CS2109S Cheatsheet AY24/25 —— @JasonYapzx

1. Intelligent Agents — PEAS

- **Performance Measure:** Optimizing for whom, the intended effect/cost
- **Environment**
  - Observable: Complete environment state accessible at each point of time
  - Deterministic vs Stochastic: Deterministic (next state predictable) vs Stochastic (randomness involved, e.g., other agents' actions)
  - Episodic vs Sequential: independent actions vs actions depend on past
  - Static vs Dynamic: Static (environment unchanged during deliberation) vs Dynamic (environment changes with time, agent's performance changes)
  - Discrete vs Continuous: Discrete (finite actions/percepts) vs Continuous (infinitely many possibilities)
  - Single vs Multi-agent: Single-agent (operates alone) vs Multi-agent (interacts with other agents)
- **Actuators:** Components that perform actions
- **Sensors:** Components that perceive the environment's inputs (percepts)

Structure of Agents

- **Simple Reflex:** If up empty → go up, down empty go down (acts on reflex)
- **Model-based:** If goal, pick the action, else if not die, pick the action
- **Goal-based:** Maps all possible states, picks one that reaches eventual goal:
  - Done by mapping out all states, then backtracking, on top of model-based
- **Utility-based:** Maximizes utility on top of goal-based agent
- **Learning:** Uses a *performance element* to select external actions.

2. Search Algorithms

Complete: always return solution if exists, Optimal: always find least-cost solution. b branching factor, d depth, m max depth

Tree vs Graph Search low to **moderate** state space, **optimal** solution or nothing

```
create frontier

insert init state to frontier
while frontier is not empty:
state = frontier.pop()
if state is goal: return solution
for action in state.actions():
next = transition(state, action)
frontier.add(next.state)
return failure
```

```
create frontier, create visited

insert init state to frontier
while frontier is not empty:
state = frontier.pop()
if state is goal: return solution
if state is visited: continue
visited.add(state)
for action in state.actions():
  next = transition(state, action)
  frontier.add(next.state)
return failure
```

Uninformed Search Algorithms

- **BFS** Tree-Search with queue
  - *Time/Space:* $O(b^d)$, expand to last child in branching (space)
  - *Complete:* Yes if finite graph
  - *Optimal:* Yes if same cost everywhere
- **UCS** Tree-Search with priority queue, min **PATH COST**
  - *Time/Space:* $O(b^{\frac{C^*}{\epsilon}})$
  - *Complete:* Yes if $\epsilon > 0$ and C^* is finite
  - *Optimal:* Yes if $\epsilon > 0$
  - $\epsilon = 0$ may cause zero cost cycle, where C^* is cost of optimal solution and ϵ is min edge cost
  - Estimate Depth $= \frac{\text{optimal path cost}}{\min(\text{edge costs})}$
  - Akin to BFS with same path cost for all nodes
- **DFS** Tree-Search with stack
  - *Time:* $O(b^m)$, *Space:* $O(bm)$
  - *Complete:* No when depth is ∞ or can go back and forth (loop)
  - *Optimal:* No
  - where m is the max. depth
- **Depth-limited Search**
  - Limit the search to depth l, backtrack when limit is hit
  - **Adv:** avoid getting into cycle, saves memory by going into a max depth —— **Dis-adv:** Sub-optimal if used with DFS
  - Number of nodes generated $N(DLS) = b^0 + b^1 + \cdots + b^d$
  - *Time:* $O(b^l)$
  - *Space:* $O(bl)$ if used with DFS
  - *Complete:* No
  - *Optimal:* No if used with DFS
- **Iterative Deepening Search**
  - Search with depth limit $0 \to 1 \to 2 \ldots \to \infty$
  - **Adv:** avoid sub-optimality of normal DLS
  - *Time (no. of nodes generated):* $b^0 + (b^0 + b^1) + \cdots + b^d \approx O(b^d)$
  - *Space:* $O(b^d)$
  - *Complete:* Yes
  - *Optimal:* Yes if step cost is the same everywhere

Informed Search Algorithms

- **Best-first Search:** p-queue with eval $f(n)$, estimates how good a state is
- ***Greedy* Best-first Search:** priority queue with eval $h(n)$, heuristic: estimates costs from n to goal
  - *Time:* $O(b^m)$, good heuristic gives improvement — *Space:* $O(b^m)$
  - *Complete:* No —— *Optimal:* No
  - Wrong heuristic $f(n)$ lead to loops / sub-optimal solutions
- **A* Search:** priority queue with eval $f(n)$, comprised of the following
  - $g(n)$: Cost to reach $n + h(n)$: heuristic: estimates costs from n to goal
  - *Time:* $O(b^m)$, good heuristic gives improvement — *Space:* $O(b^m)$
  - *Complete:* Yes —— *Optimal:* *Depends* on heuristic

Heuristics

1. **Admissible:** If every node n, $h(n) \leq h^*(n)$ where $h^*(n)$ is **true cost** to reach goal state
   - Never **over-estimates** cost to reach goal, conservative
   - If $h(n)$ is admissible, A^* using **tree search** is optimal
2. **Consistent:** If every node n, every successor of n' of n generated by any action a, $h(n) \leq c(n, a, n') + h(n')$ and $h(G) = 0$
   - If h is consistent, $A \to C$ (direct distance) $\leq A \to B \to C$ (distance to goal when going to a different point first)
   - If $h(n)$ is consistent, A^* using **graph search** is optimal
3. **Dominance:** If $h_2(n) \geq h_1(n)$ for all n, then h_2 dominates h_1
   - h_2 better for search **if admissible**
   - e.g. for misplaced tiles if h_1 is number of misplaced tiles, h_2 is Manhattan distance, then h_2 dominates h_1
   - If each tile is at most one distance away from goal, then $h_2 = h_1$, otherwise $h_2 > h_1$
4. A *consistent* heuristic is *admissible* but not the other way round.

