The AI Umbrella

* Search
  + Google maps
* NLP
  + Google translate, text-to-speech
* Computer Vision
* Robotics
* Reinforcement Learning
* Biometrics
  + Fingerprint scanner, face scanner

Our Plan

1. Common verbiage
   1. Agent
   2. World
   3. Representation
2. Search & Planning
   1. A\* algorithm
   2. Tree/graph expansion
   3. – ß pruning
   4. First order logic
3. Probabilistic Representation & Problem Solving
   1. Bayesian Networks
   2. Belief
4. Supervised & Reinforcement Learning
   1. Decision Trees
   2. Neural Networks
   3. Reward Functions
   4. Bellman Equation
5. Special Topics

Agents

* An entity in AI which makes decisions on its own behalf is called an agent (think agency)
* Agent function: how the agent works (internally)
* Agents can make multiple decisions at once
* Exist in the digital realm and accept data and take actions in the digital realm

Environment/World

* Agents interact with the physical world through an environment (often called a world)
* Agents must “sense” their world in order to receive data
  + I.e. robots have sensors
  + Environments have representations

Rational Agents

* Rational agents are agents that “do the right thing”
  + Evaluated by a performance metric
  + Performance metrics examine environment state
  + Agents that produce desired environment states -> rational
* Rationality is different than omniscience
  + Rationality optimizes expected performance, omniscience optimizes actual performance

Task Environment

* Remember: rationality requires four things:
  + Performance metric
  + Environment (description/software/interface, etc.)
  + Agent senses
  + Agent actions
* Task environment: these four attributes

Flavors of Task Environment

* Fully observable
  + Agent has access to complete state of environment
* Partially observable
  + Some information hidden from agent
* Deterministic
  + Environment’s next state completely dependent on action + previous state
* Stochastic
  + Environment’s next state is probabilistic
* Single agent
* Multi-agent
  + Competitive/cooperative
  + Communication?

Flavors of Agents

* Simple Reflex Agent
  + Only looks at current env state to decide action
* Model-based Reflex Agent
  + Maintains internal state (agent state) + can use history of env states
* Goal-based Agent
  + Goal function assists agent in making decisions (helps identify favorable states)
* Utility-based Agent
  + Utility function = internal performance metric… used to assist agent
  + Utility function does not need to match Task env performance metric
* Learning Agent
  + 1) Learning element: component that makes adjustments to the agent
  + 2) Action element: component that selects external actions
  + 3) Critic: feedback component (used by the learning element)
  + 4) Problem generator: component that suggests novel experiment

Search I

Env State

* Agent senses: collect information about the env at a point in time
  + Let us assume we have the state of the world st at time t
* Given st, our agent must produce an action at (also at time t)
* World transitions to new state st+1
  + If static, world transitions to st+1 after at is made

Agents that Search

* For now, lets assume that the state of the world is unstructured
  + Atomic: state contains no internal structure
* Let’s consider goal agents
  + Agents equipped with goal function
  + Remember: goal function g(st) -> R

Worlds with Known states

* Let’s consider worlds where all states are known beforehand
* Construct graph of states Genv = (Venv, Eenv)
  + Venv = set of all states in the world
  + (u,v) Eenv (undirected) if there exists an action which transitions u to v
* Example: maze solving in a grid world
  + State determined by pos of player (and pos of obstacles)
  + Actions: {up, down, left, right}

Uninformed (Blind) Search Strategies

* Breadth First Search (BFS)
  + Start at source vertex
    - Visit all neighbors
    - Visit all neighbor’s neighbors…….
  + Key: only explore simple paths
  + Problems with BFS
    - Current form does not work with action costs (edge weights)
    - Memory-intensive O(|V|2
    - BFS is blind
* Depth First Search (DFS)
  + Recursively probes “deep” into paths
    - Can also implement with a stack
  + Finding simple paths: do not expand a node that is already visited
    - Algorithm has unbound time if not
  + NOT optimal: returns first path found
  + Same memory complexity as BFS in worst-case but better average-case
  + Can bind depth: depth limited search
    - Called diameter of state space
  + Iterative deepening DFS:
    - Gradually increase depth limit and rerun DFS (diameter = 0,1,2…)
    - Stops when first solution is found
    - Optimal when path cost is monotonically-increasing as a fn of vertex depth
  + Problems with DFS
    - Path it finds may not be optimal
      * Just guarantees a path exists
    - DFS is also blind
* Dijkstra’s Algorithm
  + Modification of BFS
  + Key idea: shortest paths contain shortest paths
  + Neighborhoods don’t necessarily correspond to longer paths
  + Idea:
    - Keep collection of all paths explored so far
    - Expand the smallest one

