一. (30 points) 类别不平衡

信用卡欺诈检测数据集 (Credit Card Fraud Detection) 包含了 2013 年 9 月通过信用卡进行的欧洲持卡人的交易。这是一个非常典型的类别不平衡数据集，数据集中正常交易的标签远多于欺诈交易。请你根据附件中提供的该数据集完成以下问题：

1. 该数据集共有 284807 个样本，其中只有 492 个负样本。请按照训练集和测试集比例 4：1 的方式划分数据集（使用固定的随机种子）。在训练集上训练 SVM 模型，并计算该模型在测试集上的精度（如准确率、召回率、F1 分数，AUC 等）。请展示 SVM 模型训练过程的完整代码，并绘制 ROC 曲线（8 points）；

2. 请从上述训练集中的正样本中分别随机剔除 2000，20000，200000 个样本，剩余的正样本和训练集中原本的负样本共同组成新的训练集，测试集保持不变。请参照上一小问方式在这三个新的训练集上训练 svm 模型，并记录每个模型的精度。观察并比较这几组实验的结果，结合准确率与召回率的定义，请说明不平衡数据集对模型的影响（9 points）；

3. 除了上述第 2 问的随机欠采样的方式以外，对小类样本的“过采样”也是处理不平衡问题的基本策略。一种经典的方法为人工合成的过采样技术 (Synthetic Minority Over-sampling Technique, SMOTE)，其在合成样本时寻找小类中某一个样本的近邻，并在该样本与近邻之间进行差值，作为合成的新样本。请查阅相关资料，实现 SMOTE 算法中的 over sampling 函数，以代码块的形式附于下方即可（8 points）；

```python
"""
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. 记得处理数值稳定性问题，例如在计算对数时避免除以零。
"""
class SMOTE(object):
    def __init__(self , X, y, N, K, random_state =0):
        self.N = N # 每个小类样本合成样本个数
        self.K = K # 近邻个数
        self.label = y # 进行数据增强的类别
        self.sample = X
        self.n_sample , self.n = self.sample.shape # 获得样本个数，特征个数

    def over_sampling(self):
        pass
```

Listing 1: SMOTE 模型接口

4. 请说明 SMOTE 算法的缺点并讨论可能的改进方案（5 points）。

解:

1.

```python
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, recall_score, f1_score, roc_auc_score, roc_curve, auc
import matplotlib.pyplot as plt

data = pd.read_csv('creditcard.csv')

X = data.drop(columns=['Class'])
y = data['Class']

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2, random_state=42)

svm_model = SVC(kernel='rbf', probability=True, random_state=42, class_weight='balanced',max_iter=1200)
svm_model.fit(X_train, y_train)

y_pred = svm_model.predict(X_test)
y_pred_proba = svm_model.predict_proba(X_test)[:, 1]

accuracy = accuracy_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)
roc_auc = roc_auc_score(y_test, y_pred_proba)

print(f'Accuracy: {accuracy:.4f}')
print(f'Recall: {recall:.4f}')
print(f'F1 Score: {f1:.4f}')
print(f'ROC AUC: {roc_auc:.4f}')

fpr, tpr, _ = roc_curve(y_test, y_pred_proba)
roc_auc_value = auc(fpr, tpr)

plt.figure()
plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (area = {roc_auc_value:.4f})')
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc='lower right')
plt.show()
```

Listing 2: SVM 模型训练过程的完整代码

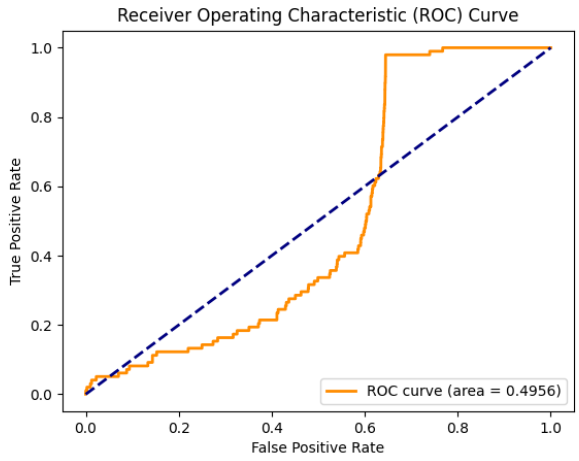

图 1: ROC 曲线

```
Accuracy: 0.2294
Recall: 1.0000
F1 Score: 0.0044
ROC AUC: 0.4956
```

图 2: 精度

2.

剔除 2000 个样本

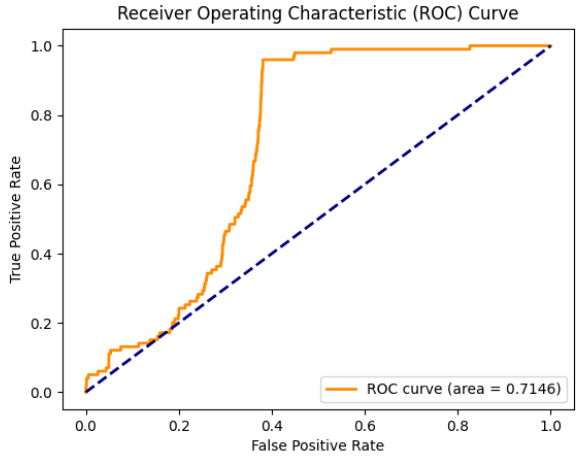

图 3: ROC 曲线

```
Accuracy: 0.4146
Recall: 0.9899
F1 Score: 0.0058
ROC AUC: 0.7146
```

图 4: 精度

剔除 20000 个样本

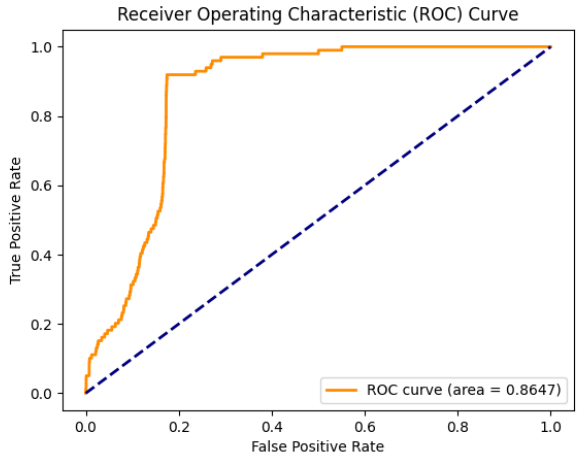

图 5: ROC 曲线

```
Accuracy: 0.4608
Recall: 0.9899
F1 Score: 0.0063
ROC AUC: 0.8647
```

