《人工智能导论》作业

题目： 基于产生式与 CNN 的动物识别方法

院　　系	计算机科学与技术学院
专业班级	
姓　　名	
学　　号	
指导教师	冯琪

计算机科学与技术学院

摘要

本项目围绕动物识别任务，分别实现了基于产生式系统和卷积神经网络(CNN)的两种方法，并对其性能进行了对比分析。产生式系统通过规则推理实现动物特征的逻辑化匹配，而CNN依赖深度学习技术对图像进行自动特征提取和分类。设计过程中，分别对两种方法进行了独立优化，并通过公开动物图像数据集进行实验验证。实验结果显示，产生式系统在小规模、结构化特征数据上具有较高的解释性，而CNN在大规模、多样化数据中表现出更优越的精度与鲁棒性。本研究为不同场景下的动物识别方法选择提供了参考。

关键词：产生式系统；卷积神经网络；动物识别；深度学习；规则推理

目　　录

1 引言

随着人工智能技术的飞速发展，动物识别技术在生态保护、生物多样性研究及农业监测等领域的应用日益广泛。传统基于规则的识别方法依赖于人工设计的逻辑规则，虽然具备良好的解释性和推理能力，但难以处理复杂和非结构化的数据。而以卷积神经网络(CNN)为代表的深度学习方法，凭借其强大的数据驱动特性和非线性学习能力，在大规模图像识别任务中展现了卓越性能。然而，这两类方法在实际应用中的优劣势尚未得到全面分析。

本课程设计旨在基于动物识别任务，分别实现基于产生式系统和CNN的两种方法，并对其性能进行比较分析。产生式系统通过领域知识驱动的逻辑规则实现识别，而CNN通过自动特征提取和深度学习实现分类。实验基于公开动物图像数据集展开，评估两种方法在准确性、效率及应用场景适应性等方面的表现。实验结果表明，产生式系统在结构化特征数据上具备更高的解释性，而CNN在大规模非结构化数据中表现出更优的精度和鲁棒性。

通过对两种方法的比较分析，本研究为动物识别任务中方法选择提供了理论支持和实践参考，助力不同领域对人工智能技术的应用拓展。

2 相关理论基础

2.1 产生式规则

产生式规则是人工智能中最广泛使用的知识表示方法之一，适用于描述问题的推理过程和动态行为。产生式规则的核心是“条件→操作”的逻辑对(IF-THEN规则)，它通过一系列规则的组合，逐步推导出结论或执行操作。一个完整的产生式系统主要由以下三个部分组成：

1. 事实库

事实库是存储系统中已知信息和推理过程中形成中间结论的综合数据库。它包括静态知识(如获取的观测数据或特征)和动态知识(通过规则推理生成的中间或最终结论)。事实库的内容会随着规则的匹配和执行而动态更新，从而推动整个推理过程的进行。

2. 规则集

规则集由一系列产生式规则组成，每条规则定义了一个条件和相应的动作：

左部(前提条件)：描述激活该规则所需满足的条件。

右部(结论或操作)：定义条件满足后将执行的行为或生成的结论。

规则的形式通常表示为：

<产生式> ::= <前提> → <结论>

<前提> ::= <简单条件> | <复合条件>

<结论> ::= <事实> | <操作>

例如，如果动物的毛色为黄色并且体型较小 → 推断该动物可能是金丝猴。

3. 规则解释器(控制器)

规则解释器负责调度规则库与事实库的交互，主要包括以下三个步骤：

(1) 匹配：检查规则的前提是否能与事实库中的数据匹配。

(2) 冲突解决：如果多条规则满足条件，选择其中一个规则执行。

(3) 操作：执行选中的规则，并将生成的结果或动作更新到事实库中。

产生式系统的推理过程通常包括以下步骤：

(1) 初始化事实库，将已知的初始事实存入事实库。

(2) 根据规则库中的规则，匹配前提条件与事实库中的事实。

(3) 执行符合条件的规则，将新生成的结论或操作结果存入事实库。

(4) 检查是否满足问题的解，若未解决则重复匹配和执行过程，直到终止条件满足或规则用尽。

产生式系统的主要优势是规则具有高度的解释性和透明性，适用于结构化特征的推理任务。但其局限性在于处理复杂和大规模数据时效率较低，对规则设计的依赖性较高，难以泛化到非结构化数据场景。

产生式规则作为知识驱动的方法，在传统模式识别任务中表现突出，尤其适合静态、明确的领域知识表达。然而，在大规模数据和动态特征提取的任务中，其能力相对有限，为此引入深度学习方法以进一步提升性能成为必要的研究方向。

2.2 卷积神经网络

卷积神经网络(Convolutional Neural Networks, CNN)是一种深度学习算法，主要用于图像识别和处理领域。CNN的设计模仿了人类视觉系统的工作机制，能够通过层次化的方式对图像中的局部特征和全局模式进行提取与分析，其应用已扩展到视频分析、自然语言处理等多个领域。

CNN的核心思想在于通过局部感知野(Local Receptive Fields)和权值共享(Shared Weights)实现特征提取和数据的高效表示：

局部感知野允许卷积核(滤波器)对输入数据的局部区域进行操作，从而提取诸如边缘、角点等局部特征，同时降低模型的计算需求。

权值共享通过在整个输入数据范围内复用同一个卷积核，大幅减少了模型的参数数量，提升了计算效率，并有助于捕捉数据中的平移不变性。

此外，池化(Pooling)作为一种下采样技术，能够进一步降低数据维度，减轻计算负担，同时保留关键特征。

CNN由多个层次化模块组成，每个模块负责特定的数据处理任务：

1. 卷积层(Convolutional Layer)

通过卷积操作，卷积核在输入数据上滑动执行点积，生成特征图(Feature Map)。引入激活函数(如ReLU)增强模型的非线性表达能力，使得网络能够处理复杂模式。

2. 池化层(Pooling Layer)

通过最大池化(Max Pooling)或平均池化(Average Pooling)减少特征图的尺寸，

同时保留关键信息，增强模型的抗噪能力。

3. 全连接层(Fully Connected Layer)

将前一层的所有激活值连接到每一个神经元，负责全局信息的整合，并实现最终的分类或回归任务。

4. 归一化层与激活层

归一化层(如批归一化，Batch Normalization)加速训练过程，并提高模型的稳定性。激活层应用非线性函数(如Sigmoid、Tanh或ReLU)增强模型表达复杂性的能力。

CNN的训练通过以下步骤完成：

1. **前向传播：**输入数据依次经过卷积、池化、全连接等层，最终输出预测结果。
2. **计算损失：**使用损失函数(如交叉熵损失)计算预测值与真实值之间的差异。
3. **反向传播：**基于梯度下降算法，通过计算梯度更新模型权重。
4. **优化：**利用优化算法(如随机梯度下降或Adam)进一步调整权重，以提高模型性能。

CNN因其强大的特征提取能力，被广泛应用于图像分类、目标检测、人脸识别和视频分析等任务中。例如，在ImageNet大规模图像识别挑战赛中，基于CNN的模型展现了超越传统方法的性能。此外，CNN结构在研究中不断进化，衍生出了深度残差网络(ResNet)、密集连接网络(DenseNet)等变种，这些改进进一步增强了CNN对复杂数据的处理能力。

3　算法步骤

3.1 产生式系统的实现步骤

3.1.1　需求分析

需要明确任务的需求，即利用产生式规则来识别马、鸡、猫、狗这四种动物，即需要建立一组规则，这些规则能够根据输入的动物特征信息，推断出动物的种类。

任务明确要求：

1. 创建一个规则库，包含识别马、鸡、猫、狗的产生式规则。
2. 根据输入的特征，通过规则匹配得出动物种类。
3. 若无规则匹配，则返回“未知动物”的提示。

为了建立准确的规则库，需要首先定义目标动物的典型特征：

1. 马：有蹄、长尾巴、用于骑乘或拉车。
2. 鸡：有羽毛、有喙、会下蛋。
3. 猫：体型小、皮肤柔软、牙齿和爪子锋利。
4. 狗：体型多样、有毛发、牙齿锋利、是人类伴侣。

这些特征是规则的核心条件，后续会以此为基础构建产生式规则。除此之外，任务还要求我们建立一些有关其它动物的规则库 R1-R15，明确需求后我们可以开始建立规则库。

3.1.2 规则库和推理机实现

基于特征信息，规则库设计如下：

```
1.    RULES = [
2.        {"if": {"蹄": True, "长尾巴": True, "骑乘或拉车
   ": True}, "then": "马"},
3.        {"if": {"羽毛": True, "喙": True, "下蛋
   ": True}, "then": "鸡"},
```

```
        {"if": {"体型小": True, "柔软皮肤": True, "锋利牙齿和爪子": True}, "then": "猫"},
        {"if": {"体型多样": True, "毛发": True, "锋利牙齿": True, "人类伴侣": True}, "then": "狗"},
        {"if": {"毛发": True}, "then": "哺乳动物"},
        {"if": {"奶": True}, "then": "哺乳动物"},
        {"if": {"羽毛": True}, "then": "鸟"},
        {"if": {"会飞": True, "下蛋": True}, "then": "鸟"},
        {"if": {"吃肉": True}, "then": "食肉动物"},
        {"if": {"犬齿": True, "爪": True, "眼盯前方": True}, "then": "食肉动物"},
        {"if": {"哺乳动物": True, "蹄": True}, "then": "有蹄类动物"},
        {"if": {"哺乳动物": True, "反刍动物": True}, "then": "有蹄类动物"},
        {"if": {"哺乳动物": True, "食肉动物": True, "黄褐色": True, "暗斑点": True}, "then": "金钱豹"},
        {"if": {"哺乳动物": True, "食肉动物": True, "黄褐色": True, "黑色条纹": True}, "then": "虎"},
        {"if": {"有蹄类动物": True, "长脖子": True, "长腿": True, "暗斑点": True}, "then": "长颈鹿"},
        {"if": {"有蹄类动物": True, "黑色条纹": True}, "then": "斑马"},
        {"if": {"鸟": True, "长脖子": True, "长腿": True, "不会飞": True, "黑白二色": True}, "then": "鸵鸟"},
        {"if": {"鸟": True, "会游泳": True, "不会飞": True, "黑白二色": True}, "then": "企鹅"},
        {"if": {"鸟": True, "善飞": True}, "then": "信天翁"}
    ]
```

