项目说明文档

目录

1 题目

基于 CNN 的深度学习网络训练测试平台

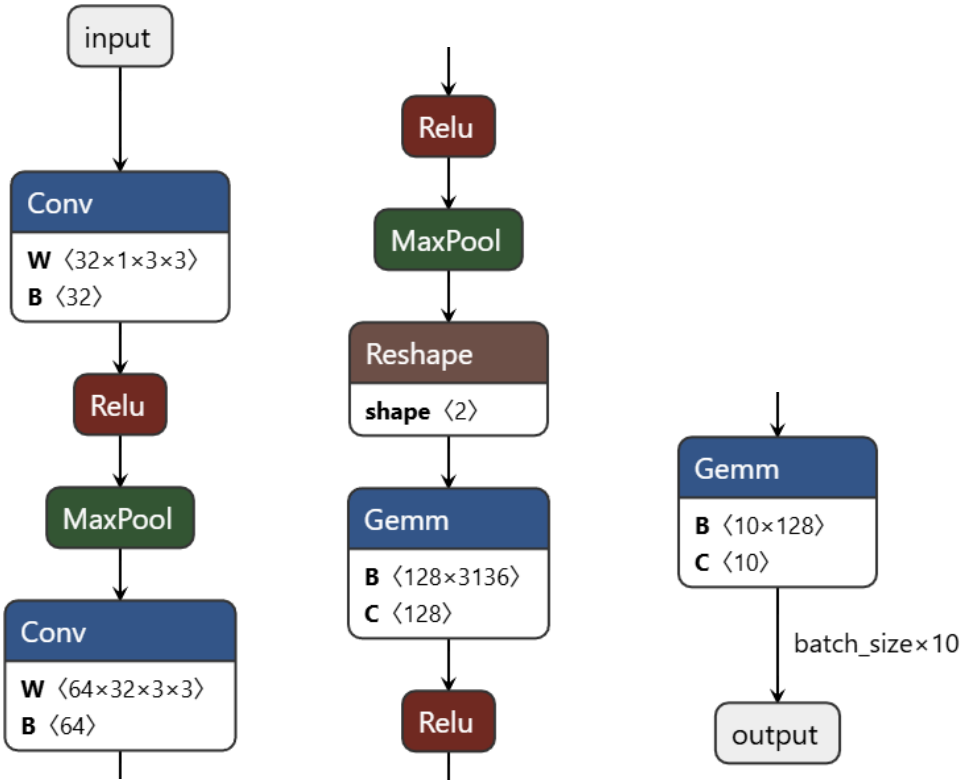
2 需求分析与设计

2.1 需求分析

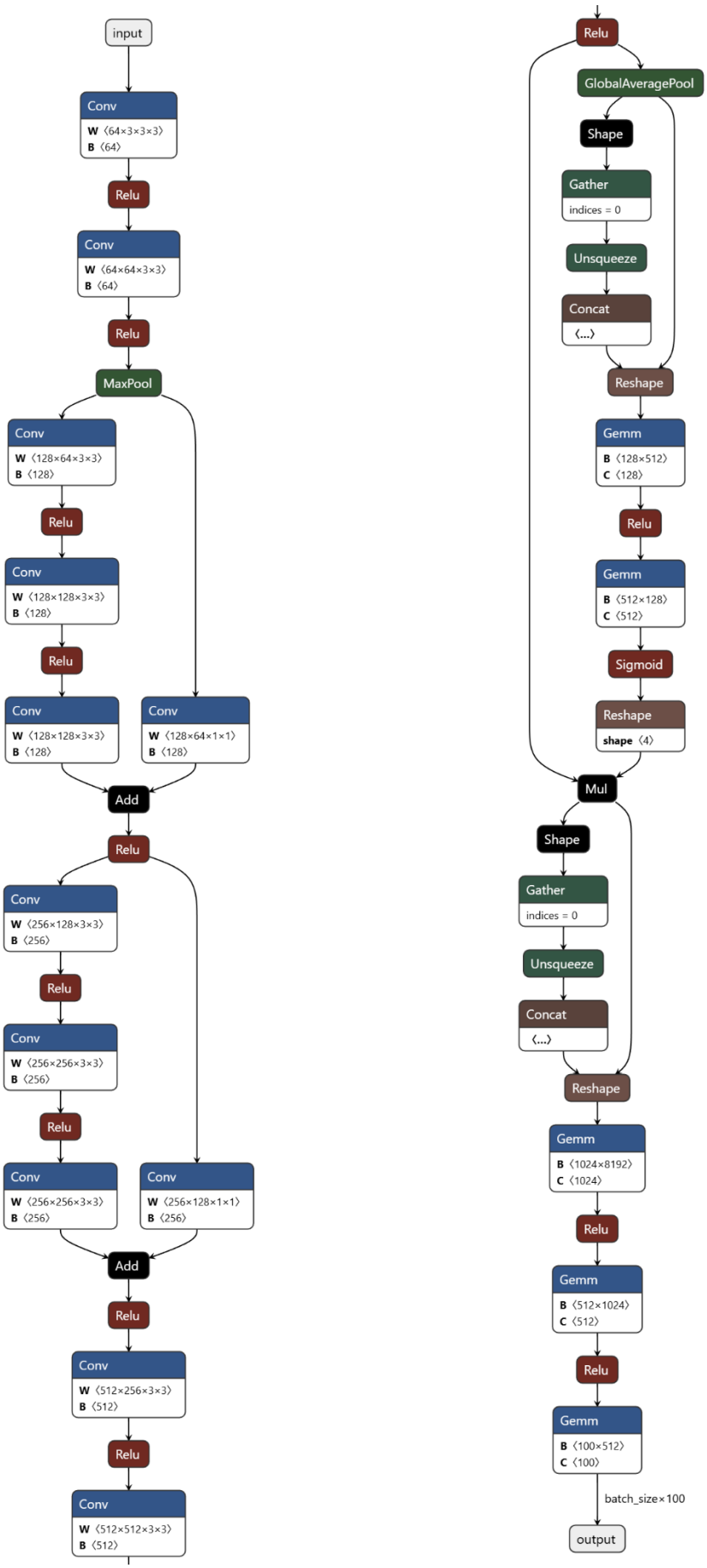
- **模型选择与管理**：用户需要能够选择不同的卷积神经网络（CNN）模型，如适用于 MNIST 数据集的 CNN_MNIST、适用于 CIFAR - 10 数据集的 CNN_CIFAR10 和适用于 CIFAR - 100 数据集的 CNN_CIFAR100，并具备保存和加载模型参数的功能。
- **计算设备选择**：支持用户选择不同的计算设备，包括 CPU（如 Intel Core i9 CPU）和 GPU（如 NVIDIA GEFORCE RTX 4090 LapTop），以满足不同的计算需求。
- **训练参数设置**：用户可以设置训练轮数（Epochs）和批处理量（Batches），以便灵活调整训练过程。
- **训练与测试功能**：提供训练模型、测试模型和使用模型的功能，并能显示训练日志和测试准确率。
- **数据显示**：允许用户显示训练数据，帮助用户了解训练过程和模型性能。

2.2 系统设计

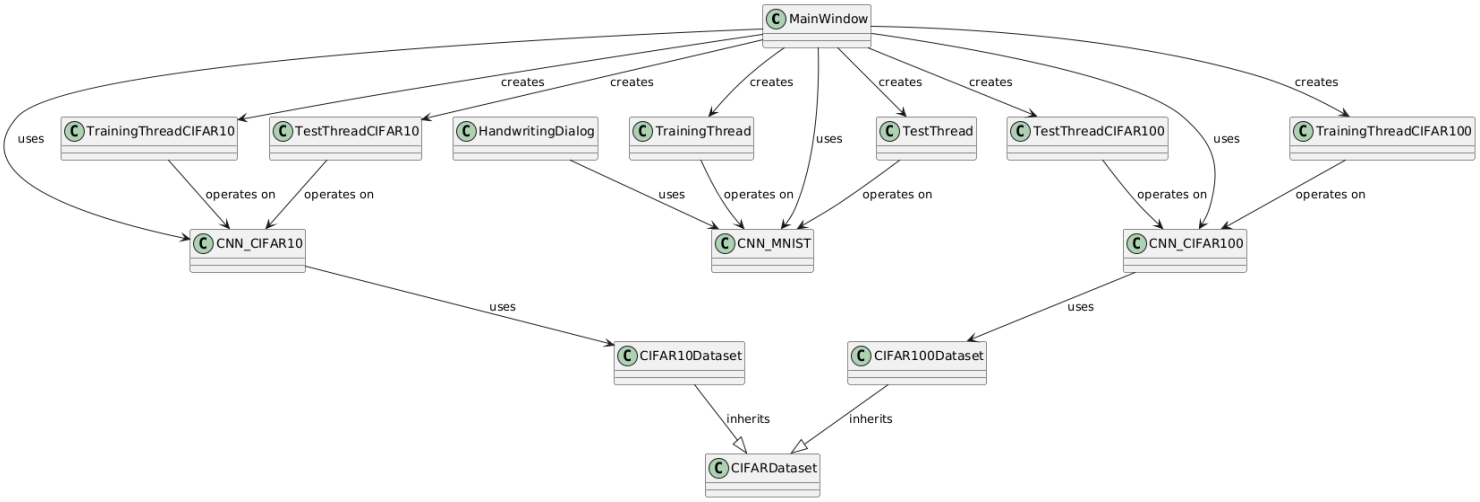
- **CNN_MNIST 网络构建：**



- **CNN_CIFAR100 网络构建（CNN_CIFAR10 与之相似）：**


input
Conv
W 〈64×3×3×3〉
B 〈64〉
Relu
Conv
W 〈64×64×3×3〉
B 〈64〉
Relu
MaxPool
Conv
W 〈128×64×3×3〉
B 〈128〉
Relu
Conv
W 〈128×128×3×3〉
B 〈128〉
Relu
Conv
W 〈128×128×3×3〉
B 〈128〉
Conv
W 〈128×64×1×1〉
B 〈128〉
Add
Relu
Conv
W 〈256×128×3×3〉
B 〈256〉
Relu
Conv
W 〈256×256×3×3〉
B 〈256〉
Relu
Conv
W 〈256×256×3×3〉
B 〈256〉
Conv
W 〈256×128×1×1〉
B 〈256〉
Add
Relu
Conv
W 〈512×256×3×3〉
B 〈512〉
Relu
Conv
W 〈512×512×3×3〉
B 〈512〉
Relu
GlobalAveragePool
Shape
Gather
indices = 0
Unsqueeze
Concat
〈...〉
Reshape
Gemm
B 〈128×512〉
C 〈128〉
Relu
Gemm
B 〈512×128〉
C 〈512〉
Sigmoid
Reshape
shape 〈4〉
Mul
Shape
Gather
indices = 0
Unsqueeze
Concat
〈...〉
Reshape
Gemm
B 〈1024×8192〉
C 〈1024〉
Relu
Gemm
B 〈512×1024〉
C 〈512〉
Relu
Gemm
B 〈100×512〉
C 〈100〉
batch_size×100
output

- **系统类框架结构示意图：**

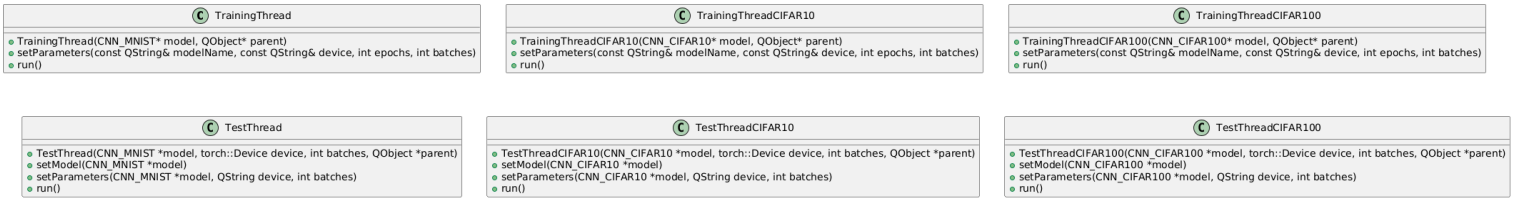


- **用户界面设计：**采用 Qt 框架设计图形用户界面（GUI），包含标题、模型选择框、设备选择框、参数输入框、功能按钮、状态标签、进度条、日志文本框和清空日志按钮等元素，方便用户操作和查看信息。

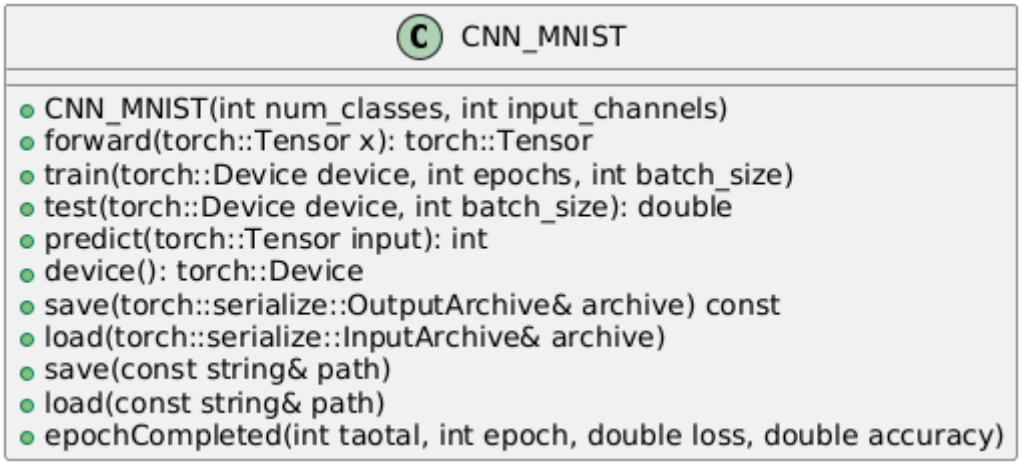
1、 MainWindow 类 UML 示意图：

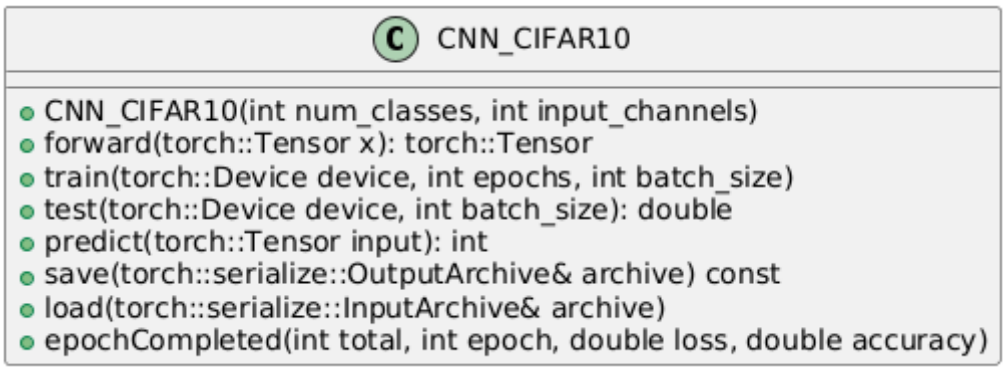
- **模块设计**：将系统划分为多个模块，包括主窗口模块、模型模块、训练线程模块和测试线程模块。主窗口模块负责界面的初始化和事件处理；模型模块定义不同的 CNN 模型；训练线程模块和测试线程模块分别负责模型的训练和测试任务，采用多线程技术，避免界面卡顿。