图 6: 精度

剔除 200000 个样本

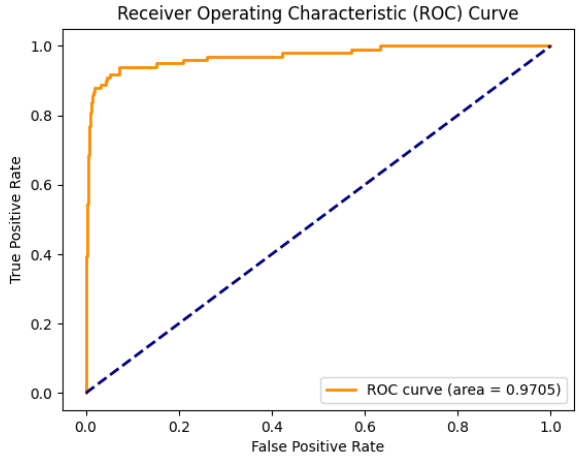

图 7: ROC 曲线

```
Accuracy: 0.9325
Recall: 0.9192
F1 Score: 0.0452
ROC AUC: 0.9705
```

图 8: 精度

在极度不平衡的数据集上，模型倾向于预测多数类（正样本），导致 Recall 较高而 Accuracy、F1 Score 较低。在这种情况下，模型可能只是通过简单地将几乎所有样本都分类为多数类来提高召回率。

随着剔除更多的多数类样本（正样本），数据集逐渐趋于平衡，模型的 Accuracy、F1 Score 和 ROC AUC 都得到了显著改善，这说明通过处理数据集的类别不平衡性，模型的整体性能得以提升。

在剔除 200000 个正样本后，模型取得了相对较高的 Accuracy 和 ROC AUC，但 Recall 略有下降，表明模型在对两类样本进行区分时取得了更好的平衡。

3.

```python
import numpy as np
from sklearn.neighbors import NearestNeighbors

class SMOTE(object):
    def __init__(self, X, y, N, K, random_state=0):
        self.N = N  # 每个小类样本合成样本个数
        self.K = K  # 近邻个数
        self.label = y  # 进行数据增强的类别
        self.sample = X[y == self.label]  # 筛选出小类样本
        self.n_sample, self.n = self.sample.shape  # 获得样本个数和特征个数
        self.random_state = random_state

    def over_sampling(self):
        np.random.seed(self.random_state)
        if self.N < 100:
            raise ValueError("N must be greater than or equal to 100")

        N = self.N // 100
        synthetic_samples = np.zeros((N * self.n_sample, self.n))

        neigh = NearestNeighbors(n_neighbors=self.K + 1)
        neigh.fit(self.sample)
        neighbors = neigh.kneighbors(self.sample, return_distance=False)

        new_index = 0
        for i in range(self.n_sample):
            nn_array = neighbors[i][1:]
            for _ in range(N):
                nn = np.random.choice(nn_array)
                diff = self.sample[nn] - self.sample[i]
                gap = np.random.rand()
                synthetic_samples[new_index] = self.sample[i] + gap * diff
                new_index += 1

        return synthetic_samples
```

Listing 3: 实现 SMOTE 算法中的 over sampling 函数

缺点：

1. SMOTE 在样本和其邻居之间进行插值，生成的新样本可能不一定真实地反映实际的数据分布，尤其在类边界附近可能会产生模糊的样本。
2. SMOTE 生成的新样本可能是噪声数据或者靠近多数类边界，这可能导致类间重叠，增加模型的误分类风险。
3. SMOTE 在生成新样本时，并没有考虑特征的实际分布。对于不均匀分布的少数类特征，生成的样本可能落在不合理的区域。
4. 对于少数类靠近多数类的边界样本，SMOTE 仍然会生成新样本，而这些新样本可能落入多数类区域，从而降低分类器对边界的辨别能力。

改进方案：

1. 使用过采样与欠采样结合的方法，在 SMOTE 过采样后使用欠采样技术来平衡多数类与少数类的比例，且剔除潜在噪声数据以提升模型的泛化能力。
2. 先使用聚类算法将少数类样本分为不同的簇，然后在每个簇内使用 SMOTE 进行采样，有助于生成更符合数据分布的新样本，避免生成脱离实际分布的样本。

二. (20 points) 机器学习中的过拟合现象

本题以决策树和多层感知机为例, 探究机器学习中的过拟合现象。在教材 2.1 节中提到, 机器学习希望训练得到的模型在新样本上保持较好的泛化性能. 如果在训练集上将模型训练得“过好”, 捕获到了训练样本中对分类无关紧要的特性, 会导致模型难以泛化到未知样本上, 这种情况称为过拟合。

1. 请简要总结决策树和多层感知机的工作原理及其对应的缓解过拟合的手段（决策树相关原理可以参考教材第 4 章, 多层感知机可以参考教材第 5 章）(5 points)；

2. 请使用 scikit-learn 实现决策树模型, 并扰动决策树的最大深度 max_depth, 一般来说, max_depth 的值越大, 决策树越复杂, 越容易过拟合, 实验并比较测试集精度, 讨论并分析观察到的过拟合现象等（5 points)；

3. 对决策树算法的未剪枝、预剪枝和后剪枝进行实验比较，并进行适当的统计显著性检验（5 points)；

4. 请使用 PyTorch 或 scikit-learn 实现一个简单的多层感知机, 并通过层数、宽度或者优化轮数控制模型复杂度, 实验并比较测试集精度, 讨论并分析观察到的过拟合现象等（5 points)。

注: 请选择 1 至 3 个 UCI 机器学习库中的数据集进行实验。

解:

1.

决策树

工作原理：决策树是一种基于树结构的分类或回归模型，通过递归地对数据集进行分割，构建出具有分支和叶节点的树。每个节点通过某个特征进行分割，直到达到停止条件。决策树根据数据集特征的信息增益或基尼系数等指标进行分裂，以找到最优的特征和分割点，使得子节点的纯度尽可能高。

过拟合的缓解手段：

1. 树剪枝：决策树在训练过程中容易生长得很深，从而导致过拟合。剪枝是通过在生成的完整决策树上删除一些不必要的节点来减少复杂度。可以进行预剪枝（在树生成时提前停止）或后剪枝（在树完全生成后再进行剪枝)。
2. 限制树的深度：限制树的最大深度可以减少模型复杂度，避免过于拟合数据。通过设定最大深度，可以防止树生长过深。
3. 限制叶节点的最小样本数：可以通过限制每个叶节点所包含的最小样本数，防止节点过于细分，减少对噪声的拟合。
4. 随机森林：随机森林是由多棵决策树组成的集成模型，通过结合多棵树的预测来缓解单棵树的过拟合问题。每棵树只使用部分特征和数据进行训练，这种随机性可以提高泛化能力。

多层感知机

工作原理：多层感知机（MLP）是一种典型的前馈神经网络，包含输入层、一个或多个隐藏层和输出层。每一层由多个神经元组成，神经元之间通过权重和激活函数进行连接。MLP 通过反向传播算法不断更新权重，以最小化预测误差。

