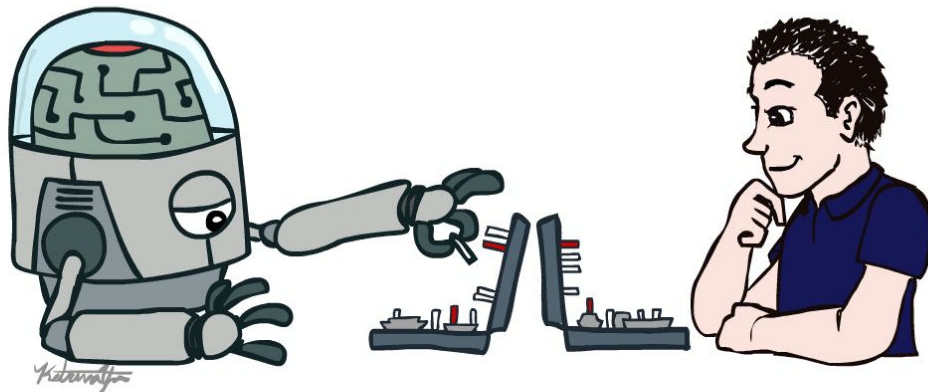
CS 188: Artificial Intelligence

Final Exam Review



Summer 2024: Eve Fleisig & Evgeny Pobachienko

University of California, Berkeley

(slides adapted from Nicholas Tomlin, Dan Klein, Pieter Abbeel, Anca Dragan, Stuart Russell)

In-Scope Mathematics

- Linear algebra:
  - Definition and properties of dot products
  - Composition of linear transformations is linear
- Vector calculus:
  - How to take partial derivatives (incl. chain rule, vector derivatives)
  - Solving optimization problems using derivatives (e.g., deriving MLE)
- Probability: definition of a probability distribution, random variables, joint and marginal distributions, conditional probabilities, Bayes’ rule, normalization

How about computing all the derivatives?

- Derivatives tables:

$$\frac{d}{dx}(a)=0$$

$$\frac{d}{dx}(x)=1$$

$$\frac{d}{dx}(au)=a\frac{du}{dx}$$

$$\frac{d}{dx}(u+v-w)=\frac{du}{dx}+\frac{dv}{dx}-\frac{dw}{dx}$$

$$\frac{d}{dx}(uv)=u\frac{dv}{dx}+v\frac{du}{dx}$$

$$\frac{d}{dx}\left(\frac{u}{v}\right)=\frac{1}{v}\frac{du}{dx}-\frac{u}{v^2}\frac{dv}{dx}$$

$$\frac{d}{dx}(u^n)=nu^{n-1}\frac{du}{dx}$$

$$\frac{d}{dx}(\sqrt{u})=\frac{1}{2\sqrt{u}}\frac{du}{dx}$$

$$\frac{d}{dx}\left(\frac{1}{u}\right)=-\frac{1}{u^2}\frac{du}{dx}$$

$$\frac{d}{dx}\left(\frac{1}{u^n}\right)=-\frac{n}{u^{n+1}}\frac{du}{dx}$$

$$\frac{d}{dx}[f(u)]=\frac{d}{du}[f(u)]\frac{du}{dx}$$

$$\frac{d}{dx}[\ln u]=\frac{d}{dx}[\log_e u]=\frac{1}{u}\frac{du}{dx}$$

$$\frac{d}{dx}[\log_a u]=\log_a e\frac{1}{u}\frac{du}{dx}$$

$$\frac{d}{dx}e^u=e^u\frac{du}{dx}$$

$$\frac{d}{dx}a^u=a^u\ln a\frac{du}{dx}$$

$$\frac{d}{dx}(u^v)=vu^{v-1}\frac{du}{dx}+\ln u\ \ u^v\frac{dv}{dx}$$

[source:

How about computing all the derivatives?

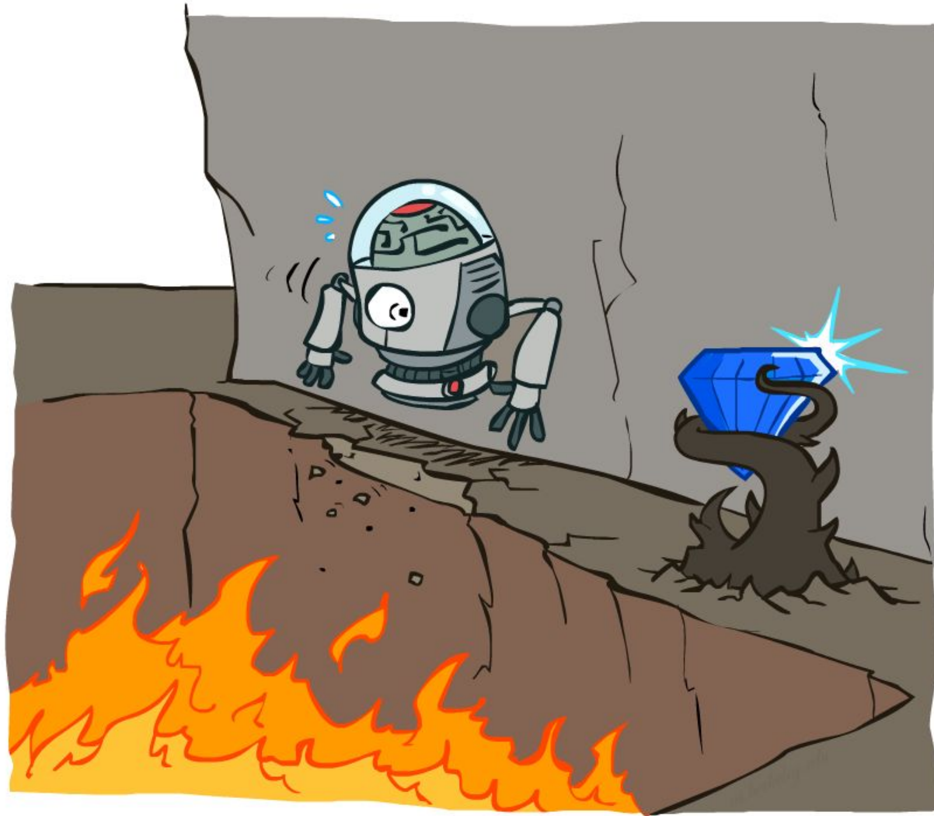
- But neural net f is never one of those?
  - No problem: CHAIN RULE:

If $$f(x) = g(h(x))$$

Then $$f'(x) = g'(h(x))h'(x)$$

Derivatives can be computed by following well-defined procedures

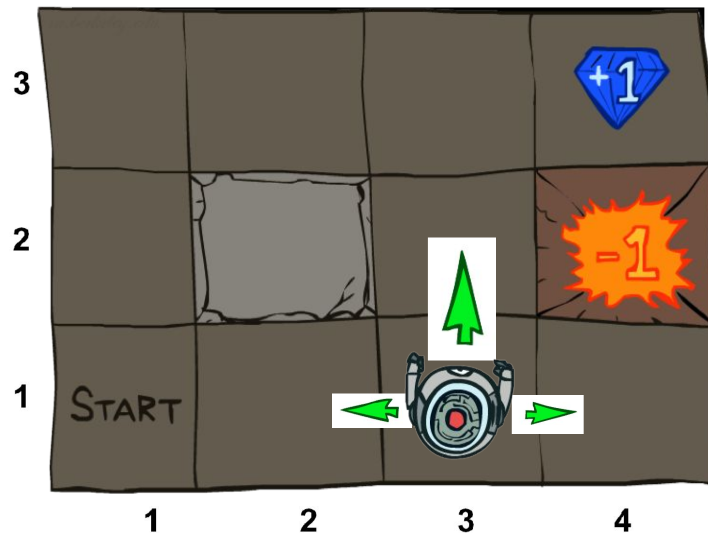
Markov Decision Processes



Markov Decision Processes

- An MDP is defined by:
  - A set of states $s \in S$
  - A set of actions $a \in A$
  - A transition function T(s, a, s')
    - Probability that a from s leads to s', i.e., P(s'| s, a)
    - Also called the model or the dynamics
  - A reward function R(s, a, s')
    - Sometimes just R(s) or R(s')
  - A start state
  - Maybe a terminal state

- We care about:
  - Policy = choice of actions for each state
  - Utility = sum of (discounted) rewards



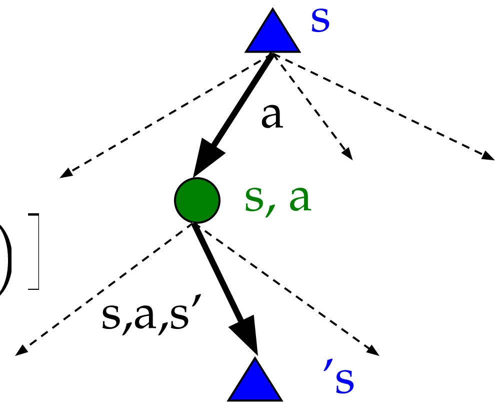
Values of States: Bellman Equation

- Recursive definition of value:

$$V^*(s) = \max_a Q^*(s,a)$$

$$Q^*(s,a) = \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma V^*(s')\right]$$

$$V^*(s) = \max_a \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma V^*(s')\right]$$



Value Iteration

- Bellman equations characterize the optimal values:

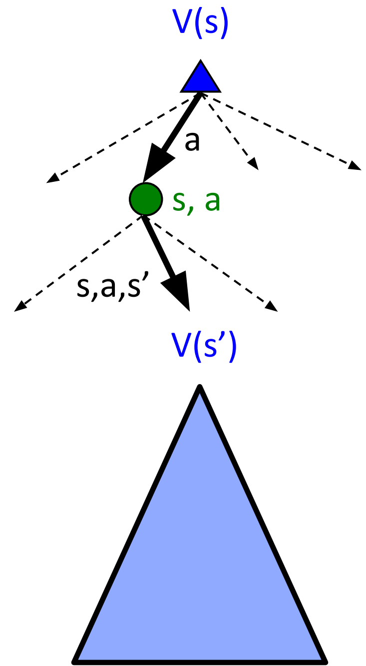
$$V^*(s) = \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V^*(s')\right]$$

- Value iteration computes them:

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_k(s')\right]$$

"Bellman Update"

- Value iteration is just a fixed point solution method

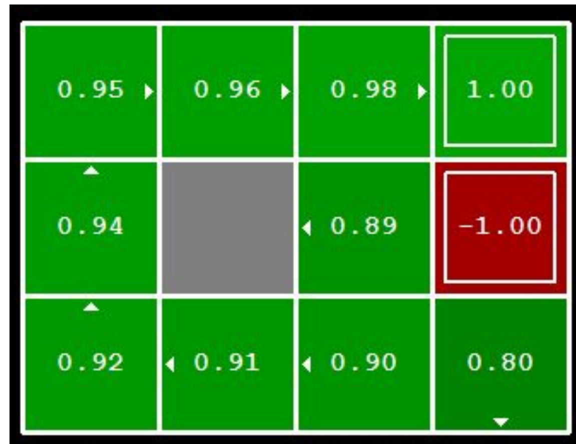


Policy Extraction from Values

- Let's imagine we have the optimal values V*(s)

- How should we act?
  - It's not obvious!

- We need to do a mini-expectimax (one step)



$$\pi^*(s) = \arg\max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$

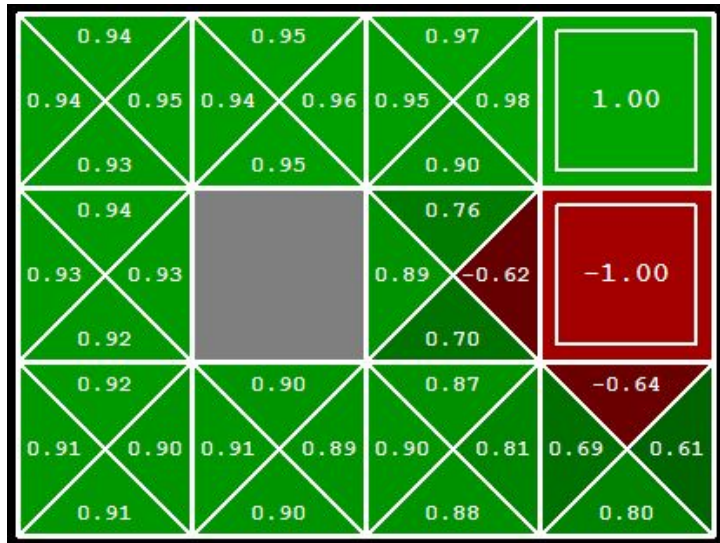
- This is called policy extraction, since it gets the policy implied by the values

Policy Extraction from Q-Values

- Let's imagine we have the optimal q-values:

- How should we act?
  - Completely trivial to decide!

$$\pi^*(s) = \arg\max_a Q^*(s,a)$$



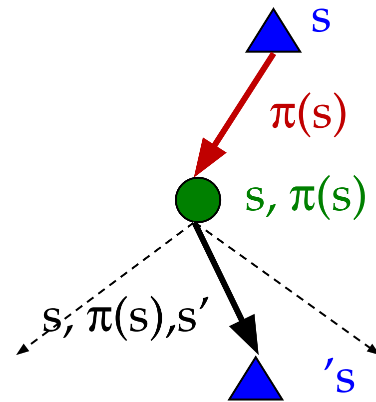
- Important lesson: actions are easier to select from q-values than values!

Policy Evaluation

- How do we calculate the V's for a fixed policy π?
- Idea 1: Turn recursive Bellman equations into updates (like value iteration)

$$V_0^\pi(s) = 0$$

$$V_{k+1}^\pi(s) \leftarrow \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V_k^\pi(s')]$$



- Efficiency: $O(S^2)$ per iteration
- Idea 2: Without the maxes, the Bellman equations are just a linear system
  - Solve the system of equations

Policy Iteration

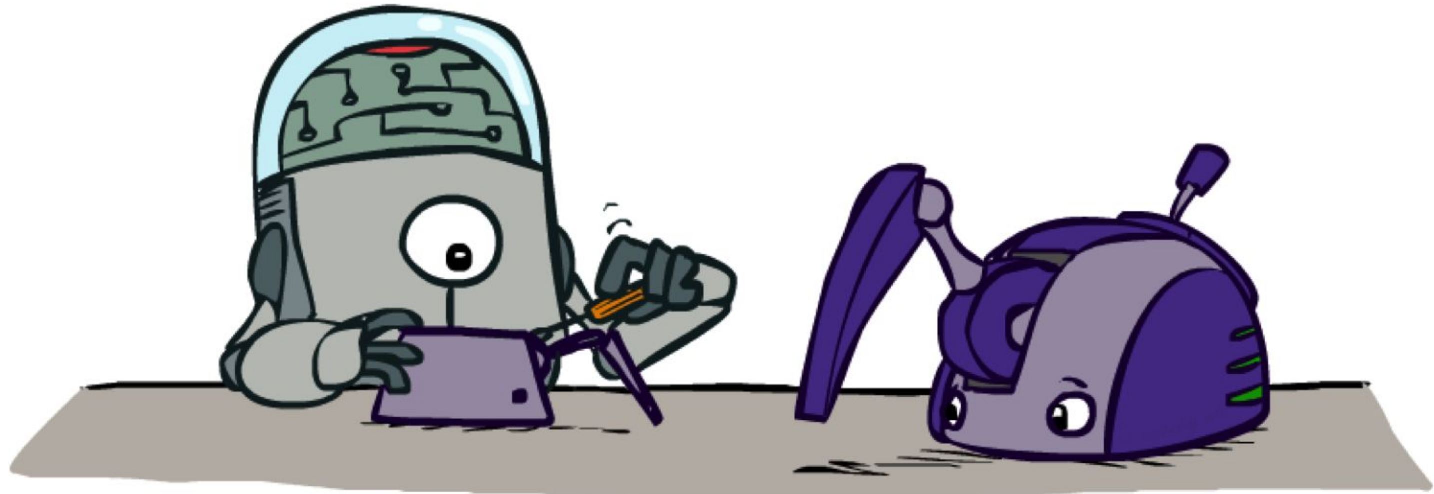
- Evaluation: For fixed current policy π, find values with policy evaluation:
  - Iterate until values converge:

$$V_{k+1}^{\pi_i}(s) \leftarrow \sum_{s'} T(s, \pi_i(s), s') \left[R(s, \pi_i(s), s') + \gamma V_k^{\pi_i}(s')\right]$$

- Improvement: For fixed values, get a better policy using policy extraction
  - One-step look-ahead:

$$\pi_{i+1}(s) = \arg\max_a \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma V^{\pi_i}(s')\right]$$

Reinforcement Learning



Map of Reinforcement Learning

Known MDP: Offline Solution

Goal	Technique
Compute V*, Q*, π*	Value / policy iteration
Evaluate a fixed policy π	Policy evaluation

Unknown MDP: Model-Based

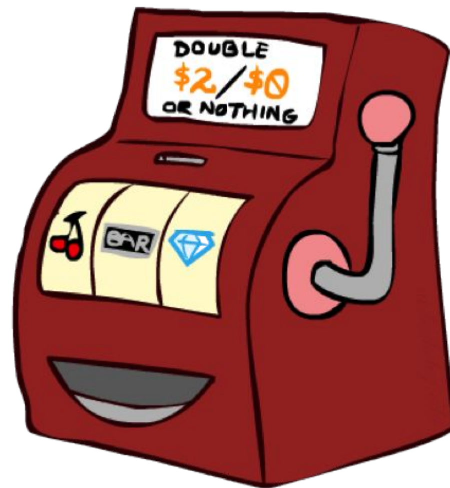
Goal	Technique
Compute V*, Q*, π*	VI/PI on approx. MDP
Evaluate a fixed policy π	PE on approx. MDP

Unknown MDP: Model-Free

Goal	Technique
Compute V*, Q*, π*	Q-learning
Evaluate a fixed policy π	Value Learning

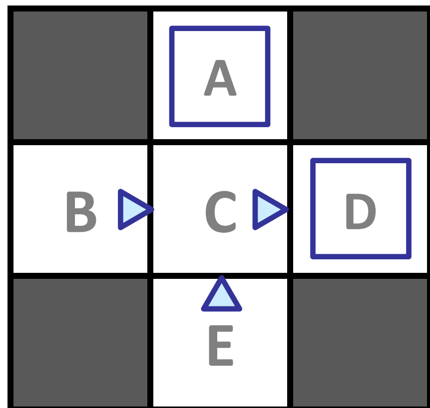
Direct Evaluation

- Goal: Compute values for each state under π
- Idea: Average together observed sample values
  - Act according to π
  - Every time you visit a state, write down what the sum of discounted rewards turned out to be
  - Average those samples
- This is called direct evaluation



Direct Evaluation

Input Policy π



Assume: γ = 1

Observed Episodes (Training)

Episode 1

B, east, C, -1
C, east, D, -1
D, exit, x, +10

Episode 2

B, east, C, -1
C, east, D, -1
D, exit, x, +10

Episode 3

E, north, C, -1
C, east, D, -1
D, exit, x, +10

Episode 4

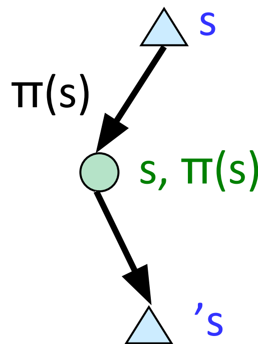
E, north, C, -1
C, east, A, -1
A, exit, x, -10

Output Values

	A: -10	
B: +8	C: +4	D: +10
	E: -2	

Temporal Difference Learning

- Big idea: learn from every experience!
  - Update V(s) each time we experience a transition (s, a, s', r)
  - Likely outcomes s' will contribute updates more often



- Temporal difference learning of values
  - Policy still fixed, still doing evaluation!
  - Move values toward value of whatever successor occurs: running average

Sample of V(s): $sample = R(s, \pi(s), s') + \gamma V^{\pi}(s')$

Update to V(s): $V^{\pi}(s) \leftarrow (1 - \alpha) V^{\pi}(s) + (\alpha) sample$

