CS221 Course Reader

v0.3 - Chapterized skeleton (lectures 1-18)

A structured reader built from the CS221 / Spring 2020 lecture deck sequence. Each chapter provides a short conceptual summary, key definitions, core algorithms (in pseudocode), common failure modes, and checkpoint questions for review.

Contents

- Lecture 1: Overview
- Lecture 2: Machine Learning I
- Lecture 3: Machine Learning II
- Lecture 4: Machine Learning III
- Lecture 5: Search I
- Lecture 6: Search II
- Lecture 7: MDPs I
- Lecture 8: MDPs II
- Lecture 9: Games I
- Lecture 10: Games II
- Lecture 11: CSPs I
- Lecture 12: CSPs II
- Lecture 13: Bayesian Networks I
- Lecture 14: Bayesian Networks II
- Lecture 15: Bayesian Networks III
- Lecture 16: Logic I
- Lecture 17: Logic II
- Lecture 18: Deep Learning

Lecture 1: Overview

Concept summary

Artificial intelligence is best viewed as a separation of concerns: a domain model defines states, actions, uncertainty, and objectives; an inference or decision procedure computes an optimal prediction, plan, or policy. Across the course, the same themes recur: exploit structure (locality, conditional independence, factorization), trade exactness for scalability (approximate search, sampling, function approximation), and quantify uncertainty when the world is partially observed. This chapter sets up the core paradigms and the common abstractions that will reappear in search, MDPs, games, CSPs, Bayesian networks, logic, and learning.

Key definitions and notation

- Model vs inference: specify how the world works, then compute consequences.
- State: a sufficient summary of the past for future prediction/control.
- Objective: cost to minimize or utility/reward to maximize.
- Exact vs approximate inference: correctness guarantees vs tractability.
- Factorization: represent a global function as a product/sum of local pieces.

Core algorithms

A unifying template

```
Given: representation (states, actions, factors), objective, and constraints
Choose: an algorithm family (search / DP / elimination / sampling / learning)
Exploit: structure (acyclicity, locality, monotonicity, independence)
Return: plan, policy, proof, or predictor with diagnostics
```

Common failure modes and sanity checks

- Conflating modeling errors with inference errors (bad model, good optimizer).
- Using the wrong state abstraction (non-Markov state causes 'surprises').
- Ignoring compute constraints; exact methods can be correct but unusable.
- Optimizing the wrong objective (proxy loss misaligned with evaluation).

Checkpoints

- 1. Give an example where a better model beats a better inference algorithm.
- 2. What makes a state representation Markov?
- 3. Name two types of structure that make inference easier.
- 4. When is approximation preferable to exactness in practice?
- 5. What is a factorization and why is it powerful?
- 6. Describe one diagnostic you would want from any inference algorithm.

Lecture 2: Machine Learning I

Concept summary

Machine learning casts prediction as empirical risk minimization: choose a parameterized predictor, define a loss function, and optimize parameters to reduce training loss. Linear predictors score examples using a dot product between weights and features. Because 0-1 loss is hard to optimize directly, we replace it with convex surrogates (for example hinge loss) and use gradient-based optimization. Stochastic gradient descent scales learning by using noisy per-example gradients instead of full passes over the dataset.

Key definitions and notation

- Predictor f(x) maps input x to output y (classification or regression).
- Feature map phi(x): transforms raw input into numeric features.
- Training loss: average loss over the training set.
- Hinge loss: max(0, 1 - y * (w dot phi(x))) for y in {-1,+1}.
- Learning rate (step size) eta controls update magnitude.
- SGD: update weights using one (or a minibatch of) example(s) at a time.

Core algorithms

Batch gradient descent

```
initialize w
repeat:
  g <- (1/n) * sum_i d/dw loss(w; x_i, y_i)
  w <- w - eta * g
```

Stochastic gradient descent (SGD)

```
initialize w
repeat:
  pick training example (x_i, y_i)
  g <- d/dw loss(w; x_i, y_i)
  w <- w - eta * g
```

Common failure modes and sanity checks

- Step size too large: divergence or oscillation; too small: slow progress.
- No feature normalization: gradients become ill-conditioned.
- Training loss decreases but validation loss increases (overfitting).
- Using non-differentiable losses without a subgradient argument.
- Data leakage: peeking at test data during tuning.

