Contents

Lectures

Contents

Lectures		**April 09, 2019**
	Machine Learning II	
Table of Contents Local		*Written by Brandon McKinzie*

Roadmap.

- Features.
- Neural Networks.
- Gradients without tears.
- Nearest Neighbors.

Linear Classifiers. Discussion of feature maps $\phi(x)$ that can be used for making a non-linear decision boundary [52:00].

Neural Networks.

Predicting Car Collision [58:00]

Input: position of two oncoming cars $x = [x_1, x_2]$.
Output: Whether safe $(y = +1)$ or collide $y = -1$.
We are told that the true function is

$$y^* = \mathrm{sign}(|x_1 - x_2| - 1)$$

- **Insight**: In (x_1, x_2) space, there is a "band" (diagonal to the right centered about origin) representing "safe" [58:30]. Recognize that these can be thought of as two linear decision boundaries.
- **Decompose** into subproblems[a].

$$h_1 = \mathbb{1}\left[x_1 - x_2 \geq 1\right] \tag{1}$$
$$h_2 = \mathbb{1}\left[x_2 - x_1 \geq 1\right] \tag{2}$$
$$y = \mathrm{sign}\left(h_1 + h_2\right) \tag{3}$$

- **Learning strategy**. Define $\boldsymbol{\phi}(\boldsymbol{x}) \triangleq [1, x_1, x_2]$.

$$h_1 = \mathbb{1}\left[\boldsymbol{v}_1 \cdot \boldsymbol{\phi}(\boldsymbol{x}) \geq 0\right] \qquad \boldsymbol{v}_1 \triangleq [-1, +1, -1] \tag{4}$$
$$h_2 = \mathbb{1}\left[\boldsymbol{v}_2 \cdot \boldsymbol{\phi}(\boldsymbol{x}) \geq 0\right] \qquad \boldsymbol{v}_2 \triangleq [-1, -1, +1] \tag{5}$$
$$f_{\boldsymbol{V},\boldsymbol{w}}(\boldsymbol{x}) = \mathrm{sign}\left(\boldsymbol{w} \cdot \boldsymbol{h}\right) \qquad \boldsymbol{w} \triangleq [1, 1] \tag{6}$$

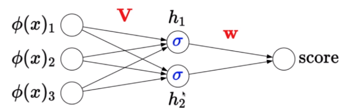
Key idea: joint learning [1:03:42]. Learn both hidden subproblems $\boldsymbol{V} = (\boldsymbol{v}_1, \boldsymbol{v}_2)$ and combination weights $\boldsymbol{w} = [w_1, w_2]$.

[a] He is defining $\mathrm{sign}\left(0\right) \equiv -1$.

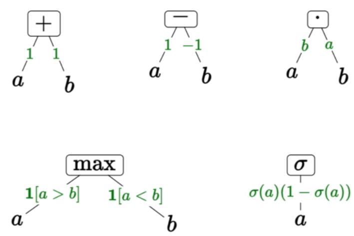
Gradients. (Continuing off example) Our parameters are $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{w}\}$. To learn these parameters, we may consider gradient descent on y (and therefore on h_1 and h_2). Notice that

$$\nabla_{\boldsymbol{v}_1} h_1 = \boldsymbol{0} \tag{7}$$

Solution: Redefine $h_1 \triangleq \sigma(\boldsymbol{v}_1 \cdot \boldsymbol{\phi}(\boldsymbol{x}))$.

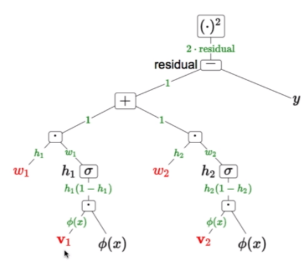


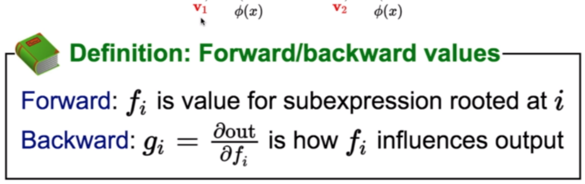
Gradients without tears [1:14:00]. Gradients of common function building blocks:



Binary classification with hinge loss [1:17:01].

$$L(x, y, w) = \max\{1 - \boldsymbol{w} \cdot \boldsymbol{\phi}(\boldsymbol{x})y, 0\} \tag{8}$$





Lectures		April 11, 2019
	Machine Learning III	
Table of Contents Local		*Written by Brandon McKinzie*

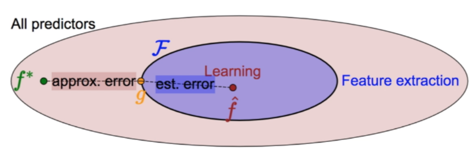
Review. Some feature extractor $\boldsymbol{\phi}(\boldsymbol{x})$.

- Linear predictor score: $\boldsymbol{w} \bullet \boldsymbol{\phi}(\boldsymbol{x})$.
- Neural network score: $\sum_{j=1}^{k} w_j \sigma\left(\boldsymbol{v}_j \bullet \phi(x)\right)$.

Generalization. Goal is to minimize error on unseen future examples.

- Suppose there is some space of all possible predictors f and, within this space, there exists the optimal predictor f^*.
- When we decide how we will do our feature extraction, we constrain the possible predictors we can get to be inside some **hypothesis class** $\mathcal{F}$ **[21:11]**. Note that there exists a predictor g that is optimal, constrained to being inside $\mathcal{F}$.
- **Approximation Error**: How good is our hypothesis class $\mathcal{F}$?
- **Estimation Error**: How good is our learned predictor $\hat{f}$ *relative to* our hypothesis class $\mathcal{F}$?

$$\mathrm{Err}(\hat{f}) - \mathrm{Err}(f^*) = \underbrace{\mathrm{Err}(\hat{f}) - \mathrm{Err}(g)}_{\text{Estimation}} + \underbrace{\mathrm{Err}(g) - \mathrm{Err}(f^*)}_{\text{Approximation}} \tag{9}$$



For linear predictors with weights $\boldsymbol{w} \in \mathbb{R}^d$, we can **control the size of** $\mathcal{F}$ by ...

- Keeping dimensionality d small.
  - Add/remove features.
  - Forward selection.
  - Boosting.
  - L_1 regularization.
- Keeping the norm $||\boldsymbol{w}||$ small.

Note that **SVMs** = hinge loss + regularization.

How do we choose **hyperparameters** [43:39]? One way is to use a **validation set**.

Unsupervised Learning [1:06:00]. Data has rich **latent structures** that we'd like to learn automatically. Two types of unupervised learning: **clustering** (e.g. K-means) and **dimensionality reduction** (e.g. PCA).

Clustering

Input: *Training set of inputs* $\mathcal{D}_{train} = \{x_1, \ldots, x_n\}$.

Output: *Assignment of each point to a cluster* $[z_1 \ldots, z_n]$ *where* $z_i \in \{1, \ldots, K\}$.

