#include "fusionstudio.h"

#include <QtWidgets/QApplication>

int main(int argc, char \*argv[])

{

QApplication a(argc, argv);

FusionStudio w;

w.show();

return a.exec();

}

#pragma once

#include <QtWidgets/QMainWindow>

#include "QProcess"

#include "ui\_fusionstudio.h"

#include "Project.h"

#include "trainingsetter.h"

#include "convertmnist.h"

#include "convertcifar.h"

#include "convertimage.h"

#include "browsedatabase.h"

#include "preferences.h"

#include "myhighlighter.h"

#include "typedef.h"

#include "editor.h"

#include "homepage.h"

#include "neteditor.h"

#include "classification.h"

#include "featureexp.h"

#include "about.h"

class FusionStudio : public QMainWindow

{

Q\_OBJECT

public:

FusionStudio(QWidget \*parent = Q\_NULLPTR);

QProcess \* p2;

static void test();

private:

Ui::FusionStudioClass ui; ///内部变量

void configureUI();

QPoint PrePosition;

//QMenu \*TreeViewMenu = NULL;

QMenu \*TreeViewMenu\_Empty = NULL;//文件树为空时的右键单击事件菜单

QMenu \*TreeViewMenu\_File = NULL;//发生在文件节点上的右键单击事件菜单

QMenu \*TreeViewMenu\_Path = NULL;//发生在文件夹节点上的右键单击事件菜单

Project \* m\_project;

Preferences \*m\_PreferencesManager;//首选项管理器,窗口关闭不能销毁

QString CurrentTreeViewSelectedPath;//当前文件树视图中被选中的项目的绝对路径

private: ///内部操作

bool DelDir(const QString &path);

public slots:

void start\_read\_output();//控制台标准输出重定向

void Train();//测试控制台重定向功能

void Refresh();//刷新文件树

void ShowEditor();

void on\_treeView\_customContextMenuRequested(QPoint pos);

//文件树->右键菜单

void newFile();//新建文件

void newFolder();//新建文件夹

void getAbsolutePath();//获取绝对路径

void openInSystemResourceManager();//在系统资源管理器中打开

void deleteFileOrPath();//删除路径或文件

//菜单->文件

void FolderCreater();

void FolderUnfolder();

void FolderCloser();

//菜单->视图

void changeVisible\_Explorer();

void changeVisible\_Console();

//菜单->工具

void MnistConverter();

void CifarConverter();

void ImageConverter();

void DatabaseBrowser();

//菜单->设置

void PreferencesSetter();//启动首选项管理器

//菜单->项目

void WorkingDirScanner();//扫描工作目录

//菜单->生成

void DoClassification();//分类

void DoFeaturesExp();//特征导出

//菜单->帮助

void ShowHome();

void ShowAbout();

};

#include "fusionstudio.h"

#include "QDebug"

#include <QFileDialog>

#include <QSplitter>

#include <QDirModel>

#include <QMessageBox>

#include <QInputDialog>

#include <QClipboard>

#include <QFileInfo>

//构造和初始化

FusionStudio::FusionStudio(QWidget \*parent)

: QMainWindow(parent)

{

ui.setupUi(this);

this->configureUI();

//显示起始页面

HomePage \*\_homePage = new HomePage();

ui.mdiArea->addSubWindow(\_homePage);

\_homePage->showMaximized();

m\_project = new Project();//当前工程初始化

m\_PreferencesManager = new Preferences();//首选项管理器

p2 = new QProcess();//控制台标准输出重定向

p2->setProcessChannelMode(QProcess::MergedChannels);

connect(p2, SIGNAL(readyRead()), this, SLOT(start\_read\_output()));

connect(m\_project, SIGNAL(ValueChanged()), this, SLOT(Refresh()));

}

//自定义UI设置

void FusionStudio::configureUI()

{

QAction\* action\_New\_Working\_Dir = new QAction(QStringLiteral("&新建工作目录"), ui.treeView);

connect(action\_New\_Working\_Dir, SIGNAL(triggered()), this, SLOT(FolderCreater()));

QAction\* action\_Open\_Working\_Dir = new QAction(QStringLiteral("&打开工作目录"), ui.treeView);

connect(action\_Open\_Working\_Dir, SIGNAL(triggered()), this, SLOT(FolderUnfolder()));

QAction\* action\_New\_File = new QAction(QStringLiteral("&新建文件"), this);

connect(action\_New\_File, SIGNAL(triggered()), this, SLOT(newFile()));

QAction\* action\_New\_Folder = new QAction(QStringLiteral("&新建文件夹"), this);

connect(action\_New\_Folder, SIGNAL(triggered()), this, SLOT(newFolder()));

QAction\* action\_Open\_In\_Explorer = new QAction(QStringLiteral("&在资源管理器中打开"), this);

connect(action\_Open\_In\_Explorer, SIGNAL(triggered()), this, SLOT(openInSystemResourceManager()));

QAction\* action\_Open\_Father\_Dir = new QAction(QStringLiteral("&打开所在目录"), ui.treeView);

connect(action\_Open\_Father\_Dir, SIGNAL(triggered()), this, SLOT(openInSystemResourceManager()));

QAction\* action\_Get\_Abs\_Path = new QAction(QStringLiteral("&复制绝对路径"), this);

connect(action\_Get\_Abs\_Path, SIGNAL(triggered()), this, SLOT(getAbsolutePath()));

QAction\* action\_Delete\_File = new QAction(QStringLiteral("&删除文件"), this);

connect(action\_Delete\_File, SIGNAL(triggered()), this, SLOT(deleteFileOrPath()));

QAction\* action\_Delete\_Path = new QAction(QStringLiteral("&删除文件夹"), this);

connect(action\_Delete\_Path, SIGNAL(triggered()), this, SLOT(deleteFileOrPath()));

//===============右键菜单的定义======================

//文件树为空时发生的右键事件菜单

TreeViewMenu\_Empty = new QMenu(ui.treeView);

TreeViewMenu\_Empty->addAction(action\_New\_Working\_Dir);

TreeViewMenu\_Empty->addAction(action\_Open\_Working\_Dir);

//发生在文件节点上的右键事件菜单

TreeViewMenu\_File = new QMenu(ui.treeView);

TreeViewMenu\_File->addAction(action\_Open\_Father\_Dir);

TreeViewMenu\_File->addAction(action\_Get\_Abs\_Path);

TreeViewMenu\_File->addSeparator();

TreeViewMenu\_File->addAction(action\_Delete\_File);

//发生在文件夹节点生的右键事件菜单

TreeViewMenu\_Path = new QMenu(ui.treeView);

//TreeViewMenu\_Path->addAction(action\_New\_File);

TreeViewMenu\_Path->addAction(action\_New\_Folder);

TreeViewMenu\_Path->addSeparator();

TreeViewMenu\_Path->addAction(action\_Open\_In\_Explorer);

TreeViewMenu\_Path->addAction(action\_Get\_Abs\_Path);

TreeViewMenu\_Path->addSeparator();

TreeViewMenu\_Path->addAction(action\_Delete\_Path);

//================其他ui设置===========================

ui.action\_view\_Console->setChecked(true);

ui.action\_view\_Explorer->setChecked(true);

}

//=====================槽函数==========================

//文件树的右键菜单响应

void FusionStudio::on\_treeView\_customContextMenuRequested(QPoint pos)

{

//qDebug() << "右键触发"<<pos.x() << "," << pos.y();

//通过测试发现,连续发生的两次调用的触发点是一样的

//暂时通过检测这次触发是否和上一次发生在同一个位置来规避该问题

if (PrePosition != pos) {//这次触发是正常触发

PrePosition = pos;

if (ui.treeView->model() == NULL) {

//此时文件树是空的

TreeViewMenu\_Empty->exec(QCursor::pos());//显示右键菜单

}

else//文件树非空的时候才能启动这个

{

QModelIndex indexSelect = ui.treeView->indexAt(pos);//当前节点索引

QString AbsPath = indexSelect.data().toString(); //当前节点数据

if (AbsPath != "")//如果当前右击发生在树形节点上

{

//迭代出当前节点的绝对路径

QString temp;

QModelIndex indexParent;

do {

indexParent = indexSelect.parent();//当前节点的父节点

temp = indexParent.data().toString();

if (temp == "")

{

break;

}

AbsPath = temp.append("\\" + AbsPath);

indexSelect = indexParent;

} while (true);

CurrentTreeViewSelectedPath = AbsPath;//把这个路径通过全局传出去

//分辨一下点到了什么

QFileInfo \_fio(AbsPath);

if (\_fio.isFile()==true)//文件

{

TreeViewMenu\_File->exec(QCursor::pos());

}

if (\_fio.isDir()==true)//文件夹

{

TreeViewMenu\_Path->exec(QCursor::pos());

}

}

}

}

else

{

//这次触发是非正常触发

PrePosition.setX(-1);

PrePosition.setY(-1);

}

}

//控制台标准输出重定向

void FusionStudio::start\_read\_output()

{

QString result = QString::fromLocal8Bit(p2->readAll());

ui.plainTextEdit->appendPlainText(result);

//ui.plainTextEdit->appendPlainText("Ok,i am working...");

}

//刷新文件树

void FusionStudio::Refresh()

{

//QMessageBox::warning(this, QStringLiteral("安娜学姐的问候"), QStringLiteral("收到了值更改消息"));

if (m\_project->getWorkingDir().isEmpty()) {

//目录是空的,可能是误操作,也可能是执行了关闭工作目录,统一处理为清空文件树

ui.treeView->setModel(NULL);

}

else

{

QDirModel \*model = new QDirModel();

ui.treeView->setModel(model);

ui.treeView->setColumnHidden(1, true);

ui.treeView->setColumnHidden(2, true);

ui.treeView->setColumnHidden(3, true);

ui.treeView->setRootIndex(model->index(m\_project->getWorkingDir()));

ui.treeView->expandAll();//全部展开

}

}

//==================文件树右键菜单=====================

//文件树右键菜单->新建文件夹

void FusionStudio::newFolder()

{

//弹出对话框询问新文件夹的名字

bool isOK;

QString FolderName = QInputDialog::getText(NULL, QStringLiteral("新建文件夹"),

QStringLiteral("请为即将新建的文件夹命名:"),

QLineEdit::Normal, QStringLiteral("新建文件夹"), &isOK);

if (isOK)//点击确定

{

QString \_temppath = CurrentTreeViewSelectedPath;