Utilities and Estimating Cost to Goal

* Utility function estimates path cost from a vertex to the goal
* Total estimated cost from src 🡪 goal = true cost + estimated cost

Utility Constraints

* Cost function = path cost + utility cost (heuristic)
* As long as utility is admissible & consistent: A\* is optimal
  + Admissible = never overestimates the true cost (i.e. utility is optimistic)
  + Consistent = triangle inequality but for action spaces:
    - f(u) c(u,a,v) + f(v)
      * cost of vertex u (to goal)
      * cost of using action a to transition from u to v
      * cost of vertex v (to goal)

Search II

Making Dijkstra Less Blind

* Design goal: make Dijkstra aware of the goal state
  + Choose to expand the path that leads “towards” the goal
  + Rather than sort by true path cost (i.e. known path cost)
    - Sort by estimated path cost from src 🡪 goal

The A\* Algorithm

* Literally just modified Dijkstra’s
  + Gets s.p. so far based on total estimated cost src 🡪 goal
  + Everything else is the same as Dijkstra
* Setback: heuristic needs to be engineered for every world

Local Search I

Problems with Classical Search

* Blind search + informed search are nice:
  + Optimality (most algorithms)
  + Finds solution to the problem
    - Path to goal from start
* What happens if we don’t care about the path?
* Not appropriate for large-scale worlds
* Does not model some real-world problems

Optimization

* Find the best state according to an objective function
* Can always convert a min 🡨🡪 max
* Classical search is optimization (for paths):
  + Path cost is objective function
  + Find the path (from source to destination) with min cost

How to Optimize

* Geometric angle
  + For every state, use objective to get value of state (“objective value”)
    - Draw the curve
  + Algorithms “move” along the surface of the curve
    - If min problem, find the lowest point on the curve
    - If max problem, find the highest point

Optimization with Local Optimizers

* Path on surface = trajectory
* Local optimization only looks at “local” region to decide where to go next

Hill Climbing

* Goal: move on the surface (to a goal state)
* Algorithm: when at state *s*:
  + Generate all children of s, select child s’ with the “best” utility value
  + If s’ is better than s, move to s’ and repeat
  + Else give up and return s
* Greedy: can get stuck at local optima
* Fast & memory efficient:
  + Generally easy to move improve state early on
  + Only keep best state in memory
* Problem: what happens if we reach a plateau?
  + Hill climbing will exit
* Allow “sideways” move on plateaus
  + Accomplished with thresholding (how to choose)
  + Accomplished probabilistically (flip a weighted coin)
    - Probability of moving can depend on “steepness” of hill before plateau
* Random restarts: run multiple independent hill-climbers
  + Different random initializations 🡪 different trajectories
  + Return the “best” solution amongst trials

Simulated Annealing

* + Short term loss can lead to long term gain
* Hill climbing 🡪 always go in a “good” direction
* Simulated Annealing:
  + Modify hill climbing
    - Instead of picking the best move, pick a move at random (candidate)
    - If candidate is good, accept it and repeat
    - If candidate is not good, accept with some probability, otherwise repeat
  + Scale probability of accepting a bad move as function of time

Local Search In Continuous Spaces

* So far, algorithms proposed are bad for continuous spaces
* Problem: enumerating child states
  + Potentially infinite
* Can still measure “good” and “bad” states
  + Use calculus, not enumeration

Derivatives

* If we have an objective f which is continuous (or piecewise continuous)
  + Can still optimize
* Optima can be found using 1st and 2nd derivatives
  + 1st tells us where optima is
  + 2nd derivative tells us what kind of optima it is

Derivatives in Higher Dimensions

* Called a “gradient”
* Algorithm:
  + For every variable in f:
    - Calculate the partial derivative
    - Collect partial derivatives into a vector

