# 报告 - 数据集简介 ANN建模分析 超参数优化分析

# 作者:Du

## 作者GitHub：<https://github.com/dqswordman/MUT_Neural_Networks_LAB> ->LAB5 代码 代码分析 文档

# 1.数据集简介

## 1.1 数据集内容 分析 预处理 特征和标签的提取、标签的数值编码、数据集的拆分以及标准化的操作

在本次分析中，我们使用了一个来源于Kaggle的网络流量数据集，数据集名称为“Friday-WorkingHours-Morning.pcap\_ISCX.csv”。

该数据集由 SYED ALI HAIDER 发布，目的是研究和检测网络中的DDoS攻击及其特征。数据集中每条记录代表一个网络流，包含了网络连接的多种特征和指标，这些特征被用来构建检测模型以区分正常流量和攻击流量。

数据集来源：<https://www.kaggle.com/code/syedali110/intrusion-detection-using-ann>

数据集内容分析：

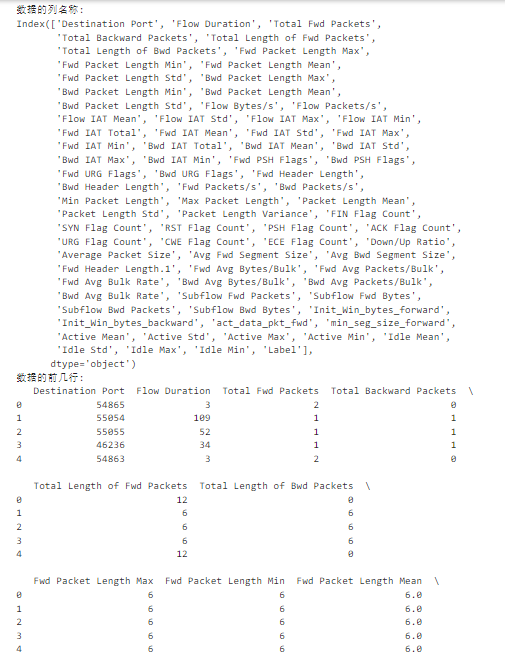
1. **Destination Port**：目标端口号，表示数据包传输的目标端口。
2. **Flow Duration**：流持续时间，表示网络流从开始到结束的时间长度（毫秒）。
3. **Total Fwd Packets**：总的向前（从源到目的地）数据包数量。
4. **Total Backward Packets**：总的向后（从目的地到源）数据包数量。
5. **Total Length of Fwd Packets**：向前数据包的总字节长度。
6. **Total Length of Bwd Packets**：向后数据包的总字节长度。
7. **Fwd Packet Length Max**：向前数据包的最大长度。
8. **Fwd Packet Length Min**：向前数据包的最小长度。
9. **Fwd Packet Length Mean**：向前数据包的平均长度。
10. **Fwd Packet Length Std**：向前数据包长度的标准差。
11. **Bwd Packet Length Max**：向后数据包的最大长度。
12. **Bwd Packet Length Min**：向后数据包的最小长度。
13. **Bwd Packet Length Mean**：向后数据包的平均长度。
14. **Bwd Packet Length Std**：向后数据包长度的标准差。
15. **Flow Bytes/s**：每秒流量字节数。
16. **Flow Packets/s**：每秒流量数据包数。
17. **Flow IAT Mean**：流的到达时间间隔的平均值。
18. **Flow IAT Std**：流的到达时间间隔的标准差。
19. **Flow IAT Max**：流的到达时间间隔的最大值。
20. **Flow IAT Min**：流的到达时间间隔的最小值。
21. **Fwd IAT Total**：所有向前数据包到达时间间隔的总和。
22. **Fwd IAT Mean**：向前数据包到达时间间隔的平均值。
23. **Fwd IAT Std**：向前数据包到达时间间隔的标准差。
24. **Fwd IAT Max**：向前数据包到达时间间隔的最大值。
25. **Fwd IAT Min**：向前数据包到达时间间隔的最小值。
26. **Bwd IAT Total**：所有向后数据包到达时间间隔的总和。
27. **Bwd IAT Mean**：向后数据包到达时间间隔的平均值。
28. **Bwd IAT Std**：向后数据包到达时间间隔的标准差。
29. **Bwd IAT Max**：向后数据包到达时间间隔的最大值。
30. **Bwd IAT Min**：向后数据包到达时间间隔的最小值。
31. **Fwd PSH Flags**：向前数据包中的PSH标志数量。
32. **Bwd PSH Flags**：向后数据包中的PSH标志数量。
33. **Fwd URG Flags**：向前数据包中的URG标志数量。
34. **Bwd URG Flags**：向后数据包中的URG标志数量。
35. **Fwd Header Length**：向前数据包的标头长度总和。
36. **Bwd Header Length**：向后数据包的标头长度总和。
37. **Fwd Packets/s**：每秒向前传输的数据包数量。
38. **Bwd Packets/s**：每秒向后传输的数据包数量。
39. **Min Packet Length**：数据包的最小长度。
40. **Max Packet Length**：数据包的最大长度。
41. **Packet Length Mean**：数据包长度的平均值。
42. **Packet Length Std**：数据包长度的标准差。
43. **Packet Length Variance**：数据包长度的方差。
44. **FIN Flag Count**：数据包中FIN标志的数量。
45. **SYN Flag Count**：数据包中SYN标志的数量。
46. **RST Flag Count**：数据包中RST标志的数量。
47. **PSH Flag Count**：数据包中PSH标志的数量。
48. **ACK Flag Count**：数据包中ACK标志的数量。
49. **URG Flag Count**：数据包中URG标志的数量。
50. **CWE Flag Count**：数据包中CWE标志的数量。
51. **ECE Flag Count**：数据包中ECE标志的数量。
52. **Down/Up Ratio**：下载和上传数据包数量的比率。
53. **Average Packet Size**：数据包的平均大小。
54. **Avg Fwd Segment Size**：向前数据包的平均段大小。
55. **Avg Bwd Segment Size**：向后数据包的平均段大小。
56. **Fwd Header Length.1**：重复的向前数据包标头长度。
57. **Fwd Avg Bytes/Bulk**：每批次向前传输的平均字节数。
58. **Fwd Avg Packets/Bulk**：每批次向前传输的平均数据包数量。
59. **Fwd Avg Bulk Rate**：向前批次传输的平均速率。
60. **Bwd Avg Bytes/Bulk**：每批次向后传输的平均字节数。
61. **Bwd Avg Packets/Bulk**：每批次向后传输的平均数据包数量。
62. **Bwd Avg Bulk Rate**：向后批次传输的平均速率。
63. **Subflow Fwd Packets**：子流中的向前数据包数量。
64. **Subflow Fwd Bytes**：子流中的向前字节数。
65. **Subflow Bwd Packets**：子流中的向后数据包数量。
66. **Subflow Bwd Bytes**：子流中的向后字节数。
67. **Init\_Win\_bytes\_forward**：向前初始窗口字节数。
68. **Init\_Win\_bytes\_backward**：向后初始窗口字节数。
69. **act\_data\_pkt\_fwd**：实际向前数据包数量。
70. **min\_seg\_size\_forward**：最小向前段大小。
71. **Active Mean**：活动时间的平均值。
72. **Active Std**：活动时间的标准差。
73. **Active Max**：活动时间的最大值。
74. **Active Min**：活动时间的最小值。
75. **Idle Mean**：空闲时间的平均值。
76. **Idle Std**：空闲时间的标准差。
77. **Idle Max**：空闲时间的最大值。
78. **Idle Min**：空闲时间的最小值。
79. **Label**：分类标签（如“BENIGN”或“DDoS”），表示网络流量是否为正常流量或攻击流量。