\_temppath.append("/");

\_temppath.append(FolderName);

QDir \*\_tempdir = new QDir;

if (\_tempdir->exists(\_temppath))

{

QMessageBox::warning(this, QStringLiteral("新建文件夹"), QStringLiteral("文件夹已经存在！"));

}

else

{

if (\_tempdir->mkdir(\_temppath))

{

QMessageBox::warning(this, QStringLiteral("创建文件夹"), QStringLiteral("文件夹创建成功！"));

Refresh();

}

else

{

QMessageBox::warning(this, QStringLiteral("新建文件夹"), QStringLiteral("未知错误导致的创建失败!"));

}

}

}

}

//文件树右键菜单->获取绝对路径到剪贴板

void FusionStudio::getAbsolutePath()

{

//QMessageBox::warning(this, QStringLiteral("节点数据"), CurrentTreeViewSelectedPath);

//使用 QApplication::clipboard() 函数获得系统剪贴板对象。这个函数的返回值是 QClipboard 指针。

QClipboard \*board = QApplication::clipboard();

//通过 setText()，setImage() 或者 setPixmap() 函数可以将数据放置到剪贴板内，也就是通常所说的剪贴或者复制的操作；

board->setText(CurrentTreeViewSelectedPath);

//使用text()，image() 或者 pixmap() 函数则可以从剪贴板获得数据，也就是粘贴。

//QString str = board->text();

}

//文件树右键菜单->在在系统文件资源管理器中打开路径

void FusionStudio::openInSystemResourceManager()

{

QFileInfo \_fio(CurrentTreeViewSelectedPath);

if (\_fio.isFile() == true)//文件

{//文件打开父目录

QString cmd = "explorer.exe ";

cmd.append(\_fio.canonicalPath());

cmd = cmd.replace(QRegExp("\\/"), "\\");//正则替换将"撇"换称"捺"

QProcess::execute(cmd);

}

if (\_fio.isDir() == true)//文件夹

{//文件夹直接打开

QString cmd = "explorer ";

cmd.append(CurrentTreeViewSelectedPath);

QProcess::execute(cmd);

}

}

//递归删除文件夹及其中全部文件

bool FusionStudio::DelDir(const QString & path)

{

if (path.isEmpty()) {

return false;

}

QDir dir(path);

if (!dir.exists()) {

return true;

}

dir.setFilter(QDir::AllEntries | QDir::NoDotAndDotDot); //设置过滤

QFileInfoList fileList = dir.entryInfoList(); // 获取所有的文件信息

foreach(QFileInfo file, fileList) { //遍历文件信息

if (file.isFile()) { // 是文件，删除

file.dir().remove(file.fileName());

}

else { // 递归删除

DelDir(file.absoluteFilePath());

}

}

return dir.rmpath(dir.absolutePath()); // 删除文件夹

}

//文件树右键菜单->删除路径或文件

void FusionStudio::deleteFileOrPath()

{

//判断是路径还是文件,以弹出不同的删除确认框

QFileInfo \_fio(CurrentTreeViewSelectedPath);

if (\_fio.isFile() == true)//文件

{//文件直接删除

QMessageBox::StandardButton rb = QMessageBox::question(NULL, QStringLiteral("确认删除?"), QStringLiteral("所选文件将会从磁盘上消失!"), QMessageBox::Yes | QMessageBox::No, QMessageBox::Yes);

if (rb == QMessageBox::Yes)

{

\_fio.dir().remove(\_fio.fileName());

Refresh();

}

}

if (\_fio.isDir() == true)//文件夹

{//文件夹递归删除全部内容

QMessageBox::StandardButton rb = QMessageBox::question(NULL, QStringLiteral("确认删除?"), QStringLiteral("删除文件夹将会删除其中所有的文件!"), QMessageBox::Yes | QMessageBox::No, QMessageBox::Yes);

if (rb == QMessageBox::Yes)

{//已经确认删除

DelDir(CurrentTreeViewSelectedPath);

Refresh();

}

}

}

//================菜单->文件===========================

//文件->创建目录

void FusionStudio::FolderCreater()

{

//1.选择父目录

QString ParentDir = QFileDialog::getExistingDirectory(this,

QStringLiteral("选择新文件夹的父目录"),

QDir::currentPath());

if (ParentDir.isEmpty() == false)//父目录非空

{

//2.询问目录名

bool isOK;

QString FolderName = QInputDialog::getText(NULL, QStringLiteral("新建工作目录"),

QStringLiteral("请为即将新建的工作目录命名:"),

QLineEdit::Normal, "NewFolder", &isOK);

if (isOK) {//点击了确认

if (FolderName.isEmpty() == false)//目录名非空

{

//3.创建

QString CompleteDir;

CompleteDir.append(ParentDir);

CompleteDir.append("\\");

CompleteDir.append(FolderName);

QDir \*DirManager = new QDir;

if (DirManager->exists(CompleteDir) == true)//已经存在同名目录

{

QMessageBox::warning(this, QStringLiteral("创建失败"), QStringLiteral("文件夹已经存在！"));

}

else

{

if (DirManager->mkdir(CompleteDir) == true)//创建成功

{

QMessageBox::warning(this, QStringLiteral("创建文件夹"), QStringLiteral("文件夹创建成功！"));

//4.设置工作目录

//setWorkingDir(CompleteDir);

m\_project->setWorkingDir(CompleteDir);

}

else

{

QMessageBox::warning(this, QStringLiteral("创建失败"), QStringLiteral("未知错误!"));

}

}

}

}

}

else

{

//父目录为空

QMessageBox::warning(this, QStringLiteral("创建失败"), QStringLiteral("父目录为空！"));

}

}

//文件->开启目录

void FusionStudio::FolderUnfolder()

{

/\*打开一个已经存在的目录,作为当前的工作目录\*/

//1.打开文件管理窗口

QString ExistedDir = QFileDialog::getExistingDirectory(this,

QStringLiteral("选择目录作为工作目录"),

QDir::currentPath());

if (ExistedDir.isEmpty() == false)//工作目录非空

{

//将这个目录设置为当前工作目录,触发值更改事件

//setWorkingDir(ExistedDir);

m\_project->setWorkingDir(ExistedDir);

}

}

//文件->关闭目录

void FusionStudio::FolderCloser()

{

//setWorkingDir(NULL);//工作路径设为空

m\_project->setWorkingDir(NULL);

}

//================菜单->视图===========================

//变更资源浏览器视图的可见性

void FusionStudio::changeVisible\_Explorer()

{

if (ui.dockWidget\_Explorer->isHidden() == true) {

ui.dockWidget\_Explorer->show();//隐藏实用工具

ui.action\_view\_Explorer->setChecked(true);

}

else

{

ui.dockWidget\_Explorer->hide();//隐藏实用工具

ui.action\_view\_Explorer->setChecked(false);

}

}

//变更控制台视图的可见性

void FusionStudio::changeVisible\_Console()

{

if (ui.dockWidget\_Console->isHidden() == true) {

ui.dockWidget\_Console->show();//隐藏实用工具

ui.action\_view\_Console->setChecked(true);

}

else

{

ui.dockWidget\_Console->hide();//隐藏实用工具

ui.action\_view\_Console->setChecked(false);

}

}

//================菜单->项目============================

//项目->扫描工作目录

void FusionStudio::WorkingDirScanner()

{

Refresh();

}

//================菜单->生成============================

//新建编辑器

void FusionStudio::ShowEditor()

{

NetEditor \*child = new NetEditor();

ui.mdiArea->addSubWindow(child);

child->show();

//暂时用一下

}

//训练

void FusionStudio::Train()

{

TrainingSetter \*w = new TrainingSetter(m\_PreferencesManager->getBinPath(), p2);

ui.mdiArea->addSubWindow(w);

w->show();

}

//分类

void FusionStudio::DoClassification()

{

Classification\*w = new Classification(m\_PreferencesManager->getBinPath(),p2);

ui.mdiArea->addSubWindow(w);

w->show();

}

//生成->特征导出

void FusionStudio::DoFeaturesExp()

{

FeatureExp\*w = new FeatureExp(m\_PreferencesManager->getBinPath(), p2);

ui.mdiArea->addSubWindow(w);

w->show();

}

//================菜单->工具============================

//工具->启动Mnist数据转换工具

void FusionStudio::MnistConverter()

{

ConvertMnist \*w = new ConvertMnist();

ui.mdiArea->addSubWindow(w);

w->show();

}

//工具->启动Cifar转换工具

void FusionStudio::CifarConverter()

{

ConvertCifar \*w = new ConvertCifar();

ui.mdiArea->addSubWindow(w);

w->show();

}

//工具->启动图片转换工具

void FusionStudio::ImageConverter()

{

ConvertImage \*w = new ConvertImage();

ui.mdiArea->addSubWindow(w);

w->show();

}

//工具->启动数据集浏览工具

void FusionStudio::DatabaseBrowser()

{

BrowseDatabase \*w = new BrowseDatabase();

ui.mdiArea->addSubWindow(w);

w->show();

}

//================菜单->设置============================

//设置->启动首选项管理器

void FusionStudio::PreferencesSetter()

{

m\_PreferencesManager->awakenPreferencesManager();//唤醒

//ui.mdiArea->addSubWindow(m\_PreferencesManager);//作为模态对话框使用

m\_PreferencesManager->show();

}

//================菜单->帮助============================

//帮助->显示开始页面

void FusionStudio::ShowHome()

{

HomePage \*\_homePage = new HomePage();

ui.mdiArea->addSubWindow(\_homePage);

\_homePage->showMaximized();

}

//帮助->显示关于页面

void FusionStudio::ShowAbout()

{

About \*\_aboutPage = new About();

\_aboutPage->show();

}

#pragma once

#include <QWidget>

#include "ui\_about.h"

class About : public QWidget

{

Q\_OBJECT

public:

About(QWidget \*parent = Q\_NULLPTR);

~About();

private:

Ui::About ui;

};

#include "about.h"

About::About(QWidget \*parent)

: QWidget(parent)

{

ui.setupUi(this);

}

About::~About()

