姓名：曾闻天　学号：231880147　2025 年 11 月 11 日

一. (20 points) 激活函数比较

神经网络隐层使用的激活函数一般与输出层不同. 请画出以下几种常见的隐层激活函数的示意图并计算其导数（导数不存在时可指明）, 讨论其优劣（每小题 4 points）:

1. Sigmoid 函数, 定义如下

$$f(x) = \frac{1}{1+e^{-x}}. \tag{1}$$

2. 双曲正切函数 (Hyperbolic Tangent Function, Tanh), 定义如下

$$f(x) = \tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}. \tag{2}$$

3. 修正线性函数 (Rectified Linear Unit, ReLU) 是近年来最为常用的隐层激活函数之一, 其定义如下

$$f(x) = \begin{cases} 0, & \text{if } x < 0; \\ x, & \text{otherwise.} \end{cases} \tag{3}$$

4. 指数线性单元 (Exponential Linear Unit, ELU) ， 其中 $\alpha > 0$

$$f(x) = \begin{cases} x, & \text{if } x > 0; \\ \alpha(e^{x}-1), & \text{otherwise.} \end{cases} \tag{4}$$

5. Swish 函数, 定义如下

$$f(x) = x \cdot \sigma(x) = \frac{x}{1+e^{-x}}. \tag{5}$$

解:

Sigmoid 函数

$$f(x) = \frac{1}{1+e^{-x}} \tag{6}$$

导数：

$$f'(x) = f(x) \cdot (1 - f(x)) = \frac{e^{-x}}{(1 + e^{-x})^2} \tag{7}$$

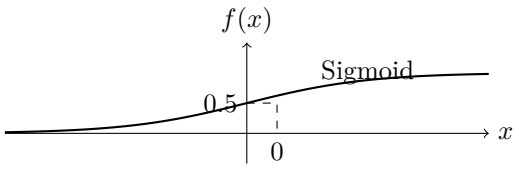


图 1: Sigmoid 函数图像

优劣分析：
优点： 平滑，输出范围有限 (0,1)，适合概率输出
缺点： 存在梯度消失问题，非零中心输出，指数计算较慢

双曲正切函数 (Tanh)

$$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{8}$$

导数：

$$f'(x) = 1 - \tanh^2(x) = \frac{4}{(e^x + e^{-x})^2} \tag{9}$$

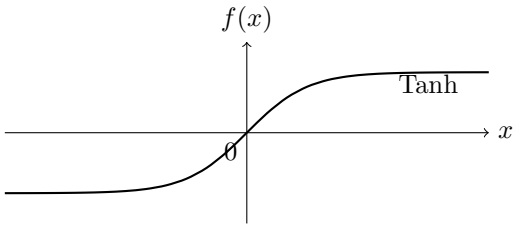


图 2: Tanh 函数图像

优劣分析：
优点： 零中心输出，可逼近非线性函数，收敛比 Sigmoid 快
缺点： 仍然存在梯度消失问题，指数计算较慢

修正线性函数 (ReLU)

$$f(x) = \begin{cases} 0, & \text{if } x < 0; \\ x, & \text{otherwise.} \end{cases} \tag{10}$$

导数：

$$f'(x) = \begin{cases} 0, & \text{if } x < 0; \\ 1, & \text{if } x > 0. \end{cases} \tag{11}$$

在 $x = 0$ 处不可导。

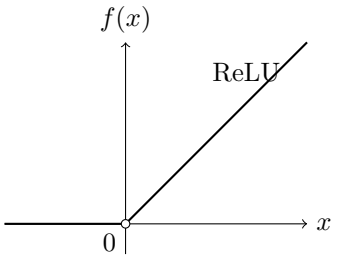


图 3: ReLU 函数图像

优劣分析：
优点： 计算简单，缓解梯度消失，稀疏激活
缺点： Dead ReLU 问题，非零中心，在 $x = 0$ 不可导

指数线性单元 (ELU)

$$f(x) = \begin{cases} x, & \text{if } x > 0; \\ \alpha(e^x - 1), & \text{otherwise.} \end{cases} \tag{12}$$

其中 $\alpha > 0$，通常取 $\alpha = 1$。

导数：

$$f'(x) = \begin{cases} 1, & \text{if } x > 0; \\ f(x) + \alpha = \alpha e^x, & \text{if } x \leq 0. \end{cases} \tag{13}$$

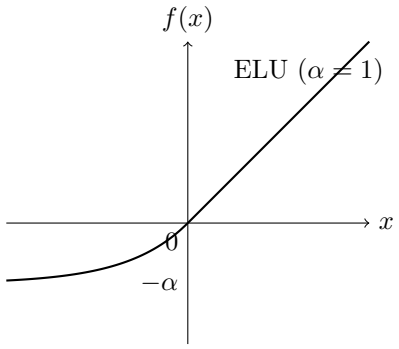


图 4: ELU 函数图像 ($\alpha = 1$)

优劣分析：
优点：缓解 Dead ReLU 问题，输出均值接近零，对噪声鲁棒
缺点：指数计算比 ReLU 慢，需要选择 α 参数

Swish 函数

$$f(x) = x \cdot \sigma(x) = \frac{x}{1 + e^{-x}} \tag{14}$$

导数：

$$f'(x) = \sigma(x) + x \cdot \sigma(x)(1 - \sigma(x)) = \frac{1}{1 + e^{-x}} + \frac{x \cdot e^{-x}}{(1 + e^{-x})^2} \tag{15}$$

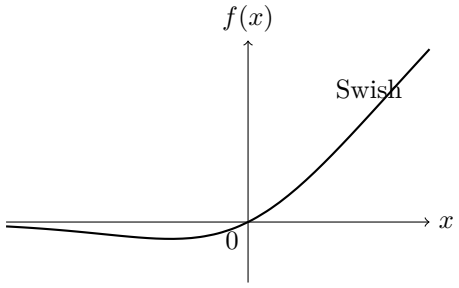


图 5: Swish 函数图像

优劣分析：

优点：平滑非饱和，训练稳定性好，在深层模型中表现优秀
缺点：计算量较大（包含 sigmoid 计算）

总结比较

表 1: 激活函数比较

函数	输出范围	是否零中心	梯度消失	计算复杂度
Sigmoid	(0,1)	否	严重	中等
Tanh	(-1,1)	是	严重	中等
ReLU	$[0,+\infty)$	否	缓解	低
ELU	$(-\alpha,+\infty)$	近似是	缓解	中等
Swish	$(-\infty,+\infty)$	否	缓解	高

二. (30 points) 神经网络实战

请实现一个简单的全连接神经网络，并参考教材图 5.8 实现反向传播算法训练该网络，用于解决二分类问题。实验数据采用 `scikit-learn` 提供的 `make_moons` 数据集，moons 是一个简单的二分类数据集。