Same update: $V^{\pi}(s) \leftarrow V^{\pi}(s) + \alpha(sample - V^{\pi}(s))$

Q-Learning

- Q-Learning: sample-based Q-value iteration

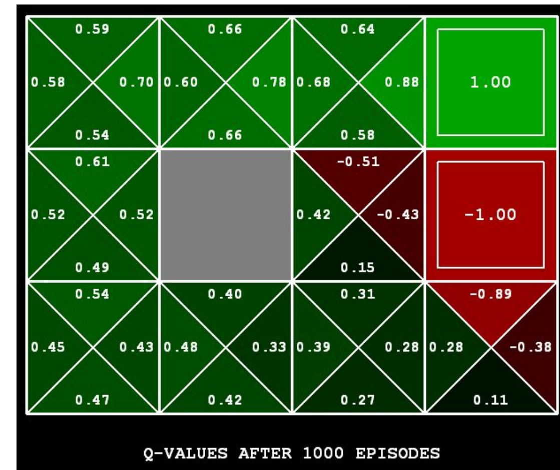
$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma \max_{a'} Q_k(s',a') \right]$$

- Learn Q(s,a) values as you go
  - Receive a sample (s,a,s',r)
  - Consider your old estimate: $Q(s,a)$
  - Consider your new sample estimate:

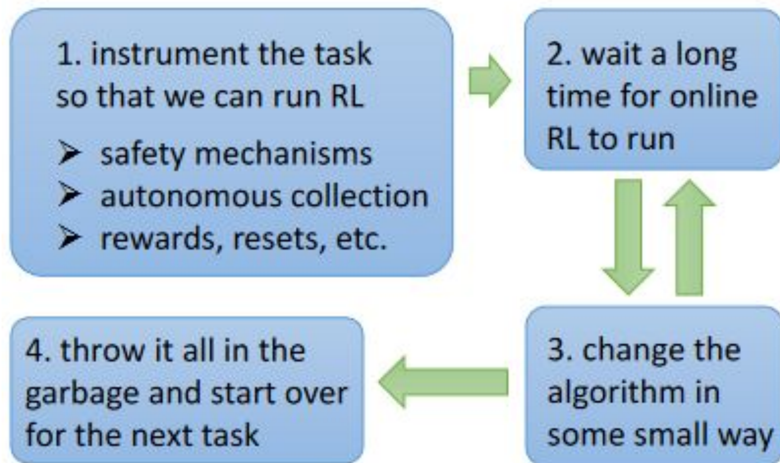
$$sample = R(s,a,s') + \gamma \max_{a'} Q(s',a')$$

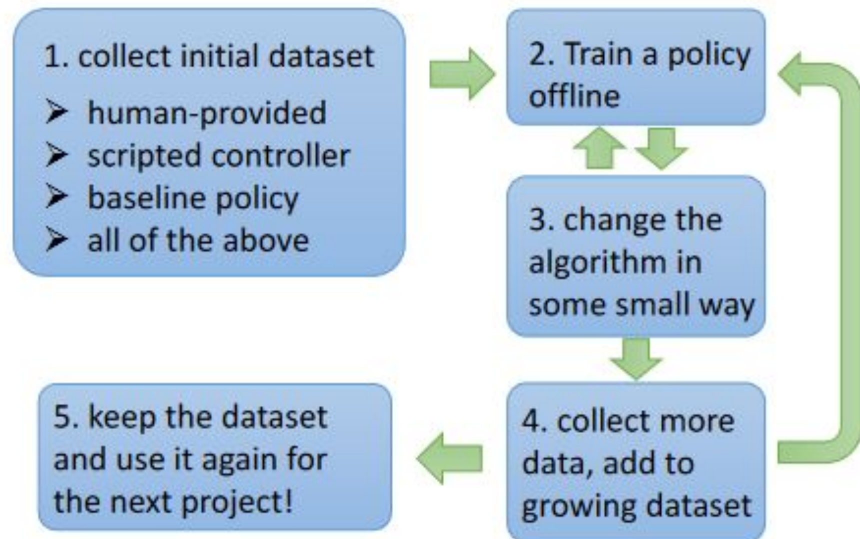
  - Incorporate the new estimate into a running average:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + (\alpha)\left[sample\right]$$



Why Off-Policy





Approximate Q-Learning

$$Q(s,a) = w_1 f_1(s,a) + w_2 f_2(s,a) + \ldots + w_n f_n(s,a)$$

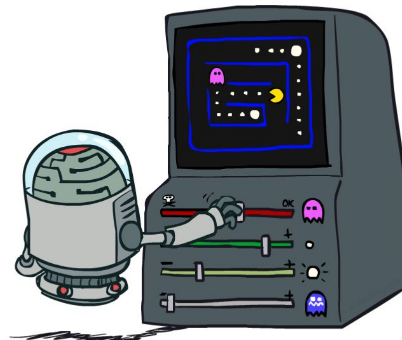
- Q-learning with linear Q-functions:

$$\text{transition } = (s, a, r, s')$$

$$\text{difference} = \left[r + \gamma \max_{a'} Q(s', a') \right] - Q(s,a)$$

$$Q(s,a) \leftarrow Q(s,a) + \alpha \, [\text{difference}] \qquad \text{Exact Q's}$$

$$w_i \leftarrow w_i + \alpha \, [\text{difference}] \, f_i(s,a) \qquad \text{Approximate Q's}$$



- Intuitive interpretation:
  - Adjust weights of active features
  - E.g., if something unexpectedly bad happens, blame the features that were on: disprefer all states with that state's features

- Formal justification: online least squares

How to Explore?

- Several schemes for forcing exploration
  - Simplest: random actions (ε-greedy)
    - Every time step, flip a coin
    - With (small) probability ε, act randomly
    - With (large) probability 1-ε, act on current policy
  - Problems with random actions?
    - You do eventually explore the space, but keep thrashing around once learning is done
    - One solution: lower ε over time
    - Another solution: exploration functions

Exploration Functions

- When to explore?
  - Random actions: explore a fixed amount
  - Better idea: explore areas whose badness is not (yet) established, eventually stop exploring

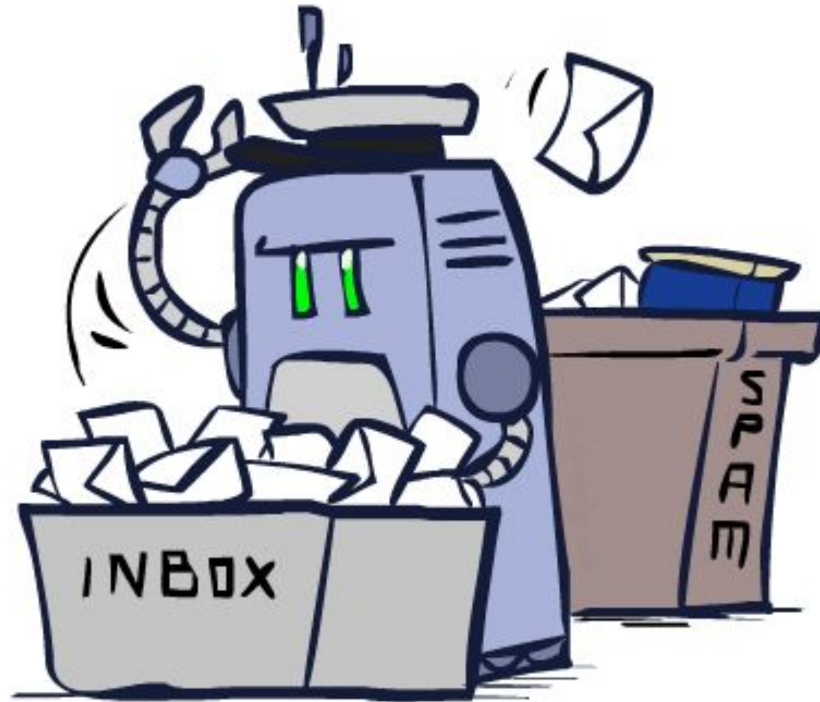
- Exploration function
  - Takes a value estimate u and a visit count n, and returns an optimistic utility, e.g. $f(u,n) = u + k/n$

Regular Q-Update: $Q(s,a) \leftarrow_\alpha R(s,a,s') + \gamma \max_{a'} Q(s',a')$

Modified Q-Update: $Q(s,a) \leftarrow_\alpha R(s,a,s') + \gamma \max_{a'} f(Q(s',a'), N(s',a'))$

  - Note: this propagates the “bonus” back to states that lead to unknown states as well!

Machine Learning



Example: Digit Recognition

- Input: images / pixel grids
- Output: a digit 0-9

- Setup:
  - Get a large collection of example images, each labeled with a digit
  - Note: someone has to hand label all this data!
  - Want to learn to predict labels of new, future digit images

- Features: The attributes used to make the digit decision
  - Pixels: (6,8)=ON
  - Shape Patterns: NumComponents, AspectRatio, NumLoops
  - …
  - Features are increasingly induced rather than crafted

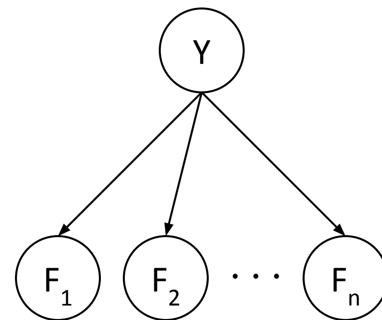
Naïve Bayes for Digits

- Naïve Bayes: Assume all features are independent effects of the label

- Simple digit recognition version:
  - One feature (variable) F_{ij} for each grid position <i,j>
  - Feature values are on / off, based on whether intensity is more or less than 0.5 in underlying image
  - Each input maps to a feature vector, e.g.

$$\rightarrow \langle F_{0,0} = 0 \;\; F_{0,1} = 0 \;\; F_{0,2} = 1 \;\; F_{0,3} = 1 \;\; F_{0,4} = 0 \;\; \ldots F_{15,15} = 0 \rangle$$

  - Here: lots of features, each is binary valued

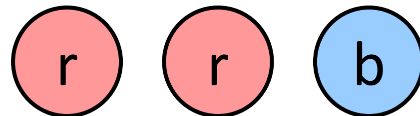


- Naïve Bayes model: $$P(Y|F_{0,0} \ldots F_{15,15}) \propto P(Y) \prod_{i,j} P(F_{i,j}|Y)$$

- What do we need to learn?

Deriving MLEs

- **Model:** X red blue



- **Data**: draw N balls. N_r come up red, N_b come up blue
  - Dataset: $D = \{x_1, \dots, x_n\}$
  - Ball draws are independent and identically distributed (i.i.d.):

$$P(D \mid \theta) = \prod_i P(x_i \mid \theta) = \prod_i P_\theta(x_i) = \theta^{N_r} \cdot (1-\theta)^{N_b}$$

- **Maximum likelihood estimation**: find θ that maximizes $P(D \mid \theta)$

$$\theta = \underset{\theta}{\operatorname{argmax}}\, P(D \mid \theta) = \underset{\theta}{\operatorname{argmax}} \log P(D \mid \theta)$$

  - Approach: take derivative and set to 0

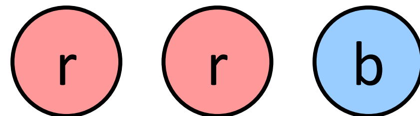
Parameter Estimation with Maximum Likelihood

- Estimating the distribution of a random variable
- Use training data (learning!)
  - For each outcome x, look at the **empirical rate** of that value:

$$P_{ML} = \frac{\text{count}(x)}{\text{total samples}}$$

X	red	blue

- Example: probability of x=red given the training data:



$$P_{ML}(r) = \frac{2}{3}$$

- This estimate maximizes the **likelihood of the data** for the parametric model:

$$L(\theta) = P(r, r, b \mid \theta) = P_\theta(r) \cdot P_\theta(r) \cdot P_\theta(b)$$
$$= \theta^2 \cdot (1 - \theta)$$

Parameter Estimation with Maximum Likelihood

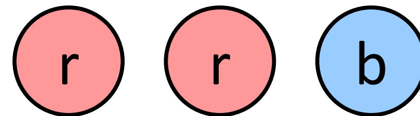
- Likelihood function:

$$L(\theta) = \mathrm{P}(\mathrm{r}, \mathrm{r}, \mathrm{b} \mid \theta) = \mathrm{P}_\theta(r) \cdot \mathrm{P}_\theta(r) \cdot \mathrm{P}_\theta(b)$$
$$= \theta^2 \cdot (1 - \theta)$$
$$= \theta^2 - \theta^3$$

X	red	blue

- MLE: find the θ that maximizes data likelihood

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}}\, L(\theta)$$



- Approach: take derivatives and set to 0

$$\frac{\partial L(\theta)}{\partial \theta} = 2\theta - 3\theta^2$$
$$= \theta(2 - 3\theta)$$

- Find the maximum at $\theta = \frac{2}{3}$

Deriving MLEs

- **Maximum likelihood estimation**: find θ that maximizes $P(D \mid \theta)$

$$\theta = \underset{\theta}{\operatorname{argmax}}\, P(D \mid \theta) = \underset{\theta}{\operatorname{argmax}} \log P(D \mid \theta)$$

$$\begin{aligned}\frac{\partial}{\partial\theta}\log P(D \mid \theta) &= \frac{\partial}{\partial\theta}[N_r \log(\theta) + N_b \log(1-\theta)] \\ &= N_r \frac{\partial}{\partial\theta}\log(\theta) + N_b \frac{\partial}{\partial\theta}\log(1-\theta) \\ &= N_r \frac{1}{\theta} - N_b \frac{1}{1-\theta} \\ &= 0\end{aligned}$$

Multiply by $\theta(1-\theta)$:

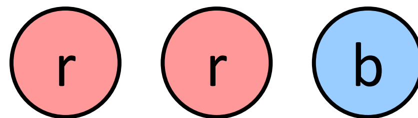
$$N_r(1-\theta) - N_b\theta = 0$$

$$N_r - \theta(N_r + N_b) = 0$$

$$\hat{\theta} = \frac{N_r}{N_r + N_b}$$

Regularization: Smoothing

- Laplace's estimate:
  - Pretend you saw every outcome once more than you actually did



$$P_{LAP}(x) = \frac{c(x)+1}{\sum_x [c(x)+1]}$$

$$= \frac{c(x)+1}{N+|X|}$$

$$P_{ML}(X) = \left\langle \frac{2}{3}, \frac{1}{3} \right\rangle$$

$$P_{LAP}(X) = \left\langle \frac{3}{5}, \frac{2}{5} \right\rangle$$

- **This is no longer a maximum likelihood estimate**

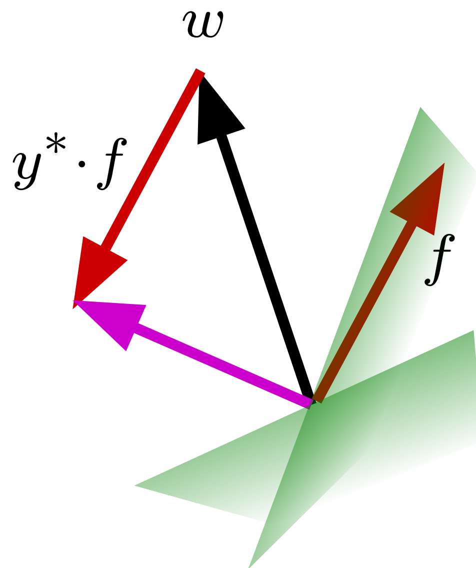
Learning: Binary Perceptron

- Start with weights = 0
- For each training instance:
  - Classify with current weights

$$y = \begin{cases} +1 & \text{if } \; w \cdot f(x) \geq 0 \\ -1 & \text{if } \; w \cdot f(x) < 0 \end{cases}$$

  - If correct (i.e., y=y*), no change!
  - If wrong: adjust the weight vector by adding or subtracting the feature vector. Subtract if y* is -1.

$$w = w + y^* \cdot f$$



Learning: Multiclass Perceptron

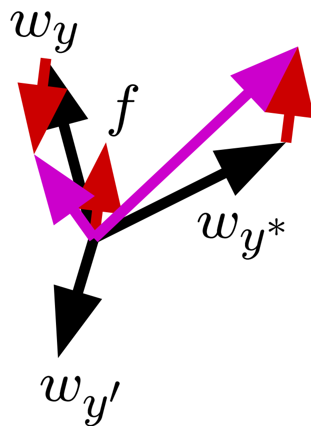
- Start with all weights = 0
- Pick up training examples one by one
- Predict with current weights

$$y = \arg\max_y \; w_y \cdot f(x)$$

- If correct, no change!
- If wrong: lower score of wrong answer, raise score of right answer

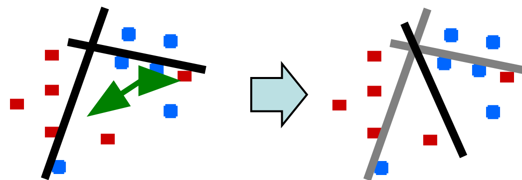
$$w_y = w_y - f(x)$$

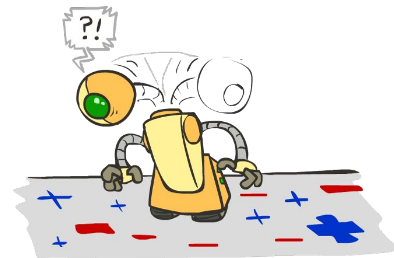
$$w_{y^*} = w_{y^*} + f(x)$$



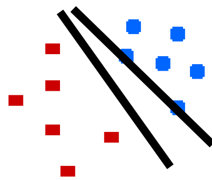
Problems with the Perceptron

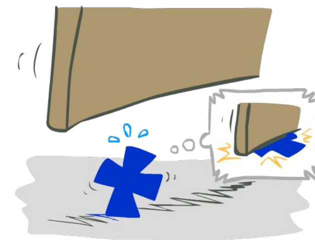
- Noise: if the data isn't separable, weights might thrash
  - Averaging weight vectors over time can help (averaged perceptron)



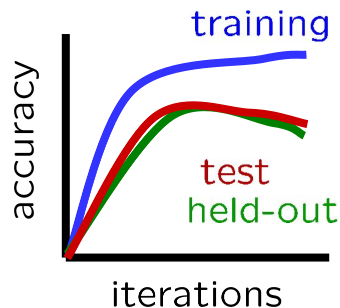


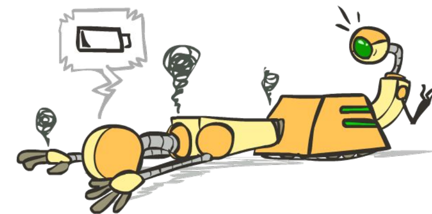
- Mediocre generalization: finds a "barely" separating solution





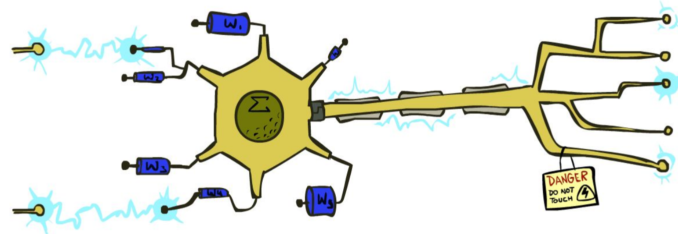
- Overtraining: test / held-out accuracy usually rises, then falls
  - Overtraining is a kind of overfitting





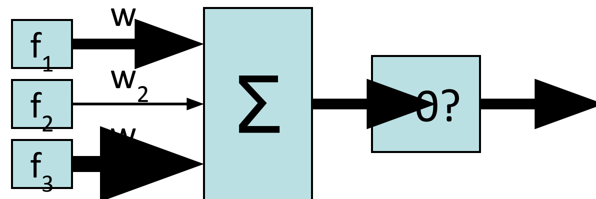
Reminder: Linear Classifiers

- Inputs are feature values
- Each feature has a weight
- Sum is the activation



$$\text{activation}_w(x) = \sum_i w_i \cdot f_i(x) = w \cdot f(x)$$

- If the activation is:
  - Positive, output +1
  - Negative, output -1

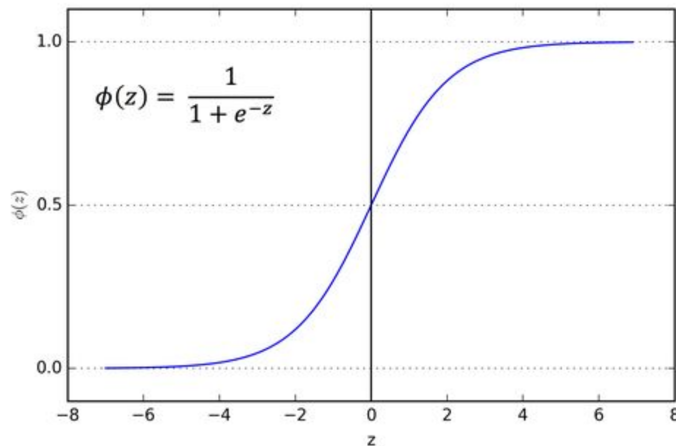


How to get probabilistic decisions?

