying to gather more information; more common situation is (observe, act, observe, act, ...). Subsequent actions depend on observations, which depend on previous actions.

Sequential Decision Problem Sequence of decision variables $D_1, ..., D_n$

- Each D_i has information set of variables parents(D_i) whose value will be known at the time decision D_i is made.

Policies

Policy Sequence $\delta_1, ..., \delta_n$ of decision functions: $\delta_i : \text{dom}(\text{parents}(D_i)) \to \text{dom}(D_i)$ which means that when agent has observed $O \in \text{dom}(\text{parents}(D_i))$ it will do $\delta_{i(O)}$

A possible world ω satisfies policy δ (written $\omega \models \delta$) if decisions of policy are those the world assigns to decision variables. That is, each world assigns values to the decision nodes that are the same as in the policy.

Expected Value of policy δ $\mathbb{E}(u \mid \delta) = \sum_{\omega \models \delta} u(\omega) \times P(\omega)$
Optimal Policy Policy that maximizes expected utility

Finding Optimal Policy

1. Create factor for each conditional probability table and a factor for the utility
2. Set remaining decision nodes (which are all decision nodes)
3. Multiply factors and sum out random variables that aren't parents of a remaining decision node.
4. Select and remove a decision variable D from the list of remaining decision nodes. Pick one that is in a factor with only itself and some of its parents (no children)
5. Eliminate D by maximizing to return optimal decision function for D, $\arg\max_D f$ and a new factor to use, $\max_D f$
6. Repeat 3-5 until there are no more remaining decision nodes.
7. Eliminate remaining random variables. Multiply the factors: this is the expected utility of the optimal policy
8. If any nodes were in evidence, divide by $P(\text{evidence})$

Agents

Agents carry out actions, forever (infinite horizon), until stopping criteria (indefinite horizon), finite and fixed number of steps (finite horizon)

Decision Theoretic Planning What should agent do for various planning horizons when:

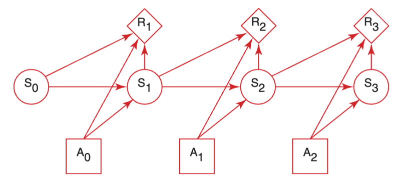
- It gets rewards and punishments and tries to maximize its rewards
- Actions are noisy, and outcome can't be fully predicted
- Model specifies probabilistic outcome of actions
- World if fully observable

World State Information such that if you knew world state, no information about past is relevant to the future.

- If S_i, A_i are state and action at time i, then $P(S_{t+1} \mid S_0, A_0, ..., S_t, A_t) = P(S_{t+1} \mid S_t, A_t)$
- Dynamics are **stationary** if distribution is same for each time point. $P(S_{i+1} \mid S_i, A_i) = P\left(S_{j+1} \mid S_j, A_j\right) \forall i, j$

Absorbing States State that once entered, can't be left (ex. lose virginity)

Markov Decision Process Augmented Markov chain with actions and values



For MDP specify:

- Set S of states
- Set A of actions
- $P(S_{t+1} \mid S_t, A_t)$ to specify dynamics
- $R(S_t, A_t, S_{t+1})$ to specify reward. Agent gets reward at each time step (rather than final reward).

Information Availability

Fully-Observable MDP Agent gets to observe S_t when deciding on action A_t
Partially-observable MDP (POMDP) Agent has some noisy sensor of the state. It is necessary to remember sensing and acting history (since it doesn't have full account of state, which it can accomplish by maintaining a sufficiently complex belief state)

Rewards and Values

If agent receives sequence of rewards $r_1, r_2, ...$:
Total Reward $V = \sum_{i=1}^{\infty} r_i$
Average Reward $V = \lim_{n\to\infty} \frac{r_1 + ... + r_n}{n}$
Discounted Reward $V = \sum_{i=1}^{\infty} \gamma^{i-1} r_i$ where γ is discount factor $0 \leq \gamma \leq 1$

Policies

Stationary Policy Function $\pi : S \to A$ where given state s, $\pi(s)$ speciesi what action the agent following π will do.
Optimal Policy Policy with maximum expected discounted reward

For fully-observable MDP with stationary dynamics and rewards with infinite or indefinite horizon, there is always an optimal stationary policy.