K-Means Clustering [1:10:35]

Setup: Each cluster $k = 1, \ldots, K$ is represented by a **centroid** $\boldsymbol{\mu}_k \in \mathbb{R}^d$. Want each $\phi(x_i)$ close to its assigned centroid $\boldsymbol{\mu}_{z_i}$.
Objective: MSE.

$$L(z, \mu) = \sum_{i=1}^{n} ||\phi(x_i) - \boldsymbol{\mu}_{z_i}||^2 \tag{10}$$

Algorithm:

1. For each point x_i, assign:

$$z_i := \arg\min_{k=1,\ldots,K} ||\phi(x_i) - \mu_k||^2 \tag{11}$$

2. For each cluster $k = 1, \ldots, K$, assign:

$$\boldsymbol{\mu}_k := \frac{1}{|\{i : z_i{=}k\}|} \sum_{i:z_i=k} \phi(x_i) \tag{12}$$

Lectures		April 16, 2019
	Search I	
Table of Contents Local		*Written by Brandon McKinzie*

Running example: farmer has a boat. Needs to get a cabbage, goat, and wolf across river. Goat and cabbage cannot be alone. Wolf and goat cannot be alone. Boat can hold at most two things (including farmer). How he do it tho?

Beyond Reflex [9:00]. Classifiers (reflex) map x to single action y. In **search**, we map x to an *action sequence* $(a_1, a_2, \ldots)$. **Key**: Need to consider consequences of future actions.

Tree search. Root note is start state. Edges are actions. Nodes are states resulting from said action. Branches represent a sequence of actions.

- S_{start} is "FCGW|" (farmer, cabbage, goat, wolf all on left of river).
- Actions: farmer can go to/from river. Actions enumerate what he takes with him and what direction he goes.
- Cost(s, a): ?
- Succ(s, a): returns the state resulting from taking action a from state s.
- IsEnd(s): ?

Transportation example [17:40]

Streets with blocks numbered 1 to n. Walking from s to $s+1$ takes 1 minute. Taking a magic tram from s to $2s$ takes 2 minutes. How to travel from 1 to n in the least time?

You should read this and then try formalizing it as a search problem.

- Actions: either take 1 step forward or take the tram.
- The costs are 1 minute and 2 minutes for the aforementioned actions.

She starts coding it up at **[19:30]**. Takeaways from coding:

- Separate modeling from inference.
- Modeling: She defines a class with methods startState, isEnd(state), succAndCost(state).

Backtracking search [23:10]. Also uses a search tree. Define branching factor b and depth D of tree[1]. BS goes down all the way leftmost to leaf. Sounds like **depth-first search** tbh[2]

- Memory: $\mathcal{O}(D)$ (small). Why? because we only need to remember our past history, which for BS is basically a straight line from leaf back up to root.
- Time: $\mathcal{O}(b^D)$ (huge). In other words, all nodes. Why? because worst-case is that it has to visit every single node.

[1]Distinguishing between uppercase D and lowercase d will be important in this lecture.

[2]At **[34:00]** she address this. Yes, it is basically DFS, but DFS will *stop* as soon as it hits a leaf (a solution). Also, DFS implicitly defines all costs as zero.

She starts coding for BS at [27:28]. Note that she is able to reuse her code from the previous part, where she just defined the *model*. Now, she implements BS which is the *inference*. Below is the BS *algorithm*[3]:

```
def backtracking_search(s, path):
  if is_end(s):
    minimum_cost_path = min(minimum_cost_path, path)

  for a in actions(s):
    path.append([succ(s, a), cost(s, a)])
    backtracking_search(Succ(s, a), path)

  return minimum_cost_path
```

Breadth-first Search [37:00]. Requires that cost for all actions is the same constant $c \geq 0$. Explores level-by-level. Like DFS, terminates as soon as it finds a solution[4]. Important: let d (small d) denote the depth of the *solution*.

- Memory: $\mathcal{O}(b^d)$ (a lot worse!).
- Time: $\mathcal{O}(b^d)$ (better, depends on d, not D).

Brief overview of **DFS with iterative deepening** at [41:00]. Hybrid of DFS with BFS. Call DFS at max depths 1, 2, ... etc. $\mathcal{O}(d)$ space and $\mathcal{O}(b^d)$ time. Nice summary slide of tree search algorithms at [43:11].

Dynamic programming [43:30]. Observation: the cost of going from s to end state can be decomposed into cost of $(s, a) \to s'$ plus the cost of going from s' to end state.

- Denote "future cost" of state s as $F(s)$.

$$F(s) \triangleq \begin{cases} 0 & IsEnd(s) \\ \min_{a \in Actions(s)} C(s,a) + F(Succ(s,a)) & \text{otherwise} \end{cases} \tag{13}$$

- **Key observation**: *we can cache intermediate results.*
- **Assumption**: State graph defined by $Actions(s)$ and $Succ(s, a)$ is acyclic.
- State should be sufficient to encode full history, so that futureCost(state) can be computed.

[3]Which is slightly different in form than her code.
[4]This is a consequence of the fixed global cost $c \geq 0$.

Route finding [47:38]

Find minimum cost path from city 1 to n, only moving forward. It costs c_{ij} to go from i to j.

Clarification: Yeah, you can go from any i to $i+j$ for $0 \leq j \leq n-i$. **Observation**: future cost only depends on the current city (since we can only move forward). Cache, cache, cache.

What if we add the **constraint** that you can't visit three odd cities in a row. How does this change our definition of a state? Well, now we need to encode history somehow.

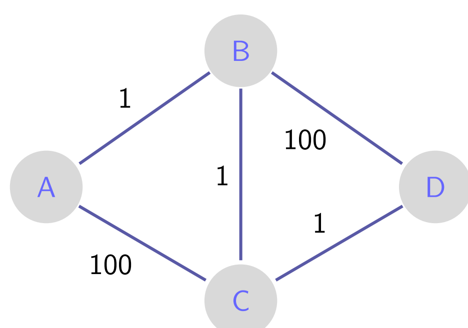
- Naive: redefine our state as (prev city, current city). This increases our state space size from n to n^2.
- More compact: redefine our state as (prevWasOdd, current city). Now our state space is $2n$.

Coding for dynamic programming:

```
def dynamic_programming(problem):
  cache = {} # state -> future_cost(state)
  def future_cost(state):
    if problem.is_end(state):
      return 0
    if state in cache:
      return cache[state]
    result = min(cost + future_cost(new_state) for _, new_state, cost in
       problem.succ_and_cost(state))
    cache[state] = result
    return result
  return future_cost(problem.start_state(), [])
```

Uniform cost search [1:05:00]. Can handle cycles.

- **Assumption**: $\forall(s, a) : C(s, a) \geq 0$.
- **Observation**: prefixes of optimal path are optimal.
- **Key idea**: Enumerate states in order of increasing past cost.



Goal: Use UCS to get from A to D.

