STATS-413-Final-F25

Author: Hochan Son

Date: @December 14, 2025

UID: 206547205

Problem 1

Chapter 8: AlphaGo Summary (One-Page)

The game of Go can be formally defined as Board(s), Legal Action(a), Terminal State, and Reward(z)

Core Concept: Planning (System 2) + Neural Networks (System 1)

AlphaGo combines **Monte Carlo Tree Search** (deliberate planning) with **deep neural networks** (intuitive evaluation) to achieve superhuman Go play.

Two Neural Networks (13-layer CNNs)

Policy Network $p_\sigma(a \mid s)$: Predicts move probabilities (softmax over 361 positions; +1 possible moves)

- Trained on expert games, refined via self-play policy gradient:

$$\nabla_\theta J(\theta) = E_{\pi\theta}[\nabla_\theta \log \pi_\theta(a \mid s) \cdot Q^{\pi\theta}(s, a)]$$

Value Network $v_\theta(s) \in [-1, +1]$: Estimates win probability from position

- Trained on self-play outcomes: minimize $(z - v_\theta(s))^2$

MCTS: The Search Engine

Four phases per simulation:

1. **Selection**: Navigate tree using PUCT formula

$$a_t = \text{arg max}\left(Q(s_t, a) + cp(a \mid s_t) \cdot \frac{\sqrt{\sum_b N(s_t, b)}}{1 + N(s_t, a)}\right)$$

2. **Expansion**: For leaf node (s_L), add new child node to tree:

$$\begin{aligned} N(s_L, a) &= 0, & \text{(visit count)} \\ W(s_L, a) &= 0, & \text{(cummulative value)} \\ Q(s_L, a) &= 0 & \text{(average value)} \end{aligned}$$

3. **Evaluation**: Use value network or rollout: $v_{leaf} = v_\theta(s_L)$

4. **Backup**: Propagate value up, update :

$$\begin{aligned} N(s_t, a_t) &\leftarrow N(s_t, a_t) + 1, & \text{(update visit count)} \\ W(s_t, a_t) &\leftarrow W(s_t, a_t) + v_{leaf}, & \text{(update cummul. v.)} \\ Q(s, a) &\leftarrow W(s_t, a_t)/N(s_t, a_t) & \text{(update avg. v)} \end{aligned}$$

Key insight: Neural networks make intractable search feasible

- Policy reduces **breadth**: 250 moves → ~10-20 promising candidates
- Value reduces **depth**: 150-move rollout → instant evaluation

AlphaGo → AlphaGo Zero Evolution

Original AlphaGo:

$$\boxed{\text{Human expert games} \rightarrow \text{self-play refinement} \rightarrow \text{freeze networks} \rightarrow \text{use MCTS}}$$

AlphaGo Zero breakthrough: Bootstrap from random weights via self-play loop

1. Run MCTS with current networks → generate move distribution π_θ MCTS
2. Train networks to imitate MCTS
3. Iterate with improved networks

Result: Zero defeats original AlphaGo 100-0, discovers novel strategies without human supervision

System 1 vs System 2 Framework

System 1 (Neural nets): Fast pattern recognition, learned intuition **System 2** (MCTS): Slow deliberate search, explicit planning

Training paradigm: System 2 teaches System 1

- MCTS planning creates high-quality targets
- Neural networks compress search into instant evaluation
- "Planning is more fundamental than learning"

Key Achievement

Beat world champion Lee Sedol (March 2016), demonstrating AI can master complex intuitive domains through hybrid architecture combining classical search with modern deep learning.

Problem 2

https://randomrealizations.com/posts/xgboost-from-scratch/

Chapter 9 – Trees and Boosting Summary

Overview

Chapter 9 recasts regression trees and boosting as alternative forms of the same incremental-improvement principle that drives gradient-descent training of neural networks. The chapter progresses from decision trees through L2 boosting and XGBoost, interprets boosting as surrogate-loss minimization with implicit regularization, contrasts AdaBoost with modern gradient boosting, and ends with Random Forests as a parallel ensemble counterpart.

9.1 Incremental Viewpoint

Gradient descent updates parameters (Eq. 9.1), trees refine partitions by recursive splits, and boosting adds trees in an additive model. All three minimize objectives of the form (Eq. 9.2) and embody bias→variance trade-offs as the number or size of increments grows.