$Q^{\pi(s,a)}$ where a is action and s is state is expected value of doing a in state s, then following policy π for afterwards decisions.

$V^{\pi(s)}$ where s is state, is the expected value of following policy π in state s

Q^{π} and V^{π} can be defined mutually recursively.

$Q^{\pi} = \sum_{s'} P(s'|a,s)\left(r(s,a,s') + \gamma V^{\pi(s')}\right)$ $V^{\pi(s)} = Q^{\pi(s,\pi(s))}$

$Q^*(s,a)$ is $Q^{\pi(s,a)}$ for the optimal policy. $\pi^*(s)$ is optimal action to take in state s $V^*(s)$ is expected value of following optimal policy in state s.

$Q^{\pi} = \sum_{s'} P(s'|a,s)(r(s,a,s') + \gamma V^*(s'))$ $V^{\pi(s)} = \max_a Q^{a(s,a)}$
$\pi^*(s) = \arg\max_a Q^*(s,a)$

Value Iteration

- t-step lookahead value function V^t is expected value with t steps to go
- Goal: Given an estimate of t-step lookahead value function, determine $t+1$-step lookahead value function

Procedure:

1. Set V^0 arbitrarily, $t = 1$
2. Compute Q^t, V^t from V^{t-1}
    - $Q^{t(s,a)} = \left[R(s) + \gamma \sum_{s'} \Pr(s'|s,a) V^{t-1}(s')\right]$
    - $V^{t(s)} = \max_a Q^{t(s,a)}$

The policy with t stages to go is simply action that maximizes $\pi^{t(s)} = \arg\max_a \left[R(s) + \gamma \sum_{s'} \Pr(s' \mid s,a) V^{t-1}(s')\right]$

- This value iteration approach is dynamic programming (since we solve looping back from t)
- This iteration convergeces exponentially fast (in t) to the optimal value function
- Convergence occurs when $\left\|V^{t(s)} - V^{t-1}(s)\right\| < \varepsilon \frac{1-\gamma}{\gamma}$ ensures V^t is within ε of optimal value. ($\|X\| = \max\{|x|, x \in X\}$)

Asynchronous Value Iteration

Asynchronous Value Iteration: Instead of having to sweep through all states, you can update the value function for each state individually, which then converges to the optimal value function, if each state and action is visited infinitely often in the limit. You can then either store $V[s]$ or $Q[s,a]$

Procedure:

1. Repeat:
    1. Select state s
    2. $V[s] \leftarrow \max_a \sum_{s'} P(s' \mid s,a)(R(s,a,s') + \gamma V[s'])$
    3. Select action a (e.g. using an exploration policy)

Alternative Procedure:

1. Repeat:
    1. Select state s, action a
    2. $Q[s,a] \leftarrow \sum_{s'} P(s' \mid s,a)(R(s,a,s') + \gamma \max_{a'} Q[s',a'])$

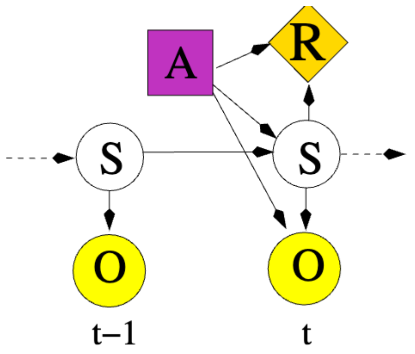
Markov Decision Processes Factored State:

- Represent $S = \{X_1, ..., X_n\}$ and X_i are random variables
- For each X_i, and each action $a \in A$, we have $P(X_{i'} \mid S, A)$
- Reward $R(X_1, ..., X_n) = \sum_i R(X_i)$
- Do value iteration as usual, but only one variable at a time (e.g. variable elimination)

Partially Observable Markov Decision Processes

POMDPs are similar to MDP but some variables aren't observed. It is a tuple $\langle S, A, T, R, O, \Omega\rangle$

- S: Finite set of unobservable states
- A: Finite set of agent actions
- $T : S \times A \to S$: Transition Function
- $R : S \times A \to \mathcal{R}$: Reward Function
- O Set of Observations
- $\Omega : S \times A \to O$: Observation Function



Value Functions: $V^{k+1}(b) = \max_a\left(R^{a(b)} + \gamma \sum_o \Pr(o \mid b,a) V^{k(b_o^a)}\right)$ You can represent $V(b)$ as a piecewise linear function over the belief space—pieces are called α vectors.

Policies

Map belief states into actions, $\pi(b(s)) \to a$ Two ways to compute:

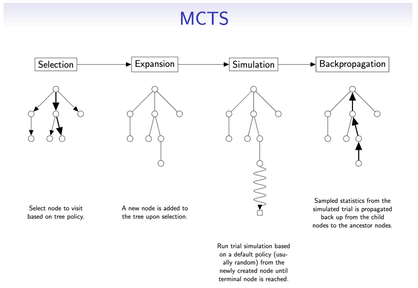
1. Backwards search
    - Dynamic programming (Variable Elimination)
    - In MDP: $Q_{t(s,a)} = R(s,a) + \gamma \sum_{s'} \Pr(s' \mid s,a) \max_{a'} Q_{t-1}(s',a')$
    - In POMDP: $Q_{t(b(s),a)}$
    - Point-based backups make this efficient
2. Forwards Search
    - Expand beliefs going forward
    - $b_{o(s')}^a$ is reached (from $b(s)$) after acting action a and observing o using Transition T and observation Ω functions
    - $b_{o(s')}^a = \sum_{s\in S} T(s' \mid a,s) \Omega(o \mid s',a) b(s) \forall b \in \beta$
    - Can expand more deeply in promising directions
    - Ensure exploration using Monte-Carlo Tree Search

Forward Search for POMDP:

- For each action-observation pair a, o:
    - $b_{o(s')}^a \leftarrow$ propagate $b(s)$ forwards for each action and observation using stochastic simulation/particle filter
    - if $b_{o(s')}^a$ not at a leaf:
        - Evaluate recursively by further growing the tree: $V_o^a \leftarrow \text{GetValue}\left(b_{o(s')}^a\right)$
    - Else:
        - Create new leaf for a, o
            - Do a series of single-belief point rollouts (e.g. propagate a single belief forward stochastically gathering reward until termination condition is met)
            - Use the total returned value as V_o^a
- Return $R(b(s)) + \max_{a\{\gamma \sum_o P(o \mid b(s),a) \sum_{s'} V_o^a b_o^a(s')\}}$

Monte Carlo Tree Search

- Selection: Select node to visit based on tree policy
- Expansion: A new node is added to the tree upon selection
- Simulation: Run trial simulation based on a default policy (usually random) from newly created node until terminal node is reached
- Backpropagation: Sampled statistics from the simulated trial is propagated back up from the child node to the ancestor nodes


MCTS
Selection
Expansion
Simulation
Backpropagation
Select node to visit based on tree policy.
A new node is added to the tree upon selection.
Run trial simulation based on a default policy (usually random) from the newly created node until terminal node is reached.
Sampled statistics from the simulated trial is propagated back up from the child nodes to the ancestor nodes.

11 Reinforcement Learning

Given prior knowledge (possible states of world, possible actions), observations (current state, immediate reward/punishment), and a goal (act to maximize accumulated reward), how do you proceed?

- Given sequence of experiences (state, action, reward, state, action, reward, ...), how do you determine what to do next.
- You must either decide to explore to gain more knowledge, or exploit knowledge you've already discovered.