接着，建立特征映射字典，它按照 RULES 列表中规则的顺序排列了所有可能用于推理的特征。这个列表简化了特征与规则的对应关系，保持了规则的顺序，并允许动态特征选择和状态跟踪。它不是直接用于推理过程，但它是生成用户界面的一部分，用户通过界面选择特征，而这些特征的选择状态又能被推理系统所使用，以确定最终的动物类型。简而言之，FEATURES 列表是用户界面与推理规则之间的桥梁，使得用户可以通过界面操作来驱动推理逻辑，进而得出结论。

```
FEATURES = [
    "蹄", "长尾巴", "骑乘或拉车",  # 规则 1
    "羽毛", "喙", "下蛋",  # 规则 2
    "体型小", "柔软皮肤", "锋利牙齿和爪子",  # 规则 3
    "体型多样", "毛发", "锋利牙齿", "人类伴侣",  # 规则 4
```

```
        # 规则 5
        "奶",  # 规则 6
        # 规则 7
        "会飞",# 规则 8
        "吃肉",  # 规则 9
        "犬齿", "爪", "眼盯前方",  # 规则 10
        "哺乳动物",  # 规则 11
        "反刍动物",  # 规则 12
        "食肉动物", "黄褐色", "暗斑点",  # 规则 13
        "黄褐色", "黑色条纹",  # 规则 14
        "长脖子", "长腿", "暗斑点",  # 规则 15
        "有蹄类动物", "黑色条纹",  # 规则 16
        "鸟", "长脖子", "长腿", "不会飞", "黑白二色",  # 规则 17
        "会游泳",  # 规则 18
        "善飞",  # 规则 19
    ]
```

最后，编写推理机，负责根据输入的动物特征信息，匹配规则集中的规则，并推断出动物的种类，推理机的工作原理如下

1. **初始化：**FEATURES 是一个包含所有可能特征名称的列表。 RULES 是一个包含规则的列表，每个规则都是一个字典，其中包含条件和结论。
2. **推理过程：**inferanimal 函数接受 features 参数，这是一个字典，包含了用户选择的特征及其状态(True 或 False)。函数内部，首先定义了一个空列表 inferenceprocess，用于记录推理过程中匹配的规则。
3. **规则匹配：**函数遍历 RULES 列表中的每个规则。对于每个规则，函数检查规则中的条件是否与用户选择的特征相匹配。这是通过遍历规则中的条件，并检查每个条件是否在 features 字典中存在且状态为 True 来实现的。如果一个规则的所有条件都匹配，那么该规则被记录在 inferenceprocess 列表中，并且将规则的结论(即动物类型)赋值给变量 matchedanimal。
4. **返回结果：** 如果在遍历完所有规则后，matchedanimal 变量不为空，说明找到了匹配的动物，函数返回匹配的动物类型和推理过程记录。如果没有找到匹配的规则，函数返回 “未知动物” 和一个表示未匹配任何规则的列表。

```
    def infer_animal(self, features):
        inference_process = []  # 用于记录推理过程
        matched_animal = None  # 用于保存匹配的动物

        # 遍历所有规则
```

```
            for rule in RULES:
                rule_conditions = rule["if"]  # 获取规则的条件部分

                # 检查每个条件是否都匹配
                # 如果特征不存在，则默认返回False
                if all(features.get(key, False) == value for key, value in rule_conditions.items()):
                    inference_process.append(f"规则匹配: {rule_conditions} => 结果是 {rule['then']}")
                    # 保存匹配到的动物
                    matched_animal = rule["then"]

            # 如果找到了匹配的动物，返回匹配结果；否则返回"未知动物"
            if matched_animal:
                return matched_animal, inference_process
            else:
                return "未知动物", ["未匹配任何规则"]
```

3.1.3 GUI 界面设计

为了让用户更直观地使用动物识别系统，我们设计了基于 PyQt5 的图形用户界面(GUI)。用户可以通过界面选择动物的特征，触发识别操作，并查看识别结果和推理过程。界面主要分为两部分：左侧为特征选择区，右侧为识别结果与推理过程展示区。

左侧特征选择区包括特征标签和对应的切换按钮，每个按钮表示某个特征是否启用(“是”或“否”)。右侧展示区包括“识别”按钮、识别结果标签和推理过程文本框。用户点击“识别”按钮后，系统根据特征选择触发推理逻辑，并在右侧展示推理过程和最终结果。

界面布局采用水平分割器实现，左侧滚动区域动态加载所有特征按钮，右侧通过标签和文本框展示识别信息。识别功能通过采集用户选择的特征，将其传递到推理机，推理机返回匹配的规则和结果，实时更新到界面。

界面设计注重直观性和扩展性。新增特征时只需更新特征列表和规则库，界面会自动生成对应按钮。整个系统通过简单的交互操作实现了特征选择、动物识别和推理过程展示，增强了系统的易用性和用户体验。

1. 特征选择模块

在左侧区域动态生成了与特征相关的标签和切换按钮。

功能：用户可以通过切换按钮选择特征的状态(“是”或“否”)。默认状态为“否”。

逻辑实现：使用一个字典 featureButtons 将特征名称与按钮状态绑定，点击按钮会触发状态切换事件 toggle_button_state，并实时更新按钮文本。

```
  def toggle_button_state(self):
        # 切换按钮的状态
        sender = self.sender()
        if sender.isChecked():
            sender.setText("是")
        else:
            sender.setText("否")
```

2. 识别功能模块

右侧区域包含一个“识别”按钮，负责收集用户选择的特征，调用推理逻辑，并展示结果和推理过程。

功能：通过点击“识别”按钮，系统将用户选择的特征作为输入，调用 on_recognize 函数触发推理过程。

逻辑实现：系统从 featureButtons 字典中获取特征选择状态，构造输入特征字典，传递给推理函数 infer_animal，并将返回的结果和推理过程显示在界面中。

```
    def on_recognize(self):
        selected_features = {}
        # 获取用户选择的特征
        for feature, button in self.featureButtons.items():
            selected_features[feature] = button.isChecked()
        # 显示推理过程
        animal, inference_process = self.infer_animal(selected
_features)
        # 显示结果
        self.resultLabel.setText(f"识别出的动物是：{animal}")
        # 展示推理过程
        self.procedureTextEdit.clear()  # 清空文本框
        if inference_process:
            for step in inference_process:
                self.procedureTextEdit.append(step)
        else:
            self.procedureTextEdit.append("未匹配任何规则")
```

3. 推理过程与结果展示模块

在右侧区域提供了两个展示组件：

结果展示标签：用于显示识别出的动物类别。结果通过 resultLabel 实时更新，使用蓝色字体以增强视觉效果。

```
        # 结果标签
        self.resultLabel = QtWidgets.QLabel('识别结果将在这里显示', self)
        self.resultLabel.setStyleSheet("font: 14pt; color: blue;")
        rightLayout.addWidget(self.resultLabel)
```

推理过程文本框：用于详细展示规则匹配的推理过程，记录匹配的规则及推导出的结果。如果没有匹配规则，展示“未匹配任何规则”。

```
# 推理过程标签
        self.procedureLabel = QtWidgets.QLabel('推理过程：', self)
        self.procedureLabel.setStyleSheet("font: 14pt;")
        rightLayout.addWidget(self.procedureLabel)
        self.procedureTextEdit = QtWidgets.QTextEdit(self)
        self.procedureTextEdit.setReadOnly(True)
        self.procedureTextEdit.setStyleSheet("font: 12pt;")
        rightLayout.addWidget(self.procedureTextEdit)
```

4. 推理逻辑模块

推理逻辑由 infer_animal 实现，负责将用户选择的特征与规则集进行匹配，返回匹配的动物类别及推理过程。

匹配逻辑：遍历规则集中的每条规则，逐一检查规则条件是否与输入特征完全匹配。如果条件全部满足，则记录匹配的规则并返回结果。

返回结果：若有规则匹配，返回对应的动物类别及匹配过程；若无规则匹配，返回“未知动物”及提示信息。

```
# 推理过程标签
        self.procedureLabel = QtWidgets.QLabel('推理过程：', self)
        self.procedureLabel.setStyleSheet("font: 14pt;")
        rightLayout.addWidget(self.procedureLabel)
        self.procedureTextEdit = QtWidgets.QTextEdit(self)
```

```
        self.procedureTextEdit.setReadOnly(True)
        self.procedureTextEdit.setStyleSheet("font: 12pt;")
        rightLayout.addWidget(self.procedureTextEdit)
```

3.1.4 系统测试

图 3-1 产生式系统的测试结果图

经简单测试，系统的各项功能基本得以实现，由于实现过程相对简陋，没有考虑假言三段论等情形，对于复杂的推理可能无法达到理想效果。

3.2 基于 CNN 的动物识别

3.2.1 数据集选取

由于没有找到同时包含马、鸡、猫、狗这四种动物的数据集，本项目采用了动物识别领域比较经典的 CIFAR-10 数据集，该数据集包含 10 类物体的彩色图像，每类 6000 张图片，共 60000 张图片。动物类别包括狗、青蛙、鸟、马、猫等。