- Maintain 3 sets of states: explored, frontier, and unexplored.
- We start with all states in the unexplored sets.

- Pop A into the frontier. What states can we reach from A?
- Pop B and C into frontier and the cost to get to them. A is now in the explored set.

It seems we don't explore states we've already been to? UCS is *correct*: when any state s is moved from frontier to explored, its priority of $PastCost(s)$ is the minimum cost to s (discussion around **[1:18:00]**).

Coding for UCS begins at **[1:15:00]**.

```
def uniform_cost_search(problem):
  frontier = PriorityQueue()
  frontier.update(problem.start_state(), 0)

  while True:
  # Move from frontier to explored.
  state, past_cost = frontier.remove_min()
  if problem.is_end(state):
    return past_cost, []
  # Push out onto frontier.
  for a, new_state, cost in problem.succ_and_cost(state):
    frontier.update(new_state, past_cost + cost)
```

Algorithm: uniform cost search [Dijkstra, 1956]

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

Lectures		**April 18, 2019**
	Search II	
Table of Contents Local		*Written by Brandon McKinzie*

Discussion of **Final Project** up until [15:00].

Review. **Key idea**: a state is a summary of all the past actions sufficient to choose future actions **optimally**. The class paradigm includes modeling, inference, and learning. Today will talk more about learning.

Learning costs [25:00]. What if we don't have access to the costs? Goal: develop a learning algorithm that outputs the costs of actions.

$$\text{Forward problem } \textbf{(Search)} \quad Cost(s, a) \longrightarrow (a_1, \ldots, a_k) \tag{14}$$

$$\text{Inverse problem } \textbf{(Learning)} \quad (a_1, \ldots, a_k) \longrightarrow Cost(s, a) \tag{15}$$

Prediction (inference) problem:

- Inputs (x): search problem without costs.
- Outputs (y): sequence of optimal actions.

So our training data is search problem inputs and the labeled solutions (action sequence) for each. We need to learn that mapping from search problem to optimal action sequence. The idea is that we'll learn the costs by doing this. A simple way of modeling this[5]:

$$Cost(s, a_i) \triangleq w[a_i] \triangleq w_i \tag{16}$$

Structured Perceptron (simplified) [32:00]

1. For each action a, initialize $w[a] \leftarrow 0$.
2. For each iteration $t = 1, \ldots, T$:
   (a) For each $(x, y) \in \mathcal{D}_{train}$:
      i. Compute minimum cost path y' given $\boldsymbol{w}$.
      ii. $\forall a \in y : w[a] \leftarrow w[a] - 1$.
      iii. $\forall a \in y' : w[a] \leftarrow w[a] + 1$.

Coding for this starts at [38:00]. It just uses a modified version of TransportationProblem (model) that incorporates weights, and uses the dynamicProgramming function (inference) unchanged. Then it updates weights as defined above.

[5]Note that w does not care about the state we are taking the action from.

Generalization to features [46:10]. Costs are parameterized by [incorporating] a feature vector:

$$Cost(s,a) = \boldsymbol{w} \cdot \phi(s,a) \tag{17}$$
$$Cost(y) = \boldsymbol{w} \cdot \sum_{(s,a)\in y} \phi(s,a) \tag{18}$$

A* search [46:30]. Improving upon UCS:

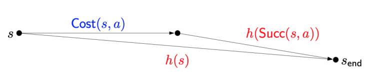
- **Problem**: UCS orders states by cost from s_{start} to s.
- **Goal**: take into account cost from s to s_{end}.

Since we obviously don't know $FutureCost(s)$, A* will explore in order $PastCost(s) + h(s)$, using a **heuristic function** that estimates future cost.

$$\underbrace{Cost'(s,a)}_{\text{A*}} = \underbrace{Cost(s,a)}_{\text{UCS}} + h\left(Succ\left(s,a\right)\right) - h(s) \tag{19}$$

- **Consistency of h.** A heuristic h is **consistent** if both of the following hold:

$$Cost'(s,a) \geq 0 \tag{20}$$
$$h(s_{end}) = 0 \tag{21}$$



Proposition: Correctness of A* [56:40]

If h is consistent, A returns the minimum cost path*[6].

- **Admissibility of h [1:02:20].** A heuristic h is admissible if

$$h(s) \leq FutureCost(s) \tag{22}$$

Intuition: admissible heuristics are *optimistic*. Theorem: $isConsistent(h) \Rightarrow isAdmissible(h)$.

[6] Proof hint: show that A* ultimately ends up returning UCS + a constant after you sum everything up.

Efficiency.

- UCS efficiency: UCS explores all states s satisfying

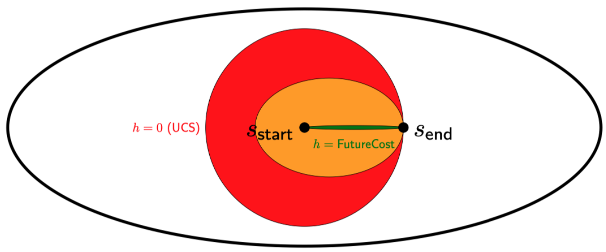
$$PastCost(s) \leq PastCost(s_{end})$$

.

- **A* efficiency: A* explores all states s satisfying**

$$PastCost(s) \leq PastCost(s_{end}) - h(s)$$

.[7]



- If $h(s) = 0$, then A* is same as UCS.
- If $h(s) =$ FutureCost(s), then A* only explores nodes on a minimum cost path.
- Usually $h(s)$ is somewhere in between.

Key Idea: distortion [1:02:00]

A* distorts edge costs to favor end states.

Relaxation [1:04:00]. How we find heuristics. Basically, removing constraints.

[7] IDGI. Plz explain here when smarter.

Lectures **April 23, 2019**

Markov Decision Processes

Table of Contents Local *Written by Brandon McKinzie*

Motivation. (Building off search problems) There is **uncertainty** in the world. Volcano crossing example starts at **[7:20]**.

Dice Game [13:12]

For each round $r = 1, 2, \ldots$
- You choose to **stay** or **quit**.
- If **quit**, you get \$10 and end the game.
- If **stay**, get \$4 and then roll a 6-sided dice.
  - If result is 1 or 2 $\rightarrow$ end game.
  - Otherwise, continue to next round.

Train of thought should roughly go as follows:
- If I follow policy **stay**, I can compute my expected reward as

$$\frac{1}{3}(4) + \frac{2}{3}\frac{1}{3}(8) + \frac{2}{3}\frac{2}{3}\frac{1}{3}(12) + \ldots = 12 \tag{23}$$

  where the $\frac{1}{3}$ represent the probability that my roll ends the game.

Using MDP lingo, we revisit this at [44:30]. We have States $= \{in, end\}$ and Actions $= \{stay, quit\}$. Our policy is the simple $\pi(in) := stay$. We can compute our expected reward (assuming $\gamma = 1$ discount) as:

$$V_\pi(in) = \frac{1}{3}(4 + \cancel{V_\pi(end)}) + \frac{2}{3}(4 + V_\pi(in)) = 12 \tag{24}$$

Markov Decision Processes. We can represent an MDP as a graph containing both state nodes and *chance* nodes. Edges from state nodes to chance nodes represent an action. Edges from chance nodes to state nodes represent the random outcomes of the given action.