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta L(\theta_t) \tag{9.1}$$

$$\sum_i L(y_i, f(x_i)) + \text{complexity}(f) \tag{9.2}$$

9.2 Decision Trees

- Trees partition feature space via binary splits guided by purity metrics such as misclassification rate (Eq. 9.3), Gini index (Eq. 9.4) or entropy (Eq. 9.5).

$$error(R) = 1 - \max_k p_k(R) \tag{9.3}$$

$$Gini(R) = 1 - \sum_k p_k(R)^2 \tag{9.4}$$

$$Entropy(R) = -\sum_k p_k(R) log_2(p_k(R)) \tag{9.5}$$

- The split at region (R) and feature threshold (s) maximizes impurity decrease (ΔI) (Eq. 9.7). For regression, each leaf predicts the average response (Eq. 9.8) and minimizes squared error (Eq. 9.9).

$$s^* = \arg\max_s \Delta I(s, r) \tag{9.7}$$

$$\hat{c}_R = \frac{1}{|R|} \sum_{i:x_i \in R} y_i \tag{9.8}$$

$$I(R) = \frac{1}{|R|} \sum_{i:x_i \in R} (y_i - \hat{c}_R)^2 \tag{9.9}$$

9.3 Regression Trees

- Represented as piecewise-constant functions (Eq. 9.10), fitted by minimizing squared error plus complexity penalties (Eq. 9.11).

$$f(x) = \sum_m c_m \mathbf{1}_{x \in R_m} \tag{9.10}$$

$$\min_{\{R_m\}_{m=1}^M, \{c_m\}_{m=1}^M} \left\{ \sum_{i=1}^n (y_i - \sum_{m=1}^M c_m I\{x_i \in R_m\})^2 + \lambda M \right\} \tag{9.11}$$

- Recursive binary splitting evaluates candidate feature-threshold pairs via RSS reduction and grows the tree until stopping rules (node size, impurity) are met.
- Cost-complexity pruning evaluates subtrees via validation or cross-validation to avoid overfitting (Eq. 9.13).

9.4 L2 Boosting / Gradient Boosted Trees

- Boosting constructs (Eq. 9.14) where each tree (h_m) has leaf regions (R_{mj}) with predictions (c_{mj}) (Eq. 9.15).

$$f_M(x) = \sum_{m=1}^M h_m(x) \tag{9.14}$$

$$h_m(x) = \sum_{j=1}^{J_m} c_{mj} I\{x \in R_{mj}\} \tag{9.15}$$

- Adding a tree solves a regularized least squares problem (Eq. 9.16), yielding optimal leaf values (Eq. 9.19). Weighted variants handle observation weights (w_i) (Eqs. 9.20–9.23).

$$Gain = \Delta L = \frac{(\sum_{i \in R_L} w_i r_i)^2}{\sum_{i \in R_L} w_i + \lambda} + \frac{(\sum_{i \in R_R} w_i r_i)^2}{\sum_{i \in R_R} w_i + \lambda} - \frac{(\sum_{i \in R} w_i r_i)^2}{\sum_{i \in R} w_i + \lambda} - \gamma \tag{9.23}$$

- Algorithm 9.4 collects residuals (negative gradients), fits a tree with shrinkage (learning rate), and updates the ensemble.
- The procedure is equivalent to functional gradient descent, and setting (λ, γ) controls tree complexity.

9.5 XGBoost for Logistic Regression

- Logistic models use ($p = 1/(1 + e^{-f(x)})$) (Eq. 9.24) with additive trees (Eq. 9.25). Logistic loss ($l(y_i, f) = -y_i \log p - (1 - y_i)\log(1 - p)$) (Eq. 9.26) yields gradients ($r_i = p - y_i$) (Eq. 9.30) and Hessians ($w_i = p(1 - p)$) (Eq. 9.31).

- Adding tree (h_m) minimizes the second-order surrogate (Eq. 9.40) leading to a weighted least squares fit with weights ($w_i = h_i$) and working responses ($z_i = -g_i/h_i$) (Eq. 9.41).

$$L(f_m + h) \approx \sum_{i=1}^{n}[l(y_i, f_m(x_i) + g_i h_m(x_i) + \tfrac{1}{2} h_i h_m(x_i) + \hat{2}] \qquad (9.40)$$

- Split scoring becomes (Eqs. 9.23).
- Multiple interpretations strengthen intuition: Newton steps in function space, geometric “golf” analogy, and connections to back-propagation and IRLS (Eqs. 9.36–9.41).

9.6 Surrogate Losses & Incremental Learning

- Gradient descent minimizes quadratic surrogates (Eqs. 9.42–9.43). Boosting minimises surrogate losses such as exponential (AdaBoost), logistic (LogitBoost/XGBoost), or squared error (L2Boosting) (Eqs. 9.44–9.46).
- XGBoost’s surrogate (Eq. 9.47) uses gradients and Hessians, leading to efficient tree search and regularization. Good surrogates upper-bound the true loss, match at current iterate, and are easily optimized.