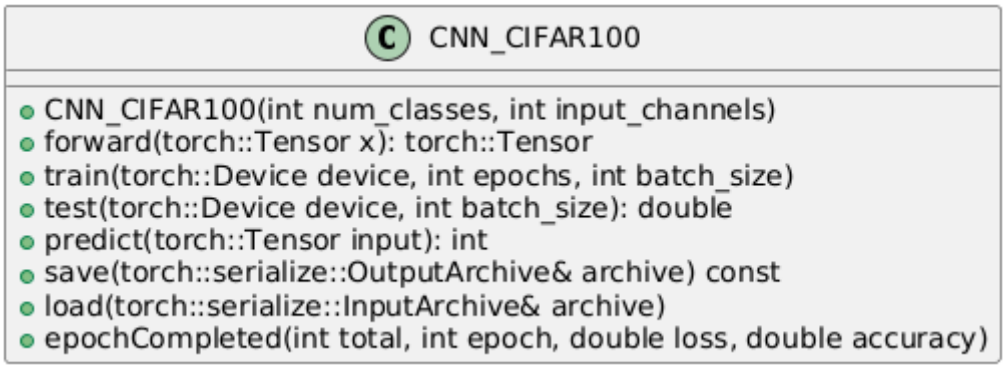
1、 线程相关模块类的 UML 示意图：



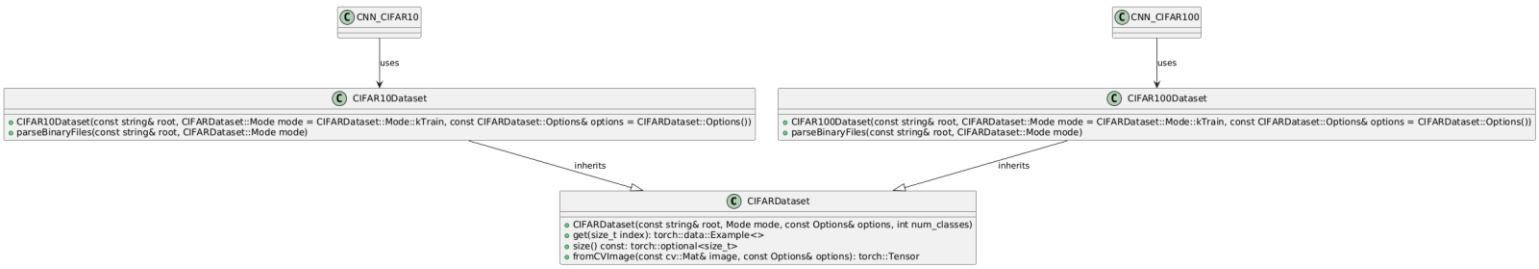
2、 模型相关类的 UML 示意图：



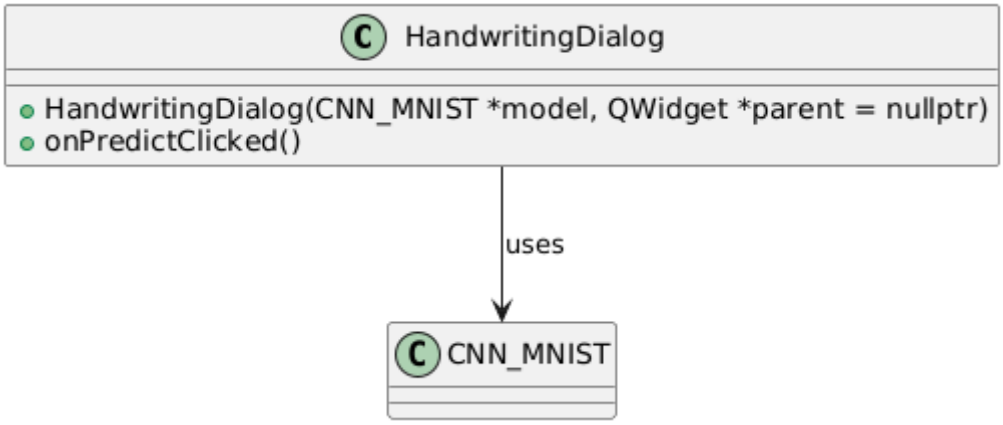




3、 数据集相关类的 UML 示意图：

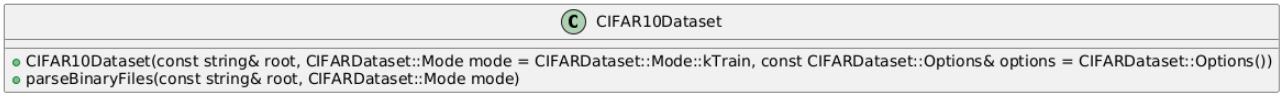


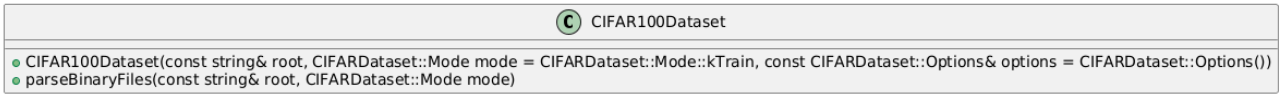
4、 其他类的 UML 示意图：



- **数据处理设计：**使用自定义的数据集类（如 CIFARDataset）处理 CIFAR - 10 和 CIFAR - 100 数据集，包括数据加载、预处理和归一化等操作。

1、 CIFAR 相关数据集：





3 系统实现与使用方法

3.1 系统开发环境

- **操作系统：**Microsoft Windows 11
- **编程语言：**C++ 17 std。
- **深度学习框架：**LibTorch(Pytorch C++ API)，用于构建和训练 CNN 模型。
- **GUI 框架：**Qt6，用于设计和实现用户界面。
- **第三方库：**OpenCV，用于图像处理。

3.2 系统界面简介

- **标题栏：**显示系统名称 “基于 CNN 的深度学习网络训练测试平台”。

- **模型选择框**：提供三种模型选项（CNN_MNIST、CNN_CIFAR10、CNN_CIFAR100），用户可以选择要使用的模型。
- **设备选择框**：提供两种计算设备选项（Intel Core i9 CPU、NVIDIA GEFORCE RTX 4090 LapTop），用户可以选择训练和测试时使用的设备。
- **参数输入框**：包括训练轮数输入框和批处理量输入框，用户可以输入有效的训练轮数和批处理量。
- **功能按钮**：包括训练模型、测试模型、使用模型、显示训练数据、保存模型参数和加载模型参数等按钮，用户可以通过点击按钮执行相应的操作。
- **状态标签**：显示系统当前状态，如“就绪”。
- **进度条**：显示训练进度，范围为 0 - 100%。
- **日志文本框**：显示训练日志，包括训练开始、训练过程中的损失和准确率信息、训练结束等。
- **清空日志按钮**：用于清空日志文本框中的内容。

3.3 系统功能模块简介

- **主窗口模块（MainWindow）**：负责界面的初始化和事件处理，包括设置界面布局、连接信号和槽、处理按钮点击事件等。
- **模型模块**：
  - CNN_MNIST：适用于 MNIST 数据集的 CNN 模型。
  - CNN_CIFAR10：适用于 CIFAR - 10 数据集的 CNN 模型，包含卷积层、批归一化层、残差连接、通道注意力机制和全连接层等。
  - CNN_CIFAR100：适用于 CIFAR - 100 数据集的 CNN 模型，结构与 CNN_CIFAR10 类似。
- **训练线程模块**：
  - TrainingThread：用于训练 CNN_MNIST 模型的线程类。
  - TrainingThreadCIFAR10：用于训练 CNN_CIFAR10 模型的线程类。
  - TrainingThreadCIFAR100：用于训练 CNN_CIFAR100 模型的线程类。

- **测试线程模块：**
  - TestThread：用于测试 CNN_MNIST 模型的线程类。
  - TestThreadCIFAR10：用于测试 CNN_CIFAR10 模型的线程类。
  - TestThreadCIFAR100：用于测试 CNN_CIFAR100 模型的线程类。
- **数据集模块：**
  - CIFARDataset：抽象基类，定义了数据集的基本操作。
  - CIFAR10Dataset：用于加载和处理 CIFAR - 10 数据集。
  - CIFAR100Dataset：用于加载和处理 CIFAR - 100 数据集。

3.4 重点算法介绍

- **卷积神经网络（CNN）**：CNN 是一种专门用于处理具有网格结构数据（如图像）的深度学习模型。本系统中的 CNN 模型包含卷积层、池化层、批归一化层、残差连接、通道注意力机制和全连接层等组件。卷积层用于提取图像的特征，池化层用于降低特征图的维度，批归一化层用于加速模型收敛，残差连接用于解决梯度消失问题，通道注意力机制用于增强模型对重要特征的关注，全连接层用于输出分类结果。
- **卷积(Conv2D)层：**卷积操作是 CNN 中特征提取的核心步骤。给定输入 X，卷积核 W 和偏置 b，卷积层的输出 Y（前向传播）的计算公式为：

$$Y = \sum_{m=0}^{M-1}\sum_{n=0}^{N-1}\sum_{c=0}^{C-1} X_{i+m,j+n}^{c}$$

- **最大值池化(MaxPooling)层**：最大池化用于降低特征图的维度，同时保留重要特征。给定输入特征图 X，池化窗口大小为 $P\times P$，步长为 S，最大池化的输出 Y 的计算公式为：

$$Y_{i,j}^{k} = max_{m=0}^{P-1} max_{n=0}^{P-1} X_{i\cdot S+m,j\cdot S+n}^{k}$$

- **全连接（Linear）层：**全连接层将特征图展平为一维向量，并通过线性变换输出分类结果。给定输入向量 x，权重矩阵 W 和偏置向量 b，全连接层的输出 y 可以通过以下公式计算：

$$y = W \cdot x + b$$

- **通道注意力机制（Channel_Attention）：**通道注意力机制用于增强模型对重要特征的关注。给定输入特征图 X，通道注意力机制的输出 Y 可以通过以下步骤计算：

  1、 **全局平均池化**（其中，H 和 W 是特征图的高度和宽度）：

$$GAP(X) = \frac{1}{H \times W}\sum_{i=0}^{H-1}\sum_{j=0}^{W-1} X_{i,j}$$

  2、 **全连接层：**

$$z_1 = ReLU(W_1 \cdot GAP(X) + b_1)$$
$$z_2 = Sigmoid(W_2 \cdot z_2 + b_2)$$

  3、 **特征图加权：**

$$Y = X \cdot z_2$$

- **Adam 优化算法**：Adam 是一种自适应学习率的优化算法，结合了 Adagrad 和 RMSProp 的优点，能够自适应地调整每个参数的学习率，在大多数情况下都能取得较好的训练效果。本系统在模型训练过程中使用 Adam 优化算法更新模型参数。

  1、 计算一阶矩估计和二阶矩估计：

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

2、 修正一阶矩估计和二阶矩估计偏差：

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

3、 更新参数：

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

- **交叉熵损失函数**：交叉熵损失函数常用于分类问题，用于衡量模型预测结果与真实标签之间的差异。本系统在模型训练过程中使用交叉熵损失函数计算损失，并通过反向传播算法更新模型参数。在此项目中，真实标签是一个 one-hot 向量，因此交叉熵损失函数可以简化为：

$$L = -\sum_{i=1}^{C} y_i \, log(\hat{y}_i)$$

- **各层的误差反向传播(backward)算法：**

1、 交叉熵误差(Cross Entropy Error)联合 SoftMax 梯度求导:

$$\frac{\partial L}{\partial z_i} = \hat{y}_i - y_i$$

2、 全连接层的反向传播：

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial y} X^T$$
$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial y}$$
$$\frac{\partial L}{\partial x} = W^T \frac{\partial L}{\partial y}$$

3、 卷积层的反向传播（*HY* 和 *WY* 是输出特征图的高度和宽度）：

$$\frac{\partial L}{\partial W_{m,n}^{c,k}} = \sum_{i=0}^{H_{Y-1}} \sum_{j=0}^{W_{Y-1}} \frac{\partial L}{\partial Y_{i,j}^k} \cdot X_{i+m,j+n}^c$$

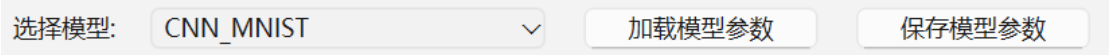
$$\frac{\partial L}{\partial b^k} = \sum_{i=0}^{H_{Y-1}} \sum_{j=0}^{W_{Y-1}} \frac{\partial L}{\partial Y_{i,j}^k}$$

$$\frac{\partial L}{\partial X_{i,j}^c} = \sum_{k=0}^{K-1} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} \frac{\partial L}{\partial Y_{i-m,j-n}^k} \cdot W_{m,n}^{c,k}$$

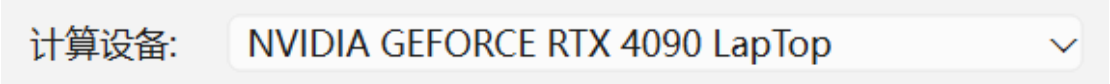
3.5 使用手册

1. **启动系统**：运行可执行文件，打开系统界面。

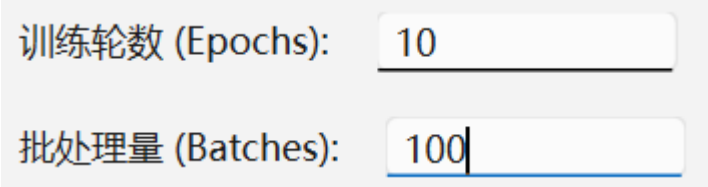
2. **选择模型**：在模型选择框中选择要使用的模型（CNN_MNIST、CNN_CIFAR10 或 CNN_CIFAR100）。



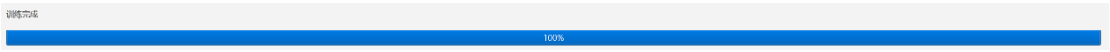
3. **选择计算设备**：在设备选择框中选择训练和测试时使用的计算设备（Intel Core i9 CPU 或 NVIDIA GEFORCE RTX 4090 LapTop）。

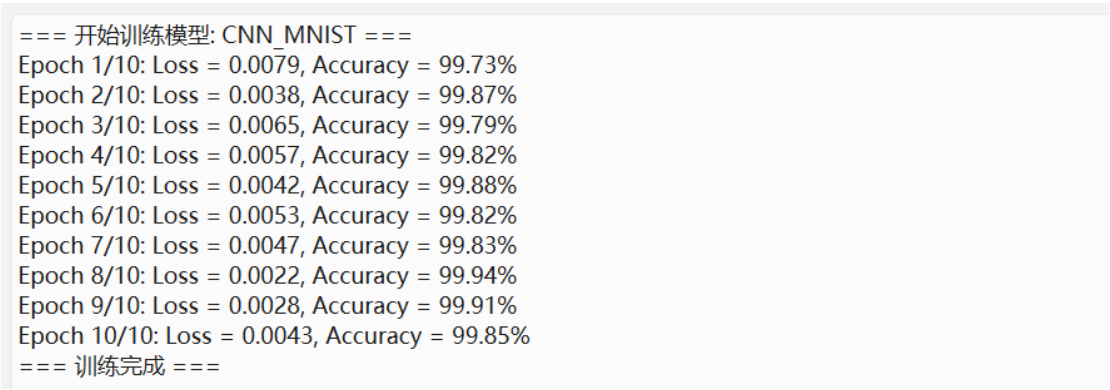


4. **设置训练参数**：在训练轮数输入框和批处理量输入框中输入有效的训训练轮数和批处理量。

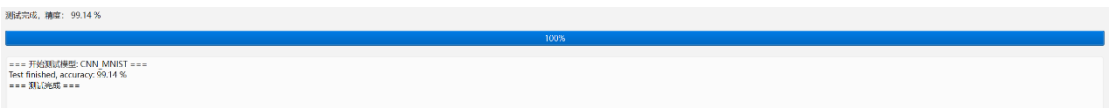


5. **训练模型**：点击“训练模型”按钮，系统开始训练所选模型，并在日志文本框中显示训练日志和进度条。

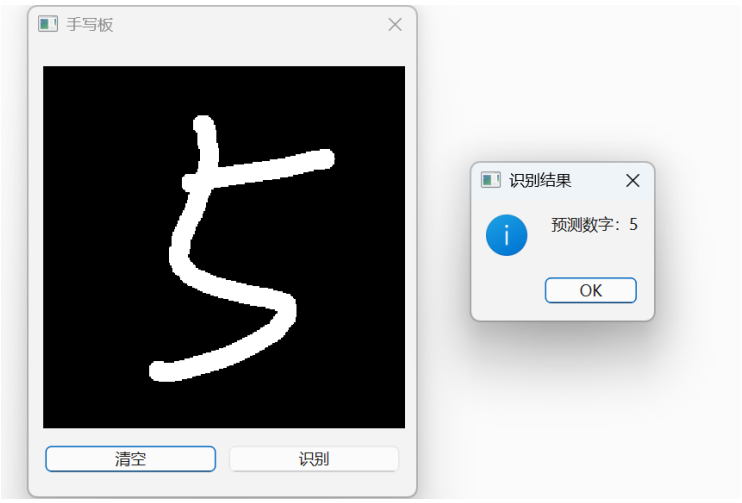




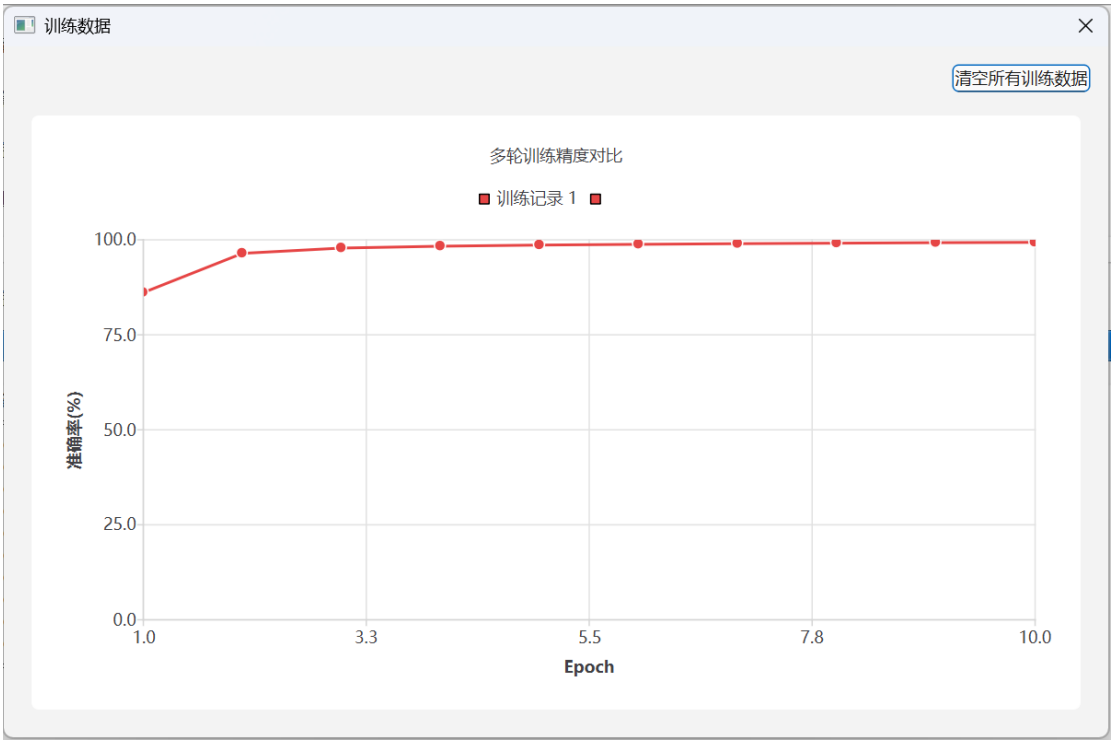
6. **测试模型**：点击“测试模型”按钮，系统使用测试数据集对训练好的模型进行测试，并在日志文本框中显示测试准确率。