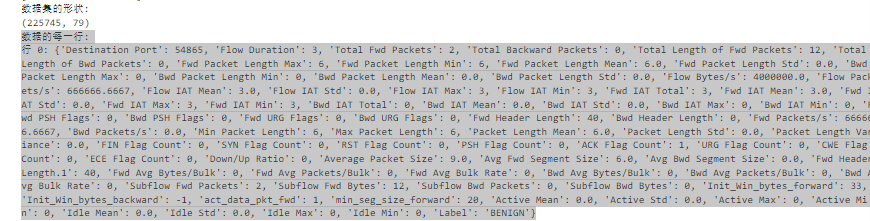
### From Lab5\_Code\_Note.ipynb #3.1

*#3.1*

1. import pandas as pd
2. *# 读取数据*
3. df = pd.read\_csv('./Friday-WorkingHours-Afternoon-DDos.pcap\_ISCX.csv')
4. *# 去除列名中的前后空格*
5. df.columns = df.columns.str.strip()
6. *# 显示数据的每一列*
7. print("数据的列名称:")
8. print(df.columns)
9. *# 显示数据的前几行*
10. print("数据的前几行:")
11. print(df.head())
12. *# 显示数据集的形状（行数和列数）*
13. print("数据集的形状:")
14. print(df.shape)
15. *# 显示数据的每一行（可能会输出大量信息，注意使用）*
16. print("数据的每一行:")
17. for index, row in df.iterrows():
18. print(f"行 {index}: {row.to\_dict()}")
19. *# 注意：这里只是一个示例，如果数据集非常大（如超过几百行），这可能会导致输出过多信息。*
20. *# 可以使用条件限制打印的行数，例如只打印前10行:*
21. if index >= 9:  *# 打印前10行*
22. break