过拟合的缓解手段：

1. 正则化：
   (a) L2 正则化（权重衰减）：通过在损失函数中增加权重参数的平方和，可以限制权重的大小，从而减少模型的复杂度，防止过拟合。
   (b) L1 正则化：通过在损失函数中增加权重的绝对值和，可以使部分权重变为零，实现特征选择，从而缓解过拟合。
2. Dropout：Dropout 是一种随机将神经网络中的一些神经元在训练过程中“丢弃”的方法。这可以防止网络对特定的神经元过于依赖，有效提高模型的泛化能力。
3. 早停：在训练过程中，如果验证集上的误差不再减小，则提前停止训练。这可以避免模型继续在训练集上学习，防止对训练数据的过拟合。
4. 数据增强：通过对训练数据进行增强（如图像的旋转、平移等），可以增加数据的多样性，从而提高模型的泛化能力。
5. Batch Normalization：对每层的输入进行归一化，使得模型训练更加稳定，加快收敛的同时也有助于缓解过拟合。

2.

```python
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
from sklearn.datasets import load_breast_cancer
import matplotlib.pyplot as plt

data = load_breast_cancer()
df = pd.DataFrame(data.data, columns=data.feature_names)
df['target'] = data.target

X = df.drop(columns=['target'])
y = df['target']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

max_depths = range(1, 21)
train_accuracies = []
test_accuracies = []

for max_depth in max_depths:
    model = DecisionTreeClassifier(max_depth=max_depth, random_state=42)
    model.fit(X_train, y_train)

    y_train_pred = model.predict(X_train)
    train_accuracy = accuracy_score(y_train, y_train_pred)
    train_accuracies.append(train_accuracy)
```

```
27
28      y_test_pred = model.predict(X_test)
29      test_accuracy = accuracy_score(y_test, y_test_pred)
30      test_accuracies.append(test_accuracy)
31
32      print(f"Max Depth: {max_depth}, Training Accuracy: {train_accuracy:.4f}, Testing Accuracy: {
        test_accuracy:.4f}")
33
34
35  plt.figure(figsize=(10, 6))
36  plt.plot(max_depths, train_accuracies, label='Training Accuracy', marker='o')
37  plt.plot(max_depths, test_accuracies, label='Testing Accuracy', marker='o')
38  plt.xlabel('Max Depth')
39  plt.ylabel('Accuracy')
40  plt.title('Decision Tree Model Complexity and Overfitting')
41  plt.legend()
42  plt.grid(True)
43  plt.show()
```

Listing 4: 使用 scikit-learn 实现决策树模型

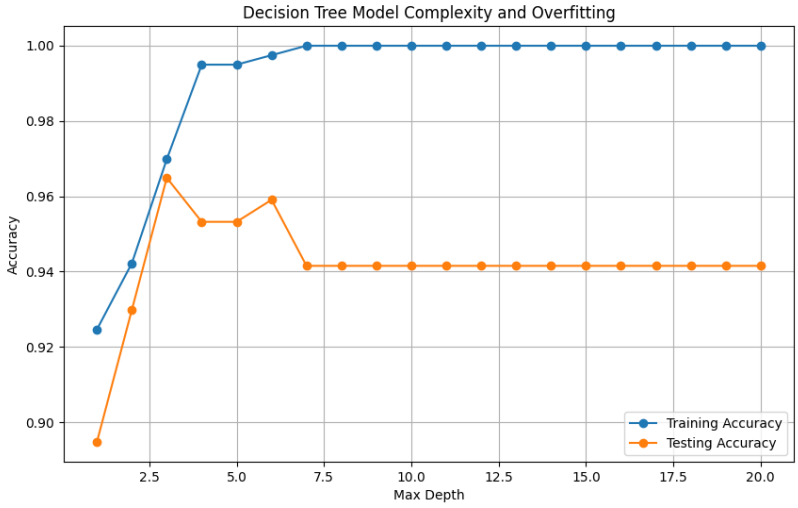

图 9: 决策树训练集和测试集的准确率

从实验结果可以观察到：

- 当 max_depth 较小（例如 1 到 3）时，模型的训练集和测试集准确率均较低。这种情况下，模型复杂度不足，无法捕捉数据中的模式，导致欠拟合。
- 随着 max_depth 增加到 4 到 6，训练集和测试集的准确率都显著提升，并且测试集的准确率也达到了较高的水平，这时模型的复杂度适中，能够较好地捕捉数据中的规律。
- 当 max_depth 继续增加到 7 及以上时，模型在训练集上的准确率达到了 100%，而测试集的准确率并没有继

续提升，反而出现了一定的下降。特别是当 max_depth 大于 6 后，测试集的准确率保持在约 94.15%，这表明模型已经开始过拟合——模型在训练集上表现完美，但由于学习到了训练数据中特有的噪声和细节，在测试集上的泛化能力变差。

这种现象说明，当决策树的深度过大时，模型会变得过于复杂，导致其在训练集上表现优异，但无法在测试集上保持良好的性能，从而表现出过拟合的特征。

3.

```python
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.metrics import accuracy_score
from sklearn.datasets import load_breast_cancer
from scipy.stats import ttest_ind
import matplotlib.pyplot as plt
import numpy as np

data = load_breast_cancer()
df = pd.DataFrame(data.data, columns=data.feature_names)
df['target'] = data.target

X = df.drop(columns=['target'])
y = df['target']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

max_depths = range(1, 21)
train_accuracies_no_pruning = []
test_accuracies_no_pruning = []
train_accuracies_pre_pruning = []
test_accuracies_pre_pruning = []
train_accuracies_post_pruning = []
test_accuracies_post_pruning = []

for max_depth in max_depths:
    model = DecisionTreeClassifier(max_depth=max_depth, random_state=42)
    model.fit(X_train, y_train)

    y_train_pred = model.predict(X_train)
    train_accuracies_no_pruning.append(accuracy_score(y_train, y_train_pred))

    y_test_pred = model.predict(X_test)
    test_accuracies_no_pruning.append(accuracy_score(y_test, y_test_pred))

    print(f"No Pruning - Max Depth: {max_depth}, Training Accuracy: {train_accuracies_no_pruning[-1]:.4f}, Testing Accuracy: {test_accuracies_no_pruning[-1]:.4f}")

min_samples_splits = [2, 5, 10, 20]
for min_samples_split in min_samples_splits:
    model = DecisionTreeClassifier(min_samples_split=min_samples_split, random_state=42)
    model.fit(X_train, y_train)
```

```
42
43      y_train_pred = model.predict(X_train)
44      train_accuracies_pre_pruning.append(accuracy_score(y_train, y_train_pred))
45
46      y_test_pred = model.predict(X_test)
47      test_accuracies_pre_pruning.append(accuracy_score(y_test, y_test_pred))
48
49      print(f"Pre-Pruning - Min Samples Split: {min_samples_split}, Training Accuracy: {
        train_accuracies_pre_pruning[-1]:.4f}, Testing Accuracy: {test_accuracies_pre_pruning[-1]:.4f}")
50
51  ccp_alphas = np.linspace(0, 0.02, 10)
52  for ccp_alpha in ccp_alphas:
53      model = DecisionTreeClassifier(ccp_alpha=ccp_alpha, random_state=42)
54      model.fit(X_train, y_train)
55
56      y_train_pred = model.predict(X_train)
57      train_accuracies_post_pruning.append(accuracy_score(y_train, y_train_pred))
58
59      y_test_pred = model.predict(X_test)
60      test_accuracies_post_pruning.append(accuracy_score(y_test, y_test_pred))
61
62      print(f"Post-Pruning - CCP Alpha: {ccp_alpha:.4f}, Training Accuracy: {train_accuracies_post_pruning
        [-1]:.4f}, Testing Accuracy: {test_accuracies_post_pruning[-1]:.4f}")
63
64  plt.figure(figsize=(15, 10))
65  plt.plot(max_depths, test_accuracies_no_pruning, label='No Pruning (Max Depth)', marker='o')
66  plt.plot(min_samples_splits, test_accuracies_pre_pruning, label='Pre-Pruning (Min Samples Split)', marker
        ='o')
67  plt.plot(ccp_alphas, test_accuracies_post_pruning, label='Post-Pruning (Cost Complexity)', marker='o')
68  plt.xlabel('Model Complexity Parameter')
69  plt.ylabel('Testing Accuracy')
70  plt.title('Decision Tree Pruning Methods Comparison')
71  plt.legend()
72  plt.grid(True)
73  plt.show()
```