Checkpoints

- 1. Why is 0-1 loss difficult for gradient methods?
- 2. Explain the difference between training loss and test error.
- 3. What does hinge loss encourage geometrically?
- 4. Why can SGD outperform batch GD in wall-clock time?
- 5. How does feature scaling affect optimization?
- 6. Give one reason to add regularization.

Lecture 3: Machine Learning II

Concept summary

Features are the interface between domain knowledge and learning: the same linear learner can solve very different problems if phi(x) changes. Neural networks reduce manual feature design by learning compositions of nonlinear transformations; backpropagation computes gradients efficiently by reusing intermediate computations. Nearest-neighbor methods represent an alternative regime: almost no training, but expensive prediction and sensitivity to distance metrics.

Key definitions and notation

- Feature template: systematic recipe to generate related features.
- Nonlinearity: enables a network to represent non-linear decision boundaries.
- Backpropagation: reverse-mode differentiation for layered computations.
- Activation function: e.g., ReLU, sigmoid; introduces nonlinearity.
- kNN: predict using labels of the k closest training points under a metric.
- Bias term: constant feature allowing an affine shift.

Core algorithms

Backprop (high level)

```
Forward pass: compute activations layer by layer
Backward pass: propagate gradients from output to input via chain rule
Update: w <- w - eta * grad_w loss
```

k-nearest neighbors

```
store training set
predict(x):
  find k training points closest to x
  return majority label (or average value)
```

Common failure modes and sanity checks

- High-dimensional distances can become uninformative for kNN (curse of dimensionality).
- Neural nets can overfit without regularization or sufficient data.
- Saturated activations can yield tiny gradients (vanishing gradients).
- Poor initialization can slow or destabilize training.
- Feature hashing/collision and sparsity issues if not managed carefully.

Checkpoints

- 1. What is the role of phi(x) in a linear predictor?
- 2. Why do neural nets need nonlinearities between layers?
- 3. What does the chain rule buy you computationally in backprop?
- 4. When might kNN be preferable to a learned parametric model?
- 5. Name two regularization strategies for neural nets.
- 6. What is a feature template and why is it useful?

Lecture 4: Machine Learning III

Concept summary

Generalization is the central concern: we want models that perform well on unseen data. This lecture focuses on evaluation protocols (train/validation/test), cross-validation, regularization, and practical debugging of learning systems. It also introduces unsupervised learning objectives and the notion that representation and objective choice often dominate algorithmic details.

Key definitions and notation

- Generalization gap: difference between training and test performance.
- Validation set: used to tune hyperparameters; test set: used once for final evaluation.
- Regularization: penalize model complexity (e.g., L2 weight decay).
- Cross-validation: rotate validation folds to estimate performance robustly.
- Bias-variance tradeoff: underfitting vs overfitting.
- Unsupervised learning: learn from x without labels y (e.g., clustering, dimensionality reduction).

Core algorithms

Train/validation/test workflow

```
Split data -> train model on train
Tune hyperparameters on validation
Lock choices -> report final metric on test
```

k-fold cross-validation

```
Partition data into k folds
For each fold j:
  train on all folds except j
  evaluate on fold j
Average metrics across folds
```

Common failure modes and sanity checks

- Repeatedly tuning on the test set (invalidates reported performance).
- Reporting a single metric without confidence or variance estimates.
- Confusing optimization failure with model misspecification.
- Not inspecting errors: same aggregate metric can hide systematic failures.
- Ignoring class imbalance or calibration where it matters.

Checkpoints

- 1. Why is the test set 'sacred'?
- 2. When do you prefer cross-validation over a fixed split?
- 3. What does L2 regularization do to weights and why?
- 4. Give an example of data leakage.
- 5. How do bias and variance show up in learning curves?
- 6. Name two diagnostics beyond overall accuracy.

Lecture 5: Search I

Concept summary

Search solves deterministic planning problems by exploring states connected by actions. The key modeling decision is the state representation, which must be rich enough to support planning but compact enough for computation. We distinguish tree search from graph search and analyze completeness and optimality. Uniform-cost search (Dijkstra) finds lowest-cost plans when action costs are nonnegative.