Derivatives & Gradients

* Sometimes:
  + We can set the derivative to 0 and solve (rare case)
  + Formula we get is called a closed form solution
  + INCREDIBLY rare
* What if we can’t get a closed form solution?
  + Derivative + gradient always points towards local maximums
  + Continuous hill climbing?
    - Called gradient descent/ascent

Gradient Descent/Ascent

* Hill Climbing-esque:
  + Instead of enumerating children, calculate gradient
  + Use gradient to update state:
    - = step size (step will be proportional to )
* How to choose ?
  + Complex:
    - Vary by time
    - Vary by objective surface geometry
* Newton-Raphson
  + Designed for finding the roots of equations
  + In 1d:
  + In higher dimensions:

Local Beam Search

* Keep k states in memory (the beam):
  + For every state in the beam, generate all possible child states
  + Recompute the beam: keep the top k best states (according to objective)
    - If any child is the goal state, stop
* How to encourage diversity in the beam?
  + Difficult: lots of risk for imposing our own beliefs on states
  + Idea:
    - Rebuild beam probabilistically (stochastic beam search)
    - Instead of top k children: choose k at random
      * Probability of choosing a child ~ objective value
      * “natural selection”-ish

Genetic Algorithms

* Form of stochastic beam search
  + Beam=population
  + Requires a “fitness function” (objective function)
    - Larger values are better
  + Child state is product of two parent states (not one)
    - Parents can be chosen deterministically or probabilistically
    - Mutation probability
* GAs can operate directly on states OR directly on agents
  + “states”, “agents” 🡪 “subject type”

Theory of Gas

* Schema = Fixed part of the subject type
  + Some properties fixed, some properties left free
* Individuals that match the schema = instances of schema
  + Let S = {all possible schema individuals}
  + Let I = {all possible individuals}
  + Let Pt = {population of individuals at time t}

CSP I

Tree Search

* Expand tree to a new level
  + Consider new moves to apply!
  + Leaf nodes in tree should be terminal states
    - Problem when leaf nodes are nonterminal
* Can apply this to single player games

Tree Search Example

* Want to assign a color (RGB) to each region of Australia
  + Model each region as a graph
  + Add edge connecting adjacent regions
* Problems
  + What if we have constraints?
    - (in this example: adjacent regions can’t have the same color)
  + Tree is massive
  + Ordering of vertices don’t need to be permuted
    - Solution doesn’t depend on ordering of vertices

CSP

* A CSP – Constrained Satisfaction Problem – meets this template
* Variables
  + Each variable Xi has its own domain Di = {vi,v2,…,vk}
    - Possible values that can be assigned to Xi
  + Each variable must be assigned to a value
* Constraints C = {C1,C2,…,Cm}
  + Each constraint is Boolean: relates variables to each other
  + In map coloring:
    - Adjacent variables must have different colors
      * Cj = Xl != Xp
* An assignment A:
  + Set of variables with their assignments
  + A partial assignment = not all variables have an assignment
  + A complete assignment = all variables have an assignment
  + A legal assignment = assignment satisfies constraints
* GOAL: find a complete + legal assignment
  + Pick ordering of variables (reduces tree size)
  + DFS the tree!
* It is possible to not find a complete and legal assignment

Tree Pruning

* Typically model CSPs as a constraint graph
  + Every n-ary (n>2) constraint can be converted to a bunch of binary constraints
  + Each variable becomes a vertex
  + (unary/binary) constraints become edges
* Goal: prune domain Di for variable Xi
  + Pruning domain = pruning tree!
  + Say Xj has some unary constraints:
    - Reduce domain to all values that satisfy this constraint
    - Node Consistency (1-consistency)
  + Lets say Xi, Xj participate in some binary constraint c
    - When we have an assignment Xi = v
      * Reduce Xj’s domain to values that satisfy c knowing Di = {v}
      * Arc consistency (2-consistency)

AC-3 + Revise: Forward Checking Neighbors

* Queue = {C(Xi,Xj)}(I,j)
* While queue is not empty:
  + C(Xi,Xj) = queue.pop()
  + Di, Dj = domains of Xi and Xj
  + Revised = false
  + For each xi in Di:
    - If no xj in Dj satisfies C(Xi,Xj):
      * Dj.remove(xi)
      * Revised = true
  + If revised is true:
    - If Di is empty:
      * Return False
    - for each *Xk* in *Xi*.neighbors.remove(*Xj*):
      * *queue*.append(*C*(*Xk*, *Xi*))
* return True