Listing 5: 未剪枝、预剪枝和后剪枝

未剪枝:

- 随着 max_depth 增加，训练集的准确率很快达到了 100%，而测试集的准确率在 max_depth 超过 6 之后趋于平稳。
- 训练集的准确率为 100%，这表明模型过度拟合了训练数据。虽然在训练集上的表现极佳，但在测试集上的泛化性能并不理想，准确率稳定在 94.15% 左右。
- 这种现象反映出未剪枝的模型在处理训练集时容易学习到训练数据中的噪声和细节，从而影响模型在测试数据上的表现。

预剪枝 (Pre-Pruning)

- 预剪枝通过设置 min_samples_split 来控制模型的复杂度。当 min_samples_split 增加时，模型的训练集准确率逐渐降低，但测试集准确率却相对保持稳定。
- 当 min_samples_split 为 20 时，测试集的准确率达到了 93.57%，但相比于未剪枝的结果，预剪枝可以有效地降低过拟合的风险。
- 预剪枝的优点在于它能够在模型过度复杂之前停止生长，从而提高泛化性能，但其准确率略低于最优未剪枝模型。

后剪枝 (Post-Pruning)

- 后剪枝通过设置 ccp_alpha 来平衡模型的复杂度。可以看到，当 ccp_alpha 增加时，训练集的准确率逐渐降低，而测试集的准确率在某些特定值 (ccp_alpha=0.0067 和 0.0089) 时达到了较高的水平 (96.49%)。
- 后剪枝可以通过控制树的复杂度来减少过拟合，使模型对测试集的泛化性能有所提升。与未剪枝相比，后剪枝可以更有效地控制模型的复杂度，使得测试集上的表现更为稳定。

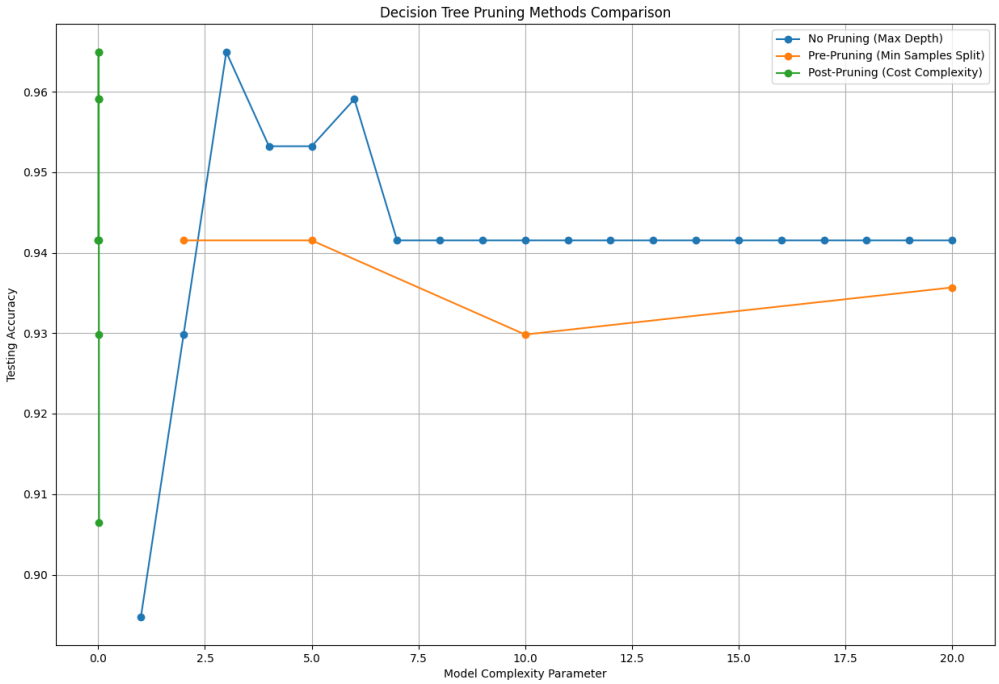

图 10: 决策树训练集和测试集的准确率

4.

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
```

```
from sklearn.preprocessing import StandardScaler
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score

data = load_breast_cancer()
X = data.data
y = data.target

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

hidden_layer_sizes = [(10,), (50,), (100,)]  # 控制层数和宽度
max_iter_list = [200, 500, 1000]  # 控制优化轮数

train_accuracies = []
test_accuracies = []
configurations = []

for layers in hidden_layer_sizes:
    for max_iter in max_iter_list:
        mlp = MLPClassifier(hidden_layer_sizes=layers, max_iter=max_iter, random_state=42)
        mlp.fit(X_train, y_train)

        y_train_pred = mlp.predict(X_train)
        y_test_pred = mlp.predict(X_test)

        train_accuracy = accuracy_score(y_train, y_train_pred)
        test_accuracy = accuracy_score(y_test, y_test_pred)

        train_accuracies.append(train_accuracy)
        test_accuracies.append(test_accuracy)
        configurations.append(f"Layers: {layers}, Iter: {max_iter}")

        print(f"Hidden layers: {layers}, Max iterations: {max_iter}")
        print(f"Train accuracy: {train_accuracy:.4f}, Test accuracy: {test_accuracy:.4f}")
        print("-" * 40)

plt.figure(figsize=(10, 6))

plt.plot(configurations, train_accuracies, marker='o', label='Train Accuracy', color='b')
plt.plot(configurations, test_accuracies, marker='o', label='Test Accuracy', color='r')

plt.xticks(rotation=90)
plt.xlabel('Configuration (Hidden Layers, Max Iterations)')
plt.ylabel('Accuracy')
plt.title('MLP Train vs Test Accuracy for Different Configurations')
plt.legend()

```