图像尺寸：每张图片为 32×32×3(宽×高×通道)。

数据划分：训练集包含 50000 张图片，测试集包含 10000 张图片。

CIFAR-10 数据集以其丰富的样本和多样的类别广泛用于图像分类任务，是验证 CNN 性能的理想选择。

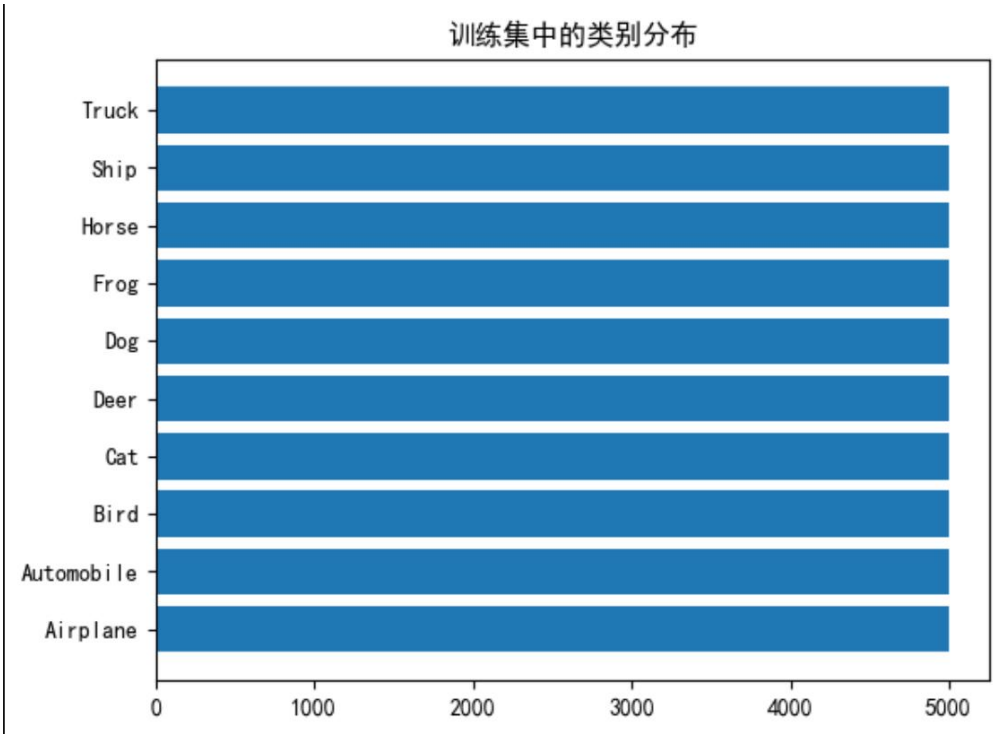

图 3-2 训练集中的类别分布

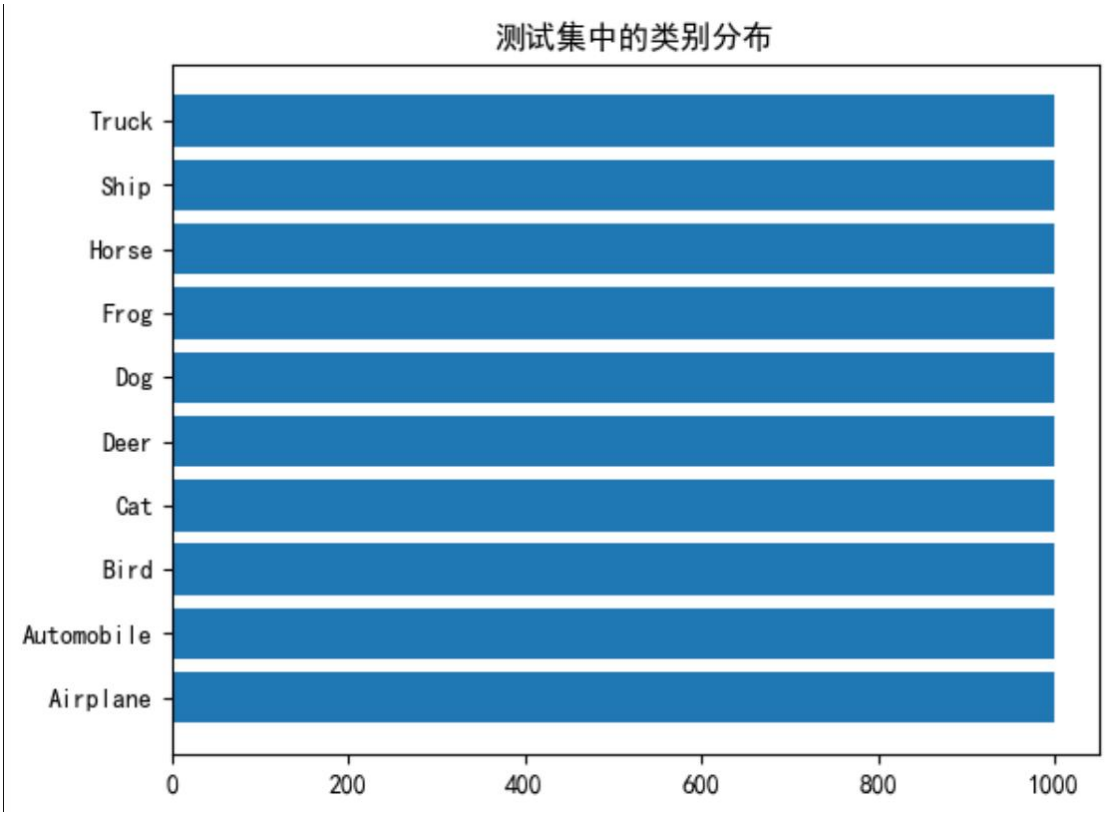

图 3-3　测试集中的类别分布

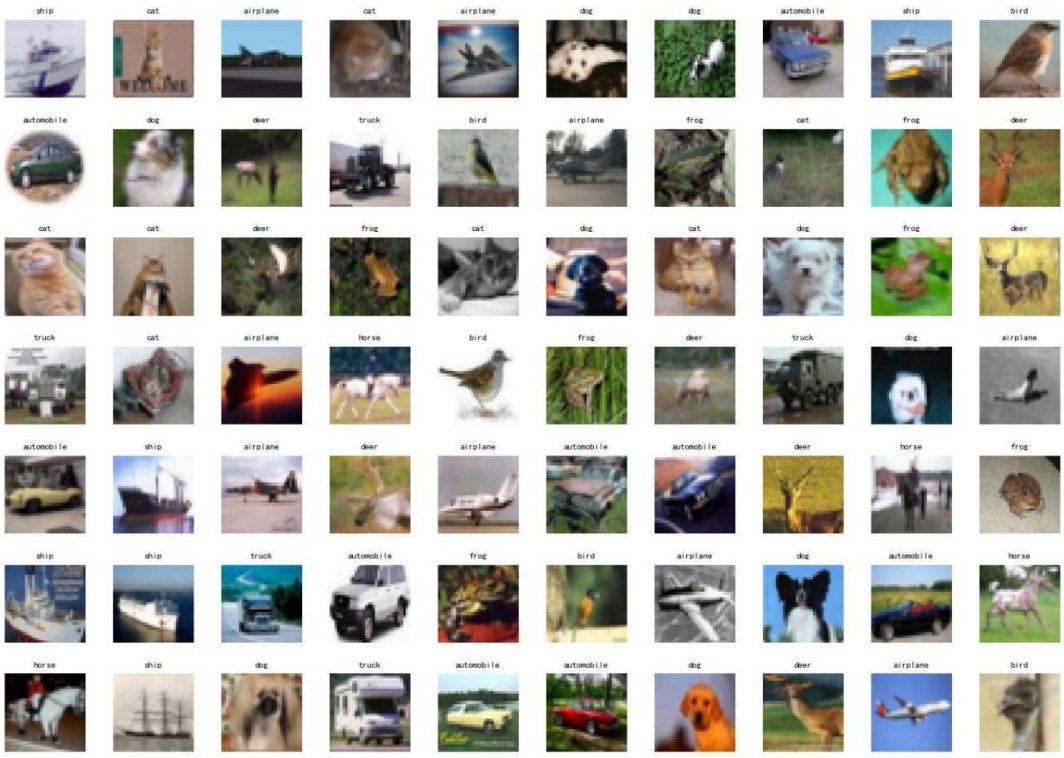

图 3-4　训练集部分图片展示

3.2.2 模型构建与实现

该模型是一个用于 CIFAR-10 图像分类任务的卷积神经网络(CNN)。CIFAR-10 数据集包含 60,000 张 32x32 像素的彩色图像，分为 10 个类别，每个类别有 6,000 张图像。模型通过多层卷积和池化操作提取图像特征，结合批归一化和 Dropout 机制提高训练的稳定性和泛化能力，最终通过全连接层进行分类。以下是详细层级解析。

1. 输入层

```
INPUT_SHAPE=(32,32,3)
KERNEL_SIZE=(3,3)
```

解释：

- INPUT_SHAPE=(32,32,3)：定义输入图像的形状为 32x32 像素，具有 3 个颜色通道(RGB)，这正好符合 CIFAR-10 数据集的图像规格。
- KERNEL_SIZE=(3,3)：卷积核的大小为 3x3，是常用的卷积核尺寸，能够有效捕捉图像的局部特征。

2. 第一组卷积层与批归一化层

```
model.add(Conv2D(filters=32,kernel_size=KERNEL_SIZE,input_shape=INPUT_SHAPE,activation='relu',padding='same'))
model.add(BatchNormalization())
model.add(Conv2D(filters=32,kernel_size=KERNEL_SIZE, activation='relu', padding='same'))
model.add(BatchNormalization())
```

(1) Conv2D 层：

- filters=32：使用 32 个 3x3 的卷积核，提取图像的初级特征，如边缘和颜色变化
- activation='relu'：使用 ReLU 激活函数，引入非线性，增强模型表达能力。
- padding='same'：采用填充方式，使输出的宽高与输入相同，避免特征图尺寸缩小
- input_shape=INPUT_SHAPE：仅在第一层需要指定输入形状。

(2) BatchNormalization 层：

- 对卷积层的输出进行批归一化，稳定和加速训练过程，减少内部协变量偏移(Internal Covariate Shift)

3. 第一组池化层与 Dropout 层

```
model.add(MaxPool2D(pool_size=(2,2)))
model.add(Dropout(0.25))
```