- Activation: $z = w \cdot f(x)$
- If $z = w \cdot f(x)$ very positive: want probability going to 1
- If $z = w \cdot f(x)$ very negative: want probability going to 0

- Sigmoid function

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



Best w?

- Maximum likelihood estimation:

$$\max_w \quad ll(w) = \max_w \quad \sum_i \log P(y^{(i)}|x^{(i)};w)$$

with:

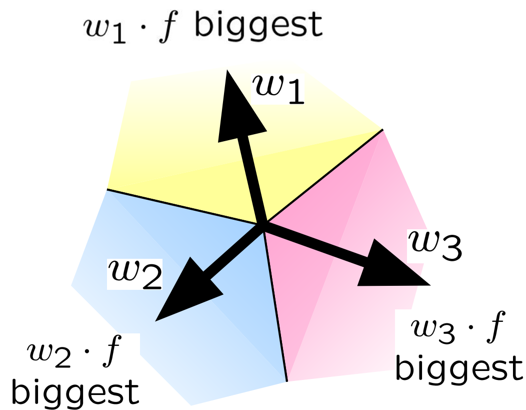
$$P(y^{(i)} = +1|x^{(i)};w) = \frac{1}{1+e^{-w\cdot f(x^{(i)})}}$$

$$P(y^{(i)} = -1|x^{(i)};w) = 1 - \frac{1}{1+e^{-w\cdot f(x^{(i)})}}$$

= Logistic Regression

Multiclass Logistic Regression

- Multi-class linear classification
  - A weight vector for each class: w_y
  - Score (activation) of a class y: $w_y \cdot f(x)$
  - Prediction w/highest score wins: $y = \arg\max_y \; w_y \cdot f(x)$



- How to make the scores into probabilities?

$$\underbrace{z_1, z_2, z_3}_{\text{original activations}} \rightarrow \underbrace{\frac{e^{z_1}}{e^{z_1} + e^{z_2} + e^{z_3}}, \frac{e^{z_2}}{e^{z_1} + e^{z_2} + e^{z_3}}, \frac{e^{z_3}}{e^{z_1} + e^{z_2} + e^{z_3}}}_{\text{softmax activations}}$$

Best w?

- Maximum likelihood estimation:

$$\max_w \quad ll(w) = \max_w \quad \sum_i \log P(y^{(i)}|x^{(i)};w)$$

with:

$$P(y^{(i)}|x^{(i)};w) = \frac{e^{w_{y^{(i)}} \cdot f(x^{(i)})}}{\sum_y e^{w_y \cdot f(x^{(i)})}}$$

= Multi-Class Logistic Regression

Batch Gradient Ascent

$$\max_w \quad ll(w) = \max_w \quad \underbrace{\sum_i \log P(y^{(i)}|x^{(i)};w)}_{g(w)}$$

- init w
- for iter = 1, 2, …

$$w \leftarrow w + \alpha * \sum_i \nabla \log P(y^{(i)}|x^{(i)};w)$$

Stochastic Gradient Ascent

$$\max_w \; ll(w) = \max_w \; \sum_i \log P(y^{(i)} | x^{(i)}; w)$$

Observation: once gradient on one training example has been computed, might as well incorporate before computing next one

- init w
- for iter = 1, 2, …
  - pick random j

$$w \leftarrow w + \alpha * \nabla \log P(y^{(j)} | x^{(j)}; w)$$

Mini-batch Gradient Ascent

$$\max_w \quad ll(w) = \max_w \quad \sum_i \log P(y^{(i)}|x^{(i)}; w)$$

Observation: gradient over small set of training examples (=mini-batch) can be computed in parallel, might as well do that instead of a single one

- init w
- for iter = 1, 2, …
  - pick random subset of training examples J

$$w \leftarrow w + \alpha * \sum_{j \in J} \nabla \log P(y^{(j)}|x^{(j)}; w)$$

Beyond SGD: Second-Order Derivatives

Newton's Method (in 1D):

- Want to optimize: $\max_{\theta} f(\theta)$
- Apply Taylor expansion:

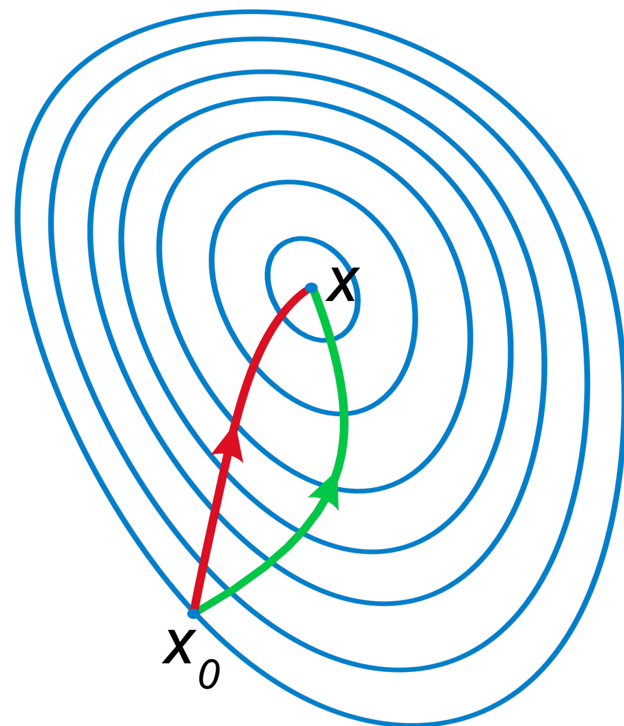
$$f(\theta + h) = f(\theta) + f'(\theta)h + \frac{1}{2}f''(\theta)h^2$$

- Find value of t that maximizes this:

$$0 = \frac{\partial}{\partial h}\left[f(\theta) + f'(\theta)h + \frac{1}{2}f''(\theta)h^2\right]$$
$$= f'(\theta) + f''(\theta)h$$

- Rearrange terms to get update:

$$h = -\frac{f'(\theta)}{f''(\theta)} \qquad \theta_{t+1} = \theta_t + h = \theta_t - \frac{f'(\theta)}{f''(\theta)}$$



These update equations out of scope for final exam; but high-level concepts are in scope

Beyond SGD: Momentum

- Potential issues with vanilla SGD:
  - Can take a long time to converge if the learning rate is too low
  - Can bounce around in “ravines” without making much progress toward a local optimum

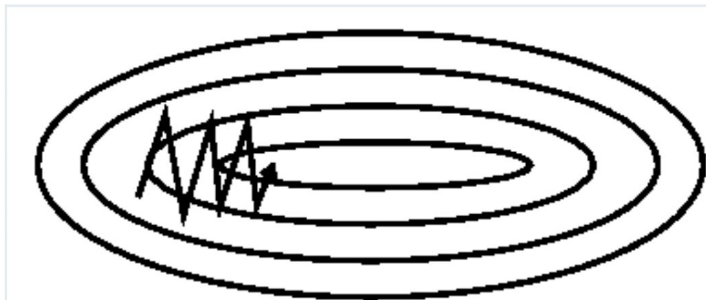


Image 2: SGD without momentum

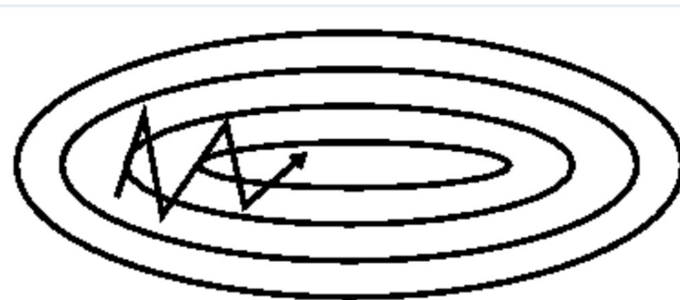


Image 3: SGD with momentum

Beyond SGD: Adaptive Learning Rates

- Recall: learning rates
  - Determines how much we update weights in the direction of the gradient
  - Often: want to set this in terms of how much it updates the weights
  - Often: want to lower learning rate over time (*learning rate scheduling*)

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta f(\theta_t)$$

- Key idea: different learning rates for each parameter
  - We can make larger or smaller updates depending on how important a feature is
  - Small updates for frequent features; big updates for rare features
  - This idea underlies: Adagrad, RMSProp, Adam, etc.

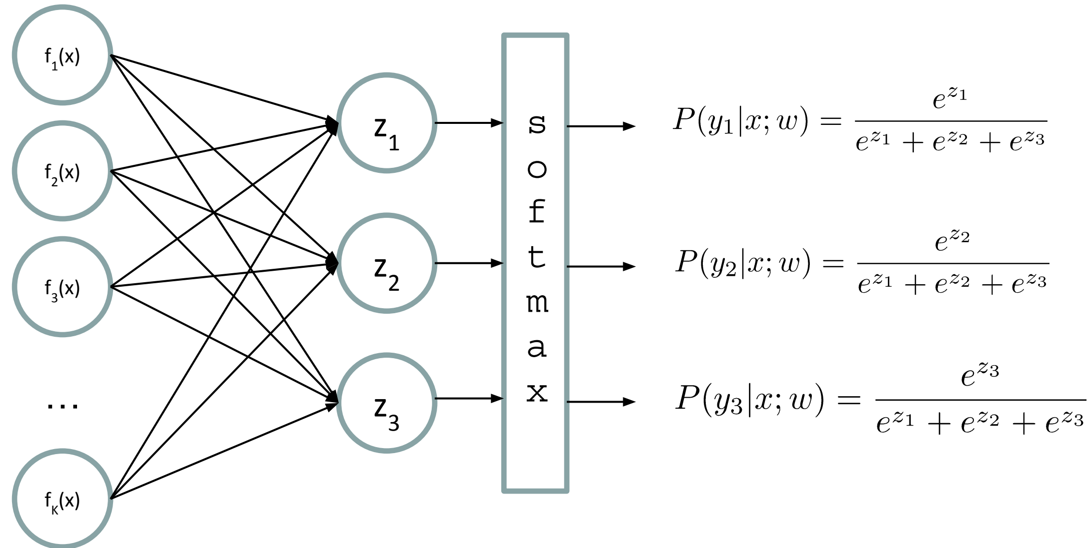
Summary: Key Ideas in Optimization

- Gradient descent
  - Batch: update based on the whole dataset
  - SGD: update based on a single randomly chosen training example
  - Minibatch: update based on *k* randomly chosen training examples

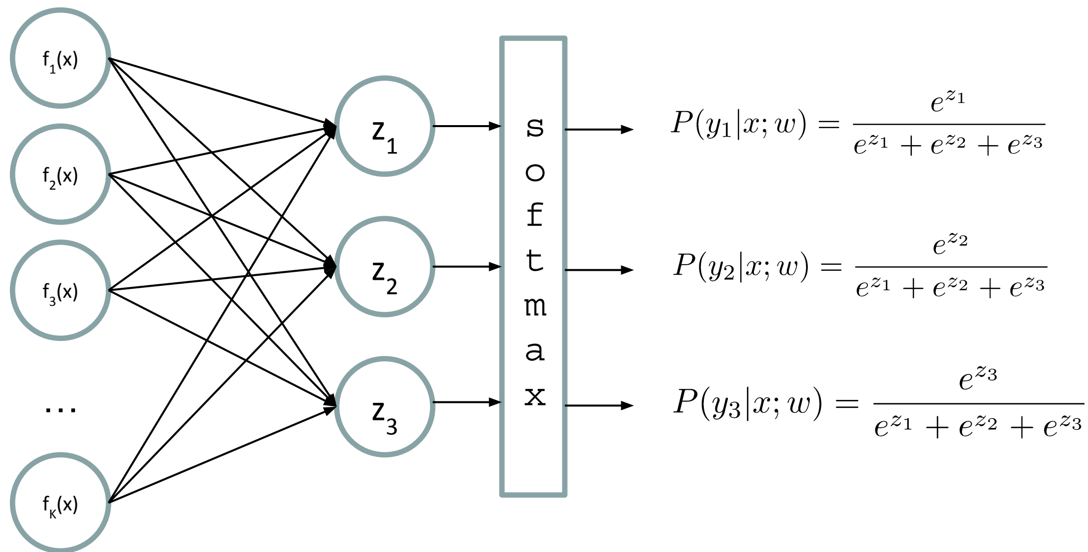
- More advanced approaches:
  - Second order optimization (e.g., Newton's method)
  - Momentum (Nesterov's accelerated gradient, Adam)
  - Adaptive learning rates (Adagrad, RMSProp, Adam, etc.)

Multi-class Logistic Regression

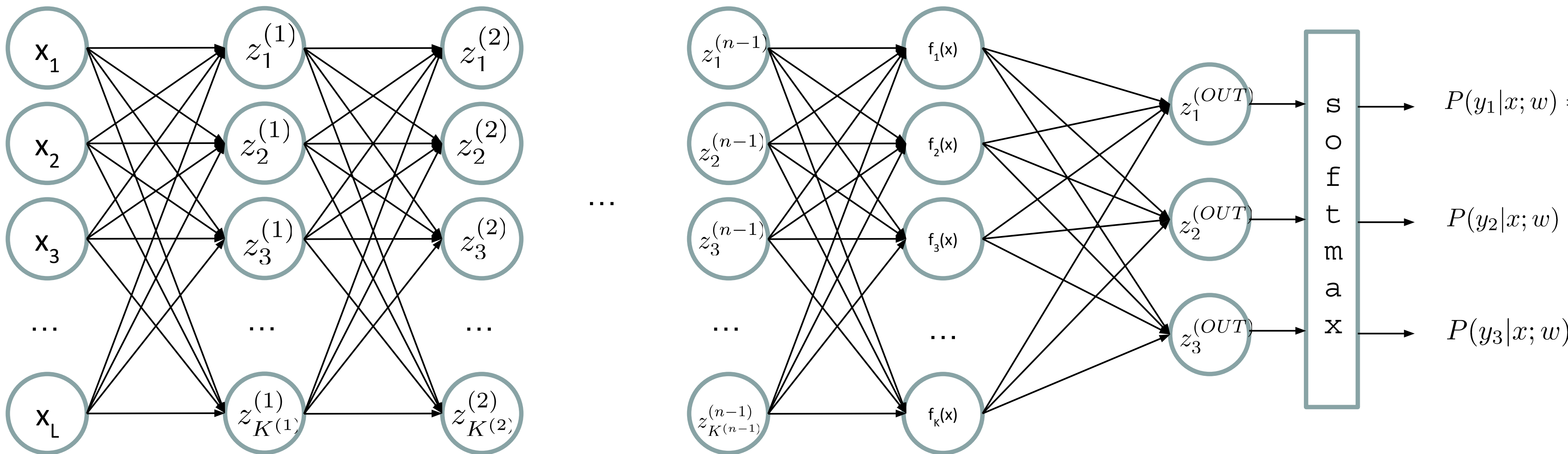
- = special case of neural network



Deep Neural Network = Also learn the features!

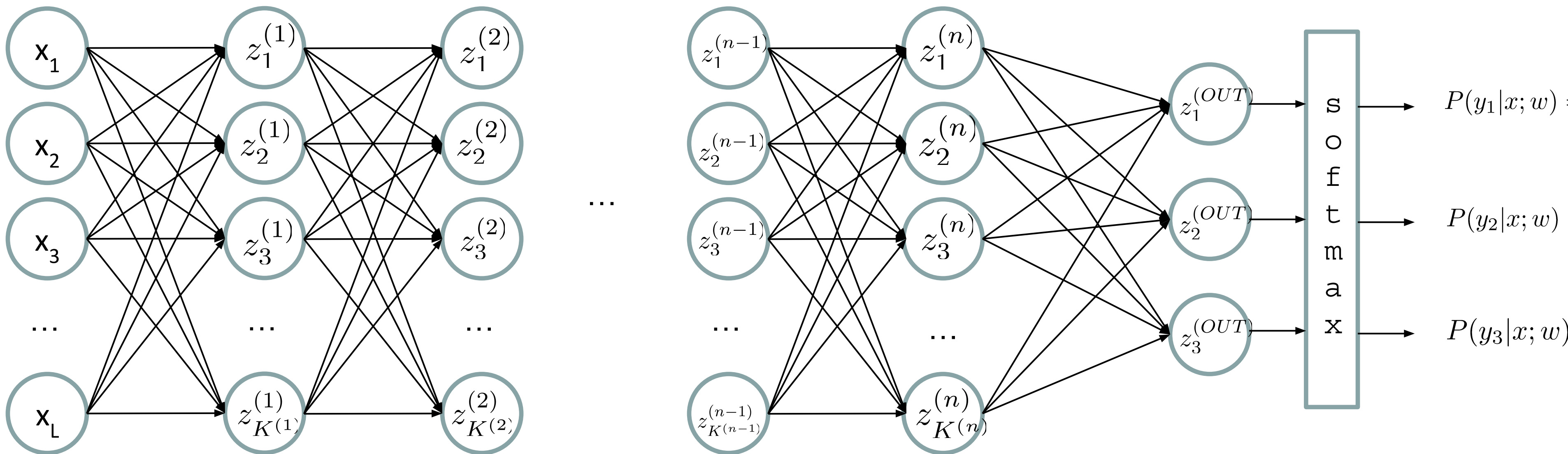


Deep Neural Network = Also learn the features!



$$z_i^{(k)} = g(\sum_j W_{i,j}^{(k-1,k)} z_j^{(k-1)})$$

Deep Neural Network = Also learn the features!

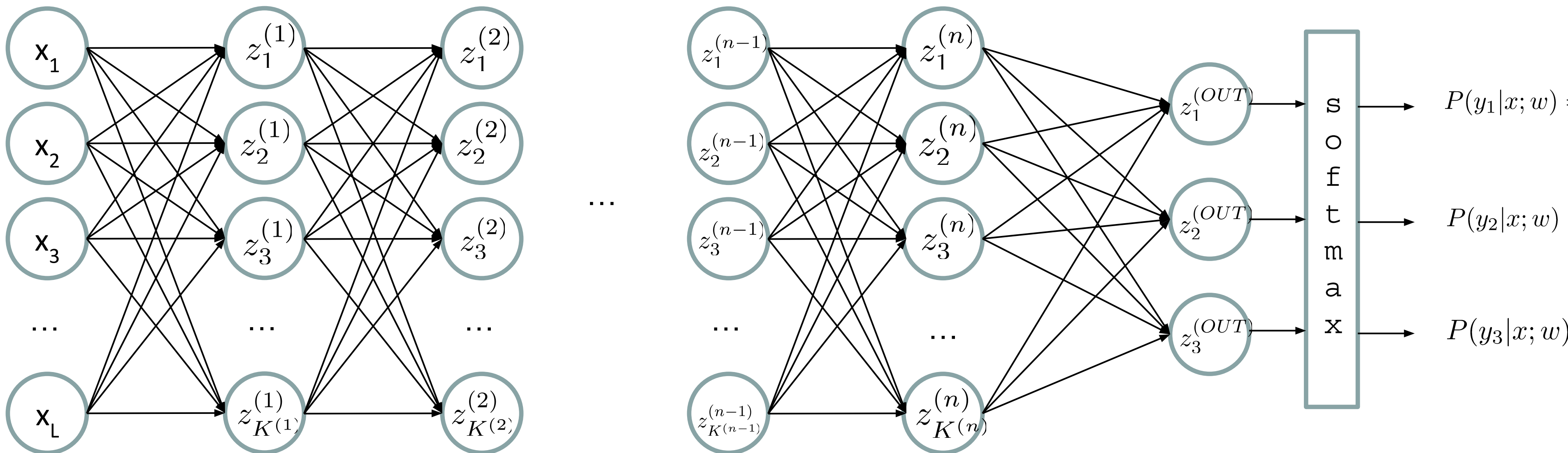


$$z_i^{(k)} = g(\sum_j W_{i,j}^{(k-1,k)} z_j^{(k-1)})$$

Importance of Nonlinear Activation Functions

- What happens if we add more layers?
  - $z_2 = W_2(W_1x + b_1) + b_2$
  - $z_2 = W_2(W_1x + b_1) + b_2 = W_2W_1x + W_2b_1 + b_2 = W_{new}x + b_{new}$
  - No gain to adding more linear layers!
  - Idea: add nonlinearities to capture more complex relationships

Deep Neural Network = Also learn the features!