Definition: Markov decision process

States: the set of states
$s_{\text{start}} \in$ States: starting state
Actions(s): possible actions from state s
$T(s, a, s')$: probability of s' if take action a in state s
Reward(s, a, s'): reward for the transition (s, a, s')
IsEnd(s): whether at end of game
$0 \leq \gamma \leq 1$: discount factor (default: 1)

- $\forall (s, a) : \sum_{s' \in \text{States}} T(s, a, s') = 1$
- Successors are defined as all s' s.t. $T(s, a, s') > 0$.

Revisit transportation example (magic tram) at **[22:00]**, with addition that tram fails with

$p = 0.5$. A *solution* to an MDP is a **policy** π: a mapping from each state $s \in$ States to an action $a \in \text{Actions}(s)$.

Policy Evaluation [30:50]. Following a policy yields a *random path*. Policy terminology:

- **Utility** U is discounted $\sum R(s, a, s')$ on the path (this is a random quantity).
- **Value** $V_\pi(s)$ of policy π from state s is the $\mathbb{E}[U]$ received by following policy π.
- **Q-value** $Q_\pi(s, a)$ is $\mathbb{E}[U]$ of taking action a from state s, and then following policy π.

We can compute these quantities via:

$$V_\pi(s) = \begin{cases} 0 & \text{if IsEnd(s)} \\ Q_\pi(s, \pi(s)) & \text{otherwise} \end{cases} \tag{25}$$

$$Q_\pi(s, a) = \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma V_\pi(s') \right] \tag{26}$$

Policy Evaluation [45:00]

Goal: automatically compute value of any policy π.

1. Initialize $V_\pi^{(0)}(s) \leftarrow 0$ for all states s.
2. For each iteration $t = 1, \ldots, t_{PE}$ and for each state s:

$$V_\pi^{(t)}(s) \leftarrow \underbrace{\sum_{s'} T(s, \pi(s), s') \left[R(s, \pi(s), s') + \gamma V_\pi^{(t-1)}(s') \right]}_{Q^{(t-1)}(s, \pi(s))} \tag{27}$$

where it's important to note that we use the values $V_\pi^{(t-1)}(()\, s')$ from the previous iteration when computing the values $V_\pi^{(t)}(s)$ for the current iteration. We are iteratively updating our estimates of V. We want to choose t_{PE} such that

$$\max_{s \in \text{States}} \left| V_\pi^{(t)}(s) - V_\pi^{(t-1)}(s) \right| \leq \epsilon \tag{28}$$

Complexity of policy evaluation:

- Time: $\mathcal{O}(t_{PE} S S')$

Value iteration [51:05]. Given an MDP, how do we find a good policy π?

Value Iteration [57:00]

1. Initialize $V_\pi^{(0)}(s) \leftarrow 0$ for all states s.
2. For each iteration $t = 1, \ldots, t_{VI}$ and for each state s:

$$V_{opt}^{(t)}(s) \leftarrow \max_{a \in \textbf{Actions(s)}} \underbrace{\sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma V_{opt}^{(t-1)}(s') \right]}_{Q_{opt}^{(t-1)}(s, a)} \tag{29}$$

We want to choose t_{PE} such that

$$\max_{s \in \text{States}} \left| V_\pi^{(t)}(s) - V_\pi^{(t-1)}(s) \right| \leq \epsilon \tag{30}$$

Complexity of value iteration:

- Time: $\mathcal{O}(t_{VI}SAS')$

Coding of VI starts around [**1:00:00**]. Convergence discussion around [**1:08:00**].

MDPs	RL
Mental model of world	Don't know how world works
Find policy to collect max R	Perform actions in world to find out and collect R

Monte Carlo Methods [18:53]. Our data consists of paths of the form $s_0; a_1, r_1, s_1; a_2, r_2, s_2; \ldots$, which in RL we also call an **episode**[8].

Model-Based [Monte Carlo] Learning

Estimate the MDP parameters $T(s,a,s')$ and $R(s,a,s')$.

$$\hat{T}(s,a,s') = \frac{Count(s \to a \to s')}{Count(s \to a)} \tag{31}$$

$$\hat{R}(s,a,s') = r \text{ in } (s,a,r,s') \tag{32}$$

We can use these parameter estimates to compute an estimate for $\hat{Q}_{opt}(s,a)$.

Problem: might not see all of the state/action space. Solution: Need π to *explore* explicitly. Next, consider that all we really need for prediction is (estimate of) $Q_{opt}(s,a)$:

$$\hat{Q}_{opt}(s,a) = \sum_{s'} \hat{T}(s,a,s') \left[\hat{R}(s,a,s') + \gamma \hat{V}_{opt}(s') \right] \tag{33}$$

Model-Free [Monte Carlo] Learning

Estimate $Q_{opt}(s,a)$ directly (without learning parameters). Recall that we define the utility at time t as

$$u_t \triangleq r_t + \gamma \cdot r_{t+1} + \gamma^2 \cdot r_{t+2} + \cdots \tag{34}$$

Collect all episodes in our data for which there exists transition[9] $s_{t-1}{=}s \longrightarrow a_t{=}a$. *Then*[10] $\hat{Q}_\pi(s,a) := average(\{u_t\})$ *for those episodes.*

Model-Free Monte Carlo is an example of an **on-policy** algorithm: estimating the value of data-generating policy. There are also **off-policy** algorithms, ones that estimate the value of a different policy than seen in the data. On-Policy vs Off-Policy

[8] Defined as beginning with a start state, and ending with an end state.
[9] And s, a doesn't occur in $s_0, \ldots, s_{t-2}$ (i.e. only care about first occurrence).
[10] Assume for now that we've decided on using some policy π.

Model-Free Equivalences [39:40].

- **Original**: $\hat{Q}_\pi(s,a) := \text{average}(\{u_t\})$
- **Convex combination**: At each occurrence of u_t, we assign $\hat{Q}_\pi(s,a)$ to the convex combination of itself (our current estimate) and u_t:

$$\hat{Q}_\pi(s,a) \leftarrow (1-\eta)\hat{Q}_\pi(s,a) + \eta u \tag{35}$$

$$\text{where} \quad \eta := \frac{1}{1 + \text{NumUpdates}(s,a)} \tag{36}$$

  Where "NumUpdates" for a given (s,a) is the number of times we've updated our estimate of $\hat{Q}_\pi(s,a)$ so far[11].
- **Stochastic gradient** [42:23]:

$$\hat{Q}_\pi(s,a) \leftarrow \hat{Q}_\pi(s,a) - \eta \left[\underbrace{\hat{Q}_\pi(s,a)}_{\text{prediction}} - \underbrace{u}_{\text{target}} \right] \tag{37}$$

  where the implicit objective is least squares regression $(Q-u)^2$.