#### 输出如图



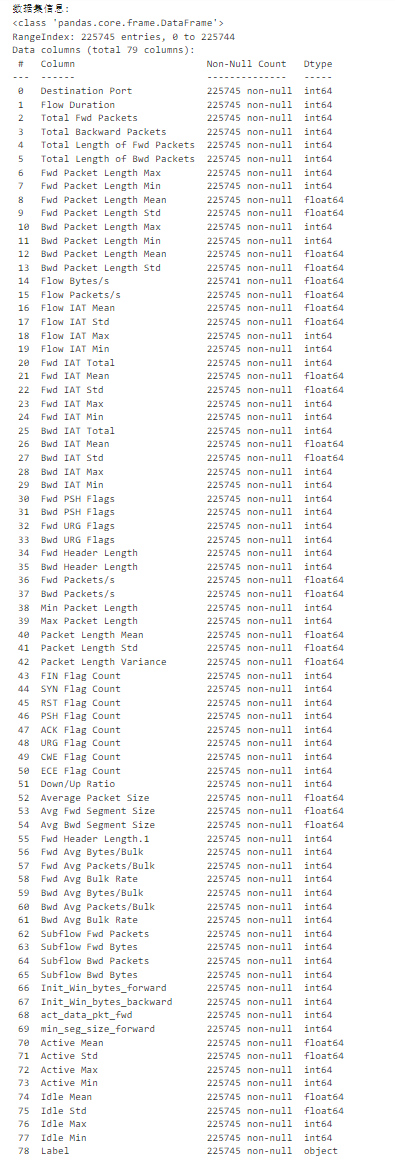


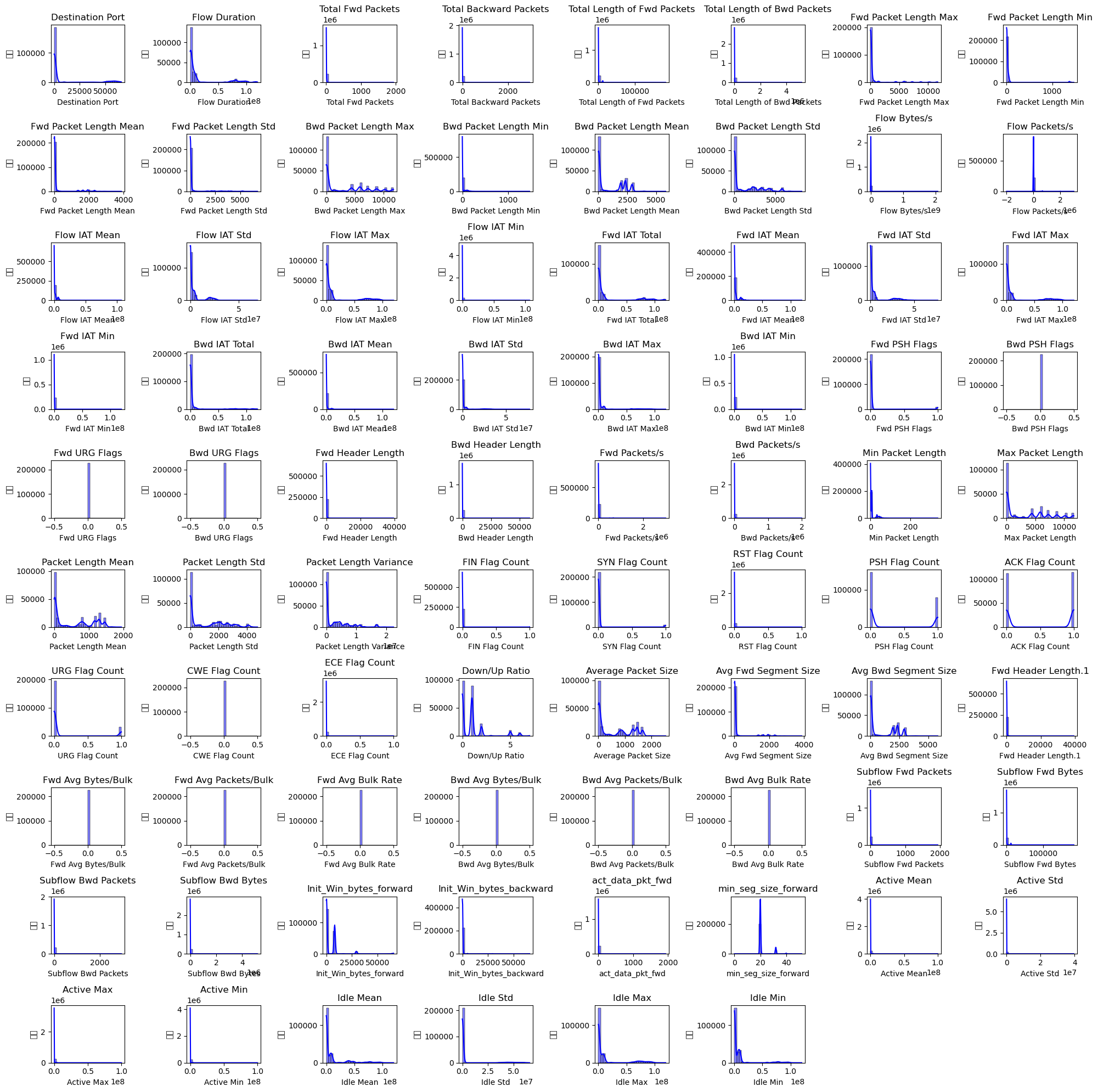
## 1.2 生成数据集字段的直方图分析

### From Lab5\_Code\_Note.ipynb #3.2

*#3.2*

1. import pandas as pd
2. import numpy as np
3. import matplotlib.pyplot as plt
4. import seaborn as sns
5. *# 读取数据集*
6. df = pd.read\_csv('./Friday-WorkingHours-Afternoon-DDos.pcap\_ISCX.csv')
7. *# 去除列名中的前后空格*
8. df.columns = df.columns.str.strip()
9. *# 检查数据集信息*
10. print("数据集信息:")
11. print(df.info())
12. *# 选择数值型特征*
13. numeric\_features = df.select\_dtypes(include=[np.number]).columns
14. *# 设置图形的大小*
15. plt.figure(figsize=(20, 20))
16. *# 生成数值型特征的直方图*
17. for i, col in enumerate(numeric\_features):
18. plt.subplot(10, 8, i + 1)  *# 根据列数调整子图的布局*
19. sns.histplot(df[col], bins=30, kde=True, color='blue')
20. plt.title(col)
21. plt.xlabel(col)
22. plt.ylabel('频率')
23. plt.tight\_layout()
24. plt.show()





### 对于直方图的数据分析在 Lab5\_Code\_Note.ipynb #3.3部分

## 1.3特征和标签的提取、标签的数值编码、数据集的拆分以及标准化的操作

### 提取特征和标签 From Lab5\_Code\_Note.ipynb #3.4

*#3.4*

1. '''
2. 步骤 1：提取特征和标签
3. 首先，我们从数据集中提取特征 (X) 和目标标签 (y)。假设 Label 列是目标变量（即要预测的结果），我们需要将其与其他特征分开。
4. '''
5. *# 提取特征和标签*
6. if 'Label' in df.columns:
7. X = df.drop('Label', axis=1).values  *# 将 'Label' 列从数据集中移除，剩余的是特征*
8. y = df['Label'].values               *# 'Label' 列是目标变量*
9. else:
10. raise ValueError("数据集中未找到 'Label' 列，请检查列名或数据结构。")
11. '''
12. 解释：
13. X 是包含所有特征的数据集（不包括 Label 列）。
14. y 是包含目标标签的数据集（即 Label 列）。
15. 如果数据集中未找到 Label 列，会抛出错误提醒，确保数据的正确性。
16. '''

### 将分类标签转换为数值编码From Lab5\_Code\_Note.ipynb #3.5

*#3.5*

1. '''
2. 将分类标签转换为数值编码
3. 对于大多数机器学习算法来说，标签需要是数值类型。我们使用 LabelEncoder 将标签从文本（如 BENIGN 和 DDoS）转换为数值编码。
4. '''
5. from sklearn.preprocessing import LabelEncoder
6. *# 将分类标签转换为数值编码*
7. le = LabelEncoder()
8. y = le.fit\_transform(y)
9. '''
10. 解释：
11. LabelEncoder 将文本标签转换为数字编码。例如，BENIGN 可能被编码为 0，DDoS 可能被编码为 1。
12. 这一步确保目标变量 (y) 能被机器学习模型正确处理
13. '''

### 数据集拆分为训练集和测试集From Lab5\_Code\_Note.ipynb #3.6

*#3.6*

1. '''
2. 数据集拆分为训练集和测试集
3. 为了评估模型的性能，我们将数据集拆分为训练集和测试集。训练集用于训练模型，测试集用于评估模型的泛化能力。
4. '''
5. from sklearn.model\_selection import train\_test\_split
6. *# 数据集拆分为训练集和测试集*
7. X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)
8. '''
9. 解释：
10. train\_test\_split 函数将数据集按 80% 和 20% 的比例拆分为训练集和测试集。
11. random\_state=42 确保结果是可重复的（每次运行代码时生成相同的拆分结果）。
12. '''

### 数据标准化From Lab5\_Code\_Note.ipynb #3.7

*#3.7*

1. '''
2. 数据标准化
3. 数据标准化是指将所有特征缩放到相同的尺度。这对于许多机器学习算法（如神经网络、支持向量机等）非常重要，因为这些算法对特征的尺度很敏感。
4. '''
5. from sklearn.preprocessing import StandardScaler
6. *# 数据标准化*
7. scaler = StandardScaler()
8. X\_train = scaler.fit\_transform(X\_train)  *# 使用训练集数据计算均值和标准差，并应用于训练集*
9. X\_test = scaler.transform(X\_test)        *# 使用相同的均值和标准差转换测试集*
10. '''
11. 解释：
12. StandardScaler 标准化数据，将数据缩放到均值为 0、方差为 1 的分布。
13. 在训练集中，使用 fit\_transform 计算并应用标准化参数（均值和标准差）。
14. 在测试集中，使用 transform 应用相同的标准化参数（不重新计算均值和标准差）以保持一致性。
15. '''
16. '''
17. 已经准备好了标准化后的训练集和测试集，接下来可以用于模型的训练和评估。标准化后的数据有助于提高模型的训练效率和预测性能。
18. '''

# ANN建模分析

### 概要：使用 Keras 构建一个简单的深度学习模型来预测网络流量是否为正常（如 BENIGN）或攻击（如 DDoS）

## 2.1构建 ANN 模型

### From Lab5\_Code\_Note.ipynb #3.8

*#3.8*

1. '''
2. 构建 ANN 模型
3. 首先，我们使用 Keras 的 Sequential 模型来构建一个多层神经网络。以下是具体的代码及其解释：
4. '''
5. from tensorflow.keras.models import Sequential
6. from tensorflow.keras.layers import Dense, Dropout
7. *# 构建 ANN 模型*
8. model = Sequential()
9. *# 第一层：全连接层，包含 64 个神经元，输入维度等于训练集的特征数量，激活函数使用 ReLU*
10. model.add(Dense(64, input\_dim=X\_train.shape[1], activation='relu'))
11. model.add(Dropout(0.5))  *# 使用 Dropout 正则化，随机丢弃 50% 的神经元，防止过拟合*
12. *# 第二层：全连接层，包含 32 个神经元，激活函数使用 ReLU*
13. model.add(Dense(32, activation='relu'))
14. model.add(Dropout(0.5))  *# 使用 Dropout 正则化*
15. *# 输出层：1 个神经元，使用 sigmoid 激活函数，适用于二分类问题*
16. model.add(Dense(1, activation='sigmoid'))
17. '''
18. 解释：
19. 第一层（输入层和隐藏层）：使用 Dense 层，包含 64 个神经元，激活函数为 ReLU（修正线性单元）。输入维度 (input\_dim) 为 X\_train 的特征数。
20. Dropout 层：每个隐藏层之后加入 Dropout 层，以 50% 的概率随机丢弃神经元，有助于防止过拟合。
21. 第二层（隐藏层）：再次使用 Dense 层，包含 32 个神经元，激活函数为 ReLU。
22. 输出层：使用 Dense 层，包含 1 个神经元，激活函数为 sigmoid，适用于二分类问题（输出范围在 0 到 1 之间）
23. '''

