COMS 6998 - 015 Homework 1

Yuehui Ruan — UNI: yr2453

September 30, 2024

Code link: Github

https://github.com/Ricky-lab/cs6998-DL-Sys/tree/main

Problem 1.1

(Statement of problem goes here.)

Proof. According to the hints and problems showen above:

- $f(x)$ be the true function (the unknown relationship between input x and output y),
- ϵ be the noise term, so that the true output $y(x)$ is given by:

$$y(x) = f(x) + \epsilon$$

- We have this from hint:

$$MSE = \frac{1}{t}\Sigma_{i=1}^{t}(f(x_i) + \epsilon - g(x_i))^2 \tag{1}$$

The Mean Squared Error (MSE) for a single test instance is defined as:

$$\text{MSE} = \mathbb{E}\left[(y(x) - \hat{y}(x))^2\right]$$

Substituting $y(x) = f(x) + \epsilon$, we get:

$$\text{MSE} = \mathbb{E}\left[(f(x) + \epsilon - g(x))^2\right]$$

Now we try to expand the square, we have:

$$\text{MSE} = \mathbb{E}\left[(f(x) - g(x))^2 + 2(f(x) - g(x))\epsilon + \epsilon^2\right]$$

We can assume that the noise ϵ is independent of $g(x)$ and has zero mean ($\mathbb{E}[\epsilon] = 0$) and variance σ^2 ($\mathbb{E}[\epsilon^2] = \sigma^2$), the cross-term involving ϵ disappears:

$$\mathbb{E}[2(f(x) - g(x))\epsilon] = 0$$

Thus, we are left with:

$$\text{MSE} = \mathbb{E}\left[(f(x) - g(x))^2\right] + \sigma^2$$

We decompose the term $\mathbb{E}\left[(f(x) - g(x))^2\right]$ by introducing the expected model prediction $\mathbb{E}[g(x)]$. This gives us the bias-variance decomposition:

$$\mathbb{E}\left[(f(x) - g(x))^2\right] = (f(x) - \mathbb{E}[g(x)])^2 + \mathbb{E}\left[(g(x) - \mathbb{E}[g(x)])^2\right]$$

Where: $(f(x) - \mathbb{E}[g(x)])^2$ is the **bias squared**, and $\mathbb{E}\left[(g(x) - \mathbb{E}[g(x)])^2\right]$ is the **variance** Substituting these terms back into the MSE expression, we get:

$$\text{MSE} = \left(f(x) - \mathbb{E}[g(x)])^2\right) + \mathbb{E}\left[(g(x) - \mathbb{E}[g(x)])^2\right] + \sigma^2$$

Therefore, the expected Mean Squared Error can be written as:

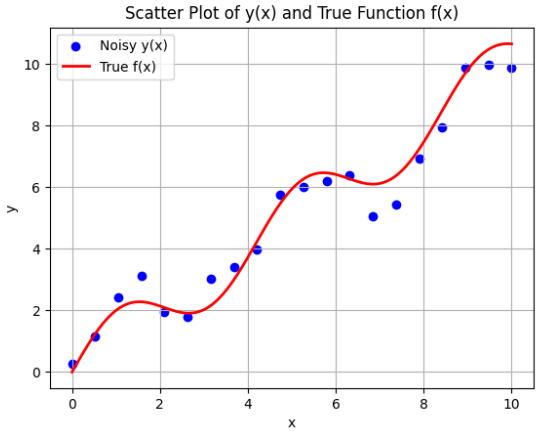
$$E[\text{MSE}] = \text{Bias}^2 + \text{Variance} + \text{Noise}$$

□

1.2

(Statement of problem goes here.)

Proof. Shown in file HW1-2.py.



□

1.3

Proof. We are tasked with fitting polynomial estimators $g_1(x), g_3(x)$, and $g_{10}(x)$ to a noisy dataset $y(x) = f(x) + \epsilon$, where $f(x) = x + \sin(1.5x)$ and $\epsilon \sim \mathcal{N}(0, 0.3)$. We will use polynomial regression to estimate the coefficients for each of the three models and analyze their behavior in terms of underfitting and overfitting.
The polynomial estimators are of the following form:

$$g_n(x) = \beta_0 + \beta_1 x + \beta_2 x^2 + \ldots + \beta_n x^n$$

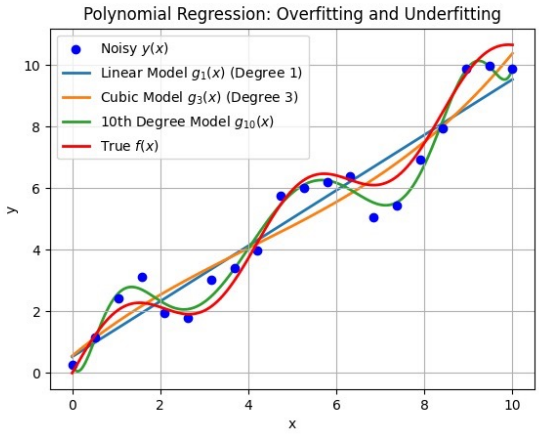
Where:
- $g_1(x)$ is a first-degree polynomial (linear regression), - $g_3(x)$ is a third-degree polynomial (cubic regression), - $g_{10}(x)$ is a tenth-degree polynomial.
We fit these estimators using least squares to minimize the sum of squared errors (SSE) between the true $y(x)$ values and the predicted $g_n(x)$ values.

For each model, we solve the following optimization problem:

$$\min_{\beta_0,\beta_1,\ldots,\beta_n} \sum_{i=1}^{t} (y_i - g_n(x_i))^2$$

1. Underfitting: The linear model $g_1(x)$ is expected to underfit the data because it is too simple to capture the nonlinearities in the function $f(x) = x + \sin(1.5x)$. The model lacks the flexibility to model the sine term effectively.
2. Reasonable Fit: The cubic model $g_3(x)$ is expected to provide a reasonable fit to the data. Cubic polynomials have enough flexibility to model moderate nonlinearities, which should capture the behavior of $f(x)$ reasonably well without overfitting.
3. Overfitting: The tenth-degree polynomial $g_{10}(x)$ is expected to overfit the data. High-degree polynomials can fit the training data very closely, including the noise ϵ, which leads to poor generalization on unseen test data.

1



We will generate a scatter plot of the noisy $y(x)$ values, along with line plots for $f(x)$, $g_1(x)$, $g_3(x)$, and $g_{10}(x)$ over the range $x \in [0, 10]$. Sorting the x-values ensures a clean, readable plot.
We analyze the three models: - $g_1(x)$ will show underfitting, characterized by poor performance on both the training data and unseen test data. - $g_3(x)$ will show a reasonable fit, providing good generalization on test data without being too sensitive to noise in the training data. - $g_{10}(x)$ will show overfitting, fitting the training data almost perfectly but performing poorly on the test data due to its sensitivity to the noise component ϵ.

□

1.4

Proof. See HW1-4.py

□

1.5

Proof. We apply L2 regularization (Ridge regression) to a 10th-degree polynomial and compare its bias, variance, and mean squared error (MSE) to the unregularized 10th-degree polynomial model. The regularized model adds a penalty term to the cost function:

$$\text{Cost} = \sum_{i=1}^{n} (y_i - g_{10}(x_i))^2 + \lambda \sum_{j=0}^{10} \beta_j^2$$

where λ is the regularization rate and β_j are the model coefficients.
Bias:
Regularized model: Introduces higher bias due to the penalty on large coefficients. It restricts the model's flexibility, preventing it from fitting the data as closely.
Unregularized model: Lower bias, as it can fit the data more exactly without restrictions on the coefficients.

Variance:
Regularized model: Reduces variance by limiting overfitting. The regularization prevents the model from fitting noise in the training data.
Unregularized model: Higher variance, as the model can overfit the training data, making it sensitive to noise.

MSE:
Regularized model: Typically achieves lower MSE on the test set due to the better balance between bias and variance.
Unregularized model: Has higher MSE on the test set, as it tends to overfit, resulting in higher variance.

The regularized 10th-degree polynomial has higher bias but lower variance compared to the unregularized model. This leads to a lower MSE on the test set for the regularized model, as it generalizes better by reducing overfitting.