{

}

#pragma once

#ifndef ARROW\_H

#define ARROW\_H

#include <QGraphicsLineItem>

#include "Diagramitem.h"

QT\_BEGIN\_NAMESPACE

class QGraphicsPolygonItem;

class QGraphicsLineItem;

class QGraphicsScene;

class QRectF;

class QGraphicsSceneMouseEvent;

class QPainterPath;

QT\_END\_NAMESPACE

//! [0]

class Arrow : public QGraphicsLineItem

{

public:

enum { Type = UserType + 4 };

Arrow(DiagramItem \*startItem, DiagramItem \*endItem,

QGraphicsItem \*parent = 0);

int type() const override { return Type; }

QRectF boundingRect() const override;

QPainterPath shape() const override;

void setColor(const QColor &color) { myColor = color; }

DiagramItem \*startItem() const { return myStartItem; }

DiagramItem \*endItem() const { return myEndItem; }

void updatePosition();

protected:

void paint(QPainter \*painter, const QStyleOptionGraphicsItem \*option, QWidget \*widget = 0) override;

private:

DiagramItem \*myStartItem;

DiagramItem \*myEndItem;

QColor myColor;

QPolygonF arrowHead;

};

//! [0]

#endif // ARROW\_H

#include "Arrow.h"

#include <math.h>

#include <QPen>

#include <QPainter>

const qreal Pi = 3.14;

//! [0]

Arrow::Arrow(DiagramItem \*startItem, DiagramItem \*endItem, QGraphicsItem \*parent)

: QGraphicsLineItem(parent)

{

myStartItem = startItem;

myEndItem = endItem;

setFlag(QGraphicsItem::ItemIsSelectable, true);

myColor = Qt::black;

setPen(QPen(myColor, 2, Qt::SolidLine, Qt::RoundCap, Qt::RoundJoin));

}

QRectF Arrow::boundingRect() const

{

qreal extra = (pen().width() + 20) / 2.0;

return QRectF(line().p1(), QSizeF(line().p2().x() - line().p1().x(),

line().p2().y() - line().p1().y()))

.normalized()

.adjusted(-extra, -extra, extra, extra);

}

QPainterPath Arrow::shape() const

{

QPainterPath path = QGraphicsLineItem::shape();

path.addPolygon(arrowHead);

return path;

}

void Arrow::updatePosition()

{

QLineF line(mapFromItem(myStartItem, 0, 0), mapFromItem(myEndItem, 0, 0));

setLine(line);

}

void Arrow::paint(QPainter \*painter, const QStyleOptionGraphicsItem \*,

QWidget \*)

{

if (myStartItem->collidesWithItem(myEndItem)) ////如果起始图元与结束图元有碰撞,则不画线

return;

QPen pen;

//pen.setColor(Qt::black);

//pen.setWidth(2);

//pen.setStyle(Qt::DashDotLine);

setPen(QPen(Qt::black, 2, Qt::DashDotLine, Qt::RoundCap, Qt::RoundJoin));

painter->setPen(pen);

painter->setRenderHint(QPainter::Antialiasing, true);//抗锯齿开启

//计算曲线的起止点

QPointF start\_pos(myStartItem->pos().x() + 170, myStartItem->pos().y() + 15);

QPointF end\_pos(myEndItem->pos().x() + 10, myEndItem->pos().y() + 15);

//开始绘制曲线

QPainterPath path(start\_pos);//曲线的起点

float theta = 2000; //灵敏度参数

float m = theta / ((abs(end\_pos.y() - start\_pos.y()) + 1)\*((abs(end\_pos.x() - start\_pos.x()) + 1))); //变距因数

float m1 = (m < 0 ? 0 : m);

float xxxx = end\_pos.y() - start\_pos.y();

m1 = (end\_pos.y() - start\_pos.y() == 0 ? 0 : m);

if (end\_pos.x() - start\_pos.x() >= 50) {//滤掉无穷间断点

m1 = 0;

}

//通过移动控制点实现曲线变距

float xk1 = (start\_pos.x() + end\_pos.x()) / 2 + m1;

float xk2 = (start\_pos.x() + end\_pos.x()) / 2 - m1;

QPoint c1(xk1, start\_pos.y() + m1 / 1000);

QPoint c2(xk2, end\_pos.y() - m1 / 1000);

path.cubicTo(c1, c2, end\_pos);//两个控制点一个结束点

//这个函数会导致残影

painter->drawPath(path);

}

//! [7]

#pragma once

#include <QWidget>

#include "ui\_browsedatabase.h"

#include <opencv2/core/core.hpp>

#include <opencv2/highgui.hpp>

#include "DbOperator.h"

class BrowseDatabase : public QWidget

{

Q\_OBJECT

public:

BrowseDatabase(QWidget \*parent = Q\_NULLPTR);

~BrowseDatabase();

private:

Ui::BrowseDatabase ui;

DbOperator \*MyDbHandle;//数据库操作句柄

private:

QImage MatToQImage(const cv::Mat&);

public slots:

void LoadDatabase();//

void Nextimage();

void ZoomImage();

void CrashWarning();

};

#include "browsedatabase.h"

#include <QFileDialog>

#include <QtCore/qmath.h>

#include <qDebug>

#include <QMessageBox>

BrowseDatabase::BrowseDatabase(QWidget \*parent)

: QWidget(parent)

{

ui.setupUi(this);

ui.radioButton\_lmdb->setChecked(true);

}

BrowseDatabase::~BrowseDatabase()

{

}

//Mat转换为QImage

QImage BrowseDatabase::MatToQImage(const cv::Mat &mat)

{

// 8-bits unsigned, NO. OF CHANNELS=1

if (mat.type() == CV\_8UC1)

{

// Set the color table (used to translate colour indexes to qRgb values)

QVector<QRgb> colorTable;

for (int i = 0; i<256; i++)

colorTable.push\_back(qRgb(i, i, i));

// Copy input Mat

const uchar \*qImageBuffer = (const uchar\*)mat.data;

// Create QImage with same dimensions as input Mat

QImage img(qImageBuffer, mat.cols, mat.rows, mat.step, QImage::Format\_Indexed8);

img.setColorTable(colorTable);

return img;

}

// 8-bits unsigned, NO. OF CHANNELS=3

if (mat.type() == CV\_8UC3)

{

// Copy input Mat

const uchar \*qImageBuffer = (const uchar\*)mat.data;

// Create QImage with same dimensions as input Mat

QImage img(qImageBuffer, mat.cols, mat.rows, mat.step, QImage::Format\_RGB888);

return img.rgbSwapped();

}

else

{

//qDebug() << "ERROR: Mat could not be converted to QImage.";

return QImage();

}

}

//加载数据库文件

void BrowseDatabase::LoadDatabase() {

//1.打开路径选择窗口

QString tempInstallPath = QFileDialog::getExistingDirectory(this, QStringLiteral("选择数据库文件路径"), QDir::currentPath());

if (tempInstallPath != "")

{

//2.绑定数据库操作句柄

std::string datatype = (ui.radioButton\_lmdb->isChecked() ? "lmdb" : "leveldb");

if (ui.checkBox\_ForcedDecoding->isChecked()==true)//使用强制解码

{

//用goto跳过判定

goto VerifyCompleted;

}

//2.1验证数据库的有效性

if (

QFile::exists(tempInstallPath) &&

(

( datatype == "lmdb"&&

QFile::exists(tempInstallPath + "/data.mdb") &&

QFile::exists(tempInstallPath + "/lock.mdb")

) || (

datatype == "leveldb"&&

QFile::exists(tempInstallPath + "/CURRENT") &&

QFile::exists(tempInstallPath + "/LOCK") &&

QFile::exists(tempInstallPath + "/LOG")

)

)

)

{

VerifyCompleted:

//通过验证

MyDbHandle = new DbOperator(datatype, tempInstallPath.toStdString());

//3.显示第一张图片

cv::Mat imageToShow;

int label = 0;

MyDbHandle->getImage(imageToShow, label);

QImage qimageToShow = MatToQImage(imageToShow);//得到了QImage

QGraphicsScene \*scene = new QGraphicsScene;

scene->addPixmap(QPixmap::fromImage(qimageToShow));

ui.graphicsView->setScene(scene);

//ui.graphicsView->resize(qimageToShow.width() + 10, qimageToShow.height() + 10);

ui.graphicsView->show();

ui.lineEdit->setText(QString::number(label, 10));

}

else

{

QMessageBox::information(this, QStringLiteral("数据集浏览"),

QStringLiteral("选中的文件目录与所选的数据库类型不符或者数据库已经损坏!"));

return;

}

}

else

{

QMessageBox::information(this, QStringLiteral("数据集浏览"),

QStringLiteral("请选择一个数据库目录!"));

return;

}

}

//下一张图片

void BrowseDatabase::Nextimage()

{

cv::Mat imageToShow;

int label = 0;

bool flag = MyDbHandle->getImage(imageToShow, label);

if (flag==true)

{

QImage qimageToShow = MatToQImage(imageToShow);//得到了QImage

QGraphicsScene \*scene = new QGraphicsScene;

scene->addPixmap(QPixmap::fromImage(qimageToShow));

ui.graphicsView->setScene(scene);

//ui.graphicsView->resize(qimageToShow.width() + 10, qimageToShow.height() + 10);

ui.graphicsView->show();

ui.lineEdit->setText(QString::number(label, 10));

}

else

{

qDebug()<< QStringLiteral("无图")<<"\n";

}

}

//缩放

void BrowseDatabase::ZoomImage()

{

qreal scale = qPow(qreal(2), (ui.zoomSlider->value()) / qreal(25));

QMatrix matrix;//缩放矩阵

matrix.scale(scale, scale);

ui.graphicsView->setMatrix(matrix);

}

void BrowseDatabase::CrashWarning()

{

if (ui.checkBox\_ForcedDecoding->isChecked() == true) {

QMessageBox::information(this, QStringLiteral("崩溃警告"),

QStringLiteral("该选项将会屏蔽程序的故障诊断功能并进行强制解码\n可能会产生严重错误甚至导致程序崩溃!\n请谨慎使用!"));

}

}

#pragma once

#include <QWidget>

#include "ui\_classification.h"

#include <QProcess>

#include <QString>

class Classification : public QWidget

{

Q\_OBJECT

public:

Classification(QString coredir,QProcess \*addprocess,QWidget \*parent = Q\_NULLPTR);

~Classification();

private:

Ui::Classification ui;

QString CoreDir;//保存核心目录

QProcess \*\_AdditionalProcess;//指向主界面类中已经重定向的那个线程,浅拷贝,构造赋值

public slots:

void Browser1();

void Browser2();

void Browser3();

void Browser4();

void Browser5();

void DoClassification();

void ZoomImage();

};

#include "classification.h"

#include <QFileDialog>

#include <QMessageBox>

#include <QtCore/qmath.h>

Classification::Classification(QString coredir, QProcess \*addprocess, QWidget \*parent)

: QWidget(parent)

{

ui.setupUi(this);

CoreDir= coredir;

\_AdditionalProcess= addprocess;

}

Classification::~Classification()

{

}

void Classification::Browser1()

{

ui.lineEdit\_deploy->setText(

QFileDialog::getOpenFileName(this,

QStringLiteral("选择网络部署描述文件"),

" ",

QStringLiteral("部署描述(\*.prototxt);;所有类型(\*.\*)")));

}

void Classification::Browser2()

{

ui.lineEdit\_caffemodel->setText(

QFileDialog::getOpenFileName(this,

QStringLiteral("选择预训练网络文件"),

" ",

QStringLiteral("caffe预训练网络(\*.caffemodel);;所有类型(\*.\*)")));

}

void Classification::Browser3()

{

ui.lineEdit\_binaryproto->setText(

QFileDialog::getOpenFileName(this,

QStringLiteral("选择图片均值文件"),

" ",

QStringLiteral("图片均值(\*.binaryproto);;所有类型(\*.\*)")));

}

void Classification::Browser4()

{

ui.lineEdit\_labels->setText(

QFileDialog::getOpenFileName(this,

QStringLiteral("选择标签辞典文件"),

" ",

QStringLiteral("标签辞典(\*.txt);;所有类型(\*.\*)")));

}

void Classification::Browser5()

{

ui.lineEdit\_img->setText(

QFileDialog::getOpenFileName(this,

QStringLiteral("选择图片"),

" ",

QStringLiteral("图片(\*.jpg);;所有类型(\*.\*)")));

}

//分类

void Classification::DoClassification()

{

//校验参数

if (ui.lineEdit\_deploy->text().toStdString() != ""&&

ui.lineEdit\_caffemodel->text().toStdString() != ""&&

ui.lineEdit\_binaryproto->text().toStdString() != ""&&

ui.lineEdit\_labels->text().toStdString() != ""&&

ui.lineEdit\_img->text().toStdString() != "")

{

//显示图片

QImage \*\_img = new QImage();

if (\_img->load(ui.lineEdit\_img->text()) == true)

{

QGraphicsScene \*scene = new QGraphicsScene;

scene->addPixmap(QPixmap::fromImage(\*\_img));

ui.graphicsView->setScene(scene);

//构造命令

QString cmd="";

cmd.append(CoreDir);

cmd.append("/classification.exe");cmd.append(" ");

cmd.append(ui.lineEdit\_deploy->text()); cmd.append(" ");

cmd.append(ui.lineEdit\_caffemodel->text()); cmd.append(" ");

cmd.append(ui.lineEdit\_binaryproto->text()); cmd.append(" ");

cmd.append(ui.lineEdit\_labels->text()); cmd.append(" ");

cmd.append(ui.lineEdit\_img->text());

//启动额外线程

\_AdditionalProcess->start(cmd);

}

else

{

QMessageBox::information(this, QStringLiteral("分类测试"), QStringLiteral("图片打开失败!"));

return;

};

}

else

{

QMessageBox::information(this, QStringLiteral("分类测试"), QStringLiteral("参数有误!"));

return;

}

}

void Classification::ZoomImage()

{

qreal scale = qPow(qreal(2), (ui.horizontalSlider->value()) / qreal(25));

QMatrix matrix;//缩放矩阵

matrix.scale(scale, scale);

ui.graphicsView->setMatrix(matrix);

}

#pragma once

#include <QWidget>

#include "ui\_convertcifar.h"

#include "DataConverter.h"

class ConvertCifar : public QWidget

{

Q\_OBJECT

public:

ConvertCifar(QWidget \*parent = Q\_NULLPTR);

~ConvertCifar();

private:

Ui::ConvertCifar ui;

DataConverter \*MyConverter;

public slots:

void Browser1();

void Browser2();

void DoConvert();

};

#include "convertcifar.h"

#include <QFileDialog>

#include <QMessageBox>

#include <QTimer>

ConvertCifar::ConvertCifar(QWidget \*parent)

: QWidget(parent)

{

ui.setupUi(this);

MyConverter = new DataConverter();

ui.LmDbButton->setChecked(true);

}

ConvertCifar::~ConvertCifar()

{

}

void ConvertCifar::Browser1()

{

ui.lineEdit\_Inputfolder->setText(

QFileDialog::getExistingDirectory(this,

QStringLiteral("选择Cifar数据集目录"), QDir::currentPath()));

}

void ConvertCifar::Browser2()

{

ui.lineEdit\_outputfolder->setText(

QFileDialog::getExistingDirectory(this,

QStringLiteral("选择输出目录"), QDir::currentPath()));

}

void ConvertCifar::DoConvert()

{

std::string \_temp = (ui.LmDbButton->isChecked()?"lmdb":"leveldb");//lmdb

bool flag = MyConverter->CifarToDb(ui.lineEdit\_Inputfolder->text().toStdString(),

ui.lineEdit\_outputfolder->text().toStdString(), \_temp);

if (flag == false)

{

QMessageBox::information(this, QStringLiteral("数据转换"), QStringLiteral("转换失败!"));

}

else {

//启动转换线程

MyConverter->start();

int cnt = 0; // 循环次数计数器，用于"计算"当前的进度值

while (MyConverter->isRunning()) // 只要子线程还没有完成，就一直循环，并更新进度条

{

ui.progressBar->setValue((cnt++ % 20) \* 5); // 进度值每次递增5%，达到100后则再次从0开始

QEventLoop eventloop; // 使用事件循环阻塞主线程

QTimer::singleShot(500, &eventloop, SLOT(quit())); // wait 0.5s

eventloop.exec(); // 每0.5秒执行一次事件循环，然后更新进度条

}

ui.progressBar->setValue(100);

QMessageBox::information(this, QStringLiteral("数据转换"), QStringLiteral("转换完成!"));

}

}

#pragma once

#include <QWidget>

#include "ui\_convertimage.h"

#include "DataConverter.h"

class ConvertImage : public QWidget

{

Q\_OBJECT

public:

ConvertImage(QWidget \*parent = Q\_NULLPTR);

~ConvertImage();

private:

Ui::ConvertImage ui;

DataConverter \*MyConverter;

public slots:

void Browser1();

void Browser2();

void Browser3();

void SetDefault();

void DoConvert();

};

#include "convertimage.h"

#include <QFileDialog>

#include <QMessageBox>

#include <QTimer>

ConvertImage::ConvertImage(QWidget \*parent)

: QWidget(parent)

{

ui.setupUi(this);

MyConverter = new DataConverter();

ui.radioButton\_Lmdb->setChecked(true);

}

//选择文件列表文件

void ConvertImage::Browser1()

{

ui.lineEdit\_ImgFileList->setText(

QFileDialog::getOpenFileName(this,

QStringLiteral("选择图片文件列表"),

" ",

QStringLiteral("文件列表(\*.txt);;所有类型(\*.\*)")));

}

//选择根目录

void ConvertImage::Browser2()

{

ui.lineEdit\_RootDir->setText(

QFileDialog::getExistingDirectory(this,

QStringLiteral("选择图片文件的目录"), QDir::currentPath()));

}

//选择输出目录

void ConvertImage::Browser3()

{

ui.lineEdit\_OutputDir->setText(

QFileDialog::getExistingDirectory(this,

QStringLiteral("选择输出目录"), QDir::currentPath()));

}

//进行数据转换

void ConvertImage::DoConvert()

{

QString absDbName = ui.lineEdit\_OutputDir->text();

absDbName.append("\\");

absDbName.append(ui.lineEdit\_DBName->text());

std::string \_temp = (ui.radioButton\_Lmdb->isChecked() ? "lmdb" : "leveldb");

bool flag = MyConverter->ImagesToDb(ui.lineEdit\_ImgFileList->text().toStdString(),

ui.lineEdit\_RootDir->text().toStdString(),

absDbName.toStdString(),

ui.checkBox\_isGray->isChecked(),

ui.checkBox\_isShuffle->isChecked(),

\_temp,

ui.lineEdit\_resize\_width->text().toInt(),

ui.lineEdit\_resize\_height->text().toInt(),

ui.checkBox\_isCheck\_size->isChecked(),

ui.checkBox\_isEncoded->isChecked(),

ui.lineEdit\_encode\_type->text().toStdString()

);

if (flag == false)

{

QMessageBox::information(this, QStringLiteral("数据转换"), QStringLiteral("转换失败!"));

}

else {

//启动转换线程

MyConverter->start();

int cnt = 0; // 循环次数计数器，用于"计算"当前的进度值

while (MyConverter->isRunning()) // 只要子线程还没有完成，就一直循环，并更新进度条

{

ui.progressBar->setValue((cnt++ % 20) \* 5); // 进度值每次递增5%，达到100后则再次从0开始

QEventLoop eventloop; // 使用事件循环阻塞主线程

QTimer::singleShot(500, &eventloop, SLOT(quit())); // wait 0.5s

eventloop.exec(); // 每0.5秒执行一次事件循环，然后更新进度条

}

ui.progressBar->setValue(100);

QMessageBox::information(this, QStringLiteral("数据转换"), QStringLiteral("转换完成!"));

}

}

#pragma once

#include <QWidget>

#include "ui\_convertmnist.h"

#include "QFileDialog"

#include "DataConverter.h"

class ConvertMnist : public QWidget

{

Q\_OBJECT

public:

ConvertMnist(QWidget \*parent = Q\_NULLPTR);

~ConvertMnist();

void TellMeWhereIsTheBin(QString BinPath);

private:

Ui::ConvertMnist ui;

DataConverter \*MyConverter;

private slots:

void Browser1();

void Browser2();

void Browser3();

void DoConvert\_Mnist\_Data();