## 2.2编译模型

### From Lab5\_Code\_Note.ipynb #3.9

*#3.9*

1. '''
2. 编译模型
3. 在构建模型后，我们需要编译它，以指定训练时的优化器、损失函数和评估指标。
4. '''
5. *# 编译模型*
6. model.compile(optimizer='adam', loss='binary\_crossentropy', metrics=['accuracy'])
7. '''
8. 解释：
9. 优化器（adam）：Adam 是一种自适应学习率优化算法，通常在深度学习任务中表现良好。
10. 损失函数（binary\_crossentropy）：二分类交叉熵损失函数适用于二分类问题，它衡量了模型预测的概率分布与真实标签的概率分布之间的距离。
11. 评估指标（accuracy）：在每次训练过程中，我们会记录准确率，以便监控模型的表现。
12. '''

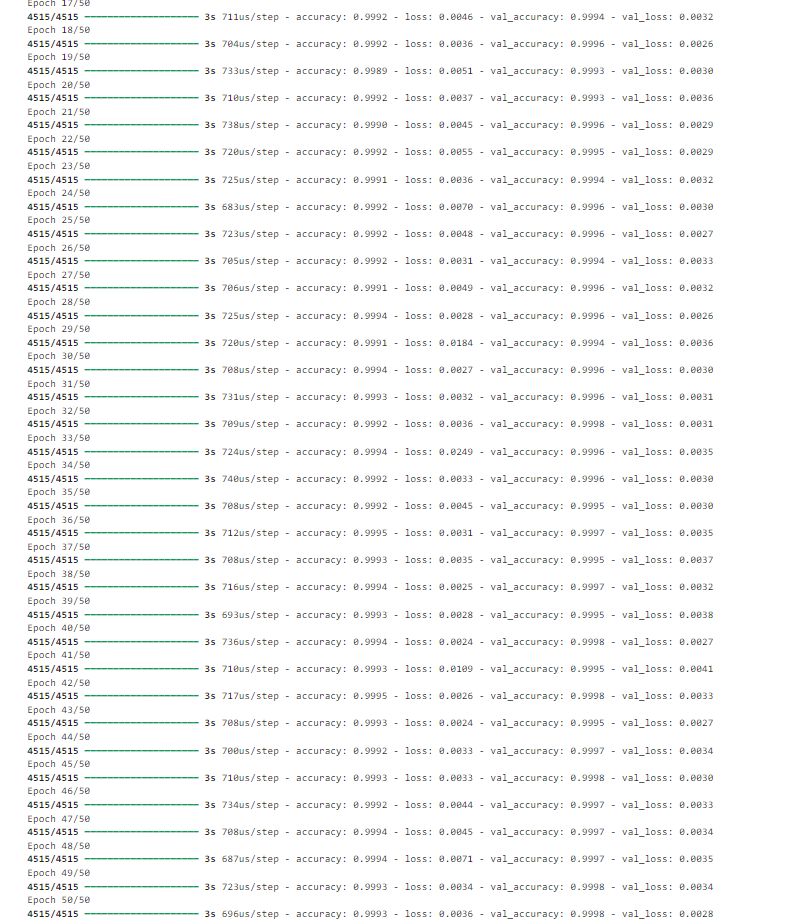
## 2.3 训练模型

### From Lab5\_Code\_Note.ipynb #3.10

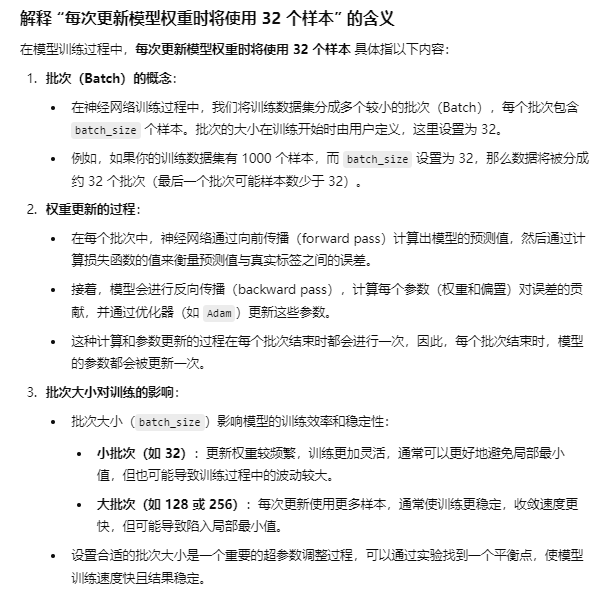
*#3.9*

1. '''
2. 训练模型
3. 接下来，我们使用训练集数据来训练模型。
4. '''
5. *# 训练模型*
6. history = model.fit(X\_train, y\_train, epochs=50, batch\_size=32, validation\_split=0.2)
7. '''
8. 解释：
9. 训练数据：X\_train 和 y\_train。
10. epochs=50：模型将遍历整个训练数据集 50 次。
11. batch\_size=32：每次更新模型权重时将使用 32 个样本。
12. validation\_split=0.2：将训练集中的 20% 数据用于验证，以监控模型在未见数据上的表现（防止过拟合）。
13. '''

### 训练过程截图：



### 问题解答：



## 2.4 模型评估：

### 生成预测值 From Lab5\_Code\_Note.ipynb #3.11

*#3.11*

1. '''
2. 模型评估
3. 生成预测值
4. 在模型评估中，我们首先使用训练好的模型对测试集进行预测。
5. '''
6. from sklearn.metrics import classification\_report, confusion\_matrix
7. *# 生成预测值*
8. y\_pred = (model.predict(X\_test) > 0.5).astype("int32")
9. '''
10. 解释：
11. model.predict(X\_test)：使用模型对测试集 X\_test 进行预测。输出是一个介于 0 和 1 之间的概率值，表示预测为正类（如 1）的概率。
12. (model.predict(X\_test) > 0.5).astype("int32")：将概率值转换为二进制标签。如果概率大于 0.5，则预测为 1，否则为 0。使用 astype("int32") 将结果转换为整型。
13. '''