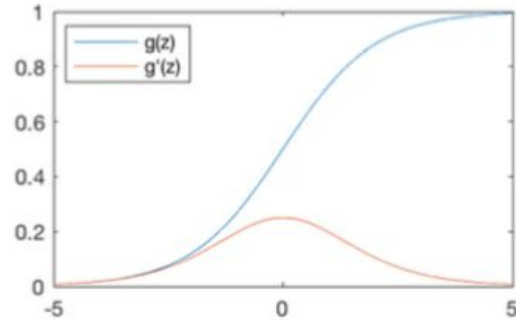


$$z_i^{(k)} = g(\sum_j W_{i,j}^{(k-1,k)} z_j^{(k-1)})$$

g = nonlinear activation function

Common Activation Functions

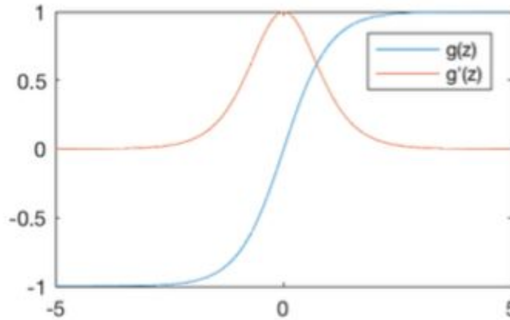
Sigmoid Function



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

$$g'(z) = g(z)(1 - g(z))$$

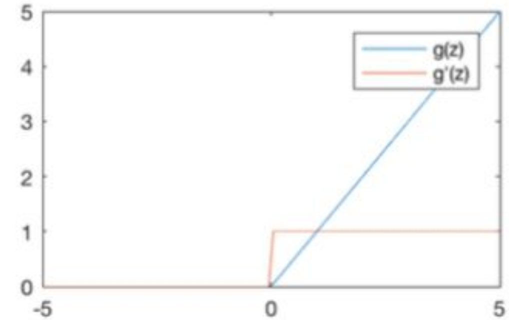
Hyperbolic Tangent



$$g(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

$$g'(z) = 1 - g(z)^2$$

Rectified Linear Unit (ReLU)



$$g(z) = \max(0, z)$$

$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

[source: MIT 6.S191 introtodeeplearning.com]

Deep Neural Network: Also Learn the Features!

- Training the deep neural network is just like logistic regression:

$$\max_{w} \quad ll(w) = \max_{w} \quad \sum_{i} \log P(y^{(i)} | x^{(i)}; w)$$

just w tends to be a much, much larger vector ☺️

Just run gradient ascent

+ stop when log likelihood of hold-out data starts to decrease

Neural Networks Properties

- Theorem (Universal Function Approximators). A two-layer neural network with a sufficient number of neurons can approximate any continuous function to any desired accuracy.

- Practical considerations
  - Can be seen as learning the features
  - Large number of neurons
    - Danger for overfitting
    - (hence early stopping!)

How about computing all the derivatives?

- Derivatives tables:

$$\frac{d}{dx}(a)=0$$

$$\frac{d}{dx}(x)=1$$

$$\frac{d}{dx}(au)=a\frac{du}{dx}$$

$$\frac{d}{dx}(u+v-w)=\frac{du}{dx}+\frac{dv}{dx}-\frac{dw}{dx}$$

$$\frac{d}{dx}(uv)=u\frac{dv}{dx}+v\frac{du}{dx}$$

$$\frac{d}{dx}\left(\frac{u}{v}\right)=\frac{1}{v}\frac{du}{dx}-\frac{u}{v^2}\frac{dv}{dx}$$

$$\frac{d}{dx}(u^n)=nu^{n-1}\frac{du}{dx}$$

$$\frac{d}{dx}(\sqrt{u})=\frac{1}{2\sqrt{u}}\frac{du}{dx}$$

$$\frac{d}{dx}\left(\frac{1}{u}\right)=-\frac{1}{u^2}\frac{du}{dx}$$

$$\frac{d}{dx}\left(\frac{1}{u^n}\right)=-\frac{n}{u^{n+1}}\frac{du}{dx}$$

$$\frac{d}{dx}[f(u)]=\frac{d}{du}[f(u)]\frac{du}{dx}$$

$$\frac{d}{dx}[\ln u]=\frac{d}{dx}[\log_e u]=\frac{1}{u}\frac{du}{dx}$$

$$\frac{d}{dx}\left[\log_a u\right]=\log_a e\frac{1}{u}\frac{du}{dx}$$

$$\frac{d}{dx}e^u=e^u\frac{du}{dx}$$

$$\frac{d}{dx}a^u=a^u\ln a\frac{du}{dx}$$

$$\frac{d}{dx}\left(u^v\right)=vu^{v-1}\frac{du}{dx}+\ln u \;\; u^v\frac{dv}{dx}$$

$$\frac{d}{dx}\sin u=\cos u\frac{du}{dx}$$

$$\frac{d}{dx}\cos u=-\sin u\frac{du}{dx}$$

$$\frac{d}{dx}\tan u=\sec^2 u\frac{du}{dx}$$

$$\frac{d}{dx}\cot u=-\csc^2 u\frac{du}{dx}$$

$$\frac{d}{dx}\sec u=\sec u\tan u\frac{du}{dx}$$

$$\frac{d}{dx}\csc u=-\csc u\cot u\frac{du}{dx}$$

[source:

How about computing all the derivatives?

- But neural net f is never one of those?
  - No problem: CHAIN RULE:

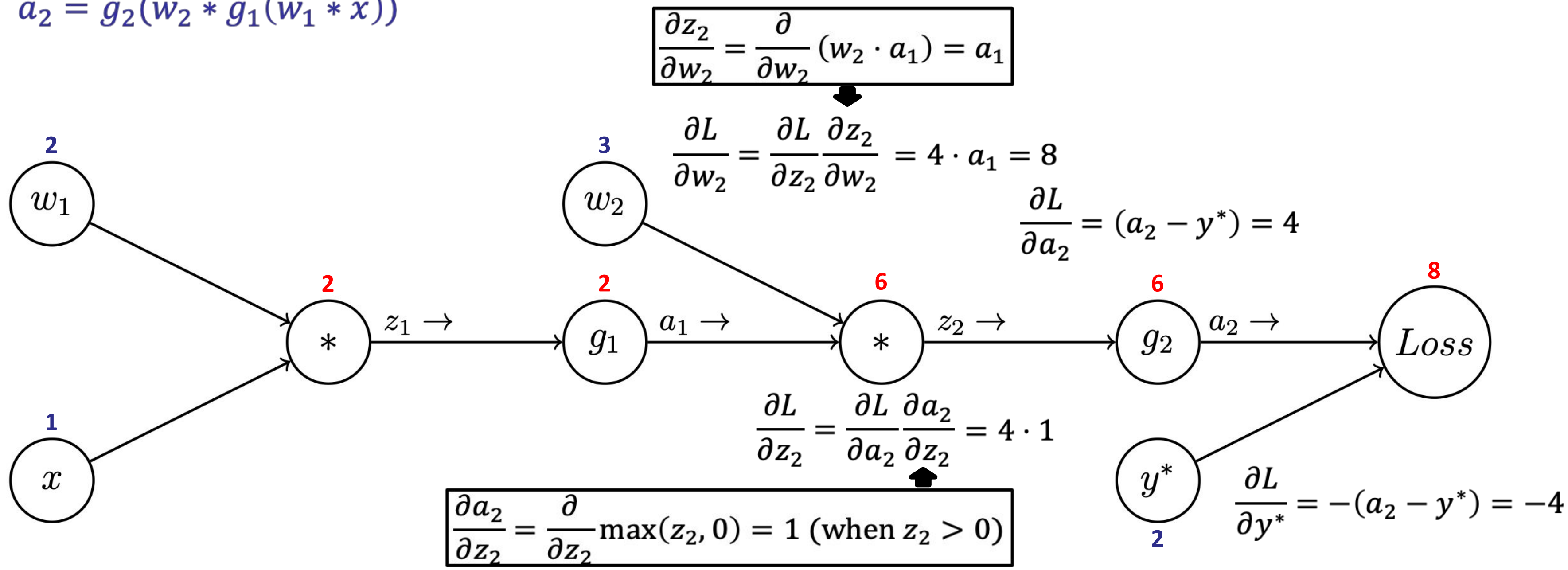
If $$f(x) = g(h(x))$$

Then $$f'(x) = g'(h(x))h'(x)$$

Derivatives can be computed by following well-defined procedures

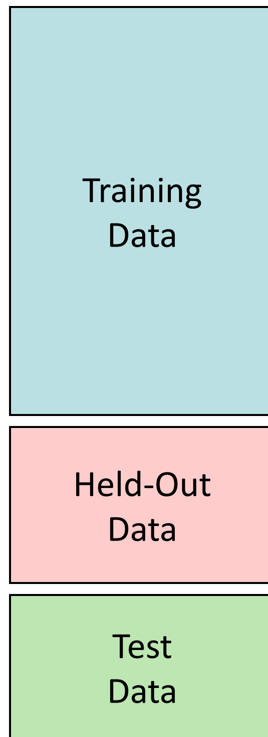
Example: Automatic Differentiation

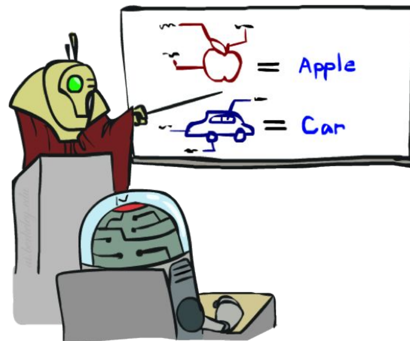
- Build a computation graph and apply chain rule: $f(x) = g(h(x))$ $f'(x) = h'(x) \cdot g'(h(x))$
  - Example: neural network with quadratic loss: $L(a_2, y^*) = \frac{1}{2}(a_2 - y^*)^2$ and ReLU activations $g(z) = \max(0, z)$
  - $a_2 = g_2(w_2 * g_1(w_1 * x))$



Important Concepts

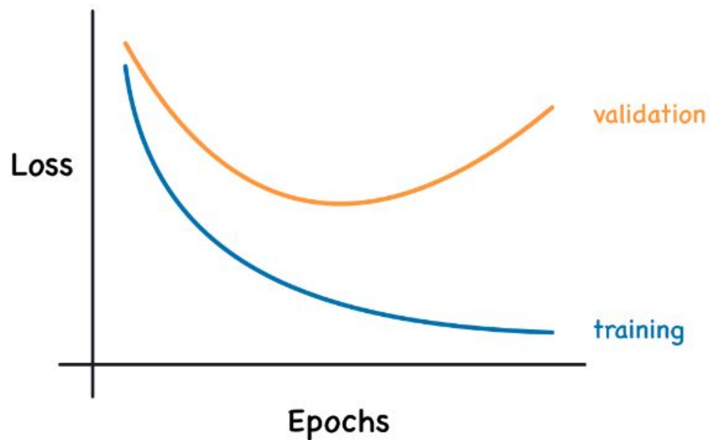
- Data: labeled instances (e.g. emails marked spam/ham)
  - Training set
  - Held out set (“development” or “validation” set)
  - Test set
- Features: attribute-value pairs which characterize each x
- Experimentation cycle
  - Learn parameters (e.g. model probabilities) on training set
  - (Tune hyperparameters on held-out set)
  - Compute accuracy of test set
  - Very important: never “peek” at the test set!
- Evaluation (many metrics possible, e.g. accuracy)
  - Accuracy: fraction of instances predicted correctly
- Overfitting and generalization
  - Want a classifier which does well on *test* data
  - Overfitting: fitting the training data very closely, but not generalizing well
  - We’ll investigate overfitting and generalization formally in a few lectures



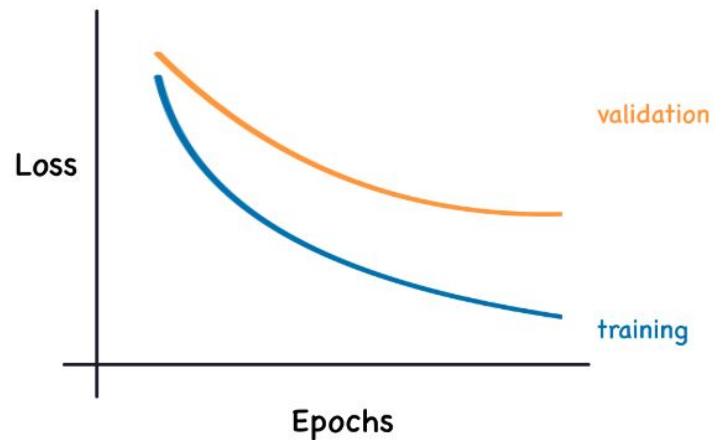


Overfitting & Underfitting

Overfitting

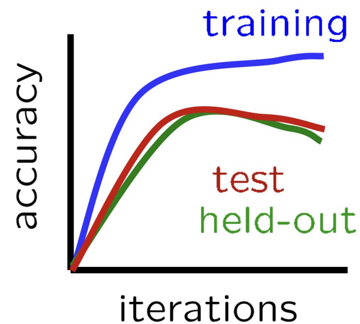


Underfitting



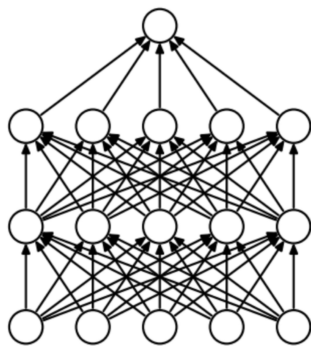
Preventing Overfitting in Neural Networks

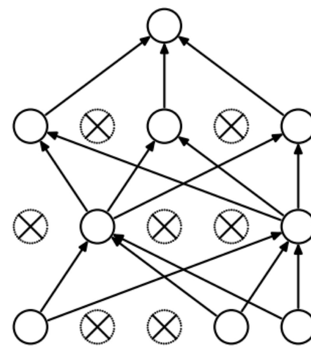
- Early stopping:



- Weight regularization: $\max_w \sum_i \log P\left(y^{(i)} \mid x^{(i)}; w\right) - \frac{\lambda}{2}\sum_j w_j^2$

- Dropout:


(a) Standard Neural Net


(b) After applying dropout.

Controlling Underfitting & Overfitting

- Underfitting
  - Increase model capacity
  - Improve quantity and quality of input features
- Overfitting
  - Limit the hypothesis space
    - E.g. limit the max depth of trees
    - Easier to analyze
  - Regularize the hypothesis selection
    - E.g. chance cutoff
    - Disprefer most of the hypotheses unless data is clear
    - Usually done in practice

Summary of Key Ideas

- Optimize probability of label given input $\max_w \quad ll(w) = \max_w \quad \sum_i \log P(y^{(i)}|x^{(i)}; w)$
- Continuous optimization
  - Gradient ascent:
    - Compute steepest uphill direction = gradient (= just vector of partial derivatives)
    - Take step in the gradient direction
    - Repeat (until held-out data accuracy starts to drop = “early stopping”)
- Deep neural nets
  - Last layer = still logistic regression
  - Now also many more layers before this last layer
    - = computing the features
    - □ the features are learned rather than hand-designed
  - Universal function approximation theorem
    - `If` neural net is large enough
    - `Then` neural net can represent any continuous mapping from input to output with arbitrary accuracy
    - But remember: need to avoid overfitting / memorizing the training data □ early stopping!
  - Automatic differentiation gives the derivatives efficiently

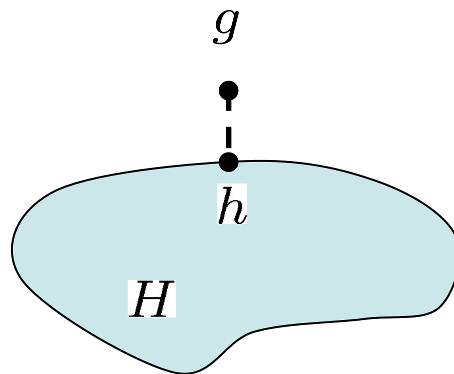
Inductive Learning (Science)

- Simplest form: learn a function from examples
  - A target function: g
  - Examples: input-output pairs $(x, g(x))$
  - E.g. x is an email and $g(x)$ is spam / ham
  - E.g. x is a house and $g(x)$ is its selling price

- Problem:
  - Given a hypothesis space H
  - Given a training set of examples x_i
  - Find a hypothesis $h(x)$ such that $h \sim g$

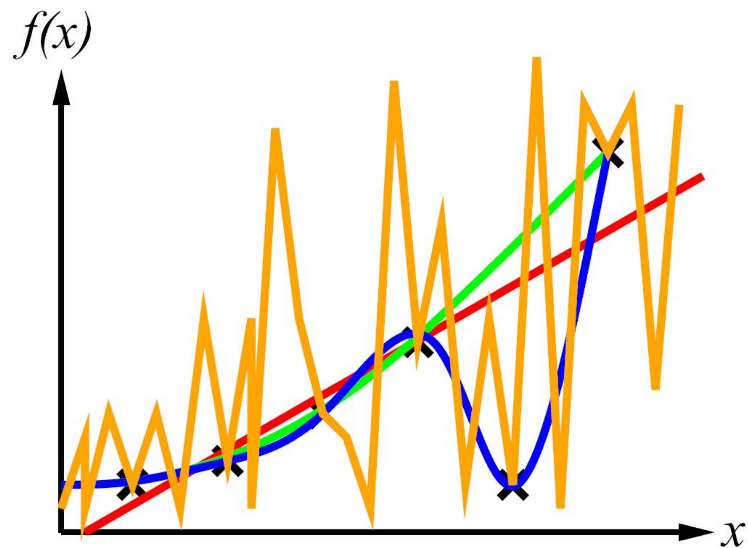
- Includes:
  - Classification (outputs = class labels)
  - Regression (outputs = real numbers)

- How do perceptron and naïve Bayes fit in? $(H, h, g,$ etc.)



