AI_Principles_Techniques CheatSheet

Huang JunHao & Chen YuXuan

Machine Learning

Hypothesis ($h_\theta(x)$) : The function that maps input features (x) to predicted outputs.

Regression: Predicts a real-valued output (e.g., predicting house prices).

Loss function for regression: Typically Mean Squared Error (MSE), which is $(h_\theta(x) - y)^2$, or Mean Absolute Error (MAE), which is $|h_\theta(x) - y|$.

Linear Regression: Models the relationship between input features and output as a linear function, $h_\theta(x) = \theta^T x$.

Closed-form solution for Linear Regression (Normal Equation): $\theta^* = (X^T X)^{-1} X^T y$.

Loss Functions for Binary Classification:

· **0/1** Loss: $1\{y \cdot h_\theta(x) \leq 0\}$ (penalizes incorrect classification with a value of 1 , correct with 0).

Logistic Loss: $l_{\text{logistic}} = \log\left(1 + \exp\left(-y \cdot h_\theta(x)\right)\right)$ (used in logistic regression).

Hinge Loss: $l_{\text{hinge}} = \max\{1 - y \cdot h_\theta(x), 0\}$ (used in Support Vector Machines).

Exponential Loss: $l_{\text{exp}} = \exp\left(-y \cdot h_\theta(x)\right)$ (used in boosting algorithms).

Multiclass Classification: Extends binary classification to more than two classes through softmax: , softmax $(z)_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$.

Cross-Entropy Loss: $l(z, y) = -\sum_i y_i \log \text{softmax}(z)_i = -\log \text{softmax}(z)_y$.

Kernel Methods: Use a kernel function $K(x, x_i)$ to measure similarity between data points in a highdimensional feature space, where $h_\theta(x) = \sum_{i=1}^{m} \theta_i K(x, x_i)$. Learn nonlinear decision boundaries.

Nearest Neighbor Methods: Classification based on the majority, where $h_\theta(x) = \arg\min_i \text{dist}(x, x_i)$.

Decision Trees: Partition the feature space into different regions, with each region corresponding to a class or value.

Neural Networks: capable of learning intricate nonlinear relationships.

Generalization: How well a model performs on unseen data, which is the ultimate goal.

Overfitting: A phenomenon where a model performs well on training data but poorly on unseen data, indicating it has learned noise rather than the underlying pattern.

Cross-validation: A technique to estimate how well a model will generalize by splitting the dataset into training and holdout sets (e.g., 70% training /30% holdout).

K-fold Cross-validation: Divides the dataset into G disjoint subsets. The model is trained on $G - 1$ subsets and evaluated on the remaining fold, repeating this G times.

Hyperparameters: Parameters whose values are set before the learning process begins (e.g., degree of polynomial, amount of regularization). Cross-validation helps determine optimal hyperparameters.

Regularization: Techniques to prevent overfitting by keeping model weights small. Typically adds a penalty term (e.g., $\frac{\lambda}{2}\|\theta\|^2$)

$\theta^* = (X^T X + I)^{-1} X^T y$ for regularized problems

Unsupervised Learning: Deals with unlabeled data, aiming to find hidden patterns or structures.

K-means: A clustering algorithm that partitions data into k clusters, where each data point belongs to the cluster with the nearest mean (center), $\theta = \{\mu_1, ..., \mu_k\}$ for $\mu_i \in \mathbb{R}^n$. $h_\theta(x) = \arg\min_i \|x - \mu_i\|^2$.

Principal Component Analysis (PCA): A dimensionality reduction technique that transforms data to a new set of orthogonal variables (principal components), $h_\theta(x) = \Theta_1 \Theta_2 X$, where $\Theta_1 \in \mathbb{R}^{n \times k}$, $\Theta_2 \in \mathbb{R}^{k \times n}$ and $k < n$.

Reconstruction Loss: Measures how well the original data can be reconstructed from the reduceddimension representation, $\|x_i - h_\theta(x_i)\|^2$.

Deep Learning

Common Choices of Activation Functions (f_i) :

Hyperbolic Tangent (tanh): $f(x) = \tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$, output range (− 1,1). Sigmoid: $f(x) = \sigma(x) = \frac{1}{1+e^{-x}}$, output range (0,1).

Rectified Linear Unit (ReLU): $f(x) = \max\{x, 0\}$, outputs 0 for negative inputs and the input itself for positive inputs.