Bootstrapping methods: SARSA [50:20]

Motivation: the values of u we observe in the data often have high variance[a].

[a]This is because we are defining u as the discounted sum of rewards from whenever (s,a) first happened *until the end of the episode*. A lot of random stuff can happen in that sequence of actions.

On each (s,a,r,s',a'):

$$\hat{Q}_\pi(s,a) \leftarrow (1-\eta)\hat{Q}_\pi(s,a) + \eta \left[\underbrace{r}_{\textbf{data}} + \gamma \underbrace{\hat{Q}_\pi(s',a')}_{\textbf{estimate}} \right] \tag{38}$$

Comparison with model-free MC (colors are unrelated to colors in eq above):

- **biased**[a]
- **small variance**
- **immediate updates** (don't have to wait until end of episode).

[a]TODO: should probably know how to prove this.

Bootstrapping methods: Q-Learning [59:50]

On each (s,a,r,s'):

$$\hat{Q}_{opt}(s,a) \leftarrow (1-\eta) \underbrace{\hat{Q}_{opt}(s,a)}_{\text{prediction}} + \eta \underbrace{\left(r + \gamma \hat{V}_{opt}(s') \right)}_{\text{target}} \tag{39}$$

$$\hat{V}_{opt}(s') = \max_{a' \in Actions(s')} \hat{Q}_{opt}(s',a') \tag{40}$$

[11]So e.g. if our data only contains u corresponding to the same (s,a), then η_i (for iteration i) would be $\frac{1}{1+i}$ (we start at $i=0$).

Covering the unknown. Note that we still have not talked about how to decide what actions to take. There are a few approaches:

$$\begin{aligned}
&\textbf{[Exploit Only]} && \pi_{exploit}(s) := \arg\max_{a \in Actions(s)} \hat{Q}_{opt}(s,a) && (41)\\
&\textbf{[Explore Only]} && \pi_{explore}(s) := \text{GetRandom}(Actions(s)) && (42)\\
&\textbf{[Epsilon-Greedy]} && \pi_{eps} := \begin{cases} \pi_{exploit} & \text{probability } 1-\epsilon \\ \pi_{explore} & \text{probability}\epsilon \end{cases} && (43)
\end{aligned}$$

- Exploit Only: $\hat{Q}(s,a)$ inaccurate, too greedy.
- Explore Only: average utility is low because exploration is not guided. Our Q-values are actually quite good, though.

Another problem is **generalization** **[1:07:00]**, which is particularly relevant when our state space is *large*. Also note that we've not yet taken advantage of similarities between states, we've basically been treating states as individual black boxes. We can't generalize to unseen states/actions.

Q-learning with function approximation

Define features $\boldsymbol{\phi}(s,a)$ and weights $\boldsymbol{w}$.

$$\hat{Q}_{opt}(s,a;\boldsymbol{w}) := \boldsymbol{w} \cdot \boldsymbol{\phi}(s,a) \tag{44}$$

On each (s,a,r,s'):

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \left[\underbrace{\hat{Q}_{opt}(s,a)}_{\text{prediction}} - \underbrace{\left(r + \gamma \hat{V}_{opt}(s')\right)}_{\text{target}} \right] \boldsymbol{\phi}(s,a) \tag{45}$$

with implied objective $(\hat{Q} - (r + \gamma \hat{V}))^2$.

Lectures	**April 30, 2019**

Games I

Table of Contents Local *Written by Brandon McKinzie*

Code for the $\lfloor \frac{N}{2} \rfloor$ game at **[13:00]**.

Two-Player Zero-Sum Game

- s_{start}*: starting state*
- *Actions*(s)
- *Succ*$(s, a) \rightarrow s'$
- *IsEnd*(s)
- *Utility*(s)*: agent's utility for* ***end state***[12] s
- *Player*(s)*: player who controls state* s

Policies of Player p

- ***Deterministic****:* $\pi_p(s) \in \textit{Actions}(s)$
- ***Stochastic****:* $\pi_p(s, a) \in [0, 1]$

Evaluation [22:00]. Let $V_{eval}(s)$ denote the expected value of state s. We also say that it's a recurrence for expected utility.

$$V_{eval}(s) = \begin{cases} \text{Utility}(s) & \text{if IsEnd}(s) \\ \sum_{a \in \text{Actions}(s)} \pi_{\text{Player}(s)}(s, a) V_{eval}(\text{Succ}(s, a)) & \text{otherwise} \end{cases} \tag{46}$$

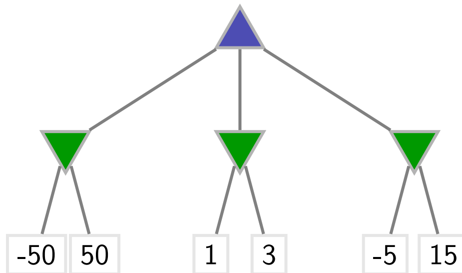
Around **[26:00]**, she introduces the idea of V_{expmax} (expectimax[13]) that uses $\max_{a \in \text{Actions}(s)}$ in the above equation, which corresponds to a player that always maximizes their expected utility. Note, however, that there is no need to clutter our brains[14] with new definitions, since this is still the same definition as above with a player that has $\pi(s, a)$ equal to 1 for the action that maximizes expected utility, and zero elsewhere.

[12]**TODO**: So all non-terminal states have zero utility? Answer: No, there is no utility for non-terminal states. It's just not a thing.

[13]It seems that we say "expectimax" when we assume our opponent plays ~~randomly~~ deterministically, and "minimax" when we assume our opponent is trying to minimize our utility.

[14]We can, however, clutter this footer with them... **[29:31]**

$$V_{minmax}(s) = \begin{cases} \text{Utility}(s) & \text{if IsEnd}(s) \\ \max_{a \in \textbf{Actions}(s)} V_{minmax}(\text{Succ}(s, a)) & \text{Player(s)=}\textit{agent} \\ \min_{a \in \textbf{Actions}(s)} V_{minmax}(\text{Succ}(s, a)) & \text{Player(s)=}\textit{opp} \end{cases} \tag{47}$$



Minimax Property 1 [41:49]. If opponent is minimizing, agent should maximize.

$$V(\pi_{agent}, \pi_{min}) \geq V(\pi_{agent}, \pi_{min}) \quad \forall \pi_{agent} \tag{48}$$

However, if e.g. your opponent plays randomly, then agent should should use expectimax (lesson: maximizing not *always* optimal).

Computation.

- **Tree Search**: $\mathcal{O}(d)$ space, $\mathcal{O}(b^{2d})$ time[15].