Key definitions and notation

- State: representation of a configuration; start state and goal test define the problem.
- Action: transforms one state into another; may have a step cost.
- Frontier: data structure of nodes to be explored next.
- Tree search: may revisit states; graph search: tracks explored set.
- Completeness: finds a solution if one exists.
- Optimality: finds a minimum-cost solution.

Core algorithms

Breadth-first search (unit costs)

```
enqueue start
while frontier not empty:
  pop oldest node
  if goal: return plan
  expand, enqueue successors
```

Uniform-cost search (Dijkstra)

```
push start with priority g(start)=0
while frontier not empty:
  pop node with smallest g
  if goal: return plan
  for each successor s':
    if better g(s') found: update and push
```

Common failure modes and sanity checks

- Using BFS with non-unit costs (can be suboptimal).
- Not handling repeated states (exponential blowup, incorrectness in graphs).
- Negative step costs break UCS assumptions.
- State lacks needed information (plan depends on history).
- Memory exhaustion: frontier grows rapidly with branching factor.

Checkpoints

- 1. Tree search vs graph search: when do they differ?
- 2. Why does UCS require nonnegative costs?
- 3. Give a domain where state must include more than position.
- 4. What is the role of an explored set?
- 5. How does branching factor affect runtime?
- 6. When is BFS optimal?

Lecture 6: Search II

Concept summary

Search can be accelerated by exploiting structure. Dynamic programming works when the state graph is acyclic or admits a suitable decomposition: compute optimal values for subproblems and reuse them. A* search uses a heuristic h(s) estimating remaining cost to guide exploration while retaining optimality when the heuristic is admissible. Designing heuristics via relaxations provides systematic lower bounds.

Key definitions and notation

- Dynamic programming (DP): solve subproblems once and reuse their values.
- Heuristic h(s): estimate of cost-to-go from state s to a goal.
- A* priority: f(s) = g(s) + h(s), where g is cost-so-far.
- Admissible heuristic: never overestimates true remaining cost.
- Consistent heuristic: obeys triangle inequality; reduces node re-expansions.
- Relaxation: remove constraints to get an easier problem whose solution is a lower bound.

Core algorithms

A* search

```
push start with priority f=g+h
while frontier not empty:
  pop node with smallest f
  if goal: return plan
  relax edges; if better g found, update and push
```

DP for shortest paths in DAG (sketch)

```
topologically order states
for states in reverse topological order:
  V(s) <- min_a [ cost(s,a) + V(succ(s,a)) ]
```

Common failure modes and sanity checks

- Inadmissible heuristic can yield suboptimal solutions.
- Inconsistent heuristic without re-opening can break optimality.
- Heuristic too weak: A* degenerates toward UCS.
- Assuming DP applies when cycles exist (requires different treatment).
- Cost modeling mismatch: optimizing a proxy path cost.

Checkpoints

- 1. Define admissibility and consistency for heuristics.
- 2. Why can A* be faster than UCS?
- 3. How can you construct a heuristic by relaxing constraints?
- 4. When does DP apply directly to shortest-path problems?
- 5. What can go wrong if you never re-open states in A*?
- 6. Give an example of a problem-specific heuristic.

Lecture 7: MDPs I

Concept summary

Markov decision processes model sequential decision making under uncertainty. An MDP specifies states, actions, transition probabilities, and rewards. Policies map states to actions; value functions quantify expected return. The Bellman optimality equations characterize optimal values and policies. Value iteration computes optimal values by repeated Bellman backups.

Key definitions and notation

- MDP: (S, A, T, R, gamma) with transitions T(s,a,s') and reward R(s,a,s').
- Policy pi(a|s): distribution over actions given state.
- Value V^pi(s): expected discounted return following pi from s.
- Q-value Q^pi(s,a): expected return starting with action a then following pi.
- Bellman backup: V(s) <- max_a sum_{s'} T(s,a,s') [R + gamma V(s')].
- Discount gamma in [0,1): trades immediate vs future reward.