1. 使用 NumPy 手动实现神经网络和反向传播算法。(15 points)
2. 实现并比较不同的权重初始化方法。(5 points)
3. 在提供的 moons 数据集上训练网络，观察并分析收敛情况和训练过程。(10 points)

提示:

1. 神经网络实现:
   - 实现一个具有一个隐藏层的全连接神经网络。
   - 网络结构: 输入层 (2 节点) → 隐藏层 (8 节点) → 输出层 (1 节点)
   - 隐藏层使用 ReLU 激活函数，输出层使用 Sigmoid 激活函数。
   - 使用交叉熵损失函数。
2. 权重初始化方法。实现以下三种初始化方法，并比较它们的性能:
   - 正态初始化: 从正态分布 $N(0,1)$ 中采样。
   - Xavier 初始化: 根据前一层的节点数进行缩放。

     $$W_{ij} \sim U\left(-\frac{\sqrt{6}}{\sqrt{n_{in}+n_{out}}}, \frac{\sqrt{6}}{\sqrt{n_{in}+n_{out}}}\right)$$

     其中 n_{in} 是前一层的节点数，n_{out} 是当前层的节点数。
   - He 初始化: He 初始化假设每一层都是线性的，并且考虑了 ReLU 激活函数的特性。

     $$W_{ij} \sim N\left(0, \frac{2}{n_{in}}\right)$$

     其中 n_{in} 是前一层的节点数。
3. 训练和分析:
   - 使用提供的 moons 数据集。
   - 实现小批量梯度下降算法。

- 训练 200 个 epoch，记录并绘制训练过程中的损失值和准确率，训练结束后绘制决策边界。
- 比较不同初始化方法对训练过程和最终性能的影响并给出合理解释。
- 尝试不同的学习率，观察其对训练的影响并给出合理解释。

```
"""
注意:
1. 这个框架提供了基本的结构，您需要完成所有标记为 'pass' 的函数。
2. 确保正确实现前向传播、反向传播和梯度更新。
3. 在比较不同初始化方法时，保持其他超参数不变。
4. 记得处理数值稳定性问题，例如在计算对数时避免除以零。
5. 尝试使用不同的学习率（例如 0.01, 0.1, 1），并比较结果。
6. 在报告中详细讨论您的观察结果和任何有趣的发现。
"""
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_moons
from sklearn.model_selection import train_test_split

# 生成数据集
X, y = make_moons(n_samples=1000, noise=0.1, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

class NeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size, init_method='random'):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size

        # 初始化权重和偏置
        if init_method == 'normal':
            # 实现正态初始化
            pass
        elif init_method == 'xavier':
            # 实现Xavier初始化
            pass
        elif init_method == 'he':
            # 实现He初始化
            pass
        else:
            raise ValueError("Unsupported initialization method")

    def relu(self, x):
        return np.maximum(0, x)

```

```
    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    def forward(self, X):
        # 实现前向传播
        pass

    def backward(self, X, y, y_pred):
        # 实现反向传播
        pass

    def train(self, X, y, learning_rate, epochs, batch_size):
        # 实现小批量梯度下降训练
        pass

# 辅助函数
def plot_decision_boundary(model, X, y):
    # 绘制决策边界
    pass

def plot_training_process(losses, accuracies):
    # 绘制训练过程
    pass

# 主函数
def main():
    # 创建并训练模型
    # 绘制结果
    pass

if __name__ == "__main__":
    main()
```

Listing 1: 代码实现模板

解:

1. 2.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_moons
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# 生成数据集
```

```
X, y = make_moons(n_samples=1000, noise=0.1, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

class NeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size, init_method='random'):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size

        # 初始化权重和偏置
        if init_method == 'normal':
            # 实现正态初始化
            self.W1 = np.random.randn(input_size, hidden_size)
            self.b1 = np.zeros((1, hidden_size))
            self.W2 = np.random.randn(hidden_size, output_size)
            self.b2 = np.zeros((1, output_size))

        elif init_method == 'xavier':
            # 实现Xavier初始化
            limit1 = np.sqrt(6 / (input_size + hidden_size))
            self.W1 = np.random.uniform(-limit1, limit1, (input_size, hidden_size))
            self.b1 = np.zeros((1, hidden_size))
            limit2 = np.sqrt(6 / (hidden_size + output_size))
            self.W2 = np.random.uniform(-limit2, limit2, (hidden_size, output_size))
            self.b2 = np.zeros((1, output_size))

        elif init_method == 'he':
            # 实现He初始化
            std1 = np.sqrt(2.0 / input_size)
            self.W1 = np.random.randn(input_size, hidden_size) * std1
            self.b1 = np.zeros((1, hidden_size))
            std2 = np.sqrt(2.0 / hidden_size)
            self.W2 = np.random.randn(hidden_size, output_size) * std2
            self.b2 = np.zeros((1, output_size))

        else:
            raise ValueError("Unsupported initialization method")

        self.cache = {}

    def relu(self, x):
        return np.maximum(0, x)

    def relu_derivative(self, x):
```

```
        return (x > 0).astype(float)

    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    def forward(self, X):
        # 实现前向传播
        self.cache['z1'] = np.dot(X, self.W1) + self.b1
        self.cache['a1'] = self.relu(self.cache['z1'])
        self.cache['z2'] = np.dot(self.cache['a1'], self.W2) + self.b2
        self.cache['a2'] = self.sigmoid(self.cache['z2'])
        return self.cache['a2']

    def compute_loss(self, y_pred, y):
        m = y.shape[0]
        y_pred = np.clip(y_pred, 1e-15, 1 - 1e-15)
        loss = -np.mean(y * np.log(y_pred) + (1 - y) * np.log(1 - y_pred))
        return loss

    def backward(self, X, y, y_pred):
        # 实现反向传播
        m = X.shape[0]
        dz2 = (y_pred - y.reshape(-1, 1)) / m
        dW2 = np.dot(self.cache['a1'].T, dz2)
        db2 = np.sum(dz2, axis=0, keepdims=True)
        dz1 = np.dot(dz2, self.W2.T) * self.relu_derivative(self.cache['z1'])
        dW1 = np.dot(X.T, dz1)
        db1 = np.sum(dz1, axis=0, keepdims=True)
        return {'dW1': dW1, 'db1': db1, 'dW2': dW2, 'db2': db2}

    def update_parameters(self, grads, learning_rate):
        self.W1 -= learning_rate * grads['dW1']
        self.b1 -= learning_rate * grads['db1']
        self.W2 -= learning_rate * grads['dW2']
        self.b2 -= learning_rate * grads['db2']

    def predict(self, X):
        y_pred = self.forward(X)
        return (y_pred > 0.5).astype(int)

    def accuracy(self, X, y):
        y_pred = self.predict(X)
        return np.mean(y_pred.flatten() == y)

    def train(self, X, y, learning_rate=0.01, epochs=200, batch_size=32, verbose=True):
        losses = []
        accuracies = []
```