Stochastic Gradient Descent (SGD): An optimization algorithm that updates model parameters θ by taking small steps in the direction of the negative gradient of the loss function, $\theta \leftarrow \theta - \alpha \nabla_\theta l\left(h_\theta\left(x^{(i)}\right), y^{(i)}\right)$. Uses mini-batches for efficiency.

Backpropagation: Time complexity $O(n^2 k)$.

Jacobian of a linear transformation: $\frac{\partial(Ax)}{\partial x} = A$. Note that jacobian is the transpose of gradient!

Derivative of Sigmoid: $\sigma'(x) = \sigma(x)(1 - \sigma(x))$.

Momentum Methods: Accelerate SGD by accumulating gradients from previous steps. Typically uses a decay rate $\beta = 0.9$. $v_t = \beta v_{t-1} + (1 - \beta)\nabla_\theta L(\theta)$, $\theta \leftarrow \theta - \alpha v_t$.

Convolutional Neural Networks (CNNs)

Number of parameters: out_channel * (in_channel * kernel_size + 1)

Convolution: Applies filters to input data, preserving spatial locality and sharing weights across activations.

Non-linearity (ReLU): Introduces non-linearity, is fast to compute, helps with sparse activation, and addresses the vanishing gradient problem.

Pooling (or Downsampling): Reduces computational requirements and the likelihood of overfitting by summarizing features in regions. Aggressive pooling can limit network depth.

Fully Connected Layer: Connects every neuron in one layer to every neuron in the next. Number: out * (in + 1)

Limitation: impractical is swapping rows/columns is still useful.

Transfer Learning: Uses pre-trained image classification networks (later-stage layers) as feature extractors for new tasks.

Neural Networks with Memory (RNNs)

Recurrent Neural Networks (RNNs): Designed to process sequences by maintaining an internal "memory" (hidden state).

Backpropagation Through Time (BPTT): The algorithm for training RNNs.

Problems with Vanilla RNNs:

Exploding Gradients: Gradients become very large during backpropagation. Handled by gradient clipping.

Vanishing Gradients: Gradients become very small, hindering learning of long-term dependencies. Addressed by LSTMs/GRUs

Adaptive Learning Rate (Adagrad): Adjusts the learning rate for each parameter, $\eta^t = \frac{\eta}{\sqrt{\sum_{i=0}^{t} \|g^i\|^2}}$.

Dropout: A regularization technique where neurons are randomly dropped during training (p% chance). During testing, no dropout, and weights are scaled by (1 − p).

Search

Tree Search (function TREE-SEARCH): Explores a search space to find a solution or declare failure. Uses a fringe (queue/stack) to store nodes to be expanded.

Uniform Cost Search: Fringe ordered by path cost g(x).

Greedy Search: Fringe ordered by heuristic cost h(x) (estimate of cost to goal).

Star Tree Search: Fringe ordered by f(x) = g(x) + h(x).

Note: node ≠state, state with different paths = different nodes

Admissible (optimistic) Heuristic: $0 \leq h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost to the goal.

A-Star Graph Search (function GRAPH-SEARCH): Modifies tree search to handle graphs by keeping a closed set to avoid re-expanding states. Note that h for nodes in fringe may be modified, but h for expanded nodes cannot be modified.

Consistency (Monotonicity): Heuristic "arc" cost ≤ actual cost for each arc, $h(A) - h(C) \leq \text{Cost(A to C)}$.

Create Heuristics: Relaxed problems, reducing edge costs, max of two heuristics.

Cell decomposition: not resolution complete.

Theta*: If y is discovered when expanding x, if y is visible by parent(x) then insert y with f(y)=g(Parent(x))+c(P(x),y)+h(y)

Weighted A*: f(x) = g(x) + k · h(x), where $k > 1$.

MDP

MDP: A framework for modeling decision-making in situations where outcomes are partly random and partly under the control of a decision-maker.

First-Order Markovian Dynamics: Next state depends only on current state and current action.

State-Dependent Reward: Reward is a deterministic function of the current state.

Stationary Dynamics: Do not depend on time.

Full Observability: The new state is known.

Dynamics: Planning: Dynamics model is known.

Reinforcement Learning: Dynamics model is unknown.

Solution: Policy π: S → A (stationary).

Value Function: $V_\pi(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R^t | \pi, s\right]$, expected discounted return from state s following policy π.

Bellman Equation (for optimal policy): $V^*(s) = R(s) + \gamma \max_{a \in A} \sum_{s' \in S} P(s'|s, a) V^*(s')$.