□

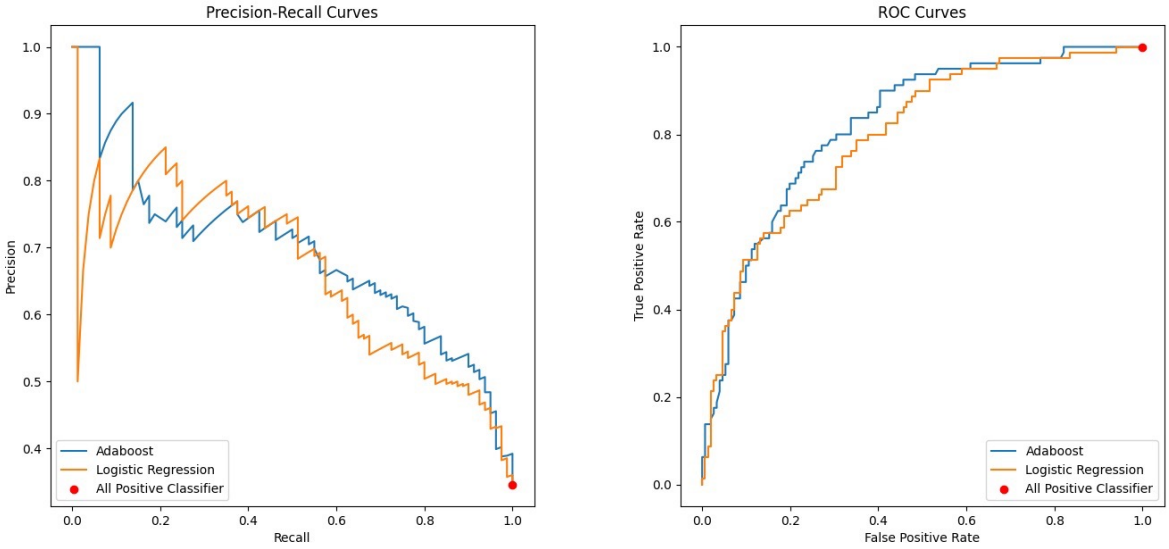
2.1

Proof. **True Negative in ROC and PR Curves:** ROC curve considers true negatives (TN) as it plots True Positive Rate (TPR) vs. False Positive Rate (FPR), where FPR involves TN. In contrast, PR curve does not consider TN, as it focuses on Precision and Recall, which involve true positives (TP) and false positives (FP).
One-to-One Correspondence: Each point on the ROC curve corresponds to a unique point on the PR curve because both are derived from the same confusion matrix components (TP, FP, FN, TN). Although PR excludes TN, it can still be inferred since both FPR and TPR affect precision and recall. Therefore, a one-to-one mapping is possible between the ROC and PR curves.

□

2.2:

Proof. See HW2-2.py



□

2.3:

Proof. The AUROC and AUPR were calculated using the following methods:

- AUROC was computed using the **roc-auc-score** function from **sklearn**.
- AUPR was computed using the **average-precision-score** function.
- AUPRG was calculated by transforming precision and recall into gain-space as described in the NIPS 2015 paper.

The PR Gain curve corrects for class imbalance by adjusting both precision and recall against their baselines. The AUPRG measures the area under this normalized curve, providing a clearer interpretation when there are imbalanced classes, as in our case.
Comparing the results, PR Gain curves are more stable and offer a better reflection of classifier performance, especially when precision and recall values are low or fluctuate due to data imbalance.
Thus, we agree with the NIPS paper's conclusion that practitioners should prefer PR Gain curves to traditional PR curves, as the latter can be misleading in imbalanced datasets.

