Draft Notes for CS221 Artificial Intelligence

Moonsu Kang

July 2016

1 Machine Learning

1.1 Linear Regression

- $f_w(x)$ based on score $w \cdot \phi(x)$

1.2 Feature Extraction

Feature Template.

1. Sparsity in feature vectors: when we have sparsity (few nonzeros), it is typically more efficient to represent the feature vector as a map from strings to doubles
2. A hypothesis class is the set of possible predictors with a fixed $\phi(x)$ and varying w:
$$F = \{f_w : w \in R^d\}$$
3. Expressivity: are the features $\phi(x)$ are powerful enough to express predictors which are good?

1.3 Loss Minimization

A loss function $Loss(x, y, w)$ measures how unhappy you would be if you used w to make a prediction on x when the correct output is y.

- **Optimization Problem**: Loss Minimization
$$\min_{w \in R^d} TrainLoss(w)$$
- **Optimization Algorithm**: Gradient Descent
$$w \leftarrow w - \eta \nabla_w TrainLoss(w)$$

1.4 Stochastic Gradient Search

Each iteration of gradient descent requires going over all training examples which can be with lots of data. . Rather than looping through all the training examples to compute a single gradient and making one step, SGD loops through the examples (x, y) and updates the weights w based on each example.

1.5 Classification Prediction

1.5.1 Support Vector Machines

$$Loss_{hinge}(x, y, w) = \max\{1 - (w \cdot \phi(x))y, 0\}$$

1.5.2 Logistic Regression

$$Loss_{logistic}(x, y, w) = log(1 + e^{-(w \cdot \phi(x))y})$$

1.5.3 Generalization

1. Approximation and estimation error
2. Approximation: how far the entire hypothesis class is from the target predictor f
3. Estimation: s how good the predictor ˆf returned by the learning algorithm is with respect to the best in the hypothesis class: $Err(\hat{f}) - Err(g)$
4. Controlling the dimensionality
   - Manual Selection
   - Regularization
   - Early Stopping
5. Validation set for training hyperparameters
6. A validation (development) set is taken out of the training data which acts as a surrogate for the test set

1.5.4 Supervised/Unsupervised Learning

1. Supervised:
   - Prediction: Dtrain contains input-output pairs (x, y)
   - Expensive
2. Unsupervised:
   - Clustering: Dtrain only contains inputs x
   - Cheaper to obtain

- Types of learning: Clustering, Dimensionality reduction (PCA)

3. K-means

   - a particular method for performing clustering which is based on associating each cluster with a centroid μ_k for $k = 1, \ldots, K$.

   $$Losskmeans(z, \mu) = \sum_{i=1}^{n} \|\phi(x_i) - \mu_{z_i}\|^2$$

   $$\min_z \min_\mu Losskmeans(z, \mu)$$

   - Repeat Step 1: set assignments z given μ and Step 2: set centroids μ given z
   - guaranteed to converge to a local minimum, but is not guaranteed to find the global minimum

1.6 Neural Networks

1.6.1 logistic function

The logistic function maps $(-\infty, \infty)$ to [0,1]:

$$\sigma(z) = (1 + e^{-z})^{-1}$$

1. Hidden Layer: activation value

   $$h_j = \sigma(v_j \cdot \phi(x))$$

2. Activation function:

   - Sigmoid
   - Rectified Linear

3. Learning: Forward and Backward Propagation

1.6.2 Nearest Neighbors

1. Non-parametric: the complexity of the decision boundary adapts to the number of training examples

2 Search

search problems define the possibilities, and search algorithms explore these possibilities. search problems is our first instance of a state-based model.In a search problem, in a sense, we are still building a predictor f which takes an input x, but f will now return an entire action sequence, not just a single action.

2.1 Search Problem

Search Tree is composed of:

- S_{start}:Root of tree
- Actions(s): possible action, a, that could be performed in state s
- Cost(s,a): cost associated with action a at state s
- Succ(s,a)
- IsGoal(s)

2.2 Backtracking search

```
def BacktrackingSearch(s, path):
    If IsGoal(s): update minimum cost path
    For each action $a \in Actions(s)$:
        Extend path with Succ(s, a) and Cost(s, a)
    Call BacktrackingSearch(Succ(s, a), path)
```

The algorithm is called recursively on the current state s and the path path leading up to that state. If we have reached a goal, then we can update the minimum cost path with the current path. Otherwise, we consider all possible actions a from state s, and recursively search each of the possibilities.

- Memory: $O(D)$
- Time: $O(b^D)$

2.3 Depth-first search

modify backtracking search to not keep track of costs and then stop searching as soon as we reach a goal. Assume action costs $Cost(s, a) = 0$.

- Memory: $O(D)$
- Time: $O(b^D)$, but could be much better

2.4 Breadth-first search

explore all nodes in order of increasing depth. Assume action costs $Cost(s, a) = c$ for some $c \geq 0$.

- Memory: $O(b^d)$
- Time: $O(b^d)$ (depends on d, not D) where $d \leq D$

2.5 DFS with iterative deepening

The idea is to modify DFS to make it stop after reaching a certain depth. Therefore, we can invoke this modified DFS to find whether a valid path exists with at most d edges, which as discussed earlier takes O(d) space and $O(b^d)$ time.

- Memory: $O(b^d)$
- Time: $O(b^d)$ (depends on d, not D) where $d \leq D$

2.6 Dynamic Programming

Minimum cost path from state s to a goal state:

$$FutureCost(s) = \min_{a \in Actions(s)} [Cost(s, a) + FutureCost(Succ(s, a))$$

```
def DynamicProgramming(s):
    If already computed for s, return cached answer.
    If IsGoal(s): return solution
    For each action $a \in Actions(s)$: ...
```

- **Memoization**: The dynamic programming algorithm is exactly backtracking search with one twist. At the beginning of the function, we check to see if we've already computed the future cost for s. If we have, then we simply return it (which takes constant time, using a hash map)
- Assumption: state graph defined by Actions(s) and Succ(s, a) is **acyclic**. If there are cycles, the computation of a future cost for s might depend on s' which might depend on s
- **state** is a summary of all the past actions sufficient to choose future actions optimally. What state is really about is forgetting the past. Since current states includes information about the past, no need to remember all the past!

2.7 Uniform Cost Search

- UCS enumerates states in order of increasing past cost
- Assume All action costs are non-negative: $Cost(s, a) \geq 0$
- Strategy: maintain three sets of nodes: explored, frontier, and unexplored

```
Add $s_{start}$ to frontier (priority queue)
Repeat until frontier is empty:
    Remove s with smallest priority p from frontier
    If IsGoal(s): return solution
    Add s to explored
    For each action $a \in Actions(s)$:
        Get successor $s' \leftarrow Succ(s,a)$
    If $s'$ already in explored: continue
    Update frontier with $s'$ and priority p + Cost(s, a)
```

DP can handle negative action costs, but is restricted to acyclic graphs. It also explores all N reachable states from s_{start}, which is inefficient. This is unavoidable due to negative action costs. UCS can handle cyclic graphs, but is restricted to non-negative action costs. An advantage is that it only needs to explore n states, where n is the number of states which are cheaper to get to than any goal state.