### 打印分类报告 From Lab5\_Code\_Note.ipynb #3.12

*#3.12*

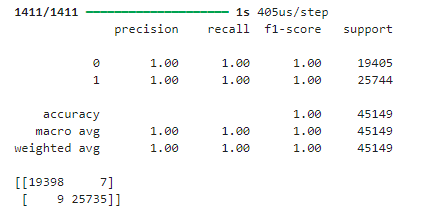
1. '''
2. 模型评估
3. 打印分类报告
4. 使用 classification\_report 函数生成分类报告，它包含了精度（Precision）、召回率（Recall）、F1-得分（F1-score）和支持度（Support）等重要指标。
5. '''
6. *# 打印分类报告*
7. print(classification\_report(y\_test, y\_pred))
8. '''
9. 解释：
10. 精度（Precision）：模型正确预测的正类样本数占所有预测为正类的样本数的比例。它反映了模型在不误判负类样本为正类样本时的准确性。
11. 召回率（Recall）：模型正确预测的正类样本数占所有实际为正类的样本数的比例。它反映了模型在不漏判正类样本时的能力。
12. F1-得分（F1-score）：精度和召回率的调和平均值，用于权衡两者的重要性。F1-得分越高，表示模型的综合表现越好。
13. 支持度（Support）：每个类别在测试集中实际出现的样本数
14. '''

### 打印分类报告 From Lab5\_Code\_Note.ipynb #3.13

*#3.13*

1. '''
2. 模型评估
3. 打印混淆矩阵
4. 混淆矩阵用于显示模型的预测结果与真实标签之间的匹配情况。
5. '''
6. *# 打印混淆矩阵*
7. print(confusion\_matrix(y\_test, y\_pred))
8. '''
9. 解释：
10. 混淆矩阵是一种可视化工具，用于描述分类模型的性能。它显示了真阳性（True Positive, TP）、假阳性（False Positive, FP）、真阴性（True Negative, TN）和假阴性（False Negative, FN）的数量。
11. 混淆矩阵的结构如下：
12. 预测为负类    预测为正类
13. 实际为负类  TN（真阴性）    FP（假阳性）
14. 实际为正类  FN（假阴性）    TP（真阳性）
15. 通过混淆矩阵，你可以直观地看到模型的分类错误类型（如误判率和漏判率）
16. '''

### 混沌矩阵结果：





## 2.5 拓展内容 可视化：验证/训练 准确率和损失函数图像 混淆矩阵图像

### 验证/训练 准确率和损失函数图像 From Lab5\_Code\_Note.ipynb #3.14

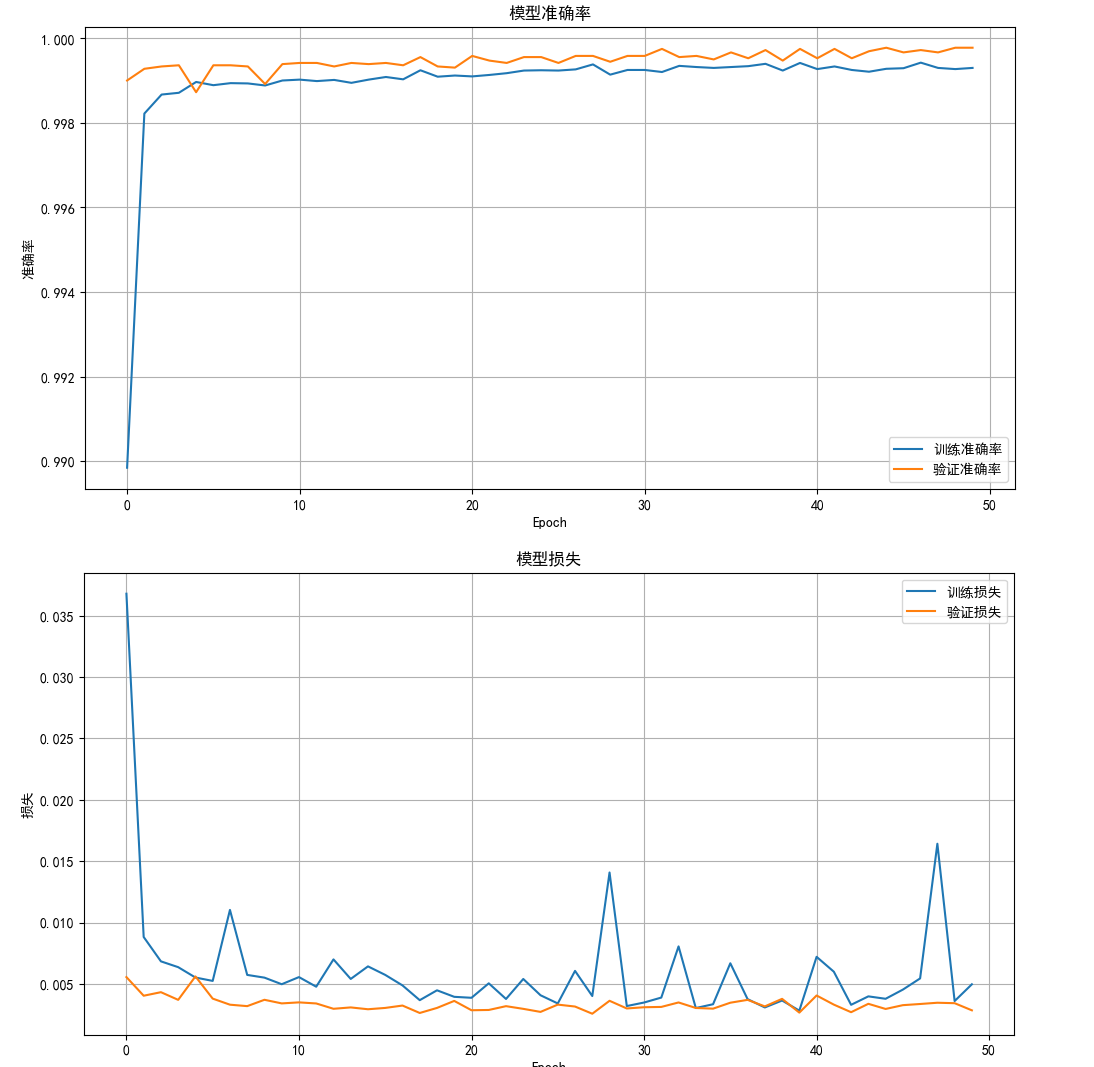
*#3.14 可视化 拓展内容*

1. import matplotlib.pyplot as plt
2. import matplotlib.font\_manager as fm
3. *# 设置字体为SimHei（黑体），以便正常显示中文*
4. plt.rcParams['font.sans-serif'] = ['SimHei']  *# 指定默认字体*
5. plt.rcParams['axes.unicode\_minus'] = False  *# 解决负号显示问题*
6. *# 绘制训练和验证的准确率变化*
7. plt.figure(figsize=(12, 6))
8. plt.plot(history.history['accuracy'], label='训练准确率')
9. plt.plot(history.history['val\_accuracy'], label='验证准确率')
10. plt.title('模型准确率')
11. plt.xlabel('Epoch')
12. plt.ylabel('准确率')
13. plt.legend()
14. plt.grid(True)
15. plt.show()
16. *# 绘制训练和验证的损失变化*
17. plt.figure(figsize=(12, 6))
18. plt.plot(history.history['loss'], label='训练损失')
19. plt.plot(history.history['val\_loss'], label='验证损失')
20. plt.title('模型损失')
21. plt.xlabel('Epoch')
22. plt.ylabel('损失')
23. plt.legend()
24. plt.grid(True)
25. plt.show()