(1) MaxPool2D 层：

- pool_size=(2，2)：使用 2x2 的最大池化窗口，减小特征图的尺寸(降采样)，减少计算量，同时提取主要特征。

(2) Dropout 层：

- Dropout(0.25)：以 25%的概率随机断开部分神经元，防正过拟合，提高模型的泛化能力。

4. 第二组卷积层与批归一化层

```
model.add(Conv2D(filters=64,kernel_size=KERNEL_SIZE,activation
='relu', padding='same'))
model.add(BatchNormalization())
model.add(Conv2D(filters=64,kernel_size=KERNEL_SIZE,activation
='relu', padding='same'))
model.add(BatchNormalization())
```

(1) Conv2D 层：

- filters=64：卷积核数量增加到 64，提取更复杂和高层次的特征。

其他参数与第一组卷积层相同。

(2) BatchNormalization 层：

- 同样进行批归一化，保持训练的稳定性

5. 第二组池化层与 Dropout 层

```
model.add(MaxPool2D(pool_size=(2,2)))
model.add(Dropout(0.25))
```

(1) MaxPool2D 层：

- 同第一组池化层，进一步减小特征图尺寸。

(2) Dropout 层：

- 同第一组 Dropout 层，继续防止过拟合。

6. 第三组卷积层与批归一化层

```
model.add(Conv2D(filters=128,kernel_size=KERNEL_SIZE,activatio
n='relu',padding='same'))
model.add(BatchNormalization())
model.add(Conv2D(filters=128,kernel_size=KERNEL_SIZE,activatio
n='relu',padding='same'))
model.add(BatchNormalization())
```

(1) Conv2D 层：

- filters=128：卷积核数量进一步增加到 128，提取更加丰富和抽象的特征。
- 其他参数与前述卷积层相同。

(2) BatchNormalization 层：

- 保持一致的批归一化，确保训练过程的稳定性。

7. 第三组池化层与 Dropout 层

```
model.add(MaxPool2D(pool_size=(2,2)))
model.add(Dropout(0.25))
```

(1) MaxPool2D 层：

- 继续减小特征图尺寸，提取主要特征，

(2) Dropout 层：

- 防止模型在深层特征提取后的过拟合。

8. 展平层与全连接层

```
model.add(Flatten())
#model.add(Dropout(0.2))
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.25))
model.add(Dense(10,activation='softmax'))
```

(1) Flatten 层：

- 将多维的特征图展平成一维向量，作为全连接层的输入

(2) Dense 层：

- Dense(128，activation='relu')：全连接层，有 128 个神经元，使用 ReLU 激活函数，进一步组合和学习特征
- Dense(10，activation='softmax')：输出层，有 10 个神经元，对应 CIFAR-10 的 10 个类别，使用 Softmax 激活函数输出每个类别的概率分布。

(3) Dropout 层：

- Dropout(0.25)：在全连接层之间使用 Dropout，继续防止过拟合，增强模型的泛化能力。

9. 编译模型

```
METRICS=['accuracy',tf.keras.metrics.Precision(name='precision
'),tf.keras.metrics.Recall(name='recall')]
model.compile(loss='categorical_crossentropy',optimizer='adam',
metrics=METRICS)
```

(1) 损失函数(loss)：

- categorical_crossentropy：适用于多分类问题的交叉熵损失函数，衡量预测概率分布与真实分布之间的差异。

(2) 优化器(optimizer)：

- adam：一种自适应学习率优化算法，结合了动量和 RMSProp 的优点，能够高效地处理大规模数据和参数。

(3) 评估指标(metrics)：

- accuracy：准确率，衡量正确分类的比例
- Precision：精确率，衡量预测为某类别的样本中实际为该类别的比例
- Recall：召回率，衡量实际为某类别的样本中被正确预测为该类别的比例

模型设计的考虑因素

1.输入适配 CIFAR-10:

CIFAR-10 的图像尺寸为 32x32，3 个颜色通道(RGB)，模型的输入层完全匹配这一规格

2.多层卷积与特征提取：

通过多层卷积层，逐步提取图像的低级到高级特征。初级层提取边缘和颜色变化，深层提取更复杂的形状和对象特征。

3.批归一化：

每个卷积层后使用批归一化，有助于稳定和加速训练过程，减少对学习率的依赖，防止梯度消失或爆炸

4.池化层：

最大池化层用于逐步减小特征图的尺寸，减少计算量，同时保留最重要的特征，增强模型的平移不变性

5.Dropout 层：

在池化层和全连接层之间使用 Dropout，防止模型过拟合，提高泛化能力，特别是在 CIFAR-10 这种相对较小的数据集上尤为重要。

6.全连接层与输出层：

展平后的特征通过全连接层进一步组合和学习，最后通过 Softmax 输出各类别的概率分布，适用于 CIFAR-10 的 10 分类任务，

7.优化与评估：

使用 Adam 优化器能够高效地处理大规模数据，并适应不同参数的更新速度。

选择多种评估指标(准确率、精确率、召回率)有助于全面评估模型的性能，尤其是类别不平衡或需要更详细性能分析时。

3.2.3 实验步骤

1. 导入和配置模块

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import tensorflow as tf
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, MaxPool2D,
Flatten, Dropout, BatchNormalization
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.preprocessing.image import ImageDataGene
rator
from sklearn.metrics import ConfusionMatrixDisplay
from sklearn.metrics import classification_report, confusion_m
atrix
plt.rcParams['font.family'] = 'SimHei'  # 使用 SimHei 字体，支持
中文
plt.rcParams['axes.unicode_minus'] = False  # 解决负号显示问题
```

2. 加载数据

```
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
print(f"X_train shape: {X_train.shape}")
print(f"y_train shape: {y_train.shape}")
print(f"X_test shape: {X_test.shape}")
print(f"y_test shape: {y_test.shape}")
```

3. 数据可视化

```
# 定义数据集的标签
labels = ['airplane', 'automobile', 'bird', 'cat', 'deer',
          'dog', 'frog', 'horse', 'ship', 'truck']
# 以网格格式显示更多图像
# 定义绘图网格的维度
W_grid = 10
L_grid = 10
```

```
# fig, axes = plt.subplots(L_grid, W_grid)
# subplot 返回图形对象和坐标轴对象
# 我们可以使用坐标轴对象在不同位置绘制特定的图形
fig, axes = plt.subplots(L_grid, W_grid, figsize = (17,17))
axes = axes.ravel() # 将15 x 15 矩阵展平为225 个数组
n_train = len(X_train) # 获取训练数据集的长度
# 从0 到n_train 之间随机选择一个数字
for i in np.arange(0, W_grid * L_grid): # 创建均匀分布的变量
    # 选择一个随机数字
    index = np.random.randint(0, n_train)
    # 使用选择的索引读取并显示图像
    axes[i].imshow(X_train[index,1:])
    label_index = int(y_train[index])
    axes[i].set_title(labels[label_index], fontsize = 8)
    axes[i].axis('off')
plt.subplots_adjust(hspace=0.4)
```

4. 数据预处理

```
# 数据归一化处理
X_train = X_train / 255.0
X_test = X_test / 255.0
# 将目标变量转换为独热编码
y_cat_train = to_categorical(y_train, 10)
y_cat_test = to_categorical(y_test, 10)
```

5. 模型构建

```
INPUT_SHAPE = (32, 32, 3)
KERNEL_SIZE = (3, 3)
model = Sequential()
# 卷积层
model.add(Conv2D(filters=32, kernel_size=KERNEL_SIZE, input_sh
ape=INPUT_SHAPE, activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(Conv2D(filters=32, kernel_size=KERNEL_SIZE, activati
on='relu', padding='same'))
model.add(BatchNormalization())
# 池化层
model.add(MaxPool2D(pool_size=(2, 2)))
```

```
# Dropout 层
model.add(Dropout(0.25))
model.add(Conv2D(filters=64, kernel_size=KERNEL_SIZE, activati
on='relu', padding='same'))
model.add(BatchNormalization())
model.add(Conv2D(filters=64, kernel_size=KERNEL_SIZE, activati
on='relu', padding='same'))
model.add(BatchNormalization())
model.add(MaxPool2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Conv2D(filters=128, kernel_size=KERNEL_SIZE, activat
ion='relu', padding='same'))
model.add(BatchNormalization())
model.add(Conv2D(filters=128, kernel_size=KERNEL_SIZE, activat
ion='relu', padding='same'))
model.add(BatchNormalization())
model.add(MaxPool2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
# model.add(Dropout(0.2))
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.25))
model.add(Dense(10, activation='softmax'))
METRICS = [
    'accuracy',
    tf.keras.metrics.Precision(name='precision'),
    tf.keras.metrics.Recall(name='recall')
]
model.compile(loss='categorical_crossentropy', optimizer='adam
', metrics=METRICS)
```

6. 早停

```
early_stop = EarlyStopping(monitor='val_loss', patience=2)
```

使用 Keras 库中的 EarlyStopping 回调函数来设置早停(Early Stopping)策略，，当连续两个 epoch 验证集的损失没有改善时，训练过程将会提前终止。这样做可以防止模型过拟合，即模型在训练数据上表现很好，但在未见过的数据(如验证集或测试集)上表现不佳。通过早停，可以节省训练时间，并且通常能获得更好的泛化性能。

7. 数据增强训练