Value Iteration: Iteratively updates value estimates until convergence.

$V(s) \leftarrow R(s) + \gamma \max_{a \in A} \sum_{s' \in S} P(s'|s, a) \hat{V}(s')$.

Convergence: $\max_s \|BV(s) - BU(s)\| < \gamma \max_s \|V(s) - U(s)\|$.

If $\|Vk - Vk\text{-}1\| < \varepsilon$,then $\|Vk - V^*\| < \varepsilon\ \gamma\ /(1\text{-}\gamma)$

If $\|Vk - V^*\| \leq \lambda$ then $\|Vg - V^*\| \leq 2\lambda\gamma/(1\text{-}\gamma)$， where Vg is value function for greedy policy

Problems: slow ($O(SA^2)$), max always change, policy converged before value

Policy Iteration: Alternates between policy evaluation (calculating V_π) and policy improvement (updating π).

$V_\pi(s) = R(s) + \beta \sum_{s'} T(s, \pi(s), s') \cdot V_\pi(s')$

Asynchronous Dynamic Programming: Backs up states individually, in any order.

In-place DP: Updates values without needing a separate temporary storage.

Prioritized Sweeping: Backs up states with the largest bellman error (need reverse knowledge)

Real-time DP: Backs up relevant states after moving.

Reinforcement Learning

Setting: Small |S|, small |A|, unknown model (agent doesn't know T or R).

Model-based RL: Learns the MDP model (or an approximation).

Model-free RL: Derives optimal policy without explicitly learning the model.

Passive Learning: Agent has a fixed policy and learns V^π by observing the environment.

Direct Estimation (Monte Carlo): Samples and averages returns of V

Adaptive Dynamic Programming (ADP) (model-based): Estimates the model and performs policy evaluation.

Temporal Difference (TD) Learning: Updates value estimates based on differences between current and predicted future rewards, $V^\pi(s) \leftarrow V^\pi(s) + \alpha\left(R(s) + \gamma V^\pi(s') - V^\pi(s)\right)$. (converges slower than ADP)

Active Learning: Agent attempts to find an optimal policy by interacting with the world.

Exploration / Exploitation: Balancing trying new actions (exploration) with choosing known good actions (exploitation).

Greedy in the Limit of Infinite Exploration (GLIE): Gradually reduces randomness in action selection. Uses ϵ-greedy exploration in practice.

Boltzmann Exploration: Selects actions based on a probability distribution, $\Pr(a|s) = \frac{\exp(Q(s,a)/T)}{\sum_{a' \in A} \exp\left(Q(s,a')/T\right)}$, where T is temperature.

Optimistic Value iteration:Equivalent to Rmax algorithm: Start with an optimistic model(assign largest possible reward to “unexplored states”)(actions from “unexplored states” only self transition)

$$V^+(s) \leftarrow R(s) + \beta \max_a \begin{cases} V^{\max}, & N(s,a) < N_e \\ \sum_{s'} T(s,a,s') V^+(s'), & N(s,a) \geq N_e \end{cases}$$

TD based learning: follow exploration/exploitation policy and perform TD update for V. requires estimation for exploration

Q-learning (model-free): Directly learns the optimal action-value function $Q(s,a)$, $Q(s,a) \leftarrow Q(s,a) + \alpha\left(R(s) + \gamma \max_{a'} Q(s',a') - Q(s,a)\right)$. (only requires $|S|*|A|$ space, but might be noisy, off policy)

Trajectory Replay: Stores and reuses past experiences. Reverse Updates: Updates Q-values in reverse order of a trajectory.

SARSA: An on-policy algorithm that updates Q-values based on the actual next action taken, $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(R(s_t, a_t) + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t))$.

V-estimation: follow e/e policy, update estimation of model, update parameters (may use TD-like target v(s))

Also for Q-learning, no need to estimate the model

$$\theta_i \leftarrow \theta_i + \alpha\left(v(s) - \hat{V}_\theta(s)\right) f_i(s) \qquad \theta_i \leftarrow \theta_i + \alpha\left(R(s) + \beta \max_{a'} \hat{Q}_\theta(s',a') - \hat{Q}_\theta(s,a)\right) \frac{\partial \hat{Q}_\theta(s,a)}{\partial \theta_i}$$

DQN: weak for long horizon games

Setting: Large $|S|$, small $|A|$, unknown model.

Function Approximation: Uses deep neural networks to approximate V or Q.