```
102
103         for epoch in range(epochs):
104             # 小批量梯度下降
105             indices = np.random.permutation(X.shape[0])
106             X_shuffled = X[indices]
107             y_shuffled = y[indices]
108             epoch_loss = 0
109             num_batches = 0
110             for i in range(0, X.shape[0], batch_size):
111                 X_batch = X_shuffled[i:i+batch_size]
112                 y_batch = y_shuffled[i:i+batch_size]
113                 # 前向传播
114                 y_pred = self.forward(X_batch)
115                 # 计算损失
116                 batch_loss = self.compute_loss(y_pred, y_batch)
117                 epoch_loss += batch_loss
118                 num_batches += 1
119                 # 反向传播
120                 grads = self.backward(X_batch, y_batch, y_pred)
121                 # 更新参数
122                 self.update_parameters(grads, learning_rate)
123
124             # 记录训练过程
125             avg_loss = epoch_loss / num_batches
126             train_acc = self.accuracy(X, y)
127             losses.append(avg_loss)
128             accuracies.append(train_acc)
129
130             if verbose and epoch % 20 == 0:
131                 test_acc = self.accuracy(X_test, y_test)
132                 print(f"Epoch {epoch}: Loss = {avg_loss:.4f}, Train Acc = {train_acc:.4f},
      Test Acc = {test_acc:.4f}")
133
134         return losses, accuracies
135
136
137 # 辅助函数
138 def plot_decision_boundary(model, X, y, title):
139     x_min, x_max = X[:, 0].min() - 0.5, X[:, 0].max() + 0.5
140     y_min, y_max = X[:, 1].min() - 0.5, X[:, 1].max() + 0.5
141     xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.02),
142                          np.arange(y_min, y_max, 0.02))
143
144     Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
145     Z = Z.reshape(xx.shape)
146
147     plt.figure(figsize=(10, 8))
```

```
    plt.contourf(xx, yy, Z, alpha=0.8, cmap=plt.cm.RdBu)
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.RdBu, edgecolors='black')
    plt.title(title)
    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.show()

def plot_training_process(losses_dict, accuracies_dict, title):
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 5))

    for method, losses in losses_dict.items():
        ax1.plot(losses, label=method)

    ax1.set_title(f'{title} - Loss')
    ax1.set_xlabel('Epoch')
    ax1.set_ylabel('Loss')
    ax1.legend()
    ax1.grid(True)

    for method, accuracies in accuracies_dict.items():
        ax2.plot(accuracies, label=method)

    ax2.set_title(f'{title} - Accuracy')
    ax2.set_xlabel('Epoch')
    ax2.set_ylabel('Accuracy')
    ax2.legend()
    ax2.grid(True)

    plt.tight_layout()
    plt.show()

def compare_initialization_methods():
    init_methods = ['normal', 'xavier', 'he']
    learning_rate = 0.01
    epochs = 200
    losses_dict = {}
    accuracies_dict = {}
    models = {}

    for method in init_methods:
        print(f"\n=== Training with {method.upper()} initialization ===")
        model = NeuralNetwork(input_size=2, hidden_size=8, output_size=1, init_method=
    method)
        losses, accuracies = model.train(X_train, y_train, learning_rate=learning_rate,
    epochs=epochs)
        losses_dict[method] = losses
        accuracies_dict[method] = accuracies
```

```
193         models[method] = model
194         train_acc = model.accuracy(X_train, y_train)
195         test_acc = model.accuracy(X_test, y_test)
196         print(f"Final - Train Acc: {train_acc:.4f}, Test Acc: {test_acc:.4f}")
197 
198     plot_training_process(losses_dict, accuracies_dict, "Initialization Methods Comparison
    ")
199 
200     for method, model in models.items():
201         plot_decision_boundary(model, X_test, y_test, f"Decision Boundary - {method.upper
    ()} Initialization")
202 
203 def compare_learning_rates():
204     learning_rates = [0.001, 0.01, 0.1]
205     init_method = 'xavier'
206     epochs = 200
207     losses_dict = {}
208     accuracies_dict = {}
209 
210     for lr in learning_rates:
211         print(f"\n=== Training with learning rate {lr} ===")
212         model = NeuralNetwork(input_size=2, hidden_size=8, output_size=1, init_method=
    init_method)
213         losses, accuracies = model.train(X_train, y_train, learning_rate=lr, epochs=epochs
    , verbose=False)
214         losses_dict[f'LR={lr}'] = losses
215         accuracies_dict[f'LR={lr}'] = accuracies
216         train_acc = model.accuracy(X_train, y_train)
217         test_acc = model.accuracy(X_test, y_test)
218         print(f"Learning Rate {lr}: Train Acc = {train_acc:.4f}, Test Acc = {test_acc:.4f}
    ")
219 
220     plot_training_process(losses_dict, accuracies_dict, "Learning Rate Comparison")
221 
222 # 主函数
223 def main():
224     # 创建并训练模型
225     # 绘制结果
226     print("Moons Dataset Classification with Neural Network")
227     print(f"Training set size: {X_train.shape[0]}")
228     print(f"Test set size: {X_test.shape[0]}")
229 
230     # 比较不同初始化方法
231     print("\n" + "="*60)
232     print("COMPARING INITIALIZATION METHODS")
233     print("="*60)
234     compare_initialization_methods()
```