Review

* How do we know if a CSP is solvable
  + We don’t know beforehand, can check through in polynomial time
* Vocab terms
  + Vertex is node consistent iff no value in its domain breaks an unary constraint
  + Vertex is arc consistent iff no value in its domain breaks a binary constraint
    - Arc consistent with respect to another vertex

DFS Tree Search & Backtrack Algorithm

* Function Backtracking-search(csp) returns a solution, or failure
  + Return backtrack({}, csp)
* Function backtrack(assignment,csp) return a solution, or failure
  + If assignment is complete then return assignment
  + Var = select-unassigned-variable(csp)
  + For each value in order-domain-values(var,assignment,csp) do
    - If value is consistent with assignment then
      * Add {var = value} to assignment
      * Inferences = inference(csp,var,value)
      * If inferences != failure then
        + Add inferences to assignment
        + Result = backtrack(assignment, csp)
        + If result != failure then

Return result

* + - Remove [var = value] and inferences from assignment

CSP Heuristics

* Heuristics in past problems:
  + Domain-specific knowledge
  + Task engineering
* Heuristics in CSPs:
  + More abstract
  + Apply to all CSPs
* Variable ordering:
  + SELECT-UNASSIGNED-VARIABLE
  + Goal: prune the tree
    - Rather than rely on a fixed ordering of variables
    - Pick new variable based on what other’s have already been chosen
    - Pick variable with smallest domain remaining
      * Minimum Remaining Values (MRV) heuristic
      * Fail-first heuristic
    - Degree heuristic:
      * Pick variable involved in the most constraints
    - Can combine multiple heuristics
      * Use MRV and settle ties with degree heuristic
    - Value ordering
      * ORDER-DOMAIN-VALUES
        + Choose value that is most “flexible”
        + Least constraining Value (LCV) heuristic:

Prefers domain values that affect neighbor domains the least

Fail-last heuristic

Inference During Search

* AC-3
  + When a domain was pruned:
    - Propagate those changes to neighbors
  + This is an inference procedure!
* How can we use leverage to prune during searching?
  + Whenever we make an assignment:
    - Infer domain reductions on neighboring vertices

Forward Checking

* A flavor of inference
  + Whenever we add Xi = v to assignment
  + Make neighbors Xj of Xi arc consistent with Xi
* Note: this is useless if AC-3 was run as preprocessing step
  + Very useful inference if you choose not to run AC-3 beforehand

Maintaining Arc Consistency (MAC)

* Another flavor of inference
  + Whenever we add Xi = v to assignment
  + Call AC-3
    - Instead of the queue initially populated with all edges in the CSP:
      * Queue initially populated with all edges to unassigned neighbors of Xi
* Note: also useless if AC-3 was run as a preprocessing step
  + Super useful if you don’t do AC-3 beforehand

Better Backtracking

* Running a DFS search
* When the recursive call fails:
  + Return to most recent DFS call
  + Try something else
  + “chronological backtracking”
* Problem: entire subtree we are exploring may already be doomed
  + May have been doomed long ago
  + Continuing to explore this subtree is wasteful
  + Goal: backtrack to the doomed variable
* How to know when we’re doomed?
  + Idea: keep track of values that conflict with a variable
    - Eg: conflict set for SA = {WA=R,NT=G,Q=B}
    - When we get to SA and need to backtrack:
      * Go to most recent addition to the conflict set for SA
        + Called “backjumping”
      * Potentially skip over lots of other decision points in the tree!
  + How to compute these conflict sets?
    - Forward checking gives them to us for free!
    - Assign Xi=v and call forward checking
      * Whenever we delete from Xj’s domain: add (Xi=v) to Xj’s config set!
* But forward checking prunes before we get there
  + Every branch pruned by forward checking is pruned by backjumping
  + “simple” backjumping is redundant when we use forward checking or MAC