In summary, Tree search algorithms are the simplest: just try exploring all possible states and actions. With backtracking search and DFS with iterative deepening, we can scale up to huge state spaces since the memory usage only depends on the number of actions in the solution path. Of course, these algorithms necessarily take exponential time in the worst case.
To solve expotential time, bookkeeping technique is neccessary; idea of state is vital as it contains all information from the past to act optimally in future. With an appropriately defined state, we can apply either dynamic programming or UCS, which have complementary strengths. The former handles negative action costs and the latter handles cycles. Both require space proportional to the number of states; but certainly lowers the time to $O(N)$ or $O(n \log n)$

3 Markov decision processes

MDP is a graph with states, chance nodes, transition probabilities, and rewards. A Markov decision process has a set of states States, a starting state s_{start}, and the set of actions Actions(s) from each state s. It also has a transition distribution T, which specifies for each state s and action a, a distribution over possible successor states s'. Specifically, we have that $\sum_{s'} T(s, a, s') = 1$. Finally, the discount factor γ is a quantity which specifies how much we value the future. Take Randomness as part of consideration in taking actions at states. The main difference, is the move from a deterministic successor function Succ(s, a) to transition probabilities over s'. We can think of the successor function Succ(s, a) as a special case of transition probabilities: where probability of the action is only 1 or 0.

3.1 Policy Evaluation

$$(MDP, \pi) \rightarrow V_\pi$$

A policy π is a mapping from each state $s \in States$ to an action $a \in Actions(s)$. policy specifies an action for every single state, not just the states along a path;

we know what action to take no matter where we are. Following a policy yields a random path.
The utility of a policy is the (discounted) sum of the rewards on the path (this is a random quantity); find a policy that maximizes the expected utility.
Let $V_{\pi(s)}$ be the expected utility received by following policy π from state s. Let $Q_{\pi(s,a)}$ be the expected utility of taking action a from states, and then following policy π.

$$V_\pi(s) = \begin{cases} 0 & \text{if } IsEnd(s) \\ Q_{\pi(s,a)} & \text{otherwise} \end{cases}$$

$$Q_{\pi(s,a)} = \sum_{s'} T(s,a,s')[Reward(s,a,s') + \gamma V_\pi(s')]$$

Policy iteration starts with a vector of all zeros for the initial values $V_\pi(0)$. Each iteration, we loop over all the states and apply the two recurrences that we had. The complexity of iteration is $O(t_{PE}SS')$

```
Init $V_\pi^{(0)}(s) \leftarrow 0$ for all states s
For $t = 1, \ldots, t_{PE}$:
    For each state s:
        $V_\pi^{(t)}(s) \leftarrow Q^{(t-1)}(s,\pi(s)) = \sum_{s'} T(s,\pi(s),s')[Reward(s,\pi(s),s') + \gamma V_\pi(s')]$
```

Repeat until values don't change much:

$$\max_{s \in States} |V_\pi^{(t)}(s) - V_\pi^{(t-1)}(s)| \leq \epsilon$$

3.2 Policy Improvement

$$(MDP, V_\pi) \rightarrow \pi_{new}$$

```
Input: value of policy $V_\pi$
Output: new policty $\pi_{new}$
1. Compute $Q_\pi(s,a)$ from $V_\pi(s)$ for each a
2. Compute $\pi_{new}(s) = arg\max_{a \in Actions(s)} Q_\pi(s,a)$
```

Complexity: step 1 takes O(SAS'); step 2 O(SA)

3.3 Policy Iteration

$$(MDP) \rightarrow V_{opt}, \pi_{opt}$$

Policy Iteration

- Policy Evaluation + Policy Improvement
- 1) from any π, evaluate value V_π
- 2) from any V_π, generate π_{new}

- Repeatedly compute the value of policy and improve
- Convergence to global optimal value
- Two loops: inner one for policy evaluation; outer for policy iteration
- Complexity: $O(t_{PI}(t_{PE}SS' + SAS'))$
- Warm start: to initialize inner loop with best guess to speed up

3.4 Value Iteration

$$(MDP) \rightarrow V_{opt}, \pi_{opt}$$

Value Interation V_{opt}, Q_{opt}

$$Q_{opt}(s,a) = \sum_{s'} T(s,a,s')[Reward(s,a,s') + \gamma V_{opt}(s')]$$

$$V_{opt}(s) = \begin{cases} 0 & \text{if } IsEnd(s) \\ \max_{a \in Actions(s)} Q_{opt(s,a)} & \text{otherwise} \end{cases}$$

Value iteration can be interpreted as policy iteration where we only do one iteration of policy evaluation in the inner loop. The intuition is that there's no need to compute the value of a policy exactly when we're just going to change the policy anyway; we just need a rough estimate. So, cost of Value iteration is: $O(t_{VI}SAS')$ Value iteration is also guaranteed to converge to the optimal value.

3.5 Convergence

Suppose either $\gamma < 1$ or MDP is acyclic, value iteration and policy iteration both converge to the correct answer. With probability $\gamma < 1$, the MDP terminates at any state.

4 Reinforcement learning

MDP was able to find optimal policy given known transitions and rewards. An important distinction between solving MDPs and reinforcement learning is that the former is offline and the latter is online. In reinforcement learning, you don't know how the world works, but you only have one life, so you just have to go out into the real world and learn how it works from experiencing it and trying to take actions that yield high rewards. How to choose actions and to update parameters based on exploration of actions is the key for Reinforcement Learning algorithm.

4.1 Monte Carlo

model-based Monte Carlo uses Monte Carlo simulation to estimate the model (transitions and rewards).

$$\hat{T}(s,a,s') = \frac{\text{\# of (s,a,s') occurrence}}{\text{\# of (s,a) occurrence}}$$

$$\hat{Reward}(s,a,s') = \text{avg of r in } (s,a,r,s')$$

Exploration to different states that we wouldn't have got to based on deterministic $\pi(s)$. Non-deterministic policy allows to explore states and actions and then the estimates of transitions and rewards will converge over time. We can plug in the estimates into MDP and solve for a policy.

$$\hat{Q}_{opt}(s,a) = \sum_{s'} \hat{T}(s,a,s')[\hat{Reward}(s,a,s') + \gamma \hat{V}_{opt}(s')]$$

4.1.1 Model-free Monte Carlo

$$\hat{Q}_\pi(s,a) = \text{avg of } \mu_t \text{where } s_{t-1} = s, a_t = a$$

Instead of estimating for Rewards and Transitions, estimate Q directly. Model-free Monte Carlo depends strongly on the policy π that is followed; after all it's computing Q_π. Because the value being computed is dependent on the policy used to generate the data, we call this an on-policy algorithm. In contrast, model-based Monte Carlo is off-policy, because the model we estimated did not depend on the exact policy (as long as it was able to explore all (s, a) pairs).