};

#include "convertmnist.h"

#include <string>

#include <QMessageBox>

#include <QTimer>

ConvertMnist::ConvertMnist(QWidget \*parent)

: QWidget(parent)

{

ui.setupUi(this);

MyConverter = new DataConverter();

ui.radioButton\_Lmdb->setChecked(true);

}

//选择图片集

void ConvertMnist::Browser1() {

ui.lineEdit\_MnistImageSet->setText(

QFileDialog::getOpenFileName(this,

QStringLiteral("选择图片集"),

" ",

QStringLiteral("Mnist图片集(\*.idx3-ubyte);;所有类型(\*.\*)")));

}

//选择标签集

void ConvertMnist::Browser2() {

ui.lineEdit\_MnistlabelSet->setText(

QFileDialog::getOpenFileName(this,

QStringLiteral("选择标签集"),

" ",

QStringLiteral("Mnist标签集(\*.idx1-ubyte);;所有类型(\*.\*)")));

}

//选择结果存放地址

void ConvertMnist::Browser3() {

ui.lineEdit\_MnistResult->setText(

QFileDialog::getExistingDirectory(this,

QStringLiteral("选择输出目录"), QDir::currentPath()));

}

//转换Mnist数据

void ConvertMnist::DoConvert\_Mnist\_Data()

{

QString tempQS = ui.lineEdit\_MnistResult->text();

tempQS.append("\\");

tempQS.append(ui.lineEdit\_ResultName->text());

std::string \_temp = (ui.radioButton\_Lmdb->isChecked() ? "lmdb" : "leveldb");

bool flag=MyConverter->MnistToDb(ui.lineEdit\_MnistImageSet->text().toStdString(),

ui.lineEdit\_MnistlabelSet->text().toStdString(),

tempQS.toStdString(), \_temp);

if (flag==false)

{

QMessageBox::information(this, QStringLiteral("数据转换"), QStringLiteral("转换失败!"));

}

else {

//启动转换线程

MyConverter->start();

int cnt = 0; // 循环次数计数器，用于"计算"当前的进度值

while (MyConverter->isRunning()) // 只要子线程还没有完成，就一直循环，并更新进度条

{

ui.progressBar->setValue((cnt++ % 20) \* 5); // 进度值每次递增5%，达到100后则再次从0开始

QEventLoop eventloop; // 使用事件循环阻塞主线程

QTimer::singleShot(500, &eventloop, SLOT(quit())); // wait 0.5s

eventloop.exec(); // 每0.5秒执行一次事件循环，然后更新进度条

}

ui.progressBar->setValue(100);

QMessageBox::information(this, QStringLiteral("数据转换"), QStringLiteral("转换完成!"));

}

}

#pragma once

#include <string>

#include <fstream> // NOLINT(readability/streams)

#include <QThread>

/\*

实现数据转换的实际业务逻辑,用来隔离Qmake和protoc的转换过程

\*/

class DataConverter:public QThread

{

public:

DataConverter();

~DataConverter();

enum mode {MNIST,CIFAR,IMAGE,DEFAULT};//运行模式,DEFAULT下不启动内核

public:

///外部操作

bool MnistToDb(std::string image\_filename, std::string label\_filename, std::string db\_path, std::string db\_backend);

bool CifarToDb(std::string& input\_folder,std::string& output\_folder,std::string& db\_type);

bool ImagesToDb(std::string&ImgFileList, std::string&RootDir, std::string&DbName, bool gray, bool shuffle, std::string & backend, int resize\_width, int resize\_height, bool check\_size, bool encoded, std::string & encode\_type);

void run();

private :

///内部变量

mode CurrentMode;//当前运行模式

//MNIST模式变量

std::string \_image\_filename;

std::string \_label\_filename;

std::string \_db\_path;

std::string \_db\_backend;

//CIFAR模式变量

std::string \_input\_folder;

std::string \_output\_folder;

std::string \_db\_type;

//IMAGE模式变量

std::string \_ImgFileList;

std::string \_RootDir;

std::string \_DbName;

bool \_gray;

bool \_shuffle;

std::string \_backend;

int \_resize\_width;

int \_resize\_height;

bool \_check\_size;

bool \_encoded;

std::string \_encode\_type;

private:

///内部操作11

uint32\_t swap\_endian(uint32\_t val);

void convert\_mnist\_dataset(std::string& image\_filename,

std::string& label\_filename,

std::string& db\_path,

std::string& db\_backend);

void read\_image(std::ifstream\* file, int\* label, char\* buffer);

void convert\_cifar\_dataset(std::string& input\_folder,

std::string& output\_folder,

std::string& db\_type);

void convert\_images(std::string&ImgFileList,//图片文件列表

std::string&RootDir,//图片集根目录

std::string&DbName,//生成的数据库的绝对文件名

bool gray,//是否将图片视为灰度图

bool shuffle,//是否对图片进行洗牌

std::string &backend,//使用何种数据库

int resize\_width,//将图片的宽度调整到多少

int resize\_height,//将图片的高度调整到多少

bool check\_size,//是否检查所有的datum尺寸的一致性

bool encoded,//是否将已编码的图片保存在Datum中

std::string &encode\_type//图片的编码形式(png,jpg等)

);

};

#include "DataConverter.h"

#include <gflags/gflags.h>

#include <glog/logging.h>

#include <google/protobuf/text\_format.h>

#include <lmdb.h>

#if defined(\_MSC\_VER)

#include <direct.h>

#define mkdir(X, Y) \_mkdir(X)

#endif

#include <stdint.h>

#include <sys/stat.h>

#include <algorithm>

#include <utility>

#include <vector>

#include "boost/scoped\_ptr.hpp"

#include "caffe/proto/caffe.pb.h"

#include "caffe/util/db.hpp"

#include "caffe/util/format.hpp"

#include "caffe/util/io.hpp"

#include "caffe/util/rng.hpp"

using namespace caffe; // NOLINT(build/namespaces)

using boost::scoped\_ptr;

using std::string;

const int kCIFARSize = 32;

const int kCIFARImageNBytes = 3072;

const int kCIFARBatchSize = 10000;

const int kCIFARTrainBatches = 5;

DataConverter::DataConverter()

:CurrentMode(DEFAULT)

, \_label\_filename(string(""))

, \_db\_path(string(""))

, \_db\_backend(string(""))

,\_image\_filename(string(""))

, \_input\_folder(string(""))

, \_output\_folder(string(""))

, \_db\_type(string(""))

, \_ImgFileList(string(""))

, \_RootDir(string(""))

, \_DbName(string(""))

, \_gray(true)

, \_shuffle(true)

, \_backend(string(""))

, \_resize\_width(0)

, \_resize\_height(0)

, \_check\_size(true)

, \_encoded(true)

, \_encode\_type(string(""))

{

}

DataConverter::~DataConverter()

{

}

uint32\_t DataConverter::swap\_endian(uint32\_t val)

{

val = ((val << 8) & 0xFF00FF00) | ((val >> 8) & 0xFF00FF);

return (val << 16) | (val >> 16);

}

void DataConverter::convert\_mnist\_dataset(string& image\_filename, string& label\_filename, string& db\_path, string& db\_backend)

{

// Open files

std::ifstream image\_file(image\_filename, std::ios::in | std::ios::binary);

std::ifstream label\_file(label\_filename, std::ios::in | std::ios::binary);

CHECK(image\_file) << "Unable to open file " << image\_filename;

CHECK(label\_file) << "Unable to open file " << label\_filename;

// Read the magic and the meta data

uint32\_t magic;

uint32\_t num\_items;

uint32\_t num\_labels;

uint32\_t rows;

uint32\_t cols;

image\_file.read(reinterpret\_cast<char\*>(&magic), 4);

magic = swap\_endian(magic);

CHECK\_EQ(magic, 2051) << "Incorrect image file magic.";

label\_file.read(reinterpret\_cast<char\*>(&magic), 4);

magic = swap\_endian(magic);

CHECK\_EQ(magic, 2049) << "Incorrect label file magic.";

image\_file.read(reinterpret\_cast<char\*>(&num\_items), 4);

num\_items = swap\_endian(num\_items);

label\_file.read(reinterpret\_cast<char\*>(&num\_labels), 4);

num\_labels = swap\_endian(num\_labels);

CHECK\_EQ(num\_items, num\_labels);

image\_file.read(reinterpret\_cast<char\*>(&rows), 4);

rows = swap\_endian(rows);

image\_file.read(reinterpret\_cast<char\*>(&cols), 4);

cols = swap\_endian(cols);

scoped\_ptr<db::DB> db(db::GetDB(db\_backend));

db->Open(db\_path, db::NEW);

scoped\_ptr<db::Transaction> txn(db->NewTransaction());

// Storing to db

char label;

char\* pixels = new char[rows \* cols];

int count = 0;

string value;

Datum datum;

datum.set\_channels(1);

datum.set\_height(rows);

datum.set\_width(cols);

LOG(INFO) << "A total of " << num\_items << " items.";

LOG(INFO) << "Rows: " << rows << " Cols: " << cols;

for (int item\_id = 0; item\_id < num\_items; ++item\_id) {

image\_file.read(pixels, rows \* cols);

label\_file.read(&label, 1);

datum.set\_data(pixels, rows\*cols);

datum.set\_label(label);

string key\_str = caffe::format\_int(item\_id, 8);

datum.SerializeToString(&value);

txn->Put(key\_str, value);

if (++count % 1000 == 0) {

txn->Commit();

}

}

// write the last batch

if (count % 1000 != 0) {

txn->Commit();

}

LOG(INFO) << "Processed " << count << " files.";

delete[] pixels;

db->Close();

}

void DataConverter::read\_image(std::ifstream \* file, int \* label, char \* buffer)

{

char label\_char;

file->read(&label\_char, 1);

\*label = label\_char;

file->read(buffer, kCIFARImageNBytes);

return;

}

void DataConverter::convert\_cifar\_dataset(string & input\_folder, string & output\_folder, string & db\_type)

{

scoped\_ptr<db::DB> train\_db(db::GetDB(db\_type));

train\_db->Open(output\_folder + "/cifar10\_train\_" + db\_type, db::NEW);

scoped\_ptr<db::Transaction> txn(train\_db->NewTransaction());