Stable Solutions (for oscillating policy and noisy data)

Experience Replay: Stores transitions and samples mini-batches for training, $\sum_{(s_t,a_t,r_t,s_{t+1})\in B} \left(r_t + \gamma \max_{a'} \hat{Q}(s_{t+1}, a') - Q(s_t, a_t)\right)^2$. Removes correlations, improves data efficiency.

Freeze Target Q-network: Uses an older, fixed set of parameters for the target Q-network to avoid oscillations. Periodically updates these parameters.

Clip Rewards: Binds rewards to a sensible range to prevent large gradient updates.

Extensions:

Double Q-learning: Trains two Q-functions (Q_1, Q_2) and uses one to select the action and the other to evaluate it, preventing overestimation.

Double DQN: use old Q function as action evaluation (target) and the current one as action selection

Prioritized Experience Replay: Samples transitions with higher TD-error more frequently. (proportion to error$^\alpha$)

Dueling Q Networks: Splits the Q-function into state-value (V(s)) and advantage (A(s,a)) functions, $Q(s,a) = V(s) + A(s,a)$.

Multistep Returns: Uses multiple future rewards to estimate the target value.

Problem: (policy gradient) Policies that work well might not approximate V/Q optimally.

Solution: Directly parameterize the policy $\pi_\theta(s,a) = P[a|s,\theta]$. Focus on model-free RL.

Policy Gradient Theorem: $\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[Q^{\pi_\theta}(s,a) \nabla_\theta \log \pi_\theta(s,a)\right]$.

Equivalent Forms: REINFORCE ($\mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a) v_t\right]$), Actor-Critic ($\mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a) Q_w(s,a)\right]$ or $\mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a) A_w(s,a)\right]$), TD Actor-Critic ($\mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a) \delta\right]$).

Why Actor-Critic? Policy gradient with trajectory might be noisy, so directly estimate Q function

Reduce Variance: Techniques like baseline subtraction (e.g., B(s)) are used to reduce variance in gradient estimates, $\mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a) B(s)\right] = 0$.

Convergence: gradient for Q and policy are identical (solution: use MLP on the same extracted feature)

Probabilistic Model

Bayes Net: A directed acyclic graph representing probabilistic relationships between variables. $P(X_1, \ldots, X_n) = \prod_{i=1}^{n} P(X_i | \text{parents}(X_i))$.

Independence in BN:

d-connected: Variables X and Y are dependent given evidence E if there's an active path between them.

d-separated: Variables X and Y are independent given evidence E if there's no active path.

Exact Inference:

Variable Elimination: Computes probabilities by iteratively eliminating variables.

Probability inference: repeatly select hidden variable, eliminate it

Approximate Inference (Sampling):

Prior Sampling: Ignores evidence, samples from the network, then counts valid samples.

Rejection Sampling: Samples from the network and rejects samples inconsistent with evidence.

Likelihood Weighting: Fixes evidence variables, samples non-evidence variables, and weights each sample by the likelihood of the evidence.

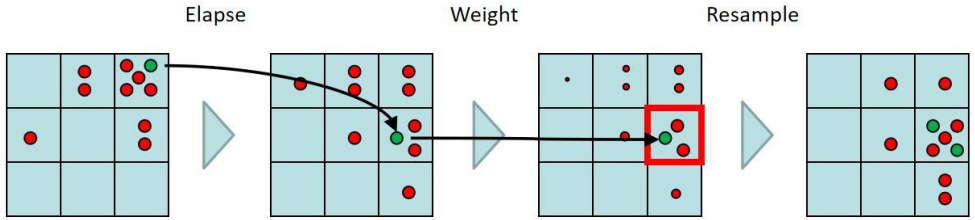
Gibbs Sampling: Initializes non-evidence variables randomly, then repeatedly resamples each non-evidence variable conditioned on others

HMMs: Models where the underlying Markov chain (states X) is hidden, and the agent observes outputs (effects e) at each time step.

Filtering / State Estimation: Tracking the distribution of the current state given all observations up to that point, $B_t(X) = P(X_t | e_{1:t})$.

Particle Filtering: An approximate inference technique that tracks samples of X rather than all values.

Elapse time (passage of time) -> weighting (observe) -> resample



NLP

Most Likely Explanation (MLE): $\arg\max_{x_{1:t}} P(x_{1:t} | e_{1:t})$. Viterbi algorithm: $m_t[x_t] = P(e_t|x_t) \max_{x_{t-1}} P(X_t|X_{t-1}) m_{t-1}[X_{t-1}]$.