9.7 Lazy/Implicit Regularization

- Gradient flow in function space (Eq. 9.50) illustrates “lazy” trajectories: small learning rates (Eq. 9.51) prioritize low-frequency/global structure first (Eq. 9.52) and control complexity roughly proportional to ($M\eta$) (Eq. 9.53).
- This implicit bias parallels deep networks: early stopping acts as regularization, incremental updates bound changes (Eqs. 9.54–9.56), and spectral bias emerges naturally.

9.8 AdaBoost vs XGBoost

- Loss functions: AdaBoost’s exponential loss (Eq. 9.78) vs XGBoost’s logistic (Eq. 9.79) highlight sensitivity to outliers and Hessian availability.
- Base learners: AdaBoost typically uses shallow stumps with ±1 outputs; XGBoost uses deeper regression trees with real-valued leaf outputs.
- Update rules: AdaBoost multiplies observation weights, while XGBoost performs Newton-style gradient/Hessian updates. Tables 9.2–9.3 summarize differences in

regularization, parallelization, memory, and use cases.

9.9 Random Forests

- Ensemble of independently grown trees: ($f_M(x) = \frac{1}{M}\sum_{m=1}^{M} T_m(x)$) (Eq. 9.80).
- Randomness sources: bootstrap sampling (Eq. 9.81) and feature subsampling with ($k \approx \sqrt{p}$) (Eq. 9.82). Construction algorithm (Alg. 11) evaluates splits using RSS (Eq. 9.83).
- Statistical properties: variance decomposes via tree correlation (Eq. 9.84); out-of-bag prediction (Eq. 9.85) yields unbiased error estimates (Eq. 9.86).
- Variable importance: mean decrease in impurity (Eq. 9.87) and mean decrease in accuracy via permutation (Eq. 9.88).
- Consistency and convergence results (Eqs. 9.89–9.90) hold under depth/size conditions. Table 9.4 compares Random Forest, AdaBoost, and XGBoost.

XGBoost from Scratch:

This post provides a walkthrough and Python implementation of the core statistical learning algorithm of **XGBoost (eXtreme Gradient Boosting)**, using only `numpy` and `pandas`. The goal is demystification by building the algorithm from the ground up, focusing on the key differences from a classic Gradient Boosting Machine (GBM).

1. **Implementation Structure**
   The implementation consists of two main classes:
   a. `XGBoostModel` **(The Booster):**
      i. **Initialization:** Handles hyperparameters (e.g., `learning_rate`, `max_depth`, `subsample`, `base_score`) and sets up a `defaultdict` for parameter handling.
      ii. `fit()` **Method:**
         1. Initializes predictions to `base_prediction`
         2. Iterates through boosting rounds:
            a. Computes the **gradients (g)** and **Hessians (h)** of the user-defined objective function w.r.t. the current predictions.
               i. (Optional) Performs row **subsampling.**
               ii. Fits a `TreeBooster` using g and h.

iii. Updates the predictions: $P_t = P_{t-1} + \eta \cdot \text{Tree}_t(X)$, where η is the learning_rate.

iv. Stores the sequence of fitted TreeBooster instances.

iii. predict() **Method:** Sums the predictions from the base score and all fitted boosters, scaled by the learning rate.

b. TreeBooster **(The Modified Decision Tree):**

$$\text{Value} = -\frac{\sum g_i}{\sum h_i + \lambda} \tag{1}$$

$$\text{Gain} = \frac{1}{2}\left[\frac{(\sum_{i\in I_L} g_i)^2}{\sum_{i\in I_L} h_i + \lambda} + \frac{(\sum_{i\in I_R} g_i)^2}{\sum_{i\in I_R} h_i + \lambda} - \frac{(\sum_{i\in I} g_i)^2}{\sum_{i\in I} h_i + \lambda}\right] - \frac{\gamma}{2} \tag{2}$$

- Node Value Calculation: The predicted value for a leaf node is defined by a simple function of the gradients and Hessians of the instances in that node, based on Equation 5 from the XGBoost paper:

  (Where λ is the L_2 regularization term reg_lambda). (eq 1)

- Split Finding Criterion: Instead of using an impurity measure (like Gini or Entropy), splits are evaluated using the XGBoost Gain Criterion (Equation 7), which incorporates gradients, Hessians, and regularization terms (γ for complexity control and λ): (eq 2)

- **Termination Criteria:** Tree growth stops if the maximum depth is reached, or if the best potential split results in a gain of 0, or if a child node's total Hessian ($\sum h_i$) is below the min_child_weight threshold.