// Data buffer

int label;

char str\_buffer[kCIFARImageNBytes];

Datum datum;

datum.set\_channels(3);

datum.set\_height(kCIFARSize);

datum.set\_width(kCIFARSize);

LOG(INFO) << "Writing Training data";

for (int fileid = 0; fileid < kCIFARTrainBatches; ++fileid) {

// Open files

LOG(INFO) << "Training Batch " << fileid + 1;

string batchFileName = input\_folder + "/data\_batch\_"

+ caffe::format\_int(fileid + 1) + ".bin";

std::ifstream data\_file(batchFileName.c\_str(),

std::ios::in | std::ios::binary);

CHECK(data\_file) << "Unable to open train file #" << fileid + 1;

for (int itemid = 0; itemid < kCIFARBatchSize; ++itemid) {

read\_image(&data\_file, &label, str\_buffer);

datum.set\_label(label);

datum.set\_data(str\_buffer, kCIFARImageNBytes);

string out;

CHECK(datum.SerializeToString(&out));

txn->Put(caffe::format\_int(fileid \* kCIFARBatchSize + itemid, 5), out);

}

}

txn->Commit();

train\_db->Close();

LOG(INFO) << "Writing Testing data";

scoped\_ptr<db::DB> test\_db(db::GetDB(db\_type));

test\_db->Open(output\_folder + "/cifar10\_test\_" + db\_type, db::NEW);

txn.reset(test\_db->NewTransaction());

// Open files

std::ifstream data\_file((input\_folder + "/test\_batch.bin").c\_str(),

std::ios::in | std::ios::binary);

CHECK(data\_file) << "Unable to open test file.";

for (int itemid = 0; itemid < kCIFARBatchSize; ++itemid) {

read\_image(&data\_file, &label, str\_buffer);

datum.set\_label(label);

datum.set\_data(str\_buffer, kCIFARImageNBytes);

string out;

CHECK(datum.SerializeToString(&out));

txn->Put(caffe::format\_int(itemid, 5), out);

}

txn->Commit();

test\_db->Close();

}

void DataConverter::convert\_images(string&ImgFileList, string&RootDir,string&DbName,bool gray, bool shuffle, string & backend, int resize\_width, int resize\_height, bool check\_size, bool encoded, string & encode\_type)

{

#ifndef GFLAGS\_GFLAGS\_H\_

namespace gflags = google;

#endif

const bool is\_color = !gray;

//const bool check\_size = FLAGS\_check\_size;

//const bool encoded = FLAGS\_encoded;

//const string encode\_type = FLAGS\_encode\_type;

std::ifstream infile(ImgFileList);

std::vector<std::pair<std::string, int> > lines;

std::string line;

size\_t pos;

int label;

while (std::getline(infile, line)) {

pos = line.find\_last\_of(' ');

label = atoi(line.substr(pos + 1).c\_str());

lines.push\_back(std::make\_pair(line.substr(0, pos), label));

}

if (shuffle) {

// randomly shuffle data

LOG(INFO) << "Shuffling data";

caffe::shuffle(lines.begin(), lines.end());

}

LOG(INFO) << "A total of " << lines.size() << " images.";

if (encode\_type.size() && !encoded)

LOG(INFO) << "encode\_type specified, assuming encoded=true.";

int \_resize\_height = std::max<int>(0, resize\_height);

int \_resize\_width = std::max<int>(0, resize\_width);

// Create new DB

scoped\_ptr<db::DB> db(db::GetDB(backend));

db->Open(DbName, db::NEW);

scoped\_ptr<db::Transaction> txn(db->NewTransaction());

// Storing to db

//std::string root\_folder(RootDir);

Datum datum;

int count = 0;

int data\_size = 0;

bool data\_size\_initialized = false;

for (int line\_id = 0; line\_id < lines.size(); ++line\_id) {

bool status;

std::string enc = encode\_type;

if (encoded && !enc.size()) {

// Guess the encoding type from the file name

string fn = lines[line\_id].first;

size\_t p = fn.rfind('.');

if (p == fn.npos)

LOG(WARNING) << "Failed to guess the encoding of '" << fn << "'";

enc = fn.substr(p);

std::transform(enc.begin(), enc.end(), enc.begin(), ::tolower);

}

status = ReadImageToDatum(RootDir + lines[line\_id].first,

lines[line\_id].second, \_resize\_height, \_resize\_width, is\_color,

enc, &datum);

if (status == false) continue;

if (check\_size) {

if (!data\_size\_initialized) {

data\_size = datum.channels() \* datum.height() \* datum.width();

data\_size\_initialized = true;

}

else {

const std::string& data = datum.data();

CHECK\_EQ(data.size(), data\_size) << "Incorrect data field size "

<< data.size();

}

}

// sequential

string key\_str = caffe::format\_int(line\_id, 8) + "\_" + lines[line\_id].first;

// Put in db

string out;

CHECK(datum.SerializeToString(&out));

txn->Put(key\_str, out);

if (++count % 1000 == 0) {

// Commit db

txn->Commit();

txn.reset(db->NewTransaction());

LOG(INFO) << "Processed " << count << " files.";

}

}

// write the last batch

if (count % 1000 != 0) {

txn->Commit();

LOG(INFO) << "Processed " << count << " files.";

}

}

//转换Mnist数据集到本地数据库

bool DataConverter::MnistToDb(string image\_filename, string label\_filename, string db\_path, string db\_backend)

{

if (image\_filename != ""&&label\_filename != ""&&db\_path != ""&&

(db\_backend=="lmdb"|| db\_backend == "leveldb")) {

//传递值

\_image\_filename = image\_filename;

\_label\_filename = label\_filename;

\_db\_path = db\_path;

\_db\_backend = db\_backend;

CurrentMode = MNIST;

return true;

}

else {

return false;

}

}

//转换Cifar数据集到本地数据库

bool DataConverter::CifarToDb(string & input\_folder, string & output\_folder, string & db\_type)

{

if (input\_folder != ""&&output\_folder != "" &&

(db\_type == "lmdb" || db\_type == "leveldb")) {

\_input\_folder = input\_folder;

\_output\_folder = output\_folder;

\_db\_type = db\_type;

CurrentMode = CIFAR;

return true;

}

else

{

return false;

}

}

//将图片集装换到本地数据库

bool DataConverter::ImagesToDb(string & ImgFileList, string & RootDir, string & DbName, bool gray, bool shuffle, string & backend, int resize\_width, int resize\_height, bool check\_size, bool encoded, string & encode\_type)

{

if (ImgFileList!="" && RootDir!=""&& DbName!=""&&(backend=="lmdb"|| backend == "leveldb") )

{

if (encode\_type!="png"&&encode\_type != "jpg")

{

string temp = "";

\_ImgFileList = ImgFileList;

\_RootDir= RootDir;

\_DbName= DbName;

\_gray= gray;

\_shuffle= shuffle;

\_backend= backend;

\_resize\_width= resize\_width;

\_resize\_height= resize\_height;

\_check\_size= check\_size;

\_encoded= false;

\_encode\_type= temp;

}

else

{//正常调用

\_ImgFileList = ImgFileList;

\_RootDir = RootDir;

\_DbName = DbName;

\_gray = gray;

\_shuffle = shuffle;

\_backend = backend;

\_resize\_width = resize\_width;

\_resize\_height = resize\_height;

\_check\_size = check\_size;

\_encoded = encoded;

\_encode\_type = encode\_type;

}

CurrentMode = IMAGE;

return true;

}

else

{

return false;

}

}

//重载线程方法

void DataConverter::run()

{

//1.判断这一次运行是要转换什么数据集

switch (CurrentMode)

{

case MNIST:

convert\_mnist\_dataset(\_image\_filename, \_label\_filename, \_db\_path, \_db\_backend);

break;

case CIFAR:

convert\_cifar\_dataset(\_input\_folder, \_output\_folder, \_db\_type);

break;

case IMAGE:

convert\_images(\_ImgFileList, \_RootDir, \_DbName, \_gray, \_shuffle, \_backend, \_resize\_width, \_resize\_height, \_check\_size, \_encoded, \_encode\_type);

break;

default:

break;

}

/\*this->quit();\*/

}

#pragma once

//#include "boost/scoped\_ptr.hpp"

//#include "caffe/util/db.hpp"

#include <opencv2/core/core.hpp>

#include <opencv2/highgui.hpp>

class DbOperator

{

public:

DbOperator(std::string DbType,std::string filename);//初始化

~DbOperator();

public:

int numOfImage;

int length;

int width;

bool isColour;

public:

bool getImage(cv::Mat & img,int &label);

void heheh();

};

#include "DbOperator.h"

#include <stdint.h>

#include <algorithm>

#include <string>

#include <utility>

#include <vector>

#include <stdexcept>

#include <opencv2/core/core.hpp>

#include <opencv2/highgui.hpp>

#include "boost/scoped\_ptr.hpp"

#include "gflags/gflags.h"

#include "glog/logging.h"

#include "caffe/proto/caffe.pb.h"

#include "caffe/util/db.hpp"

#include "caffe/util/io.hpp"

using namespace caffe; // NOLINT(build/namespaces)

using std::max;

using std::pair;

using boost::scoped\_ptr;

boost::scoped\_ptr<db::DB> \_db;//db这个命名与某个命名空间冲突

boost::scoped\_ptr<db::Cursor> cursor;

//初始化

DbOperator::DbOperator(string DbType, string filename)

{

try

{

//默认使用lmdb

if (DbType=="leveldb")

{

\_db.reset(db::GetDB("leveldb"));

\_db->Open(filename, db::READ);//打开leveldb数据库

}

else {

\_db.reset(db::GetDB("lmdb"));

\_db->Open(filename, db::READ);//打开lmdb数据库

}

cursor.reset(\_db->NewCursor());//绑定游标

}

catch (const std::exception&e)

{

std::cout << "有问题!"<<e.what();

}

}

//获取当前游标下的图片,并把游标后移一位

bool DbOperator::getImage(cv::Mat & img, int &label)

{

if (cursor->valid())//当前游标有效

{

Datum datum;

bool flag = datum.ParseFromString(cursor->value());

string DarkMagic = cursor->value();//使用黑魔法强制解码

DecodeDatumNative(&datum);

if (datum.has\_data())