```
batch_size = 32  # 设置批次大小为 32
# 创建数据增强生成器，进行图像的宽度平移、长度平移和水平翻转
data_generator = ImageDataGenerator(width_shift_range=0.1, hei
ght_shift_range=0.1, horizontal_flip=True)
# 使用数据增强生成器生成训练数据
train_generator = data_generator.flow(X_train, y_cat_train, ba
tch_size)
# 计算每个 epoch 的步骤数
steps_per_epoch = X_train.shape[0] // batch_size
# 训练模型
r = model.fit(train_generator,
              epochs=50,  # 训练 50 个 epoch
              steps_per_epoch=steps_per_epoch,  # 每个 epoch 执
行的步数
              validation_data=(X_test, y_cat_test),  # 使用测试
集进行验证
#               callbacks=[early_stop],  # 可以添加早停回调(未启
用)
#               batch_size=batch_size,  # 批次大小(此处未使用)
             )
```

3.2.4 模型评估与分析

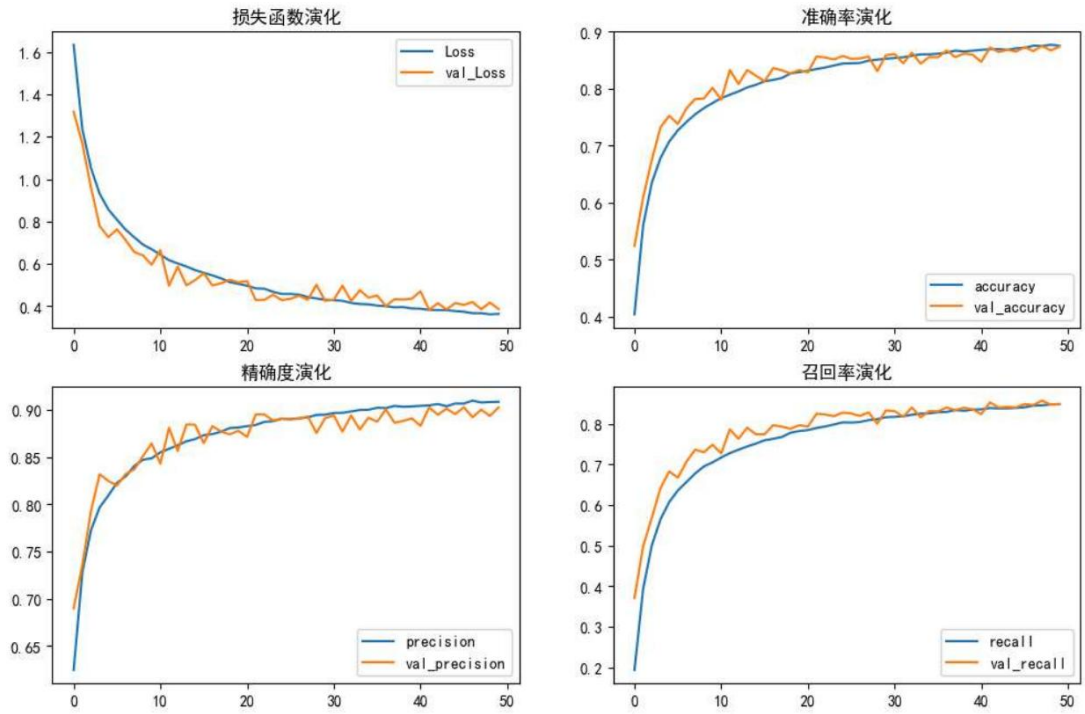

图 3-5 演化图

1. 训练过程图像分析

(1) 损失函数演化：

从训练和验证损失函数的变化曲线可以看出，模型在训练过程中损失值逐渐下降，且验证集的损失值与训练集的损失值趋于一致。这表明模型的拟合情况较好，没有出现严重的过拟合。

(2) 准确率演化：

训练集和验证集的准确率逐渐上升，并趋于稳定。最终的验证准确率接近87.41%，与训练集准确率 87.57% 非常接近，这进一步表明模型的泛化能力较好。

(3) 精确度(Precision)演化：

训练集和验证集的精确度曲线一致性较高，验证精确度达到 90.26%，说明模型对预测类别的置信度较高，预测结果中正类的准确性较好。

(4) 召回率(Recall)演化：

训练集和验证集的召回率稳定上升，验证集召回率达到 84.87%，表明模型对正类样本的覆盖能力较好。

2. 混淆矩阵分析

混淆矩阵显示了模型在每个类别上的预测情况(真实标签 vs. 预测标签)。

对于类别 "airplane" 和 "automobile"，模型的正确分类数量较高，误分类较少，说明模型对这两个类别的识别效果非常好。

类别 "cat" 和 "dog" 的误分类较多，这可能是因为它们在 CIFAR-10 数据集中具有较高的视觉相似性。

类别 "bird" 和 "cat" 之间的混淆较多，可能是因为训练集中的特征没有完全区分这两个类别的模式。

类别 "ship" 和 "truck" 的分类效果较好，模型对这些类别的特征学习较充分。

误分类改进建议：

针对误分类较多的类别(如 "cat" 和 "dog")，可以通过数据增强(如旋转、翻转、裁剪等)或增加更多具有挑战性的样本来改进模型的区分能力。

增加混淆类别之间的样本权重，重点优化模型在这些类别上的性能。

```
evaluation = model.evaluate(X_test, y_cat_test)  # 评估模型在测试集上的表现
print(f'测试集准确率 : {evaluation[1] * 100:.2f}%')  # 输出测试集准确率
y_pred = model.predict(X_test)  # 使用模型对测试集进行预测
y_pred = np.argmax(y_pred, axis=1)  # 获取每个样本预测的类别标签(最大概率对应的类别)
cm = confusion_matrix(y_test, y_pred)  # 计算混淆矩阵
disp = ConfusionMatrixDisplay(confusion_matrix=cm,
                              display_labels=labels)  # 创建混淆矩阵的可视化对象
# 绘制混淆矩阵
fig, ax = plt.subplots(figsize=(10, 10))  # 设置画布大小
disp = disp.plot(xticks_rotation='vertical', ax=ax, cmap='summer')  # 绘制混淆矩阵，设置标签旋转和颜色映射
plt.show()  # 显示混淆矩阵图
```