### 混淆矩阵图像 From Lab5\_Code\_Note.ipynb #3.15

1. *#3.15 可视化 拓展内容*
2. from sklearn.metrics import ConfusionMatrixDisplay
3. *# 绘制混淆矩阵*
4. ConfusionMatrixDisplay.from\_predictions(y\_test, y\_pred)
5. plt.title('混淆矩阵')
6. plt.show()

## 可视图：



## 1726488050977

# 拓展 使用 Keras Tuner 进行超参数优化

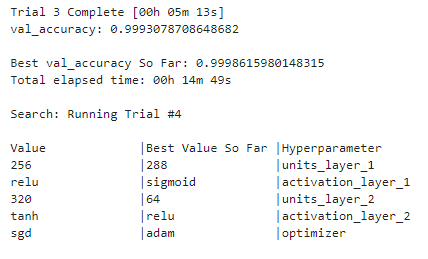
## 3.1导入所需的库 超参数优化的模型构建函数 From Lab5\_Code\_Note.ipynb #3.16

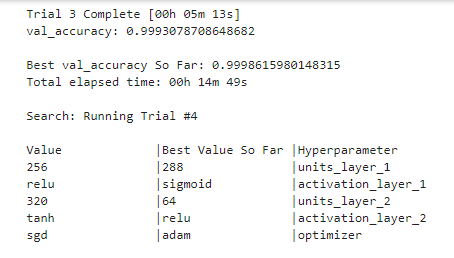
1. *#3.16*
2. '''
3. 导入库和定义模型构建函数
4. '''
5. from keras\_tuner import RandomSearch
6. from tensorflow.keras.models import Sequential
7. from tensorflow.keras.layers import Dense, Input
8. import tensorflow as tf
9. '''
10. keras\_tuner: 这是一个用于超参数搜索的开源库。我们导入了 RandomSearch，它是一种基于随机搜索策略的超参数调优方法。
11. tensorflow.keras.models.Sequential: 导入 Keras 中的 Sequential 模型，这是一个线性堆叠的神经网络模型。
12. tensorflow.keras.layers.Dense 和 tensorflow.keras.layers.Input: 导入了 Keras 的层模块，Dense 用于全连接层，Input 用于定义输入层。
13. tensorflow: TensorFlow 是一个开源的深度学习框架，我们使用它来构建和训练神经网络。
14. '''
15. '''
16. 定义模型构建函数 build\_model
17. Keras Tuner 需要一个模型构建函数来动态构建不同的模型结构，这个函数接受一个 HyperParameters 对象（即 hp）作为参数，以便在模型中使用不同的超参数。
18. '''
19. *# 超参数优化的模型构建函数*
20. def build\_model(hp):
21. model = Sequential()
22. *# 使用 Input 层来定义输入的形状*
23. model.add(Input(shape=(X\_train.shape[1],)))
24. '''
25. model = Sequential()：创建一个 Sequential 模型实例。Sequential 是 Keras 提供的一种简单的顺序模型类型，适合于从输入到输出的线性堆叠层。
26. Input(shape=(X\_train.shape[1],))：添加一个输入层，并定义输入数据的形状，X\_train.shape[1] 表示输入特征的数量。
27. 使用 Input 层显式地定义输入的形状，便于模型的可视化和调试。
28. '''
30. *# 第一层隐藏层，超参数优化层数和激活函数*
31. model.add(Dense(
32. units=hp.Int('units\_layer\_1', min\_value=32, max\_value=512, step=32),  *# 随机选择 32 到 512 个神经元*
33. activation=hp.Choice('activation\_layer\_1', values=['relu', 'tanh', 'sigmoid'])  *# 随机选择激活函数*
34. ))
35. '''
36. Dense：表示一个全连接层（或密集层）。这层中的每个神经元都连接到上一层的所有神经元。
37. units 参数：
38. hp.Int('units\_layer\_1', min\_value=32, max\_value=512, step=32)：这里使用 hp.Int 方法定义了超参数 units\_layer\_1，表示神经元的数量。
39. min\_value=32 和 max\_value=512：表示神经元数量的范围是从 32 到 512，每次增量为 32。Keras Tuner 将在这个范围内随机选择一个值作为该层的神经元数量。
40. activation 参数：
41. hp.Choice('activation\_layer\_1', values=['relu', 'tanh', 'sigmoid'])：这里使用 hp.Choice 方法定义了超参数 activation\_layer\_1，表示激活函数的类型。
42. values=['relu', 'tanh', 'sigmoid']：表示激活函数的选择可以是 ReLU、Tanh 或 Sigmoid。Keras Tuner 将在这些选择中随机挑选一个激活函数来构建模型。
43. '''
45. *# 第二层隐藏层，超参数优化层数和激活函数*
46. model.add(Dense(
47. units=hp.Int('units\_layer\_2', min\_value=32, max\_value=512, step=32),  *# 随机选择 32 到 512 个神经元*
48. activation=hp.Choice('activation\_layer\_2', values=['relu', 'tanh', 'sigmoid'])  *# 随机选择激活函数*
49. ))
50. '''
51. units 和 activation 参数的用法与第一层类似：
52. 这一层同样使用了 hp.Int 和 hp.Choice 方法来定义可调的超参数，Keras Tuner 会在指定的范围和选项内随机选择神经元数量和激活函数。
53. '''
55. *# 输出层，sigmoid 激活函数适用于二分类问题*
56. model.add(Dense(1, activation='sigmoid'))
57. '''
58. Dense：一个全连接输出层。
59. units=1：输出层只有一个神经元，因为我们解决的是一个二分类问题（即正常流量和攻击流量）。
60. activation='sigmoid'：使用 Sigmoid 激活函数，将输出值限制在 0 到 1 之间，表示样本属于某个类别的概率。
61. '''
62. *# 编译模型，优化器、损失函数和评估指标也作为超参数*
63. model.compile(
64. optimizer=hp.Choice('optimizer', values=['adam', 'rmsprop', 'sgd']),  *# 随机选择优化器*
65. loss='binary\_crossentropy',
66. metrics=['accuracy']
67. )
68. '''
69. optimizer 参数：
70. hp.Choice('optimizer', values=['adam', 'rmsprop', 'sgd'])：这里使用 hp.Choice 方法定义了优化器的选择。优化器可以是 Adam、RMSprop 或 SGD，Keras Tuner 将在这些选项中随机选择一个。
71. loss='binary\_crossentropy'：二分类交叉熵损失函数，适用于二分类问题。
72. metrics=['accuracy']：使用准确率（accuracy）作为模型评估的指标。
73. '''
74. return model
75. '''
76. hp.Int：用于选择整数超参数，这里选择的范围是 32 到 512（每步增加 32），表示隐藏层的神经元数量。
77. hp.Choice：用于选择离散值的超参数，这里用于选择激活函数（relu、tanh、sigmoid）和优化器（adam、rmsprop、sgd）。
78. 我们定义了一个可调整的模型架构，Keras Tuner 将通过随机搜索方法来选择最优的超参数组合。
79. '''