4.1.2 Model-free MC: Interpolation

Instead of thinking of averaging as a batch operation that takes a list of numbers (realizations of μ_t) and computes the mean, we can view it as an iterative procedure for building the mean as new numbers are coming in. The interpolation ratio η is set carefully so that u contributes exactly the right amount to the average.

```
On each (s,a,$\mu$):
    $\eta = \frac{1}{1+(\#updatesto(s,a))}$
    $\hat{Q}_\pi(s,a) \leftarrow (1-\eta)\hat{Q}_\pi(s,a) + \eta\mu$
```

4.1.3 Model-free MC: Stochastic gradient update

```
On each (s,a,$\mu$):
    $\hat{Q}_\pi(s,a) \leftarrow \hat{Q}_\pi(s,a) - \eta(\hat{Q}_\pi(s,a) - \mu)$
```

The second equivalent formulation is making the update look like a stochastic gradient update. the model-free Monte Carlo is just performing stochastic gradient descent on a least squares regression problem, where the weight vector is $\hat{Q}_\pi$.

4.2 Bootstrapping

4.2.1 SARSA

Broadly speaking, reinforcement learning algorithms interpolate between new data (which specifies the target value) and the old estimate of the value (the prediction). It is taken purely from the data. Since μ is based on one path and could have large variance, there is a need to wait until the end.
An alternative to model-free Monte Carlo is SARSA, whose target is $r+\gamma\hat{Q}_\pi(s,a)$. SARSA's target is a combination of the data (the first step) and the estimate (for the rest of the steps).

```
On each (s,a,r,s',a'):
    $\hat{Q}_\pi(s,a) \leftarrow (1-\eta)\hat{Q}_\pi(s,a) + \eta[r + \gamma\hat{Q}_\pi(s',a')]$
```

- If estimates are good, SARSA is more reliable
- It is done faster while for MC we have to wait til the end of episode to update Q-value
- In advanced, interpolate between model-free MC and SARSA; where λ determines the relative weighting between the two
- Similar approach can be taken for V_π instead of Q_π

4.3 Q-learning

Q-learning is an off-policy algorithm that estimate Q_{opt} in model-free. (model-free MC and SARSA estimates Q_π)

```
On each (s,a,r,s'):
    $\hat{Q}_{opt}(s,a) \leftarrow (1-\eta)\hat{Q}_{opt}(s,a) + \eta[r + \gamma\hat{V}_{opt}(s')]$
    where $\hat{V}_{opt}(s') = \max_{a' \in Actions(s')} \hat{Q}_{opt}(s',a')$
```

4.4 Exploration and Generalization

Determining how to choose an action in exploration is as important as algorithms for updating parameters. Two complementary ways of getting determining exploration policy:
(i) explicitly explore (s, a)
(ii) explore (s, a) implicitly by actually exploring (s',a') with similar features and generalizing. **Epsilon-greedy**

1. Exploration/exploitation trade-off

2. Epsilon-greedy algorithm, decrease ϵ over time

$$\pi_{act}(s) = \begin{cases} \text{argmax}_{a \in Actions} \hat{Q}_{opt}(s,a) & \text{prob } 1-\epsilon, \\ \text{random from Actions(s)} & \text{prob } \epsilon \end{cases}$$

Function Approximation

1. So far, Q-learning memorized $\hat{Q}_{opt}(s,a)$ every time; no generalization
2. Define features $\phi(s,a)$ and weights w:

$$\hat{Q}_{opt}(s,a;w) = w\dot{\phi}(s,a)$$

3. features are supposed to be properties of the state-action (s,a) pair that are indicative of the quality of taking action a in state s. The ramification is that all the states that have similar features will have similar Q-values

```
On each (s,a,r,s'):
    $w \leftarrow w - \eta[\hat{Q}_{opt}(s,a;w) - (r + \gamma\hat{V}_{opt}(s'))]\phi(s,a)$

The objective function is:
    $\min_w \sum_{(s,a,r,s')}(\hat{Q}_{opt}(s,a;w) - (r + \gamma\hat{V}_{opt}(s')))^2$
```

4. Deep reinforcement learning: Use neural network for $\hat{Q}_{opt}$

4.5 Challenges with RL

- Delayed rewards: RL requires the modeling of state; rewards comes across time and after actions taken; temporal depth
- Partial Information; necessary to explore
- RL currently only works for small problems or where there is a lot of prior knowledge / constraints

5 Adversial Games

5.1 Games, expectimax

Two-player zero-sum games Its Components are: s_{start}, Actions(s), Succ(s,a), IsEnd(s), Utility(s), Player(s). Main difference is that each state has a designated Player(s), and a player p only gets to choose the action for the states s such that Player(s) = p. Also all the utility, Utility(s), is at the end state
The probability of player p choosing action a in state s is defined as $\pi = (\pi_{agent}, \pi_{opp})$. and define $V[\pi_{agent}, \pi_{opp}] = V_\pi(s_{start})$.

$$V_\pi(s) = \begin{cases} Utility(s) & \text{IsEnd(s)} \\ \sum_{a \in Actions(s)} \pi_{agent}(s,a) V_\pi(Succ(s,a)) & \text{Player(s)=agent} \\ \sum_{a \in Actions(s)} \pi_{opp}(s,a) V_\pi(Succ(s,a)) & \text{Player(s)=opp} \end{cases}$$

Expectimax $V_{opt,\pi}(s)$ is the expected utility if the agent follows the best policy assuming the opponent follows π_{opp}; maximize over the agent's actions rather than following a fixed agent policy (which we don't know now).

$$V_{opt,\pi}(s) = \max_{a \in Actions(s)} V_{opt,\pi}(Succ(s,a))$$

5.2 Minimax, expectiminimax

Minimax The value $V_{opt}(s)$ is the utility if the agent follows the best policy and the opponent follows the adversarial policy.

$$V_{\pi}(s) = \begin{cases} Utility(s) & \text{IsEnd(s)} \\ \max_{a \in Actions(s)} V_{opt}(Succ(s,a)) & \text{Player(s)=agent} \\ \min_{a \in Actions(s)} V_{opt}(Succ(s,a)) & \text{Player(s)=opp} \end{cases}$$

- Property1 (best against adversarial opponent):

$$V[\pi^*_{agent}, \pi^*_{opp}] \geq V[\pi_{agent}, \pi^*_{opp}]\text{for all } \pi_{agent}$$

- Property2 (lower bound against any opponent):

$$V[\pi^*_{agent}, \pi^*_{opp}] \leq V[\pi^*_{agent}, \pi_{opp}]\text{for all } \pi_{opp}$$