Relaxed Problem: A problem with **fewer restrictions** on actions. Cost of *optimal solution* to relaxed problem is an **admissible heuristic** for the original problem.

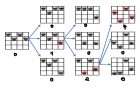
3. Local Search → state is solution

- **very large** state space, **good enough** solution preferred over no solution

1. States, Initial State
2. Goal Test (optional)
3. Successor function → possible states from state
4. Evaluation function
   - outputs goodness of a state, **objective functions** (N-Queens)

Hill Climbing Algorithm

```
current = initial state
while True:
  neighbour = a highest-val successor of current
  if value(neighbour) <= value(current):
    return current
  current = neighbour
```



- Could lead to reaching local maximum
- **Escape techniques:** Tabu Search, Random Restarts, Random Walk

Simulated Annealing —— allow bad moves from time to time

```
current = initial state
T = large positive value
while T > 0:
  next = randomly selected successor of current
  if value(neighbor) > value(current):
    current = next
  else with probability P(current,next, T):
    current = next
  decrease T
return current
```

- $P(current, next, T) = e^{\frac{value(next) - value(current)}{T}}$
- **Theorem:** If T decreases slowly enough, simulated annealing will find a global optimum with high probability

Adversarial Search

For games of the following characteristics, we *do not do normal searches* → lack of information of what **other** agent does

- Fully observable
- Deterministic
- Discrete
- 2 player zero-sum
- Turn-taking
- Terminal states exist

Components of Adversarial Search

- States, Initial States, Actions, Transitions
- **Terminal States**: when game ends — **Utility Function**: output value of state

Minimax

- *Time:* $O(b^m)$ —— *Space:* $O(bm)$, with depth first exploration
- *Complete:* Yes if tree is finite —— *Optimal:* Yes against **optimal opponent**

Minimax with Cutoff — Replace `if is_terminal(state)` with the following:

```
if is_cutoff(state): return eval(state)
```

e.g. for Nim Minimax with Cutoff

- **Cutoff**: Terminal OR Depth ≥ 2
- **Evaluation function**:
  - Terminal: utility(state) e.g. Number of sticks > 1: $+1$ Else: -1
  - Create based on Expert Knowledge/Learn from data
  - e.g. **heuristic functions** estimates how 'good' a state is

Alpha-beta Pruning

Evaluating a node is sometimes not useful, as it does not change decision, α **highest value** for MAX, β **smallest value** for MAX

- Pruning does not affect final result
- Good move ordering improves effectiveness of pruning
  - Perfect Ordering: $O(b^{\frac{m}{2}})$

```
def alpha_beta_search(state):
  v = max_value(state, -inf, inf)
  return action in successors(state) with value v
def max_value(state, alpha, beta):
  if is_terminal(state): return utility(state)
  v = -float('inf')
  for action, next_state in successors(state):
    v = max(v, min_value(next_state))
    alpha = max(alpha, v)
    if v >= beta: return v
  return v
def min_value(state, alpha, beta):
  if is_terminal(state): return utility(state)
  v = float('inf')
  for action, next_state in successors(state):
    v = min(v, max_value(next_state))
    beta = min(beta, v)
    if v <= alpha: return v
  return v
```

Local Beam Search

Pick k random initial states and generate their successors. If goal is found, terminate, else, pick the k best successors and continue. Stochastic Beam search by choosing sucessors with probability proportional to value

4. Intro to Machine Learning and Decision Trees

Supervised Learning

Learns from being given the **right answers** $X \to Y$

- **Regression:** predict continuous output (e.g. temp $= 0.567$)
- **Classification:** predict discrete output (e.g. animal $=$ cat vs dog)

Formalism

- we assume that y generated by a **true mapping function:** $f : x \to y$
- want to find **hypothesis** $h : x \to \hat{y}$ (from **hypothesis class** H) s.t. $h \approx f$ given a **training set** $\{((x_1, f(x_1)), \ldots (x_N, f(x_N))\}$
  - Find this hypothesis using a **learning algorithm**

Performance Measure

- **Regression: Error**
  - If output of hypothesis is **continuous**, then we can measure its **error** For an input x with true output y, we can compute:
    - Absolute Err $= |\hat{y} - y|$ — Squared Err $= (\hat{y} - y)^2$, where $\hat{y} = h(x)$
- **Mean Squared Error**
  - For set of N examples $\{(x_1, y_1), \ldots, (x_N, y_N)\}$ we can compute the average (mean) squared error:
    - $MSE = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2$, where $\hat{y}_i = h(x_i)$ and $y_i = f(x_i)$
- **Mean Absolute Error**
  - For set of N examples $\{(x_1, y_1), \ldots, (x_N, y_N)\}$ we can compute the average (mean) absolute error:
    - $MAE = \frac{1}{N}\sum_{i=1}^{N}|\hat{y}_i - y_i|$, where $\hat{y}_i = h(x_i)$ and $y_i = f(x_i)$
- **Classification: Correctness & Accuracy**
  - Classification is correct when prediction $\hat{y} = y$ (true label)
  - For set N examples $\{(x_1, y_1), \ldots, (x_N, y_N)\}$ we can compute the average **correctness** (accuracy):
    - $Accuracy = \frac{1}{N}\sum_{i=1}^{N}\mathbb{1}_{\hat{y}_i = y_i}$, where $\hat{y}_i = h(x_i)$ and $y_i = f(x_i)$

Classification: Confusion Matrix

Accuracy $= \frac{TP+TN}{TP+FN+FP+TN}$

FP: Type I error

FN: Type II error

Can we combine the best of both metrics?