□

3.1

Proof. See HW1-3-1.ipynb □

3.2

Proof. See HW1-3-2.ipynb □

3.3

Proof. See HW1-3-3.ipynb □

4.1

Layer	Number of Activations (Memory)	Parameters (Compute)
Input	$224 \times 224 \times 3 = 150K$	0
Conv3-64	$224 \times 224 \times 64 = 3.2M$	$(3 \times 3 \times 3) \times 64 = 1,728$
Conv3-64	$224 \times 224 \times 64 = 3.2M$	$(3 \times 3 \times 64) \times 64 = 36,864$
POOL2	$112 \times 112 \times 64 = 800K$	0
Conv3-128	$112 \times 112 \times 128 = 1.6M$	$(3 \times 3 \times 64) \times 128 = 73,728$
Conv3-128	$112 \times 112 \times 128 = 1.6M$	$(3 \times 3 \times 128) \times 128 = 147,456$
POOL2	$56 \times 56 \times 128 = 400K$	0
Conv3-256	$56 \times 56 \times 256 = 800K$	$(3 \times 3 \times 128) \times 256 = 294,912$
Conv3-256	$56 \times 56 \times 256 = 800K$	$(3 \times 3 \times 256) \times 256 = 589,824$
Conv3-256	$56 \times 56 \times 256 = 800K$	$(3 \times 3 \times 256) \times 256 = 589,824$
Conv3-256	$56 \times 56 \times 256 = 800K$	$(3 \times 3 \times 256) \times 256 = 589,824$
POOL2	$28 \times 28 \times 256 = 200K$	0
Conv3-512	$28 \times 28 \times 512 = 400K$	$(3 \times 3 \times 256) \times 512 = 1,179,648$
Conv3-512	$28 \times 28 \times 512 = 400K$	$(3 \times 3 \times 512) \times 512 = 2,359,296$
Conv3-512	$28 \times 28 \times 512 = 400K$	$(3 \times 3 \times 512) \times 512 = 2,359,296$
Conv3-512	$28 \times 28 \times 512 = 400K$	$(3 \times 3 \times 512) \times 512 = 2,359,296$
POOL2	$14 \times 14 \times 512 = 100K$	0
Conv3-512	$14 \times 14 \times 512 = 100K$	$(3 \times 3 \times 512) \times 512 = 2,359,296$
Conv3-512	$14 \times 14 \times 512 = 100K$	$(3 \times 3 \times 512) \times 512 = 2,359,296$
Conv3-512	$14 \times 14 \times 512 = 100K$	$(3 \times 3 \times 512) \times 512 = 2,359,296$
Conv3-512	$14 \times 14 \times 512 = 100K$	$(3 \times 3 \times 512) \times 512 = 2,359,296$
POOL2	$7 \times 7 \times 512 = 25K$	0
FC	4096	$7 \times 7 \times 512 \times 4096 = 102,760,448$
FC	4096	$4096 \times 4096 = 16,777,216$
FC	1000	$4096 \times 1000 = 4,096,000$
TOTAL	15,237,608	138,344,128

Table 1: VGG19 Parameters and Memory Requirements

Proof. □

4.2-a

Proof. The Inception module, as proposed by Szegedy et al. in the GoogLeNet architecture, is designed to efficiently capture information at multiple scales within the same layer by using parallel convolutional filters of different sizes, followed by pooling and concatenation of their outputs.
Key Ideas:

- **Multiple Convolutions:** The Inception module applies 1x1, 3x3, and 5x5 convolutional filters in parallel to the same input. Each filter size captures features at different spatial resolutions:
  - 1x1 convolution: Captures fine-grained, local features and also reduces dimensionality.
  - 3x3 convolution: Captures medium-scale spatial features.
  - 5x5 convolution: Captures larger-scale spatial features.
- **Pooling:** A max pooling layer is applied in parallel to capture downsampled information, which is useful for translation invariance.
- **Concatenation:** The outputs of all parallel operations (1x1, 3x3, 5x5 convolutions, and pooling) are concatenated depth-wise, combining different levels of feature abstraction from the same input.

Naive vs. Dimension Reduction (Figure 2):

- **Naive Inception Module (a):** Directly applies the convolutional filters and pooling, followed by concatenation. This approach increases computational cost, especially with large filter sizes like 5x5.
- **Inception with Dimension Reduction (b):** Introduces 1x1 convolutions before applying 3x3 and 5x5 filters to reduce the input depth, thereby significantly reducing computational cost and memory requirements while retaining multi-scale feature extraction.

□

4.2-b

Proof. **Input size:** $32 \times 32 \times 256$
Naive Inception Module:

- 1×1 convolutions with 128 filters:

$$\text{Output size} = 32 \times 32 \times 128$$

- 3×3 convolutions with 192 filters:

$$\text{Output size} = 32 \times 32 \times 192$$

- 5×5 convolutions with 96 filters:

$$\text{Output size} = 32 \times 32 \times 96$$

- 3×3 max pooling:

$$\text{Output size} = 32 \times 32 \times 256$$

- Total output size after concatenation:

$$32 \times 32 \times (128 + 192 + 96 + 256) = 32 \times 32 \times 672$$

Inception with Dimensionality Reduction:

- 1×1 convolutions with 128 filters:

$$\text{Output size} = 32 \times 32 \times 128$$

- 1×1 convolutions with 32 filters followed by 3×3 convolutions with 192 filters:

$$\text{Output size} = 32 \times 32 \times 192$$

- 1×1 convolutions with 32 filters followed by 5×5 convolutions with 96 filters:

$$\text{Output size} = 32 \times 32 \times 96$$

- 3×3 max pooling followed by 1×1 convolutions with 64 filters:

$$\text{Output size} = 32 \times 32 \times 64$$

- Total output size after concatenation:

$$32 \times 32 \times (128 + 192 + 96 + 64) = 32 \times 32 \times 480$$

Conclusion:

- **Naive Inception Module output size:** $32 \times 32 \times 672$
- **Inception with Dimensionality Reduction output size:** $32 \times 32 \times 480$

□

4.2-c

Proof. **Naive Inception Module:**

- 1×1 convolutions with 128 filters:

$$32 \times 32 \times 128 \times (1 \times 1 \times 256) = 33,554,432$$

- 3×3 convolutions with 192 filters:

$$32 \times 32 \times 192 \times (3 \times 3 \times 256) = 56,623,104$$

- 5×5 convolutions with 96 filters:

$$32 \times 32 \times 96 \times (5 \times 5 \times 256) = 62,914,560$$

- **Total operations (naive version):**

$$33,554,432 + 56,623,104 + 62,914,560 = 153,092,096$$

Inception with Dimensionality Reduction:

- 1×1 convolutions with 128 filters:

$$32 \times 32 \times 128 \times (1 \times 1 \times 256) = 33,554,432$$

- 1×1 convolutions with 32 filters followed by 3×3 convolutions with 192 filters:

$$8,388,608 + 56,623,104 = 65,011,712$$

- 1×1 convolutions with 32 filters followed by 5×5 convolutions with 96 filters:

$$8,388,608 + 7,864,320 = 16,252,928$$

- 3×3 max pooling followed by 1×1 convolutions with 64 filters:

$$16,777,216$$

- **Total operations (dimensionality reduction version):**

$$33,554,432 + 65,011,712 + 16,252,928 + 16,777,216 = 131,596,288$$

Conclusion:

- **Naive Inception Module Total Operations:** 153,092,096
- **Inception with Dimensionality Reduction Total Operations:** 131,596,288

□

4.2-d

Proof. **Naive Architecture Problem:** The naive inception module applies large filters (e.g., 3×3 and 5×5) directly to the input with 256 channels, resulting in a total of 153,092,096 operations. This makes it computationally expensive due to the high number of convolutions on the full-depth input.
Dimensionality Reduction Solution: The dimensionality reduction version uses 1×1 convolutions to reduce the input depth before applying larger filters, lowering the computational cost to 131,596,288 operations.
Computational Saving: The computational saving is calculated as:

$$\text{Savings} = \frac{153,092,096 - 131,596,288}{153,092,096} \times 100 \approx 14\%$$

Conclusion: By reducing the input depth, the dimensionality reduction architecture saves approximately 14% of the operations compared to the naive approach, while still capturing multi-scale features.

□

5

Proof. **Staleness Calculation:**
Given: - Learner 1 runs at 2.5x the speed of Learner 2. - Learner 1 computes gradients at seconds 1, 2, 3, and 4. - Learner 2 computes gradients at seconds 2.5 and 5.
We calculate the staleness (number of weight updates between gradient computation and update) as follows:

- **Staleness of** $g[L1, 1]$ (computed at second 1):
  No updates before it is applied.
  Staleness = 0
- **Staleness of** $g[L1, 2]$ (computed at second 2):
  No updates before it is applied.
  Staleness = 0
- **Staleness of** $g[L1, 3]$ (computed at second 3):
  Learner 2 computes $g[L2, 1]$ at second 2.5 (one update before it is applied).
  Staleness = 1
- **Staleness of** $g[L1, 4]$ (computed at second 4):
  No updates before it is applied.
  Staleness = 0
- **Staleness of** $g[L2, 1]$ (computed at second 2.5):
  Learner 1 updates weights at seconds 1 and 2 (two updates before it is applied).
  Staleness = 2

- **Staleness of** $g[L2, 2]$ (computed at second 5):
  Learner 1 updates weights at seconds 1, 2, 3, and 4 (four updates before it is applied).
  Staleness = 4

Final Staleness Values:

$$g[L1, 1] = 0, \quad g[L1, 2] = 0, \quad g[L1, 3] = 1, \quad g[L1, 4] = 0$$

$$g[L2, 1] = 2, \quad g[L2, 2] = 4$$

□