- Property3 (not necessarily best):

$$V[\pi^*_{agent}, \pi^*_{opp}] \ngeq V[\pi_{agent}, \pi_{opp}]$$

Expectiminimax The value Vopt(s) is the utility if the agent follows the best policy, the opponent follows the adversarial policy, the coin follows a fixed policy. Players = agent, opp, coin. Coin for any nature of randomness.

$$V_{\pi}(s) = \begin{cases} Utility(s) & \text{IsEnd(s)} \\ \sum_a \pi_{coin}(s,a) V_{opt}(Succ(s,a)) & \text{Player(s)=coin} \\ \max_{a \in Actions(s)} V_{opt}(Succ(s,a)) & \text{Player(s)=agent} \\ \min_{a \in Actions(s)} V_{opt}(Succ(s,a)) & \text{Player(s)=opp} \end{cases}$$

- Complexity: $O(d)$ space, $O(b^{2d})$ time
- To speed up:
- Evaluation Functions: use domain-specific knowledge, compute approximate answer
- Alpha-beta pruning: general-purpose, compute exact answer

5.3 Evaluation Functions

The first idea on how to speed up minimax is to search only the tip of the game tree, that is down to depth d_{max}, which is much smaller than the total depth of the tree D. If $d = 0$, then we don't do any more search, but fall back to an evaluation function Eval(s), which is supposed to approximate the value of $V_{opt}(s)$.

$$V_\pi(s) = \begin{cases} Utility(s) & \text{IsEnd(s)} \\ Eval(s) & \text{d=0} \\ \max_{a \in Actions(s)} V_{opt}(Succ(s,a), d) & \text{Player(s)=agent} \\ \min_{a \in Actions(s)} V_{opt}(Succ(s,a), d-1) & \text{Player(s)=opp} \end{cases}$$

Function Approximations In order for Eval(s) to estimate V_{opt}, domain knowledge about game is necessary. While we don't know who's going to win, there are some features of the game that are likely indicators.

$$Eval(s; w) = w \cdot \phi(s)$$

Two approximations to $V_{opt}(s) = V[\pi^*_{agent}, \pi^*_{opp}]$

1. Replace optimal policies with heuristic, random policies
   - Replace $\pi^*_{agent}, \pi^*_{opp}$ with random π_{agent}, π_{opp}
2. Monte Carlo approximation
   - Simulate n random paths by applying the policies
   - Average the utilities of the n paths and call that as $\hat{V}[\pi_{agent}, \pi_{opp}]$
   - as n (sample) increases, variance to true value lowers

5.3.1 Alpha-beta pruning

- Branch and bound principle
- keep lower and upper bounds on values
- If intervals of two branches do not overlap, choose preferred one and compute its precise value
- The optimal path is path that minimax policies take. Values of all nodes on path are the same.
- Prune a node if its interval doesn't have non-trivial overlap with every ancestor
- store $\alpha_s = \max_{s' \leq s} a_s$ and $\beta_s = \min_{s' \leq s} b_s$
- Best ordering: $O(b^{2 \cdot 0.5d})$ time
- In practice, use a heuristic ordering based on a simple evaluation function: Intuitively, search for children that are going to give us the largest lower bound for max nodes and the smallest upper bound for min nodes

5.4 Temporal difference (TD) learning

TD learning combines these two ideas, using Monte Carlo simulations to generate data, which can be used to fit w. Note that as in Q-learning, simulation and learning will be heavily intertwined. **Function Approximation**:

$$Eval(s) = V(s; w)$$

Monte Carlo simulation:

$$Eval(s) = \hat{V}[\pi_{agent}, \pi_{opp}]$$

Determine (i) model family (in this case, V (s; w)), (ii) where the data comes from, and (iii) the actual learning algorithm

1. (i) $V(s; w) = w \cdot \phi(s)$; or $= \sum_j w_j \sigma(v_j \cdot \phi(s))$
2. (ii) generate a sequence of states, actions, and rewards by simulation
   - s0; a1, r1, s1; a2, r2,$\ldots, a)n, r_n, s_n$ based on π_{agent}, π_{opp}
3. (iii) Figure out the prediction and the target based on (s, a, r, s')
   - The prediction is just the value of the current function at the current state s, $V(s; w)$
   - The target uses the data by looking at the immediate reward r plus the value of the function applied to to the successor state s, $r + \gamma V(s'; w)$
   - Objective: $\frac{1}{2}(\text{prediction(w)} - \text{target})^2$
   - $w \leftarrow w - \eta(\text{prediction(w)} - \text{target})\nabla_w \text{prediction(w)}$

TD Learning

$$w \leftarrow w - \eta(V(s; w) - (r + \gamma V(s'; w)))\nabla_w V(s; w)$$

- Similar to Q-learning: learn from the same data and are based on gradient-based weight updates
- Q-Learning
  - learns on Q-value: measures how good an action is to take in a state
  - Off-policy Algorithm; compute Q_{opt}, associated with the optimal policy
  - No need to know MDP transitions $T(s, a, s')$
- TD learning
  - learns on Value function: measures how good it is to be in a state

  - on-policy; compute V_π, the value associated with a fixed policy π. The action a does not show up in TD updates because a is given by fixed policy π. In attempt to optimize policy, generally set π to be the current guess of optimal policy $\pi(s) = \text{argmax}_{a \in Actions(s)} V(Succ(s,a); w)$
  - Need to know rules of the game, Succ(s,a). Need to know what effect our actions will have on transition. However, in the game playing setting, we do know the transitions (the rules of the game)

6 Simultaneous Games

Game Evaluation: The value of the game if player A follows π_A and player B follows pi_B is

$$V(\pi_A, \pi_B) = \sum_{a,b} \pi_A(a)\pi_B(b)V(a,b)$$

Pure Strategy: going second is no worse

$$\max_a \min_b V(a,b) \leq \min_b \max_a V(a,b)$$

For any fixed mixed strategy π_A, $\min_{\pi_B} V(\pi_A, \pi_B)$ can be attained by a pure strategy.

Mixed Strategy: Minimax Theorem For every simultaneous two-player zero-sum game with a finite number actions:

$$\max_{\pi_A} \min_{\pi_B} V(\pi_A, \pi_B) = \min_{\pi_B} \max_{\pi_A} V(\pi_A, \pi_B)$$

The theorem states that for any simultaneous two-player zero-sum game with a finite set of actions (like the ones we've been considering), we can just swap the min and the max: it doesn't matter which player reveals his/her strategy first, as long as their strategy is optimal.