```
313/313 [==============================] - 6s 21ms/step - loss: 0.3860 - accuracy: 0.8741 - precision: 0.9026 - recall: 0.8487
测试集准确率 : 87.41%
313/313 [==============================] - 6s 20ms/step
```

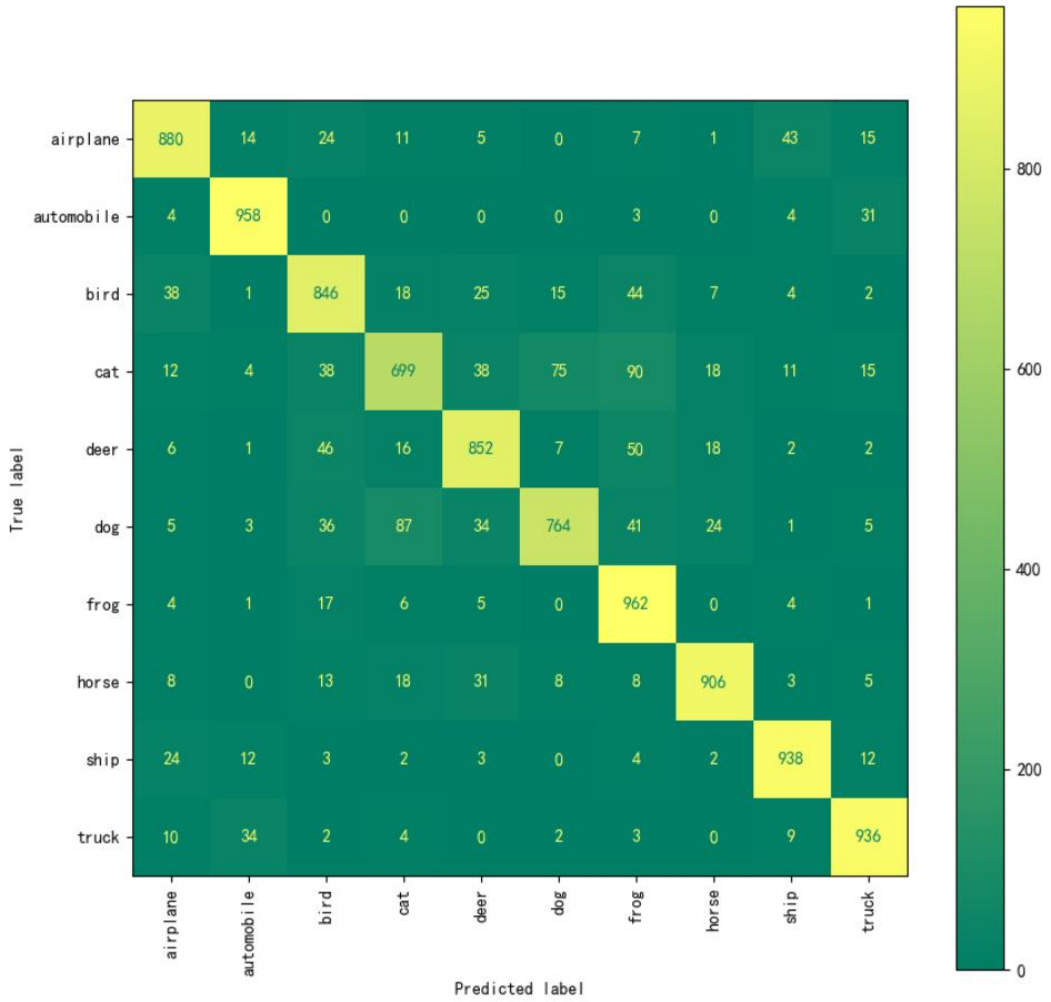

图 3-6 混淆矩阵图

表 3-1 分类性能表

	precision	recall	f1-score	support
0	0.89	0.88	0.88	1000
1	0.93	0.96	0.94	1000
2	0.83	0.85	0.84	1000
3	0.81	0.70	0.75	1000
4	0.86	0.85	0.85	1000
5	0.88	0.76	0.82	1000
6	0.79	0.96	0.87	1000
7	0.93	0.91	0.92	1000
8	0.92	0.94	0.93	1000
9	0.91	0.94	0.92	1000
accuracy	0.87	0.87	0.87	10000

macro avg	0.87	0.87	0.87	10000
weighted avg	0.87	0.87	0.87	10000

- **总体性能**

- 准确率(Accuracy)：模型的总体准确率为 87%，说明在 10,000 张测试样本中，大约有 8,700 张被正确分类。
- 宏平均(Macro Avg)和 加权平均(Weighted Avg)：宏平均(Macro Avg)分别为87%(精确度)、87%(召回率)、87%(F1 分数)，说明模型在不同类别上的性能较为均衡。加权平均与宏平均一致，表明类别分布对整体性能没有明显偏斜。

- **类别性能：**

以下是对各类别分类性能的具体分析：

(1) 类别 0(airplane)：

精确度：89%

召回率：88%

F1 分数：88%

表现：模型对 "airplane" 分类的性能稳定，预测的置信度和覆盖率较高，误分类较少。

(2) 类别 1(automobile)：

精确度：93%

召回率：96%

F1 分数：94%

表现：模型对 "automobile" 分类的表现最好，召回率接近 96%，说明几乎所有的 "automobile" 图像都能被正确识别。

(3) 类别 2(bird)：

精确度：83%

召回率：85%

F1 分数：84%

表现：模型对 "bird" 的预测略显不足，但精确度和召回率基本持平，说明在该类别上的误分类主要集中在少量的非目标类别。

(4) 类别 3(cat)：

精确度：81%

召回率：70%

F1 分数：75%

表现："cat" 类别的表现最差，尤其是召回率仅为 70%，说明有较多的 "cat" 被误分类为其他类别(可能与 "dog" 混淆较多)。

(5) 类别 4(deer)：

精确度：86%

召回率：85%

F1 分数：85%

表现：对 "deer" 的预测性能稳定，精确度和召回率接近。

(6) 类别 5(dog)：

精确度：88%

召回率：76%

F1 分数：82%

表现：对 "dog" 的召回率偏低，说明有较多的 "dog" 被误分类为其他类别，可能与 "cat" 和其他哺乳动物类别(如 "horse")混淆。

(7) 类别 6(frog)：

精确度：79%

召回率：96%

F1 分数：87%

表现："frog" 的召回率表现最好，但精确度稍低，说明模型对 "frog" 的分类倾向于预测为正类，但有一些预测可能是误判。

(8) 类别 7(horse)：

精确度：93%

召回率：91%

F1 分数：92%

表现：对 "horse" 分类的表现非常好，召回率和精确度均较高，说明分类稳定。

(9) 类别 8(ship)：

精确度：92%

召回率：94%

F1 分数：93%

表现：对 "ship" 的分类效果非常好，误分类极少。

(10)类别 9(truck)：

精确度：91%

召回率：94%

F1 分数：92%

表现：对 "truck" 的分类也较为优异，与 "automobile" 类别的性能接近。

- **总体表现观察**

模型对大部分类别的分类表现良好，特别是 "automobile"、"ship"、"horse" 和 "truck" 类别，这些类别的 F1 分数均在 92%-94% 之间。

表现较差的类别主要是 "cat" 和 "dog"，尤其是 "cat"，其召回率仅为 70%，说明很多 "cat" 被误分类为其他类别。

- **改进建议**

为了提升模型的分类性能，尤其是对 "cat" 和 "dog" 类别的表现，可以考虑以下改进措施：

(1) **数据增强：**对 "cat" 和 "dog" 类别使用更多的数据增强技术(如旋转、裁剪、颜色抖动等)，增加模型的泛化能力，特别是对容易混淆的特征进行优化。

(2) **类别权重调整：**在损失函数中对 "cat" 和 "dog" 设置更高的类别权重，以减少误分类对模型训练的影响。

(3) **模型优化：**尝试更复杂的模型架构(如 ResNet、EfficientNet 或加入 Attention 机制)，提升模型对复杂类别的特征提取能力。

(4) **样本增加：**如果可能，收集更多带有挑战性的样本，尤其是容易混淆的类别(如 "cat" 和 "dog")。

(5) **混淆矩阵分析：**通过混淆矩阵观察 "cat" 和其他类别(如 "dog")的具体混淆情况，进一步针对性优化模型。

- 总结

模型总体准确率为 87%，在 CIFAR-10 数据集上表现良好，分类性能较为均衡。需要特别关注 "cat" 和 "dog" 类别的优化，通过数据增强、权重调整等方法，提升模型对这些类别的分类性能。在 "automobile"、"ship" 等类别上，模

型的表现已经非常优异，无需进一步调整。

3.2.5 模型测试

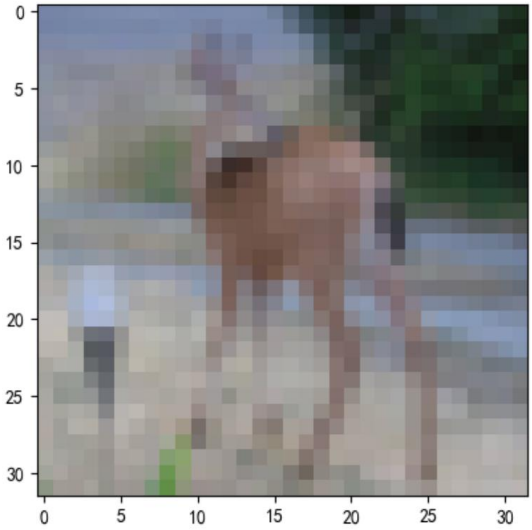

图 3-7 测试用例图

```
my_image = X_test[100]
plt.imshow(my_image)
# that's a Deer
print(f" Image 100 is {y_test[100]}")
# correctly predicted as a Deer
pred_100 = np.argmax(model.predict(my_image.reshape(1, 32, 32,
 3)))
print(f"The model predict that image 100 is {pred_100}")
```

测试结果：

```
 Image 100 is [4]
1/1 [==============================] - 0s 30ms/step
The model predict that image 100 is 4
```

可见，模型成功识别对象为鹿。

接下来，对测试集进行预测：

```
predictions = model.predict(X_test)  # 使用模型对测试集进行预测
# 绘制前 num_images 张测试图像、它们的预测标签和真实标签。
# 正确的预测用蓝色标记，错误的预测用红色标记。
num_rows = 8  # 设置图像的行数
num_cols = 5  # 设置图像的列数
num_images = num_rows * num_cols  # 总共展示的图像数量
```

```
plt.figure(figsize=(2 * 2 * num_cols, 2 * num_rows))  # 设置图形的尺寸
for i in range(num_images):
    # 绘制每个图像及其预测标签
    plt.subplot(num_rows, 2 * num_cols, 2 * i + 1)
    plot_image(i, predictions[i], y_test, X_test)  # 绘制图像和标签
    # 绘制每个图像的预测概率分布
    plt.subplot(num_rows, 2 * num_cols, 2 * i + 2)
    plot_value_array(i, predictions[i], y_test)  # 绘制预测概率分布
plt.tight_layout()  # 自动调整子图间距
plt.show()  # 显示图形
```

至此，我们已完成全部工作，保存模型即可。

```
from tensorflow.keras.models import load_model
model.save('cnn_20_epochs.h5')
```

3.3 产生式系统与深度学习方法的比较

本研究分别构建了基于产生式系统和卷积神经网络（CNN）的动物识别模型，并通过实验验证了两种模型的有效性。本部分将对两种模型在动物识别任务中的优劣进行对比分析，并探讨其在不同场景下的适用性。

1. 产生式系统

(1)优点：

① 自然性： 知识表示形式为因果关系，直观自然，便于推理。

② 模块性： 规则是最基本的知识单元，形式相同，易于模块化。

③ 有效性： 可表示确定性、不确定性、启发性和过程性知识。

④ 清晰性： 固定格式便于规则设计，易于进行一致性和完整性检测。

(2)缺点：

① 效率不高：求解过程需要反复进行“匹配-冲突消解-执行”循环。规则库通常较大，匹配过程耗时。求解复杂问题时容易引发组合爆炸。

② 不能表示具有结构性的知识：

③ 无法表示“具有结构关系的事物”之间的区别与联系。

④ 常需结合其他知识表示方法（如框架、语义网络）使用。

(3)求解流程：

① 初始化事实库（综合数据库、观察数据集或特征集）。

② 检查是否存在未用过的规则前提，与事实库匹配时转步骤 3，否则转步骤 5。

③ 使用匹配规则，更新事实库，并标记所用规则。

④ 检查事实库是否包含解。若有，终止求解过程；若无，返回步骤 2。

⑤ 请求更多关于问题的知识（特征）。若无法提供，则求解失败；否则更新事实库并返回步骤 2。

2. 卷积神经网络（CNN）

(1)优点：

① 泛化能力强： 能自动学习图像特征，适应新场景或未见过的数据。

② 计算效率高： 推理阶段具有较高效率，可快速处理大规模数据。

③ 无需专家知识： 训练完全基于数据，无需手动设计规则。

(2)缺点：

① 解释性差： 推理过程是一个“黑盒”，难以解释具体决策过程。

② 数据需求高： 需要大量标注数据以保证性能。

③ 容易过拟合： 小数据集上容易过拟合，需使用数据增强和正则化等技术。

3. 适用性分析

(1)产生式系统的适用场景：

④ 数据有限且特征结构化的任务，如：

⑤ 专家系统：利用明确规则支持诊断或决策，例如疾病诊断。

⑥ 故障诊断：通过特征条件分析推断设备问题。

⑦ 游戏 AI：根据规则推理制定策略。

⑧ 对透明性和解释性要求高的任务。

(2)卷积神经网络的适用场景：

① 数据丰富且特征复杂的任务，如：

② 图像识别：处理物体检测、人脸识别等。

③ 自然语言处理：例如文本生成或机器翻译。

④ 语音识别：从音频信号中提取语义信息。

⑤ 泛化能力要求较高且决策透明性要求低的任务。

综上所述，产生式系统与卷积神经网络（CNN）构成了两种相辅相成的动物识别技术。前者在数据量有限且需求明确的场景中表现出色，而后者则因其卓越的学习能力和适应性，在处理复杂任务时占据显著优势。选择何种技术应基于任务特性、数据规模及应用场景需求进行综合考量。

展望未来研究方向，可考虑将产生式系统与 CNN 融合，构建一种混合模型，以期充分发挥两者的优势。例如，在处理结构化特征数据时，可利用产生式系统处理简单规则；而在非结构化数据处理中，则可运用 CNN 提取高级特征。此外，通过将产生式系统应用于 CNN 的黑盒模型，可以为决策过程提供解释性，从而增强系统的透明度。

此方法不仅有望提升模型的性能，还能拓展其在多样化场景中的应用范围，为人工智能技术在动物识别及其他领域的进一步发展指明新的研究路径。

附录 A 产生式系统的源代码