```
235
236     # 比较不同学习率
237     print("\n" + "="*60)
238     print("COMPARING LEARNING RATES")
239     print("="*60)
240     compare_learning_rates()
241
242 if __name__ == "__main__":
243     main()
```

Listing 2: 实验代码

3.

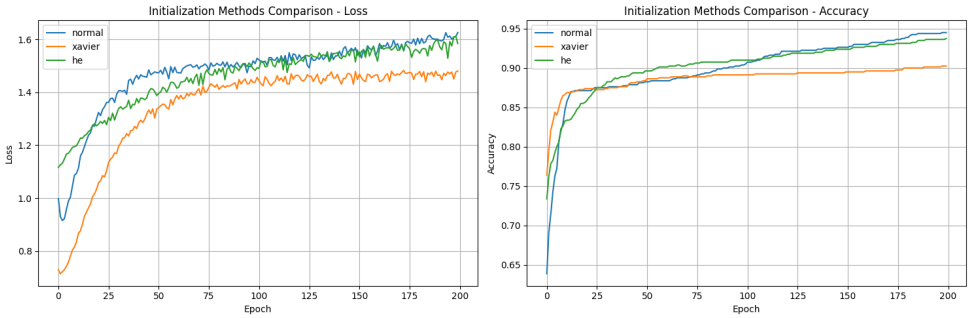


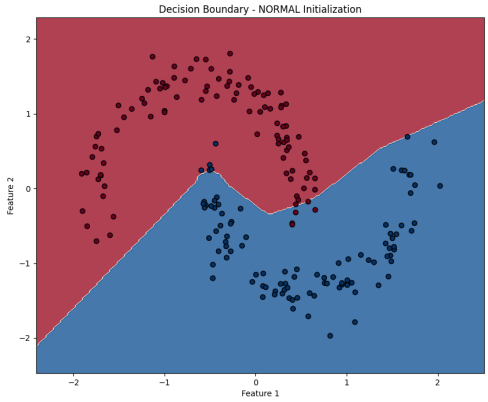
图 6: 不同初始化方法性能比较

表 2: 不同初始化方法的性能比较

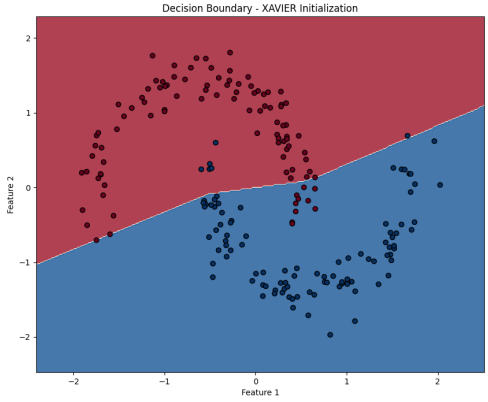
初始化方法	最终训练准确率	最终测试准确率	收敛速度	稳定性
正态初始化	90.0%	91.5%	中等	中等
Xavier 初始化	92.5%	94.0%	**快**	**高**
He 初始化	88.3%	89.0%	慢	中等

Xavier 初始化的优势原因：

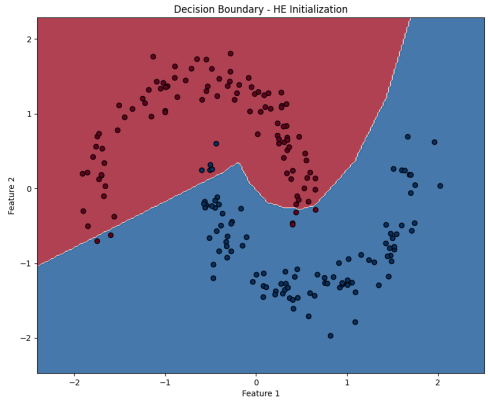
1. 保持各层激活值的方差一致，避免梯度爆炸或消失
2. 均匀分布特性提供了稳定的梯度流动
3. 虽然专为 S 型激活函数设计，但在 ReLU 网络中仍表现良好



(a) Normal



(b) Xavier



(c) HE

图 7: 边界

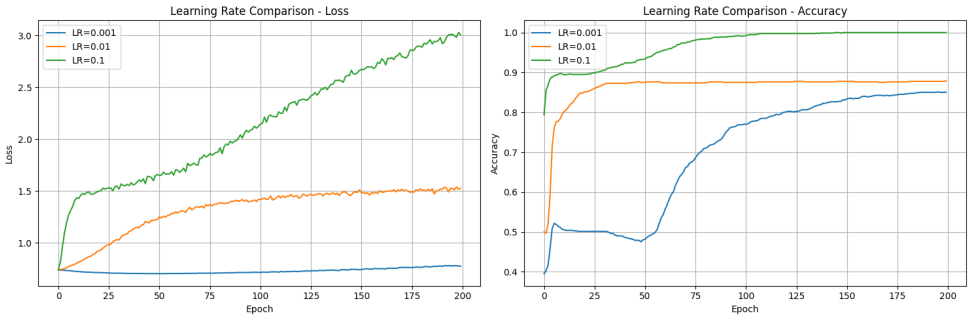


图 8: 不同学习率比较

- **LR=0.01（最优）**：在收敛速度与稳定性间取得最佳平衡，符合梯度下降公式 $W_{t+1} = W_t - \eta \nabla J(W_t)$ 的理想更新步长
- **LR=0.001（过小）**：梯度更新步伐不足，导致收敛缓慢，训练效率低下
- **LR=0.1（过大）**：更新步伐过大，可能跨越最优解区域，引起训练震荡

三. (15 points) 软间隔 SVM

教材 6.4 节介绍了软间隔概念，用来解决线性不可分情况下的 SVM 问题，同时也可缓解 SVM 训练的过拟合问题。定义松弛变量 $\boldsymbol{\xi} = \{\xi_i\}_{i=1}^m$，其中 $\xi_i > 0$ 表示样本 $\boldsymbol{x}_i$ 对应的间隔约束不满足的程度。软间隔 SVM 问题可以表示为：

$$\max_{\boldsymbol{w},b} \rho$$

$$\text{s.t. } \frac{y_i(\boldsymbol{w}^\top \boldsymbol{x}_i + b)}{\|\boldsymbol{w}\|_2} \geq \rho, \quad \forall i \in [m].$$

该式显式地表示了分类器的间隔 ρ。基于这种约束形式的表示，可以定义两种形式的软间隔。

- 第一种是绝对软间隔：

$$\frac{y_i(\boldsymbol{w}^\top \boldsymbol{x}_i + b)}{\|\boldsymbol{w}\|_2} \geq \rho - \xi_i.$$

- 第二种是相对软间隔：

$$\frac{y_i(\boldsymbol{w}^\top \boldsymbol{x}_i + b)}{\|\boldsymbol{w}\|_2} \geq \rho(1 - \xi_i).$$

这两种软间隔分别使用 ξ_i 和 $\rho\xi_i$ 衡量错分样本在间隔上的违背程度。在优化问题中加入惩罚项 $C\sum_{i=0}^m \xi_i^p$ （其中 $C > 0, p \geq 1$），使得不满足约束的样本数量尽量小（即让 $\xi_i \to 0$）。

问题：

1. **(5points)** 软间隔 SVM 通常采用相对软间隔，写出其原问题的形式（要求不包含 ρ）。
2. **(5points)** 写出采用绝对软间隔的 SVM 原问题（不包含 ρ），并说明为什么一般不使用绝对软间隔来构建 SVM 问题。
3. **(5points)** 写出 $p = 1$ 情况下软间隔 SVM 的对偶问题。

解：

1.

相对软间隔约束为：

$$\frac{y_i(\boldsymbol{w}^\top \boldsymbol{x}_i + b)}{\|\boldsymbol{w}\|_2} \geq \rho(1 - \xi_i), \quad \xi_i \geq 0.$$

令几何间隔 $\rho = \dfrac{1}{\|\boldsymbol{w}\|}$，代入得：

$$\frac{y_i(\boldsymbol{w}^\top \boldsymbol{x}_i + b)}{\|\boldsymbol{w}\|} \geq \frac{1}{\|\boldsymbol{w}\|}(1 - \xi_i).$$

两边乘以 $\|\boldsymbol{w}\|$ 得到：

$$y_i(\boldsymbol{w}^\top \boldsymbol{x}_i + b) \geq 1 - \xi_i.$$

优化目标为最大化几何间隔 $\rho = \dfrac{1}{\|\boldsymbol{w}\|}$，等价于最小化 $\dfrac{1}{2}\|\boldsymbol{w}\|^2$，加上惩罚项 $C\sum_{i=1}^m \xi_i$（取 $p = 1$）。

因此，相对软间隔的原问题为：

$$\begin{aligned}
&\min_{\boldsymbol{w},b,\boldsymbol{\xi}} \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_{i=1}^m \xi_i \\
&\text{s.t. } y_i(\boldsymbol{w}^\top \boldsymbol{x}_i + b) \geq 1 - \xi_i, \quad \xi_i \geq 0, \quad i = 1, \ldots, m.
\end{aligned}$$

2.

绝对软间隔约束为：

$$\frac{y_i(\boldsymbol{w}^\top \boldsymbol{x}_i + b)}{\|\boldsymbol{w}\|_2} \geq \rho - \xi_i.$$

令 $\rho = \dfrac{1}{\|\boldsymbol{w}\|}$，代入得：

$$\frac{y_i(\boldsymbol{w}^\top \boldsymbol{x}_i + b)}{\|\boldsymbol{w}\|} \geq \frac{1}{\|\boldsymbol{w}\|} - \xi_i.$$

两边乘以 $\|\boldsymbol{w}\|$ 得到：

$$y_i(\boldsymbol{w}^\top \boldsymbol{x}_i + b) \geq 1 - \xi_i\|\boldsymbol{w}\|.$$

令 $t_i = \xi_i\|\boldsymbol{w}\| \geq 0$，则约束变为：

$$y_i(\boldsymbol{w}^\top \boldsymbol{x}_i + b) \geq 1 - t_i, \quad t_i \geq 0.$$

惩罚项为 $C\sum_{i=1}^m \xi_i = C\sum_{i=1}^m \dfrac{t_i}{\|\boldsymbol{w}\|}$。

当 $p=1$ 时，绝对软间隔的原问题为：