Inductive Learning

- Curve fitting (regression, function approximation):



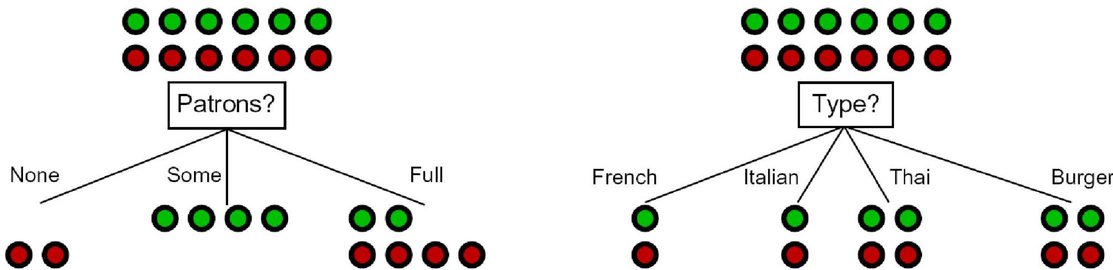
- Consistency vs. simplicity
- Ockham's razor

Consistency vs. Simplicity

- Fundamental tradeoff: bias vs. variance
- Usually algorithms prefer consistency by default (why?)
- Several ways to operationalize “simplicity”
  - Reduce the hypothesis space
    - Assume more: e.g. independence assumptions, as in naïve Bayes
    - Have fewer, better features / attributes: feature selection
    - Other structural limitations (decision lists vs trees)
  - Regularization
    - Smoothing: cautious use of small counts
    - Many other generalization parameters (pruning cutoffs today)
    - Hypothesis space stays big, but harder to get to the outskirts

Decision Trees: Choosing an Attribute

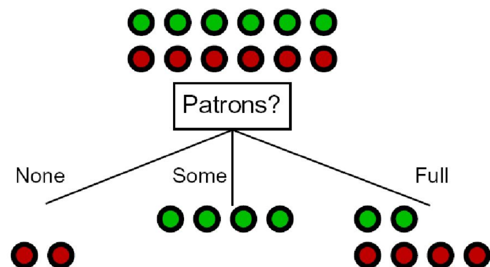
- Idea: a good attribute splits the examples into subsets that are (ideally) “all positive” or “all negative”

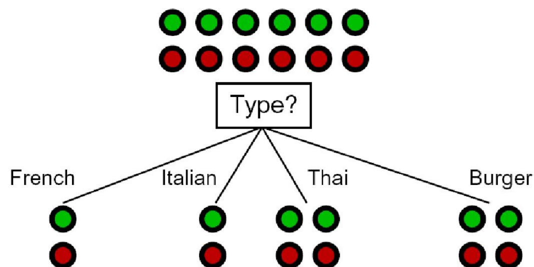


- So: we need a measure of how “good” a split is, even if the results aren’t perfectly separated out

Information Gain

- Back to decision trees!
- For each split, compare entropy before and after
  - Difference is the information gain
  - Problem: there's more than one distribution after split!





  - Solution: use expected entropy, weighted by the number of examples

Advanced Topics: NLP

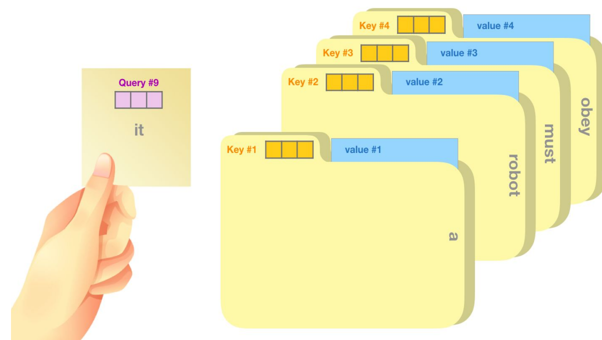
- N-gram models
  - Regularization techniques (smoothing, ba
- RNNs -> LSTMs -> Attention, Transform
  - Address long-term memory issues
- Causal (autoregressive) vs. masked LMs
  - Predict tokens in order vs. mask some out randomly and predict
- Pretraining & fine-tuning

Training Counts

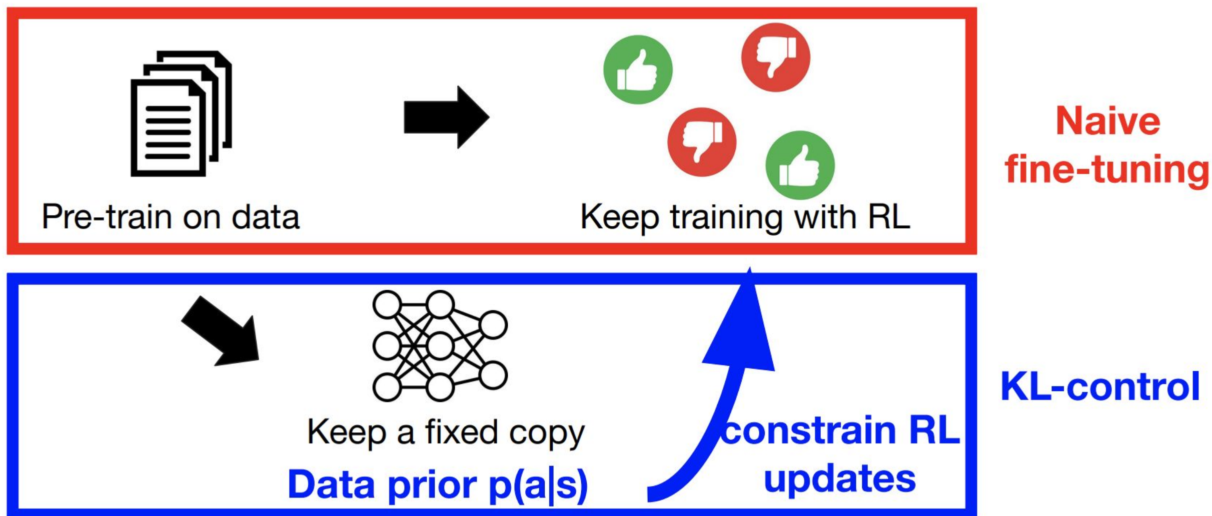
198015222 the first
194623024 the same
168504105 the following
158562063 the world
…
14112454 the door

23135851162 the *

$$\hat{P}(\text{door}|\text{the}) = \frac{14112454}{23135851162} = 0.0006$$



Advanced Topics: RL

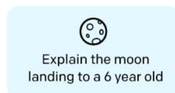


Advanced Topics: RL

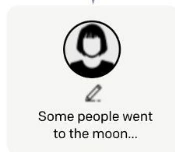
Step 1

Collect demonstration data, and train a supervised policy.

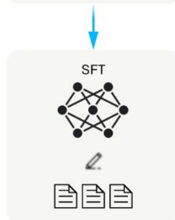
A prompt is sampled from our prompt dataset.



A labeler demonstrates the desired output behavior.



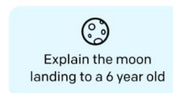
This data is used to fine-tune GPT-3 with supervised learning.



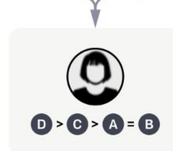
Step 2

Collect comparison data, and train a reward model.

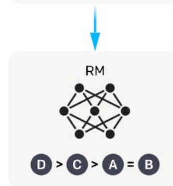
A prompt and several model outputs are sampled.



A labeler ranks the outputs from best to worst.



This data is used to train our reward model.

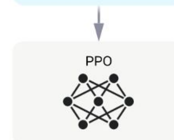


Step 3

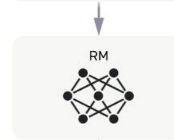
Optimize a policy against the reward model using reinforcement learning.

A new prompt is sampled from the dataset.

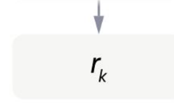
The policy generates an output.



The reward model calculates a reward for the output.



The reward is used to update the policy using PPO.



Advanced Topics: Ethics, Fairness, Safety

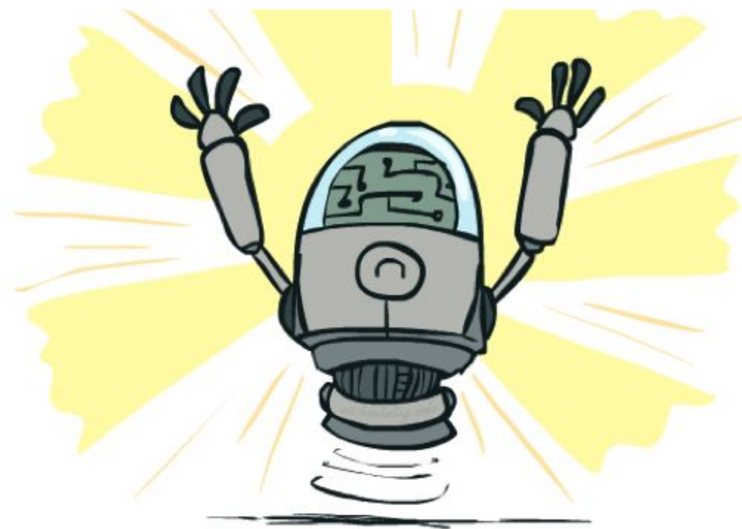
- Allocational & representational harms
- Dataset bias + bias amplification + automation bias
- Training data extraction
- Data poisoning
- Model stealing
- Safety in physical environments
- Jailbreaking & adversarial attacks

Questions

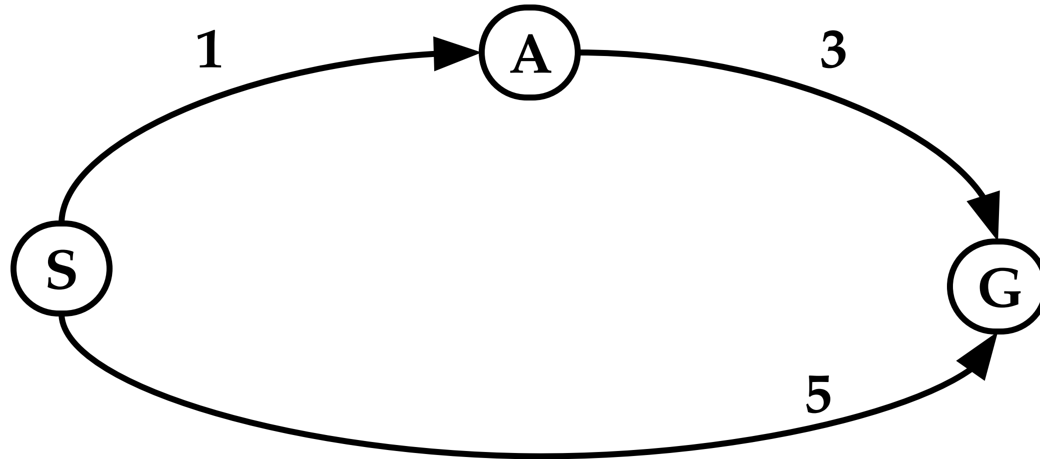
Search

A* Search

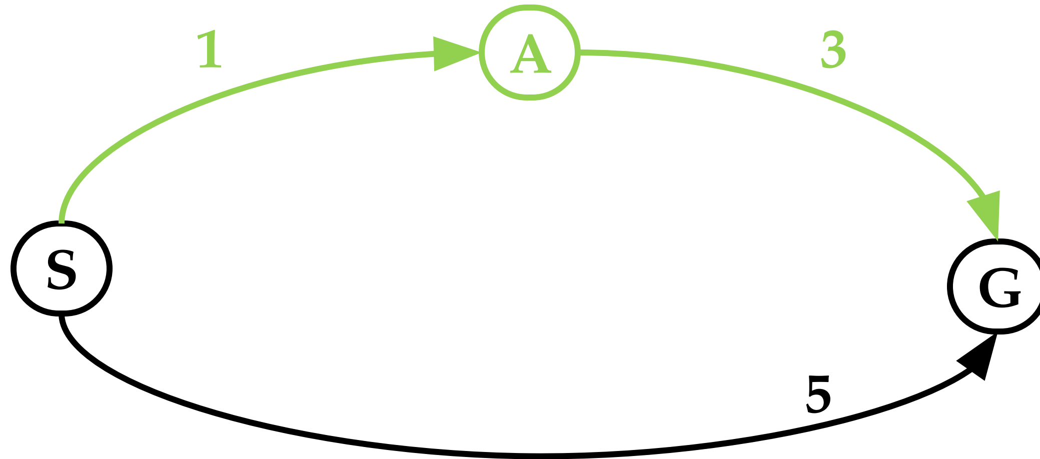
- Expand nodes based on sum: backward cost + forward cost
  - f(n) = g(n) + h(n)
  - g(n): cost to get to node
  - h(n): heuristic of future costs
- We ideally want heuristic functions that satisfy:
  - Admissibility: underestimate true cost to the goal
  - Consistency: “triangle inequality”
- Consistency => admissibility



A* Search

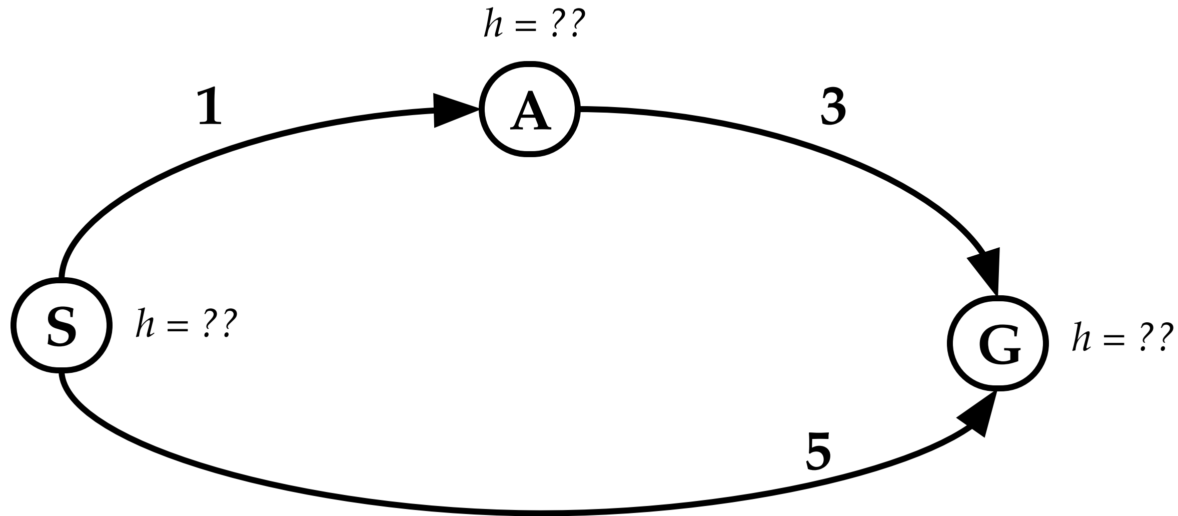


A* Search



A* Search

g: backward cost (S -> current node)
h: forward cost (heuristic for current node -> goal)

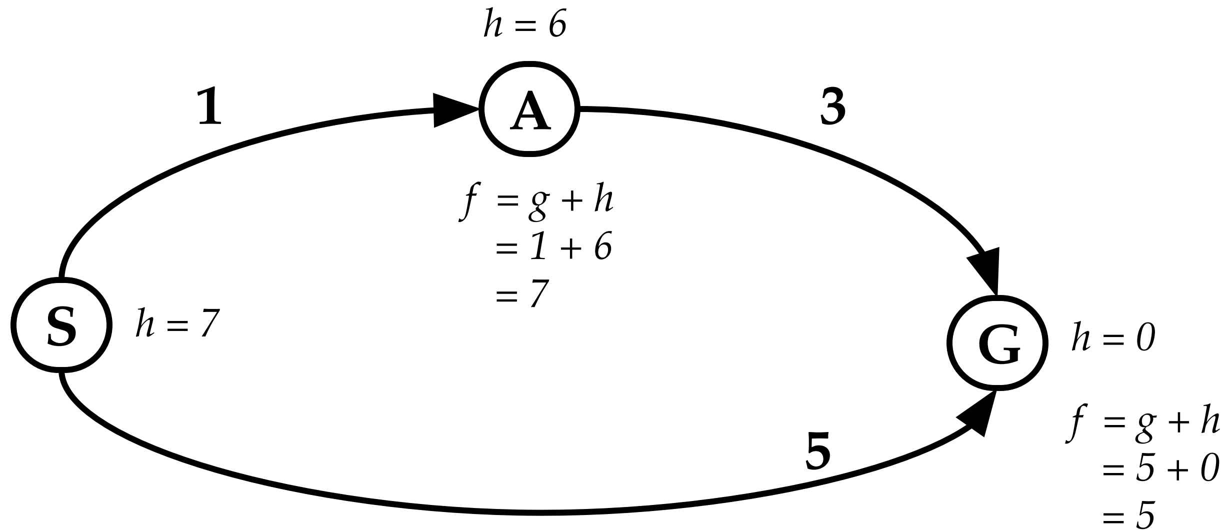


A* Search

Q: Where do heuristics come from?

A: We have to create them!

g: backward cost (S -> current node)
h: forward cost (heuristic for current node -> goal)



Not the best heuristic…

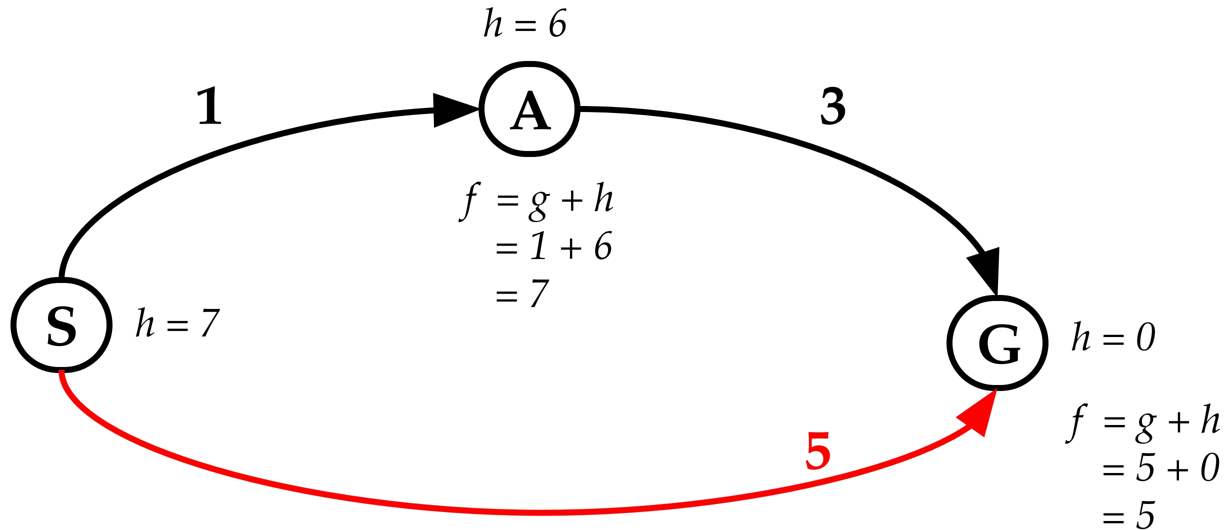
A* Search

Q: Where do heuristics come from?

A: We have to create them!

g: backward cost (S -> current node)

h: forward cost (heuristic for current node -> goal)



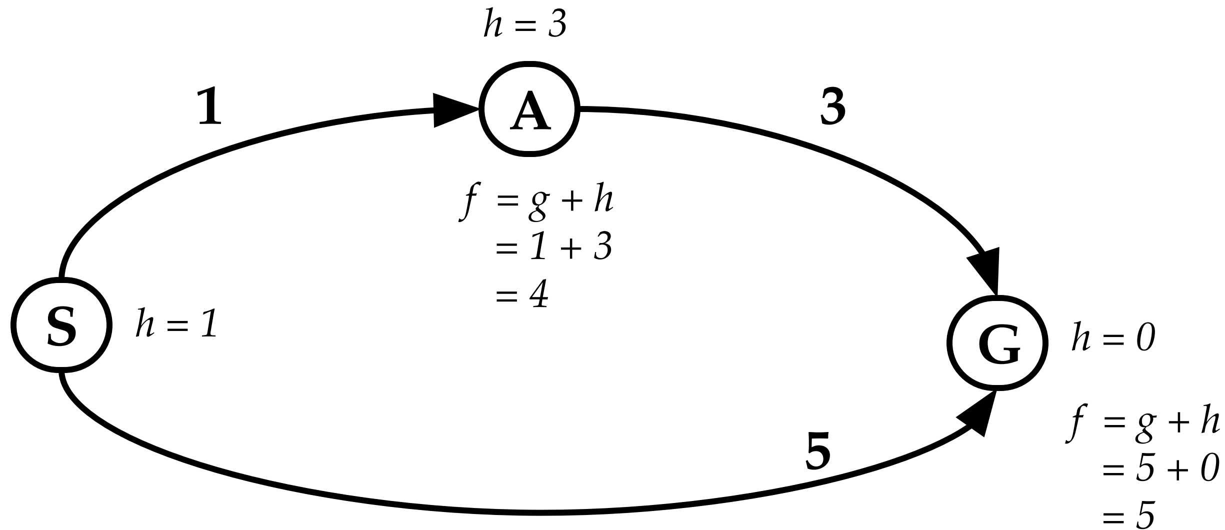
Not the best heuristic…

A* Search: Admissibility

Q: Where do heuristics come from?