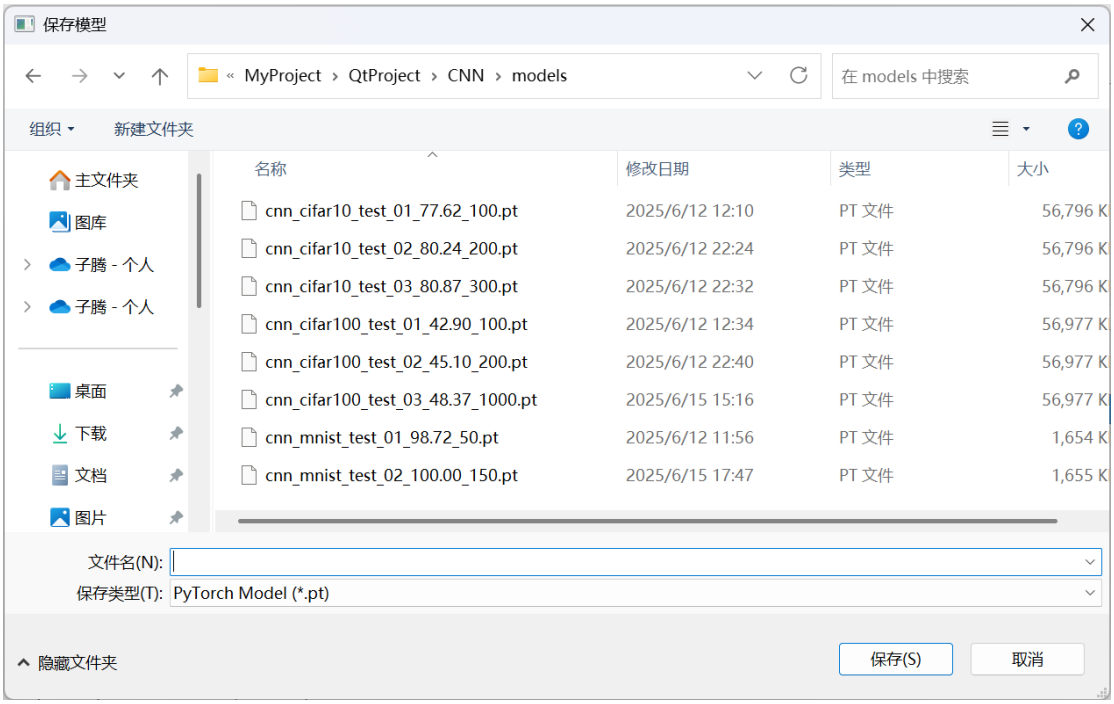
7. **使用模型**：点击“使用模型”按钮，系统可以使用训练好的模型进行预测。



8. **显示训练数据**：点击“显示训练数据”按钮，系统可以显示训练过程中的损失和准确率等信息。



9. **保存和加载模型参数**：点击“保存模型参数”按钮可以将训练好的模型参数保存到文件中；点击“加载模型参数”按钮可以从文件中加载模型参数。



10. **清空日志：**点击“清空日志”按钮可以清空日志文本框中的内容。

4 运行实例与系统功能测试

4.1 系统功能测试

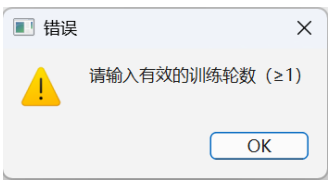
- **模型选择功能测试：**分别选择 CNN_MNIST、CNN_CIFAR10 和 CNN_CIFAR100 模型，检查系统是否能够正确响应并进行相应的操作。

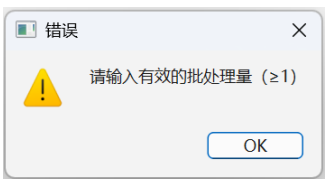
```
模型设置为: CNN_CIFAR10
模型设置为: CNN_CIFAR100
模型设置为: CNN_MNIST
```

- **计算设备选择功能测试：**分别选择 Intel Core i9 CPU 和 NVIDIA GEFORCE RTX 4090 LapTop 作为计算设备，检查系统是否能够正确切换设备并进行训练和测试。

```
设备设置为: NVIDIA GEFORCE RTX 4090 LapTop
设备设置为: Intel Core i9 CPU
```

- **训练参数设置功能测试：**输入不同的训练轮数和批处理量，检查系统是否能够正确识别并使用这些参数进行训练。





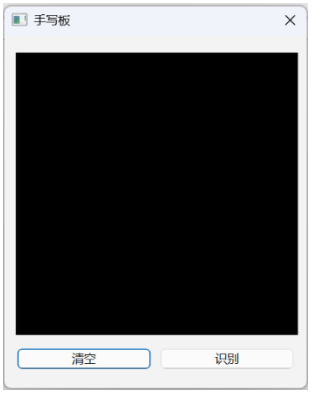
- **训练功能测试：**点击“训练模型”按钮，检查系统是否能够正常开始训练，并在日志文本框中显示训练日志和进度条。训练完成后，检查模型是否能够正常保存。

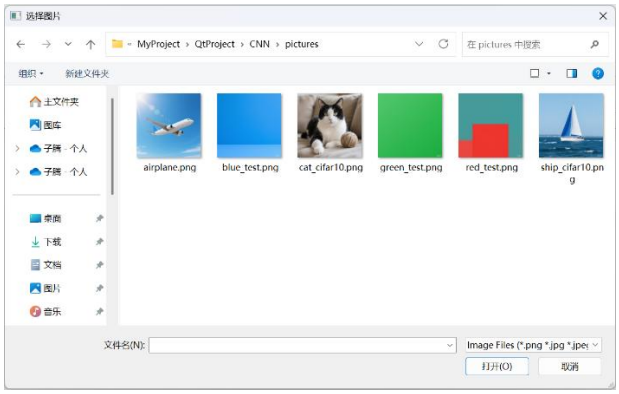
```
=== 开始训练模型: CNN_CIFAR10 ===
Epoch 1/3: Loss = 2.2229, Accuracy = 17.34%
Epoch 2/3: Loss = 1.8252, Accuracy = 28.68%
Epoch 3/3: Loss = 1.5738, Accuracy = 38.78%
=== 训练完成 ===
```

- **测试功能测试：**点击“测试模型”按钮，检查系统是否能够使用测试数据集对训练好的模型进行测试，并在日志文本框中显示测试准确率。

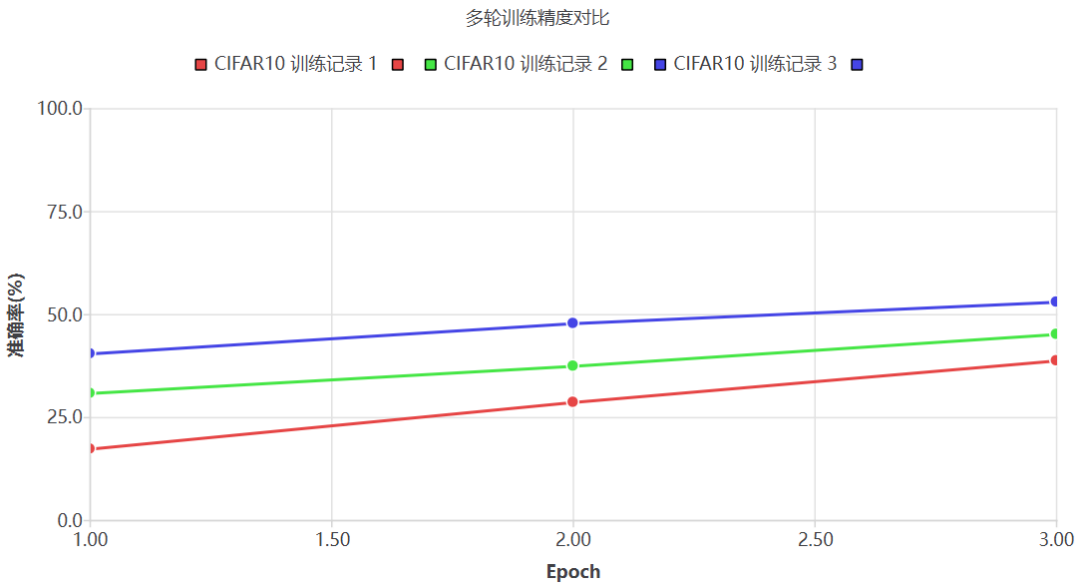
```
模型设置为: CNN_MNIST
=== 开始测试模型: CNN_MNIST ===
Test finished, accuracy: 99.16 %
=== 测试完成 ===
```

- **使用模型功能测试：**点击“使用模型”按钮，检查系统是否能够使用训练好的模型进行预测。





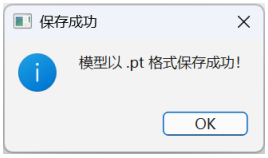
- **显示训练数据功能测试：**点击“显示训练数据”按钮，检查系统是否能够显示训练过程中的损失和准确率等信息。



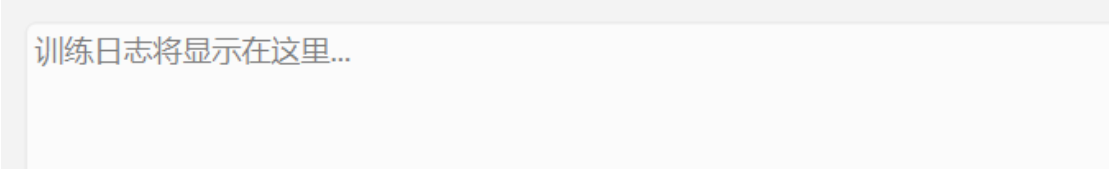
- **保存和加载模型参数功能测试：**点击“保存模型参数”按钮将模型参数保存到文件中，然后点击“加载模型参数”按钮从文件中加载模型参数，检查系统是否能够正

确保存和加载模型参数。

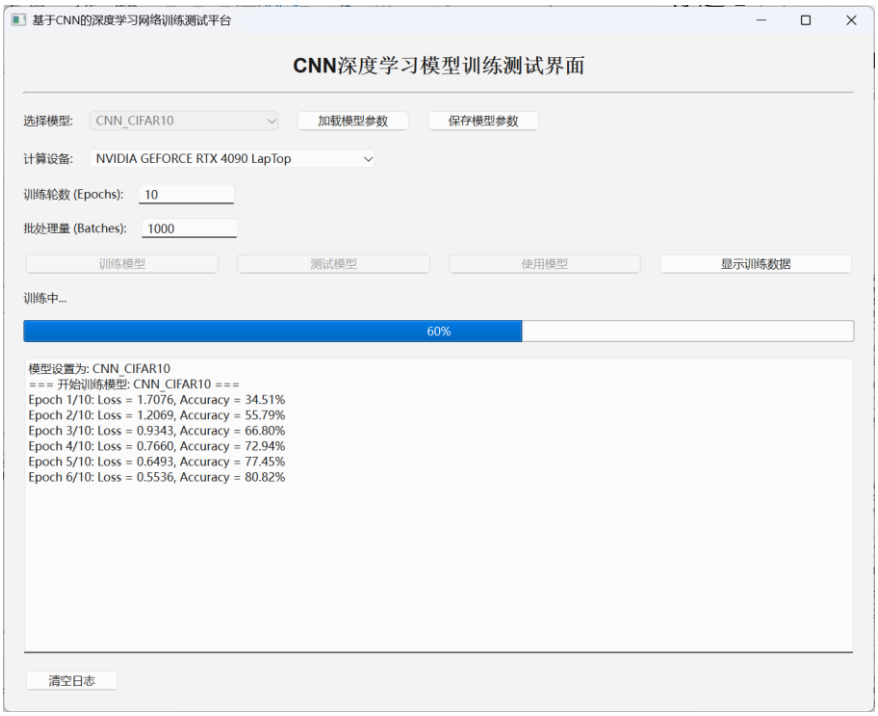
```
=== 开始加载 CNN_MNIST 模型参数 ===
成功加载 CNN_MNIST 模型参数，路径：D:/MyProject/QtProject/CNN/models/cnn_mnist_test_02_100.00_150.pt
```



- **清空日志功能测试：**点击“清空日志”按钮，检查日志文本框中的内容是否能够被清空。



4.2 运行实例



5 总结与进一步改进

5.1 总结（心得体会）

通过本次项目的开发，我对深度学习和卷积神经网络有了更深入的理解。在项目中，我学会了使用 LibTorch 框架构建和训练 CNN 模型，掌握了 Qt 框架设计和实现用户界面的方法，了解了多线程编程在深度学习应用中的重要性。同时，我也深刻体会到了数据预处理和模型调优的重要性，不同的数据集和模型结构需要不同的预处理方法和调优策略。在开发过程中，

我遇到了一些问题，如模型训练过程中出现的梯度消失问题和内存溢出问题，通过查阅资料和不断尝试，我最终解决了这些问题。通过本次项目，我不仅提高了自己的编程能力和解决问题的能力，还增强了对深度学习领域的兴趣和信心。

5.2 进一步改进及建议

- **模型优化**：可以尝试使用更复杂的模型结构，如 ResNet、Inception 等，以提高模型的性能。同时，可以使用数据增强技术，如随机裁剪、旋转、翻转等，增加训练数据的多样性，提高模型的泛化能力。
- **用户界面优化**：可以增加更多的可视化功能，如绘制训练过程中的损失和准确率曲线，帮助用户更直观地了解训练过程和模型性能。同时，可以优化界面布局，提高用户体验。
- **错误处理和提示**：可以增加更详细的错误处理和提示信息，当用户输入无效参数或系统出现错误时，能够及时向用户反馈错误信息，并提供相应的解决方案。
- **分布式训练**：可以引入分布式训练技术，如使用多个 GPU 或多台计算机进行并行训练，以提高训练效率。
- **模型评估指标**：除了准确率之外，可以增加更多的模型评估指标，如召回率、F1 值等，以更全面地评估模型的性能。

6 附录——源程序

项目的 CMake 文件

```
# CMake 最低版本要求
cmake_minimum_required(VERSION 3.16)
# 启用 UTF-8 支持策略 (CMP0118)
cmake_policy(SET CMP0118 NEW)

# 针对 MSVC 编译器设置 UTF-8 编码
if(MSVC)
    add_compile_options(
        /utf-8                    # 强制源文件和执行文件使用 UTF-8
        /source-charset:utf-8 # 指定源文件编码为 UTF-8
        /execution-charset:utf-8 # 指定执行文件编码为 UTF-8
    )
endif()

# 项目定义
project(CNN VERSION 0.1 LANGUAGES CXX)

# 自动包含当前目录
set(CMAKE_INCLUDE_CURRENT_DIR ON)
```

```
# 启用 Qt 自动处理工具
set(CMAKE_AUTOUIC ON)  # 自动处理 UI 文件
set(CMAKE_AUTOMOC ON)  # 自动处理元对象系统
set(CMAKE_AUTORCC ON)  # 自动处理资源文件

# C++ 标准设置
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)  # 必须使用 C++17

# 查找 Qt6 核心组件
find_package(Qt6 REQUIRED COMPONENTS Core Gui Charts)
# 兼容 Qt5/Qt6 的查找方式
find_package(QT NAMES Qt6 Qt5 REQUIRED COMPONENTS Widgets)
find_package(Qt${QT_VERSION_MAJOR} REQUIRED COMPONENTS Widgets)

# 强制使用特定 Visual Studio 编译器
if(MSVC)
    set(CMAKE_C_COMPILER "D:/Program Files/Visual Studio/Visual Studio/VC/Tools/MSVC/14.44.35207/bin/Hostx64/x64/cl.exe")
    set(CMAKE_CXX_COMPILER "D:/Program Files/Visual Studio/Visual Studio/VC/Tools/MSVC/14.44.35207/bin/Hostx64/x64/cl.exe")
endif()

# LibTorch 配置
set(Torch_DIR D:/Libtorch-release/share/cmake/Torch)  # Libtorch 路径

# Vulkan SDK 配置
set(Vulkan_DIR "D:/VulkanSDK/1.4.313.1" CACHE PATH "Vulkan SDK 安装目录")

# 禁用 CUDA 性能分析工具
set(USE_PERF_COUNTER OFF CACHE BOOL "" FORCE)
set(USE_NVTX OFF CACHE BOOL "" FORCE)

# 查找 CUDA 工具包
find_package(CUDAToolkit QUIET)
if(CUDAToolkit_FOUND)
    message(STATUS "已找到 CUDA 工具包，版本: ${CUDAToolkit_VERSION}")
else()
    message(STATUS "未找到 CUDA 工具包，将使用 CPU 模式编译")
endif()

# 防止 PyTorch 尝试定义 nvToolsExt 目标
set(_CUDA_HAS_nvToolsExt OFF CACHE BOOL "" FORCE)
```

```
# 查找 PyTorch
find_package(Torch REQUIRED)

# OpenCV 配置
set(OpenCV_DIR "D:/Program Files/OpenCV/opencv/build/x64/vc16/lib" CACHE PATH "OpenCV 库路径")
find_package(OpenCV REQUIRED COMPONENTS core highgui imgproc imgcodecs)

# OpenCV 验证
if(OpenCV_FOUND)
    message(STATUS "OpenCV 版本: ${OpenCV_VERSION}")
    message(STATUS "OpenCV 包含路径: ${OpenCV_INCLUDE_DIRS}")
    message(STATUS "OpenCV 库文件: ${OpenCV_LIBS}")
else()
    message(FATAL_ERROR "未找到 OpenCV 库，请检查路径配置")
endif()

# 项目源文件定义
set(PROJECT_SOURCES
    main.cpp
    Mainwindow.cpp
    Mainwindow.h
    Mainwindow.ui

    # MNIST 相关文件
    model_mnist.h
    model_mnist.cpp
    testthread.h
    testthread.cpp
    trainingthread.h
    trainingthread.cpp
    handwritingdialog.h
    handwritingdialog.cpp

    # CIFAR 相关文件
    model_cifar.h
    model_cifar.cpp
    cifardataset.h
    cifardataset.cpp
    testthread_cifar.h
    testthread_cifar.cpp
    trainingthread_cifar.h
    trainingthread_cifar.cpp
```

```
)

# 创建可执行文件
if(${QT_VERSION_MAJOR} GREATER_EQUAL 6)
    qt_add_executable(CNN
        MANUAL_FINALIZATION  # 需要手动完成可执行文件处理
        ${PROJECT_SOURCES}
    )
else()
    if(ANDROID)
        add_library(CNN SHARED ${PROJECT_SOURCES})
    else()
        add_executable(CNN ${PROJECT_SOURCES})
    endif()
endif()

# 链接 Qt Widgets 模块
target_link_libraries(CNN PRIVATE Qt${QT_VERSION_MAJOR}::Widgets)

# macOS 应用包配置
if(${QT_VERSION} VERSION_LESS 6.1.0)
  set(BUNDLE_ID_OPTION MACOSX_BUNDLE_GUI_IDENTIFIER com.example.CNN)
endif()
set_target_properties(CNN PROPERTIES
    ${BUNDLE_ID_OPTION}
    MACOSX_BUNDLE_BUNDLE_VERSION ${PROJECT_VERSION}
    MACOSX_BUNDLE_SHORT_VERSION_STRING
${PROJECT_VERSION_MAJOR}.${PROJECT_VERSION_MINOR}
    MACOSX_BUNDLE TRUE
    WIN32_EXECUTABLE TRUE  # Windows 下生成 GUI 应用
)

# 链接 Torch 和 OpenCV 库
target_link_libraries(CNN PUBLIC "${TORCH_LIBRARIES}")
target_link_libraries(CNN PUBLIC ${OpenCV_LIBS})

# 安装配置
include(GNUInstallDirs)
install(TARGETS CNN
    BUNDLE DESTINATION .          # macOS 应用包安装位置
    LIBRARY DESTINATION ${CMAKE_INSTALL_LIBDIR}  # 库文件安装位置
    RUNTIME DESTINATION ${CMAKE_INSTALL_BINDIR}  # 可执行文件安装位置
)
```

```
# Qt6 的最终化处理
if(QT_VERSION_MAJOR EQUAL 6)
    qt_finalize_executable(CNN)
endif()

# 屏蔽 MSVC 的 size_t 到 int 转换警告
if(MSVC)
    target_compile_options(CNN PRIVATE /wd4267)
endif()

# 链接额外的 Qt 模块
target_link_libraries(CNN PRIVATE
    Qt6::Core
    Qt6::Gui
    Qt6::Charts
)
```