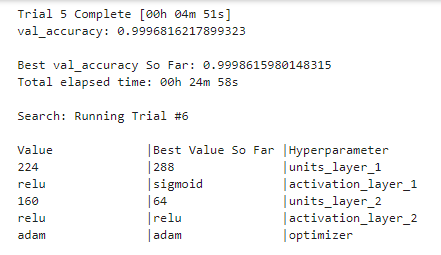
## 3.2开始超参数优化 From Lab5\_Code\_Note.ipynb #3.17

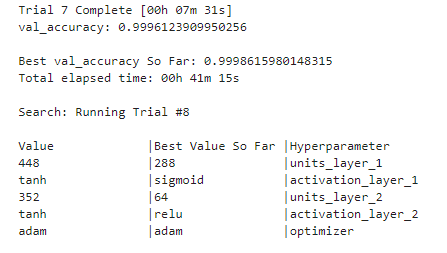
1. *#3.17*
2. *# 使用 Keras Tuner 进行超参数优化*
3. tuner = RandomSearch(
4. build\_model,  *# 模型构建函数*
5. objective='val\_accuracy',  *# 目标是验证集上的准确率*
6. max\_trials=10,  *# 最多尝试 10 次不同的超参数组合*
7. executions\_per\_trial=2  *# 每次试验的模型重复执行两次*
8. )
9. '''
10. build\_model: 模型构建函数，它将使用不同的超参数组合动态创建模型。
11. objective='val\_accuracy': 我们的优化目标是验证集的准确率 (val\_accuracy)。Keras Tuner 将尝试最大化这个指标。
12. max\_trials=10: 指定最多进行 10 次不同的超参数组合尝试。每一次尝试将基于 build\_model 函数构建一个模型。
13. executions\_per\_trial=2: 指定每次试验的模型重复执行两次，以确保结果的稳定性。每个超参数组合的评估结果将取两次试验的平均值。
14. '''
15. *# 搜索最佳模型参数*
16. '''
17. 搜索最佳模型参数
18. 使用 Keras Tuner 的 search 方法来进行超参数优化。
19. '''
20. tuner.search(X\_train, y\_train, epochs=50, validation\_split=0.2)
21. '''
22. X\_train 和 y\_train: 训练数据集的特征和标签。
23. epochs=50: 每个模型训练的最大迭代次数为 50 次。通过这个设置，所有超参数组合的模型都会在 50 个 epochs 内进行训练。
24. validation\_split=0.2: 将训练数据中的 20% 用于验证。Keras Tuner 将使用这部分数据来评估模型在每个 epoch 后的表现，从而找到最优的超参数组合。
25. Keras Tuner 将使用随机搜索（Random Search）策略在指定的超参数空间中进行搜索，以找到最佳的模型配置
26. '''
27. *# 从搜索结果中获得最佳模型*
28. '''
29. 从搜索结果中获得最佳模型
30. 搜索完成后，我们可以获取找到的最佳模型。
31. '''
32. best\_model = tuner.get\_best\_models(num\_models=1)[0]
33. '''
34. tuner.get\_best\_models(num\_models=1): 返回性能最好的 num\_models 个模型。这里我们取第一个模型 ([0])，即最优模型。
35. best\_model: 这个模型是根据验证集上的准确率 (val\_accuracy) 最优的超参数组合所训练出来的模型。
36. '''
37. *# 训练最佳模型*
38. history = best\_model.fit(X\_train, y\_train, epochs=50, batch\_size=32, validation\_split=0.2)
39. '''
40. X\_train 和 y\_train: 训练数据集的特征和标签。
41. epochs=50: 再次训练的最大迭代次数为 50 次。我们可以根据需要调整这个值。
42. batch\_size=32: 每次更新模型权重时使用的样本数。
43. validation\_split=0.2: 将训练数据中的 20% 用于验证，帮助监控训练过程中的模型性能。
44. '''
45. '''
46. 超参数优化: 使用 Keras Tuner 进行超参数搜索，找到了最优的模型配置。
47. 搜索过程: Keras Tuner 基于验证集的准确率来选择最优的超参数组合，通过随机搜索方法，尝试了不同的超参数组合。
48. 最佳模型训练: 在找到的最佳超参数组合的基础上，进一步训练模型，以充分发挥其潜力
49. '''

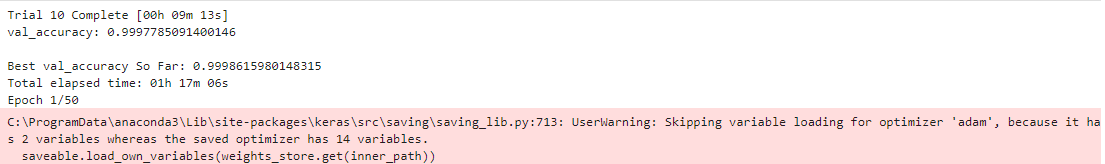
## 经过10次随机后 找到最佳参数：











## 当前最佳的超参数组合（在第二次尝试后）仍然是：

## 第一层神经元数：288

## 第一层激活函数：sigmoid

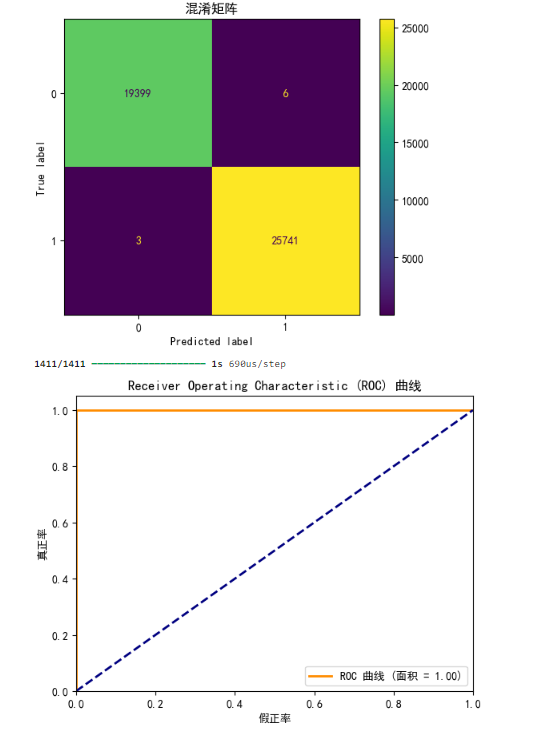
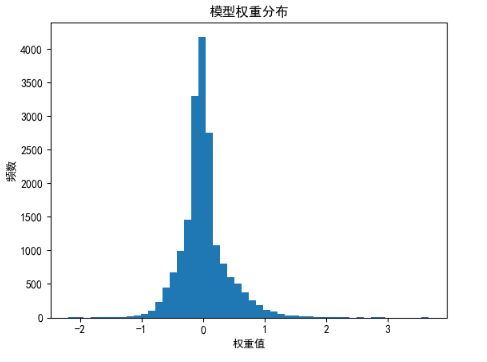
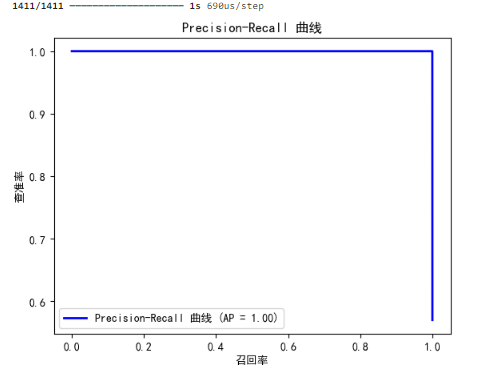
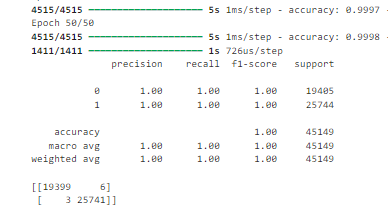
## 第二层神经元数：64