A: We have to create them!

g: backward cost (S -> current node)
h: forward cost (heuristic for current node -> goal)



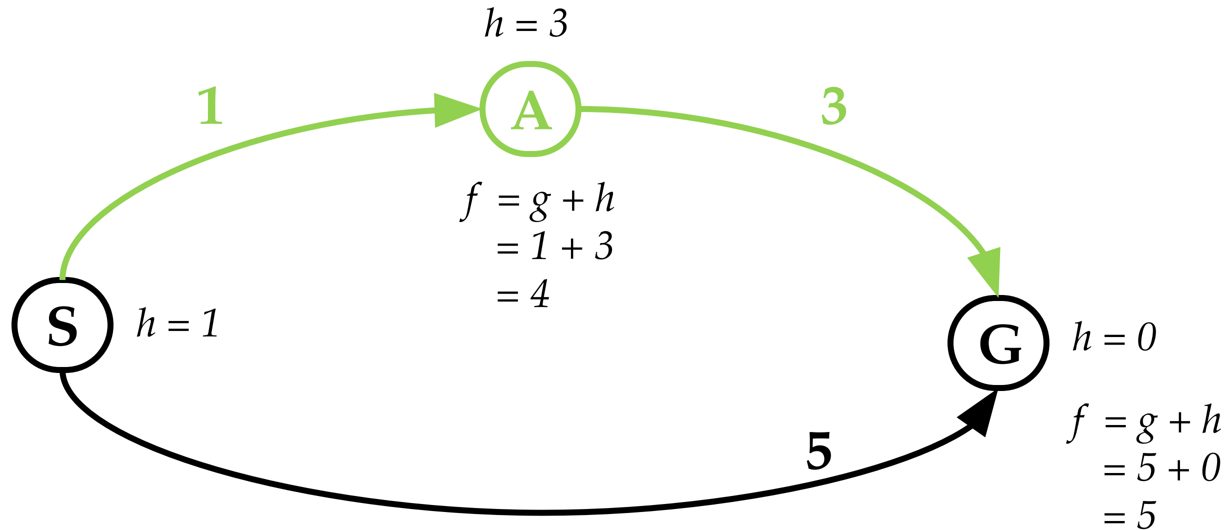
What's a better heuristic?

A* Search: Admissibility

Q: Where do heuristics come from?

A: We have to create them!

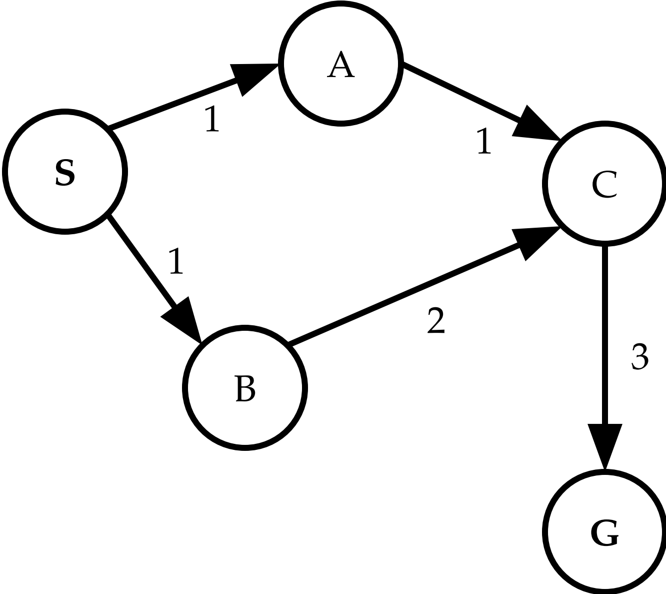
g: backward cost (S -> current node)
h: forward cost (heuristic for current node -> goal)



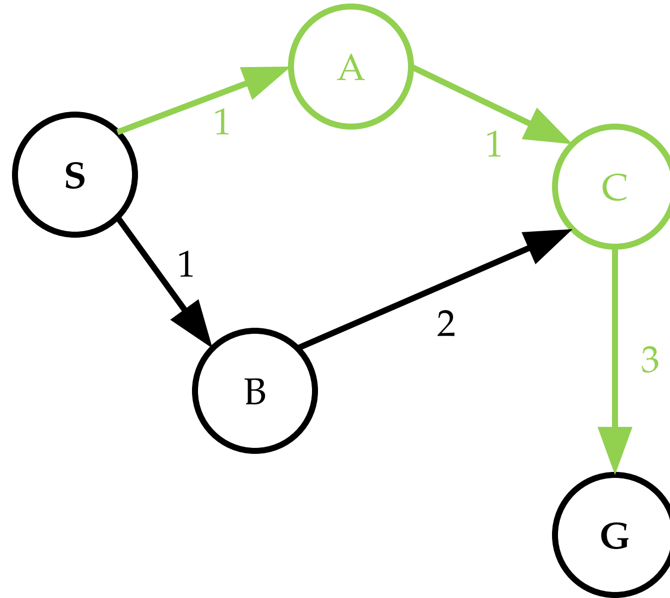
What's a better heuristic?

Admissible = Underestimates cost from any node to the goal

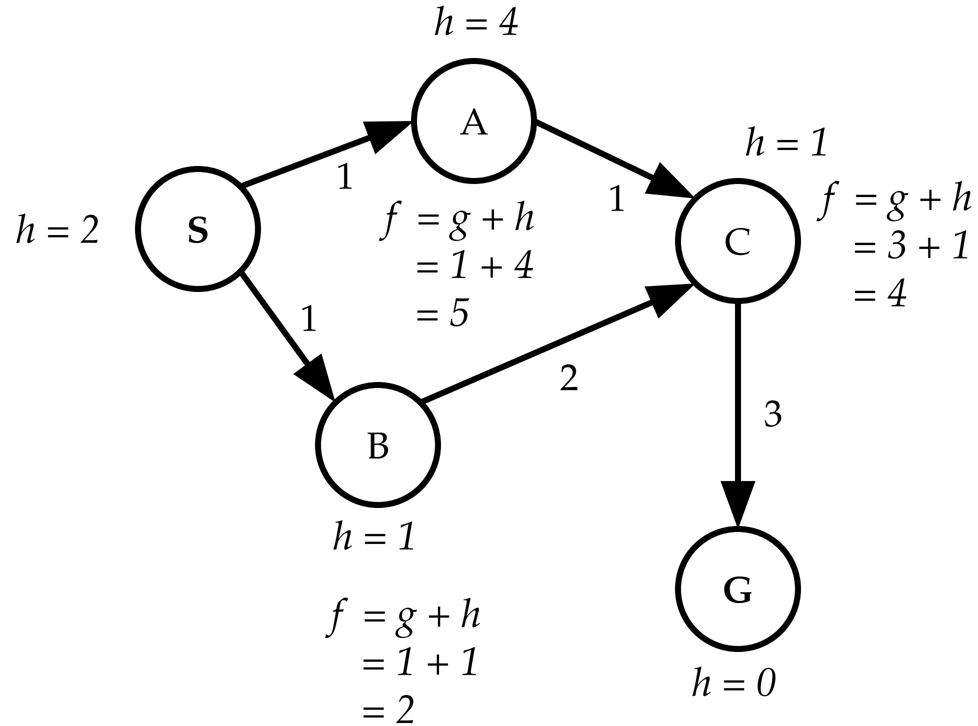
A* Search



A* Search: Consistency

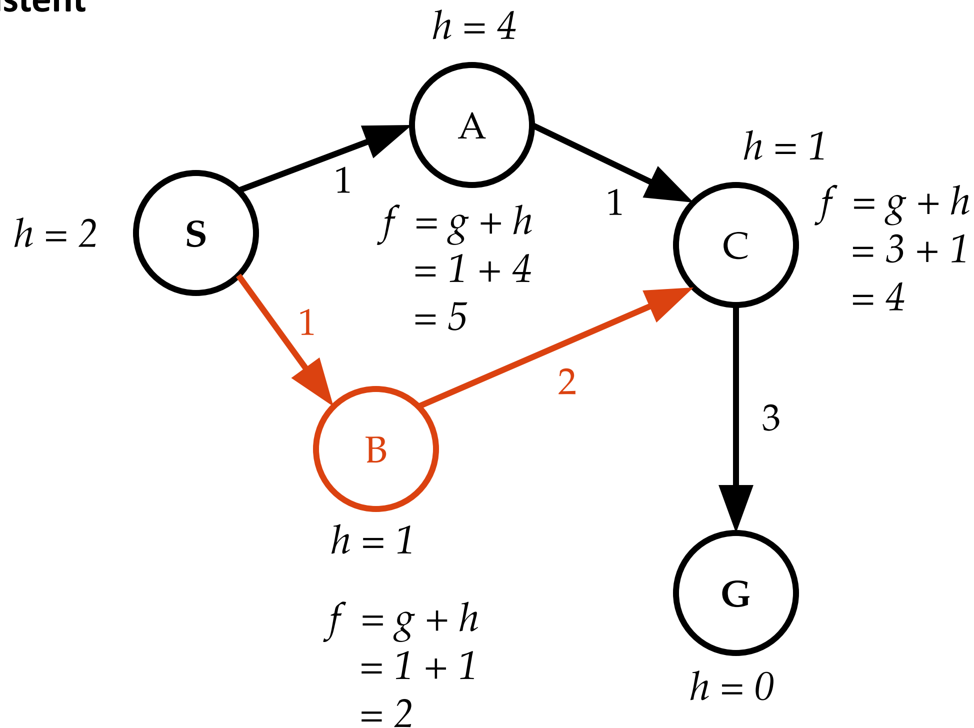


A* Search: Consistency



A* Search: Consistency

This heuristic isn't **consistent**



"Triangle inequality"

$h(u) \leq d(u,v) + h(v)$

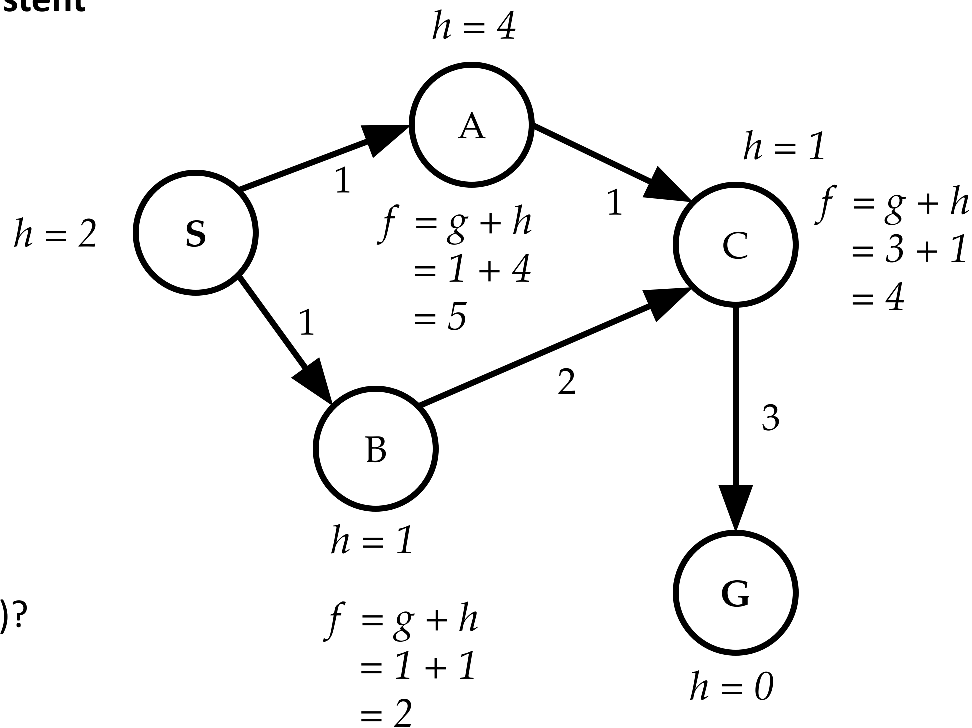
A* Search: Consistency

This heuristic isn’t **consistent**

“Triangle inequality”

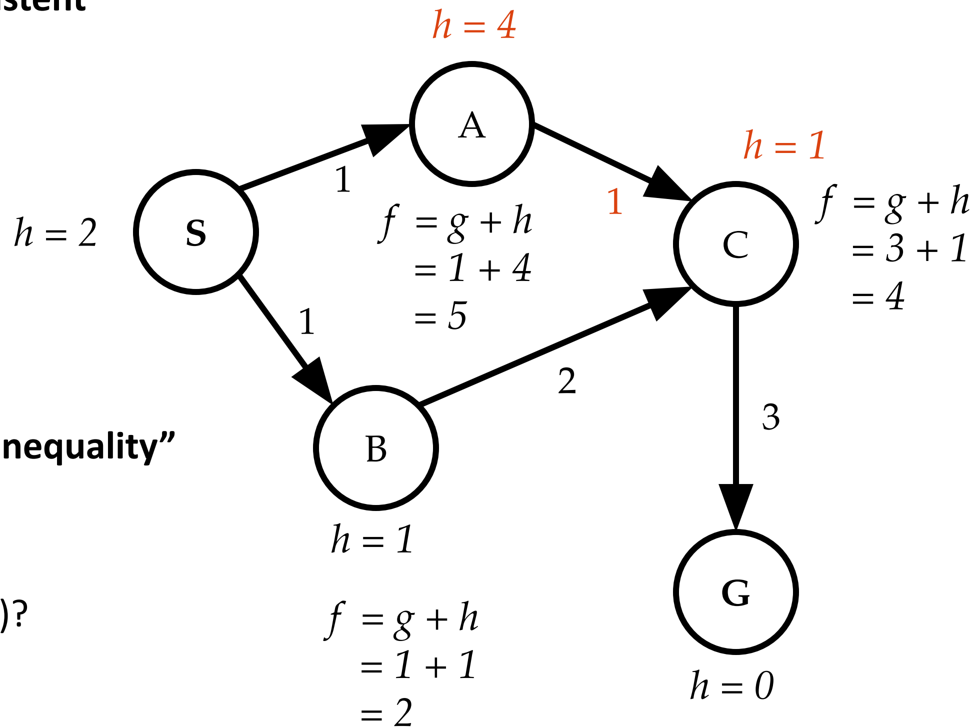
h(u) ≤ d(u,v) + h(v)

Q: Is h(A) ≤ d(A,C) + h(C)?



A* Search: Consistency

This heuristic isn't **consistent**



Consistency: "Triangle inequality"

h(u) ≤ d(u,v) + h(v)

Q: Is h(A) ≤ d(A,C) + h(C)?

A: No: 4 ≰ 1 + 1

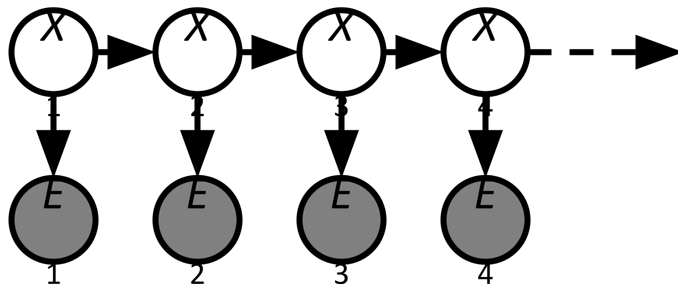
Summary of A*

- Tree search:
  - A* is optimal if heuristic is admissible
  - UCS is a special case (h = 0)
- Graph search:
  - A* optimal if heuristic is consistent
  - UCS optimal (h = 0 is consistent)
- Consistency implies admissibility
- In general, most natural admissible heuristics tend to be consistent, especially if it comes from a relaxed problem

Hidden Markov Models

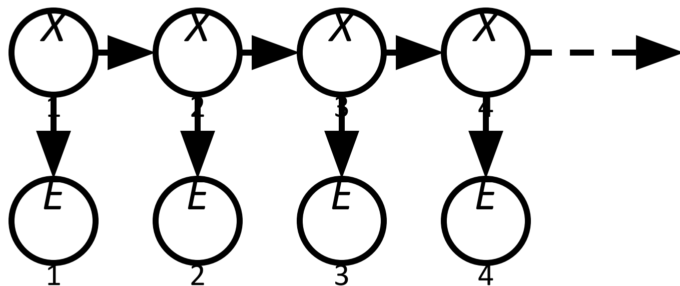
- Hidden Markov models (HMMs)
  - Underlying Markov chain over states X_i
  - You observe outputs (effects) at each time step

- An HMM is defined by:
  - Initial distribution: $P(X_1)$
  - Transitions: $P(X_t \mid X_{t-1})$
  - Emissions: $P(E_t \mid X_t)$



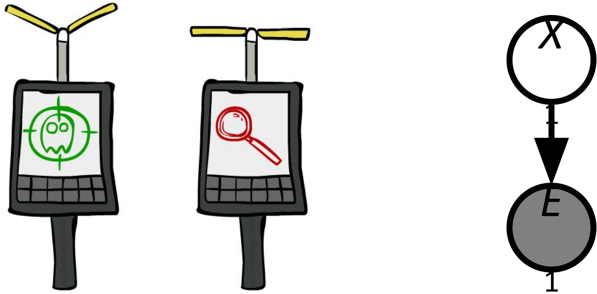
Conditional Independence

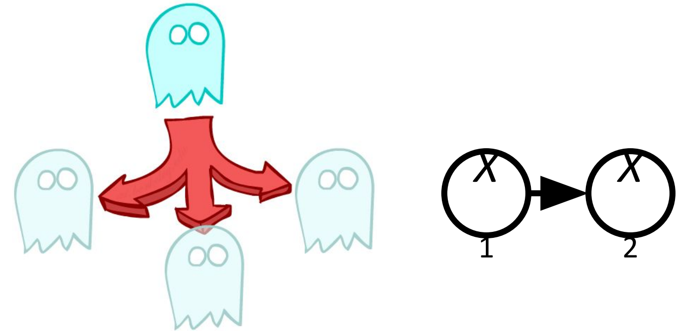
- HMMs have two important independence properties:
  - Markovian assumption of hidden process
  - Current observation independent of all else given current state



- Does this mean that evidence variables are guaranteed to be independent?
  - [No, they tend to correlated by the hidden state]

Inference: Base Cases





$$P(X_1|e_1)$$

$$P(X_2)$$

$$P(X_1|e_1) = \frac{P(X_1, e_1)}{\sum_{x_1} P(x_1, e_1)}$$

$$P(X_1|e_1) = \frac{P(e_1|X_1)P(X_1)}{\sum_{x_1} P(e_1|x_1)P(x_1)}$$

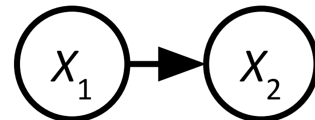
$$P(X_2) = \sum_{x_1} P(x_1, X_2)$$

$$P(X_2) = \sum_{x_1} P(X_2|x_1)P(x_1)$$

Passage of Time

- Assume we have current belief P(X | evidence to date)

$$P(X_t|e_{1:t})$$

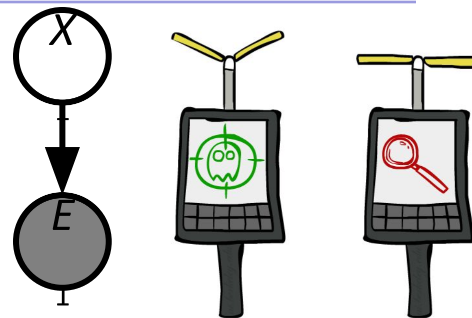


- Then, after one time step passes:

$$\begin{aligned} P(X_{t+1}|e_{1:t}) &= \sum_{x_t} P(X_{t+1}, x_t|e_{1:t}) \\ &= \sum_{x_t} P(X_{t+1}|x_t, e_{1:t})P(x_t|e_{1:t}) \\ &= \sum_{x_t} P(X_{t+1}|x_t)P(x_t|e_{1:t}) \end{aligned}$$