{

try

{

const string& data = datum.data();

int height = datum.height();

int width = datum.width();

label =datum.label();

cv::Mat cv\_img\_temp(height, width, CV\_8UC1);//构造一个空的Mat

for (int i = 0; i < height; i++) {

for (int j = 0; j < width; j++) {

char \_temp = data.at(i \* width + j);

cv\_img\_temp.at<char>(i,j)= data.at(i \* width + j);

}

}

cv\_img\_temp.copyTo(img);//传递结果

}

catch (std::out\_of\_range &exc)

{

std::cerr << exc.what() << " Line:" << \_\_LINE\_\_ << " File:" << \_\_FILE\_\_ << "\n";

}

}

else

{

//正常解码方案已经无法使用,尝试使用黑魔法

try

{

const string& data = datum.data();

int height = datum.height();

int width = datum.width();

label = datum.label();

cv::Mat cv\_img\_temp(height, width, CV\_8UC1);//构造一个空的Mat

for (int i = 0; i < height; i++) {

for (int j = 0; j < width; j++) {

cv\_img\_temp.at<char>(i, j) = DarkMagic.at(i \* width + j);

}

}

cv\_img\_temp.copyTo(img);//传递结果

}

catch (std::out\_of\_range &exc)

{

std::cerr << exc.what() << " Line:" << \_\_LINE\_\_ << " File:" << \_\_FILE\_\_ << "\n";

}

//return false;//当前游标指向了空位置

}

cursor->Next();

return true;

}

else

{

return false;//当前游标无效

}

}

#ifndef DIAGRAMITEM\_H

#define DIAGRAMITEM\_H

#include <QGraphicsPixmapItem>

#include <QList>

QT\_BEGIN\_NAMESPACE

class QPixmap;

class QGraphicsItem;

class QGraphicsScene;

class QTextEdit;

class QGraphicsSceneMouseEvent;

class QMenu;

class QGraphicsSceneContextMenuEvent;

class QPainter;

class QStyleOptionGraphicsItem;

class QWidget;

class QPolygonF;

QT\_END\_NAMESPACE

class Arrow;

//! [0]

class DiagramItem : public QGraphicsPixmapItem

{

public:

enum { Type = UserType + 15 };

enum DiagramType { InputLayer, Conditional, DataSource, Io, ConvLayer, PoolingLayer };

DiagramItem(DiagramType diagramType, QGraphicsItem \*parent = 0);

void removeArrow(Arrow \*arrow);

void removeArrows();

DiagramType diagramType() const { return myDiagramType; }

QPolygonF polygon() const { return myPolygon; }

void addArrow(Arrow \*arrow);

QPixmap image() const;

int type() const override { return Type; }

protected:

QVariant itemChange(GraphicsItemChange change, const QVariant &value) override;

private:

DiagramType myDiagramType;

QPolygonF myPolygon;

QList<Arrow \*> arrows;

QPixmap \* \_image;//当前图元对象的基础图形

};

//! [0]

#endif // DIAGRAMITEM\_H

#include "Diagramitem.h"

#include "Arrow.h"

#include <QGraphicsScene>

#include <QGraphicsSceneContextMenuEvent>

#include <QMenu>

#include <QPainter>

//! [0] 图元对象初始化

DiagramItem::DiagramItem(DiagramType diagramType, QGraphicsItem \*parent) : QGraphicsPixmapItem(parent)

{

myDiagramType = diagramType;

QPainterPath path;

int a = 0;

int b = 0;

switch (myDiagramType) {

case DataSource:

path.moveTo(100, -25);

path.arcTo(50, -25, 50, 50, 0, 90);

path.arcTo(-50, -25, 50, 50, 90, 90);

path.arcTo(-50, 25, 50, 50, 180, 90);

path.arcTo(50, 25, 50, 50, 270, 90);

//path.lineTo(200, 25);

myPolygon = path.toFillPolygon();

\_image = new QPixmap("./Resources/BP\_DataSource.png");

setPixmap(\_image->scaled(180, 90));

break;

case PoolingLayer:

myPolygon << QPointF(0, 0) << QPointF(300, 0)

<< QPointF(300, 150) << QPointF(0, 150) << QPointF(0, 0);

\_image = new QPixmap("./Resources/BP\_PoolingLayer.png");

setPixmap(\_image->scaled(180, 90));

/\*a = (\_image->logicalDpiX()) ;

b = (\_image->logicalDpiY()) ;

myPolygon << QPointF(0, 0) << QPointF(a, 0)

<< QPointF(a, b\*(1)) << QPointF(0, b\*(1)) << QPointF(0, 0);\*/

break;

case InputLayer:

myPolygon << QPointF(-100, -100) << QPointF(100, -100)

<< QPointF(100, 100) << QPointF(-100, 100)

<< QPointF(-100, -100);

\_image = new QPixmap("./Resources/BP\_DataLayer.png");

setPixmap(\_image->scaled(180, 90));

break;

case ConvLayer:

\_image = new QPixmap("./Resources/BP\_ConvLayer.png");

setPixmap(\_image->scaled(180, 90));

break;

default:

myPolygon << QPointF(-120, -80) << QPointF(-70, 80)

<< QPointF(120, 80) << QPointF(70, -80)

<< QPointF(-120, -80);

break;

}

//setPolygon(myPolygon);

setFlag(QGraphicsItem::ItemIsMovable, true);

setFlag(QGraphicsItem::ItemIsSelectable, true);

setFlag(QGraphicsItem::ItemSendsGeometryChanges, true);

}

void DiagramItem::removeArrow(Arrow \*arrow)

{

int index = arrows.indexOf(arrow);

if (index != -1)

arrows.removeAt(index);

}

void DiagramItem::removeArrows()

{

foreach(Arrow \*arrow, arrows) {

arrow->startItem()->removeArrow(arrow);

arrow->endItem()->removeArrow(arrow);

scene()->removeItem(arrow);

delete arrow;

}

}

void DiagramItem::addArrow(Arrow \*arrow)

{

arrows.append(arrow);

}

QPixmap DiagramItem::image() const

{

QPixmap pixmap(250, 250);

pixmap = \*\_image;

QPainter painter(&pixmap);

painter.setPen(QPen(Qt::black, 8));

painter.translate(125, 125);

return pixmap;

}

QVariant DiagramItem::itemChange(GraphicsItemChange change, const QVariant &value)

{

if (change == QGraphicsItem::ItemPositionChange) {

foreach(Arrow \*arrow, arrows) {

arrow->updatePosition();

}

}

return value;

}

//! [6]

#pragma once

#include "Diagramitem.h"

#include <QGraphicsScene>

#include <QAction>

QT\_BEGIN\_NAMESPACE

class QGraphicsSceneMouseEvent;

class QMenu;

class QPointF;

class QGraphicsLineItem;

class QFont;

class QGraphicsTextItem;

class QColor;

QT\_END\_NAMESPACE

//! [0]

class DiagramScene : public QGraphicsScene

{

Q\_OBJECT

public:

enum Mode { InsertItem, InsertLine, InsertText, MoveItem };

explicit DiagramScene(QObject \*parent = 0);

QFont font() const { return myFont; }

QColor textColor() const { return myTextColor; }

QColor itemColor() const { return myItemColor; }

QColor lineColor() const { return myLineColor; }

void setLineColor(const QColor &color);

void setTextColor(const QColor &color);

void setItemColor(const QColor &color);

void setFont(const QFont &font);

public slots:

void setMode(Mode mode);

void setItemType(DiagramItem::DiagramType type);

//void editorLostFocus(DiagramTextItem \*item);

signals:

void itemInserted(DiagramItem \*item);//插入了新的图元

void textInserted(QGraphicsTextItem \*item);//插入了文字

void itemSelected(QGraphicsItem \*item);//选中了图元

protected:

void mousePressEvent(QGraphicsSceneMouseEvent \*mouseEvent) override;

void mouseMoveEvent(QGraphicsSceneMouseEvent \*mouseEvent) override;

void mouseReleaseEvent(QGraphicsSceneMouseEvent \*mouseEvent) override;

private:

///内部变量

bool isItemChange(int type);

DiagramItem::DiagramType myItemType;

Mode myMode;

bool leftButtonDown;

QPointF startPoint;

QGraphicsLineItem \*line;

QFont myFont;

//DiagramTextItem \*textItem;

QColor myTextColor;

QColor myItemColor;

QColor myLineColor;

};

#include "DiagramScene.h"

#include "Arrow.h"

#include <QTextCursor>

#include <QGraphicsSceneMouseEvent>

#include <QAction>

#include <QMenu>

//! [0]构造

DiagramScene::DiagramScene(QObject \*parent) : QGraphicsScene(parent)

{

myMode = MoveItem;

myItemType = DiagramItem::InputLayer;

line = 0;

//textItem = 0;

myItemColor = Qt::white;

myTextColor = Qt::black;

myLineColor = Qt::black;

}

//! [0]

//! [1]设置线颜色

void DiagramScene::setLineColor(const QColor &color)

{

myLineColor = color;

if (isItemChange(Arrow::Type)) {

Arrow \*item = qgraphicsitem\_cast<Arrow \*>(selectedItems().first());

item->setColor(myLineColor);

update();

}

}

//! [1]

////! [2]设置文字颜色

//void DiagramScene::setTextColor(const QColor &color)

//{

// myTextColor = color;

// if (isItemChange(DiagramTextItem::Type)) {

// DiagramTextItem \*item = qgraphicsitem\_cast<DiagramTextItem \*>(selectedItems().first());

// item->setDefaultTextColor(myTextColor);

// }

//}

////! [2]

//! [3]设置图元颜色

void DiagramScene::setItemColor(const QColor &color)

{

myItemColor = color;

if (isItemChange(DiagramItem::Type)) {

DiagramItem \*item = qgraphicsitem\_cast<DiagramItem \*>(selectedItems().first());

//item->setBrush(myItemColor);

}

}

//! [3]

////! [4]设置字体

//void DiagramScene::setFont(const QFont &font)

//{

// myFont = font;

// if (isItemChange(DiagramTextItem::Type)) {

// QGraphicsTextItem \*item = qgraphicsitem\_cast<DiagramTextItem \*>(selectedItems().first());

// //At this point the selection can change so the first selected item might not be a DiagramTextItem