How to Compute Conflict Sets

* Idea of backtracking is still good:
  + When you run into a conflict: don’t chronologically backtrack
  + Backtrack to the source of the problems
* Key is to not forget future problems:
  + Dooming a branch is not (necessarily) the fault of one individual variable
  + Multiple variables together doom a branch
* Whenever we backjump:
  + Let’s say the variable we backjump to is Xj
  + We came from Xi
  + Xj’s conflict set should “absorb” Xi’s conflict set
    - There is no solution from Xj onward given the preceding assignments to Xj’s new conflict set
  + Conf(Xj) = conf(Xj) U conf(Xi)-{Xj}

Backtrack

* Let’s pretend no inference
* Static ordering of variables & domains

Probability and Stats Refresher

* Random Experiment:
  + Process where we observe something uncertain
* Outcome: result of the experiment
* Sample Space: set of all possible outcomes
* Event: any subset of the sample space
* obeys three axioms:
  + For any event ,
  + For two events , if , then

Conditional Probability

* Consider two events, A and B
* If we know A has already occurred, what can be said about Pr[B]?
* Roll two die; let X1 = value of die 1, X2 = value on die 2
  + Pr[X1 + X2 5]?
  + Pr[X1 + X2 5|X1 = 2]?
* Formally if Pr[A] != 0:
  + Pr[B|A] =

Independence

* Two events A and B are independent if Pr[A|B] = Pr[A] (and vice versa)
  + Therefore = Pr[A]Pr[B]

Law of Total Prob & Total Conditional Prob

* Let E1, E2, … En partition the sample space (i.e. no overlap)
* For any event A:
  + Pr[A] =
* Special case is when B is Boolean
  + Pr[A] =

Events vs Random Variables

* Random variable is a more useful tool:
  + Placeholder for an outcome
  + A function:
    - Assigns a value to each possible outcome of a random experiment
  + When R.V. is tied to some values:
    - It becomes an event
* “Binding” a R.V. creates an event
  + Notationally very elegant to use
* Probability Mass Functions (PMFs) and Prob. Density Functions (PDFS)
  + PMF:
    - Assigns probability to every element in range of R.V.
    - obeys axioms of probability
  + PDF:
    - Assigns probability to every interval within range of R.V.
    - obeys axioms of probability

R.V.s and Independence/Conditionals

* Independence defined same way as for probability functions
* Consider to discrete R.V.s X and Y:
* Can also do conditional probabilities:

Cumulative Distribution Functions (CDFs)

* Measures

Bernoulli Trials

* Binary experiment (flip a coin that has prob p of landing H)
* Let X be 1 if coin in H, otherwise 0
  + Binary R.V.
  + Notation: X~Bernoulli(p)
* Binomial Distribution
  + Count the number of successes in *n* independent Bernoulli trials
  + Let Yi be the R.V. for an ith trial – Yi~Bern (p)
  + Let X be the number of successes in *n* trials X~Bin(p)
* Geometric Distribution
  + The number of independent Bernoulli trials until the first success occurs
  + Let Yi be the R.V. for an ith trial Yi~Bern(p)
  + Let X be the number of trials that occur when the first success occurs X~Geom(p)

Is the Geometric Distribution Actually a Distribution

* Have to check!
* Rx? {1,2,3,…}
* is always 0
* Does ?

Markov inequality:

* + Only works if all values are nonnegative

Chebyshev inequality:

Probabilistic Agents I

* Pitfall
* Modeling the world
  + Let Bi,j be R.V. for whether breeze in (j,j)
  + What random variable do we need to track?

Bayesian Network

* A Directed Acyclic Graph (DAG)
  + Every R.V. is a vertex
  + Directed edge = conditional relationship between R.V.s
* Formally:
  + Each vertex corresponds to a R.V.
  + Directed edges connects pairs of vertices
    - Parent/child relationship
  + Each vertex Xi stores conditional probability distribution
    - Pr[Xi|Parents(Xi)]
* Discrete Distributions
  + When R.V.s are discrete
    - Need to store entire pmf
      * Each row contains specific conditioning case distribution
      * Each row is a separate distribution
      * Shortcut indicator for R.V.s:
        + Only need 1 column (2nd col is 1-Pr[CRS =t])
      * Indicator R.V. w/ k indicator R.V. parents has how many values in CPT?
        + Counted with uncertainty in edge between CouchRippingSound and WifeMessage
        + Blessing: lots of different ways event might/not happen
* First way to view a Bayesian Network:
  + Representation of the joint probability distribution
* View Network as DAG w/ parameters attached to each vertex
  + For now Pr[Xi|Parents(Xi)] 🡪0[Xi]Parents(Xi)]
  + If we ever want to know Pr[X1=x1…Xn=xn]
    - How can we use 0[Xi]Parents(Xi)] to calculate this?
      * If we pick model:
* How do we build good representations of ?