- Basic idea: beliefs get “pushed” through the transitions

Observation



- Assume we have current belief P(X | previous evidence):

$$P(X_{t+1}|e_{1:t})$$

- Then, after evidence comes in:

$$P(X_{t+1}|e_{1:t+1}) = P(X_{t+1}, e_{t+1}|e_{1:t})/P(e_{t+1}|e_{1:t})$$

$$\propto_{X_{t+1}} P(X_{t+1}, e_{t+1}|e_{1:t})$$

$$= P(e_{t+1}|e_{1:t}, X_{t+1})P(X_{t+1}|e_{1:t})$$

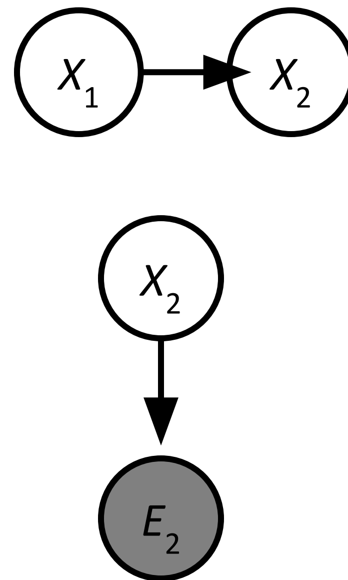
$$= P(e_{t+1}|X_{t+1})P(X_{t+1}|e_{1:t})$$

- Basic idea: beliefs “reweighted” by likelihood of evidence
- Unlike passage of time, we have to renormalize

Online Belief Updates

- Every time step, we start with current P(X | evidence)
- We update for time:

$$P(x_t|e_{1:t-1}) = \sum_{x_{t-1}} P(x_{t-1}|e_{1:t-1}) \cdot P(x_t|x_{t-1})$$



- We update for evidence:

$$P(x_t|e_{1:t}) \propto_X P(x_t|e_{1:t-1}) \cdot P(e_t|x_t)$$

- The forward algorithm does both at once (and doesn't normalize)

The Forward Algorithm

- We are given evidence at each time and want to know

$$P(X_t|e_{1:t})$$

- We can derive the following updates

$$P(x_t|e_{1:t}) \propto_{X_t} P(x_t, e_{1:t})$$

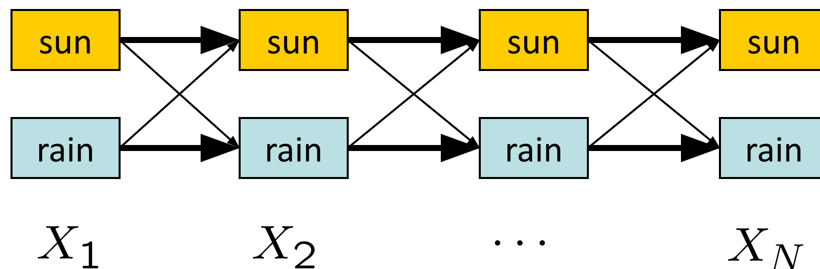
$$= \sum_{x_{t-1}} P(x_{t-1}, x_t, e_{1:t})$$

$$= \sum_{x_{t-1}} P(x_{t-1}, e_{1:t-1}) P(x_t|x_{t-1}) P(e_t|x_t)$$

$$= P(e_t|x_t) \sum_{x_{t-1}} P(x_t|x_{t-1}) P(x_{t-1}, e_{1:t-1})$$

We can normalize as we go if we want to have P(x|e) at each time step, or just once at the end…

Forward / Viterbi Algorithms



Forward Algorithm (Sum)

For each state at time *t*, keep track of the ***total probability of all paths*** to it

$$f_t[x_t] = P(x_t, e_{1:t})$$

$$= P(e_t|x_t) \sum_{x_{t-1}} P(x_t|x_{t-1}) f_{t-1}[x_{t-1}]$$

Viterbi Algorithm (Max)

For each state at time *t*, keep track of the ***maximum probability of any path*** to it

$$m_t[x_t] = \max_{x_{1:t-1}} P(x_{1:t-1}, x_t, e_{1:t})$$

$$= P(e_t|x_t) \max_{x_{t-1}} P(x_t|x_{t-1}) m_{t-1}[x_{t-1}]$$

Constraint Satisfaction Problems

Example: Map Coloring

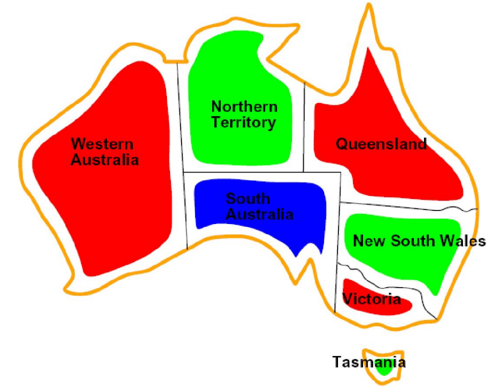
- Variables: WA, NT, Q, NSW, V, SA, T
- Domains: $D = \{\text{red}, \text{green}, \text{blue}\}$
- Constraints: adjacent regions must have different colors

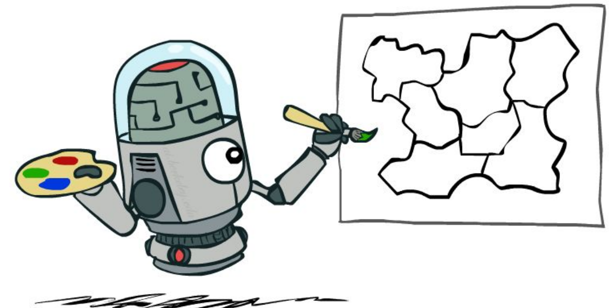
  Implicit: $\text{WA} \neq \text{NT}$

  Explicit: $(\text{WA}, \text{NT}) \in \{(\text{red}, \text{green}), (\text{red}, \text{blue}), \ldots\}$

- Solutions are assignments satisfying all constraints, e.g.:

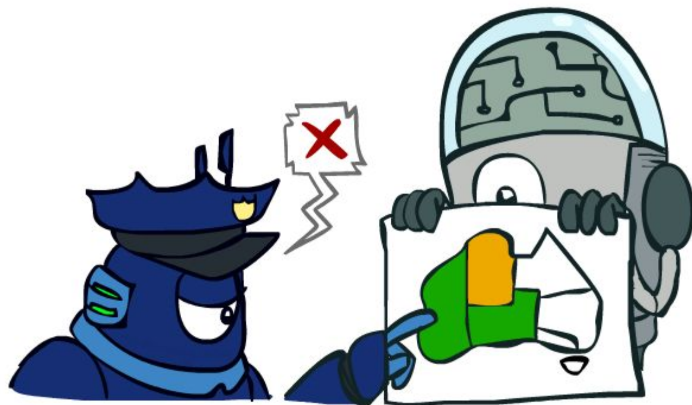
  {WA=red, NT=green, Q=red, NSW=green, V=red, SA=blue, T=green}





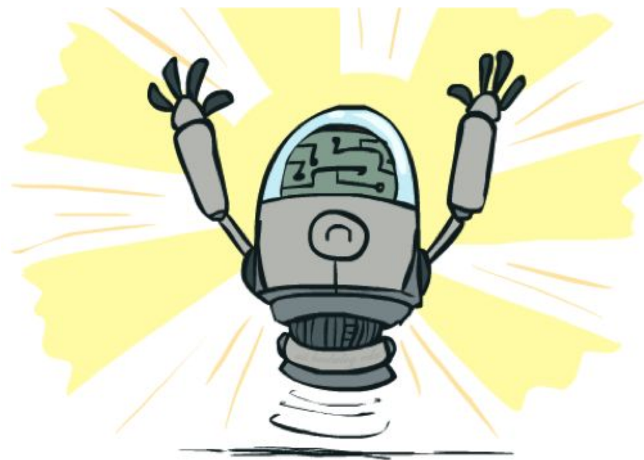
General Approach #1: Backtracking Search

- Backtracking search is the basic uninformed algorithm for solving CSPs
- Idea 1: One variable at a time
  - Variable assignments are commutative, so fix ordering -> better branching factor!
  - I.e., [WA = red then NT = green] same as [NT = green then WA = red]
  - Only need to consider assignments to a single variable at each step
- Idea 2: Check constraints as you go
  - I.e. consider only values which do not conflict previous assignments
  - Might have to do some computation to check the constraints
  - "Incremental goal test"
- Depth-first search with these two improvements is called *backtracking search* (not the best name)
- Can solve n-queens for $n \approx 25$



Improving Backtracking

General-purpose ideas give huge gains in speed



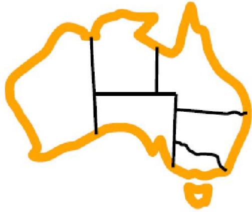
1. Ordering:
   - Which variable should be assigned next?
   - In what order should its values be tried?

2. Filtering: Can we detect inevitable failure early?

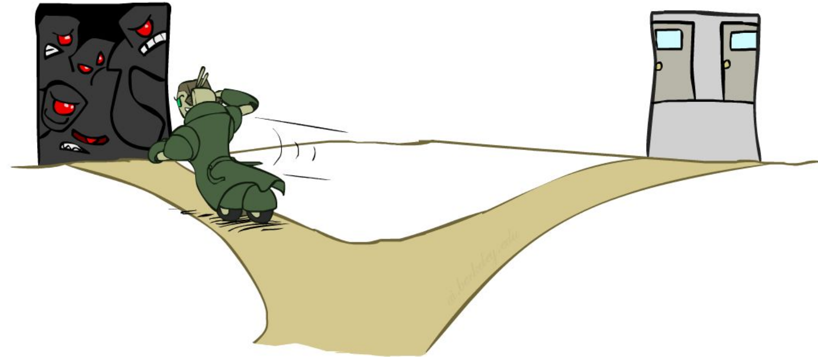
3. Leveraging the structure of the constraint graph

Ordering: Minimum Remaining Values

- Variable Ordering: Minimum remaining values (MRV):
  - Choose the variable with the fewest legal values left in its domain

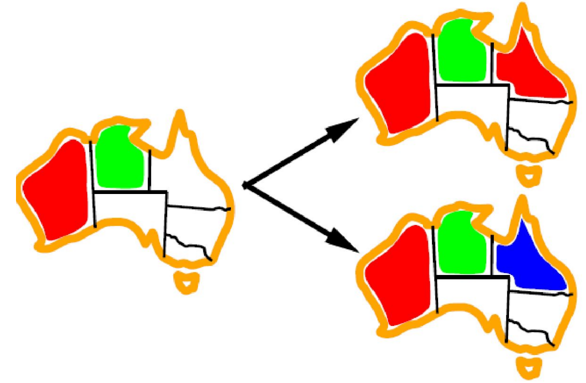


- Why min rather than max?
- Also called "most constrained variable"
- "Fail-fast" ordering



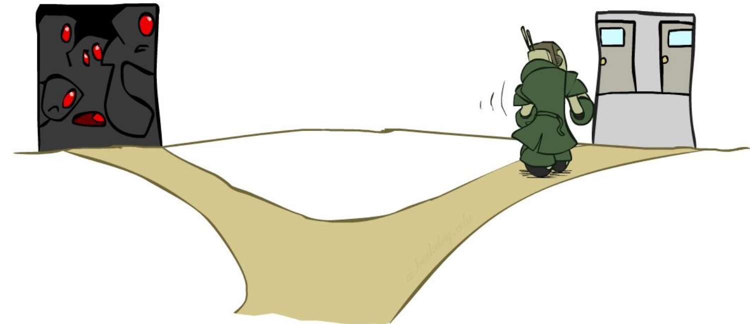
Ordering: Least Constraining Value

- Value Ordering: Least Constraining **Value**
  - Given a choice of variable, choose the *least constraining value*
  - I.e., the one that rules out the fewest values in the remaining variables
  - Note that it may take some computation to determine this! (E.g., rerunning filtering)



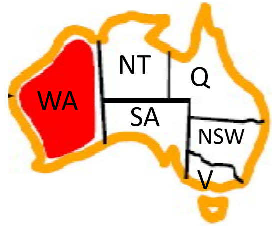
- Why least rather than most?

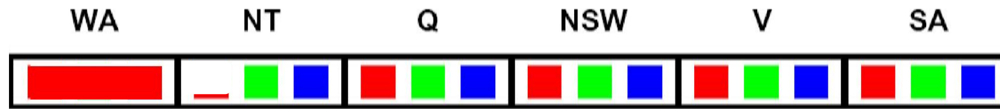
- Combining these ordering ideas makes 1000 queens feasible

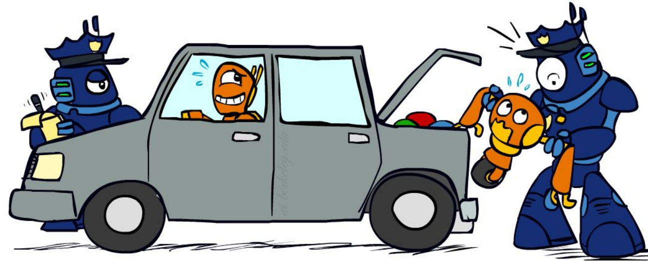


Filtering: Arc Consistency

- An arc X → Y is consistent iff for *every* x in the tail there is *some* y in the head which could be assigned without violating a constraint







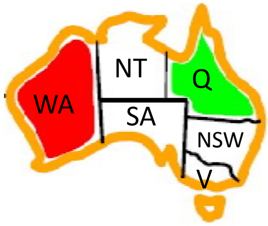
Delete from the tail!

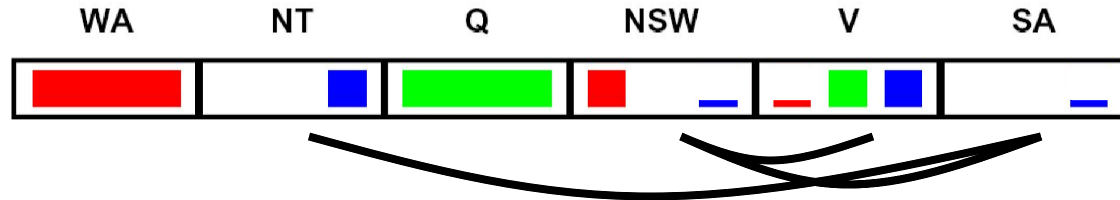
Forward checking?

Enforcing consistency of arcs pointing to each new assignment

Filtering: Arc Consistency

- A simple form of propagation makes sure all arcs are consistent:





- Important: If X loses a value, neighbors of X need to be rechecked!
- Arc consistency detects failure earlier than forward checking
- Can be run as a preprocessor or after each assignment
- What's the downside of enforcing arc consistency?

Remember: Delete from the tail!

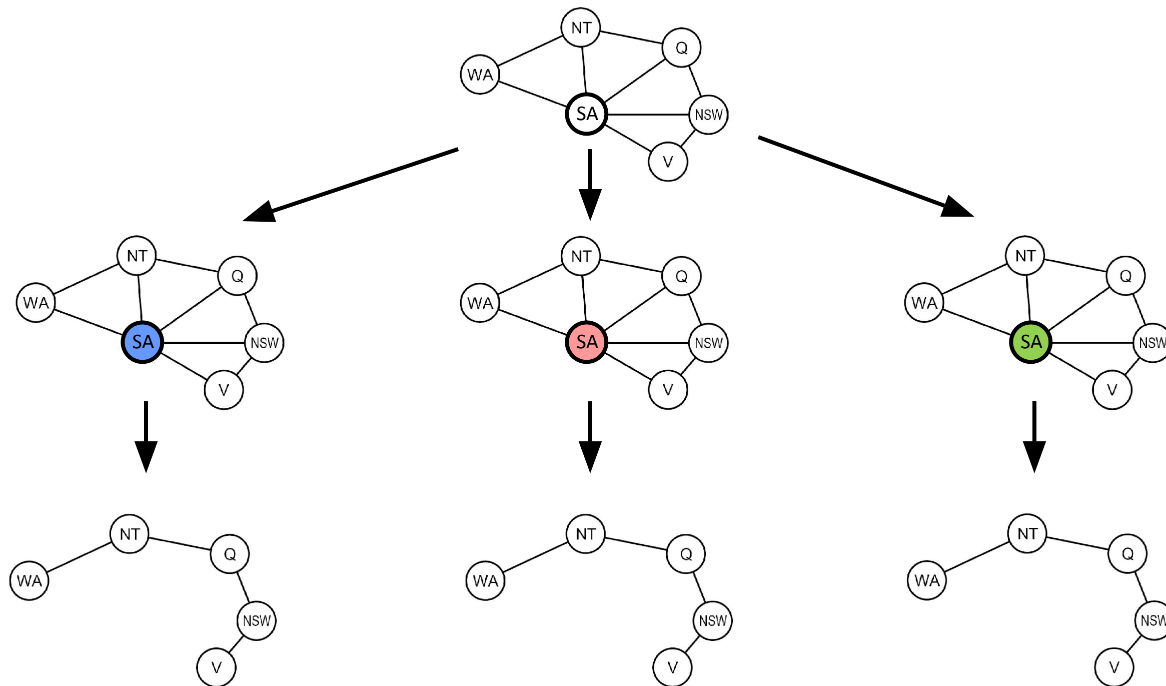
Leveraging Structure: Cutsets

Choose a cutset

Instantiate the cutset (all possible ways)

Compute residual CSP for each assignment

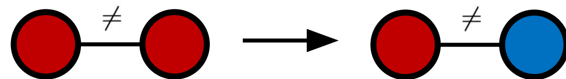
Solve the residual CSPs (tree structured)



General Approach #2: Iterative Improvement

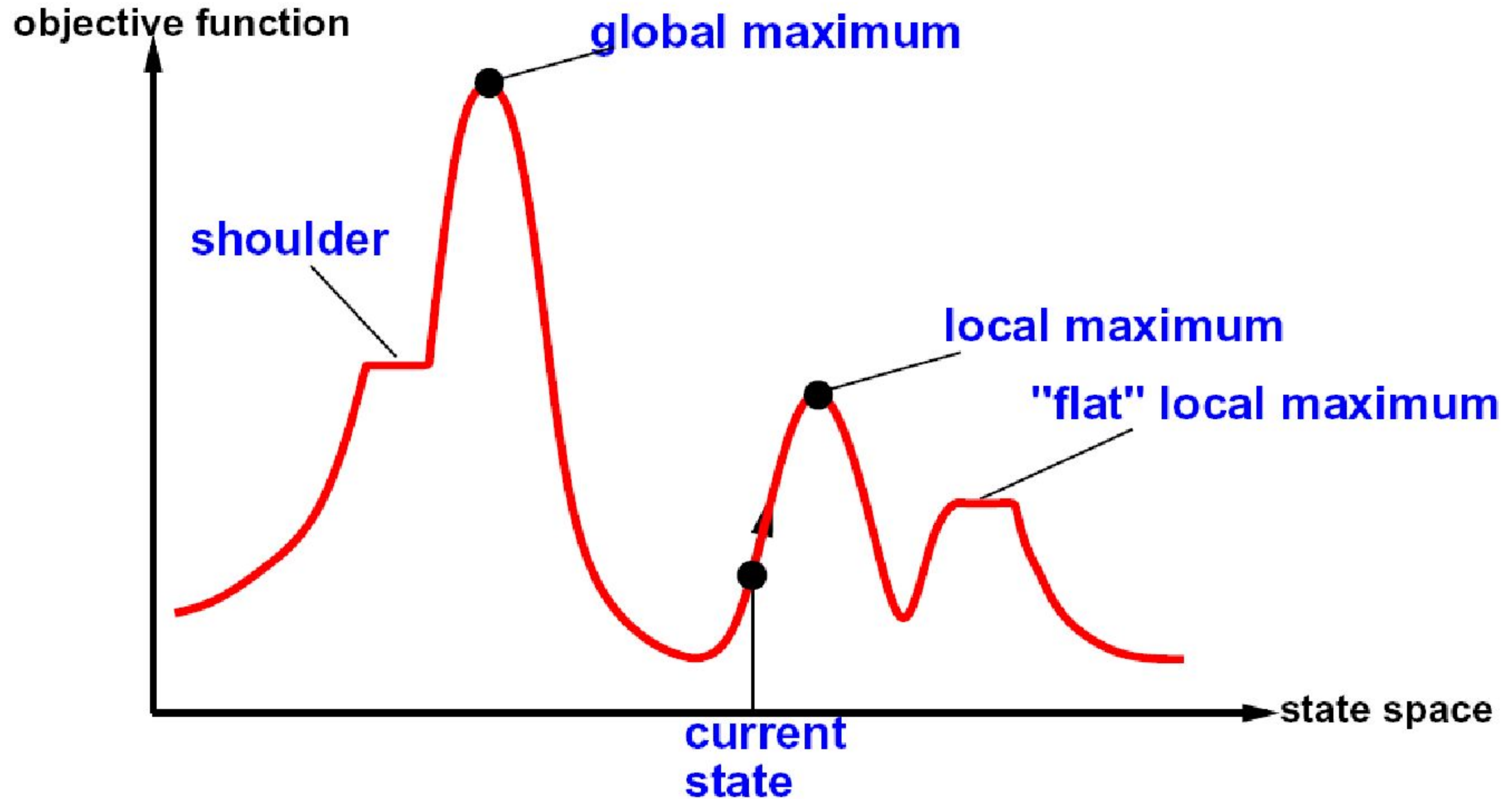
- Local search methods typically work with “complete” states, i.e., all variables assigned

- To apply to CSPs:
  - Take an assignment with unsatisfied constraints
  - Operators *reassign* variable values
  - No fringe! Live on the edge.