项目的源文件

mainwindow.cpp

```
#include "mainwindow.h"
#include "cifardataset.h"
#include "handwritingdialog.h"
#include <QtCharts/QChart>
#include <QtCharts/QLineSeries>
#include <QtCharts/QChartView>
#include <QtCharts/QScatterSeries>

void MainWindow::setupUI() {
    QWidget *centralWidget = new QWidget(this);
    setCentralWidget(centralWidget);

    QVBoxLayout *mainLayout = new QVBoxLayout(centralWidget);
    mainLayout->setSpacing(15);
    mainLayout->setContentsMargins(20, 20, 20, 20);

    QLabel *titleLabel = new QLabel("CNN 深度学习模型训练测试界面", this);
    titleLabel->setFont(QFont("Arial", 16, QFont::Bold));
    titleLabel->setAlignment(Qt::AlignCenter);
    mainLayout->addWidget(titleLabel);

    QFrame *line = new QFrame(this);
    line->setFrameShape(QFrame::HLine);
```

```
line->setFrameShadow(QFrame::Sunken);
mainLayout->addWidget(line);

QHBoxLayout *modelSelectLayout = new QHBoxLayout();
QLabel *modelLabel = new QLabel("选择模型:", this);
modelComboBox = new QComboBox(this);
modelComboBox->addItem("CNN_MNIST");
modelComboBox->addItem("CNN_CIFAR10");
modelComboBox->addItem("CNN_CIFAR100");
modelComboBox->setMinimumWidth(200);

saveModelButton = new QPushButton("保存模型参数", this);
saveModelButton->setMinimumWidth(120);
loadModelButton = new QPushButton("加载模型参数", this);
loadModelButton->setMinimumWidth(120);

modelSelectLayout->addWidget(modelLabel);
modelSelectLayout->addWidget(modelComboBox);
modelSelectLayout->addWidget(loadModelButton);
modelSelectLayout->addWidget(saveModelButton);
modelSelectLayout->addStretch();

mainLayout->addLayout(modelSelectLayout);

QHBoxLayout *deviceSelectLayout = new QHBoxLayout();
QLabel *deviceLabel = new QLabel("计算设备:", this);
deviceComboBox = new QComboBox(this);
deviceComboBox->addItem("Intel Core i9 CPU");
deviceComboBox->addItem("NVIDIA GEFORCE RTX 4090 LapTop");
deviceComboBox->setMinimumWidth(300);

deviceSelectLayout->addWidget(deviceLabel);
deviceSelectLayout->addWidget(deviceComboBox);
deviceSelectLayout->addStretch();

mainLayout->addLayout(deviceSelectLayout);

QHBoxLayout *paramLayout = new QHBoxLayout();
QLabel *epochsLabel = new QLabel("训练轮数  (Epochs):", this);
epochsLineEdit = new QLineEdit(this);
epochsLineEdit->setPlaceholderText("输入训练轮数");
epochsLineEdit->setValidator(new QIntValidator(1, 1000, this));
epochsLineEdit->setMaximumWidth(100);
```

```
paramLayout->addWidget(epochsLabel);
paramLayout->addWidget(epochsLineEdit);
paramLayout->addStretch();

mainLayout->addLayout(paramLayout);

QHBoxLayout *paramLayout1 = new QHBoxLayout();
QLabel *batchesLabel = new QLabel("批处理量  (Batches):", this);
batchesLineEdit = new QLineEdit(this);
batchesLineEdit->setPlaceholderText("输入批处理量");
batchesLineEdit->setValidator(new QIntValidator(1, 1000, this));
batchesLineEdit->setMaximumWidth(100);

paramLayout1->addWidget(batchesLabel);
paramLayout1->addWidget(batchesLineEdit);
paramLayout1->addStretch();

mainLayout->addLayout(paramLayout1);

QHBoxLayout *buttonLayout = new QHBoxLayout();
trainButton = new QPushButton("训练模型", this);
testButton = new QPushButton("测试模型", this);
showDataButton = new QPushButton("显示训练数据", this);
useModelButton = new QPushButton("使用模型");

buttonLayout->addWidget(trainButton);
buttonLayout->addWidget(testButton);
buttonLayout->addWidget(useModelButton);
buttonLayout->addWidget(showDataButton);

mainLayout->addLayout(buttonLayout);

statusLabel = new QLabel("就绪", this);
mainLayout->addWidget(statusLabel);

progressBar = new QProgressBar(this);
progressBar->setRange(0, 100);
progressBar->setValue(0);
progressBar->setStyle(QStyleFactory::create("Fusion"));
mainLayout->addWidget(progressBar);

logTextEdit = new QTextEdit(this);
logTextEdit->setReadOnly(true);
logTextEdit->setPlaceholderText("训练日志将显示在这里...");
```

```
    mainLayout->addWidget(logTextEdit);

    clearLogButton = new QPushButton("清空日志", this);
    clearLogButton->setFixedWidth(logTextEdit->width());
    connect(clearLogButton, &QPushButton::clicked, logTextEdit, &QTextEdit::clear);
    mainLayout->addWidget(clearLogButton);

    setWindowTitle("基于 CNN 的深度学习网络训练测试平台");
    setMinimumSize(800, 600);
    resize(900, 700);
}

void MainWindow::setupConnections() {
    connect(trainButton, &QPushButton::clicked, this, &MainWindow::onTrainButtonClicked);
    connect(testButton, &QPushButton::clicked, this, &MainWindow::onTestButtonClicked);
    connect(useModelButton, &QPushButton::clicked, this,
&MainWindow::onUseModelButtonClicked);
    connect(showDataButton, &QPushButton::clicked, this,
&MainWindow::onShowDataButtonClicked);

    connect(deviceComboBox, &QComboBox::currentTextChanged, this,
&MainWindow::onDeviceChanged);

    connect(modelComboBox, &QComboBox::currentTextChanged, this,
&MainWindow::onModelChanged);

    connect(loadModelButton, &QPushButton::clicked, this,
&MainWindow::onLoadModelClicked);
    connect(saveModelButton, &QPushButton::clicked, this,
&MainWindow::onSaveModelClicked);

    connect(m_trainingThread, &TrainingThread::trainingStarted, this,
&MainWindow::onTrainingStarted);
    connect(m_trainingThread, &TrainingThread::progressUpdated, this,
&MainWindow::onTrainingProgress);
    connect(m_trainingThread, &TrainingThread::epochCompleted, this,
&MainWindow::onTrainingEpochCompleted);
    connect(m_trainingThread, &TrainingThread::trainingFinished, this,
&MainWindow::onTrainingFinished);
    connect(m_trainingThread, &TrainingThread::errorOccurred, this,
&MainWindow::onTrainingErrorOccurred);

    connect(m_trainingThreadCIFAR10, &TrainingThreadCIFAR10::trainingStarted, this,
&MainWindow::onTrainingStarted);
```

```
    connect(m_trainingThreadCIFAR10, &TrainingThreadCIFAR10::progressUpdated, this,
&MainWindow::onTrainingProgress);
    connect(m_trainingThreadCIFAR10, &TrainingThreadCIFAR10::epochCompleted, this,
&MainWindow::onTrainingEpochCompleted);
    connect(m_trainingThreadCIFAR10, &TrainingThreadCIFAR10::trainingFinished, this,
&MainWindow::onTrainingFinished);
    connect(m_trainingThreadCIFAR10, &TrainingThreadCIFAR10::errorOccurred, this,
&MainWindow::onTrainingErrorOccurred);

    connect(m_trainingThreadCIFAR100, &TrainingThreadCIFAR100::trainingStarted, this,
&MainWindow::onTrainingStarted);
    connect(m_trainingThreadCIFAR100, &TrainingThreadCIFAR100::progressUpdated, this,
&MainWindow::onTrainingProgress);
    connect(m_trainingThreadCIFAR100, &TrainingThreadCIFAR100::epochCompleted, this,
&MainWindow::onTrainingEpochCompleted);
    connect(m_trainingThreadCIFAR100, &TrainingThreadCIFAR100::trainingFinished, this,
&MainWindow::onTrainingFinished);
    connect(m_trainingThreadCIFAR100, &TrainingThreadCIFAR100::errorOccurred, this,
&MainWindow::onTrainingErrorOccurred);

    connect(m_testThread, &TestThread::testStarted, this, &MainWindow::onTestStart);
    connect(m_testThread, &TestThread::testFinished, this, &MainWindow::onTestFinished);
    connect(m_testThread, &TestThread::errorOccurred, this,
&MainWindow::onTestErrorOccurred);

    connect(m_testThreadCIFAR10, &TestThreadCIFAR10::testStarted, this,
&MainWindow::onTestStart);
    connect(m_testThreadCIFAR10, &TestThreadCIFAR10::testFinished, this,
&MainWindow::onTestFinished);
    connect(m_testThreadCIFAR10, &TestThreadCIFAR10::errorOccurred, this,
&MainWindow::onTestErrorOccurred);

    connect(m_testThreadCIFAR100, &TestThreadCIFAR100::testStarted, this,
&MainWindow::onTestStart);
    connect(m_testThreadCIFAR100, &TestThreadCIFAR100::testFinished, this,
&MainWindow::onTestFinished);
    connect(m_testThreadCIFAR100, &TestThreadCIFAR100::errorOccurred, this,
&MainWindow::onTestErrorOccurred);
}

void MainWindow::onTrainButtonClicked() {
    QString modelName = modelComboBox->currentText();
    QString device = deviceComboBox->currentText();
    QString epochsText = epochsLineEdit->text();
```

```
    QString batchesText = batchesLineEdit->text();
    if (modelName.isEmpty()) {
        QMessageBox::warning(this, "错误", "请选择模型");
        return;
    }
    bool ok;
    int epochs = epochsText.toInt(&ok);
    if (!ok || epochs <= 0) {
        QMessageBox::warning(this, "错误", "请输入有效的训练轮数（≥1）");
        return;
    }
    bool ok1;
    int batches = batchesText.toInt(&ok1);
    if (!ok1 || batches <= 0) {
        QMessageBox::warning(this, "错误", "请输入有效的批处理量（≥1）");
        return;
    }
    logTextEdit->append(QString("===  开始训练模型: %1 ===").arg(modelName));
    currentAccuracy.clear();
    currentEpochs.clear();
    if (modelName == "CNN_MNIST") {
        m_trainingThread->setParameters(modelName, device, epochs, batches);
        m_trainingThread->start();
    }
    else if (modelName == "CNN_CIFAR10") {
        m_trainingThreadCIFAR10->setParameters(modelName, device, epochs, batches);
        m_trainingThreadCIFAR10->start();
    }
    else if (modelName == "CNN_CIFAR100") {
        m_trainingThreadCIFAR100->setParameters(modelName, device, epochs, batches);
        m_trainingThreadCIFAR100->start();
    }
}

void MainWindow::onTestButtonClicked() {
    QString device = deviceComboBox->currentText();
    QString batchesText = batchesLineEdit->text();
    QString modelName = modelComboBox->currentText();
    bool ok1;
    int batches = batchesText.toInt(&ok1);
    if (!ok1 || batches <= 0) {
        QMessageBox::warning(this, "错误", "请输入有效的批处理量（≥1）");
        return;
    }
```

```
    if (modelName.isEmpty()) {
        QMessageBox::warning(this, "测试错误", "请选择一个模型");
        return;
    }
    statusLabel->setText("测试中...");
    logTextEdit->append(QString("=== 开始测试模型: %1 ===").arg(modelName));

    if (modelName == "CNN_MNIST") {
        m_testThread->setParameters(m_mnistModel, device, batches);
        m_testThread->start();
    }
    else if (modelName == "CNN_CIFAR10") {
        m_testThreadCIFAR10->setParameters(m_cifar10Model, device, batches);
        m_testThreadCIFAR10->start();
    }
    else if (modelName == "CNN_CIFAR100") {
        m_testThreadCIFAR100->setParameters(m_cifar100Model, device, batches);
        m_testThreadCIFAR100->start();
    }
}

void MainWindow::onUseModelButtonClicked() {
    QString modelName = modelComboBox->currentText();
    if (modelName.isEmpty()) {
        QMessageBox::warning(this, "错误", "请选择模型");
        return;
    }
    logTextEdit->append(QString("=== 开 始 使 用 %1 模 型 处 理 图 片
===").arg(modelName));
    if (modelName == "CNN_MNIST") {
        HandwritingDialog dialog(m_mnistModel, this);
        dialog.exec();
    }
    else {
        QString filePath = QFileDialog::getOpenFileName(this, "选择图片", QDir::homePath(),
"Image Files (*.png *.jpg *.jpeg)");
        if (filePath.isEmpty()) {
            logTextEdit->append("用户取消选择图片");
            return;
        }
        try {
            cv::Mat image = cv::imread(filePath.toStdString());
            if (image.empty()) {
                throw runtime_error("无法读取图片文件: " + filePath.toStdString());
```

```
            }
            torch::Tensor inputTensor;
            if (modelName == "CNN_CIFAR10") {
                inputTensor = CIFARDataset::fromCVImage(image, CIFARDataset::Options());
                int result = m_cifar10Model->predict(inputTensor);
                QString result_str = QString::fromStdString(CNN_CIFAR10::classify(result));
                QMessageBox::information(this, "预测结果", QString("CIFAR10 模型预测结果: %1").arg(result_str));
            }
            else if (modelName == "CNN_CIFAR100") {
                inputTensor = CIFARDataset::fromCVImage(image, CIFARDataset::Options());
                int result = m_cifar100Model->predict(inputTensor);
                QString result_str = QString::fromStdString(CNN_CIFAR100::classify(result));
                QMessageBox::information(this, "预测结果", QString("CIFAR100 模型预测结果: %1").arg(result_str));
            }
            logTextEdit->append(QString("成功使用 %1 模型处理图片，路径：%2").arg(modelName).arg(filePath));
        } catch (const exception& e) {
            logTextEdit->append(QString("处理失败：%1").arg(e.what()));
        }
    }
}

void MainWindow::onShowDataButtonClicked() {
    QString modelName = modelComboBox->currentText();
    if (modelName == "CNN_MNIST" && allAccuracyHistory.isEmpty()) {
        QMessageBox::information(this, "提示", "无历史训练数据");
        return;
    }
    else if (modelName == "CNN_CIFAR10" && allAccuracyHistory_cifar10.isEmpty()) {
        QMessageBox::information(this, "提示", "无历史训练数据");
        return;
    }
    else if (modelName == "CNN_CIFAR100" && allAccuracyHistory_cifar100.isEmpty()) {
        QMessageBox::information(this, "提示", "无历史训练数据");
        return;
    }

    QDialog *dialog = new QDialog(this);
    dialog->setWindowTitle("训练数据");
```

```
dialog->resize(800, 500);

QVBoxLayout *mainLayout = new QVBoxLayout(dialog);
QChart *chart = new QChart();
chart->setTitle("多轮训练精度对比");
QChartView *chartView = new QChartView(chart);
chartView->setRenderHint(QPainter::Antialiasing);

// 清空按钮
QHBoxLayout *buttonLayout = new QHBoxLayout();
QPushButton *clearButton = new QPushButton("清空所有训练数据");
connect(clearButton, &QPushButton::clicked, this, [this, dialog, chart]() {
    allAccuracyHistory.clear();
    allEpochList.clear();
    trainRecordColors.clear();
    allAccuracyHistory_cifar10.clear();
    allEpochList_cifar10.clear();
    trainRecordColors_cifar10.clear();
    allAccuracyHistory_cifar100.clear();
    allEpochList_cifar100.clear();
    trainRecordColors_cifar100.clear();
    chart->removeAllSeries();
    dialog->close();
});
buttonLayout->addStretch();
buttonLayout->addWidget(clearButton);

if (modelName == "CNN_MNIST") {
    for (int recordIdx = 0; recordIdx < allAccuracyHistory.size(); ++recordIdx) {
        const QList<int>& epochs = allEpochList[recordIdx];
        const QList<double>& accuracies = allAccuracyHistory[recordIdx];

        QColor color;
        if (recordIdx < trainRecordColors.size()) {
            color = trainRecordColors[recordIdx];
        }
        else {
            color = Qt::red;
            qDebug() << "警告：颜色列表长度不足，使用默认 Qt::red";
        }

        QLineSeries *lineSeries = new QLineSeries();
        lineSeries->setName(QString("训练记录 %1").arg(recordIdx + 1));
        lineSeries->setPen(QPen(color, 2));
```

```
            QScatterSeries *scatterSeries = new QScatterSeries();
            scatterSeries->setMarkerSize(8);
            scatterSeries->setMarkerShape(QScatterSeries::MarkerShapeCircle);
            scatterSeries->setColor(color);
            scatterSeries->setBorderColor(Qt::white);

            for (int i = 0; i < epochs.size(); ++i) {
                qDebug() << "添加点：epoch=" << epochs[i] << " accuracy=" << accuracies[i];
                lineSeries->append(epochs[i], accuracies[i]);
                scatterSeries->append(epochs[i], accuracies[i]);
            }

            chart->addSeries(lineSeries);
            chart->addSeries(scatterSeries);
        }
    }
    else if (modelName == "CNN_CIFAR10") {
        for (int recordIdx = 0; recordIdx < allAccuracyHistory_cifar10.size(); ++recordIdx) {
            const QList<int>& epochs = allEpochList_cifar10[recordIdx];
            const QList<double>& accuracies = allAccuracyHistory_cifar10[recordIdx];
            if (epochs.isEmpty() || accuracies.isEmpty()) continue;

            QColor color;
            if (recordIdx < trainRecordColors_cifar10.size()) {
                color = trainRecordColors_cifar10[recordIdx];
            }
            else {
                color = Qt::blue;
                qDebug() << "警告：颜色列表长度不足，使用默认 Qt::blue";
            }

            QLineSeries *lineSeries = new QLineSeries();
            lineSeries->setName(QString("CIFAR10 训练记录 %1").arg(recordIdx + 1));
            lineSeries->setPen(QPen(color, 2));

            QScatterSeries *scatterSeries = new QScatterSeries();
            scatterSeries->setMarkerSize(8);
            scatterSeries->setMarkerShape(QScatterSeries::MarkerShapeCircle);
            scatterSeries->setColor(color);
            scatterSeries->setBorderColor(Qt::white);

            for (int i = 0; i < epochs.size(); ++i) {
                lineSeries->append(epochs[i], accuracies[i]);
```

```
                scatterSeries->append(epochs[i], accuracies[i]);
            }

            chart->addSeries(lineSeries);
            chart->addSeries(scatterSeries);
        }
    }
    else if (modelName == "CNN_CIFAR100") {
        for (int recordIdx = 0; recordIdx < allAccuracyHistory_cifar100.size(); ++recordIdx) {
            const QList<int>& epochs = allEpochList_cifar100[recordIdx];
            const QList<double>& accuracies = allAccuracyHistory_cifar100[recordIdx];
            if (epochs.isEmpty() || accuracies.isEmpty()) continue;

            QColor color;
            if (recordIdx < trainRecordColors_cifar10.size()) {
                color = trainRecordColors_cifar10[recordIdx];
            }
            else {
                color = Qt::green;
                qDebug() << "警告：颜色列表长度不足，使用默认 Qt::green";
            }

            QLineSeries *lineSeries = new QLineSeries();
            lineSeries->setName(QString("CIFAR100 训练记录 %1").arg(recordIdx + 1));
            lineSeries->setPen(QPen(color, 2));

            QScatterSeries *scatterSeries = new QScatterSeries();
            scatterSeries->setMarkerSize(8);
            scatterSeries->setMarkerShape(QScatterSeries::MarkerShapeCircle);
            scatterSeries->setColor(color);
            scatterSeries->setBorderColor(Qt::white);

            for (int i = 0; i < epochs.size(); ++i) {
                lineSeries->append(epochs[i], accuracies[i]);
                scatterSeries->append(epochs[i], accuracies[i]);
            }

            chart->addSeries(lineSeries);
            chart->addSeries(scatterSeries);
        }
    }

    chart->createDefaultAxes();
    chart->axes(Qt::Horizontal).first()->setTitleText("Epoch");
```

```
    chart->axes(Qt::Vertical).first()->setTitleText("准确率(%)");
    chart->axes(Qt::Vertical).first()->setRange(0, 100);

    mainLayout->addLayout(buttonLayout);
    mainLayout->addWidget(chartView);
    dialog->show();
}

void MainWindow::onDeviceChanged(const QString &device) {
    statusLabel->setText(QString("设备已切换到: %1").arg(device));
    logTextEdit->append(QString("设备设置为: %1").arg(device));
}

void MainWindow::onModelChanged(const QString &model) {
    statusLabel->setText(QString("模型已切换到: %1").arg(model));
    logTextEdit->append(QString("模型设置为: %1").arg(model));
}


void MainWindow::onSaveModelClicked() {
    QString modelName = modelComboBox->currentText();
    QString fileFilter = "PyTorch Model (*.pt);;ONNX Model (*.onnx)";
    QString savePath = QFileDialog::getSaveFileName(this, "保存模型", "", fileFilter);

    if (savePath.isEmpty()) {
        return; // 用户取消了保存操作
    }

    torch::Tensor dummy_input;
    if (modelName == "CNN_MNIST") {
        dummy_input = torch::randn({1, 1, 28, 28}); // MNIST 输入尺寸
        if (savePath.endsWith(".pt")) {
            m_mnistModel->save(savePath.toStdString());
            QMessageBox::information(this, "保存成功", "模型以 .pt 格式保存成功！");
        } else if (savePath.endsWith(".onnx")) {
            try {
                //m_mnistModel->saveAsONNX(savePath.toStdString(), dummy_input);
                QMessageBox::information(this, "保存成功", "模型以 .onnx 格式保存成功！
");
            } catch (const std::exception& e) {
                QMessageBox::critical(this, "保存失败", QString("保存为 ONNX 格式失
败: %1").arg(e.what()));
            }
        }
```

```
    } else if (modelName == "CNN_CIFAR10") {
        dummy_input = torch::randn({1, 3, 32, 32}); // CIFAR10 输入尺寸
        if (savePath.endsWith(".pt")) {
            m_cifar10Model->save(savePath.toStdString());
            QMessageBox::information(this, "保存成功", "模型以 .pt 格式保存成功！");
        } else if (savePath.endsWith(".onnx")) {
            try {
                //m_cifar10Model->saveAsONNX(savePath.toStdString(), dummy_input);
                QMessageBox::information(this, "保存成功", "模型以 .onnx 格式保存成功！
");
            } catch (const std::exception& e) {
                QMessageBox::critical(this, "保存失败", QString("保存为 ONNX 格式失
败: %1").arg(e.what()));
            }
        }
    } else if (modelName == "CNN_CIFAR100") {
        dummy_input = torch::randn({1, 3, 32, 32}); // CIFAR100 输入尺寸
        if (savePath.endsWith(".pt")) {
            m_cifar100Model->save(savePath.toStdString());
            QMessageBox::information(this, "保存成功", "模型以 .pt 格式保存成功！");
        } else if (savePath.endsWith(".onnx")) {
            try {
                //m_cifar100Model->saveAsONNX(savePath.toStdString(), dummy_input);
                QMessageBox::information(this, "保存成功", "模型以 .onnx 格式保存成功！
");
            } catch (const std::exception& e) {
                QMessageBox::critical(this, "保存失败", QString("保存为 ONNX 格式失
败: %1").arg(e.what()));
            }
        }
    }
}
// void MainWindow::onSaveModelClicked() {
//     QString modelName = modelComboBox->currentText();
//     if (modelName.isEmpty()) {
//         QMessageBox::warning(this, "错误", "请选择模型");
//         return;
//     }

//     QString savePath = QFileDialog::getSaveFileName(
//         this, "保存模型参数", QDir::homePath(), "Model Files (*.pt)");
//     if (savePath.isEmpty()) {
//         logTextEdit->append("用户取消保存模型参数");
//         return;
```

```
//         }

//         logTextEdit->append(QString("=== 开始保存 %1 模型参数 ===").arg(modelName));
//         try {
//             if (modelName == "CNN_MNIST") {
//                 m_mnistModel->save(savePath.toStdString());
//             }
//             else if (modelName == "CNN_CIFAR10") {
//                 m_cifar10Model->save(savePath.toStdString());
//             }
//             else if (modelName == "CNN_CIFAR100") {
//                 m_cifar100Model->save(savePath.toStdString());
//             }
//                         logTextEdit->append(QString("成功保存 %1 模型参数
到：%2").arg(modelName).arg(savePath));
//         } catch (const exception& e) {
//             logTextEdit->append(QString("保存失败：%1").arg(e.what()));
//         }
// }

void MainWindow::onLoadModelClicked() {
    QString modelName = modelComboBox->currentText();
    if (modelName.isEmpty()) {
        QMessageBox::warning(this, "错误", "请选择模型");
        return;
    }

    QString loadPath = QFileDialog::getOpenFileName(this, "加载模型参数", QDir::homePath(),
"Model Files (*.pt)");
    if (loadPath.isEmpty()) {
        logTextEdit->append("用户取消加载模型参数");
        return;
    }

    logTextEdit->append(QString("=== 开始加载 %1 模型参数 ===").arg(modelName));
    try {
        if (modelName == "CNN_MNIST") {
            m_mnistModel->load(loadPath.toStdString());
        }
        else if (modelName == "CNN_CIFAR10") {
            m_cifar10Model->load(loadPath.toStdString());
        }
        else if (modelName == "CNN_CIFAR100") {
            m_cifar100Model->load(loadPath.toStdString());
```

```
        }
        logTextEdit->append(QString("成功加载 %1 模型参数，路径：%2").arg(modelName).arg(loadPath));
    } catch (const exception& e) {
        logTextEdit->append(QString("加载失败：%1").arg(e.what()));
    }
}

void MainWindow::onTrainingStarted() {
    progressBar->setValue(0);
    statusLabel->setText("训练中...");
    trainButton->setEnabled(false);
    testButton->setEnabled(false);
    useModelButton->setEnabled(false);
    modelComboBox->setEnabled(false);
}

void MainWindow::onTrainingProgress(int progress) {
    progressBar->setValue(progress);
}

void MainWindow::onTrainingEpochCompleted(int total, int epoch, double loss, double accuracy)
{
    logTextEdit->append(QString("Epoch %1/%2: Loss = %3, Accuracy = %4%").arg(epoch).arg(total).arg(loss, 0, 'f', 4).arg(accuracy * 100.0, 0, 'f', 2));
    int progress = 100 * (epoch) / total;
    progressBar->setValue(progress);
    QString modelName = modelComboBox->currentText();
    if (modelName == "CNN_MNIST") {
        currentAccuracy.append(accuracy * 100.0);
        currentEpochs.append(epoch);
    }
    else if (modelName == "CNN_CIFAR10") {
        currentAccuracy_cifar10.append(accuracy * 100.0);
        currentEpochs_cifar10.append(epoch);
    }
    else if (modelName == "CNN_CIFAR100") {
        currentAccuracy_cifar100.append(accuracy * 100.0);
        currentEpochs_cifar100.append(epoch);
    }
}

void MainWindow::onTrainingFinished() {
    statusLabel->setText("训练完成");
```

```
    progressBar->setValue(100);
    trainButton->setEnabled(true);
    testButton->setEnabled(true);
    useModelButton->setEnabled(true);
    modelComboBox->setEnabled(true);
    logTextEdit->append("=== 训练完成 ===");

    QString modelName = modelComboBox->currentText();
    if (modelName == "CNN_MNIST") {
        if (!this->currentEpochs.isEmpty() && !this->currentAccuracy.isEmpty()) {
            allEpochList.append(this->currentEpochs);
            allAccuracyHistory.append(this->currentAccuracy);

            QColor newColor = QColor::fromHsvF(fmod(trainRecordColors.size() * 0.15, 1.0),
0.7, 0.9);
            trainRecordColors.append(newColor);
        }

        currentEpochs.clear();
        currentAccuracy.clear();
    }
    else if (modelName == "CNN_CIFAR10") {
        if                                          (!this->currentEpochs_cifar10.isEmpty()
&& !this->currentAccuracy_cifar10.isEmpty()) {
            allEpochList_cifar10.append(this->currentEpochs_cifar10);
            allAccuracyHistory_cifar10.append(this->currentAccuracy_cifar10);

            QColor newColor = QColor::fromHsvF(fmod(trainRecordColors_cifar10.size() *
0.333, 1.0), 0.7, 0.9);
            trainRecordColors_cifar10.append(newColor);
        }

        currentEpochs_cifar10.clear();
        currentAccuracy_cifar10.clear();
    }
    else if (modelName == "CNN_CIFAR100") {
        if                                          (!this->currentEpochs_cifar100.isEmpty()
&& !this->currentAccuracy_cifar100.isEmpty()) {
            allEpochList_cifar100.append(this->currentEpochs_cifar100);
            allAccuracyHistory_cifar100.append(this->currentAccuracy_cifar100);

            QColor newColor = QColor::fromHsvF(fmod(trainRecordColors_cifar100.size() *
0.667, 1.0), 0.7, 0.9);
            trainRecordColors_cifar100.append(newColor);
```

```
            }

            currentEpochs_cifar100.clear();
            currentAccuracy_cifar100.clear();
        }
}

void MainWindow::onTrainingErrorOccurred(const QString &msg) {
        QMessageBox::critical(this, "TestError", msg);
        statusLabel->setText("就绪");
        trainButton->setEnabled(true);
        testButton->setEnabled(true);
        useModelButton->setEnabled(true);
        modelComboBox->setEnabled(true);
}

void MainWindow::onTestStart() {
        statusLabel->setText("测试中...");
        trainButton->setEnabled(true);
        testButton->setEnabled(true);
        useModelButton->setEnabled(true);
        modelComboBox->setEnabled(true);
}

void MainWindow::onTestFinished(double acc) {
        statusLabel->setText(QString("测试完成，精度： %1 %2").arg(acc * 100.0, 0, 'f', 2).arg('%'));
        trainButton->setEnabled(true);
        testButton->setEnabled(true);
        useModelButton->setEnabled(true);
        modelComboBox->setEnabled(true);
        logTextEdit->append(QString("Test finished, accuracy: %1 %2").arg(acc * 100.0, 0, 'f',
2).arg('%'));
        logTextEdit->append("=== 测试完成 ===");
}

void MainWindow::onTestErrorOccurred(const QString &msg) {
        QMessageBox::critical(this, "TestError", msg);
        statusLabel->setText("测试失败");
        trainButton->setEnabled(true);
        testButton->setEnabled(true);
        useModelButton->setEnabled(true);
        modelComboBox->setEnabled(true);
}
```