```
from PyQt5 import QtCore, QtGui, QtWidgets
# 定义规则集
RULES = [
    {"if": {"蹄": True, "长尾巴": True, "骑乘或拉车
": True}, "then": "马"},
    {"if": {"羽毛": True, "喙": True, "下蛋": True}, "then": "鸡
"},
    {"if": {"体型小": True, "柔软皮肤": True, "锋利牙齿和爪子
": True}, "then": "猫"},
    {"if": {"体型多样": True, "毛发": True, "锋利牙齿": True, "
人类伴侣": True}, "then": "狗"},
    {"if": {"毛发": True}, "then": "哺乳动物"},
    {"if": {"奶": True}, "then": "哺乳动物"},
    {"if": {"羽毛": True}, "then": "鸟"},
    {"if": {"会飞": True, "下蛋": True}, "then": "鸟"},
    {"if": {"吃肉": True}, "then": "食肉动物"},
    {"if": {"犬齿": True, "爪": True, "眼盯前方
": True}, "then": "食肉动物"},
    {"if": {"哺乳动物": True, "蹄": True}, "then": "有蹄类动物
"},
    {"if": {"哺乳动物": True, "反刍动物": True}, "then": "有蹄类
动物"},
    {"if": {"哺乳动物": True, "食肉动物": True, "黄褐色
": True, "暗斑点": True}, "then": "金钱豹"},
    {"if": {"哺乳动物": True, "食肉动物": True, "黄褐色
": True, "黑色条纹": True}, "then": "虎"},
    {"if": {"有蹄类动物": True, "长脖子": True, "长腿": True, "
暗斑点": True}, "then": "长颈鹿"},
    {"if": {"有蹄类动物": True, "黑色条纹": True}, "then": "斑马
"},
    {"if": {"鸟": True, "长脖子": True, "长腿": True, "不会飞
": True, "黑白二色": True}, "then": "鸵鸟"},
```

```
    {"if": {"鸟": True, "会游泳": True, "不会飞": True, "黑白二色": True}, "then": "企鹅"},
    {"if": {"鸟": True, "善飞": True}, "then": "信天翁"}
]
# 特征映射字典（根据 RULES 顺序排列）
FEATURES = [
    "蹄", "长尾巴", "骑乘或拉车",  # 规则 1
    "羽毛", "喙", "下蛋",  # 规则 2
    "体型小", "柔软皮肤", "锋利牙齿和爪子",  # 规则 3
    "体型多样", "毛发", "锋利牙齿", "人类伴侣",  # 规则 4
    # 规则 5
    "奶",  # 规则 6
    # 规则 7
    "会飞",# 规则 8
    "吃肉",  # 规则 9
    "犬齿", "爪", "眼盯前方",  # 规则 10
    "哺乳动物",  # 规则 11
    "反刍动物",  # 规则 12
    "食肉动物", "黄褐色", "暗斑点",  # 规则 13
    "黄褐色", "黑色条纹",  # 规则 14
    "长脖子", "长腿", "暗斑点",  # 规则 15
    "有蹄类动物", "黑色条纹",  # 规则 16
    "鸟", "长脖子", "长腿", "不会飞", "黑白二色",  # 规则 17
    "会游泳",  # 规则 18
    "善飞",  # 规则 19
]
class AnimalRecognitionApp(QtWidgets.QWidget):
    def __init__(self):
        super().__init__()
        self.initUI()
    def initUI(self):
        self.setWindowTitle('动物识别系统')
        self.setGeometry(100, 100, 800, 600)
        # 创建一个水平分割器
        splitter = QtWidgets.QSplitter(QtCore.Qt.Horizontal, self)
        splitter.setGeometry(0, 0, 800, 600)
```

```
        # 左侧 - 特征选择区域与推理过程、结果展示区域
        leftWidget = QtWidgets.QWidget(self)
        leftLayout = QtWidgets.QVBoxLayout(leftWidget)
        # 设置标签字体
        self.label = QtWidgets.QLabel("请选择动物特征：", self)
        self.label.setStyleSheet("font: 16pt;")
        leftLayout.addWidget(self.label)
        self.featureButtons = {}
        y_position = 0
        for feature in FEATURES:
            label = QtWidgets.QLabel(feature, self)
            label.setStyleSheet("font: 14pt;")  # 增加字体大小
            leftLayout.addWidget(label)
            # 创建切换按钮（默认为"否"）
            button = QtWidgets.QPushButton("否", self)
            button.setStyleSheet("font: 14pt;")
            button.setCheckable(True)
            button.setChecked(False)  # 默认为"否"
            button.clicked.connect(self.toggle_button_state)
            self.featureButtons[feature] = button
            leftLayout.addWidget(button)
        # 将左侧的选择框添加到滚动区域
        scroll_area = QtWidgets.QScrollArea(self)
        scroll_area.setWidget(leftWidget)
        scroll_area.setWidgetResizable(True)
        scroll_area.setVerticalScrollBarPolicy(QtCore.Qt.Scrol
lBarAlwaysOn)
        # 右侧 - 推理过程和结果展示区域
        rightWidget = QtWidgets.QWidget(self)
        rightLayout = QtWidgets.QVBoxLayout(rightWidget)
        # 识别按钮
        self.recognizeButton = QtWidgets.QPushButton('识别
', self)
        self.recognizeButton.setStyleSheet("font: 14pt;")
        self.recognizeButton.clicked.connect(self.on_recognize
)
        rightLayout.addWidget(self.recognizeButton)
        # 结果标签
```

```
        self.resultLabel = QtWidgets.QLabel('识别结果将在这里显
示', self)
        self.resultLabel.setStyleSheet("font: 14pt; color: blu
e;")
        rightLayout.addWidget(self.resultLabel)
        # 推理过程标签
        self.procedureLabel = QtWidgets.QLabel('推理过程：
', self)
        self.procedureLabel.setStyleSheet("font: 14pt;")
        rightLayout.addWidget(self.procedureLabel)
        self.procedureTextEdit = QtWidgets.QTextEdit(self)
        self.procedureTextEdit.setReadOnly(True)
        self.procedureTextEdit.setStyleSheet("font: 12pt;")
        rightLayout.addWidget(self.procedureTextEdit)
        # 向分割器添加左右布局的控件
        splitter.addWidget(scroll_area)
        splitter.addWidget(rightWidget)
        self.show()
    def toggle_button_state(self):
        # 切换按钮的状态
        sender = self.sender()
        if sender.isChecked():
            sender.setText("是")
        else:
            sender.setText("否")
    def on_recognize(self):
        selected_features = {}
        # 获取用户选择的特征
        for feature, button in self.featureButtons.items():
            selected_features[feature] = button.isChecked()
        # 显示推理过程
        animal, inference_process = self.infer_animal(selected
_features)
        # 显示结果
        self.resultLabel.setText(f"识别出的动物是：{animal}")
        # 展示推理过程
        self.procedureTextEdit.clear()  # 清空文本框
        if inference_process:
```