- Algorithm: While not solved,
  - Variable selection: randomly select any conflicted variable
  - Value selection: min-conflicts heuristic:
    - Choose a value that violates the fewest constraints
    - I.e., hill climb with h(x) = total number of violated constraints

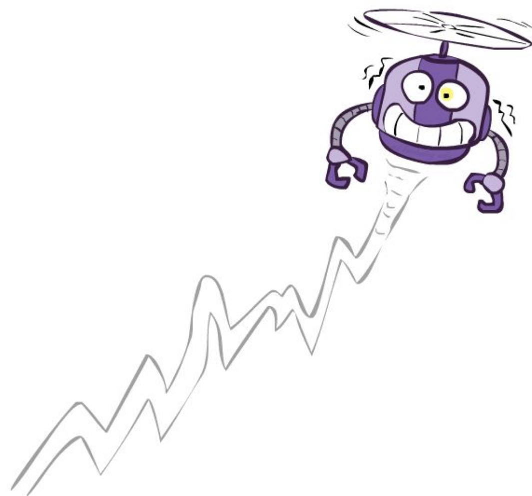
Hill Climbing Diagram



Simulated Annealing

- Idea: Escape local maxima by allowing downhill moves
  - But make them rarer as time goes on

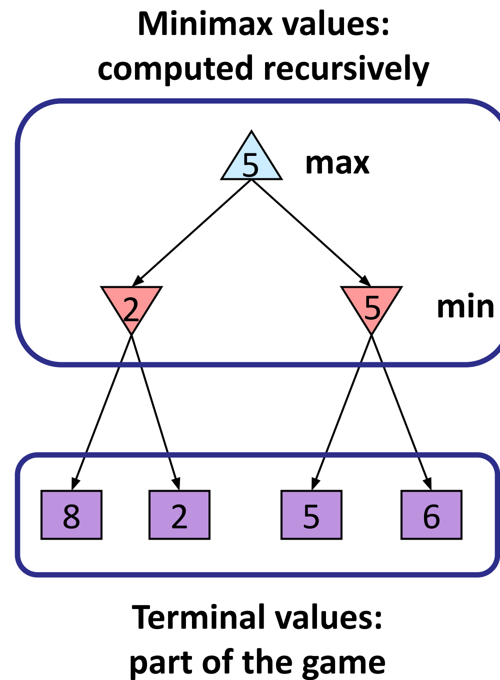
```
function SIMULATED-ANNEALING( problem, schedule) returns a solution state
   inputs: problem, a problem
           schedule, a mapping from time to "temperature"
   local variables: current, a node
                    next, a node
                    T, a "temperature" controlling prob. of downward steps

   current ← MAKE-NODE(INITIAL-STATE[problem])
   for t ← 1 to ∞ do
       T ← schedule[t]
       if T = 0 then return current
       next ← a randomly selected successor of current
       ΔE ← VALUE[next] – VALUE[current]
       if ΔE > 0 then current ← next
       else current ← next only with probability e^(ΔE/T)
```



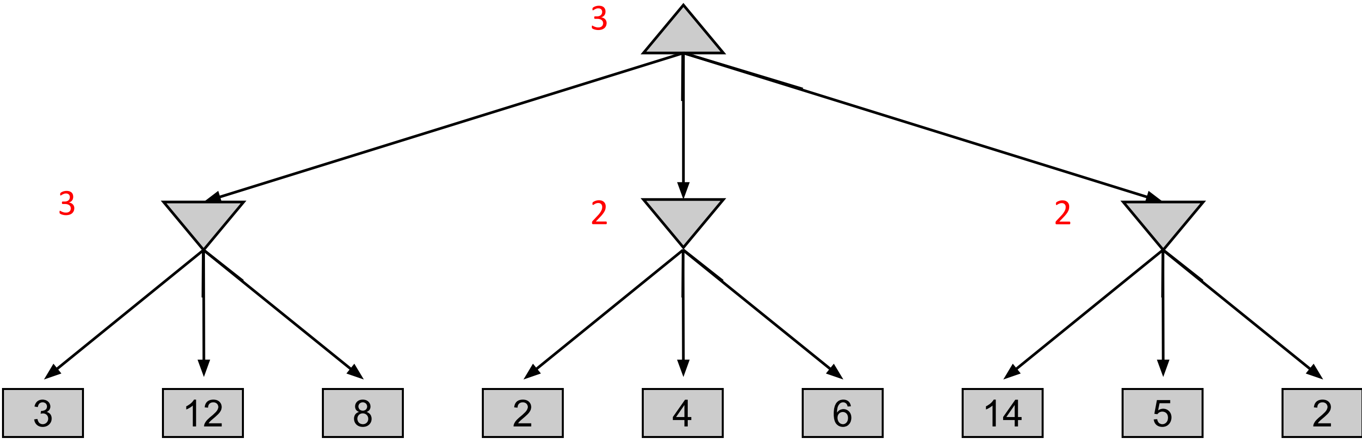
Game Trees

Adversarial Search (Minimax)

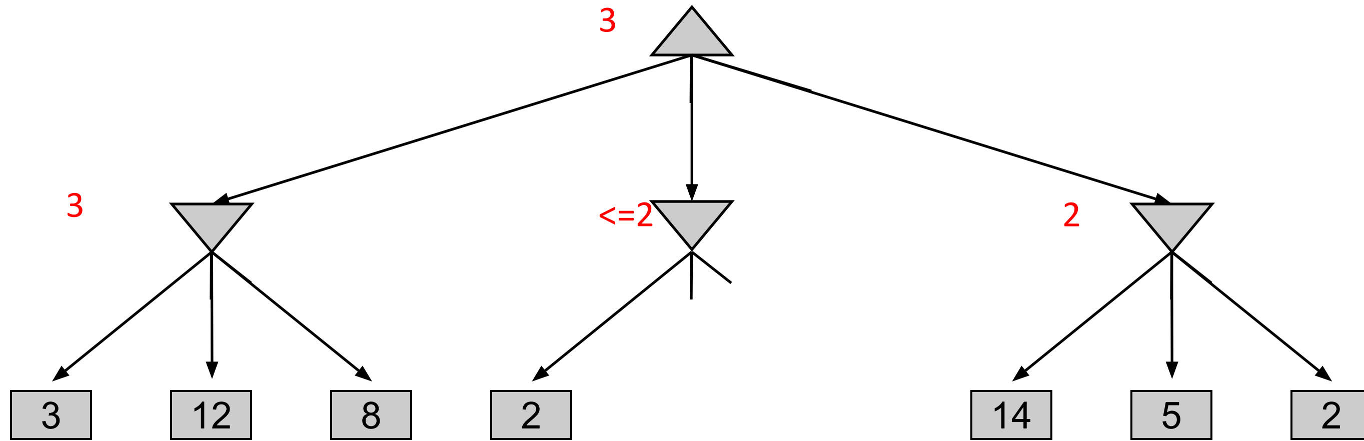
- Deterministic, zero-sum games:
  - Tic-tac-toe, chess, checkers
  - One player maximizes result
  - The other minimizes result
- Minimax search:
  - A state-space search tree
  - Players alternate turns
  - Compute each node's minimax value: the best achievable utility against a rational (optimal) adversary



Minimax Example

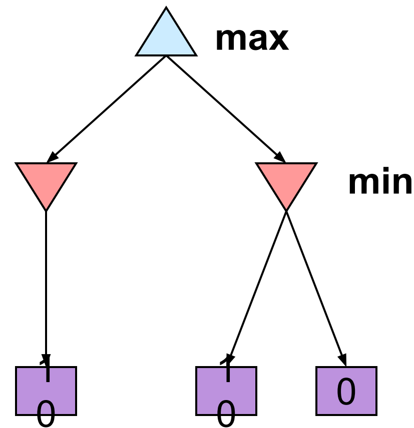


Minimax Example: Pruning

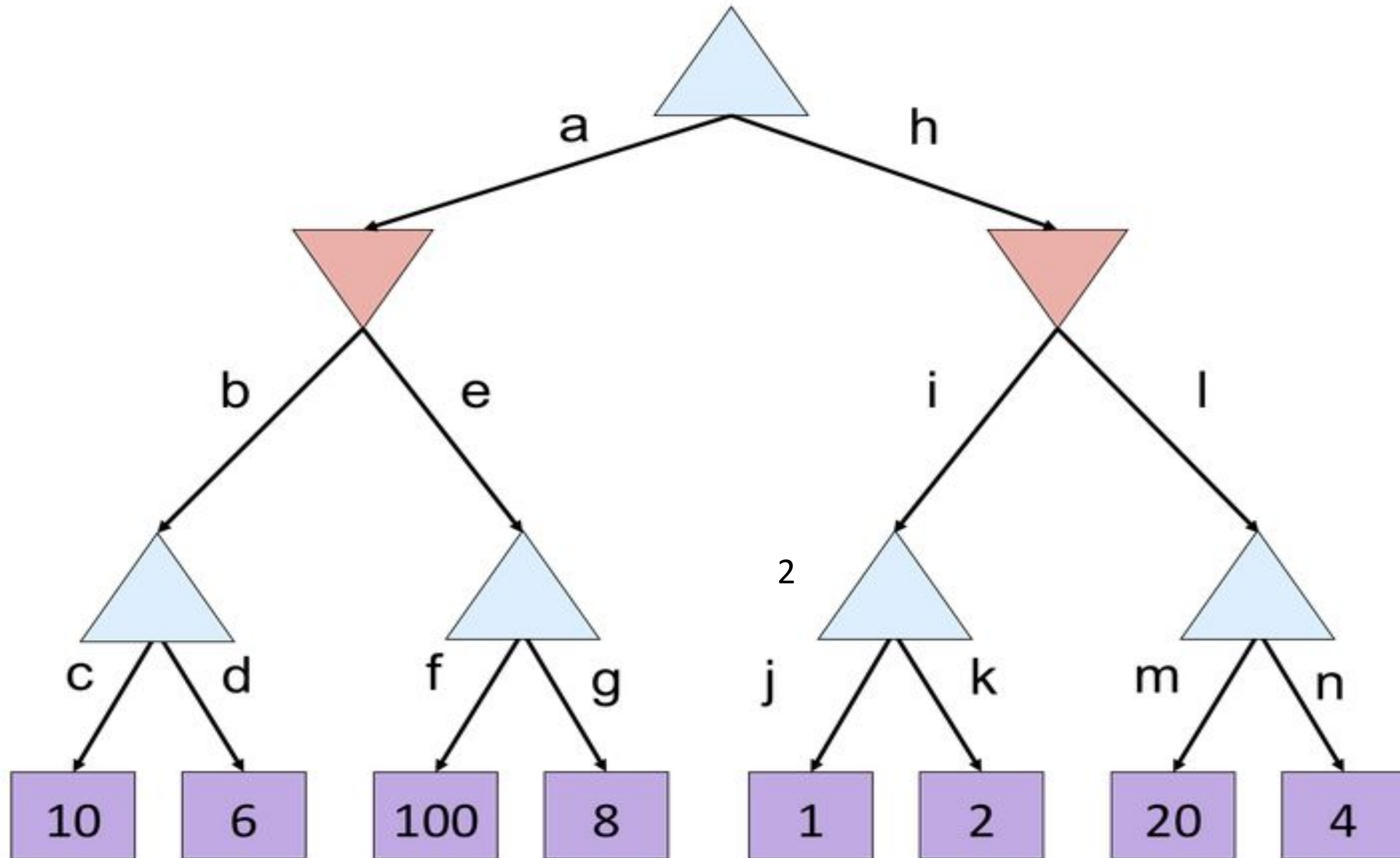


Alpha-Beta Pruning Properties

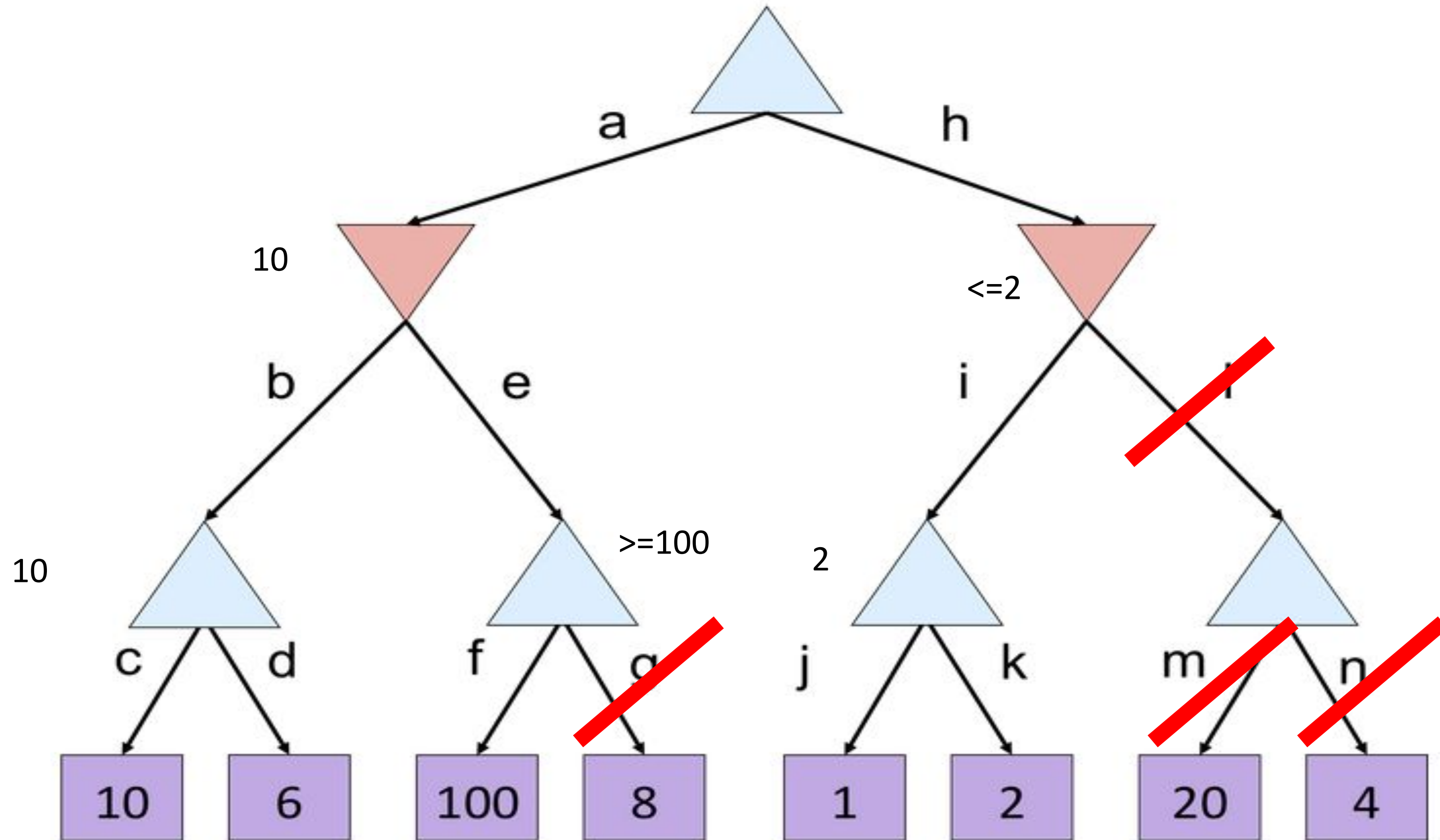
- This pruning has no effect on minimax value computed for the root!
- Values of intermediate nodes might be wrong
  - Important: children of the root may have the wrong value
  - So the most naïve version won't let you do action selection
- Good child ordering improves effectiveness of pruning
- With "perfect ordering":
  - Time complexity drops to $O(b^{m/2})$
  - Doubles solvable depth!
  - Full search of, e.g. chess, is still hopeless…
- This is a simple example of metareasoning (computing about what to compute)



Alpha-Beta Quiz 2



Alpha-Beta Quiz 2

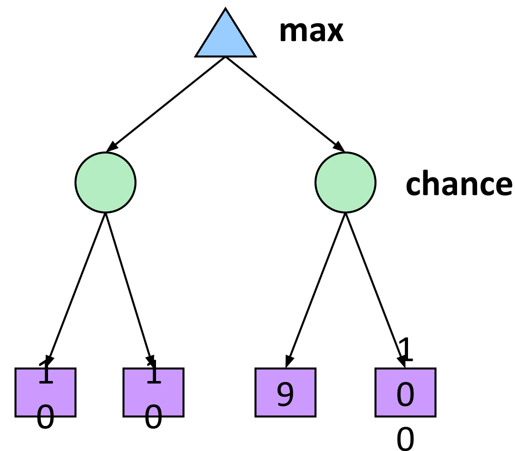


Expectimax Search

- Why wouldn't we know what the result of an action will be?
  - Explicit randomness: rolling dice
  - Unpredictable opponents: the ghosts respond randomly
  - Unpredictable humans: humans are not perfect
  - Actions can fail: when moving a robot, wheels might slip

- Values should now reflect average-case (expectimax) outcomes, not worst-case (minimax) outcomes

- Expectimax search: compute the average score under optimal play
  - Max nodes as in minimax search
  - Chance nodes are like min nodes but the outcome is uncertain
  - Calculate their expected utilities
  - I.e. take weighted average (expectation) of children



Remaining Topics

Bayes Nets:
- Inference by enumeration
- Variable elimination
- D-separation
- Sampling approaches

HMMs:
- Forward algorithm
- Viterbi algorithm
- Particle filtering

Decision networks and VPIs

Out of scope: learning theory, decision tree classifiers, details of non-SGD optimizers (e.g., NAG, Adagrad, Adam), NLP/CV/RL