F1 Score $F1 = \frac{2}{\frac{1}{P} + \frac{1}{R}}$

Predicted Label \ Actual Label	Cancer	Benign
Cancer	2 True Positive	1 False Positive
Benign	3 False Negative	4 True Negative

Precision $P = TP / (TP+FP)$

How many **selected** items are relevant?

How **precise** were the positive predicted instances?

Maximize this if false positive (FP) is very costly.
E.g., email spam, satellite launch date prediction

Recall $R = TP / (TP+FN)$

How many relevant items are **selected**?

How many positive instances can be **recalled** (predicted)?

Maximize this if false negative (FN) is very dangerous.
E.g., cancer prediction but not music recommendation

Decision Trees

Expressiveness

- Decision tress can express **any function** of the input attributes
- e.g. Boolean functions, each row → path from root to leaf
- Trivially there is a *consistent* decision tree for any training set, but it may not **generalize to new examples**

Size of hypothesis class

Distinct decision tress with n boolean attributes $=$ number of Boolean functions $=$ number of distinct truth tables with 2^n rows $= 2^{2^n}$

Decision Tree Learning — We need to define `choose_attribute`

```
def DTL(examples, attributes, default):
  if examples is empty: return default
  if examples have same classification:
    return classification
  if attributes is empty: return mode(examples)
  best = choose_attribute(attributes, example)
  tree = new decision tree with root best
  for value v_i in best:
    examples_i = {rows in examples with best = v_i}
    subtree = DTL(examples_i, attributes - best, mode(↩
        examples))
    add a branch to tree with label v_i and subtree
```

Choosing attributes

Ideally, want to select attribute that splits examples into all positive/negative

Entropy

- Measure of pure randomness: $I(P(v_1), \ldots, P(v_n)) = -\sum_{i=1} nP(v_i)\log_2 P(v_i)$
- For data set containing p positive and n negative examples: $I(\frac{p}{p+n}, \frac{n}{p+n}) = -\frac{p}{p+n}\log_2\frac{p}{p+n} - \frac{n}{p+n}\log_2\frac{n}{p+n}$

Information Gain

Entropy of this node - Total entropy of children nodes

- chosen attribute A divides training set E into subsets $E_1, \ldots, E_V$ according to values for A, where A has v distinct values $remainder(A) = \sum_{i=1}^{v}\frac{p_i+n_i}{p+n}I(\frac{p_i}{p_i+n_i}, \frac{n_i}{p_i+n_i})$
- Information Gain (IG) or reduction in entropy $IG(A) = (\frac{p_i}{p_i+n_i}, \frac{n_i}{p_i+n_i}) - remainder(A)$

Occam's Razor

- Short/simple hypothesis, favouring:
  - **Short/simple** hypothesis fits data is **unlikely coincidental**
  - **Long/complex** hypothesis that fits data **may be coincidence**
- Against:
  - Many ways to define small sets of hypothesis → trees with prime number of nodes that uses attribute beginning with "Z"
  - Different hypotheses representations may be used instead

Pruning in DT

Prevent nodes from being split even when if fails to cleanly separate examples. e..g if does not meet min sample (T: 1, F:2, Total: 5), combine both nodes into 1 take majoirty (F).

5. Linear Regression

For a set of m examples $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$

MSE: $J_{MSE}(w) = \frac{1}{2m}\sum_{i=1}^{m}(h_w(x^{(i)}) - y^{(i)})^2$

- $J_{MSE}(w)$ — Loss function
- $\frac{1}{2m}$ — mathematical convenience
- $(x^{(i)})$ — a.k.a $\hat{y}^{(i)}$
- want to minimize the loss/error

if we fix $w_0 = 0$ for easier visualization:

- *Loss Function:* $J_{MSE}(w) = \frac{1}{2m}\sum_{i=1}^{m}(w_1(x^{(i)}) - y^{(i)})^2$
- $-\frac{\delta J_{MSE}(w)}{\delta w_1} = -\frac{1}{m}\sum_{i=1}^{m}(w_1(x^{(i)}) - y^{(i)})(x^{(i)})$
- *partial derivative to find the minimum point (smallest slope) in loss function*

Linear Regression & ML

- **Vector:** $w_0, w_1, \ldots, w_n$, column $w = \begin{bmatrix} w_0 \\ \ldots \\ w_n \end{bmatrix}$ row $w^T = \begin{bmatrix} w_0 \ldots w_n \end{bmatrix}$
- **Dot product:** $u^T v = w^T = \begin{bmatrix} w_0 w_1 \ldots w_n \end{bmatrix}\begin{bmatrix} w_0 \\ \ldots \\ w_n \end{bmatrix} = \sum_{j=0}^{n} u_j v_k$
- **Partial derivative:** $\frac{\delta J(w)}{\delta w_1}$ e.g. $J(w) = -w_0^2 - w_1^2 = \frac{\delta J(w)}{\delta w_1} = -2w_1$
- **Gradient:** $\begin{bmatrix} \frac{\delta J(w)}{\delta w_0} \\ \frac{\delta J(w)}{\delta w_1} \\ \ldots \end{bmatrix}$

Gradient Descent

1. Start at some $w \to$ pick nearby w that $\downarrow J(w)$
2. $w_j \leftarrow w_j - \gamma\frac{\delta J(w_0, w_1, \ldots)}{\delta w_j}$, repeat until minimum reached
3. **Learning Rate:** γ
   - small $\gamma \to$ model takes long time to run, may not find minimum
   - large $\gamma \to$ model might overshot minimum
   - Each variable in the linear equation should have its own γ
   - MSE loss function is **convex** for linear regression (1 global minimum)