Word2Vec: Learns word embeddings (vector representations of words).

Skip-grams: Predicts context words given a center word.

Continuous Bag of Words (CBOW): Predicts a center word from its context words.

Negative Sampling: An efficient training method for Word2Vec, $L = \log \sigma(u_o^T v_c) + \sum_{i=1}^{k} \mathbb{E}_{j\sim P(w)}\left[\log \sigma(-u_j^T v_c)\right]$.

Language Model (LM): Assigns a probability to any sequence of words.

N-gram LM: Predicts the next word based on the previous $N-1$ words, $P(W_1 \ldots W_T) = \prod_t P(W_t | W_{t-N+1:t-1})$.

Perplexity: A measure of how well a probability model predicts a sample, $PPL(W) = 2^{-l}$, where $l = \frac{\log_2(P(W))}{\text{token\_len}(W)}$.

Byte Pair Encoding (BPE) Tokenization: A subword tokenization algorithm that iteratively merges the most frequent pairs of characters or character sequences.

Attention Layer: A mechanism that allows models to weigh the importance of different parts of the input sequence.

Positional Encoding: Adds information about the position of tokens in a sequence, as self-attention lacks this inherent notion.

Masked Self-Attention: Prevents a token from attending to future tokens, used in decoders and language modeling.

Multi-headed Self-Attention: Runs multiple attention mechanisms in parallel, combining their outputs.

Feed-Forward Network: Adds non-linearities after self-attention layers.

Residual Connection: Helps with training deep networks by allowing gradients to flow directly through layers, $X^{(i)} = X^{(i-1)} + \text{Layer}\left(X^{(i-1)}\right)$.

Layer Normalization: Normalizes activations within a layer to stabilize training, $\text{LayerNorm}(x) = \frac{x-\mu}{\sqrt{\sigma^2+\epsilon}} * \gamma + \beta$.

Vision Transformers (ViT): Apply transformer architecture to image data using 2D positional embeddings.

LLM Pretraining: Unsupervised learning for Large Language Models.

Encoders (BERT): Masked language modeling (predicting masked tokens) and next sentence prediction.

Encoder-Decoder (T5): Trains the decoder to predict masked parts of the input.

Decoder (GPT): Next token prediction.

Scaling Laws: Describe how model performance improves with increased compute, data, and parameters.

LLM Post-training:

Supervised Fine-Tuning (SFT): Trains the model on demonstration data.

Reward Modeling (RM): Trains a reward model to score model outputs based on human preferences.

Reinforcement Learning from Human Feedback (RLHF): Optimizes the policy against the reward model using algorithms like PPO.

LLM Prompting: Techniques to improve performance without gradient updates.

Few-shot learning: Provides a few examples in the prompt.

Zero-shot learning: No examples, just the task description.

Chain-of-thought prompting: Encourages the model to show its reasoning steps.

LLM Agent (ReAct): LLMs that can reason and act, calling external tools like search engines, calculators, and code interpreters.

Logic

Logic-based Models: Use propositional logic and first-order logic.

Core Components:

Syntax: Defines valid formulas.

Semantics: Specifies meaning through models (assignments).

Inference Rules: Derive new formulas from existing ones.

Knowledge Base (KB): A set of formulas representing their conjunction.

Key Concepts:

Entailment ($\mathbf{KB} \vDash \mathbf{f}$) : KB implies f if all models of KB are also models of f.

Contradiction: KB contradicts f if no models of KB are models of f.

Satisfiability: A formula is satisfiable if there exists at least one model where it is true.

Inference in Propositional Logic:

Modus Ponens: If $P_1, \ldots, P_k$ and $(P_1 \wedge \cdots \wedge P_k) \rightarrow Q$ are true, then Q is true. Sound but incomplete for general propositional logic; complete for Horn clauses.

Resolution: A complete inference rule for propositional logic, works on clauses in Conjunctive Normal Form (CNF).

Trade-offs: Horn clauses + Modus Ponens (linear time, less expressive); General clauses + Resolution (exponential time, more expressive).

First-Order Logic:Why FOL? Expresses relationships between objects, uses quantifiers (universal ∀, existential ∃), variables, and structured predicates.

Syntax: Terms (constants, variables, functions), Formulas (predicates applied to terms).

Semantics: Models map constants to objects, predicates to relations.

Inference: First-order Modus Ponens: Complete for Horn clauses.

Complexity: Semi-decidable (if KB ⊨ f, it will prove it; if KB ⊭ f, it may not terminate).