```
56 plt.tight_layout()
57 plt.show()
```

Listing 6: MLP

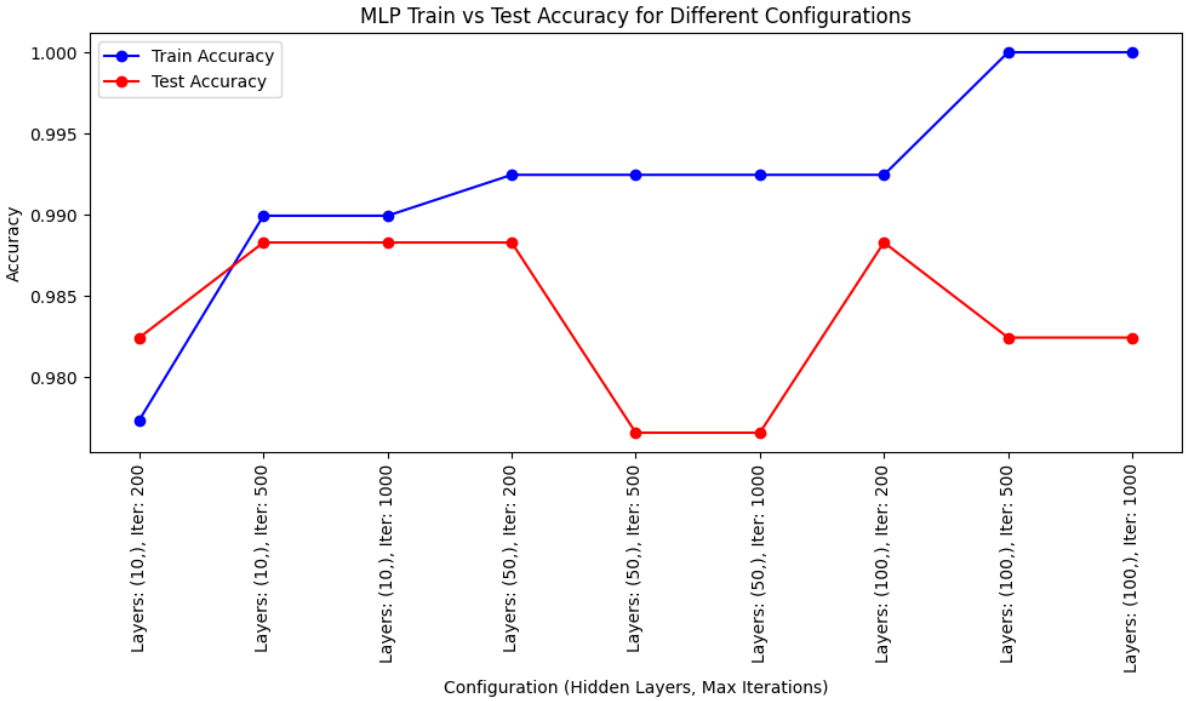

图 11: MLP 训练集和测试集的准确率

三. (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. Softplus 函数, 定义如下

$$f(x) = \ln(1+e^{x}). \tag{4}$$

5. Leaky Rectified Linear Unit (Leaky ReLU) 函数, 定义如下

$$f(x) = \begin{cases} \alpha x, & \text{if } x < 0; \\ x, & \text{otherwise.} \end{cases} \tag{5}$$

其中 α 为一个较小的正数, 比如 0.01.

解:

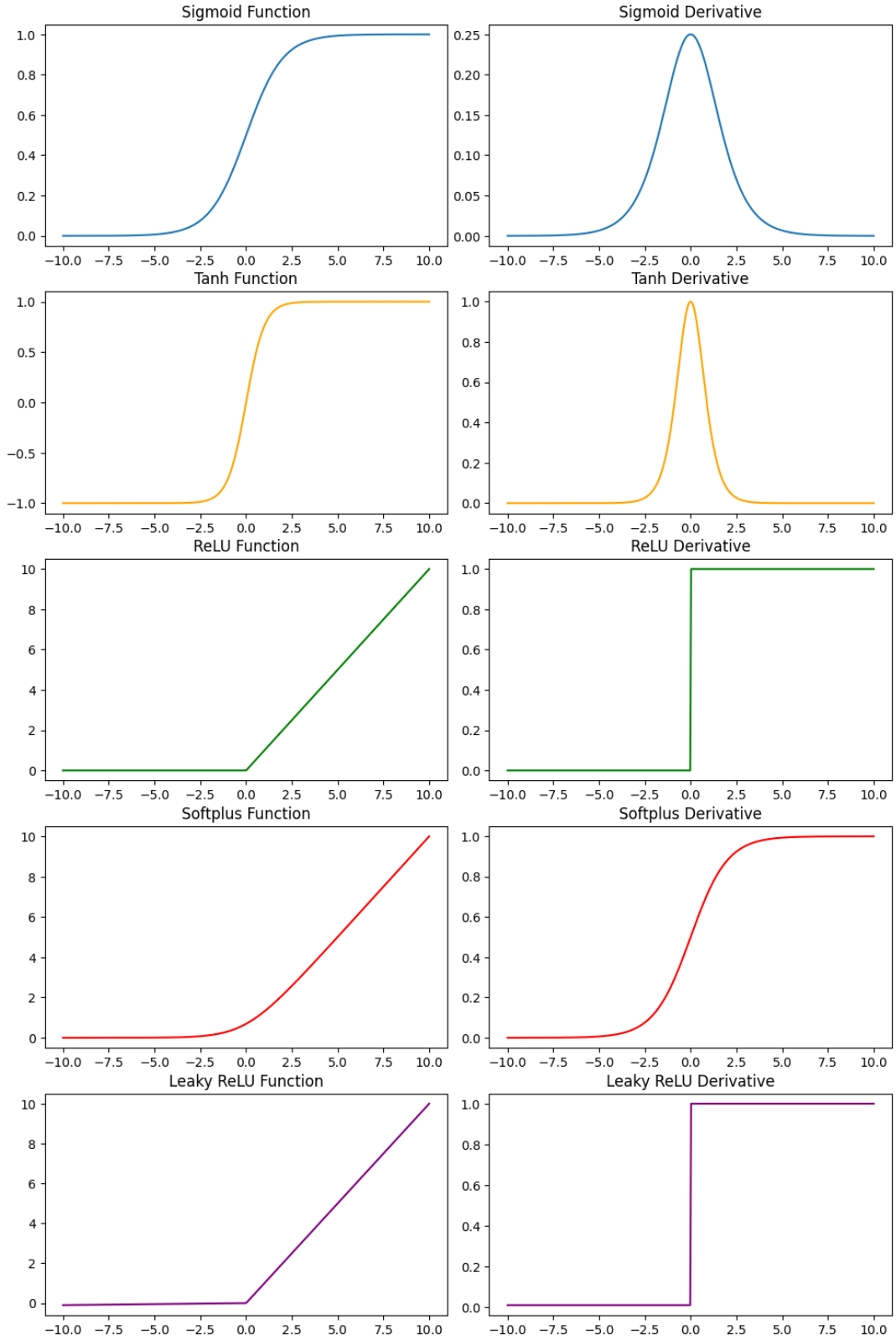

图 12: 隐层激活函数示意图

- Sigmoid 函数:
  - 优点：输出值介于 (0, 1)，适合用于概率估计。
  - 缺点：容易产生梯度消失问题，尤其是在深层网络中，较大的输入会导致梯度非常小，更新缓慢。
- Tanh 函数:
  - 优点：输出值介于 (-1, 1)，对称性好，通常收敛速度比 Sigmoid 快。
  - 缺点：同样存在梯度消失问题，较大的输入也会导致梯度趋于零。
- ReLU 函数:
  - 优点：计算简单，不会在正区间出现梯度消失问题。有效解决了深度网络中的梯度消失问题，通常能加速收敛。
  - 缺点：负区间的梯度恒为 0，可能导致“神经元死亡”问题（即神经元不再更新）。
- Softplus 函数:
  - 优点：是 ReLU 的平滑版本，避免了 ReLU 的不连续性，适合需要连续平滑函数的应用。
  - 缺点：与 ReLU 相比，计算量更大，在实践中不如 ReLU 常用。
- Leaky ReLU 函数:
  - 优点：允许负区间有小的负斜率，避免了 ReLU 的神经元死亡问题。
  - 缺点：负斜率的选择比较依赖超参数，且效果不如 ReLU 明显。

四. (30 points) 神经网络实战

moons 是一个简单的二分类数据集，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。

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 (4 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:
   - 随机初始化：从均匀分布 $U(-0.5, 0.5)$ 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。
   - 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 == 'random':
            # 实现Random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 7: 代码实现模板

解:

```python
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)
y = y.reshape(-1, 1)
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 == 'random':
            # 实现Random初始化
            self.W1 = np.random.uniform(-0.5, 0.5, (self.input_size, self.hidden_size))
            self.b1 = np.zeros((1, self.hidden_size))
            self.W2 = np.random.uniform(-0.5, 0.5, (self.hidden_size, self.output_size))
            self.b2 = np.zeros((1, self.output_size))
        elif init_method == 'xavier':
            # 实现Xavier初始化
            limit1 = np.sqrt(6 / (self.input_size + self.hidden_size))
            limit2 = np.sqrt(6 / (self.hidden_size + self.output_size))
            self.W1 = np.random.uniform(-limit1, limit1, (self.input_size, self.hidden_size))
            self.b1 = np.zeros((1, self.hidden_size))
            self.W2 = np.random.uniform(-limit2, limit2, (self.hidden_size, self.output_size))
            self.b2 = np.zeros((1, self.output_size))
        elif init_method == 'he':
            # 实现He初始化
            self.W1 = np.random.randn(self.input_size, self.hidden_size) * np.sqrt(2 / self.input_size)
            self.b1 = np.zeros((1, self.hidden_size))
            self.W2 = np.random.randn(self.hidden_size, self.output_size) * np.sqrt(2 / self.hidden_size)
            self.b2 = np.zeros((1, self.output_size))
        else:
            raise ValueError("Unsupported initialization method")

    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 sigmoid_derivative(self, x):
```

```
        return self.sigmoid(x) * (1 - self.sigmoid(x))

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

    def backward(self, X, y, y_pred):
        # 实现反向传播
        m = X.shape[0]
        dz2 = y_pred - y
        dW2 = np.dot(self.a1.T, dz2) / m
        db2 = np.sum(dz2, axis=0, keepdims=True) / m
        da1 = np.dot(dz2, self.W2.T)
        dz1 = da1 * self.relu_derivative(self.z1)
        dW1 = np.dot(X.T, dz1) / m
        db1 = np.sum(dz1, axis=0, keepdims=True) / m
        return dW1, db1, dW2, db2


    def train(self, X, y, learning_rate, epochs, batch_size):
        # 实现小批量梯度下降训练
        losses = []
        accuracies = []

        for epoch in range(epochs):
            permutation = np.random.permutation(X_train.shape[0])
            X_shuffled = X[permutation]
            y_shuffled = y[permutation]

            for i in range(0, X_train.shape[0], batch_size):
                X_batch = X_shuffled[i:i + batch_size]
                y_batch = y_shuffled[i:i + batch_size]

                # 前向传播
                y_pred = self.forward(X_batch)

                # 反向传播
                dW1, db1, dW2, db2 = self.backward(X_batch, y_batch, y_pred)

                # 更新权重和偏置
                self.W1 -= learning_rate * dW1
                self.b1 -= learning_rate * db1
                self.W2 -= learning_rate * dW2
                self.b2 -= learning_rate * db2

                # 计算准确率和损失
                y_pred_full = self.forward(X)
```

```
                loss = -np.mean(y * np.log(y_pred_full + 1e-8) + (1 - y) * np.log(1 - y_pred_full + 1e-8)
    )
                losses.append(loss)
                accuracy = np.mean((y_pred_full > 0.5) == y)
                accuracies.append(accuracy)

            if epoch % 100 == 0:
                print(f"Epoch {epoch}, Loss: {loss:.4f}, Accuracy: {accuracy:.4f}")

        return losses, accuracies

# 辅助函数
def plot_decision_boundary(model, X, y):
    # 绘制决策边界
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),np.arange(y_min, y_max, 0.1))
    Z = model.forward(np.c_[xx.ravel(), yy.ravel()])
    Z = (Z > 0.5).astype(int)
    Z = Z.reshape(xx.shape)
    plt.contourf(xx, yy, Z, alpha=0.8)
    plt.scatter(X[:, 0], X[:, 1], c=y.ravel(), edgecolor='k', marker='o')
    plt.show()

def plot_training_process(losses, accuracies):
    # 绘制训练过程
    epochs = len(losses)

    fig, ax1 = plt.subplots()
    ax1.set_xlabel('Iteration')
    ax1.set_ylabel('Loss', color='tab:red')
    ax1.plot(range(epochs), losses, color='tab:red', label="Loss")
    ax1.tick_params(axis='y', labelcolor='tab:red')

    ax2 = ax1.twinx()
    ax2.set_ylabel('Accuracy', color='tab:blue')
    ax2.plot(range(epochs), accuracies, color='tab:blue', label="Accuracy")
    ax2.tick_params(axis='y', labelcolor='tab:blue')

    fig.tight_layout()
    plt.show()

# 主函数
def main():
    # 创建并训练模型
    # 绘制结果
    input_size = 2
    hidden_size = 4
    output_size = 1
    learning_rate = 0.01
    epochs = 1000
```