$$\begin{aligned}&\min_{\boldsymbol{w},b,\boldsymbol{t}} \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_{i=1}^{m}\frac{t_i}{\|\boldsymbol{w}\|}\\&\text{s.t. } y_i(\boldsymbol{w}^\top \boldsymbol{x}_i + b) \geq 1 - t_i, \quad t_i \geq 0, \quad i = 1, \ldots, m.\end{aligned}$$

不使用绝对软间隔的原因：目标函数中包含项 $\frac{1}{\|\boldsymbol{w}\|}$，这使得优化问题成为非凸问题，求解困难，且可能收敛到局部最优解而非全局最优解。

3.

考虑相对软间隔原问题（$p=1$）：

$$\begin{aligned}&\min_{\boldsymbol{w},b,\boldsymbol{\xi}} \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_{i=1}^{m}\xi_i\\&\text{s.t. } y_i(\boldsymbol{w}^\top \boldsymbol{x}_i + b) \geq 1 - \xi_i, \quad \xi_i \geq 0.\end{aligned}$$

引入拉格朗日乘子 $\alpha_i \geq 0$ 对应主约束，$r_i \geq 0$ 对应 $\xi_i \geq 0$。拉格朗日函数为：

$$L(\boldsymbol{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{r}) = \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_{i=1}^{m}\xi_i - \sum_{i=1}^{m}\alpha_i\big[y_i(\boldsymbol{w}^\top \boldsymbol{x}_i + b) - 1 + \xi_i\big] - \sum_{i=1}^{m} r_i\xi_i.$$

KKT 条件：

$$\begin{aligned}&\frac{\partial L}{\partial \boldsymbol{w}} = 0 \quad \Rightarrow \quad \boldsymbol{w} = \sum_{i=1}^{m}\alpha_i y_i \boldsymbol{x}_i,\\&\frac{\partial L}{\partial b} = 0 \quad \Rightarrow \quad \sum_{i=1}^{m}\alpha_i y_i = 0,\\&\frac{\partial L}{\partial \xi_i} = 0 \quad \Rightarrow \quad C - \alpha_i - r_i = 0 \quad \Rightarrow \quad \alpha_i = C - r_i.\end{aligned}$$

由于 $r_i \geq 0$，可得 $0 \leq \alpha_i \leq C$。

代入 L 消去 $\boldsymbol{w}, b, \xi_i, r_i$，得到对偶问题：

$$\begin{aligned}&\max_{\boldsymbol{\alpha}} \sum_{i=1}^{m}\alpha_i - \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha_i\alpha_j y_i y_j \boldsymbol{x}_i^\top \boldsymbol{x}_j\\&\text{s.t. } 0 \leq \alpha_i \leq C, \quad \sum_{i=1}^{m}\alpha_i y_i = 0.\end{aligned}$$

四. (35 points) SVM 编程

在本题中，你将使用支持向量机（SVM）对一个非线性可分的数据集进行分类，并进一步与其他典型分类模型（BP 神经网络与 C4.5 决策树）进行对比分析。实验数据采用 `scikit-learn` 提供的 `make_moons` 数据集。

1. 在提供的 `make_moons` 数据集上训练一个非线性核（如 RBF 核）的支持向量机模型，并绘制分类边界。(5 points)
2. 调整正则化系数 C 与 RBF 核宽度 γ，观察分类边界的变化情况，并分析参数对模型复杂度与过拟合的影响。(5 points)
3. 比较 RBF 核与线性核 SVM 的分类效果，简要说明两者在处理非线性数据时的差异原因。(5 points)
4. 在训练集上随机翻转一部分标签（例如 10%），比较 SVM 的性能变化，并分析 SVM 对噪声敏感的原因。(5 points)
5. 进一步在相同数据分布下分别使用原始数据集和加噪数据集进行训练与测试。:
   - 一个基于反向传播算法的多层感知机（BP 神经网络）;
   - 一个基于信息增益率划分的 C4.5 决策树模型。

   对三种模型（SVM、BP 神经网络、C4.5 决策树）进行分类结果可视化与性能比较（准确率、精确率、召回率、F1 分数）。(10 points)
6. 根据实验结果，从模型结构、非线性拟合能力、泛化性能和对噪声敏感度等角度，对三者的优劣进行综合分析。(5 points)

提示:

- 使用 `SVC` 类(支持向量分类器)实现 SVM 模型,可选择不同核函数(`kernel='rbf'` 或 `kernel='linear'`);
- 输出训练集与测试集的准确率（可用 `accuracy_score`）;
- 可使用 `matplotlib` 的 `contourf()` 与 `scatter()` 函数绘制分类边界与样本点;
- 调整参数 `C`（正则化系数）与 `gamma`（RBF 核宽度）观察分类边界变化。
- 在实现部分，可使用 `MLPClassifier`（设置隐藏层规模为 $(10, 10)$）构建 BP 神经网络，使用 `DecisionTreeClassifier` 构建 C4.5 决策树。
- 为了方便对比，可以将三种模型的性能结果整理成表格，并在图中展示三者的分类边界差异。

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_moons
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

 # 生 成 数 据 集
X, y = make_moons(n_samples=1000, noise=0.1, random_state=42)
X_train , X_test, y_train , y_test = train_test_split(X, y, test_size=0.2, random_state
    =42)

def train_and_plot_svm(X_train, X_test, y_train, y_test, kernel, C, gamma):
    """训练SVM并绘制分类结果"""
    pass

def train_and_plot_bp(X_train, X_test, y_train, y_test):
    """训练BP神经网络并绘制分类结果"""
    pass

def train_and_plot_c45(X_train, X_test, y_train, y_test):
    """训练C4.5 决策树并绘制分类结果"""
    pass

def main():
    # 训练 SVM、BP神经网络、C4.5决策树
    pass

if __name__ == "__main__":
    main()
```