mainwindow.h

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include "model_mnist.h"
#include "model_cifar.h"
#include "trainingthread.h"
#include "testthread.h"
#include "trainingthread_cifar.h"
#include "testthread_cifar.h"

#include <QMainWindow>
#include <QPushButton>
#include <QVBoxLayout>
#include <QHBoxLayout>
#include <QGroupBox>
#include <QComboBox>
#include <QtWidgets>
#include <QLabel>

constexpr int BUTTON_WIDTH = 150;
constexpr int BUTTON_HEIGHT = 30;

QT_BEGIN_NAMESPACE
namespace Ui {
class MainWindow;
}
QT_END_NAMESPACE

class MainWindow : public QMainWindow {
    Q_OBJECT

public:
    MainWindow(QWidget *parent = nullptr) : QMainWindow(parent) {
        setupUI();

        m_mnistModel = new CNN_MNIST(10, 1);
        m_cifar10Model = new CNN_CIFAR10(10, 3);
        m_cifar100Model = new CNN_CIFAR100(100, 3);

        m_trainingThread = new TrainingThread(m_mnistModel, this);
        m_trainingThreadCIFAR10 = new TrainingThreadCIFAR10(m_cifar10Model, this);
        m_trainingThreadCIFAR100 = new TrainingThreadCIFAR100(m_cifar100Model, this);
```

```
        m_testThread = new TestThread(m_mnistModel, torch::kCUDA, 100, this);
        m_testThreadCIFAR10 = new TestThreadCIFAR10(m_cifar10Model, torch::kCUDA,
100, this);
        m_testThreadCIFAR100 = new TestThreadCIFAR100(m_cifar100Model, torch::kCUDA,
100, this);

        setupConnections();
    }

public slots:
    void onTrainButtonClicked();
    void onTestButtonClicked();
    void onUseModelButtonClicked();
    void onShowDataButtonClicked();

    void onDeviceChanged(const QString &device);

    void onModelChanged(const QString &model);
    void onLoadModelClicked();
    void onSaveModelClicked();

    void onTrainingStarted();
    void onTrainingProgress(int progress);
    void onTrainingEpochCompleted(int total, int epoch, double loss, double accuracy);
    void onTrainingFinished();
    void onTrainingErrorOccurred(const QString &msg);

    void onTestStart();
    void onTestFinished(double acc);
    void onTestErrorOccurred(const QString &msg);

private:
    void setupUI();

    void setupConnections();

    QComboBox *modelComboBox;
    QComboBox *deviceComboBox;
    QLineEdit *epochsLineEdit;
    QLineEdit *batchesLineEdit;
    QPushButton *trainButton;
    QPushButton *testButton;
    QPushButton *useModelButton;
    QPushButton *clearLogButton;
```

```
    QPushButton *showDataButton;
    QPushButton *saveModelButton;
    QPushButton *loadModelButton;
    QLabel *statusLabel;
    QProgressBar *progressBar;
    QTextEdit *logTextEdit;

    QList<QList<int>> allEpochList;
    QList<QList<double>> allAccuracyHistory;
    QList<QColor> trainRecordColors;
    QList<double> currentAccuracy;
    QList<int> currentEpochs;

    QList<QList<int>> allEpochList_cifar10;
    QList<QList<double>> allAccuracyHistory_cifar10;
    QList<QColor> trainRecordColors_cifar10;
    QList<double> currentAccuracy_cifar10;
    QList<int> currentEpochs_cifar10;

    QList<QList<int>> allEpochList_cifar100;
    QList<QList<double>> allAccuracyHistory_cifar100;
    QList<QColor> trainRecordColors_cifar100;
    QList<double> currentAccuracy_cifar100;
    QList<int> currentEpochs_cifar100;

    TrainingThread *m_trainingThread;
    TrainingThreadCIFAR10 *m_trainingThreadCIFAR10;
    TrainingThreadCIFAR100 *m_trainingThreadCIFAR100;