Variants of Gradient Descent

- **(Batch) Gradient Descent:** consider all training examples
- **Mini-batch Gradient Descent:**
  - subset of training examples at a time
  - cheaper, faster per iteration. — random, may escape local minima
- **Stochastic Gradient Descent (SGD):**
  - select one random data point at a time
  - cheapest, fastest per iteration. — more random, may escape local minima
- To speed up we can:
  - Perform feature scaling/mean normalization, apply PCA but keep variance
  - Use **larger/adaptive** learning rate to speed convergence/use variants above

Linear Regression with Many Attributes

- **Hypothesis** $h_w(x) = w_0x_0 + w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4$
- **Hypothesis — for n features**
  - $h_w(x) = \sum_{j=0}^{n} w_j x_j = \begin{bmatrix} w_0 \\ w_1 \\ \ldots \\ w_n \end{bmatrix}^T \begin{bmatrix} x_0 \\ x_1 \\ \ldots \\ x_n \end{bmatrix} = w^T x$
- **Weight Update — for n features**
  - $w_j \leftarrow w_j - \gamma\frac{\delta J(w_0, w_1, \ldots)}{\delta w_j}$
  - $w_j \leftarrow w_j - \gamma\frac{1}{m}\sum_{i=1}^{m}(h_w(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$

Features with different scales

- Use standardization e.g. mean normalization, min-max scaling, robust scaling etc.
- **Mean normalization:** $x_j = \frac{x_j - \mu_j}{\sigma_j}$, where $\sigma_j =$ std. dev.

Non-Linear Relationships

- We need to scale the polynomial regressions e.g.:
- $h_w = w_0 + w_1x + w_2x^2$ (polynomial regression)
- transform $f_1 = x, f_2 = x^2$, we can get $h_w = w_0 + w_1f_1 + w_2f_2$

6. Logistic Regression

Decision Trees works with discrete/categorical inputs, not with many options.

Logistic Function (Sigmoid)

$\sigma(z) = \frac{1}{1+e^{-z}}$ —— $h_w(x) = \sigma(w_0 + w_1x)$

- Output of sigmoid is in $[0, 1]$, treat each output as **probability**
- $h_w(x) = P(x = \text{mal}) \to P(x = \text{mal}) > \alpha$ e.g. 0.5 then malignant
- Consider it a decision boundary, for continous variables

Logistic Regression N-D case

$h_w(x) = \sigma(w_0 + w_1x_1 + w_2x_2 + \cdots + w_nx_n)$ — For a set of m examples $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$, where fn is non-linear → non-convex:

$$J_{MSE}(w) = \frac{1}{2m}\Sigma_{i=1}^{m}(h_w(x^{(i)}) - y^{(i)})^2 = \frac{1}{2m}\Sigma_{i=1}^{m}(\frac{1}{1+e^{-(w_0+w_1x_1+w_2x_2+\cdots+w_nx_n)}} - y^{(i)})^2$$

Measuring Closeness Between Prob Distribution

Cross-entropy for C classes: $CE(y, \hat{y}) = \sum_{i=1}^{C} -y_i\log(\hat{y}_i)$

Binary cross-entropy: $BCE(y, \hat{y}) = -y\log(\hat{y}) - (1-y)\log(1-\hat{y})$

Logistic Regression with Cross-Entropy Loss

For a set of m examples $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$, we can compute the binary cross entropy loss → convex for logistic regression:

$$J_{BCE}(w) = \frac{1}{m}\Sigma_{i=1}^{m}BCE(y^{(i)}, h_w(x^{(i)}))$$

$$h_w(x) = \sigma(w_0 + w_1x_1 + w_2x_2 + \cdots + w_nx_n) \quad \text{(prob. output)}$$

Logistic Regression with Gradient Descent

Weight Update: $w_j \leftarrow w_j - \gamma\frac{\partial J_{BCE}(w_0, w_1, \ldots)}{\partial w_j}$, same as linear regression:

$$\frac{\partial J_{BCE}(w)}{\partial w_j} = \frac{\partial}{\partial w_j}\frac{1}{m}\sum_{i=1}^{m}BCE(y^{(i)}, h_w(x^{(i)}))$$

$$\frac{\partial J_{BCE}(w)}{\partial w_0} = \frac{1}{m}\sum_{i=1}^{m}(h_w(x^{(i)}) - y^{(i)})$$

$$\vdots$$

$$\frac{\partial J_{BCE}(w)}{\partial w_j} = \frac{1}{m}\sum_{i=1}^{m}(h_w(x^{(i)}) - y^{(i)})x_j^{(i)}$$

Multi-class Classification

- **One vs All**: 1 classifier/class, fit against **all other** classes, pick **highest prob.**
- **One vs One**: 1 classifier/class pair, pick **most wins**

Receiver Operator Characteristic (ROC) Curve

- Model is more accurate than random chance if ROC curve is above the diagonal random line
- Graphical plot that illustrates the performance of a binary classifier.