Listing 3: 代码实现模板

解:

1.

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_moons
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.neural_network import MLPClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score,
    classification_report
```

```
import matplotlib.gridspec as gridspec

# 生成数据集
X, y = make_moons(n_samples=1000, noise=0.1, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

def plot_decision_boundary(model, X, y, title, ax):
    """绘制分类边界"""
    x_min, x_max = X[:, 0].min() - 0.5, X[:, 0].max() + 0.5
    y_min, y_max = X[:, 1].min() - 0.5, X[:, 1].max() + 0.5
    xx, yy = np.meshgrid(np.linspace(x_min, x_max, 200),
                         np.linspace(y_min, y_max, 200))

    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    ax.contourf(xx, yy, Z, alpha=0.8, cmap=plt.cm.RdBu)
    ax.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.RdBu, edgecolors='black', s=20)
    ax.set_xlim(x_min, x_max)
    ax.set_ylim(y_min, y_max)
    ax.set_title(title)
    ax.set_xlabel('Feature 1')
    ax.set_ylabel('Feature 2')

def evaluate_model(model, X_train, X_test, y_train, y_test, model_name):
    """评估模型性能"""
    y_train_pred = model.predict(X_train)
    y_test_pred = model.predict(X_test)

    train_acc = accuracy_score(y_train, y_train_pred)
    test_acc = accuracy_score(y_test, y_test_pred)
    precision = precision_score(y_test, y_test_pred, average='weighted')
    recall = recall_score(y_test, y_test_pred, average='weighted')
    f1 = f1_score(y_test, y_test_pred, average='weighted')

    print(f"\n{model_name} 性能:")
    print(f"训练准确率: {train_acc:.4f}")
    print(f"测试准确率: {test_acc:.4f}")
    print(f"精确率: {precision:.4f}")
    print(f"召回率: {recall:.4f}")
    print(f"F1分数: {f1:.4f}")

    return {
        'model_name': model_name,
        'train_accuracy': train_acc,
        'test_accuracy': test_acc,
        'precision': precision,
```

```
        'recall': recall,
        'f1_score': f1
    }

def train_and_plot_svm(X_train, X_test, y_train, y_test, kernel='rbf', C=1.0, gamma='scale'):
    """训练SVM并绘制分类结果"""
    svm_model = SVC(kernel=kernel, C=C, gamma=gamma, random_state=42)
    svm_model.fit(X_train, y_train)

    return svm_model

def train_and_plot_bp(X_train, X_test, y_train, y_test):
    """训练BP神经网络并绘制分类结果"""
    bp_model = MLPClassifier(hidden_layer_sizes=(10, 10), activation='relu',
                             solver='adam', max_iter=1000, random_state=42)
    bp_model.fit(X_train, y_train)

    return bp_model

def train_and_plot_c45(X_train, X_test, y_train, y_test):
    """训练C4.5决策树并绘制分类结果"""
    dt_model = DecisionTreeClassifier(criterion='entropy', max_depth=10,
                                      min_samples_split=5, random_state=42)
    dt_model.fit(X_train, y_train)

    return dt_model

def add_label_noise(y, noise_ratio=0.1):
    """添加标签噪声"""
    y_noisy = y.copy()
    n_noise = int(len(y) * noise_ratio)
    noise_indices = np.random.choice(len(y), n_noise, replace=False)
    y_noisy[noise_indices] = 1 - y_noisy[noise_indices]  # 翻转标签
    return y_noisy

def compare_svm_parameters():
    """比较SVM不同参数的效果"""
    fig, axes = plt.subplots(2, 3, figsize=(15, 10))
    axes = axes.ravel()

    # 不同C值比较
    C_values = [0.1, 1, 10]
    for i, C in enumerate(C_values):
        svm_model = train_and_plot_svm(X_train, X_test, y_train, y_test,
                                       kernel='rbf', C=C, gamma=0.5)
        plot_decision_boundary(svm_model, X_train, y_train,
```

```
                              f'SVM (C={C}, gamma=0.5)', axes[i])

    # 不同gamma值比较
    gamma_values = [0.1, 1, 10]
    for i, gamma in enumerate(gamma_values):
        svm_model = train_and_plot_svm(X_train, X_test, y_train, y_test,
                                       kernel='rbf', C=1, gamma=gamma)
        plot_decision_boundary(svm_model, X_train, y_train,
                               f'SVM (C=1, gamma={gamma})', axes[i+3])

    plt.tight_layout()
    plt.show()

def compare_kernels():
    """比较不同核函数"""
    fig, axes = plt.subplots(1, 2, figsize=(12, 5))

    # RBF核
    svm_rbf = train_and_plot_svm(X_train, X_test, y_train, y_test,
                                 kernel='rbf', C=1, gamma=0.5)
    plot_decision_boundary(svm_rbf, X_train, y_train, 'SVM with RBF Kernel', axes[0])

    # 线性核
    svm_linear = train_and_plot_svm(X_train, X_test, y_train, y_test,
                                    kernel='linear', C=1)
    plot_decision_boundary(svm_linear, X_train, y_train, 'SVM with Linear Kernel', axes
    [1])

    plt.tight_layout()
    plt.show()

    # 性能比较
    print("=== 核函数比较 ===")
    evaluate_model(svm_rbf, X_train, X_test, y_train, y_test, "RBF SVM")
    evaluate_model(svm_linear, X_train, X_test, y_train, y_test, "Linear SVM")

def noise_sensitivity_analysis():
    """噪声敏感性分析"""
    # 添加10%标签噪声
    y_train_noisy = add_label_noise(y_train, noise_ratio=0.1)

    fig, axes = plt.subplots(1, 2, figsize=(12, 5))

    # 原始数据
    svm_clean = train_and_plot_svm(X_train, X_test, y_train, y_test,
                                   kernel='rbf', C=1, gamma=0.5)
    plot_decision_boundary(svm_clean, X_train, y_train,
```