6.1 Non-zero-sum Games

A **Nash equilibrium** is $(\pi_A^*.\pi_B^*)$ such that no player has an incentive to change his/her strategy:

$$V_A(\pi_A^*, \pi_B^*) \geq V_A(\pi'_A \pi_B^*) \text{ for all } \pi_A$$
$$V_B(\pi_A^*, \pi_B^*) \geq V_B(\pi_A^*, \pi_B^*) \text{ for all } \pi_B$$

A Nash equilibrium is kind of a state point, where no player has an incentive to change his/her policy unilaterally.
Nash's existence theorem is that in any finite-player game with finite number of actions, there exists at least one Nash equilibrium.

7 Constraint Satisfaction Problem

Variable-based models is an umbrella term that includes constraint satisfaction problems (CSPs), Markov networks, Bayesian networks, hidden Markov models (HMMs), conditional random fields (CRFs). It is the idea of thinking about solutions to problems as assignments of values to variables (this is the modeling part). All the details about how to find the assignment (in particular, which variables to try first) are delegated to the algorithms. So the advantage of using variable-based models over state-based models is that it's making the algorithms do more of the work, freeing up more time for modeling.

By moving to a higher language, you might forgo some amount of ability to optimize manually, but the advantage is that (i) you can think at a higher level and (ii) there are more opportunities for optimizing automatically.

7.1 Factor Graph

Factor Graph consists of a set of variables and a set of factors: (i) n variables $X_1, \dots, X_n$; and (ii) m factors (also known as potentials) $f_1, \dots, f_m$. Each factor f_j is a function that takes an assignment x to all the variables and returns a non-negative number representing how good that assignment is. The key aspect that makes factor graphs useful is that each factor f_j only depends on a subset of variables, called the scope. The arity of the factors is generally small (think 1 or 2).
Scope of a factor f_j : set of variables it depends on. Arity of f_j is the number of variables in the scope. Unary factors (arity 1); Binary factors (arity 2).

Each assignment $x = (x_1, \dots, x_n)$ has a weight:

$$Weight(x) = \Pi_{j=1}^{m} f_j(x)$$

A CSP is a factor graph where all factors are constraints: $f_j(x) \in 0, 1 forall j = 1, \dots, m$ The constraint is satisfied iff $f_j(x) = 1$. An assignment x is consistent iff Weight(x) = 1 (i.e., all constraints are satisfied)

7.2 Dynamic Ordering

Partial Assignment ; Let D(x, Xi) be set of factors depending on Xi but not on unassigned variables.

Backtracking search: a recursive procedure that takes in a partial assignment x, its weight w, and the domains of all the variables $Domains = (Domain_1, ..., Domain_n)$

```
Backtrack(x, w, Domains):
If x is complete assignment: update best and return
```

```
Choose unassigned VARIABLE $X_i$
Order VALUES $Domain_i$ of chosen $X_i$
For each value v in that order:
$\delta \leftarrow Y f_j \in D(x, X_i) f_j(x \cup Xi : v)$
If $\delta = 0$: continue
Domains' ← Domains via LOOKAHEAD
Backtrack($x \cup Xi : v, w\delta, Domains'$)
```

Dynamic Ordering:

1. Most constrained variable (MCV): to fail early
2. Least constrained value (LCV): to leave as many options

Forward checking

1. After assigning a variable X_i , eliminate inconsistent values from the domains of X_i's neighbors
2. If any domain becomes empty, don't recurse
3. When unassign X_i , restore neighbors' domains

Arc Consistency:
A variable X_i is arc consistent with respect to X_j if for each $x_i \in Domain_i$, there exists $x_j \in Domain_j$ such that $f(X_i : x_i, X_j : x_j) \neq 0$ for all factors f whose scope contains X_i and X_j

EnforceArcConsistency(Xi , Xj): Remove values from $Domain_i$ to make X_i arc consistent with respect to X_j.

AC-3: enforce arc consistency recursively The idea behind enforcing arc consistency is to look at all the factors that involve just two variables Xi and Xj and rule out any values in the domain of Xi which are obviously bad without even looking at another variables.

Turn N-ary constraints into binary constraint by setting auxiliary variable relation extraction: the task of building systems that can process the enormous amount of unstructured text on the web, and populate a structured knowledge base

7.3 Beam Search and Local Search: Approximate Algorithm

Dynamic ordering and look ahead are only helpful when there are hard constraints, which allow us to prune away values in the domains of variables which definitely are not compatible. At worst there can be no pruning, and we face exponential time.

7.3.1 Beam Search

Graph Operations: conditioning and elimination

1. **Greedy Search**

```
Partial assignment $x \leftarrow \{\}$
For each $i = 1, \ldots, n$:
    Extend:
        Compute weight of each $x_v = x \cup \{X_i : v\}$
    Prune:
        $x \leftarrow x_v$ with highest weight
```

2. Beam Search

   - keep $\leq K$ candidate list C of partial assignments

```
Initialize $C \leftarrow [\{\}]$
For each $i = 1, \ldots, n$:
    Extend:
        $C' \leftarrow \{x \cup \{Xi : v\} : x \in C, v \in Domain_i$
    Prune:
        $C \leftarrow K$ elements of $C'$ with highest weights
```

   - Running time: $O(n(Kb)log(Kb))$ with branching factor $b = |Domain|$, beam size K
   - Beam size K controls tradeoff between efficiency and accuracy (When $K = 1$, it is greedy search with $O(nb)$; When K is large, it is BFS search)

7.3.2 Local Search

Local search

- modifies complete assignments and makes repairs by changing one variable at a time
- When evaluating possible re-assignments to Xi, only need to consider the factors that Xi depends on.
- Iterated conditional modes (ICM)

```
Initialize x to a random complete assignment
Loop through $i = 1, \ldots, n$ until convergence:
Compute weight of $x_v = x \cup X_i : v$ for each v
$x \leftarrow x_v$ with highest weight
```

- Converges in a finite number of iterations; Can get stuck in local optima
- Gibbs Sampling: 'Choose $x \leftarrow x_v$ with probability prop to its weight

7.4 Conditioning

- disconnect the graph; remove edges from X_i to dependent factors
- To condition on a variable $X_i = v$, consider all factors $f_1, \ldots, f_k$ that depend on Xi. Remove Xi and $f_1, \ldots, f_k$. Add $g_j(x) = f_j(x \cup X_i : v)$ for $j = 1, \ldots, k$.
- A and B are conditionally independent given C if conditioning on C produces a graph in which A and B are independent
- Markov Blanket: MarkovBlanket(A) be the neighbors of A that are not in A; condition on to make A conditionally independent
- Conditioning breaks large problem into smaller pieces; Smaller pieces can be solved in parallel

7.5 Elimination

Elimination removes variables from the graph, but actually chooses the best value for the eliminated variable Xi. compute the best one for all possible assignments to the Markov blanket of Xi.

$$\text{Add } f_new(x) = \max_{x_i} \Pi_{j=1}^{k} f_j(x)$$

- Solves a mini-problem over Xi conditioned on its Markov blanket
- $O(n \cdot |Domain|^{max-arity+1})$
- eliminate variables with the smallest Markov blanket
- The treewidth of a factor graph is the maximum arity of any factor created by variable elimination with the best variable ordering

8 Bayesian Networks

8.1 Bayesian Networks

Bayesian Networks factor graphs imbued with the language of probability. Let $X = (X_1, \ldots, X_n)$ be random variables. A Bayesian network is a directed acyclic graph (DAG) that specifies a joint distribution over X as a product of local conditional distributions, one for each node:

$$P(X_1 = x_1, \ldots, X_n = x_n) = \Pi_{i=1}^{n} p(x_i | x_{parents(i)})$$

- A Bayesian network specifies two parts: (i) a graph structure which governs the qualitative relationship between the variables, and (ii) local conditional distributions, which specify the quantitative relationship.