Area Under Curve (AUC) of ROC

- AUC is concise **metric** instead of a full figure
- Concise metrics enable clearer comparisons
- AUC > 0.5 means the model is better than chance
- AUC ≈ 1 means model is very accurate
- Usually plot as TPR (sensitivity) against FPR (1 - specificity)

Model Evaluation & Selection

Given dataset D, error function $error$, expected error of a model/hypothesis h: $J_D(h) = \frac{1}{N}\sum_{i=1}^{N} error(h(x^{(i)}), y^{(i)})$, where$(x^{(i)}), y^{(i)}) \in D$

1. Split data into $D_{train}, D_{val}, D_{test}$
2. Train with D_{train}, minimizing J to obtain model
3. Choose model with the minimum $J_{D_{val}}$
4. Once picked ideal model, test it against test set D_{test} by computing $J_{D_{val}}$

Diagnosing Bias and Variance

- $J_{D_{val}}(w) \approx J_{D_{train}}$(high), meaning model is underfit, has high bias
- $J_{D_{val}}(w) >> J_{D_{train}}$(low), meaning model is overfit, has high variance

Hyperparameter Tuning

- Pick hyperparameters: e.g. deg. of polynomials, learning rate
- Train model with hyperparameters, → Evaluate model

Tuning Methods

- **Grid search (exhaustive search):** Try all possible hyperparameters
- **Random search:** Randomly select hyperparameters
- **Successive halving:** Use all possible hyperparameters but with ↓ resources. successively increase the resources with smaller set of hyperparameters
- **Bayesian optimization:** Use Bayesian methods to estimate the optimization space of the hyperparameters
- **Evolutionary algorithms:** Use evolutionary algorithms (e.g., genetic algo) to select a population of hyperparameters

7. Support Vector Machines

Regularization

Ways to address over fitting: (1) reduce number of features, (2) reduce magnitude of weights (regularization)

- **Ridge Regression:** $J(w) = \frac{1}{2m}[\Sigma_{i-1}^{m}(h_w(x^{(i)}) - y^{(i)})^2 + \lambda \Sigma_{j=1}^{n} w_j{}^2]$
- update function (**gradient descent**):
  - $w_n = (1 - \frac{\alpha\lambda}{m} w_n) - \alpha \frac{1}{m} \Sigma_{i=1}^{m} (h_w(x^{(i)}) - y^{(i)}) \cdot x_n^{(i)}$

$$w = (X^T X + \lambda \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ & & \ddots & \\ 0 & 0 & & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix})^{-1} X^T Y$$

- This works even if $X^T X$ is non-invertible is $\lambda > 0$
  1. Too large λ can cause your model to underfit.
  2. Regularization assure that your model performs better on unseen data, sacrificing performance on training data. (higher bias but low variance)
  3. Ridge penalizes larger parameters, attempting to pull all parameters towards small values.

Support Vector Machines

- In a hyperplane in $N-$dimensional space that distinctly classifies data points
- Want to maximize margin distance to provide reinforcement to future data points to be classified with more confidence. To *maximize margin*:
- Penalize *minority class more* otherwise model naturally bias to majority
  - **Decision Rule:** $w \cdot x + b \geq 0$ then + else –
  - **Equation** for margin: margin $= \frac{2}{||w||}$
  - **Constrained optimization problem:**

$$\max_w \frac{2}{||w||} \rightarrow s.t.\ \ y^{(i)}(w \cdot x^{(i)} + b) + 1 \geq 0$$

    - **Primal:** $\min_w \frac{1}{2}\|w\|^2, \quad \text{s.t.} \quad y^{(i)}\left(w \cdot x^{(i)} + b\right) - 1 \geq 0$
    - **Dual:** $\max_{\alpha \geq 0} \sum_i \alpha^{(i)} - \frac{1}{2} \sum_{i,j} \alpha^{(i)} \alpha^{(j)} y^{(i)} y^{(j)} x^{(i)} \cdot x^{(j)}$

$J(w) = C \sum_{i=1}^{m} \left(y^{(i)} c_1 + (1 - y^{(i)}) c_0\right) + \frac{1}{2} \sum_{i=1}^{n} w_i^2$ where $c_1 = -\log\left(\frac{1}{1+e^{-w^T x}}\right)$, $c_0 = -\log\left(1 - \frac{1}{1+e^{-w^T x}}\right)$ This is a constrained optimization problem to minimize the number of misclassifications and maximize correct classifications ("soft" margin)

Properties:

1. SVM is robust to outliers (increase regularization)
2. support Vectors influence the position and orientation of the hyperplane
3. binary classifier, n-class problems, n models required to correctly classify
4. ↑ C — ↓ effect of regularization, allowing the data to fit more (lower bias)

Kernels

Bijective function mapping data points to a *higher-dimensional* plane so that data is linearly separable. Allows us to operate in the original feature space without computing coordinates of data in higher dimensional space (costly).

- Polynomial degree d (n^d terms) $K(u, v) = \phi(u) \cdot \phi(v) = (u \cdot v)^d$
- **Guassian kernel:** $K(x, l^{(i)}) = e^{-\frac{||x - l^{(i)}||^2}{2\sigma^2}}$
- Kernel trick ensures that no need to compute transformed features **explicitly**

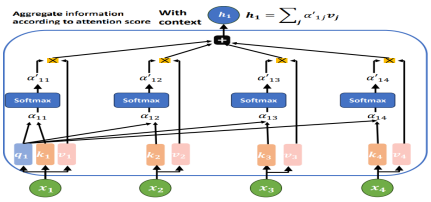
  - **Input gate:** $a = g^{[1]}(z_i) \times g^{[2]}(z)$, produces a which goes into C
  - **Forget gate:** $b = c \times g^{[1]}(z_f)$, producinges b which goes into C
  - **Output gate:** $h = g^{[1]}(z_o) \times g^{[2]}(c)$, produces h output
2. **Memory** C: can be referred to as the state: $c \leftarrow a + b$

Self-Attention and Transformer

Problems with RNN

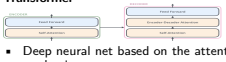
- ✓ Able to capture context info from previous time steps
- ✗ Output at time step t must wait for all steps $< t$ to complete → this is not parallelism-friendly