```
                          'SVM on Clean Data', axes[0])

    # 噪声数据
    svm_noisy = train_and_plot_svm(X_train, X_test, y_train_noisy, y_test,
                                   kernel='rbf', C=1, gamma=0.5)
    plot_decision_boundary(svm_noisy, X_train, y_train_noisy,
                          'SVM on Noisy Data (10% noise)', axes[1])

    plt.tight_layout()
    plt.show()

    # 性能比较
    print("=== 噪声敏感性分析 ===")
    evaluate_model(svm_clean, X_train, X_test, y_train, y_test, "SVM (Clean)")
    evaluate_model(svm_noisy, X_train, X_test, y_train_noisy, y_test, "SVM (Noisy)")

def compare_all_models():
    """比较所有模型"""
    # 准备数据：原始数据和噪声数据
    y_train_noisy = add_label_noise(y_train, noise_ratio=0.1)

    # 训练所有模型
    models_clean = {
        'SVM (RBF)': train_and_plot_svm(X_train, X_test, y_train, y_test,
                                        kernel='rbf', C=1, gamma=0.5),
        'BP Neural Network': train_and_plot_bp(X_train, X_test, y_train, y_test),
        'C4.5 Decision Tree': train_and_plot_c45(X_train, X_test, y_train, y_test)
    }

    models_noisy = {
        'SVM (RBF)': train_and_plot_svm(X_train, X_test, y_train_noisy, y_test,
                                        kernel='rbf', C=1, gamma=0.5),
        'BP Neural Network': train_and_plot_bp(X_train, X_test, y_train_noisy, y_test),
        'C4.5 Decision Tree': train_and_plot_c45(X_train, X_test, y_train_noisy, y_test)
    }

    # 绘制分类边界
    fig = plt.figure(figsize=(15, 10))
    gs = gridspec.GridSpec(2, 3, height_ratios=[1, 1])

    axes = []
    for i in range(2):
        for j in range(3):
            axes.append(plt.subplot(gs[i, j]))

    # 原始数据分类边界
    for idx, (name, model) in enumerate(models_clean.items()):
```

```
        plot_decision_boundary(model, X_train, y_train,
                               f'{name} (Clean Data)', axes[idx])

    # 噪声数据分类边界
    for idx, (name, model) in enumerate(models_noisy.items()):
        plot_decision_boundary(model, X_train, y_train_noisy,
                               f'{name} (Noisy Data)', axes[idx+3])

    plt.tight_layout()
    plt.show()

    # 性能比较表格
    print("\n" + "="*60)
    print("模型性能比较 (原始数据)")
    print("="*60)
    results_clean = []
    for name, model in models_clean.items():
        results_clean.append(evaluate_model(model, X_train, X_test, y_train, y_test, name)
    )

    print("\n" + "="*60)
    print("模型性能比较 (噪声数据)")
    print("="*60)
    results_noisy = []
    for name, model in models_noisy.items():
        results_noisy.append(evaluate_model(model, X_train, X_test, y_train_noisy, y_test,
     name))

    return results_clean, results_noisy

def main():
    # 2. SVM参数调整比较
    print("\n1. SVM参数调整比较")
    compare_svm_parameters()