Difficulties of reinforcement learning:
Credit Assignment Problem What actions are responsible for reward? Reward may have occurred a long time after action that caused it.

- Long term effect of action of agent depends on what it will do in future, but we don't have a model to predict this.

Explore-Exploit Dilemma At each time should the robot be greedy or inquisitive?

Main Approaches:

- Model-Free RL: learn $Q^*(s,a)$ and use this to guide action
- Model-Based RL: Learn a model consisting of state transition function $P(s'|a,s)$ and reward function $R(s,a,s')$ and solve this as an MDP.
- Search through space of policies

Running Average Value

Suppose we have values $v_1, v_2, v_3, ...$, a running average could be $A_k = \frac{v_1 + ... + v_k}{k}$. With new value v_k, we can re-write this as $A_k = \frac{k-1}{k} A_{k-1} + \frac{v_k}{k}$. If we let $\alpha = \frac{1}{k}$ then we simplify to $A_k = (1-\alpha) A_{k-1} + \alpha v_k = A_{k-1} + \alpha(v_k - A_{k-1})$ This is called the **TD**

Formula for temporal difference. We often use this update with α fixed.

Q-Learning

Goal: Store Q[State, Action] and update this as in asynchronous value iteration, but using experience (empirical probabilities and rewards)

- Suppose this agent has an experience $\langle s, a, r, s' \rangle$, that's one piece of data to update Q[s, a]
- The experience $\langle s, a, r, s' \rangle$ provides data point $r + \gamma \max_{a'} Q[s', a']$
- Plugged into TD formula gives: $Q[s,a] \leftarrow Q[s,a] + \alpha(r + \gamma \max_{a'} Q[s', a'] - Q[s,a])$

Procedure:

1. Initialize Q[S, A] arbitrarily
2. Observe current state s
3. repeat forever
   1. select and carry out an action a
   2. Observe reward r and state s'
   3. $Q[s,a] \leftarrow Q[s,a] + \alpha(r + \gamma \max_{a'} Q[s', a'] - Q[s,a])$
   4. $s \leftarrow s'$

Properties:

- Q-learning convergences to optimal policy. No matter what agent does as long as it tries each action in each state enough (infinitely often)
- Exploit: When in state s, select action that maximizes $Q[s,a]$
- Explore: Select another action

Exploration Strategies

- ϵ-greedy: Choose a random action with probability ϵ and choose best action with probability $1 - \epsilon$
- Softmax action selection: In state s, choose action a with probability $\frac{e^{\frac{Q[s,a]}{\tau}}}{\sum_a e^{\frac{Q[s,a]}{\tau}}}$ where $\tau > 0$ is temperature. Good actions are chosen more often than bad actions, and τ defines how often good actions are chosen. For $\tau \to \infty$, all actions are equiprobable. For $\tau \to 0$, only the best is chosen.
- Optimism in face of uncertainty: Initialize Q to values that encourage exploration
- Upper Confidence Bound: Also store $N[s,a]$ (number of times that state-action pair has been tried) and use $\arg\max_a \left[Q(s,a) + k\sqrt{\frac{N[s]}{N[s,a]}}\right]$ where $N[s] = \sum_a N[s,a]$

Off/On-policy learning

- Q-learning does off-policy learning: It learns value of optimal oplicy, no matter what it does
- This could be bad if exploration policy is dangerous
- On-policy learning learns the value of the policy being followed (e.g. act greedily 80% of time and act randomly 20% of time)
- If agent is actually going to explore, it would be better to optimize the actual policy it is going to do
- SARSA uses experience $\langle s, a, r, s', a' \rangle$ to update $Q[s,a]$
- SARSA is on-policy because it uses empirical values for s' and a'

SARSA procedure:

1. Initialize Q[S, A] arbitrarily
2. Observe current state s
3. Select action a using a policy based on Q including exploration
4. Repeat forever:
   1. Carry out action a
   2. Observe reward r and state s'
   3. Select action a' using a policy based on Q
   4. $Q[s,a] \leftarrow Q[s,a] + \alpha(r + \gamma Q[s', a'] - Q[s,a])$
   5. $s \leftarrow s'$
   6. $a \leftarrow a'$

Model-based Reinforcement Learning

- Model-based RL uses experiences in a more effective manner
- It is used when collecting experiences is expensive (e.g. in robot or online game) and you can do lots of computation between each experience
- Idea: Learn MDP and interleave acting and planning
- After each experience, update probabilities and reward, then do some steps of asynchronous value iteration

Procedure:

- Data structures: Q[S, A]; T[S, A, S], R[S, A]

1. Assign Q, R arbitrarily, T = prior counts
2. α is learning rate
3. observe current state s
4. Repeat:
   1. Select and carry out action a
   2. Observe reward r and state s'
   3. $T[s,a,s'] \leftarrow T[s,a,s'] + 1$
   4. $R[s,a,] \leftarrow \alpha \times r + (1-\alpha) \times R[s,a]$
   5. Repeat for a while (asynchronous value iteration):
      1. select state s_1 action a_1
      2. Let $P = \sum_{s_2} T[s_1, a_1, s_2]$
      3. $Q[s_1, a_1] \leftarrow \sum_{s_2} \frac{T[s_1,a_1,s_2]}{P}\left(R[s_1,a_1] + \gamma \max_{a_2} Q[s_2, a_2]\right)$
   6. $s \leftarrow s'$

Q-Function Approximations

Let $s = (x_1, x_2, ..., x_n)^\top$, where x are features

- Linear approximation: $Q_{w(s,a)} \approx \sum_i w_{ai} x_i$
- Non-linear (e.g. neural network g): $Q_{w(s,a)} \approx g(x; w_a)$

Logistic Regression

Logistic function of linear weighted inputs: $\hat{Y}^{\overline{w}}(e) = f\left(w_0 + w_1 X_1(e) + ... + w_n X_{n(e)}\right) = f\left(\sum_{i=0}^n w_i X_{i(e)}\right)$

Sum of squares error is $\text{Error}(E, \overline{w}) = \sum_{e \in E}\left[Y(e) - f\left(\sum_{i=0}^n w_i X_{i(e)}\right)\right]^2$

Partial derivative with respect to weight w_i is $\frac{\partial \text{ Error}(E,\overline{w})}{\partial w_i} = -2\delta f'\left(\sum_i w_i X_i(e)\right) X_i(e)$ where $\delta = \left(Y(e) - f\left(\sum_{i=0}^n w_i X_i(e)\right)\right)$

Therefore each example e updates each weight w_i by $w_i \leftarrow w_i + \mu\delta f'\left(\sum_i w_i X_i(e)\right) X_i(e)$

Approximating Q-Function

- Assign weights $w = \overline{w} = \langle w_0, ..., w_n \rangle$ arbitrarily
- for experience tuple $\langle s, a, r, s', a' \rangle$ we have:
  - target Q-function: $[R(s) + \gamma \max_a Q_{\overline{w}}(s', a)]$ or $[R(s) + \gamma Q_{\overline{w}}(s', a')]$
  - Current Q-function: $Q_w(s,a)$

Squared Error: $\text{Err}(w) = \frac{1}{2}\left[Q_{w(s,a)} - R(s) - \gamma \max_{a'} Q_{\overline{w}}(s', a')\right]^2$

Gradient: $\frac{\partial \text{ Err}}{\partial w} = \left[Q_{w(s,a)} - R(s) - \gamma \max_{a'} Q_{\overline{w}}(s', a')\right] \frac{\partial Q_w(s,a)}{\partial w}$