```
    batch_size = 32
    # 随机初始化创建训练模型、绘制训练过程和决策边界
    model_random = NeuralNetwork(input_size, hidden_size, output_size, 'random')
    losses_random, accuracies_random = model_random.train(X_train, y_train, learning_rate, epochs,
    batch_size)
    plot_training_process(losses_random, accuracies_random)
    plot_decision_boundary(model_random, X, y)

    # Xavier初始化创建训练模型、绘制训练过程和决策边界
    model_xavier = NeuralNetwork(input_size, hidden_size, output_size, 'xavier')
    losses_xavier, accuracies_xavier = model_xavier.train(X_train, y_train, learning_rate, epochs,
    batch_size)
    plot_training_process(losses_xavier, accuracies_xavier)
    plot_decision_boundary(model_xavier, X, y)

    # He初始化创建训练模型、绘制训练过程和决策边界
    model_he = NeuralNetwork(input_size, hidden_size, output_size, 'he')
    losses_he, accuracies_he = model_he.train(X_train, y_train, learning_rate, epochs, batch_size)
    plot_training_process(losses_he, accuracies_he)
    plot_decision_boundary(model_he, X, y)

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

Listing 8: 代码实现

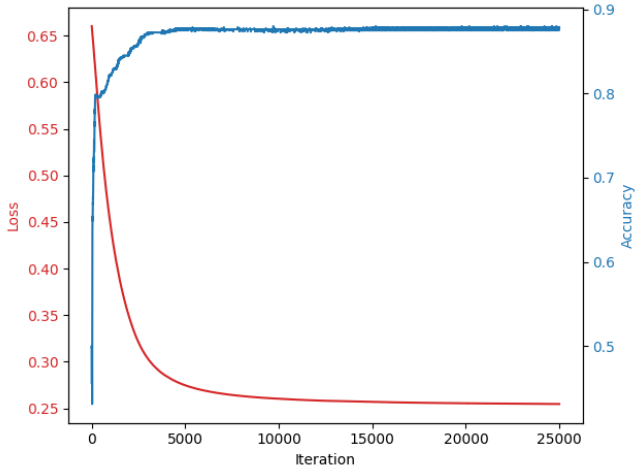

图 13: 随机初始化训练过程中的损失值和准确率

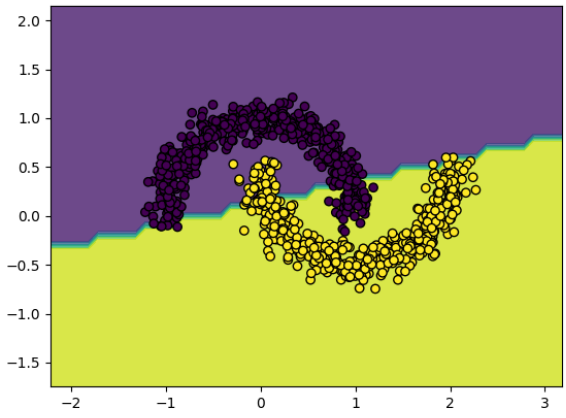

图 14: 随机初始化训练过程中的决策边界

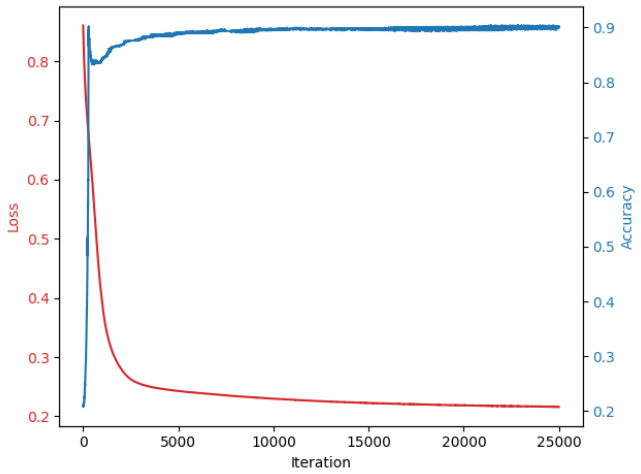

图 15: Xavier 初始化训练过程中的损失值和准确率

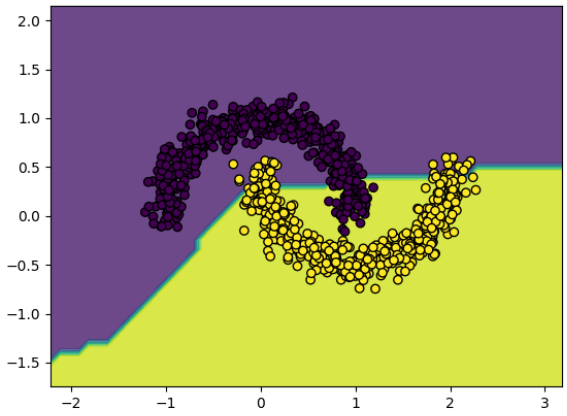

图 16: Xavier 初始化训练过程中的决策边界

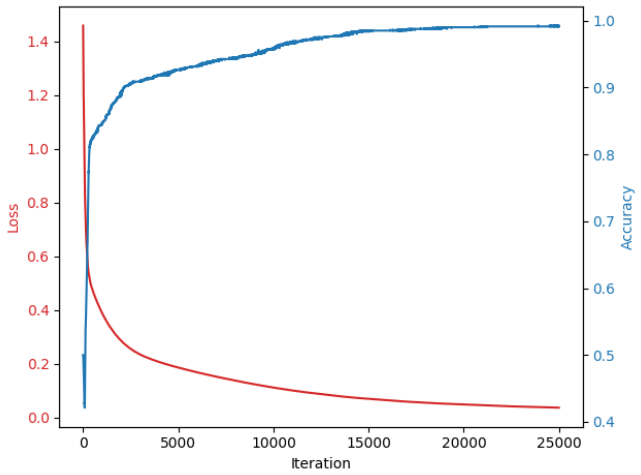

图 17: He 初始化训练过程中的损失值和准确率

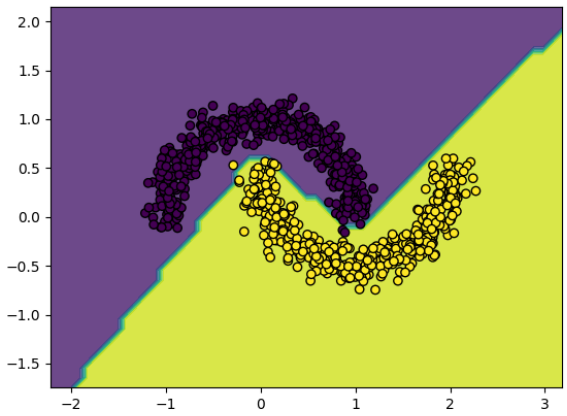

图 18: He 初始化训练过程中的决策边界

由图可以看到：

1. 随机初始化损失值逐渐从初始的 0.6548 下降到 0.2549，准确率从 44.62% 提升至 87.75%。收敛速度较慢，准确率提升较慢。
2. Xavier 初始化损失值从初始的 0.8338 降到 0.2144，准确率从 37.75% 提升至 90%。收敛速度较快，准确率也有所提高。
3. He 初始化损失值从初始的 1.2198 降到 0.0419，准确率从 47.63% 提升至 99.12%。损失迅速下降并且准确率大幅提升。