Building Bayesian Networks

* Procedure:
  1. Write down the set of R.V.s needed to model the domain. Any order will work, but:
     + Causes before effects
     + “Direct influence”
  2. For i=1 to n do:
     + Choose from {X1,…,Xi-1} a minimal set of parents for Xi.
     + Add directed edge from the parent to Xi
     + Store the conditional probability distribution
* Reminder
  + Need to divide and conquer
  + Bayesian networks = compact joint distribution representations
    - Take advantage of local structures
  + Causal relationships are better than diagnostic ones
* Topological semantics 🡪 numerical semantics
  + Each R.V. is conditionally independent of its non-descendants given parents
  + Each R.v. is conditionally independent of all other R.V.s given parents, children, and children’s parents (Markov Blanket)

Efficient Representation of Conditional Dists

* Lets say we have a CPT with k R.V.s
  + Need 0(2k) values in the best case
  + Lots of times, these values follow pattern
    - Called a canonical distribution
    - If we figure this pattern out, we don’t need to store all values, can just generate them instead

Noisy Logical Relationships

* In prepositional logic:
  + Fever=t iff cold=t OR flue=t OR malaria=t
  + What about inhibitors
  + Two assumptions
    - All possible causes are listed
    - Inhibition of parent is independent of inhibition of other parents
  + Can build entire CPT from small inhibitions

Continuous R.V.s

* Can always discretize to make discrete R.V.s
* Use PDFs:
  + Two new distributions
    - Continuous R.V. conditioned on discrete/continuous parents
    - Discrete R.V. conditioned on continuous parents
* Pr[cont.|cont.,discrete]
  + Pr[c|h,s] ~ N(a1h + b1, o12)

Bayesian Network for Pitfall

* Variables needed: P1,1,P1,2,…B1,1,B1,2…
* Order variables: Bi,js caused by Pi,js
* Compute parents
  + Breezes caused by adjacent pits
  + Pits are i.i.d

Inference in Bayesian Networks

* Once we have the network and conditional distributions, want to use it to answer queries
  + Consider queries with only one variable:
  + This is called a posterior distribution
  + Observations are known values for other R.V.s in the network
    - Called evidence variables E =
    - E is the current bindings for evidence variables
  + Lots of other “unobserved” variable in the network
    - Called “hidden variables” Y =
    - Let y be a hypothetic binding to unobserved variables
  + All variables in the network X =

Exact Inference

* How do we compute exactly?

The Variable Elimination Algorithm

* Whenever we sum out factors
  + Any factor that doesn’t depend on the R>V. being summed out is a constant, move outside the sum
* Not that x operator is lazy: only happens when we need to sum out a R.V.

Variable Elimination and ORDER

* Different orderings of variables produce different intermediary factors
  + Want to choose the ordering that does less work!
  + Interactable to know optimal ordering
    - Can order using heuristics that are “good enough”
    - Common choice: greedy heuristic
      * Eliminate whichever variable minimizes the size of the output factor
* In general:
  + Remove any leaf node that is not a query R.V. or an evidence R.V.
  + Every R.V. that is not an ancestor of a query R.V. or evidence R.V. is irrelevant

Runtime for Exact Inference

* Depends heavily on the topology of the graph
  + If you graph a polytree (singly connected graph):
    - Time + space complexity is linear in size of the graph
    - Upper bound for parents is a constant 🡪 O(|V|)
  + For multiply connected graphs:
    - Variable elimination can have exponential time/space even if upper bound for parents is bounded
* CSPs + Bayesian networks are related

Naïve Bayes

* Since we assumed conditional independence:
  + Only need to focus on one feature at a time
    - Pr[xOutlook|Y]
    - Pr[xhumidity|Y]
    - Pr[xtemp|Y]

Continuous Features

* The node no longer contains a mpf
  + Parameterize with a pdf
    - Assume any pdf and learn the parameters of the pdf from data using MLE estimates

Supervised Learning

* This time we have inputs and the ground truth given too us
  + Truth came in two forms:
    - Bellman equation w/ rewards
    - Policy Gradients
* Ground truth
  + Pro: we don’t worry about generating it
  + Con: what is the quality of the dataset?