c. **Testing and Benchmarking**

The scratch-built model is tested on the California housing dataset using a custom SquaredErrorObjective class (which defines the loss, gradient, and Hessian functions). The performance (mean squared error) is benchmarked against the official XGBoost library, showing a highly consistent result, validating the from-scratch implementation of the core algorithm

Implementation

- Main.py

main.py

- xgboost_scratch.py

xgboost_scratch.py

```
(base)  🐍 base y hobangu@Mac-mini y /Volumes/T7_Touch/Projects/UCLA-MASD
S/stats413/Final y ↰ HW_submit ±✚ y python main.py

/Volumes/T7_Touch/Projects/UCLA-MASDS/stats413/Final/xgboost_scratch.py:114:
FutureWarning: Series.__getitem__ treating keys as positions is deprecated. In a fut
ure version, integer keys will always be treated as labels (consistent with DataFram
e behavior). To access a value by position, use `ser.iloc[pos]`
  child = self.left if row[self.split_feature_idx] <= self.threshold \
/Volumes/T7_Touch/Projects/UCLA-MASDS/stats413/Final/xgboost_scratch.py:114:
FutureWarning: Series.__getitem__ treating keys as positions is deprecated. In a fut
ure version, integer keys will always be treated as labels (consistent with DataFram
e behavior). To access a value by position, use `ser.iloc[pos]`
  child = self.left if row[self.split_feature_idx] <= self.threshold \

scratch score: 0.2434125759558149
xgboost score: 0.24385224475634615

scratch training time: 29.99s
xgboost training time: 0.18s
speedup: 166.18x
```

Problem 3

Introduction

The Support Vector Machine (SVM), introduced by Vladimir Vapnik and Alexey Chervonenkis, is a powerful classification algorithm that works by mapping observations as points in space and finding an optimal **hyperplane** or **decision boundary** to separate two classes. The goal is to maximize the **marginal distance**, which is the gap between the decision boundary and the closest data points, called **support vectors**. A larger margin leads to better generalization performance on unseen data. While SVMs are fast, robust, and effective for many classification tasks, they can struggle with **non-linear separable** data, a problem often addressed using a **Kernel function** to map the data into a higher-dimensional space where a linear separation is possible. Key applications include sentiment analysis, spam detection, and image recognition.

It is considered one of the top five classification algorithms in machine learning, alongside K-Nearest Neighbors, Random Forest, and Naive Bayes.

Goal of Primal SVM:

- The primary objective is maximization of the margin between the two difference classes. This is achieved by finding the optimal hyperplane (the decision boundary) that maximizes the distance to the closest data points, known as support vectors.
- Marginal distance (M) between the positive and negative hyperplanes is given by $M = \frac{2}{||w||}$, where w is the weight vector normal to the hyperplane. To maximize M, one must minimize the term $||w||^2$

Key Terminology

- Kernel function: a mechanism to transform the data point into higher dimensional space, then hyperplane can be in this space
- Linear, polynomial, radial basis function and sigmoid
- Hyperplane: the decision boundary that can be used to classify data when the decision boundary is more than 2-dimensional, it is called a hyperplane

Soft Margin SVM (Handling Errors):

The ideal large-margin classification assumes perfect linear separation, in reality, some training errors are often allowed to prevent overfitting, leading to the Soft Margin SVM.

The optimization problem is modified by introducing two additional terms: the number of allowed in the training(controlled by the parameter C) and the sum of the value of the error($\sum \xi_i$, where ξ_i are slack variables)

Optimization Term:

$$SVC = \underbrace{\min \frac{1}{2}||w||^2}_{regularizer} + \underbrace{C \sum_{i=1}^{N} \xi_i}_{error\ term} \tag{3}$$

$$SVR = \underbrace{\min \frac{1}{2}||w||^2}_{regularizer} + \underbrace{C \sum_{i=1}^{N} (\xi_i + \xi_i^*)}_{error\ term} \tag{4}$$

where,

- C= number of error in training
- ξ= the deviation from the point to positive edge of the hyperplane
- ξ^*= the deviation from the point to the negative edge of the hyperplane

Loss Function (hinge loss) w.r.t. SVR(support vector regression)

- To minimize: $\frac{1}{2}||w||^2$

$$\text{subject to} \begin{cases} y_i - (w^T x_i + b) \le \varepsilon + \xi_i \\ (w^T x_i + b) - y_i \le \varepsilon + \xi_i^* \\ \xi_i, \xi_i^* \ge 0 \end{cases} \tag{5}$$

where

- Target Value(y_i): This is the actual (unknown) target value for the input x_i.
- Prediction ($w^T x_i + b$): the prediction value (the regression hyperplane).
- Tolerance(ε): the margin of error, or the tube around the regression line.
- Slack Variable (ξ_i, ξ_i^*): These variable (analogous to ζ in Soft margin SVM) represent the error or deviation of a data point outside the ε-tube.