Core algorithms

Value iteration

```
initialize V(s)=0
repeat:
  for each state s:
    V_new(s) <- max_a sum_{s'} T(s,a,s') [ R(s,a,s') + gamma * V(s') ]
  V <- V_new
until convergence
```

Extract greedy policy

```
pi(s) <- argmax_a sum_{s'} T(s,a,s') [ R + gamma V(s') ]
```

Common failure modes and sanity checks

- Non-Markov state: transition depends on hidden history.
- Ignoring discounting can cause divergence with infinite horizons.
- Using a coarse state abstraction can destroy optimality.
- Confusing rewards with values (rewards are local, values are long-term).
- Stopping value iteration too early without a convergence criterion.

Checkpoints

- 1. What makes an MDP 'Markov'?
- 2. Write the Bellman optimality equation for V*(s).
- 3. How do you obtain a policy once you have V?
- 4. What is the effect of gamma approaching 1?
- 5. Why are values not the same as rewards?
- 6. When would you prefer policy iteration over value iteration?

Lecture 8: MDPs II

Concept summary

Reinforcement learning solves MDPs when transitions and rewards are unknown. Model-free methods learn value functions directly from experience using temporal-difference (TD) updates. Q-learning learns an optimal action-value function off-policy; SARSA learns on-policy. Exploration is essential: policies must gather information while still accumulating reward, typically via epsilon-greedy or softmax action selection.

Key definitions and notation

- Episode: a trajectory (s0,a0,r0,s1,...) generated by interaction.
- TD error: delta = r + gamma V(s') - V(s).
- Q-learning: off-policy TD control targeting max_a' Q(s',a').
- SARSA: on-policy TD control targeting Q(s',a') under the current policy.
- Exploration-exploitation tradeoff: learn vs earn.
- Model-based RL: learn T,R then plan; model-free RL: learn values/policy directly.

Core algorithms

TD(0) value learning

```
observe transition (s,a,r,s')
V(s) <- V(s) + alpha * (r + gamma*V(s') - V(s))
```

Q-learning

```
observe transition (s,a,r,s')
Q(s,a) <- Q(s,a) + alpha * (r + gamma * max_{a'} Q(s',a') - Q(s,a))
```

Epsilon-greedy exploration

```
with prob epsilon: pick random action
else: pick argmax_a Q(s,a)
```

Common failure modes and sanity checks

- Insufficient exploration: converges to a bad local behavior.
- Learning rate too high or not decayed: unstable updates.
- Function approximation can destabilize off-policy learning.
- Reward scaling issues: very large rewards cause exploding updates.
- Partial observability: RL assumptions fail without memory/belief state.

Checkpoints

- 1. Explain the TD error in words.
- 2. Q-learning vs SARSA: what is the key difference?
- 3. Why is exploration necessary even with a good current policy?
- 4. What does 'off-policy' mean?
- 5. How does model-based RL differ from model-free RL?
- 6. Name one failure mode when using function approximation.

Lecture 9: Games I

Concept summary

Adversarial games extend search to settings with opponents. Minimax computes the optimal strategy in deterministic, turn-taking, zero-sum games under perfect play. Because game trees grow exponentially, practical agents combine limited-depth search with evaluation functions. Alpha-beta pruning preserves optimality while pruning branches that cannot affect the final decision.

Key definitions and notation

- Zero-sum game: utilities of players sum to zero; one maximizes, the other minimizes.
- Game tree: alternating max and min layers over states.
- Minimax value: utility assuming optimal play by both sides.
- Evaluation function: heuristic value for non-terminal states at cutoff depth.
- Alpha-beta pruning: tracks bounds (alpha, beta) to prune provably irrelevant branches.
- Branching factor b and depth d drive complexity.

Core algorithms

Minimax

```
V(s):
  if terminal: return U(s)
  if MAX to play: return max_{a} V(succ(s,a))
  if MIN to play: return min_{a} V(succ(s,a))
```

Alpha-beta pruning (sketch)

```
search with bounds alpha (best for MAX) and beta (best for MIN)
prune when alpha >= beta
```

Common failure modes and sanity checks

- Evaluation function misleads at shallow depth (horizon effect).
- Ignoring stochasticity or hidden information in the model.
- Not ordering moves: alpha-beta effectiveness depends on ordering.
- Assuming minimax applies to non-zero-sum objectives without modification.
- Excessive depth: exponential growth overwhelms compute.