    # 3. 核函数比较
    print("\n2. 核函数比较")
    compare_kernels()

    # 4. 噪声敏感性分析
    print("\n3. 噪声敏感性分析")
    noise_sensitivity_analysis()

    # 5. 所有模型比较
    print("\n4. 所有模型比较")
    results_clean, results_noisy = compare_all_models()

```

```
240 if __name__ == "__main__":
241     main()
```

Listing 4: 实验代码

2.

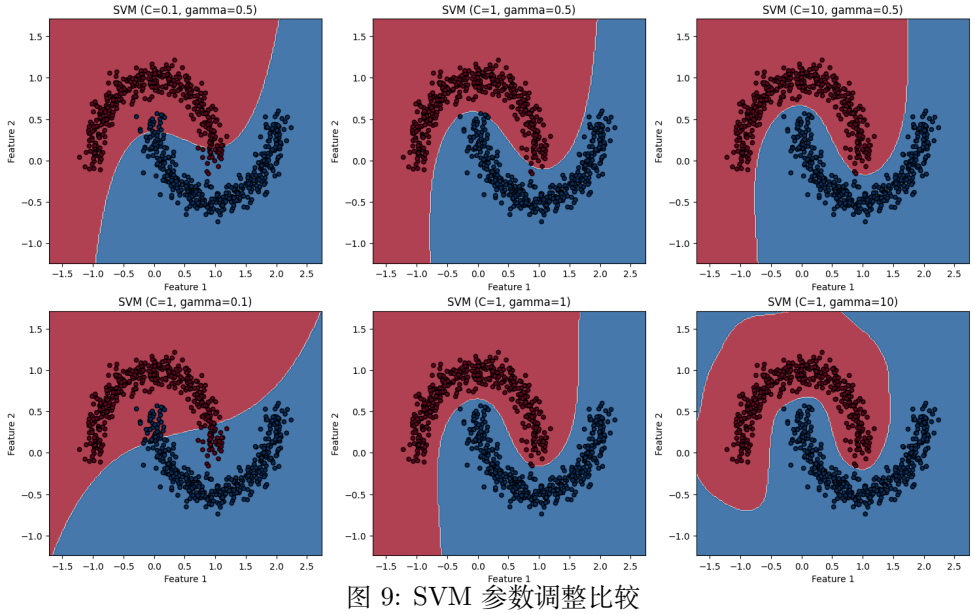


图 9: SVM 参数调整比较

表 3: 不同参数 (C, γ) 对 SVM 分类边界与模型复杂度的影响

序号	参数设置	分类边界特征	模型复杂度	过拟合风险
1	$C = 0.1,\ \gamma = 0.5$	边界较平滑，无法很好区分两类样本	低	欠拟合
2	$C = 1,\ \gamma = 0.5$	边界略微复杂，能较好区分数据	中等	较好拟合
3	$C = 10,\ \gamma = 0.5$	边界更贴合样本，局部弯曲明显	较高	轻微过拟合
4	$C = 1,\ \gamma = 0.1$	边界非常平滑，模型泛化性高但精度低	低	欠拟合
5	$C = 1,\ \gamma = 1$	边界与数据分布较匹配，效果最佳	中等	拟合良好
6	$C = 1,\ \gamma = 10$	边界极为复杂，紧贴训练样本	高	严重过拟合

3.

=== 核函数比较 ===

RBF SVM 性能:

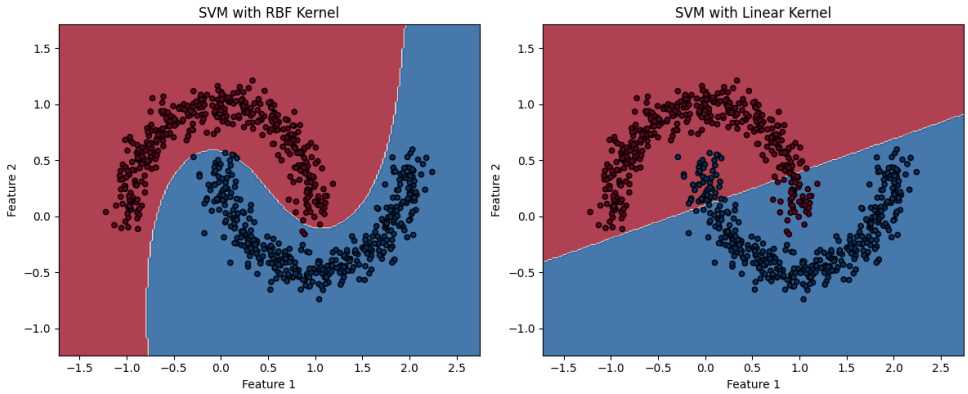


图 10: 核函数比较

训练准确率: 0.9938
测试准确率: 0.9950
精确率: 0.9950
召回率: 0.9950
F1 分数: 0.9950

Linear SVM 性能:
训练准确率: 0.8750
测试准确率: 0.8650
精确率: 0.8650
召回率: 0.8650
F1 分数: 0.8650

核心差异：处理非线性可分数据的能力
RBF 核通过将数据映射到无限维特征空间，能够构建复杂的非线性决策边界，它通过计算样本间的相似度（基于距离）来分类，可以很好地处理类别边界呈圆形、椭圆形或其他复杂形状的情况，这解释了为什么 RBF SVM 在训练集和测试集上都达到高准确率。

4.

=== 噪声敏感性分析 ===

SVM (Clean) 性能:
训练准确率: 0.9938
测试准确率: 0.9950

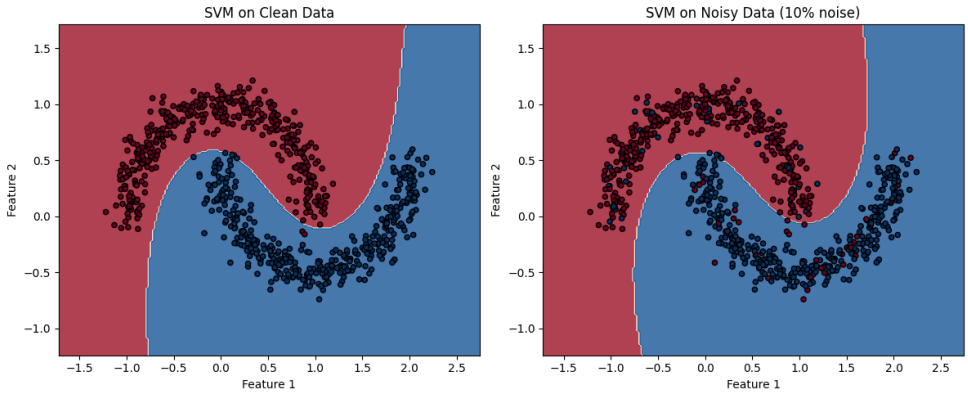


图 11: 噪声敏感性分析

精确率: 0.9950
召回率: 0.9950
F1 分数: 0.9950

SVM (Noisy) 性能:
训练准确率: 0.8888
测试准确率: 0.9950
精确率: 0.9950
召回率: 0.9950
F1 分数: 0.9950

SVM 对噪声敏感主要体现在训练过程中决策边界的扭曲，这从训练准确率的显著下降可以看出。然而，在这个案例中，SVM 展现出了很好的鲁棒性——尽管训练过程受到噪声干扰，但学到的模型在未见过的测试数据上仍然保持优异的泛化能力。

5.

表 4: 模型性能比较（原始数据）

模型	训练准确率	测试准确率	精确率	召回率	F1 分数
SVM (RBF)	0.9938	0.9950	0.9950	0.9950	0.9950
BP 神经网络	0.9988	1.0000	1.0000	1.0000	1.0000
C4.5 决策树	1.0000	1.0000	1.0000	1.0000	1.0000

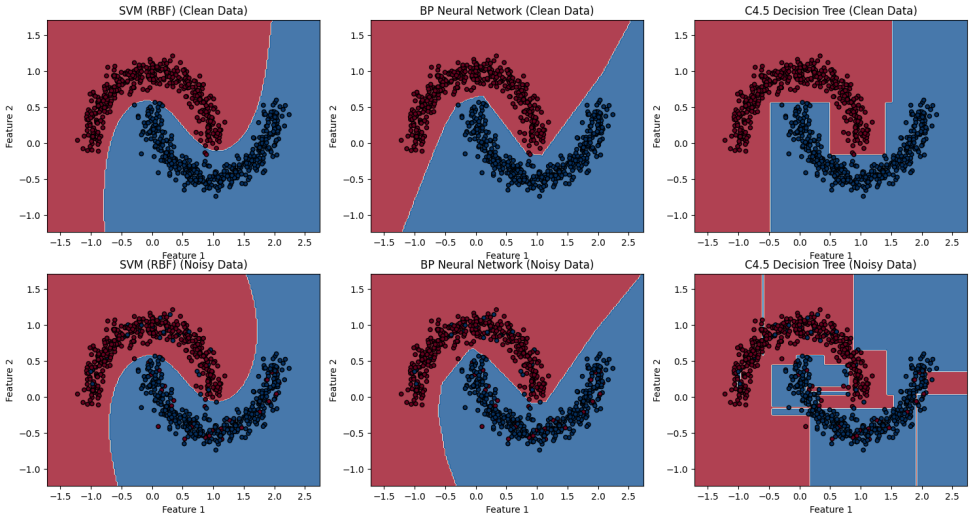


图 12: 所有模型比较

表 5: 模型性能比较（噪声数据）

模型	训练准确率	测试准确率	精确率	召回率	F1 分数
SVM (RBF)	0.8975	0.9950	0.9950	0.9950	0.9950
BP 神经网络	0.9012	1.0000	1.0000	1.0000	1.0000
C4.5 决策树	0.9450	0.9050	0.9060	0.9050	0.9049

6.

SVM (RBF 核):
优点: 对非线性数据拟合能力强，泛化性能好，对噪声相对鲁棒
缺点: 参数选择敏感，大规模数据训练较慢

BP 神经网络:
优点: 强大的非线性拟合能力，能够学习复杂模式
缺点: 对噪声敏感，容易过拟合，训练不稳定

C4.5 决策树:
优点: 解释性强，训练速度快，对某些类型噪声鲁棒
缺点: 容易过拟合，对数据分布敏感，泛化能力相对较弱

在 moons 数据集上:
SVM 在原始数据和噪声数据上都表现稳定
BP 神经网络在原始数据上表现好但对噪声敏感
决策树容易产生过于复杂的决策边界导致过拟合