The Data

* Theoretically we have a data space
  + Want:
    - Dataset contains every possible sample (and label)
    - Don’t know how to distinguish “hard” and “easy” examples
  + Settle for:
    - Dataset that is representative of the space
* Assumption:
  + Data we have is drawn i.i.d from data space
  + Dataset =
* The Hypothesis Space
  + If we have N data points, what functions could we learn?
  + Goal: Search through possible functions to select the best one
* Important factor:
  + Our data is most likely not the entire data space, with no way to check if it is representative
  + We want our function to generalize
    - Function needs to perform well on data it hasn’t seen before

How to get Unseen Data

* Split your dataset into a training set Dtrain and a testing set Dtest
* Methodologies:
  + Cross validation
  + Random split
  + Fixed partitions
* For now let us assume random splits
  + Something like 90% of your examples go into Dtrain and 10% into Dtest

The Learning Objective

* Hypothesis space – H
* Evaluate the quality of a function
* Therefore:

Training/Validation/Testing Data

* Instead of splitting data into Dtrain and Dtest
* Make three groups: Dtrain, Dval and Dtest
  + Dval is quiz data: data that h doesn’t directly train on but we use to check learning
* When we claim we have found the “best” h, then give it a test (Dtest)
* How to split?
  + Typically 90%/5%/5%

Problem

* ML is ill posed:
  + The more we train, the less likely we will generalize
  + Need to train though!
* Why?
  + If we only have training data:
    - What is inside of H?
    - If our task is realizable, then htrue H
    - What else is in there?
      * Avoid hmemorize at all costs
      * Can’t tell difference with training data

Information Gain & Entropy

* Entropy tells us how “well distributed” a distribution is
  + Think of it like compression: how many bits needed to compress distribution?
    - If distribution is skewed: fewer bits
    - If distribution is balanced: more bits
* How can we use this to determine the “quality” of a split?
  + Measure entropy before & after
  + Measure the difference (information gain)

Continuous Features

* Sort the features and ground truth by the feature value
* Potential values of t?
  + Infinitely many
    - Can restrict to those within range of feature

Decision Boundary

* Lets think about models geometrically
  + Consider 2d data w/ a Binary classification task
* How do decision trees chop up the feature space?
  + Let’s consider continuous features in this example
  + Each node picks a feature and makes a binary split
    - Splits are always axis-parallel cuts

Tree Complexity

* Trees can get very big; the bigger the tree, the more complex the decision surface
* More complex functions = more likely to memorize
* Fix: only make a bigger tree if it’s necessary

Decision Tree Pruning

* Two kinds of pruning:
  + Pre-pruning:
    - Prune while the decision tree is being constructed
    - Stop growing when some criteria is met
  + Post-pruning:
    - Prune subtrees after it grows
    - Two types:
      * Top-down pruning
      * Bottom-up pruning

Top-down Post Pruning

* Pessimistic Error Pruning:
  + When at node n that has children c1,c2,…,ck
    - Delete a child one at a time, measure % error
      * Estimate generalization error e’(.)
      * e’(.)e(DTrain) + ½|nleaves|
    - If % error increases, leave child alone
    - If % error decreases, prune child
  + Repeat for all surviving child nodes

Bottom-Up Post Pruning

* Reduced Error Pruning
  + Use validation data to estimate generalization error
* Minimum Error pruning
  + Build sequence of trees from initial tree
* Minimum cost complexity pruning
  + Also build sequence of trees from initial tree
  + Each tree (in sequence) created by removing a subtree (from prev. tree in sequence)
    - How to pick subtree to remove?
    - Pick subtree n that minimizes:

Minimum Description Length (MDL)

* Often used to measure complexity of a function:
  + MDL(h) = # bits(h) + # bits(D|h)
    - Bits used to describe h + bits for data to be described by h
* For decision trees:
  + MDL(T) = size(T) + # misclassifications(T)