    TestThread *m_testThread;
    TestThreadCIFAR10 *m_testThreadCIFAR10;
    TestThreadCIFAR100 *m_testThreadCIFAR100;

    CNN_MNIST *m_mnistModel;
    CNN_CIFAR10 *m_cifar10Model;
    CNN_CIFAR100 *m_cifar100Model;
};

#endif // MAINWINDOW_H
```

main.cpp

```
#include "mainwindow.h"

#include <QApplication>
```

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();
    return a.exec();
}
```

model_cifar.cpp

```
#include "model_cifar.h"
#include "cifardataset.h"
#include <filesystem>

CNN_CIFAR10::CNN_CIFAR10(int num_classes, int input_channels) : m_device(torch::kCUDA)
{
    try {
        conv1_1 = register_module("conv1_1", torch::nn::Conv2d(torch::nn::Conv2dOptions(input_channels, 64, 3).padding(1)));
        bn1_1 = register_module("bn1_1", torch::nn::BatchNorm2d(64));
        conv1_2 = register_module("conv1_2", torch::nn::Conv2d(torch::nn::Conv2dOptions(64, 64, 3).padding(1)));
        bn1_2 = register_module("bn1_2", torch::nn::BatchNorm2d(64));

        conv2_1 = register_module("conv2_1", torch::nn::Conv2d(torch::nn::Conv2dOptions(64, 128, 3).padding(1).stride(2)));
        bn2_1 = register_module("bn2_1", torch::nn::BatchNorm2d(128));
        conv2_2 = register_module("conv2_2", torch::nn::Conv2d(torch::nn::Conv2dOptions(128, 128, 3).padding(1)));
        bn2_2 = register_module("bn2_2", torch::nn::BatchNorm2d(128));
        conv2_3 = register_module("conv2_3", torch::nn::Conv2d(torch::nn::Conv2dOptions(128, 128, 3).padding(1)));
        bn2_3 = register_module("bn2_3", torch::nn::BatchNorm2d(128));

        conv3_1 = register_module("conv3_1", torch::nn::Conv2d(torch::nn::Conv2dOptions(128, 256, 3).padding(1).stride(2)));
        bn3_1 = register_module("bn3_1", torch::nn::BatchNorm2d(256));
        conv3_2 = register_module("conv3_2", torch::nn::Conv2d(torch::nn::Conv2dOptions(256, 256, 3).padding(1)));
        bn3_2 = register_module("bn3_2", torch::nn::BatchNorm2d(256));
        conv3_3 = register_module("conv3_3", torch::nn::Conv2d(torch::nn::Conv2dOptions(256, 256, 3).padding(1)));
        bn3_3 = register_module("bn3_3", torch::nn::BatchNorm2d(256));
```

```
        conv4_1 = register_module("conv4_1", torch::nn::Conv2d(torch::nn::Conv2dOptions(256, 512, 3).padding(1)));
        bn4_1 = register_module("bn4_1", torch::nn::BatchNorm2d(512));
        conv4_2 = register_module("conv4_2", torch::nn::Conv2d(torch::nn::Conv2dOptions(512, 512, 3).padding(1)));
        bn4_2 = register_module("bn4_2", torch::nn::BatchNorm2d(512));

        residual1 = register_module("residual1", torch::nn::Conv2d(torch::nn::Conv2dOptions(64, 128, 1).stride(2)));
        residual_bn1 = register_module("residual_bn1", torch::nn::BatchNorm2d(128));
        residual2 = register_module("residual2", torch::nn::Conv2d(torch::nn::Conv2dOptions(128, 256, 1).stride(2)));
        residual_bn2 = register_module("residual_bn2", torch::nn::BatchNorm2d(256));

        global_pool = register_module("global_pool", torch::nn::AdaptiveAvgPool2d(1));
        attention_fc1 = register_module("attention_fc1", torch::nn::Linear(512, 128));
        attention_fc2 = register_module("attention_fc2", torch::nn::Linear(128, 512));

        fc1 = register_module("fc1", torch::nn::Linear(512 * 4 * 4, 1024));
        fc2 = register_module("fc2", torch::nn::Linear(1024, 512));
        fc3 = register_module("fc3", torch::nn::Linear(512, num_classes));

        spatial_dropout = register_module("spatial_dropout", torch::nn::Dropout2d(0.1));
        dropout = register_module("dropout", torch::nn::Dropout(0.3));
    }
    catch (const exception& e) {
        cerr<<"ERROR CNN_CIFAR10"<<e.what()<<endl;
        throw;
    }

    if (torch::cuda::is_available()) {
        m_device = torch::kCUDA;
        cout << "CUDA is available! Using GPU auto." << endl;
    }
    else {
        m_device = torch::kCPU;
        cout << "CUDA not available. Using CPU auto." << endl;
    }

    this->to(m_device);
}

torch::Tensor CNN_CIFAR10::channel_attention(torch::Tensor x) {
    auto input = x;
```

```
    x = global_pool->forward(x);
    x = x.reshape({x.size(0), 512});
    x = torch::relu(attention_fc1->forward(x));
    x = torch::sigmoid(attention_fc2->forward(x));
    x = x.reshape({-1, 512, 1, 1});
    return input * x;
}

torch::Tensor CNN_CIFAR10::forward(torch::Tensor x) {
    try{
        auto residual = x;
        if (x.device() != m_device) {
            x = x.to(m_device);
            residual = residual.to(m_device);
        }

        x = torch::relu(bn1_1->forward(conv1_1->forward(x)));
        x = torch::relu(bn1_2->forward(conv1_2->forward(x)));
        x = torch::max_pool2d(x, 2);

        residual = x;
        residual = residual_bn1->forward(residual1->forward(residual));
        x = torch::relu(bn2_1->forward(conv2_1->forward(x)));
        x = torch::relu(bn2_2->forward(conv2_2->forward(x)));
        x = torch::relu(bn2_3->forward(conv2_3->forward(x)) + residual);
        x = spatial_dropout->forward(x);

        auto residual3 = x;
        residual3 = residual_bn2->forward(residual2->forward(residual3));
        x = torch::relu(bn3_1->forward(conv3_1->forward(x)));
        x = torch::relu(bn3_2->forward(conv3_2->forward(x)));
        x = torch::relu(bn3_3->forward(conv3_3->forward(x)) + residual3);
        x = spatial_dropout->forward(x);

        x = torch::relu(bn4_1->forward(conv4_1->forward(x)));
        x = torch::relu(bn4_2->forward(conv4_2->forward(x)));
        x = channel_attention(x);

        x = x.reshape({x.size(0), -1});

        x = torch::relu(fc1->forward(x));
        x = dropout->forward(x);

        x = torch::relu(fc2->forward(x));
```

```
        x = dropout->forward(x);

        x = fc3->forward(x);

        return x;
    } catch(exception &e){
        cerr<<"ERROR forward "<<e.what()<<endl;
    }
    return x;
}

void CNN_CIFAR10::train(torch::Device device, int epochs, int batch_size) {
    string root = "../../data/cifar-10-batches-bin";
    if (!filesystem::exists(root)) {
        cerr << "错误：数据集路径不存在！" << endl;
        return;
    }

    if (device == torch::kCUDA && torch::cuda::is_available()) {
        device = torch::kCUDA;
        cout << "CUDA is available! Using GPU." << endl;
    }
    else {
        device = torch::kCPU;
        cout << "CUDA not available. Using CPU." << endl;
    }
    m_device = device;

    this->to(m_device);

    auto train_dataset = CIFAR10Dataset(root, CIFARDataset::Mode::kTrain).map(torch::data::transforms::Stack<>());
    const size_t train_dataset_size = train_dataset.size().value();
    cout << "train_dataset_size: " << train_dataset_size << endl;
    auto train_loader = torch::data::make_data_loader<torch::data::samplers::SequentialSampler>(std::move(train_dataset), batch_size);

    torch::optim::Adam optimizer(this->parameters(), /*lr=*/0.001);

    cout << "Start training..." << endl;

    for (int epoch = 1; epoch <= epochs; ++epoch) {
        float running_loss = 0.0;
```

```
            int64_t correct = 0;
            int64_t total = 0;
            int64_t batch_count = 0;

            this->torch::nn::Module::train();

            for (auto const &batch : *train_loader) {
                auto data = batch.data.to(device);
                auto targets = batch.target.to(device);

                optimizer.zero_grad();

                auto output = this->forward(data);

                auto loss = torch::cross_entropy_loss(output, targets);

                loss.backward();

                optimizer.step();

                running_loss += loss.item<double>();
                auto predicted = output.argmax(1);
                total += targets.size(0);
                correct += predicted.eq(targets).sum().item<int64_t>();

                batch_count++;
            }

            cout << "Epoch [" << epoch << "/" << epochs << "], " << "Loss: " << fixed << setprecision(4) << running_loss / batch_count << ", " << "Accuracy: " << static_cast<double>(correct) / total << endl;

            emit epochCompleted(epochs, epoch, running_loss / batch_count, static_cast<double>(correct) / total);
        }
        cout << "Finish training!" << endl;
}

double CNN_CIFAR10::test(torch::Device device, int batch_size) {
        string root = "../../data/cifar-10-batches-bin";
        if (!filesystem::exists(root)) {
            cerr << "错误：数据集路径不存在！" << endl;
            return 0.0;
        }
```

```
    if (device == torch::kCUDA && torch::cuda::is_available()) {
        device = torch::kCUDA;
        cout << "CUDA is available! Using GPU." << endl;
    }
    else {
        device = torch::kCPU;
        cout << "CUDA not available. Using CPU." << endl;
    }
    m_device = device;

    this->to(m_device);

    auto test_dataset = CIFAR10Dataset(root, CIFARDataset::Mode::kTest).map(torch::data::transforms::Stack<>());
    const size_t test_dataset_size = test_dataset.size().value();
    cout << "test_dataset_size: " << test_dataset_size << endl;
    auto test_loader = torch::data::make_data_loader<torch::data::samplers::SequentialSampler>(std::move(test_dataset), batch_size);

    cout << "开始测试..." << endl;

    this->eval();

    int64_t correct = 0;
    int64_t total = 0;

    torch::NoGradGuard no_grad;

    for (const auto &batch : *test_loader) {
        auto data = batch.data.to(device);
        auto targets = batch.target.to(device);

        auto output = this->forward(data);
        auto predicted = output.argmax(1);

        total += targets.size(0);
        correct += predicted.eq(targets).sum().item<int64_t>();
    }

    double accuracy = static_cast<double>(correct) / total;
    cout << "准确率: " << accuracy << endl;
```

```
    return accuracy;
}

int CNN_CIFAR10::predict(torch::Tensor input) {
    this->eval();
    torch::Tensor output = forward(input);
    if (m_device == torch::kCUDA) {
        output = output.to(torch::kCPU);
    }
    return output.argmax(1).item<int>();
}

void CNN_CIFAR10::save(torch::serialize::OutputArchive& archive) const {
    torch::nn::Module::save(archive);
    archive.write("device", m_device.type() == torch::kCUDA ? "cuda" : "cpu");
}

void CNN_CIFAR10::load(torch::serialize::InputArchive& archive) {
    torch::nn::Module::load(archive);
    c10::IValue ivalue_device;
    if (archive.try_read("device", ivalue_device)) {
        if (ivalue_device.isString()) {
            string device_str = ivalue_device.toStringRef();
            m_device = (device_str == "cuda" && torch::cuda::is_available()) ? torch::kCUDA :
torch::kCPU;
            this->to(m_device);
        }
    }
}

void CNN_CIFAR10::save(const string& path) {
    try {
        torch::serialize::OutputArchive archive;
        torch::nn::Module::save(archive);
        archive.write("device", m_device.type() == torch::kCUDA ? "cuda" : "cpu");
        archive.save_to(path);
        cout << "CNN_CIFAR10 模型参数保存成功，路径：" << path << endl;
    } catch (const exception& e) {
        cerr << "保存模型参数失败：" << e.what() << endl;
        throw;
    }
}

void CNN_CIFAR10::load(const string& path) {
```

```
    try {
        if (!filesystem::exists(path)) {
            throw runtime_error("文件不存在：" + path);
        }
        torch::serialize::InputArchive archive;
        archive.load_from(path);
        torch::nn::Module::load(archive);
        c10::IValue device_str;
        if (archive.try_read("device", device_str)) {
            m_device = (device_str == "cuda" && torch::cuda::is_available()) ?
                        torch::kCUDA : torch::kCPU;
            this->to(m_device);
        }
        cout << "CNN_CIFAR10 模型参数加载成功，路径：" << path << endl;
    } catch (const exception& e) {
        cerr << "加载模型参数失败：" << e.what() << endl;
        throw;
    }
}

torch::Device CNN_CIFAR10::device() const {
    return m_device;
}

CNN_CIFAR100::CNN_CIFAR100(int num_classes, int input_channels) :
m_device(torch::kCUDA) {
    try {
        conv1_1 = register_module("conv1_1",
torch::nn::Conv2d(torch::nn::Conv2dOptions(input_channels, 64, 3).padding(1)));
        bn1_1 = register_module("bn1_1", torch::nn::BatchNorm2d(64));
        conv1_2 = register_module("conv1_2", torch::nn::Conv2d(torch::nn::Conv2dOptions(64,
64, 3).padding(1)));
        bn1_2 = register_module("bn1_2", torch::nn::BatchNorm2d(64));

        conv2_1 = register_module("conv2_1", torch::nn::Conv2d(torch::nn::Conv2dOptions(64,
128, 3).padding(1).stride(2)));
        bn2_1 = register_module("bn2_1", torch::nn::BatchNorm2d(128));
        conv2_2 = register_module("conv2_2",
torch::nn::Conv2d(torch::nn::Conv2dOptions(128, 128, 3).padding(1)));
        bn2_2 = register_module("bn2_2", torch::nn::BatchNorm2d(128));
        conv2_3 = register_module("conv2_3",
torch::nn::Conv2d(torch::nn::Conv2dOptions(128, 128, 3).padding(1)));
        bn2_3 = register_module("bn2_3", torch::nn::BatchNorm2d(128));
```

```
        conv3_1 = register_module("conv3_1", torch::nn::Conv2d(torch::nn::Conv2dOptions(128, 256, 3).padding(1).stride(2)));
        bn3_1 = register_module("bn3_1", torch::nn::BatchNorm2d(256));
        conv3_2 = register_module("conv3_2", torch::nn::Conv2d(torch::nn::Conv2dOptions(256, 256, 3).padding(1)));
        bn3_2 = register_module("bn3_2", torch::nn::BatchNorm2d(256));
        conv3_3 = register_module("conv3_3", torch::nn::Conv2d(torch::nn::Conv2dOptions(256, 256, 3).padding(1)));
        bn3_3 = register_module("bn3_3", torch::nn::BatchNorm2d(256));

        conv4_1 = register_module("conv4_1", torch::nn::Conv2d(torch::nn::Conv2dOptions(256, 512, 3).padding(1)));
        bn4_1 = register_module("bn4_1", torch::nn::BatchNorm2d(512));
        conv4_2 = register_module("conv4_2", torch::nn::Conv2d(torch::nn::Conv2dOptions(512, 512, 3).padding(1)));
        bn4_2 = register_module("bn4_2", torch::nn::BatchNorm2d(512));

        residual1 = register_module("residual1", torch::nn::Conv2d(torch::nn::Conv2dOptions(64, 128, 1).stride(2)));
        residual_bn1 = register_module("residual_bn1", torch::nn::BatchNorm2d(128));
        residual2 = register_module("residual2", torch::nn::Conv2d(torch::nn::Conv2dOptions(128, 256, 1).stride(2)));
        residual_bn2 = register_module("residual_bn2", torch::nn::BatchNorm2d(256));

        global_pool = register_module("global_pool", torch::nn::AdaptiveAvgPool2d(1));
        attention_fc1 = register_module("attention_fc1", torch::nn::Linear(512, 128));
        attention_fc2 = register_module("attention_fc2", torch::nn::Linear(128, 512));

        fc1 = register_module("fc1", torch::nn::Linear(512 * 4 * 4, 1024));
        fc2 = register_module("fc2", torch::nn::Linear(1024, 512));
        fc3 = register_module("fc3", torch::nn::Linear(512, num_classes));

        spatial_dropout = register_module("spatial_dropout", torch::nn::Dropout2d(0.1));
        dropout = register_module("dropout", torch::nn::Dropout(0.3));
    } catch (const exception& e) {
        cerr<<"ERROR CNN_CIFAR100"<<e.what()<<endl;
        throw;
    }

    if (torch::cuda::is_available()) {
        m_device = torch::kCUDA;
        cout << "CUDA is available! Using GPU auto." << endl;
    }
    else {
```

```
        m_device = torch::kCPU;
        cout << "CUDA not available. Using CPU auto." << endl;
    }

    this->to(m_device);
}

torch::Tensor CNN_CIFAR100::channel_attention(torch::Tensor x) {
    auto input = x;
    x = global_pool->forward(x);
    x = x.reshape({x.size(0), 512});
    x = torch::relu(attention_fc1->forward(x));
    x = torch::sigmoid(attention_fc2->forward(x));
    x = x.reshape({-1, 512, 1, 1});
    return input * x;
}

torch::Tensor CNN_CIFAR100::forward(torch::Tensor x) {
    try{
        auto residual = x;
        if (x.device() != m_device) {
            x = x.to(m_device);
            residual = residual.to(m_device);
        }

        x = torch::relu(bn1_1->forward(conv1_1->forward(x)));
        x = torch::relu(bn1_2->forward(conv1_2->forward(x)));
        x = torch::max_pool2d(x, 2);

        residual = x;
        residual = residual_bn1->forward(residual1->forward(residual));
        x = torch::relu(bn2_1->forward(conv2_1->forward(x)));
        x = torch::relu(bn2_2->forward(conv2_2->forward(x)));
        x = torch::relu(bn2_3->forward(conv2_3->forward(x)) + residual);
        x = spatial_dropout->forward(x);

        auto residual3 = x;
        residual3 = residual_bn2->forward(residual2->forward(residual3));
        x = torch::relu(bn3_1->forward(conv3_1->forward(x)));
        x = torch::relu(bn3_2->forward(conv3_2->forward(x)));
        x = torch::relu(bn3_3->forward(conv3_3->forward(x)) + residual3);
        x = spatial_dropout->forward(x);

        x = torch::relu(bn4_1->forward(conv4_1->forward(x)));
```

```
        x = torch::relu(bn4_2->forward(conv4_2->forward(x)));
        x = channel_attention(x);

        x = x.reshape({x.size(0), -1});

        x = torch::relu(fc1->forward(x));
        x = dropout->forward(x);

        x = torch::relu(fc2->forward(x));
        x = dropout->forward(x);

        x = fc3->forward(x);

        return x;
    }
    catch(exception &e){
        cerr<<"ERROR forward"<<e.what()<<endl;
    }
    return x;
}

void CNN_CIFAR100::train(torch::Device device, int epochs, int batch_size) {
    string root = "../../data/cifar-100-binary";
    if (!filesystem::exists(root)) {
        cerr << "错误：数据集路径不存在！" << endl;
        return;
    }

    if (device == torch::kCUDA && torch::cuda::is_available()) {
        device = torch::kCUDA;
        cout << "CUDA is available! Using GPU." << endl;
    }
    else {
        device = torch::kCPU;
        cout << "CUDA not available. Using CPU." << endl;
    }
    m_device = device;

    this->to(m_device);

    auto train_dataset = CIFAR100Dataset(root,
CIFARDataset::Mode::kTrain).map(torch::data::transforms::Stack<>());
    const size_t train_dataset_size = train_dataset.size().value();
    cout << "train_dataset_size: " << train_dataset_size << endl;
```

```
    auto train_loader = torch::data::make_data_loader<torch::data::samplers::SequentialSampler>(std::move(train_dataset), batch_size);

    torch::optim::Adam optimizer(this->parameters(), /*lr=*/0.001);

    cout << "Start training..." << endl;

    for (int epoch = 1; epoch <= epochs; ++epoch) {
        float running_loss = 0.0;
        int64_t correct = 0;
        int64_t total = 0;
        int64_t batch_count = 0;

        this->torch::nn::Module::train();

        for (auto const &batch : *train_loader) {
            auto data = batch.data.to(device);
            auto targets = batch.target.to(device);

            optimizer.zero_grad();

            auto output = this->forward(data);

            auto loss = torch::cross_entropy_loss(output, targets);

            loss.backward();

            optimizer.step();

            running_loss += loss.item<double>();
            auto predicted = output.argmax(1);
            total += targets.size(0);
            correct += predicted.eq(targets).sum().item<int64_t>();

            batch_count++;
        }

        cout << "Epoch [" << epoch << "/" << epochs << "], " << "Loss: " << fixed << setprecision(4) << running_loss / batch_count << ", " << "Accuracy: " << static_cast<double>(correct) / total << endl;

        emit epochCompleted(epochs, epoch, running_loss / batch_count, static_cast<double>(correct) / total);
```

```
    }
    cout << "Finish training!" << endl;
}

double CNN_CIFAR100::test(torch::Device device, int batch_size) {
    string root = "../../data/cifar-100-binary";
    if (!filesystem::exists(root)) {
        cerr << "CNN_CIFAR100::test PATH FAIL!" << endl;
        return 0.0;
    }

    if (device == torch::kCUDA && torch::cuda::is_available()) {
        device = torch::kCUDA;
        cout << "CUDA is available! Using GPU." << endl;
    }
    else {
        device = torch::kCPU;
        cout << "CUDA not available. Using CPU." << endl;
    }
    m_device = device;

    this->to(m_device);

    auto test_dataset = CIFAR100Dataset(root, CIFARDataset::Mode::kTest).map(torch::data::transforms::Stack<>());
    const size_t test_dataset_size = test_dataset.size().value();
    cout << "test_dataset_size: " << test_dataset_size << endl;
    auto test_loader = torch::data::make_data_loader<torch::data::samplers::SequentialSampler>(std::move(test_dataset), batch_size);

    cout << "开始测试..." << endl;

    this->eval();

    int64_t correct = 0;
    int64_t total = 0;

    torch::NoGradGuard no_grad;

    for (const auto &batch : *test_loader) {
        auto data = batch.data.to(device);
        auto targets = batch.target.to(device);
```

```
            auto output = this->forward(data);
            auto predicted = output.argmax(1);

            total += targets.size(0);
            correct += predicted.eq(targets).sum().item<int64_t>();
        }

        double accuracy = static_cast<double>(correct) / total;
        cout << "准确率: " << accuracy << endl;

        return accuracy;
}

int CNN_CIFAR100::predict(torch::Tensor input) {
        this->eval();
        torch::Tensor output = forward(input);
        if (m_device == torch::kCUDA) {
            output = output.to(torch::kCPU);
        }
        return output.argmax(1).item<int>();
}

void CNN_CIFAR100::save(torch::serialize::OutputArchive& archive) const {
        torch::nn::Module::save(archive);
        archive.write("device", m_device.type() == torch::kCUDA ? "cuda" : "cpu");
}

void CNN_CIFAR100::load(torch::serialize::InputArchive& archive) {
        torch::nn::Module::load(archive);
        c10::IValue ivalue_device;
        if (archive.try_read("device", ivalue_device)) {
            if (ivalue_device.isString()) {
                string device_str = ivalue_device.toStringRef();
                m_device = (device_str == "cuda" && torch::cuda::is_available()) ? torch::kCUDA :
torch::kCPU;
                this->to(m_device);
            }
        }
}

void CNN_CIFAR100::save(const string& path) {
        try {
            torch::serialize::OutputArchive archive;
            torch::nn::Module::save(archive);
```

```
        archive.write("device", m_device.type() == torch::kCUDA ? "cuda" : "cpu");
        archive.save_to(path);
        cout << "CNN_CIFAR100 模型参数保存成功，路径：" << path << endl;
    } catch (const exception& e) {
        cerr << "保存模型参数失败：" << e.what() << endl;
        throw;
    }
}

void CNN_CIFAR100::load(const string& path) {
    try {
        if (!filesystem::exists(path)) {
            throw runtime_error("文件不存在：" + path);
        }

        torch::serialize::InputArchive archive;
        archive.load_from(path);
        torch::nn::Module::load(archive);

        c10::IValue device_str;
        if (archive.try_read("device", device_str)) {
            m_device = (device_str == "cuda" && torch::cuda::is_available()) ?
                                torch::kCUDA : torch::kCPU;
            this->to(m_device);
        }
        cout << "CNN_CIFAR100 模型参数加载成功，路径：" << path << endl;
    } catch (const exception& e) {
        cerr << "加载模型参数失败：" << e.what() << endl;
        throw;
    }
}

torch::Device CNN_CIFAR100::device() const {
    return m_device;
}

string CNN_CIFAR10::classify(int result) {
    const vector<string> CIFAR10_CLASSES = {
        "airplane", "automobile", "bird", "cat", "deer",
        "dog", "frog", "horse", "ship", "truck"
    };
    return CIFAR10_CLASSES[result];
}
```

```
string CNN_CIFAR100::classify(int result) {
    const vector<string> CIFAR100_CLASSES = {
        "apple", "aquarium_fish", "baby", "bear", "beaver", "bed", "bee", "beetle",
        "bicycle", "bottle", "bowl", "boy", "bridge", "bus", "butterfly", "camel",
        "can", "castle", "caterpillar", "cattle", "chair", "chimpanzee", "clock",
        "cloud", "cockroach", "couch", "crab", "crocodile", "cup", "dinosaur",
        "dolphin", "elephant", "flatfish", "forest", "fox", "girl", "hamster",
        "house", "kangaroo", "keyboard", "lamp", "lawn_mower", "leopard", "lion",
        "lizard", "lobster", "man", "maple_tree", "motorcycle", "mountain", "mouse",
        "mushroom", "oak_tree", "orange", "orchid", "otter", "palm_tree", "pear",
        "pickup_truck", "pine_tree", "plain", "plate", "poppy", "porcupine",
        "possum", "rabbit", "raccoon", "ray", "road", "rocket", "rose",
        "sea", "seal", "shark", "shrew", "skunk", "skyscraper", "snail", "snake",
        "spider", "squirrel", "streetcar", "sunflower", "sweet_pepper", "table",
        "tank", "telephone", "television", "tiger", "tractor", "train", "trout",
        "tulip", "turtle", "wardrobe", "whale", "willow_tree", "wolf", "woman",
        "worm"
    };
    return CIFAR100_CLASSES[result];
}
```