Hinge loss

The hinge loss function, $f(t) = max\{0, 1-t\}$, measures the penalty for a misclassified or poorly classified data point. If the value of $t \quad \ge 1, f(x)$, ie, the hinge loss will be zero for correct prediction. If $t \leq 0$, the hinge loss returns a value of 1 or larger. By using this hinge loss function it gives an **unconstrained optimization term**:

$$\underbrace{\min \frac{1}{2}||w||^2}_{regularizer} + \underbrace{C\sum_{i=1}^{N}(max\{0, 1 - y_n(w^T + b)\})}_{error\ term} \tag{6}$$

- Regularization term: as established, minimizing this term maximize the margin
- Error term: This is the sum of the Hinge Losses for all data point

The goal of the algorithm is to use an optimization method like **Gradient Descent** to find the optimal $\mathbf{w}$ and b that minimize this entire expression.

Implementation

- SVM.py

  SVM.py

- prediction.py

  prediction.py

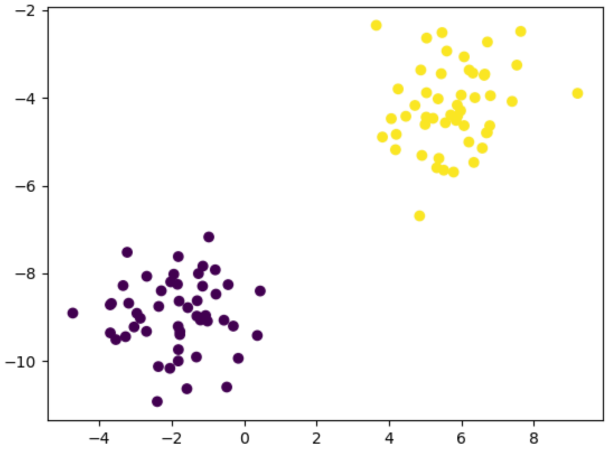
Outcome

- CLI

```
(base)  🐍 base y hobangu@Mac-mini y /Volumes/T7_Touch/Projects/UCLA-MASDS/stats413/Final y ÷ main ± y python prediction.py
Loss: 0.1105510127774915
Prediction: [-1.  1. -1. -1.  1.  1.  1.  1. -1.  1. -1. -1.  1.  1. -1. -1.  1. -1.
  1. -1.  1.  1. -1.  1.  1.  1. -1. -1. -1. -1.  1. -1. -1.  1.  1. -1.
  1. -1. -1.  1.  1.  1.  1.  1.  1. -1.  1.  1.  1.  1.]
Accuracy: 1.0
w, b: [array([[0.44477983, 0.15109913]]), 0.05700000000000004]
```

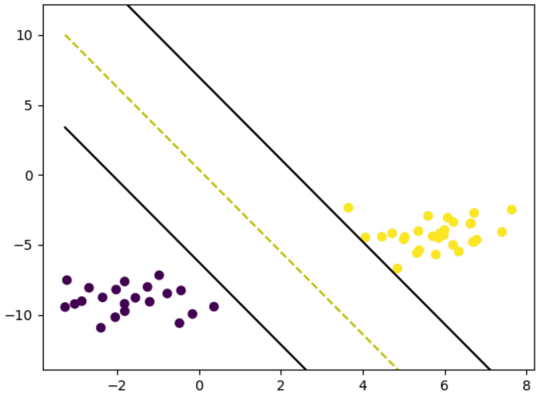
- SVM Plots
  - The plot1 shows the raw dataset (x) with the points colored according to their class labels (y)

    STATS413_Final_p3_figure_1.png



  - The plot2 visualizes the result of the training on the test set(X_test). It overlays the learned geometric structures onto the data
    - The hyperplane (decision boundary): this is the central line (typically dashed or yellow in the reference implementation) defined by the equation $(w \cdot x + b = 0)$
    - The margins (the gutters): these are the two parallel lines (typically solid or black) defined by $(w \cdot x + b = 1 \text{ and } w \cdot x + b = -1)$

    STATS413_Final_p3_figure_2.png



Reference

https://www.quarkml.com/2022/11/support-vector-regression-a-complete-guide-with-example.html

https://www.quarkml.com/2022/09/primal-formulation-of-svm-simplified.html

https://www.quarkml.com/2022/09/primal-formulation-of-svm-simplified.html