Checkpoints

- 1. Define minimax value and its assumptions.
- 2. Why does alpha-beta pruning not change the returned action?
- 3. What is the horizon effect?
- 4. How does move ordering affect alpha-beta performance?
- 5. When do you prefer expectimax over minimax?
- 6. Give an example of a good evaluation feature.

Lecture 10: Games II

Concept summary

Learning can replace hand-designed evaluation functions by fitting value functions from experience, often using temporal-difference updates and self-play. The lecture also moves beyond turn-taking, zero-sum games to simultaneous-move and non-zero-sum settings. Mixed strategies randomize actions; the minimax theorem guarantees equilibrium values in zero-sum matrix games, while Nash equilibria generalize to non-zero-sum games.

Key definitions and notation

- TD learning in games: update V(s) toward r + gamma V(s').
- Self-play: generate training data by playing against yourself/current policy.
- Simultaneous game: players choose actions at the same time; payoff matrix describes outcomes.
- Mixed strategy: distribution over actions.
- Minimax theorem: max-min equals min-max in finite zero-sum games.
- Nash equilibrium: no player can improve unilaterally.

Core algorithms

TD update for value function

```
after transition s -> s':
V(s) <- V(s) + alpha * (r + gamma V(s') - V(s))
```

Mixed strategy value (matrix game)

```
Given payoff matrix M:
value(p,q) = p^T M q
choose p to maximize min_q value(p,q)
```

Common failure modes and sanity checks

- Non-stationarity during self-play: the target moves as the opponent changes.
- Overfitting to a narrow opponent distribution.
- Assuming unique equilibrium; many games have multiple equilibria.
- Confusing optimal responses with equilibrium strategies in non-zero-sum games.
- Using deterministic policies where randomization is required.

Checkpoints

- 1. Why can self-play generate useful training data?
- 2. What is a mixed strategy and why use it?
- 3. State the minimax theorem informally.
- 4. What changes in non-zero-sum games compared to zero-sum?
- 5. Define Nash equilibrium in one sentence.
- 6. Give an example where deterministic play is exploitable.

Lecture 11: CSPs I

Concept summary

Constraint satisfaction problems (CSPs) specify variables with domains and constraints over subsets of variables. Weighted CSPs or factor graphs assign a weight to each complete assignment via a product of local factors; the goal is to find the maximum-weight assignment. Backtracking search is the canonical exact solver. It becomes practical by using variable ordering, value ordering, and early pruning via local consistency checks.

Key definitions and notation

- Variable X_i with domain Dom(X_i).
- Constraint/factor f over a subset of variables; returns 0/1 (hard) or a weight (soft).
- Assignment: mapping from variables to values; partial assignment assigns a subset.
- Consistency: property that enables pruning (e.g., arc consistency).
- Backtracking: depth-first assignment with recursion and pruning.
- Heuristics: MRV (min remaining values), degree heuristic, LCV (least constraining value).

Core algorithms

Backtracking search (sketch)

```
choose unassigned variable X
for value v in order:
  if consistent with current assignment:
    assign X=v
    recurse
    undo assignment
```

Arc consistency (AC-3, sketch)

```
initialize queue with all arcs (Xi,Xj)
while queue not empty:
  if revise(Xi,Xj):
    add arcs (Xk,Xi) for neighbors Xk
```

Common failure modes and sanity checks

- Poor variable ordering can cause exponential blowups.
- Constraints evaluated too late; missing early pruning.
- Using backtracking without memoization in highly symmetric problems.
- Domains too large without decomposition or propagation.
- Confusing satisfiable CSP with optimization (max-weight) CSP.

Checkpoints

- 1. What is the difference between CSP and weighted CSP?
- 2. Why do ordering heuristics matter?
- 3. What does arc consistency guarantee?
- 4. Give an example of a constraint that is not binary.
- 5. When can propagation be more valuable than search depth?
- 6. Name two CSP applications.

Lecture 12: CSPs II

Concept summary

When exact search is too expensive, CSPs admit approximate alternatives. Beam search keeps only the top-K partial assignments. Local search methods operate on complete assignments and iteratively improve them, often using randomization to escape local optima. Exact inference can also exploit graph structure through conditioning (splitting into independent components) and elimination, which generalizes dynamic programming by marginalizing or maximizing out variables to create new factors.