Evaluation Functions [53:25]. We can use **Depth-limited [tree] search**: stop at some maximum depth d_{max}. This can be defined by making a slight modification to eq. 46:

$$V_{minmax}(s, d) = \begin{cases} \text{Utility}(s) & \text{if IsEnd}(s) \\ \textbf{Eval}(s) & \mathbf{d = 0} \\ \max_{a \in \text{Actions}(s)} V_{minmax}(\text{Succ}(s,a), d) & \text{Player(s)=}agent \\ \min_{a \in \text{Actions}(s)} V_{minmax}(\text{Succ}(s,a), d-1) & \text{Player(s)=}opp \end{cases} \tag{49}$$

where Eval(s) is an estimate[16] of $V_{minimax}(s)$. For example, in chess we could define Eval$(s) :=$ *sum*(material, mobility, king-safety). Pacman code example at [57:28].

[15] She defines d really confusingly ([52:00]), but wikipedia says d should be the number of moves a given player makes, while plies (2d here since 2 players) is the total number of moves made.

[16] No guarantees on error of estimate, unlike A*.

Alpha-Beta Pruning [59:19]. **Key Idea**: branch and bound. Maintain lower and upper bounds on values. If intervals don't overlap, we can choose optimally without considering the non-optimal interval choices. Specifically, when exploring tree, we can keep track of...

- a_s: lower bound on value of max node s (we know value is *at least* ($\geq$) a_s)
- b_s: upper bound on value of min node s (we know value is *at most* ($\leq$) b_s).
- $\alpha_s \triangleq \max_{s' \preceq s} a_s$, where the max is over ancestors of s.
- $\beta_s \triangleq \min_{s' \preceq s} b_s$.

which clearly only makes sense in a minimax context[17]. Detailed walkthrough example at **[1:05:00]**. The values of a_s and b_s can really be thought of locally, in the sense that they are determined by either a max (a_s) or min (b_s) over the values presented to them by their immediate successors (values are propagated "up" the tree when we are computing these quantities).

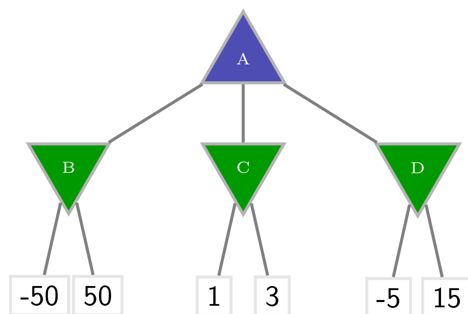
Clearly, **ordering** (when traversing/considering states) makes a big difference with pruning. In practice, ordering by Eval(s) can be a good strategy:

- Max nodes: order successors by decreasing Eval(s).
- Min nodes: order successors by increasing Eval(s).

[17]**TODO**: think about situations where we aren't doing minimax and want to prune. This was an exam question in CS188.

1.7.1 Adversarial Search (AIMA Ch. 5)

First, I think it's crucial to understand the game tree in depth. Below is a 2-ply game tree[18], which translates for game problems to 2-player game tree. The leaf nodes show the *utility values as perceived by the root node*. **IMPORTANT**: the only reason it is valid for us to focus on the utility of the root is because this is a *two-player zero-sum game* (the MIN utilities are defined as the negation of the MAX utilities). If this were a multiplayer game, we'd have to track a *vector* of utilities for each player.



Define the **minimax value** $V_{minmax}(s)$ of state s to be the utility [19] of being in state s, *assuming both players play optimally* from there to the end of the game.

$$V_{minmax}(s) = \begin{cases} \text{Utility}(s) & \text{if IsEnd}(s) \\ \max_{a \in \text{Actions}(s)} V_{minmax}(\text{Succ}(s,a)) & \text{Player(s)=}agent \\ \min_{a \in \text{Actions}(s)} V_{minmax}(\text{Succ}(s,a)) & \text{Player(s)=}opp \end{cases} \tag{50}$$

[18] This tree has depth 1, with 2 plys.
[19] as perceived by root

```python
def minimax_decision(root: State) -> Action:
   """We assume the root is a MAX node."""
   best_action = None
   best_value = -float("inf")
   for action in root.actions():
      value = min_value(succ(root, action))
      if value > best_value:
         best_value = value
         best_action = action
   return best_action

def min_value(state: State) -> float:
   """Returns V_minmax from perspective of a MIN node."""
   assert isMinNode(state)
   if isEnd(state):
      return state.utility()
   v = float("inf")
   for action in state.actions():
      v = min(v, max_value(succ(state, action)))
   return v

def max_value(state: State) -> float:
   """Returns V_minmax from perspective of a MAX node."""
   assert isMaxNode(state)
   if isEnd(state):
      return state.utility()
   v = -float("inf")
   for action in state.actions():
      v = max(v, min_value(succ(state, action)))
   return v
```

Alpha-Beta Pruning (5.3). General principle:

- Consider some node n somewhere in the tree, such that *agent* has a choice of moving to that node (i.e. n is an *immediate descendent* of an *agent* (MAX) node[20]).

Conceptual algorithm for the two-ply tree we've been using.

1. Make your way down the left-most path of the tree until you hit the MIN node just before the leafs.
   (a) Collect each leaf utility. You can update the interval of possible values that the parent MIN node can have as you go, but it doesn't seem like we really do anything with those intermediate values (**right**?).
   (b) You now know that the given MIN node will choose the smallest value. Let's call that value m_1 (the first min node value we've determined).
2. The parent of MIN (which is the root MAX node for our tree) sees that MIN has set its value/action choice as m_1. This means the root's value will be *at least* m_1.

[20] This also implies (for a two-player game) that n is either a MIN node or a leaf.

Key points to remember:

- When we speak about “ranges” or intervals, we are talking about the range of possible values $V(s)$ could be, given the information we’ve collected thus far.

Lectures **May 2, 2019**

Games II

Table of Contents Local *Written by Brandon McKinzie*

Review: Evaluation Function. New idea: can we *learn* the eval function? We can redefine $\text{Eval}(s) = V(s; \boldsymbol{w})$. Some possible models might be a linear model, a simple neural network:

$$V(s; \boldsymbol{w}) = \boldsymbol{w} \cdot \phi(s) \tag{51}$$

$$V(s; \boldsymbol{w}, \boldsymbol{v}_{1:k}) = \sum_{j=1}^{k} w_j \sigma(\boldsymbol{v}_j \cdot \phi(s)) \tag{52}$$

Temporal Difference Learning (TD-Learning). In the learning scenario, we treat our estimate of V as the *prediction*, and the value of $r + \gamma V(s')$ as the *label*. We then define an objective and perform gradient descent. If we use MSE objective, then TD-learning is defined as performing, for each (s, a, r, s') [25:00].