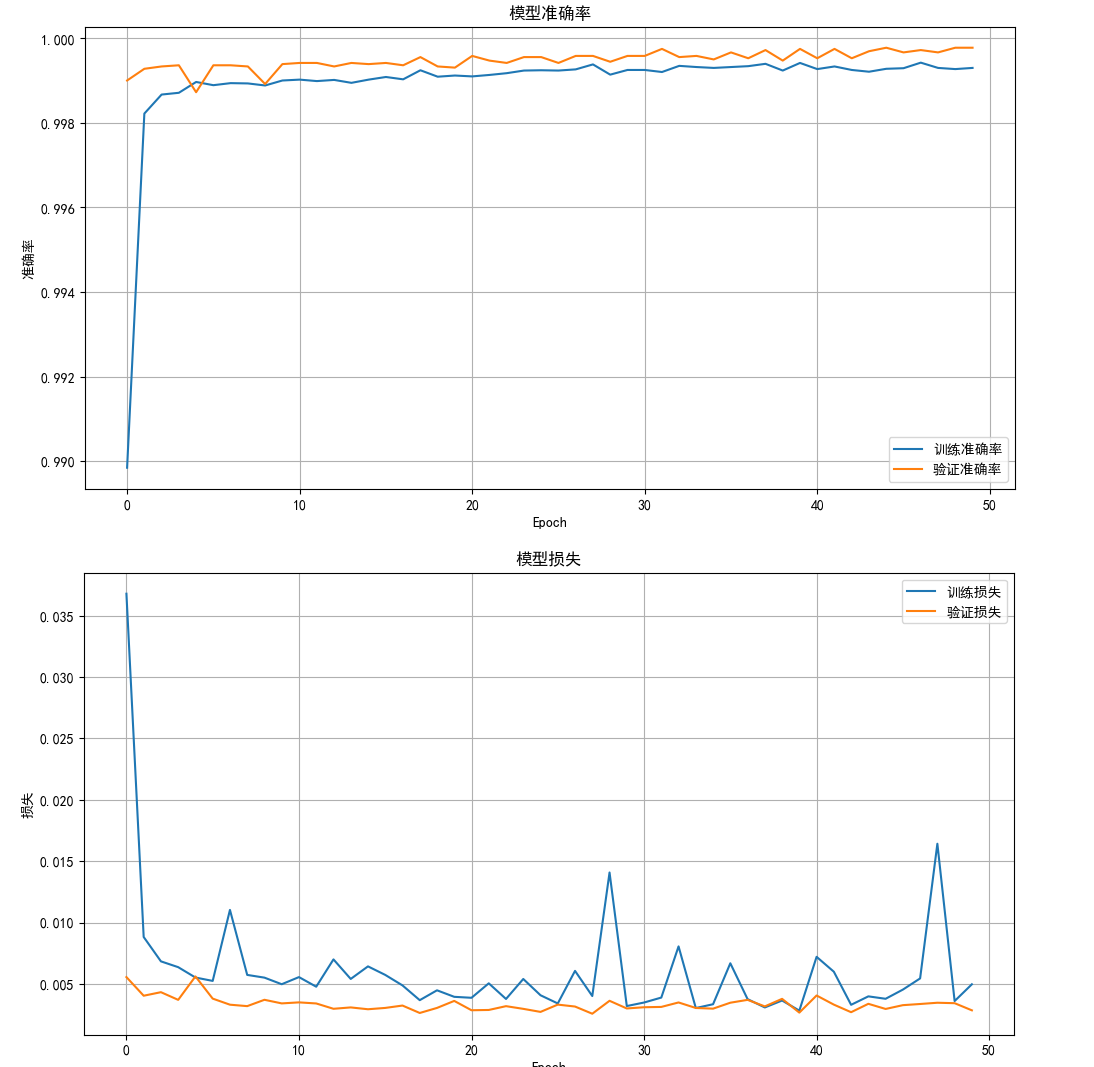
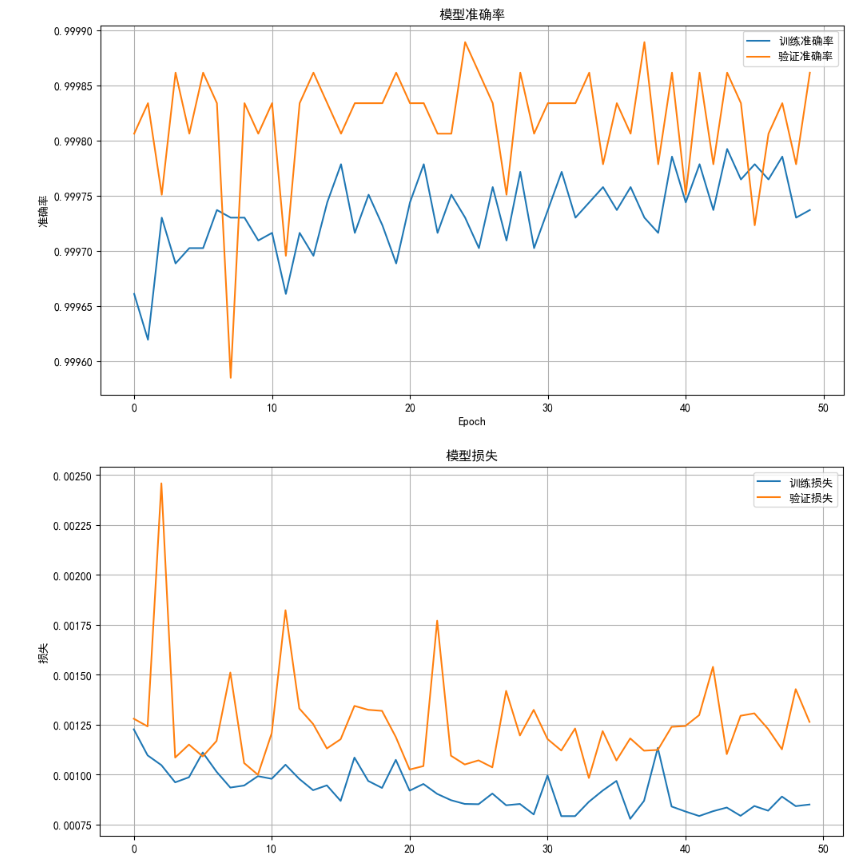
## 第二层激活函数：relu

## 优化器：adam

## 按照最佳参数跑一边后 可视化代码 同 2 章内代码

## 优化后可视化展示：



.

## 超参数优化后（左图） 第二章构建的模型