Key definitions and notation

- Beam search: keep K best partial assignments at each depth.
- Local search: start from a full assignment and apply local moves to improve score.
- Min-conflicts: choose a variable in conflict and change it to reduce conflicts.
- Conditioning: fix variable values to decompose the factor graph.
- Variable elimination: remove a variable by combining neighboring factors then maximizing/summing it out.
- Induced width/treewidth: governs elimination complexity.

Core algorithms

Beam search

```
beam <- {empty assignment}
for i in 1..n variables:
  expand each assignment in beam by assigning next variable
  beam <- top K expansions by score
```

Variable elimination for max

```
pick variable X
g <- product of all factors involving X
h <- max_X g   # new factor over neighbors of X
replace factors involving X with h
```

Common failure modes and sanity checks

- Beam search can discard the optimal partial assignment early.
- Local search can get stuck; random restarts often needed.
- Elimination order can cause factor blowup (treewidth).
- Conditioning can still be exponential if too many variables fixed.
- Approximate methods need diagnostics; otherwise silent failure.

Checkpoints

- 1. What is the beam width K controlling?
- 2. Why does randomness help in local search?
- 3. What is treewidth and why does it matter?
- 4. How does conditioning create independence?
- 5. Elimination for max vs sum: what changes?
- 6. Give an example where local search excels over backtracking.

Lecture 13: Bayesian Networks I

Concept summary

Bayesian networks represent joint distributions using a directed acyclic graph where each node has a local conditional distribution given its parents. This factorization enables compact modeling and inference via conditional independence. Inference queries ask for posterior probabilities conditioned on evidence. A central qualitative phenomenon is explaining away: observing a common effect induces dependence between alternative causes.

Key definitions and notation

- Bayesian network: DAG + conditional probability tables (CPTs) P(X|Parents(X)).
- Joint factorization: P(x1..xn) = product_i P(x_i | parents_i).
- Evidence E=e: observed variables fixed to values.
- Query Q: variables of interest; goal is P(Q | E=e).
- Conditional independence: graph structure implies independences (d-separation).
- Explaining away: causes become negatively correlated given their common effect.

Core algorithms

Variable elimination (high level)

```
Start with factors from CPTs, incorporate evidence
Repeat:
  pick a variable Z to eliminate (not in query/evidence)
  multiply factors containing Z -> g
  sum out Z from g -> h
  replace those factors with h
Normalize result
```

Compute posterior

```
P(Q|E=e) = alpha * P(Q,E=e)  where alpha normalizes
```

Common failure modes and sanity checks

- Bad elimination order can blow up intermediate factors.
- Treating dependent variables as independent (incorrect priors).
- Confusing likelihood P(E|H) with posterior P(H|E).
- Overconfident CPTs with little data (needs smoothing).
- Ignoring numerical stability in products of small probabilities.

Checkpoints

- 1. How does a BN factorize a joint distribution?
- 2. What is a posterior and how is it computed from a joint?
- 3. Explain explaining away with a two-cause one-effect example.
- 4. Why does elimination order matter?
- 5. What is d-separation used for?
- 6. How does evidence enter variable elimination?

Lecture 14: Bayesian Networks II

Concept summary

Temporal models such as hidden Markov models (HMMs) are Bayesian networks with repeated structure. Forward-backward computes smoothed marginals efficiently by dynamic programming. When exact inference is hard, sampling methods approximate posteriors. Particle filtering tracks a weighted set of samples online; resampling prevents weight collapse. Gibbs sampling is a Markov chain Monte Carlo method that updates one variable at a time conditioned on the rest.

Key definitions and notation

- HMM: latent state X_t with transitions P(X_t|X_{t-1}) and emissions P(E_t|X_t).
- Filtering: P(X_t | e_{1:t}); smoothing: P(X_t | e_{1:T}).
- Forward message: alpha_t(x) proportional to P(x, e_{1:t}).
- Backward message: beta_t(x) proportional to P(e_{t+1:T} | x).
- Particle filter: weighted samples approximating filtering distribution.
- Gibbs sampling: repeatedly sample each variable from its conditional distribution given others.