Weight Update: $w \leftarrow w - \alpha \frac{\partial \text{ Err}}{\partial w}$

Occasionally: $\overline{w} \leftarrow w$

SARSA with linear function approximation

Given γ as discount factor and α as learning rate

1. Assign weights $\overline{w} = \langle w_0, ..., w_n \rangle$ arbitrarily
2. Observe current state s
3. Select action a
4. Repeat forever
   1. Carry out action a
   2. Observe reward r and state s'
   3. Select action a' (using a policy based on $Q_{\overline{w}}$)
   4. let $\delta = r + \gamma Q_{\overline{w}}(s', a') - Q_w(s,a)$
   5. For i = 0 to n
      1. $w_i \leftarrow w_i + \alpha \times \delta \times \frac{\partial Q_w(s,a)}{\partial w_i}$
   6. $s \leftarrow s'$ and $a \leftarrow a'$ and sometimes $\overline{w} \leftarrow w$

Convergence

Linear Q-Learning converges under same conditions as Q-learning $(w_i \leftarrow w_i + \alpha\left[Q_w(s,a) - R(s) - \gamma Q_{w(s',a')}\right] x_i)$

Non-linear Q-learning may diverge. Adjusting w to increase Q at (s,a) might introduce errors at nearby state-action pairs.

Mitigating Divergence

1. Experience Replay
   - Idea: Store previous experiences (s, a, r, s', a') in a buffer and sample a mini-batch of previous experiences at each step to learn by Q-learning
   - Breaks correlations between successive updates for more stable learning
   - Few interactios with environment are needed to converge, which increases data efficiency
2. Use two Q functions: Q network (currently being updated) and Target network (occasionally updated)
   - Idea: Use separate target network updated only periodically
   - Target network has weights $\overline{w}$ and computes $Q_{\overline{w}}(s,a)$
   - Repeat for each (s, a, r, s', a') in a mini-batch:
     - $w \leftarrow w + \alpha\left[Q_{w(s,a)} - R(s) - \gamma Q_{\overline{w}}(s', a')\right] \frac{\partial Q_w(s,a)}{\partial w}$
   - $\overline{w} \leftarrow w$

Deep Q-Network:

1. Assign weights $\overline{w} = \langle w_0, ..., w_n \rangle$ at random in $[-1, 1]$
2. Observe current state s
3. Select action a
4. Repeat
   1. Carry out action a
   2. Observe reward r and state s'
   3. Select action a' (using a policy based on $Q_{\overline{w}}$))
   4. Add (s, a, r, s', a') to experience buffer
   5. Sample mini-batch of experiences from buffer
   6. For each experience $\left(\hat{s}, \hat{a}, \hat{r}, \hat{s'}, \hat{a'}\right)$ in mini-batch:
      1. Let $\delta = \hat{r} + \gamma Q_{\overline{w}}\left(\hat{s'}, \hat{a'}\right) - Q_w(\hat{s}, \hat{a})$
      2. $w \leftarrow w + \alpha \times \delta \times \frac{\partial Q_w(\hat{s},\hat{a})}{\partial w}$
   7. $s \leftarrow s'$ and $a \leftarrow a'$
   8. every c steps, update target $\overline{w} \leftarrow w$

Policy search using policy gradient

- Instead of approximating Q, we can also approximate policy π
- $\pi(a \mid s)$ is now a function that maps state s to a distribution over actions a
- Let $s = (x_1, ..., x_n)^\top$
- Linear: $Q_w(s,a) \approx \sum_i w_{ai} x_i$, $\pi_\theta(a \mid s) \approx \sum_i \theta_{ai} x_i$
- Non-linear (neural networks) $Q_{w(s,a)} \approx g(x; w_a)$, $\pi_{\theta(a \mid s)} \approx g(x; \theta_a)$

To change parameters θ of policy π_θ, consider V, Q to be value functions under π_θ. $d^{\pi(s)} = \lim_{t \to \infty P(s_t = s | s_0, \pi_\theta)}$ is the stationary distribution of Markov chain for π_θ