Images & Your Eyes

* How do your eyes parse visual data?
* How do we localize different patterns in images
  + Answer: convolution

Convolutional NNs

* Implement convolution operator (over images) using learnable kernels
* Generate kernels (like gabor wavelets)
* Successive layers 🡪 more abstract features

Backpropagation Through Time

* What about sequences? Current NNs only deal with grids/vector input
* Recurrent NNs: derived from state machines

Sequential Problems

* Stochastic environment
  + No longer an “easy” search
  + Deterministic solution
    - [^ ^ > > >]
    - Actions unreliable, may not go according to play

Trajectories

* The observed sequence of states and actions is called a trajectory
* From a single start state, lots of trajectories possible

Markovian Assumption & Utilities

* Transition model Pr[s’|s,a] = Pr[s’|st,st-1,…,s0,a]
  + Transition probability does not depend on history, only on the current state and action
* Utility function?
  + No longer dependent on a single state
  + Depends on sequence of states
  + Define a reward function: R(s)
    - Must be bounded

Utilities from rewards?

* Many ways of doing this
* For now, consider utility = sum of rewards along that trajectory
* Utility of a 10-step trajectory (ending in positive goal) = 0.6

MDP

* A sequential decision problem
  + Fully observable world, stochastic environment, Markovian transition model
  + Additive rewards

Optimal Policies

* Suppose we have policy pi
  + Every time we start a new turn, trajectory may be different
* The quality of a policy is the expected utility of possible trajectories generated by that policy
* An optimal policy maximizes this expected utility

Finite Horizons

* A finite horizon = there is a fixed time N after which no further actions matter (i.e. we consider the game over)
* Utility for a trajectory Uh(s0,s1,…,sN+k)
  + For example, N = 3 (agent has a 3-move budget)

Infinite Horizons

* No max. move budget
* No reason to behave differently in the same state at different times

Uh(T)

* Additive rewards: the utility of a state sequence is
* Discounted rewards: the utility of a state sequence is

Value Iteration

* General idea:
  + Calculate U(s) for each state s
  + Use utilities to select optimal action in each state
* Bellman equation
  + s’ – next state
  + s – current state
  + U(s) – utility of current state
* If we know U(s) for each state, we can select the optimal action
* How to get these utilities?
  + Each state gets its own bellman equation (for that state)
  + n states 🡪 n equations
  + The n equations collectively have n unknowns
  + Problem: Bellman is nonlinear – can’t represent with lin. Alg.
    - Can instead solve iteratively
    - Initially set all state’s utility to zero
    - Propagate true utility throughout map as we go
* Value iteration always converges to reveal true values
  + Bellman equation is a contraction function

Value Iteration and Inaccurate Utilities

* If our utilities are wrong, policy iteration still works

Exploration function f(u,n)

* Tradeoff between greed and curiosity
  + Greed = preference for large values of u
  + Curiosity = preference for low values of n
* F should be increasing in u and decreasing in n
  + R+ = optimistic estimate of best possible reward in any state (R+ max u)
  + Ne = threshold for “seen this action-state pair enough times”

Learning Action-Utility functions

* Active TD agent?
  + Stop fixing policy
  + Passive TD agent learns utilities -> need to learn model to choose actions
* Why not learn both utilities and model at the same time?
  + Q-function Q(s,a) = “utility” of choosing action a in state s
  + U(s) = maxaQ(s,a)
  + Q-function takes the place of learned utilities and transition probs

Q-function Q(s,a)

* Also obeys equilibrium constrains (similar to bellman)
* Given a model Pr[s’|s,a], we can solve this directly
  + Problem: requires a model (avoid)
* TD approach requires no model
  + Just nudge Q values in right direction

State-Action-Reward-State-Action(SARSA)

* Close relative of Q-learning
  + Q-learning uses the best action in next state
    - More flexible
  + SARSA uses actual action taken (ignores best action unless chosen)
    - Better if policy depends on other agents

Generalized RL

* So far, functions have been dictionaries (Q-learning, U(s))
* Doesn’t scale well
* Function approximation
  + Represent function f(x) as an approximation function g0(x)

Function Approximation

* Compress entire table into n parameters
  + Might not be able to be fully accurate, but good enough