Core algorithms

Forward-backward (sketch)

```
Forward: alpha_t <- normalize( emission_t * (T^T alpha_{t-1}) )
Backward: beta_t <- T (emission_{t+1} * beta_{t+1})
Smoothed: P(X_t|e_{1:T}) proportional to alpha_t * beta_t
```

Particle filtering

```
for each time t:
  propagate particles via transition model
  weight by likelihood of evidence
  resample proportional to weights
```

Gibbs sampling

```
initialize all hidden variables
repeat:
  for each variable Z:
    sample Z ~ P(Z | MarkovBlanket(Z))
```

Common failure modes and sanity checks

- Particle degeneracy: weights collapse without resampling.
- Poor proposal distributions yield high-variance importance weights.
- Gibbs mixing can be slow in strongly coupled models.
- Exact inference scales poorly with high treewidth.
- For HMMs, forgetting to normalize leads to numerical underflow.

Checkpoints

- 1. Difference between filtering and smoothing?
- 2. What do forward and backward messages represent?
- 3. Why do particle filters need resampling?
- 4. What is a Markov blanket in a BN?
- 5. When might Gibbs sampling mix slowly?
- 6. How does repeated structure help inference?

Lecture 15: Bayesian Networks III

Concept summary

This lecture focuses on learning probabilistic models. With labeled data, maximum likelihood estimation learns parameters of CPTs, but naive estimates can be overconfident when data are scarce. Laplace smoothing corrects this by adding pseudo-counts. With hidden variables, expectation-maximization (EM) alternates between computing expected sufficient statistics under the current model (E-step) and maximizing parameters given those expectations (M-step). Parameter sharing and structured factorization reduce sample complexity.

Key definitions and notation

- Maximum likelihood estimation (MLE): choose parameters maximizing P(data | theta).
- Sufficient statistics: counts needed to estimate CPT parameters.
- Laplace smoothing: add-one (or add-alpha) pseudo-counts to counts.
- Hidden variables: variables not observed in training data.
- EM: iterative algorithm for latent-variable MLE.
- Parameter sharing: tie parameters across contexts to reduce variance.

Core algorithms

Laplace-smoothed CPT estimate

```
theta_{x|u} = (count(x,u) + alpha) / (count(u) + alpha * |Dom(X)|)
```

EM (sketch)

```
initialize parameters theta
repeat:
  E-step: compute expected counts under P(H | E, theta)
  M-step: update theta from expected counts
until convergence
```

Common failure modes and sanity checks

- MLE overfits with tiny counts (probabilities 0 or 1).
- EM can get stuck in local optima; initialization matters.
- Unidentifiable models: different parameters explain data equally well.
- Posterior computations in E-step can be expensive (needs structure/approximation).
- Mis-specified latent structure yields misleading interpretations.

Checkpoints

- 1. Why does Laplace smoothing help?
- 2. What is computed in the E-step of EM?
- 3. Why can EM converge to a local optimum?
- 4. What are sufficient statistics for a CPT?
- 5. How does parameter sharing reduce overfitting?
- 6. Give an example of a hidden variable in a real system.

Lecture 16: Logic I

Concept summary

Logic provides a language for representing knowledge and a set of sound inference rules for deriving consequences. In propositional logic, formulas are built from Boolean variables and connectives. Semantics assigns truth values under an interpretation (model). Inference can be done by model checking (truth tables) or by proof systems such as resolution. Algorithmic solvers such as DPLL exploit structure in conjunctive normal form (CNF) to decide satisfiability efficiently in many practical cases.

Key definitions and notation

- Syntax: the set of well-formed formulas; semantics: what formulas mean (truth under models).
- Model: assignment of truth values to variables.
- Entailment: KB entails phi if every model of KB is a model of phi.
- Satisfiable: exists a model making a formula true; unsatisfiable otherwise.
- CNF: conjunction of clauses; clause is disjunction of literals.
- Resolution: inference rule combining clauses to produce a new implied clause.

Core algorithms

Resolution (propositional)

```
(A or x) and (B or not x)  =>  (A or B)
```

DPLL (sketch)

```
if all clauses satisfied: SAT
if any clause empty: UNSAT
choose variable x
recurse with x=true; if SAT return SAT
recurse with x=false; return result
```

Common failure modes and sanity checks

- Confusing entailment with implication in one direction only.
- Failing to convert correctly to CNF (changes meaning).
- Resolution without strategy can explode in clause count.
- DPLL without propagation/heuristics can be exponential.
- Model checking is complete but infeasible for large variable sets.