分析：初始化方式直接影响到神经网络的学习速度和收敛效果。He 初始化考虑到了 ReLU 的激活特性，从而能够加

速网络的学习。而 Xavier 初始化则适合于 sigmoid 或 tanh 激活函数，也表现出不错的收敛速度。随机初始化尽管可行，但因为没有考虑层的结构，可能导致学习速度慢或收敛不稳定。

如果将学习率上调至 0.05 可以得到如下结果：

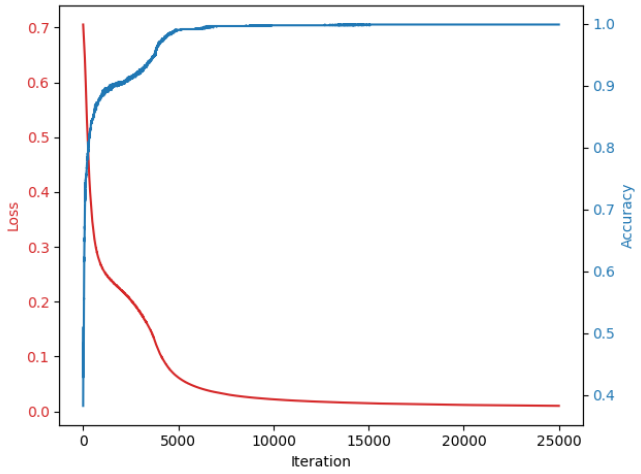

图 19: 随机初始化训练过程中的损失值和准确率

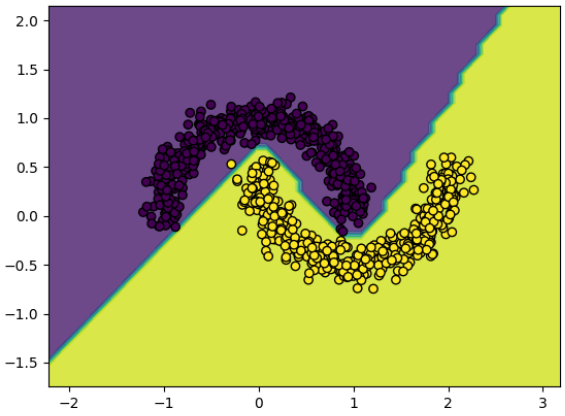

图 20: 随机初始化训练过程中的决策边界

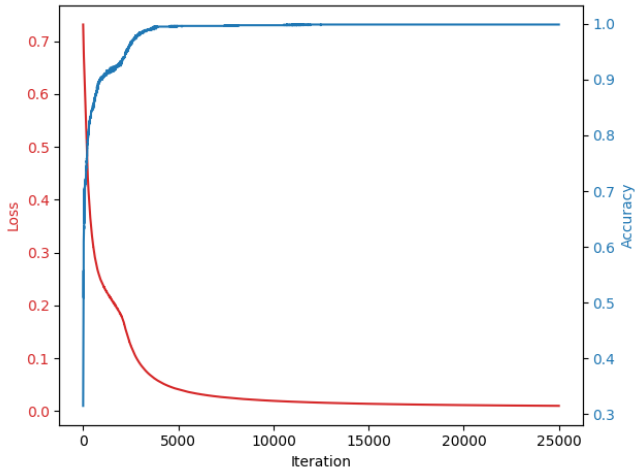

图 21: Xavier 初始化训练过程中的损失值和准确率

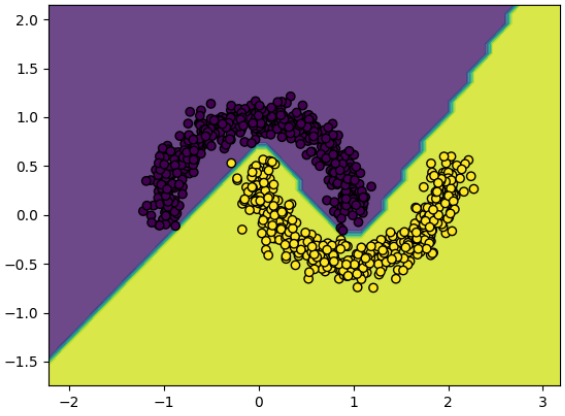

图 22: Xavier 初始化训练过程中的决策边界

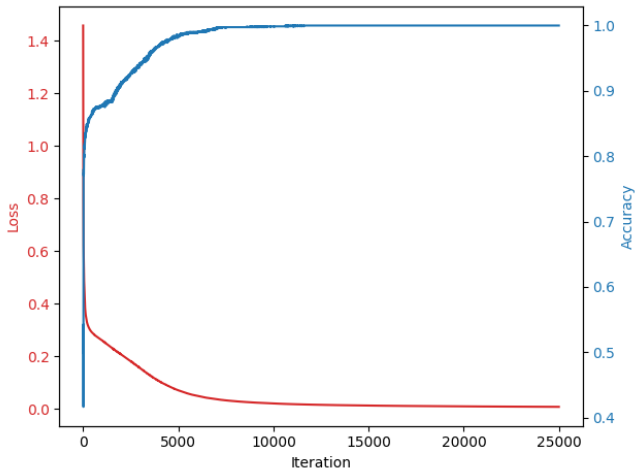

图 23: He 初始化训练过程中的损失值和准确率

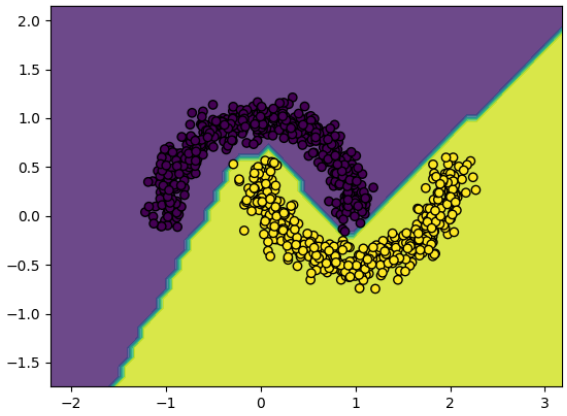

图 24: He 初始化训练过程中的决策边界

可以发现三种初始化方法下准确率均有提升，且随机初始化准确率显著提升至接近 0.1

适当提高学习率可以加快权重的更新速度，使模型更快地找到局部最优点或全局最优点。在 0.05 的学习率下，模型能够更快地降低损失值，并显著提升准确率。