- Property: all the local conditional distributions, being distributions, sum to 1 over the first argument

$$\sum_{x_i} p(x_i|x_{parents(i)}) = 1 \text{ for each } x_{parents(i)}$$

- This implies: (i) Consistency of sub-Bayesian networks
  - Marginalization of a leaf node yields a Bayesian network without the node
- (ii) Consistency of conditional distributions
  - Local conditional distributions (factors) are the true conditional distributions
  - $P(D = d|A = a, B = b) = p(d|a, b)$
- Bayes rule

$$P(H = h|E = e) \propto P(H = h)P(E = e|H = h)$$

- explaining away: Suppose two causes positively influence an effect. Conditioned on the effect, conditioning on one cause reduces the probability of the other cause.
- A probabilistic program is a randomized program that invokes a random number generator to make random choices.
- Markov model: a chain-structured Bayesian network

$$X_i \approx p(X_i|X_{i-1})$$

$$p(x_i|x_{i-1}) = a \cdot [x_i = x_{i-1} + (1, 0)] + (1 - a) \cdot [x_i = x_{i-1} + (0, 1)]$$

- Hidden Markov Model, Factorial HMM
- Latent Dirichlet Allocation

```
Generate a distribution over topics $a \in R^K$
For each position $i = 1, \ldots, L$:
Generate a topic $Z_i \approx p(Z_i|a)$
Generate a word $W_i \approx p(W_i|Z_i)$
```

8.1.1 Independence

- Factor graph independence: Let A and B be two subsets of X. We have $A \perp\!\!\!\perp B$ iff there is no path from A and B.
- Probabilistic independence: $P(A = a, B = b) = P(A = a)P(B = b)$ for all a, b
- factor graph independence implies probabilistic independence. However, the converse is not true.

$$factorgraphindependence \rightarrow Probabilisticindependence$$

- Structural Independence is a property of factor graph structures
- A factor graph structure defines a family of different probability distributions:
$$P(X = x) \propto Weight(x) = \Pi_{j=1}^{m} f_j(x)$$

8.1.2 Bayesian network independence

Given a Bayesian network structure, we say $A \perp\!\!\!\perp B$ if after marginalizing all descendants of A and B, the resulting factor graph has $A \perp\!\!\!\perp B$.
Marginalize out as many of the irrelevant variables as possible, which includes all descendants of A and B. Then take factor graph independence of the rest. Factor Graph independence implies Bayesian network independence, not the other way.

V-structure:

- Parents B and E are conditionally dependent (condition on A).When we condition on A, that only removes A from the picture, but B and E still have a factor in common
- If two variables A, B are Bayesian network conditionally independent given C, then they are said to be d-separated
- eliminating children renders parents independent

8.2 Probabilistic Inference

In order to find $P(Q|E = e)$ based on $P(X_1 = x_1, \ldots, X_n = x_n)$

- Remove (marginalize) variables not ancestors of Q or E
- Convert Bayesian network to factor graph
- Condition (shade nodes / disconnect) on E = e
- Remove (marginalize) nodes disconnected from Q
- Run probabilistic inference algorithm (manual, variable elimination, Gibbs sampling, particle filtering)

8.2.1 Forward-backward

- HMM: $P(H = h, E = e) = p(h_1)\Pi_{i=2}^{n} p(h_i|h_{i-1})\Pi_{i=1}^{n} p(e_i|h_i)$
- Two common type of query on HMM: filtering and smoothing
- Filtering asks for the distribution of some hidden variable Hi conditioned on only the evidence up until that point. $P(H_3|E_1 = e_1, E_2 = e_2, E_3 = e_3)$
- Smoothing asks for the distribution of some hidden variable Hi conditioned on all the evidence, including the future. This is useful when you have collected all the data and want to retroactively go and figure out what H_i was. $P(H_3|E_1 = e_1, E_2 = e_2, E_3 = e_3, E_4 = e_4)$
- Lattice representation: reduction from a variable-based model to a state-based model; contains $O(Kn)$ nodes and $O(K^2n)$ edges
- Forward: $F_i(h_i) = \sum_{h_{i-1}} F_{i-1}h_{i-1}w(h_{i-1}, h_i)$
- Backward: $B_i(h_i) = \sum_{h_{i+1}} B_{i+1}(h_{i+1})w(h_i, h_{i+1})$
- $S_i(h_i) = F_i(h_i)B_i(h_i)$; the sum of the weights over all paths from the start node to the end node that pass through
- Smoothing query: $P(H_i = h_i|E = e) \propto S_i(h_i)$; Running time: $O(LK^2)$

8.2.2 Gibbs Sampling

- Use of particles to represent a probability distribution; the probabilistic analog of local search methods such as ICM

```
Initialize x to a random complete assignment
Loop through $i = 1, \ldots, n$ until convergence:
Set $X_i = v$ with prob. $P(X_i = v|X_{-i} = x_{-i})$
(notation: $X_{-i} = X/X_i$)
```

8.2.3 Particle Filtering

- the probabilistic analog of partial search such as beam search Beam search does generate a set of particles, but there are two problems. First, it can be slow if $Domain_i$ is large. Second, we are greedily taking the top K candidates, which can be too myopic. Particle filtering addresses both of these problems. There are three steps:

1. Propose: extends the current parital assignment
   - For each old particle (x1, x2), sample $X3 \approx p(x3|x2)$
   - New particle: $P(X1, X2, X3|E_1 = 0, E_2 = 1)$
2. weight: redistributes resources on the particles based on evidence

    - For each old particle (x1, x2, x3), weight it by $w(x1, x2, x3) = p(e3|x3)$
    - $P(X1, X2, X3|E_1 = 0, E_2 = 1, E_3 = 1)$

3. Resample

    - Given a distribution $P(A = a)$ with n possible values, draw a sample K times

```
Initialize $C \leftarrow []$
For each $i = 1, \dots, n$:
Propose (extend):
$C' \leftarrow (x1 : i-1, xi) : x_{1:i-1} \in C, xi \approx p(x_i|x_{i-1})$
Reweight:
Compute weights $w(x_{1:i}) = p(e_i|x_i)$ for $x \in C'$
Resample (prune):
$C \leftarrow K$ elements drawn independently from $\propto w(x_{1:i})$
```