Our objective to maximize is $J(\theta) = \sum_{s \in S} d^{\pi(s)} V(s) = \sum_{s \in S} d^{\pi(s)} \sum_{a \in A} \pi_\theta(a \mid s) Q(s,a)$

Policy Gradient Theorem

- Finding a gradient is difficult because it depends on both action selection π_θ and state distribution $d(s)$ but updating policy changes the state distribution. But policy gradient theorem allows us to simplify gradient.

$$\begin{aligned}\nabla_\theta J(\theta) &= \nabla_\theta \sum_{s \in S} d^{\pi(s)} \sum_{a \in A} \pi_\theta(a \mid s) Q(s,a) \\ &\approx \sum_{s \in S} d^{\pi(s)} \sum_{a \in A} Q(s,a) \nabla_\theta \pi_\theta(a \mid s) \\ &= \sum_{s \in S, a \in A} d^{\pi(s)} \pi_{\theta(a \mid s)} Q(s,a) \frac{\nabla_\theta \pi_{\theta(a \mid s)}}{\pi_{\theta(a \mid s)}} \\ &= \mathbb{E}_\pi\left[Q(s,a) \nabla_\theta \ln \pi_{\theta(a \mid s)}\right]\end{aligned}$$

where $\mathbb{E}_\pi$ refers to $\mathbb{E}_{s \sim d^\pi, a \sim \pi_\theta}$ where both state and action distributions follow the policy (on policy)

REINFORCE algorithm

- Use sampled trajectories to compute expectation over $Q(s,a)$

$$\begin{aligned}\nabla_\theta J(\theta) &\approx \mathbb{E}_{\pi\left[Q(s,a) \nabla_\theta \ln \pi_{\theta(a \mid s)}\right]} \\ &= \mathbb{E}_\pi\left[\left(\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}\right) \nabla_\theta \ln \pi_\theta(a_t \mid s_t)\right]\end{aligned}$$

1. Initialize θ to random
2. Sample one trajectory using π_θ: $s_1, a_1, r_2, s_2, a_2, ..., s_T$
3. For $t = 1, 2, ..., T$
   1. Estimate $G_t = \sum_{k=0}^{T-t-2} \gamma^k r_{t+k+1}$
   2. Update $\theta \leftarrow \theta + \alpha \gamma^t G_t \nabla_\theta \ln \pi_{\theta(a_t \mid s_t)}$

The algorithm samples trajectories to estimate gradient results in high variance. The goal is to subtract a baseline value from return G_t to reduce variance of gradient estimation. You can also use **Advantage** $A(s_t, a_t) = Q(s_t, a_t) - V(s_t)$ instead of G_t

Actor-Critic

- Learn two models, Actor π_θ and critic Q_w
- α and β are learning rates
- Use actor to update $\pi_{\theta(a \mid s)}$ in direction suggested by critic.

```
Initialize $s$, $w$, $\theta$ to random
Sample $a \sim \pi_\theta(a|s)$
**For** $t = 1, 2 \ldots T$
  Sample reward $r_t \sim R(s,a)$, next state $s' \sim P(s'|s,a)$
  Sample next action $a' \sim \pi_\theta(a', s')$
  Update $\theta \leftarrow \theta + \alpha Q_w(s,a) \nabla_\theta \ln \pi_\theta(a|s)$
  Compute TD correction $\delta_t = r_t + \gamma Q_w(s', a') - Q_w(s,a)$
  Update $w \leftarrow w + \beta \delta_t \nabla_w Q_w$
  Update $a \leftarrow a'$, $s \leftarrow s'$
**end**
```

Bayesian Reinforcement Learning

- Include parameters (transition function and observation function) in state space
- Model-based learning through inference (belief state)
- State space is now continuous, belief space is a space of continuous functions
- Can mitigate complexity by modeling reachable beliefs
- Optimal exploration-exploitation tradeoff