Self-Attention



- **Query** represents info we want to focus on.
- **Key** represents info associated with each input that can be compared to query
- **Value** contains actual info retrieved based on the attention scores
- $\alpha'_{1j} = \frac{\exp(\alpha_{1j})}{\Sigma_j \exp(\alpha_{1j})}$, $\alpha'_{11} + \alpha'_{12} + \alpha'_{13} + \alpha'_{14} = 1$
- **query** $q_1 = W^q x_1$ — **key** $k_i = W^i x_i$ — **value** $v_i = W^v x_i$
- All q, k and v can be computed together with matrix multiplication

Transformer

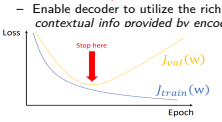


- Deep neural net based on the attention mechanism.
- Input a sequence, output a sequence
- **Encoder-Decoder Attention:**
  - **Query:** generated based on previous decoder block's output
  - **Key, Value:** generated based on encoder's output
  - Enable decoder to utilize the rich *contextual info provided by encoder*

Issues with Deep Learning

Overfitting → Regularization

- **Dropout:** during training, *randomly* set some activations to 0
- **Early stopping:**



Gradient Vanishing/Exploding

- **Vanishing gradient:** *small* gradients multiplied repeatedly → almost 0
- **Exploding gradient:** *large* gradients multiplied repeatedly → overflows
- **Mitigation:** Using non-saturaing activation functions e.g. ReLU or Gradient clipping (clip with range $[\min, \max]$)

10. Unsupervised Learning

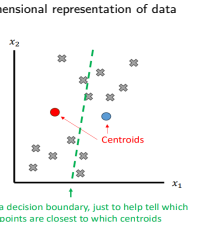
No labels, use learning by experiencing raw data, obtaining as obtaining labels can be expensive. Given a set of m data points $x^{(1)}, \ldots, x^{(m)}$, learn patterns

- **Clustering:** identify clusters in the data
- **Dimensionality reduction:** find a lower-dimensional representation of data

K-Means Clustering

Let points $x^{(i)}$ for $i = 1, \ldots, m_1$ be assigned to cluster 1. For these points, cluster centroid is defined as $u_1 = \frac{1}{m} \sum_{i=1}^{m_1} x^{(i)}$

- Randomly initialize K centroids: $\mu_1, \ldots, \mu_K$
- Repeat until convergence
  - For $i = 1, \ldots, m$:
    - $c^{(i)} \leftarrow$ index of cluster centroid $(\mu_1, \ldots, \mu_K)$ closest to $x^{(i)}$
  - For $k = 1, \ldots, K$:
    - $\mu_k \leftarrow$ centroid of data points $x^{(i)}$ assigned to cluster k



1. Given a data point x, compute new features based on the proximity to landmarks. Features $f_1, f_2, \ldots, f_n$ will be ≈ 1 if x is close, else ≈ 0. Replace all x_i's with f_i's.
2. Apply feature scaling.
3. Some kernels may not converge. Need to satisfy Mercer's theorem.
4. ($n >> m$) SVM with no kernel, ($n < m$) SVM with guassian kernel, ($m >> n$) SVM with no kernel

Bias and Variance Tradeoff

A model with a **high bias** makes more assumptions, and unable to capture the important features of our dataset. A high bias model cannot perform well on new data. To reduce, (1) increase the input features (2) decrease regularization term (3) use more complex models. A model that shows **high variance** learns a lot and perform well with the *training dataset*, but does not generalize well with the unseen dataset. To reduce, (1) decrease the input features (2) do not use more complex model (3) increase training data (4) increase regularization term.

8. Intro to Neural Networks

Perceptron, Linear classifier: algorithm predicts labels via $\hat{y} = \sigma(w^T x)$

Perceptron Learning Algorithm

1. Initialize weights, wi (can be zero or random small values).
2. Loop (*until convergence or max steps reached*)
   - For each instance $(x^{(i)}, y^{(i)})$, classify $\hat{y}^{(i)} = h_w x^{(i)}$
   - Select misclassified instance $(x^{(i)}, y^{(i)})$
   - Update weights $w \leftarrow w + \gamma(y^{(i)} - \hat{y}^{(i)}) x^{(j)}$

Perceptron properties

- Not robust: can select any model to linear, **not deterministic**
- Cannot converge on non-linearly separable data

Single-layer Neural Networks

- Single forward pass → back propagate to update weights for model to learn
- In single-layer perceptron, weight updates as follows:
- $w_i \leftarrow w_i - \gamma \frac{dL}{dw_i} = w_i + \gamma(y - \hat{y}) g'(f) x_i$

Multi-layer Perceptron

A multi layer perceptron is formed by combining many single layer perceptrons. This allows the model to learn more complex function

9. Neural Networks +

Forward Propagation

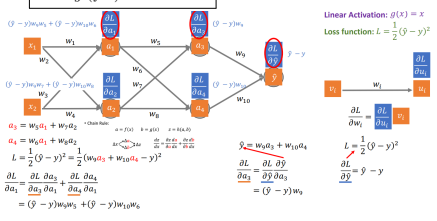
Propagate the sum of all node * weight in the target node, applying this notation recusively from the input layer: $a^{[l]} = g^{[l]}(W^{[l]} a^{[l-1]})$

e.g. $a_1 = w_{11}x_1 + w_{21}x_2 + w_{31}x_3$, $a_2 = w_{12}x_1 + w_{22}x_2 + w_{32}x_3 \cdots$ **Chain Rule** $a = f(x)z = g(a) \Rightarrow \triangle x \rightarrow \triangle a \rightarrow \triangle z \Rightarrow \frac{dz}{dx} = \frac{dz}{da}\frac{da}{dx}$