8.3 Learning

Where do all the local conditional distributions come from? These local conditional distributions are the parameters of the Bayesian network.
Learning: $D_{train} \rightarrow \theta$(local conditional probabilities)

8.3.1 Supervised Learning

- Data consists of a set of complete assignments
- Parameter sharing: The local conditional distributions of different variables use the same parameters

$$P(X_1 = x_1, \dots, X_n = x_n) = \Pi_{i=1}^{n} p_{d_i}(x_i|x_{parents(i)})$$

  where collection of distributions $\theta = p_d : d \in D$ and d_i could be same for multiple i

- Algorithm:

```
Count:
For each $x \in D_{train}$:
    For each variable $x_i$:
        Increment $count_{d_i}(x_{Parents(i)}, x_i)$
Normalize:
    For each d and local assignment $x_{Parents(i)}$:
        Set $p_d(x_i|x_{Parents(i)}) \propto count_d(x_{Parents(i)}, x_i)$
```

- This is equivalent to Maximum likelihood objective:

$$\max_{\theta} \Pi_{x \in D_{train}} P(X = x; \theta)$$

8.3.2 Laplace Smoothing:Regularization

For each distribution d and partial assignment $(x_{Parents(i)}, xi)$, add λ to $count_d(x_{Parents(i)}, x_i)$. Then normalize to get probability estimates.

$$P(H) = \frac{1+1}{1+2} = \frac{2}{3}$$

8.3.3 Unsupervised Learning

- Data consists of partial assignments
- Maximum marginal likelihood

$$\max_\theta \Pi_{e \in D_{train}} P(E = e; \theta) = \max_\theta \Pi_{e \in D_{train}} \sum_h P(H = h, E = e; \theta)$$

- Expectation Maximization (EM): tries to maximize marginal likelihood

```
E-step:
  Compute $q(h) = P(H = h|E = e; \theta)$ for each h
  (use any probabilistic inference algorithm)
  Create weighted points: (h, e) with weight q(h)
M-step:
  Compute maximum likelihood (just count and normalize) to get $\theta$
Repeat until convergence.
```

9 Logical Inference

think of learning as induction, where we need to generalize, and inference as deduction. In logic-based models, inference is on applying a set of rules; logical formulas and inference rules. How to handle uncertainty and fine tuning with data were problematic; as logic model is deterministic and rule-based. Its strength is in expressiveness.

- Logical language: language that captures declarative knowledge rather than concrete procedures and is better connected with natural language (e.g. propositional logic and first-order logic)
- Two goals of logic: i) Represent knowledge about the world; ii) Reason with that knowledge
- Three key ingredients of a logic: Syntax, Semantic, Inference rules
  1. Syntax: defines a set of valid formulas; what are valid expressions?
  2. Semantics: for each formula, specify a set of models (assignments/-configurations of the world); specify the meaning of a formula which is a set of configurations of the world in which the formula holds; what do these expressions mean?

3. Inference Rules: given f, what new formulas g can be added without changing semantics ($\frac{f}{g}$)?
4. Different syntax can have same semantics (just different way of expressions); Same syntax can have different semantics (different way of interpretations)

- Trade-off: in balance of expressivity and computational efficiency

9.1 Logical Languages

9.1.1 Propositional Logic

- **Syntax**: Propositional symbols (atomic formulas) and Logical connectives
- Logical connectives is composed of: Negation, Conjunction, Disjunction, Implication, Biconditional
- **Semantics**: A model w in propositional logic is an assignment of truth values to propositional symbols
- Interpretation function I(f,w) returns true or false depending on whether w, model, satisfies f, formula. (Is f true in w?)
  - In base case: $I(p, w) = w(p)$ where p is propositional
  - In recursive case: recursively compute the truth value of the parts, and then finally combines these truth values based on the connective
- Set of Models: let M(f) be a set of models w for which $I(f, w) = 1$; A formula compactly represents a set of models
- Knowledge base: a set of formulas representing their conjunction/intersection
$$M(KB) = \cap_{f \in KB} M(f)$$
  - Entailment: KB entails f iff $M(f) \supseteq M(KB)$; f added no information/constraints
  - Contradiction: KB contradicts f iff $M(KB) \cap M(f) = \emptyset$
  - Contingency: $\emptyset \subsetneq M(KB) \cap M(f) \subsetneq M(KB)$
- Tell operation: tell[f] results in indicating whether f is entailed, contradicted, or contingent to KB
- Ask operation: ask[f] results in yes,no, or uncertain status based on KB
- Digression: probabilistic generalization
$$P(f|KB) = \frac{\sum_{w \in M(KB \cup \{f\})} P(W = w)}{\sum_{w \in M(KB)} P(W = w)}$$

- Satisfiability: KB is satisfiable if $M(KB) \neq \emptyset$
- Checking satisfiability (SAT) in propositional logic is special case of solving CSPs; where we solve for a model given propositional symbol and formula
- Model Checking: checking satisfiability of a knowledge base
  1. DPLL (Backtracking search and pruning)
  2. WalkSat (randomized local search)
- **Inference Rules**: allow us to do reasoning on the formulas themselves without ever instantiating the models; powerful when lots of formulas and proportional symbols are involved
  - If $f_1, \ldots, f_k, g$ are formulas,then the following is an inference rule:

$$\frac{f_1, \ldots, f_k}{g}$$

  - Derivation: KB derives/proves $f(KB \vdash f)$ iff f eventually gets added to KB.

```
Input: set of inference rules Rules.
Repeat until no changes to KB:
Choose set of formulas $f_1, \ldots, f_k \in KB$.
If matching rule $\frac{f_1, \ldots, f_k}{g}$ exists:
Add g to KB
```

- Interpretation defines entailed/true formulas ($KB \models f$), while Inference rules derive formulas ($KB \vdash f$)
- Soundness: A set of inference rules Rules is sound if:

$$\{f : KB \vdash f\} \subseteq \{f : KB \models f\}$$

- Completeness: A set of inference rules Rules is complete if:

$$\{f : KB \vdash f\} \supseteq \{f : KB \models f\}$$

- Soundness: nothing but truth; Completeness: whole truth
- Definite clauses: if a conjunction of propositional symbols holds then q hold

$$(p_1 \wedge \ldots \wedge p_k) \rightarrow q$$

- Horn clauses: either a definite clause or goal clause
- Modus ponens:

$$\frac{p_1, \ldots, p_k, (p_1 \wedge \ldots \wedge p_k) \rightarrow q}{q}$$

- Derivation: a proof that the root formula was derived using inference rules. Each node is a formula derived by applying an inference rule with the children as premises

- **Modus ponens on Horn clauses**: Modus ponens is complete with respect to Horn clauses:
  • Suppose KB contains only Horn clauses and f is an entailed propositional symbol
  • Then applying modus ponens will derive f