```
            for step in inference_process:
                self.procedureTextEdit.append(step)
        else:
            self.procedureTextEdit.append("未匹配任何规则")
    def infer_animal(self, features):
        inference_process = []  # 用于记录推理过程
        matched_animal = None  # 用于保存匹配的动物
        # 遍历所有规则
        for rule in RULES:
            rule_conditions = rule["if"]  # 获取规则的条件部分
            # 检查每个条件是否都匹配
            # 如果特征不存在，则默认返回False
            if all(features.get(key, False) == value for key,
value in rule_conditions.items()):
                inference_process.append(f"规则匹
配: {rule_conditions} => 结果是 {rule['then']}")
                # 保存匹配到的动物
                matched_animal = rule["then"]
        # 如果找到了匹配的动物，返回匹配结果；否则返回"未知动物"
        if matched_animal:
            return matched_animal, inference_process
        else:
            return "未知动物", ["未匹配任何规则"]
if __name__ == '__main__':
    app = QtWidgets.QApplication([])
    ex = AnimalRecognitionApp()
    app.exec_()
```

附录 B CNN 方法的源代码

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import tensorflow as tf
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, MaxPool2D,
Flatten, Dropout, BatchNormalization
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.preprocessing.image import ImageDataGene
rator
from sklearn.metrics import ConfusionMatrixDisplay
from sklearn.metrics import classification_report, confusion_m
atrix
plt.rcParams['font.family'] = 'SimHei'  # 使用 SimHei 字体，支持
中文
plt.rcParams['axes.unicode_minus'] = False  # 解决负号显示问题
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
print(f"X_train shape: {X_train.shape}")
print(f"y_train shape: {y_train.shape}")
print(f"X_test shape: {X_test.shape}")
print(f"y_test shape: {y_test.shape}")
# 定义数据集的标签
labels = ['airplane', 'automobile', 'bird', 'cat', 'deer',
          'dog', 'frog', 'horse', 'ship', 'truck']
# 以网格格式显示更多图像
# 定义绘图网格的维度
W_grid = 10
L_grid = 10
# fig, axes = plt.subplots(L_grid, W_grid)
# subplot 返回图形对象和坐标轴对象
# 我们可以使用坐标轴对象在不同位置绘制特定的图形
```

```
fig, axes = plt.subplots(L_grid, W_grid, figsize = (17,17))
axes = axes.ravel() # 将15 x 15 矩阵展平为225 个数组
n_train = len(X_train) # 获取训练数据集的长度
# 从0 到n_train 之间随机选择一个数字
for i in np.arange(0, W_grid * L_grid): # 创建均匀分布的变量
    # 选择一个随机数字
    index = np.random.randint(0, n_train)
    # 使用选择的索引读取并显示图像
    axes[i].imshow(X_train[index,1:])
    label_index = int(y_train[index])
    axes[i].set_title(labels[label_index], fontsize = 8)
    axes[i].axis('off')
plt.subplots_adjust(hspace=0.4)
classes_name = ['Airplane', 'Automobile', 'Bird', 'Cat', 'Deer
', 'Dog', 'Frog', 'Horse', 'Ship', 'Truck']
classes, counts = np.unique(y_train, return_counts=True)
plt.barh(classes_name, counts)
plt.title('训练集中的类别分布')
classes, counts = np.unique(y_test, return_counts=True)
plt.barh(classes_name, counts)
plt.title('测试集中的类别分布')
# 数据预处理
# 数据归一化处理
X_train = X_train / 255.0
X_test = X_test / 255.0
# 将目标变量转换为独热编码
y_cat_train = to_categorical(y_train, 10)
y_cat_test = to_categorical(y_test, 10)
y_cat_train
# 模型构建
INPUT_SHAPE = (32, 32, 3)
KERNEL_SIZE = (3, 3)
model = Sequential()
# 卷积层
model.add(Conv2D(filters=32, kernel_size=KERNEL_SIZE, input_sh
ape=INPUT_SHAPE, activation='relu', padding='same'))
model.add(BatchNormalization())
```

```
model.add(Conv2D(filters=32, kernel_size=KERNEL_SIZE, activati
on='relu', padding='same'))
model.add(BatchNormalization())
# 池化层
model.add(MaxPool2D(pool_size=(2, 2)))
# Dropout 层
model.add(Dropout(0.25))
model.add(Conv2D(filters=64, kernel_size=KERNEL_SIZE, activati
on='relu', padding='same'))
model.add(BatchNormalization())
model.add(Conv2D(filters=64, kernel_size=KERNEL_SIZE, activati
on='relu', padding='same'))
model.add(BatchNormalization())
model.add(MaxPool2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Conv2D(filters=128, kernel_size=KERNEL_SIZE, activat
ion='relu', padding='same'))
model.add(BatchNormalization())
model.add(Conv2D(filters=128, kernel_size=KERNEL_SIZE, activat
ion='relu', padding='same'))
model.add(BatchNormalization())
model.add(MaxPool2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
# model.add(Dropout(0.2))
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.25))
model.add(Dense(10, activation='softmax'))
METRICS = [
    'accuracy',
    tf.keras.metrics.Precision(name='precision'),
    tf.keras.metrics.Recall(name='recall')
]
model.compile(loss='categorical_crossentropy', optimizer='adam
', metrics=METRICS)
model.summary()
## 早停
early_stop = EarlyStopping(monitor='val_loss', patience=2)
## 数据增强训练
```

```
batch_size = 32  # 设置批次大小为 32
# 创建数据增强生成器，进行图像的宽度平移、长度平移和水平翻转
data_generator = ImageDataGenerator(width_shift_range=0.1, hei
ght_shift_range=0.1, horizontal_flip=True)
# 使用数据增强生成器生成训练数据
train_generator = data_generator.flow(X_train, y_cat_train, ba
tch_size)
# 计算每个 epoch 的步骤数
steps_per_epoch = X_train.shape[0] // batch_size
# 训练模型
r = model.fit(train_generator,
              epochs=50,  # 训练 50 个 epoch
              steps_per_epoch=steps_per_epoch,  # 每个 epoch 执
行的步数
              validation_data=(X_test, y_cat_test),  # 使用测试
集进行验证
#               callbacks=[early_stop],  # 可以添加早停回调（未启
用）
#               batch_size=batch_size,  # 批次大小（此处未使用）
             )
# 模型评估
plt.figure(figsize=(12, 16))
plt.subplot(4, 2, 1)
plt.plot(r.history['loss'], label='Loss')
plt.plot(r.history['val_loss'], label='val_Loss')
plt.title('损失函数演化')
plt.legend()
plt.subplot(4, 2, 2)
plt.plot(r.history['accuracy'], label='accuracy')
plt.plot(r.history['val_accuracy'], label='val_accuracy')
plt.title('准确率演化')
plt.legend()
plt.subplot(4, 2, 3)
plt.plot(r.history['precision'], label='precision')
plt.plot(r.history['val_precision'], label='val_precision')
plt.title('精确度演化')
plt.legend()
plt.subplot(4, 2, 4)
```

```
plt.plot(r.history['recall'], label='recall')
plt.plot(r.history['val_recall'], label='val_recall')
plt.title('召回率演化')
plt.legend()
evaluation = model.evaluate(X_test, y_cat_test)  # 评估模型在测试集上的表现
print(f'测试集准确率 : {evaluation[1] * 100:.2f}%')  # 输出测试集准确率
y_pred = model.predict(X_test)  # 使用模型对测试集进行预测
y_pred = np.argmax(y_pred, axis=1)  # 获取每个样本预测的类别标签（最大概率对应的类别）
cm = confusion_matrix(y_test, y_pred)  # 计算混淆矩阵
disp = ConfusionMatrixDisplay(confusion_matrix=cm,
                              display_labels=labels)  # 创建混淆矩阵的可视化对象
# 绘制混淆矩阵
fig, ax = plt.subplots(figsize=(10, 10))  # 设置画布大小
disp = disp.plot(xticks_rotation='vertical', ax=ax, cmap='summer')  # 绘制混淆矩阵，设置标签旋转和颜色映射
plt.show()  # 显示混淆矩阵图
print(classification_report(y_test, y_pred))
my_image = X_test[100]
plt.imshow(my_image)
# that's a Deer
print(f" Image 100 is {y_test[100]}")
# correctly predicted as a Deer
pred_100 = np.argmax(model.predict(my_image.reshape(1, 32, 32, 3)))
print(f"The model predict that image 100 is {pred_100}")
# 定义数据集的标签
labels = ['airplane', 'automobile', 'bird', 'cat', 'deer',
          'dog', 'frog', 'horse', 'ship', 'truck']
# 以网格形式展示更多的图像
# 定义绘图网格的尺寸
W_grid = 5  # 网格的宽度
L_grid = 5  # 网格的长度
# fig, axes = plt.subplots(L_grid, W_grid)
```

```
# subplot 返回的是图形对象和坐标轴对象
# 我们可以使用 axes 对象在不同的位置绘制图像
fig, axes = plt.subplots(L_grid, W_grid, figsize = (17,17))
axes = axes.ravel()  # 将 5x5 的矩阵展平为 25 个元素的一维数组
n_test = len(X_test)  # 获取测试集的长度
# 随机选择数字从 0 到 n_test
for i in np.arange(0, W_grid * L_grid):  # 创建均匀分布的变量
    # 随机选择一个数字
    index = np.random.randint(0, n_test)
    # 根据选择的索引读取并显示图像
    axes[i].imshow(X_test[index, 1:])  # 显示选择的图像
    label_index = int(y_pred[index])  # 获取预测标签
    axes[i].set_title(labels[label_index], fontsize = 8)  # 设置子图的标题
    axes[i].axis('off')  # 关闭坐标轴
plt.subplots_adjust(hspace=0.4)  # 调整子图之间的间距
def plot_image(i, predictions_array, true_label, img):
    predictions_array, true_label, img = predictions_array, true_label[i], img[i]
    plt.grid(False)  # 关闭网格
    plt.xticks([])  # 不显示 x 轴刻度
    plt.yticks([])  # 不显示 y 轴刻度
    plt.imshow(img, cmap=plt.cm.binary)  # 显示图片，使用二进制颜色映射
    predicted_label = np.argmax(predictions_array)  # 获取预测结果的标签
    if predicted_label == true_label:  # 如果预测标签和真实标签相同，显示蓝色
        color = 'blue'
    else:  # 如果预测标签和真实标签不同，显示红色
        color = 'red'
    # 设置图像标题，显示预测标签和其概率，以及真实标签
    plt.xlabel(f"{labels[int(predicted_label)]} {100*np.max(predictions_array):2.0f}% ({labels[int(true_label)]})",
               color=color)
def plot_value_array(i, predictions_array, true_label):
```

```
    predictions_array, true_label = predictions_array, int(tru
e_label[i])
    plt.grid(False)  # 关闭网格
    plt.xticks(range(10))  # 显示 10 个类别的 x 轴刻度
    plt.yticks([])  # 不显示 y 轴刻度
    thisplot = plt.bar(range(10), predictions_array, color="#7
77777")  # 绘制预测值的柱状图
    plt.ylim([0, 1])  # 设置 y 轴的范围为 [0, 1]
    predicted_label = np.argmax(predictions_array)  # 获取预测
结果的标签
    thisplot[predicted_label].set_color('red')  # 将预测标签的柱
子颜色设为红色
    thisplot[true_label].set_color('blue')  # 将真实标签的柱子颜
色设为蓝色
predictions = model.predict(X_test)  # 使用模型对测试集进行预测
# 绘制前 num_images 张测试图像、它们的预测标签和真实标签。
# 正确的预测用蓝色标记，错误的预测用红色标记。
num_rows = 8  # 设置图像的行数
num_cols = 5  # 设置图像的列数
num_images = num_rows * num_cols  # 总共展示的图像数量
plt.figure(figsize=(2 * 2 * num_cols, 2 * num_rows))  # 设置图
形的尺寸
for i in range(num_images):
    # 绘制每个图像及其预测标签
    plt.subplot(num_rows, 2 * num_cols, 2 * i + 1)
    plot_image(i, predictions[i], y_test, X_test)  # 绘制图像和
标签
    # 绘制每个图像的预测概率分布
    plt.subplot(num_rows, 2 * num_cols, 2 * i + 2)
    plot_value_array(i, predictions[i], y_test)  # 绘制预测概率
分布
plt.tight_layout()  # 自动调整子图间距
plt.show()  # 显示图形
from tensorflow.keras.models import load_model
model.save('cnn_20_epochs.h5')
```