model_cifar.h

```
#ifndef MODEL_CIFAR_H
#define MODEL_CIFAR_H

#undef slots
#include <torch/torch.h>
#include <torch/serialize/archive.h>
#define slots Q_SLOTS

#include <QObject>

using namespace std;

class CNN_CIFAR10 : public QObject, public torch::nn::Module {
    Q_OBJECT
public:
    CNN_CIFAR10(int num_classes, int input_channels);
    torch::Tensor forward(torch::Tensor x);
    void train(torch::Device device, int epochs, int batch_size);
    double test(torch::Device device, int batch_size);
    int predict(torch::Tensor input);
    torch::Device device() const;
```

```
    static string classify(int result);

    void save(torch::serialize::OutputArchive& archive) const override;
    void load(torch::serialize::InputArchive& archive) override;
    void save(const string& path);
    void load(const string& path);

signals:
    void epochCompleted(int total, int epoch, double loss, double accuracy);

private:
    torch::nn::Conv2d conv1_1{nullptr};
    torch::nn::BatchNorm2d bn1_1{nullptr};
    torch::nn::Conv2d conv1_2{nullptr};
    torch::nn::BatchNorm2d bn1_2{nullptr};

    torch::nn::Conv2d conv2_1{nullptr};
    torch::nn::BatchNorm2d bn2_1{nullptr};
    torch::nn::Conv2d conv2_2{nullptr};
    torch::nn::BatchNorm2d bn2_2{nullptr};
    torch::nn::Conv2d conv2_3{nullptr};
    torch::nn::BatchNorm2d bn2_3{nullptr};

    torch::nn::Conv2d conv3_1{nullptr};
    torch::nn::BatchNorm2d bn3_1{nullptr};
    torch::nn::Conv2d conv3_2{nullptr};
    torch::nn::BatchNorm2d bn3_2{nullptr};
    torch::nn::Conv2d conv3_3{nullptr};
    torch::nn::BatchNorm2d bn3_3{nullptr};

    torch::nn::Conv2d conv4_1{nullptr};
    torch::nn::BatchNorm2d bn4_1{nullptr};
    torch::nn::Conv2d conv4_2{nullptr};
    torch::nn::BatchNorm2d bn4_2{nullptr};

    torch::nn::Conv2d residual1{nullptr};
    torch::nn::BatchNorm2d residual_bn1{nullptr};
    torch::nn::Conv2d residual2{nullptr};
    torch::nn::BatchNorm2d residual_bn2{nullptr};

    torch::nn::Linear fc1{nullptr};
    torch::nn::Linear fc2{nullptr};
    torch::nn::Linear fc3{nullptr};
```

```
    torch::nn::Dropout2d spatial_dropout{nullptr};
    torch::nn::Dropout dropout{nullptr};

    torch::nn::AdaptiveAvgPool2d global_pool{nullptr};
    torch::nn::Linear attention_fc1{nullptr};
    torch::nn::Linear attention_fc2{nullptr};

    torch::Tensor channel_attention(torch::Tensor x);

    torch::Device m_device;
};

class CNN_CIFAR100 : public QObject, public torch::nn::Module {
    Q_OBJECT
public:
    CNN_CIFAR100(int num_classes, int input_channels);
    torch::Tensor forward(torch::Tensor x);
    void train(torch::Device device, int epochs, int batch_size);
    double test(torch::Device device, int batch_size);
    int predict(torch::Tensor input);
    torch::Device device() const;

    static string classify(int result);

    void save(torch::serialize::OutputArchive& archive) const override;
    void load(torch::serialize::InputArchive& archive) override;
    void save(const string& path);
    void load(const string& path);

signals:
    void epochCompleted(int total, int epoch, double loss, double accuracy);
    void errorOccurred(const QString& errorMessage);

private:
    torch::nn::Conv2d conv1_1{nullptr};
    torch::nn::BatchNorm2d bn1_1{nullptr};
    torch::nn::Conv2d conv1_2{nullptr};
    torch::nn::BatchNorm2d bn1_2{nullptr};

    torch::nn::Conv2d conv2_1{nullptr};
    torch::nn::BatchNorm2d bn2_1{nullptr};
    torch::nn::Conv2d conv2_2{nullptr};
    torch::nn::BatchNorm2d bn2_2{nullptr};
    torch::nn::Conv2d conv2_3{nullptr};
```

```
    torch::nn::BatchNorm2d bn2_3{nullptr};

    torch::nn::Conv2d conv3_1{nullptr};
    torch::nn::BatchNorm2d bn3_1{nullptr};
    torch::nn::Conv2d conv3_2{nullptr};
    torch::nn::BatchNorm2d bn3_2{nullptr};
    torch::nn::Conv2d conv3_3{nullptr};
    torch::nn::BatchNorm2d bn3_3{nullptr};

    torch::nn::Conv2d conv4_1{nullptr};
    torch::nn::BatchNorm2d bn4_1{nullptr};
    torch::nn::Conv2d conv4_2{nullptr};
    torch::nn::BatchNorm2d bn4_2{nullptr};

    torch::nn::Conv2d residual1{nullptr};
    torch::nn::BatchNorm2d residual_bn1{nullptr};
    torch::nn::Conv2d residual2{nullptr};
    torch::nn::BatchNorm2d residual_bn2{nullptr};

    torch::nn::Linear fc1{nullptr};
    torch::nn::Linear fc2{nullptr};
    torch::nn::Linear fc3{nullptr};

    torch::nn::Dropout2d spatial_dropout{nullptr};
    torch::nn::Dropout dropout{nullptr};

    torch::nn::AdaptiveAvgPool2d global_pool{nullptr};
    torch::nn::Linear attention_fc1{nullptr};
    torch::nn::Linear attention_fc2{nullptr};

    torch::Tensor channel_attention(torch::Tensor x);

    torch::Device m_device;
};

#endif // MODEL_CIFAR_H
```

trainingthread.cpp

```
#include "trainingthread.h"
#include <QThread>

TrainingThread::TrainingThread(CNN_MNIST* model, QObject* parent) : QThread(parent),
m_model(model), m_device(torch::kCUDA)
{
```

```
    if (!m_model) {
        emit errorOccurred("模型指针为空！");
        return;
    }

    connect(m_model, &CNN_MNIST::epochCompleted, [this](int total, int epoch, double loss, double accuracy) {
        emit epochCompleted(total, epoch, loss, accuracy);
    });
}

void TrainingThread::setParameters(const QString& modelName, const QString& device, int epochs, int batches) {
    m_modelName = modelName;
    m_deviceStr = device;
    m_epochs = epochs;
    m_batches = batches;

    m_device = (device == "NVIDIA GEFORCE RTX 4090 LapTop" && torch::cuda::is_available()) ? torch::kCUDA : torch::kCPU;
}

void TrainingThread::run() {
    if (m_epochs <= 0) {
        emit errorOccurred("训练轮数必须大于 0！");
        return;
    }

    try {
        emit trainingStarted();

        if (m_device == torch::kCUDA && !torch::cuda::is_available()) {
            emit errorOccurred("请求使用 CUDA 但设备不可用，自动切换至 CPU！");
            m_device = torch::kCPU;
            m_deviceStr = "CPU";
        }

        m_model->train(m_device, m_epochs, m_batches);

        emit trainingFinished();

    } catch (const c10::Error& e) {
        emit errorOccurred(QString("LibTorch 错误: %1").arg(e.what()));
    } catch (const exception& e) {
```

```
        emit errorOccurred(QString("训练错误: %1").arg(e.what()));
    } catch (...) {
        emit errorOccurred("未知错误");
    }
}
```

trainingthread.h

```
#ifndef TRAININGTHREAD_H
#define TRAININGTHREAD_H

#include <QThread>
#include <QString>
#undef slots
#include <torch/torch.h>
#define slots Q_SLOTS
#include "model_mnist.h"

class TrainingThread : public QThread {
    Q_OBJECT

public:
    explicit TrainingThread(CNN_MNIST* model, QObject* parent);
    void setParameters(const QString& modelName, const QString& device, int epochs, int batches);

signals:
    void trainingStarted();
    void progressUpdated(int progress);
    void epochCompleted(int total, int epoch, double loss, double accuracy);
    void trainingFinished();
    void errorOccurred(const QString& message);

protected:
    void run() override;

private:
    CNN_MNIST* m_model;
    QString m_modelName;
    QString m_deviceStr;
    int m_epochs;
    int m_batches;
    torch::Device m_device;
};
```

```
#endif // TRAININGTHREAD_H
```

testthread.cpp

```
#include "testthread.h"

TestThread::TestThread(CNN_MNIST *model, torch::Device device, int batches, QObject *parent)
    : QThread(parent), m_model(model), m_device(device), m_batches(batches)
{
    m_device = (device == torch::kCUDA && torch::cuda::is_available()) ? torch::kCUDA : torch::kCPU;
}

void TestThread::setModel(CNN_MNIST *model) {
    m_model = model;
}

void TestThread::setParameters(CNN_MNIST *model, QString device, int batches) {
    setModel(model);
    m_batches = batches;
    m_device = (device == "NVIDIA GEFORCE RTX 4090 LapTop" && torch::cuda::is_available()) ? torch::kCUDA : torch::kCPU;
}

void TestThread::run() {
    emit testStarted();

    try {
        emit testStarted();

        if (m_device == torch::kCUDA && !torch::cuda::is_available()) {
            emit errorOccurred("请求使用 CUDA 但设备不可用，自动切换至 CPU！");
            m_device = torch::kCPU;
        }

        double acc = m_model->test(m_device, m_batches);

        emit testFinished(acc);

    } catch (const c10::Error& e) {
        emit errorOccurred(QString("LibTorch 错误: %1").arg(e.what()));
    } catch (const std::exception& e) {
        emit errorOccurred(QString("Training 错误: %1").arg(e.what()));
    } catch (...) {
        emit errorOccurred("Unkown 错误");
```

```
    }
}
```