9.1.2 Resolution

The idea behind resolution is that it takes two general clauses, where one of them has some propositional symbol p and the other clause has its negation $\neg p$, and simply takes the disjunction of the two clauses with p and $\neg p$ removed. Here, $f_1, \ldots, f_n, g_1, \ldots, g_m$ are all propositional symbols.

$$\frac{f_1 \vee \ldots \vee f_n \vee p, \neg p \vee g_1 \vee \ldots \vee g_m}{f_1 \vee \ldots \vee g_1}$$

- Conjunctive normal form: conjunction of clauses

- conversion to CNF: Every formula f in propositional logic can be converted into an equivalent CNF form f':

$$M(f) = M(f')$$

- Resolution algorithm:
  1. Convert all formulas into CNF
  2. Repeatedly apply resolution rule
  3. Return unsatisfiable iff derive false

- Time complexity: when each rule application adds clause with many propositional symbols, it takes exponential time

- To summarize, we can either content ourselves with the limited expressivity of Horn clauses and obtain an efficient inference procedure (via modus ponens) OR have the expressivity of full propositional logic (general clauses) via resolution with exponential time

9.1.3 First-order Logic

Limitations of propositional logic: it is clunky because it is missing i) Objects and relations, ii) Quantifiers and variables.

1. Syntax
   - Terms: Constant symbol, Variable, Function of terms

- Formulas: Atomic formulas, Connectives, **Quantifier**
- Quantifier: Universal quantification ($\forall$), Existential quantification ($\exists$)

2. Semantics

   - First-order logic: Model w maps **grounded atomic formulas** to truth values vs. propositional symbol
   - A model w maps: constant symbols to objects
   - maps predicate symbols to tuples of objects
   - Restrictions on model: i) Unique names assumption and ii) Domain closure (one-to-one relationship between constant symbols in syntax-land and objects in semantics-land)
   - Proportionalize: If a one-to-one mapping exists, then propositionalizing formulas, which basically unrolls all the quantifiers into explicit conjunctions and disjunctions, is possible
   - propositionalization is needed if we want to use model checking to do inference.

3. Inference Rule: compactly, implicitly

   - Definite clauses

   $$\forall x_1 \dots x_n (a_1 \wedge \dots \wedge a_k) \to b$$

   - **Substitution** θ is a mapping from variables to constant symbols or other variables. Subst$[\theta, f]$ returns the result of performing substitution θ on f.
   - **Unification** takes two formulas f and g and returns a substitution θ which is the most general unifier: $Unify[f, g] = \theta$ such that $Subst[\theta, f] = Subst[\theta, g]$ or fail if no θ exists
   - Modus ponens

   $$\frac{a'_1, \dots, a'_k \forall x_1 \dots \forall x_n (a_1 \wedge \dots \wedge a_k) \to b}{b'}$$

   and $Subst[\theta, b] = b'$
   - Complexity
   - Completeness: Modus ponens is complete for first-order logic with only Horn clauses.
   - Semi-decidability: First-order logic is semidecidable. (the number of atomic formulas can be infinite)
   - Resolution
     - Strategy is same: convert to CNF and apply resolution

  - Skolem functions (e.g., Y (x)) to represent existential quantified variables

    $$\frac{f_1 \vee \ldots \vee f_n \vee h_1, \neg h_2 \vee g_1 \vee \ldots \vee g_m}{subst[\theta, f_1 \vee \ldots \vee g_m]}$$

    where $\theta = Unif[h_1, h_2]$

4. The main idea in first-order logic is the use of variables. First-order logic variables denote objects. Variables in first-order logic in essence allows us to effectively express large complex propositional formulas compactly using a small piece of first-order syntax.

9.1.4 Other Logics

There are two ways we can improve on first-order logic. The first makes modeling easier by increasing expressiveness, allowing us to "say more things", so we can talk about time, beliefs, not just objects and relations. We can also improve the modeler's life by making notation simpler. The second makes life easier for algorithms by decreasing expressiveness.

1. Temporal logic
   - Time as an additional piece of syntax
   - consists four such temporal quantifiers, which correspond to existential or universal quantification over the past or future: P,F,H,G
2. Epistemic logic
   - B_a, which takes a formula f and checks if it is entailed by a's beliefs F_a; that is, for every possible world that a might belief we're in, f holds
   - When an agent a believes Fa, that agent must believe all the entailed consequences of a
3. Lambda calculus
   - use lambda calculus to define higher-order functions
   - $\lambda x P(x)$ denotes the set of x for which P(x) is true. This set can be passed in as an argument into the function Count, which produces a term
   - One of the powerful aspects of lambda calculus is exactly that functions can be passed into other functions
4. Description logic
   - more compact (no variables), supports counting, but still decidable
   - With lambda calculus, we can really capture any statement that we want, but it is often too powerful, which makes life difficult for algorithms. Description logic is a reaction to this, and carves out a logic which can be more intuitive to read, and also is actually decidable

9.1.5 Markov Logic

Instead of hard logic, Model a distribution over models $P(W = w; \theta)$. Consists a set of ground formulas, which are formulas without quantifiers and whose atomic formulas do not contain variables.

$$P(W = w; \theta) \propto exp\{\sum_{j=1}^{m} \theta_j \sum_{f \in S_j} I(f, w)\}$$

Maximum likelihood:

$$\max_{\theta} \log P(W = w_0; \theta)$$

Algorithm:

$$\theta \leftarrow \theta + \eta \nabla_{\theta} \log P(W = w; \theta)$$

- Defines distribution over possible worlds
- Parameter sharing between all grounded instances of a formula
- Probabilistic inference: Gibbs sampling
- Lifted Inference: the factors are not arbitrary, but rather based on first-order logic formulas;

9.2 Semantic Parsing: Language to Logic

Principle of compositionality: The semantics of a sentence is combination of meanings of its parts.

- A grammar is a set of rules. The lexicon contains a set of lexical rules, which map natural language to formulas. We also have two rules that perform forward and backward application
- Negation, Coordination, Quantification
- Ambiguity
  - Lexical ambiguity: multiple word meaning
  - Scope ambiguity
- Algorithm
  - Inference (parsing): construct derivations recursively (dynamic programming)
  - Learning: define ranking loss function, optimize with stochastic gradient descent

10 Deep Learning

10.1 Supervised Learning

10.2 Unsupervised Learning

10.3 Convolutional Neural Networks

10.4 Recurrent Neural Networks

1. State-based models: search problems, MDPs, games Applications: route finding, game playing, etc. Think in terms of states, actions, and costs

2. Variable-based models: CSPs, Bayesian networks Applications: scheduling, tracking, medical diagnosis, etc. Think in terms of variables and factors

3. Logic-based models: propositional logic, first-order logic Applications: theorem proving, verification, reasoning Think in terms of logical formulas and inference rules