Backward Propagation

First compute the values of the nodes in orange ($v_i \xrightarrow{w_i} u_i$)

- Convert non-linear activation function to $g'(x)$
- All values in orange computed in one forward pass v_i
- Next we compute the values in blue with one backward pass $\frac{\delta L}{\delta u_i}$
- When implementing custom function/layers, ensure they are **differentiable**
  - $\delta^{[l]} = g'(f^{[l]}) \cdot W^{[l+1]} \delta^{[l+1]}$



K-Means: Measuring

- The algorithm could possibly get stuck on a local optima, where outcome is a stable configuration as centroids do not move
- **Measuring the Goodness:** we use *Loss/Distortion* which is the average distance of each sample to its centroid
- $J(c^{(1)}), c^{(2)}, \ldots, c^{(m)}, \mu_1, \mu_2, \ldots, \mu_K) = \frac{1}{m} \sum_{i=1}^{m} ||x^{(i)} - \mu_{c^{(i)}}||^2$

K-Means: Picking the number of clusters

- **Elbow Method:** seeing when $J()$ against K samples reaches an eblow, but heuristic method, data may not have an elbow/has multiple elbows
- **Application dependent:** if we have 5 sizes, then we will pick $k = 5$

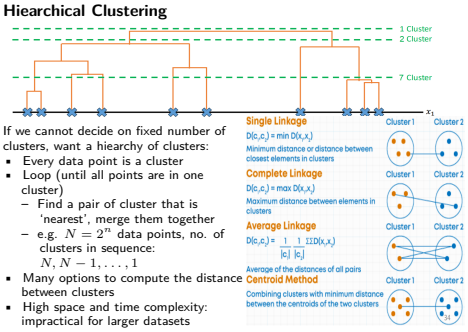
K-Means: Variants

- Pick K initial centroids randomly from points in the data
- K-Medoids: pick data points that are closests to the centroids, use them as centroids → 'snap' centroids to the nearest data points

K-Means: Bad Performance

- Large dataset, many features → curse of dimensionality, run PCA to reduce number of dimensions OR Poor init of clusters/bad k choice/noisy data
- Features at different scales, affect clustering (normalize data to solve)

Hiearchical Clustering



If we cannot decide on fixed number of clusters, want a hierarchy of clusters:

- Every data point is a cluster
- Loop (until all points are in one cluster)
  - Find a pair of cluster that is 'nearest', merge them together
  - e.g. $N = 2^n$ data points, no. of clusters in sequence: $N, N-1, \ldots, 1$
- Many options to compute the distance between clusters
- High space and time complexity: impractical for larger datasets

Hiearchical Cluster: Applications

- **Customer Segmentation:** Utilizing hierarchical clustering enables the segmentation of customers according to their purchasing behavior, preferences, or demographic data.
- **Gene Expression Analysis:** The application of hierarchical clustering can aid in the analysis of gene expression data, revealing patterns or clusters of genes exhibiting similar expression profiles.
- **Recommender Systems:** Hierarchical clustering serves as a valuable tool in constructing recommender systems, grouping similar users or items based on their preferences or behavior.
- **Social Network Analysis:** The implementation of hierarchical clustering is beneficial for analyzing social networks, uncovering communities or groups of individuals sharing similar social connections or interests.

Dimensionality Reduction

Many machine learning systems have data with **high-dimensional features**

- Curse of dimensionality: number of samples to learn a hypothesis class increases exponentially with the number of features
- Want to reduce and remove feature that captures the *least variations in data* ⇒ identifying the 'most important' concepts

Singular Value Decomposition (SVD)

- $X = U\Sigma V^T$
- Intuition: Action of any matrix on a vector is (rotate) × (stretch) × (rotate)
- where X is an $n \times m$ matrix and m is the # of samples
- **Fact:** Take without loss of generality $n > m$. For any $n \times m$ rectangular real-valued matrix X, there exists a factorization $X = U\Sigma V^T$ called SVD
  - U is $n \times m$ and has m orthonormal columns (left singular vectors)
  - Σ is $n \times m$ and has m $\sigma_j \geq 0$ (singular values)
  - V is $m \times m$ and has m orthonormal columns and rows (right singular vectors)
- Always possible to order the $\sigma_j \geq 0$ from largest to smallest, which gives ordering of the corresponding singular vectors. **'Ordered by importance'**

Convolutional Neural Network

- Leverage on **spatial context** to extract local features
- Enjoy **translation invariance**, features can be recognized elsewhere
- **Fewer parameters:** kernels share weights → faster training
- number of params in convolutional layer (with **bias**): $\#params = ((kernel\_size \times |input\_channels|) + 1) \times |output\_channels|$

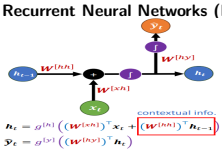
Layers in CNN — (Size, padding, kernel size, stride)

Recusively do *input* → *convolution* + *relu* → *pooling* → *convolution* + *relu* ⋯ for feature learning, *flattening* → *fully connected*, → *softmax* for predicting

- **Convolutional**: Element-wise multiply-sum operation between an image section (dependent on the stride and padding) and the kernel.
  - Dimensions of the output for one kernel is given $\left\lfloor \frac{n+2p-K}{s} \right\rfloor + 1$
  - Add *padding* in order to keep size of output feature map
- **Pooling Layer**: Extract the most relevant features, and reduce dimensionality, parameters, and noise from previous convolutional layer
  - Max-pool, Average-pool, sum-pool
- **Fully Connected**: There can be multiple fully connected layers. An activation function is used in each fully connected layer
- **Softmax Layer**: Use for classification tasks, by normalizing results to follow a probability distribution, allows us to avoid binary classification, accomodate as many classes as needed