Checkpoints

- 1. Define entailment in terms of models.
- 2. What is CNF and why do SAT solvers use it?
- 3. State the resolution rule and why it is sound.
- 4. What does DPLL decide?
- 5. Why is model checking exponential?
- 6. Give a simple KB and a query it entails.

Lecture 17: Logic II

Concept summary

First-order logic (FOL) extends propositional logic with predicates, functions, variables, and quantifiers, enabling compact statements about objects and relations. Inference requires handling variable substitutions via unification. Many practical systems restrict to fragments that support efficient inference, such as Horn clauses enabling forward and backward chaining. Resolution generalizes to FOL when paired with unification and standardization-apart.

Key definitions and notation

- Predicate P(t1,...,tk): relation over terms; term can be constant, variable, or function application.
- Quantifiers: forall x (universal), exists x (existential).
- Substitution: map variables to terms; apply to a formula.
- Unification: find a substitution making two expressions identical.
- Horn clause: clause with at most one positive literal.
- Forward chaining / backward chaining: data-driven vs goal-driven inference on Horn KBs.

Core algorithms

Unification (sketch)

```
unify(expr1, expr2):
  if both same variable/constant: ok
  if variable vs term: bind variable (occurs-check)
  if function/predicate: unify arguments recursively
  else: fail
```

Backward chaining (Horn)

```
prove(goal):
  if goal in facts: success
  for each rule (premises => goal):
    prove all premises with substitutions
```

Common failure modes and sanity checks

- Ignoring occurs-check yields incorrect infinite terms in unification.
- Quantifier scope mistakes change meaning dramatically.
- Resolution in full FOL is semi-decidable (may not terminate).
- Naive chaining can loop without memoization/loop checks.
- Overusing FOL where probabilistic uncertainty dominates.

Checkpoints

- 1. What does unification compute?
- 2. Why do Horn clauses matter computationally?
- 3. Forward vs backward chaining: when prefer each?
- 4. Give an example where quantifier scope matters.
- 5. What is the occurs-check preventing?
- 6. Why can full FOL inference fail to terminate?

Lecture 18: Deep Learning

Concept summary

Deep learning uses multi-layer neural networks to learn hierarchical representations. Feedforward networks learn nonlinear functions via backpropagation; convolutional networks exploit spatial locality and weight sharing for images; sequence models (RNNs, LSTMs, attention) capture temporal structure. Unsupervised methods such as autoencoders learn representations without labels. Practical performance depends heavily on optimization choices, regularization, and data scale.

Key definitions and notation

- Feedforward network: layered affine transforms + nonlinearities.
- Convolution: local receptive fields with shared filters; yields translation equivariance.
- Pooling: downsampling to gain invariance and reduce computation.
- Sequence model: processes ordered inputs; attention selectively aggregates context.
- Autoencoder: encoder compresses, decoder reconstructs; trains by reconstruction loss.
- Regularization: dropout, weight decay, early stopping, data augmentation.

Core algorithms

Backprop through a network (schematic)

```
Forward: compute activations a^l for layers l
Backward: compute gradients dL/da^l and dL/dW^l via chain rule
Update: W^l <- W^l - eta * dL/dW^l
```

Convolution (conceptual)

```
for each spatial position:
  output[pos] = sum_{i,j,c} filter[i,j,c] * input[pos+i,pos+j,c]
```

Common failure modes and sanity checks

- Overfitting without sufficient data augmentation/regularization.
- Vanishing/exploding gradients in deep or recurrent networks.
- Train/serving skew: preprocessing differs between training and deployment.
- Mis-calibration: high confidence but wrong predictions.
- Ignoring compute/memory constraints; architecture not deployable.

Checkpoints

- 1. Why do CNNs use weight sharing?
- 2. What problem does attention solve compared to fixed-window models?
- 3. How does dropout affect training vs inference?
- 4. Why can deep nets overfit even with low training loss?
- 5. What is an autoencoder optimizing?
- 6. Name two practical issues that are not solved by 'more layers'.