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \left[V(s; \boldsymbol{w}) - (r + \gamma V(s'; \boldsymbol{w})) \right] \nabla_{\boldsymbol{w}} V(s; \boldsymbol{w}) \tag{53}$$

Algorithm: TD learning

On each (s, a, r, s'):

$$\mathbf{w} \leftarrow \mathbf{w} - \eta[\underbrace{\hat{V}_\pi(s; \mathbf{w})}_{\text{prediction}} - \underbrace{(r + \gamma \hat{V}_\pi(s'; \mathbf{w}))}_{\text{target}}] \nabla_{\mathbf{w}} \hat{V}_\pi(s; \mathbf{w})$$

Algorithm: Q-learning

On each (s, a, r, s'):

$$\mathbf{w} \leftarrow \mathbf{w} - \eta[\underbrace{\hat{Q}_{\text{opt}}(s, a; \mathbf{w})}_{\text{prediction}} - \underbrace{(r + \gamma \max_{a' \in \text{Actions}(s)} \hat{Q}_{\text{opt}}(s', a'; \mathbf{w}))}_{\text{target}}] \nabla_{\mathbf{w}} \hat{Q}_{\text{opt}}(s, a; \mathbf{w})$$

Q-Learning	TD Learning
$\hat{Q}_{opt}(s, a; \boldsymbol{w})$	$\hat{V}_\pi(s; \boldsymbol{w})$
Off-policy	On-policy
Don't need $T(s, a, s')$	Need $\text{Succ}(s, a)$

Two-Finger Morra [40:00]

Players **A** and **B** each show 1 or 2 fingers.

- If both show 1, **B** gives **A** 2 dollars.
- If both show 2, **B** gives **A** 4 dollars.
- Otherwise, **A** gives **B** 3 dollars

Single-Move Simultaneous Game [42:00]

- *Players* $= \{A, B\}$
- $V(a, b)$*:* ***A's utility*** *if A chooses action a, B chooses b. We call V the* ***payoff matrix****.*
- *Evaluation: the value of the game if A follows π_A and B follows π_B:*

$$V(\pi_A, \pi_B) = \sum_{a,b} \pi_A(a)\pi_B(b)V(a, b) \tag{54}$$

where we seem to be using a horrifying overload of notation. In the above, $\pi : a \mapsto [0, 1]$.

Here we'll consider **pure strategies** (a single action) and **mixed strategies** (probability distribution of action). Let's pretend for a moment that the players *do* move sequentially (non-simultaneous). For ***pure strategies***, who should go first?[21] Proposition **[52:34]**: going second is preferable to going first (pure strategies only).

$$\underbrace{\max_a \min_b V(a, b)}_{\text{A goes 1st}} \leq \underbrace{\min_b \max_a V(a, b)}_{\text{A goes 2nd}} \tag{55}$$

Note that the above is not specific to two-finger Morra. What if A is playing a mixed strategy [and tells B what it is]? Proposition: B should always choose a pure strategy. For any mixed strategy π_A, **[56:56]**

$$\min_{\pi_B} V(\pi_A, \pi_B) \tag{56}$$

can be attained by a pure strategy.

[21] It is crucial to remember that $V(a, b)$ is *defined* to be the utility from player A's perspective. In light of this, the proposition should not be surprising at all.

What if A only tells us that they're using some mixed strategy $\pi_A := [p, 1-p]$ (still on two-finger Morra example)? Let's also pretend that player A is going first. We should enumerate the values for all possible choices of B.

A plays mixed; pretend A goes first

$$\textbf{[B plays 1]} \qquad p \cdot (2) + (1-p) \cdot (-3) = 5p - 3 \tag{57}$$
$$\textbf{[B plays 2]} \qquad p \cdot (-3) + (1-p) \cdot (4) = -7p + 4 \tag{58}$$

remember that the numerical values (2, -3, 4) above are still from the perspective of A (negative is good for B). B will want to choose action $b \in \text{Actions}(b)$ such that

$$b = \arg\min\{5p{-}3, -7p{+}4\}$$

Notice that both options are linear functions of p, and that whatever the value of p, it is always the same for both of the linear functions[22]. If we plot these linear functions, we can find the possible line segments (rays, technically) where this minimum lies[23]. Since B is trying to minimize, we (re: player A) know that B will choose the smallest of these two options. Therefore, if player A is going first (and wants to *maximize*), while knowing that B will minimize, it should choose p that results in the largest value *in the minimum region* of our two linear functions – *this turns out to be the point where the functions intersect.* Therefore, we can compute the optimal choice of p for player A by setting both linear equations equal to each other and solving for p.

How does the aforementioned analysis change if we pretend B goes first **[1:02:00]**? I guess it doesn't change anything.

A plays mixed; pretend B goes first

Minimax Theorem [von Neumann, 1928]

For every simultaneous two-player zero-sum game with a finite number of actions:

$$\max_{\pi_A} \min_{\pi_B} V(\pi_A, \pi_B) = \min_{\pi_B} \max_{\pi_A} V(\pi_A, \pi_B) \tag{59}$$

*where pi_A, π_B range over **mixed strategies**.*

[22] There is only one p. It is "shared". Don't overthink it.

[23] Red marker area at **[1:00:47]**

Non Zero-Sum Games [1:06:00]. Difference of utility functions:

- (Zero-Sum) Competitive games: minimax (linear programming)
- Collaborative games: pure maximization (plain search)

Example zero-sum game is *Prisoner's dilemma* **[1:07:00]**. One difference is that now we need to denote which player's perspective to use with V (before we always were using A's perspective). Now, we use $V_p(\cdot)$ to denote the value as perceived by player p.

Nash Equilibrium

> *A* ***Nash equilibrium*** *is* (π_A^*, π_B^*) *such that no player has an incentive to change his/her strategy:*
>
> $$V_A(\pi_A^*, \pi_B^*) \geq V_A(\pi_A, \pi_B^*) \quad \forall \pi_A \tag{60}$$
> $$V_B(\pi_A^*, \pi_B^*) \geq V_B(\pi_A^*, \pi_B) \quad \forall \pi_B \tag{61}$$
>
> ***Nash's existence theorem****: In any finite-player game with finite number of actions, there exists* ***at least one*** *Nash equilibrium.*

Lectures **May 7, 2019**

CSPs 1

Table of Contents Local *Written by Brandon McKinzie*

Map coloring problem starts at **[13:00]**. Formulate as a search problem where ~~states~~ nodes represent individual countries and edges denote that they are touching (neighbors). A *state* is a partial assignment of color to the set of countries (nodes). Actions are assigning a particular country a color.

Variable-Based Models [20:48]. AKA graphical models (yay). Here, solutions are assignments to [sets of] variables (**modeling**), while decisions about variable ordering, etc., are chosen via **inference**.

Factor Graphs [22:40]. You know all this already. Circles represent variables. They are connected to squares, which denote factors. Scope of factor is set of variables it depends on. Arity of factor is number of variables in its scope. Each assignment of variables has a weight equal to the product of factors over the assignment (weird wording but ok) **[33:00]**.

Constraint Satisfaction Problem [34:43]

A CSP is a factor graph where all factors are ***contraints****:*

$$f_j(x) \in \{0, 1\} \qquad \forall j = 1, \ldots, m \tag{62}$$

The constraint is satisfied iff $f_j(x) = 1$.