10. Neural Networks on Sequential Data

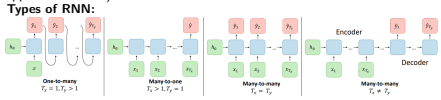
Recurrent Neural Networks (RNN)



- Takes information from prior inputs to influence current input & output
- Same weights applied at each step, handles sentences of varying length

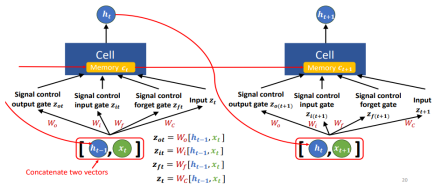
Other RNN

- **Deep RNN:** Chain h_i up to N layers, before output $\hat{y}_i$
- **Bidirectional RNN:** Capture information for content before x_t and process as $\hat{y}_{t-1}$, then contact the result of capturing information for content after x_t (the opposite direction)
- **Types of RNN:**



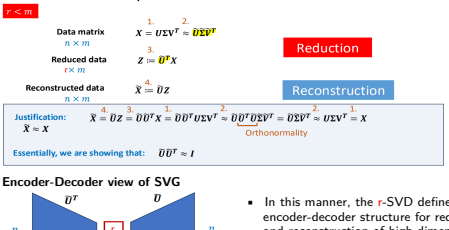
1. *one-to-one:* Each I/O is independent. Not RNN
2. *one-to-many:* Image captioning, text generation (decoders)
3. *many-to-one:* Text classification, sentiment recognition (encoders)
4. *many-to-many:* Language translation, music generation
   - Process all inputs first, and then predict later OR Process 1 at a time
- **Applications of RNN:** sentiment analysis, speech/video activity recognition

Long Short-Term Memory (LSTM)

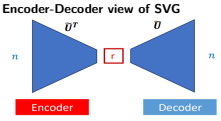


- Recurrent neural network with gating:
1. **4 inputs:** 3 controls (input z, signal control input gate z_i, signal control forget gate z_f, signal control output gate z_o), **1 output:** h
   - For the gates the signal control $z_i/z_o/z_f$ goes through a sigmoid function: ranges from 0 (*closing the gate*) to 1 (*opening the gate*)

Dimensionality reduction via SVD



Encoder-Decoder view of SVG



- In this manner, the r-SVD defines an encoder-decoder structure for reduction and reconstruction of high-dimensional data
- Equivalent ideas appear also in auto-encoder (AE), variational auto-encoder (VAE).

Principal Component Analysis (PCA)

- $Var(x) = E[(x - \hat{x})^2]$, $Cov(x, y) = E[(x - \hat{x})(y - \hat{y})]$
- **Statistical Application** of SVD, capture components that maximize the *statistical variations* of data
- **Sample covariance matrix:** Given data matrix $X = (x^{(1)}, \ldots, x^{(m)})$
  1. Compute mean over samples: $\bar{x} = \frac{1}{m} \sum_{i=1}^{m} x^{(i)}$
  2. Compute mean-centered data: $\hat{x}^{(i)} = x^{(i)} - \bar{x}$
  3. Define data matrix $\hat{X} = \left(\hat{x}^{(1)}, \ldots, \hat{x}^{(m)}\right)$
  4. Create the covariance matrix of the data: $\text{Cov}(X) = \frac{1}{m} \hat{X} \hat{X}^\top$
- **Steps:**
  1. Create the covariance matrix of the data: $Cov(X) = \frac{1}{m} \hat{X} \hat{X}^T$
  2. Compute SVD on $Cov(X)$ to obtain the U matrix (new basis)
  3. Reduce to r components to obtain $\tilde{U}$

PCA Picking Number of Components — Retain $\geq 99\%$ of variance in data

Note that X's singular values^2 are related to the singular values (variances) in Cov:

$$XX^T = U\Sigma V^T (U\Sigma V^T)^T = U\Sigma V^T V \Sigma U^T = U\Sigma\Sigma U^T = U \begin{bmatrix} \sigma_1^2 & & \\ & \sigma_2^2 & \\ & & \ddots \\ & & & \sigma_m^2 \end{bmatrix} U^T$$

Solution: choose minimum r s.t. $\frac{\Sigma_{i=1}^{r} \sigma_i^2}{\Sigma_{i=1}^{m} \sigma_i^2} \geq 0.99$. It takes a few steps to show a bound for the 'closeness' of original and reconstructed data points: $\frac{\Sigma_{i=1}^{m} ||\hat{x}^{(i)} - \tilde{x}_i^{(i)}||^2}{\Sigma_{i=1}^{m} ||\hat{x}^{(i)}||^2} \leq 0.01$, being r most random variables with largest variance

PCA Drawbacks

Lossy compression as we do not regain 100% of the variance (no redundancy). For data with less redundancy, the value of k might be too large such that we would actually end requiring more space since decompression matrix U_{red}, $Z_{(i)}$ and the row means needs to be stored as wel

12. AI & Ethics

- **Privacy and Surveillance, Manipulation of Behavior, Opacity of AI Systems, Bias in detection Systems, Automation and Automous Systems:**
- **Machine Ethics:** Machine ethics is concerned with ensuring that the behavior of machines toward human users, and perhaps other machines as well, is ethically acceptable. (Anderson and Anderson 2007: 15)
- **3 Laws of Robotics:**
  - *First Law:* A robot may not injure a human being or, through inaction, allow a human being to come to harm.
  - *Second Law:* A robot must obey the orders given it by human beings except where such orders would conflict with the *First Law*.
  - *Third Law:* A robot must protect its own existence as long as such protection does not conflict with the *First or Second Laws*.