testthread.h

```
#ifndef TESTTHREAD_H
#define TESTTHREAD_H

#undef slots
#include <torch/torch.h>
#define slots Q_SLOTS
#include <QThread>
#include "model_mnist.h"

class TestThread : public QThread {
    Q_OBJECT

public:
    explicit TestThread(CNN_MNIST *model, torch::Device device, int batches, QObject *parent
= nullptr);
    void setModel(CNN_MNIST *model);
    void setParameters(CNN_MNIST *model, QString device, int batches);

signals:
    void testStarted();
    void testFinished(double acc);
    void errorOccurred(const QString &msg);

protected:
    void run() override;

private:
    CNN_MNIST *m_model;
    torch::Device m_device;
    int m_batches;
};

#endif // TESTTHREAD_H
```

cifardataset.cpp

```
#include "cifardataset.h"
#include <filesystem>
#include <fstream>

CIFARDataset::CIFARDataset(const string& root, Mode mode, const Options& options, int
```

```
num_classes)
    : m_device(options.device), m_normalize(options.normalize) {
    if (!filesystem::exists(root)) {
        throw runtime_error("Dataset path not found: " + root);
    }
    if (m_images.size() != m_labels.size()) {
        throw runtime_error("Image and label count mismatch");
    }
}

torch::data::Example<> CIFARDataset::get(size_t index) {
    auto image = m_images.at(index).to(m_device);
    auto label = torch::tensor(m_labels.at(index), torch::kInt64).to(m_device);
    return {image, label};
}

torch::Tensor CIFARDataset::fromCVImage(const cv::Mat& image, const Options& options) {
    if (image.channels() != 3) {
        cerr << "Error: Image must have 3 channels (BGR or RGB)" << endl;
    }

    vector<cv::Mat> channels;
    split(image, channels);

    namedWindow("Channels", cv::WINDOW_NORMAL);
    cv::resizeWindow("Channels", 1200, 400);

    cv::Mat display;
    hconcat(channels, display);

    putText(display, "Blue Channel", cv::Point(10, 30), cv::FONT_HERSHEY_SIMPLEX, 1,
cv::Scalar(255, 255, 255), 2);
    putText(display, "Green Channel", cv::Point(image.cols + 10, 30),
cv::FONT_HERSHEY_SIMPLEX, 1, cv::Scalar(255, 255, 255), 2);
    putText(display, "Red Channel", cv::Point(2 * image.cols + 10, 30),
cv::FONT_HERSHEY_SIMPLEX, 1, cv::Scalar(255, 255, 255), 2);

    imshow("Channels", display);
    cv::waitKey(0);

    cv::Mat resized;
    cv::resize(image, resized, cv::Size(32, 32));

    cv::cvtColor(resized, resized, cv::COLOR_BGR2RGB);
```

```
    resized.convertTo(resized, CV_32F, 1.0 / 255.0);

    return torch::from_blob(resized.data, {1, 32, 32, 3}, torch::kFloat32).permute({0, 3, 1, 2}).to(options.device);
}

CIFAR10Dataset::CIFAR10Dataset(const string& root, Mode mode, const Options& options)
    : CIFARDataset(root, mode, options, 10) {
    CIFAR10Dataset::parseBinaryFiles(root, mode);
    m_mean[0] = 0.5071f; m_mean[1] = 0.4867f; m_mean[2] = 0.4408f;
    m_std[0] = 0.2675f;    m_std[1] = 0.2565f;    m_std[2] = 0.2761f;
}

void CIFAR10Dataset::parseBinaryFiles(const string& root, Mode mode) {
    cout << "parseBinaryFiles in CIFAR10Dataset Class." << endl;

    const int IMAGE_SIZE = 32 * 32 * 3;
    vector<string> filenames;

    if (mode == Mode::kTrain) {
        for (int i = 1; i <= 5; ++i) {
            filenames.push_back(root + "/data_batch_" + to_string(i) + ".bin");
        }
    } else {
        filenames.push_back(root + "/test_batch" + ".bin");
    }

    for (const auto& file : filenames) {
        ifstream ifs(file, ios::binary);
        if (!filesystem::exists(root)) {
            cout << "Dataset path not found: " << root << endl;
            throw runtime_error("Dataset path not found: " + root);
        }
        if (!ifs.is_open()) {
            cout << "Failed to open file: " << file << endl;
            throw runtime_error("Failed to open file: " + file);
        }
        char label;
        vector<char> image_data(IMAGE_SIZE);

        while (ifs.read(&label, 1) && ifs.read(image_data.data(), IMAGE_SIZE)) {
            torch::Tensor image = torch::from_blob(image_data.data(), {3, 32, 32}, torch::kUInt8).to(torch::kFloat32).div(255.0);
```

```
            if (m_normalize) {
                for (int c = 0; c < 3; ++c) {
                    image[c] = (image[c] - m_mean[c]) / m_std[c];
                }
            }

            m_images.push_back(image);
            m_labels.push_back(static_cast<int64_t>(static_cast<unsigned char>(label)));
        }

        if (!ifs.eof()) {
            throw runtime_error("File read error: " + file);
        }
    }

    cout << "parseBinaryFiles in CIFAR10Dataset completed." << endl;
}

CIFAR100Dataset::CIFAR100Dataset(const string& root, Mode mode, const Options& options)
    : CIFARDataset(root, mode, options, 100) {
    CIFAR100Dataset::parseBinaryFiles(root, mode);
    m_mean[0] = 0.5071f; m_mean[1] = 0.4867f; m_mean[2] = 0.4408f;
    m_std[0] = 0.2675f;    m_std[1] = 0.2565f;    m_std[2] = 0.2761f;
}

void CIFAR100Dataset::parseBinaryFiles(const string& root, Mode mode) {
    cout << "parseBinaryFiles in CIFAR100Dataset Class." << endl;

    const int IMAGE_SIZE = 32 * 32 * 3;
    string filename = (mode == Mode::kTrain) ? "train" : "test";
    filename = root + "/" + filename + ".bin";

    ifstream ifs(filename, ios::binary);
    if (!ifs.is_open()) {
        throw runtime_error("Failed to open file: " + filename + ".bin");
    }

    struct CIFAR100Record {
        char coarse_label;
        char fine_label;
        char image_data[IMAGE_SIZE];
    };
```

```
    CIFAR100Record record;
    while (ifs.read(reinterpret_cast<char*>(&record), sizeof(CIFAR100Record))) {
        torch::Tensor image = torch::from_blob(record.image_data, {3, 32, 32}, torch::kUInt8)
        .to(torch::kFloat32)
            .div(255.0);

        if (m_normalize) {
            for (int c = 0; c < 3; ++c) {
                image[c] = (image[c] - m_mean[c]) / m_std[c];
            }
        }

        m_images.push_back(image);
        m_labels.push_back(static_cast<int64_t>(static_cast<unsigned
char>(record.fine_label)));
    }
    cout<<"TEST"<<endl;
    if (!ifs.eof()) {
        throw runtime_error("File read error: " + filename + ".bin");
    }
}
```

cifardataset.h

```
#ifndef CIFARDATASET_H
#define CIFARDATASET_H

#undef slots
#include <torch/torch.h>
#define slots Q_SLOTS
#include <opencv2/opencv.hpp>
#include <vector>

using namespace std;

namespace cv { class Mat; }

class CIFARDataset : public torch::data::Dataset<CIFARDataset> {
public:
    enum class Mode { kTrain, kTest };

    struct Options {
        torch::Device device = torch::kCPU;
        bool normalize = true;
        bool shuffle = false;
```

```
    };

    CIFARDataset(const string& root, Mode mode, const Options& options, int num_classes);

    torch::data::Example<> get(size_t index) override;

    torch::optional<size_t> size() const override { return m_images.size(); }

    static torch::Tensor fromCVImage(const cv::Mat& image, const Options& options);

protected:
    vector<torch::Tensor> m_images;
    vector<int64_t> m_labels;
    torch::Device m_device;
    bool m_normalize;
    float m_mean[3] = {0.4914f, 0.4822f, 0.4465f};
    float m_std[3] = {0.2023f, 0.1994f, 0.2010f};

private:
    virtual void parseBinaryFiles(const string& root, Mode mode) { cout << "parseBinaryFiles in
Base Class." << endl; };
};

class CIFAR10Dataset : public CIFARDataset {
public:
    CIFAR10Dataset(const string& root, Mode mode = Mode::kTrain, const Options& options =
Options());

private:
    void parseBinaryFiles(const string& root, Mode mode) override;
};

class CIFAR100Dataset : public CIFARDataset {
public:
    CIFAR100Dataset(const string& root, Mode mode = Mode::kTrain, const Options& options =
Options());

private:
    void parseBinaryFiles(const string& root, Mode mode) override;
};

#endif // CIFARDATASET_H
```

trainingthread_cifar.cpp

```
#include "trainingthread_cifar.h"
#include <QThread>

TrainingThreadCIFAR10::TrainingThreadCIFAR10(CNN_CIFAR10* model, QObject* parent) :
QThread(parent), m_model(model), m_device(torch::kCUDA) {
    if (!m_model) {
        emit errorOccurred("模型指针为空！");
        return;
    }

    connect(m_model, &CNN_CIFAR10::epochCompleted, [this](int total, int epoch, double loss,
double accuracy) {
        emit epochCompleted(total, epoch, loss, accuracy);
    });
}

void TrainingThreadCIFAR10::setParameters(const QString& modelName, const QString& device,
int epochs, int batches) {
    m_modelName = modelName;
    m_deviceStr = device;
    m_epochs = epochs;
    m_batches = batches;
    m_device = (device == "NVIDIA GEFORCE RTX 4090 LapTop" &&
torch::cuda::is_available()) ? torch::kCUDA : torch::kCPU;
}

void TrainingThreadCIFAR10::run() {
    if (m_epochs <= 0) {
        emit errorOccurred("训练轮数必须大于 0！");
        return;
    }

    try {
        emit trainingStarted();

        if (m_device == torch::kCUDA && !torch::cuda::is_available()) {
            emit errorOccurred("请求使用 CUDA 但设备不可用，自动切换至 CPU！");
            m_device = torch::kCPU;
            m_deviceStr = "CPU";
        }

        m_model->train(m_device, m_epochs, m_batches);
```

```
            emit trainingFinished();
        } catch (const c10::Error& e) {
            emit errorOccurred(QString("LibTorch 错误: %1").arg(e.what()));
        } catch (const exception& e) {
            emit errorOccurred(QString("训练错误: %1").arg(e.what()));
        } catch (...) {
            emit errorOccurred("未知错误");
        }
}

TrainingThreadCIFAR100::TrainingThreadCIFAR100(CNN_CIFAR100* model, QObject* parent) : QThread(parent), m_model(model), m_device(torch::kCUDA) {
    if (!m_model) {
        emit errorOccurred("模型指针为空！");
        return;
    }

    connect(m_model, &CNN_CIFAR100::epochCompleted, [this](int total, int epoch, double loss, double accuracy) {
        emit epochCompleted(total, epoch, loss, accuracy);
    });
}

void TrainingThreadCIFAR100::setParameters(const QString& modelName, const QString& device, int epochs, int batches) {
    m_modelName = modelName;
    m_deviceStr = device;
    m_epochs = epochs;
    m_batches = batches;
    m_device = (device == "NVIDIA GEFORCE RTX 4090 LapTop" && torch::cuda::is_available()) ? torch::kCUDA : torch::kCPU;
}

void TrainingThreadCIFAR100::run() {
    if (m_epochs <= 0) {
        emit errorOccurred("训练轮数必须大于 0！");
        return;
    }

    try {
        emit trainingStarted();

        if (m_device == torch::kCUDA && !torch::cuda::is_available()) {
            emit errorOccurred("请求使用 CUDA 但设备不可用，自动切换至 CPU！");
```

```
            m_device = torch::kCPU;
            m_deviceStr = "CPU";
        }

        m_model->train(m_device, m_epochs, m_batches);

        emit trainingFinished();
    } catch (const c10::Error& e) {
        emit errorOccurred(QString("LibTorch 错误: %1").arg(e.what()));
    } catch (const exception& e) {
        emit errorOccurred(QString("训练错误: %1").arg(e.what()));
    } catch (...) {
        emit errorOccurred("未知错误");
    }
}
```

trainingthread_cifar.h

```
#ifndef TRAININGTHREAD_CIFAR_H
#define TRAININGTHREAD_CIFAR_H

#include <QThread>
#include <QString>
#undef slots
#include <torch/torch.h>
#define slots Q_SLOTS
#include "model_cifar.h"

class TrainingThreadCIFAR10 : public QThread {
    Q_OBJECT

public:
    explicit TrainingThreadCIFAR10(CNN_CIFAR10* model, QObject* parent);
    void setParameters(const QString& modelName, const QString& device, int epochs, int batches);

signals:
    void trainingStarted();
    void progressUpdated(int progress);
    void epochCompleted(int total, int epoch, double loss, double accuracy);
    void trainingFinished();
    void errorOccurred(const QString& message);

protected:
    void run() override;
```

```
private:
    CNN_CIFAR10* m_model;
    QString m_modelName;
    QString m_deviceStr;
    int m_epochs;
    int m_batches;
    torch::Device m_device;
};

class TrainingThreadCIFAR100 : public QThread {
    Q_OBJECT

public:
    explicit TrainingThreadCIFAR100(CNN_CIFAR100* model, QObject* parent);
    void setParameters(const QString& modelName, const QString& device, int epochs, int batches);

signals:
    void trainingStarted();
    void progressUpdated(int progress);
    void epochCompleted(int total, int epoch, double loss, double accuracy);
    void trainingFinished();
    void errorOccurred(const QString& message);

protected:
    void run() override;

private:
    CNN_CIFAR100* m_model;
    QString m_modelName;
    QString m_deviceStr;
    int m_epochs;
    int m_batches;
    torch::Device m_device;
};

#endif // TRAININGTHREAD_CIFAR_H
```

testthread_cifar.cpp

```
#include "testthread_cifar.h"

TestThreadCIFAR10::TestThreadCIFAR10(CNN_CIFAR10 *model, torch::Device device, int batches, QObject *parent)
```

```
    : QThread(parent), m_model(model), m_device(device), m_batches(batches) {
    m_device = (device == torch::kCUDA && torch::cuda::is_available()) ? torch::kCUDA :
torch::kCPU;
}

void TestThreadCIFAR10::setParameters(CNN_CIFAR10 *model, QString device, int batches) {
    m_model = model;
    m_batches = batches;
    m_device = (device == "NVIDIA GEFORCE RTX 4090 LapTop" &&
torch::cuda::is_available()) ? torch::kCUDA : torch::kCPU;
}

void TestThreadCIFAR10::run() {
    emit testStarted();

    try {
        if (m_device == torch::kCUDA && !torch::cuda::is_available()) {
            emit errorOccurred("请求使用 CUDA 但设备不可用，自动切换至 CPU！");
            m_device = torch::kCPU;
        }

        double acc = m_model->test(m_device, m_batches);

        emit testFinished(acc);

    } catch (const c10::Error& e) {
        emit errorOccurred(QString("LibTorch 错误: %1").arg(e.what()));
    } catch (const exception& e) {
        emit errorOccurred(QString("测试错误: %1").arg(e.what()));
    } catch (...) {
        emit errorOccurred("未知错误");
    }
}

TestThreadCIFAR100::TestThreadCIFAR100(CNN_CIFAR100 *model, torch::Device device, int
batches, QObject *parent)
    : QThread(parent), m_model(model), m_device(device), m_batches(batches) {
    m_device = (device == torch::kCUDA && torch::cuda::is_available()) ? torch::kCUDA :
torch::kCPU;
}

void TestThreadCIFAR100::setParameters(CNN_CIFAR100 *model, QString device, int batches)
{
    m_model = model;
```

```
    m_batches = batches;
    m_device = (device == "NVIDIA GEFORCE RTX 4090 LapTop" && torch::cuda::is_available()) ? torch::kCUDA : torch::kCPU;
}

void TestThreadCIFAR100::run() {
    emit testStarted();

    try {
        if (m_device == torch::kCUDA && !torch::cuda::is_available()) {
            emit errorOccurred("请求使用 CUDA 但设备不可用，自动切换至 CPU！");
            m_device = torch::kCPU;
        }
        double acc = m_model->test(m_device, m_batches);
        emit testFinished(acc);
    } catch (const c10::Error& e) {
        emit errorOccurred(QString("LibTorch 错误: %1").arg(e.what()));
    } catch (const exception& e) {
        emit errorOccurred(QString("测试错误: %1").arg(e.what()));
    } catch (...) {
        emit errorOccurred("未知错误");
    }
}
```

testthread_cifar.h

```
// testthread_cifar.h
#ifndef TESTTHREAD_CIFAR_H
#define TESTTHREAD_CIFAR_H

#undef slots
#include <torch/torch.h>
#define slots Q_SLOTS
#include <QThread>
#include "model_cifar.h"

class TestThreadCIFAR10 : public QThread {
    Q_OBJECT

public:
    explicit TestThreadCIFAR10(CNN_CIFAR10 *model, torch::Device device, int batches, QObject *parent = nullptr);
    void setParameters(CNN_CIFAR10 *model, QString device, int batches);

signals:
```

```
    void testStarted();
    void testFinished(double acc);
    void errorOccurred(const QString &msg);

protected:
    void run() override;

private:
    CNN_CIFAR10 *m_model;
    torch::Device m_device;
    int m_batches;
};

class TestThreadCIFAR100 : public QThread {
    Q_OBJECT

public:
    explicit TestThreadCIFAR100(CNN_CIFAR100 *model, torch::Device device, int batches,
QObject *parent = nullptr);
    void setParameters(CNN_CIFAR100 *model, QString device, int batches);

signals:
    void testStarted();
    void testFinished(double acc);
    void errorOccurred(const QString &msg);

protected:
    void run() override;

private:
    CNN_CIFAR100 *m_model;
    torch::Device m_device;
    int m_batches;
};

#endif // TESTTHREAD_CIFAR_H
```