An assignment x *is* ***consistent*** *iff* $Weight(x) = 1$ *(all constraints satisfied).*

Dynamic Ordering [37:40]. Every time we assign a value to a variable, there are additional factors we can evaluate. Formally, let x be some partial assignment, and X_i be a variable we are considering assigning a value to. We denote the **dependent factors** as $D(x, X_i)$: the set of factors depending on X_i and x but not on unassigned variables **[42:10]**.

Backtracking Search [43:00]

Inputs: partial assignment x, weight w (not sure why we can't compute this ourselves given x), and Domains set of possible values to assign to the unassigned variables.

1. If x is complete assignment $\longrightarrow$ update best and return
2. **Choose** unassigned variable X_i.
3. **Order** the possible values in Domains_i for chosen X_i.
4. For each value v in that order:
   - Compute
$$\delta \leftarrow \prod_{f_j \in D(x, X_i)} f_j\left(x \cup \{X_i : v\}\right) \tag{63}$$
   - If $\delta = 0$, continue
   - $\text{Domains}' \leftarrow \text{Domains}$ via **lookahead**
   - Recurse: $\text{Backtrack}(x \cup \{X_i : v\}, w \cdot \delta, \text{Domains}')$.

lookahead begins at **[47:25]**

Review

Contents

Review **April 07, 2019**

Discrete Math and Probability

Table of Contents Local *Written by Brandon McKinzie*

Sequences and Summations.

$$\sum_{k=0}^{n} ar^k \ (r{\neq}0) = \frac{ar^{n+1} - a}{r - 1}, \ r \neq 1 \tag{64}$$

$$\sum_{k=1}^{\infty} kx^{k-1}, \ |x|{<}1 = \frac{1}{(1-x)^2} \tag{65}$$

Recurrence Relations.

Bit Strings of Length Five

Find number of bit strings of length n, denoted as a_n, that do NOT have two consecutive 0s.

Suggested thought process:

1. I can probably define a_n as a **recurrence relation**.
2. I can think about this problem in terms of whether the last bit is a 1 or a 0:
   (a) Last bit is 1: there are a_{n-1} such bit strings satisfying the question.
   (b) Last bit is a 0: well then the n-2 bit can't be a zero, since that would violate the question. Therefore, there are a_{n-2} such bit strings.
3. $\therefore a_n = a_{n-1} + a_{n-2}$.

Themes:

- Splitting the problem into two sets corresponding to 0 and 1 somehow.
- Using the **sum rule**.

Patterns/Themes.

- **Bit Strings**.
  - Think in terms of cases: (1) the last bit is one, and (2) the last bit is zero.
- **Number of times until something happens**.
  - Probably follows a **geometric distribution**:

$$p(X{=}k) = (1-p)^{k-1}p \quad \text{for} \quad k = 1, 2, 3, \ldots \tag{66}$$

$$\mathbb{E}[X] = \frac{1}{p} \tag{67}$$

Review **May 04, 2019**

Course Synthesis

Table of Contents Local *Written by Brandon McKinzie*

A lot of the course topics have strange overlap and overloaded notation. Let's simplify.

Markov Decision Process

- ***Definition***:
  - *States* $\triangleq$ *Set*[*State*]
    - $s_{start} \in$ *States*.
    - *IsEnd*(s)
  - *Actions*$(s) \mapsto$ *Set*[*Action*]
  - $T(s, a, s') \mapsto [0, 1]$
  - *Reward*(s, a, s')
  - $0 \leq \gamma \leq 1$.
- ***Representation***: *graph with state nodes s and chance nodes c.*
  - *$s \to c$ edges are actions.*
  - *$c \to s$ edges are random outcomes.*
- ***Solution***: *a policy π.*
- ***Evaluation*** *of a policy π:*

$$U(s_1, a_1, s_2, \ldots, a_{T-1}, s_T) \triangleq \sum_{t=1}^{T-1} \gamma^{t-1} R(s_t, a_t, s_{t+1}) \tag{68}$$

$$V_\pi(s) \triangleq \begin{cases} 0 & \textit{if IsEnd(s)} \\ Q_\pi(s, \pi(s)) & \textit{otherwise} \end{cases} \tag{69}$$

$$Q_\pi(s, a) \triangleq \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma V_\pi(s') \right] \tag{70}$$

Now we need some way of *solving* the MDP: finding the optimal policy π_{opt} that maximizes our expected utility.

Solving MDPs

- *Value Iteration: converges to correct answer (π_{opt}), provided that either $\gamma < 1$ or MDP graph acyclic.*

Value Iteration

1. Initialize $V_\pi^{(0)}(s) \leftarrow 0$ for all states s.
2. For each iteration $t = 1, \ldots, t_{VI}$ and for each state s:

$$V_{opt}^{(t)}(s) \leftarrow \max_{a \in \textbf{Actions(s)}} \underbrace{\sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_{opt}^{(t-1)}(s') \right]}_{Q_{opt}^{(t-1)}(s,a)} \tag{71}$$

We want to choose t_{PE} such that

$$\max_{s \in \text{States}} \left| V_\pi^{(t)}(s) - V_\pi^{(t-1)}(s) \right| \leq \epsilon \tag{72}$$

Q-learning with function approximation

Define features $\boldsymbol{\phi}(s,a)$ and weights $\boldsymbol{w}$.

$$\hat{Q}_{opt}(s,a;\boldsymbol{w}) := \boldsymbol{w} \cdot \boldsymbol{\phi}(s,a) \tag{73}$$

On each (s,a,r,s'):

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \left[\underbrace{\hat{Q}_{opt}(s,a)}_{\text{prediction}} - \underbrace{\left(r + \gamma \hat{V}_{opt}(s') \right)}_{\text{target}} \right] \boldsymbol{\phi}(s,a) \tag{74}$$

with implied objective $(\hat{Q} - (r + \gamma \hat{V}))^2$.

Learning from Mistakes

Contents

Learning from Mistakes **April 21, 2019**

Homework 1: Foundations

Table of Contents Local *Written by Brandon McKinzie*

Problem 1: Optimization and Probability.

(a) **Mistake**: I didn't show the 2nd derivative was positive. **Lesson**: Always check 2nd deriv is positive when finding a minimum.

(a) **Mistake**: my interpretation of the case where w_i not guaranteed positive was incorrect (?) **Lesson**: No idea (TODO: figure out why I was wrong).

(b) **Mistake**: I got it right, but my approach was way more complicated than theirs. They just pulled the max to the left of the summation and refactored both f and g to maximize over $s_1, \ldots, s_d$ and used a more elegant proof by intuition. I basically brute-forced it. **Lesson**: get more comfortable moving around summations with maxes, and understand that when comparing two quantities (like f and g), you'll make it easier on yourself if you get them in the same form.

(c) **Mistake**: I got it right but, again, their solution was way simpler. Apparently, MDPs are relevant for solving these their way. They just used a simple expectation recursion relation, and I went full-on infinite series. **Lesson**: I need to get way more comfortable with recursive expectation value problems.