// if (item)

// item->setFont(myFont);

// }

//}

////! [4]

//槽函数:设置模式?

void DiagramScene::setMode(Mode mode)

{

myMode = mode;

}

//槽函数:设置图元的类型?

void DiagramScene::setItemType(DiagramItem::DiagramType type)

{

myItemType = type;

}

//! [6]重载:鼠标左键按下事件(画图形,划线,插入文本框)

void DiagramScene::mousePressEvent(QGraphicsSceneMouseEvent \*mouseEvent)

{

if (mouseEvent->button() != Qt::LeftButton) {

qDebug("Pos:(%.2f,%.2f)", mouseEvent->scenePos().rx(), mouseEvent->scenePos().ry());

return;

}

DiagramItem \*item;

switch (myMode) {

case InsertItem:

item = new DiagramItem(myItemType);

//item->setBrush(myItemColor);

addItem(item);

item->setPos(mouseEvent->scenePos());

emit itemInserted(item);//已经把图元插入到了指定位置

break;

//! [6] //! [7]

case InsertLine:

line = new QGraphicsLineItem(QLineF(mouseEvent->scenePos(),

mouseEvent->scenePos()));

line->setPen(QPen(Qt::black, 2));

addItem(line);

break;

//! [7] //! [8]

case InsertText:

//textItem = new DiagramTextItem();

//textItem->setFont(myFont);

//textItem->setTextInteractionFlags(Qt::TextEditorInteraction);

//textItem->setZValue(1000.0);

//connect(textItem, SIGNAL(lostFocus(DiagramTextItem\*)),

// this, SLOT(editorLostFocus(DiagramTextItem\*)));

//connect(textItem, SIGNAL(selectedChange(QGraphicsItem\*)),

// this, SIGNAL(itemSelected(QGraphicsItem\*)));

//addItem(textItem);

//textItem->setDefaultTextColor(myTextColor);

//textItem->setPos(mouseEvent->scenePos());

//emit textInserted(textItem);

////! [8] //! [9]

default:

;

}

QGraphicsScene::mousePressEvent(mouseEvent);

}

//! [9]

//! [10]重载:鼠标移动事件

void DiagramScene::mouseMoveEvent(QGraphicsSceneMouseEvent \*mouseEvent)

{

if (myMode == InsertLine && line != 0) {

QLineF newLine(line->line().p1(), mouseEvent->scenePos());

line->setLine(newLine);

}

else if (myMode == MoveItem) {

QGraphicsScene::mouseMoveEvent(mouseEvent);

}

}

//! [10]

//! [11]重载:鼠标释放事件

void DiagramScene::mouseReleaseEvent(QGraphicsSceneMouseEvent \*mouseEvent)

{

if (line != 0 && myMode == InsertLine) {

QList<QGraphicsItem \*> startItems = items(line->line().p1());

if (startItems.count() && startItems.first() == line)

startItems.removeFirst();

QList<QGraphicsItem \*> endItems = items(line->line().p2());

if (endItems.count() && endItems.first() == line)

endItems.removeFirst();

removeItem(line);

delete line;

//! [11] //! [12]

if (startItems.count() > 0 && endItems.count() > 0 &&

startItems.first()->type() == DiagramItem::Type &&

endItems.first()->type() == DiagramItem::Type &&

startItems.first() != endItems.first()) {

DiagramItem \*startItem = qgraphicsitem\_cast<DiagramItem \*>(startItems.first());

DiagramItem \*endItem = qgraphicsitem\_cast<DiagramItem \*>(endItems.first());

Arrow \*arrow = new Arrow(startItem, endItem);

arrow->setColor(myLineColor);

startItem->addArrow(arrow);

endItem->addArrow(arrow);

arrow->setZValue(-1000.0);

addItem(arrow);

arrow->updatePosition();

}

}

//! [12] //! [13]

line = 0;

QGraphicsScene::mouseReleaseEvent(mouseEvent);

}

//! [13]

//! [14]选中的图元发生变化

bool DiagramScene::isItemChange(int type)

{

foreach(QGraphicsItem \*item, selectedItems()) {

if (item->type() == type)

return true;

}

return false;

}

//! [14]

#ifndef MDICHILD\_H

#define MDICHILD\_H

#include <QTextEdit>

#include "typedef.h"

#include <QPlainTextEdit>

#include "myhighlighter.h"

class Editor : public QPlainTextEdit

{

Q\_OBJECT

public:

Editor(QWidget \*parent = 0);

void newFile();

bool loadFile(const QString &fileName);

bool save();

bool saveAs();

bool saveFile(const QString &fileName);

QString userFriendlyCurrentFile();

QString currentFile() { return curFile; }

void setMode(editorMode mode);

void lineNumberAreaPaintEvent(QPaintEvent \*event);

int lineNumberAreaWidth();

protected:

void closeEvent(QCloseEvent \*event) Q\_DECL\_OVERRIDE;

void resizeEvent(QResizeEvent \*event) Q\_DECL\_OVERRIDE;

private slots:

void documentWasModified();

void updateLineNumberAreaWidth(int newBlockCount);

void highlightCurrentLine();

void updateLineNumberArea(const QRect &, int);

private:

bool maybeSave();

void setCurrentFile(const QString &fileName);

QString strippedName(const QString &fullFileName);

QWidget \*lineNumberArea;

QString curFile;

bool isUntitled;

};

//行号

class LineNumberArea : public QWidget

{

public:

LineNumberArea(Editor \*editor) : QWidget(editor) {

codeEditor = editor;

}

QSize sizeHint() const Q\_DECL\_OVERRIDE {

return QSize(codeEditor->lineNumberAreaWidth(), 0);

}

protected:

void paintEvent(QPaintEvent \*event) Q\_DECL\_OVERRIDE {

codeEditor->lineNumberAreaPaintEvent(event);

}

private:

Editor \*codeEditor;

};

#endif

#include "editor.h"

#include <QtWidgets>

Editor::Editor(QWidget \*parent) : QPlainTextEdit(parent)

{

setAttribute(Qt::WA\_DeleteOnClose);

isUntitled = true;

lineNumberArea = new LineNumberArea(this);

connect(this, SIGNAL(blockCountChanged(int)), this, SLOT(updateLineNumberAreaWidth(int)));

connect(this, SIGNAL(updateRequest(QRect, int)), this, SLOT(updateLineNumberArea(QRect, int)));

connect(this, SIGNAL(cursorPositionChanged()), this, SLOT(highlightCurrentLine()));

QFont \*font = new QFont();//"MicrosoftYaHei"

font->setFamily(QStringLiteral("微软雅黑"));

font->setPointSize(12);

this->setFont(\*font);

updateLineNumberAreaWidth(0);

//setMode(BROWSE);

this->setStyleSheet("background:#f2f2f3;");

highlightCurrentLine();

setMode(EDIT);

MyHighLighter \*highlighter = new MyHighLighter(this->document());

}

//新建文件

void Editor::newFile()

{

static int sequenceNumber = 1;

isUntitled = true;

curFile = tr("document%1.txt").arg(sequenceNumber++);

setWindowTitle(curFile + "[\*]");

connect(document(), &QTextDocument::contentsChanged,

this, &Editor::documentWasModified);

}

//加载文件

bool Editor::loadFile(const QString &fileName)

{

QFile file(fileName);

if (!file.open(QFile::ReadOnly | QFile::Text)) {

QMessageBox::warning(this, tr("MDI"),

tr("Cannot read file %1:\n%2.")

.arg(fileName)

.arg(file.errorString()));

return false;

}

QTextStream in(&file);

QApplication::setOverrideCursor(Qt::WaitCursor);

setPlainText(in.readAll());

QApplication::restoreOverrideCursor();

setCurrentFile(fileName);

connect(document(), &QTextDocument::contentsChanged,

this, &Editor::documentWasModified);

return true;

}

//保存

bool Editor::save()

{

if (isUntitled) {

return saveAs();

}

else {

return saveFile(curFile);

}

}

//另存为

bool Editor::saveAs()

{

QString fileName = QFileDialog::getSaveFileName(this, tr("Save As"),

curFile);

if (fileName.isEmpty())

return false;

return saveFile(fileName);

}

bool Editor::saveFile(const QString &fileName)

{

QFile file(fileName);

if (!file.open(QFile::WriteOnly | QFile::Text)) {

QMessageBox::warning(this, tr("MDI"),

tr("Cannot write file %1:\n%2.")

.arg(QDir::toNativeSeparators(fileName), file.errorString()));

return false;

}

QTextStream out(&file);

QApplication::setOverrideCursor(Qt::WaitCursor);

out << toPlainText();

QApplication::restoreOverrideCursor();

setCurrentFile(fileName);

return true;

}

QString Editor::userFriendlyCurrentFile()

{

return strippedName(curFile);

}

//设置编辑器的模式

void Editor::setMode(editorMode mode)

{

if (mode == BROWSE)

{

this->setReadOnly(true);

this->setStyleSheet("background:#f2f2f3;");

highlightCurrentLine();

}

else if (mode == EDIT)

{

this->setReadOnly(false);

this->setStyleSheet("background:#ffffff;");

highlightCurrentLine();

}

}

//行号绘制事件

void Editor::lineNumberAreaPaintEvent(QPaintEvent \* event)

{

QPainter painter(lineNumberArea);

painter.fillRect(event->rect(), Qt::lightGray);

QTextBlock block = firstVisibleBlock();

int blockNumber = block.blockNumber();

int top = (int)blockBoundingGeometry(block).translated(contentOffset()).top();

int bottom = top + (int)blockBoundingRect(block).height();

while (block.isValid() && top <= event->rect().bottom()) {

if (block.isVisible() && bottom >= event->rect().top()) {

QString number = QString::number(blockNumber + 1);

painter.setPen(Qt::black);

painter.drawText(-2, top, lineNumberArea->width(), fontMetrics().height(),

Qt::AlignRight, number);

}

block = block.next();

top = bottom;

bottom = top + (int)blockBoundingRect(block).height();

++blockNumber;

}

}

//行号区域宽度

int Editor::lineNumberAreaWidth()

{

int digits = 1;

int max = qMax(1, blockCount());

while (max >= 10) {

max /= 10;

++digits;

}

int space = 3 + fontMetrics().width(QLatin1Char('9')) \* digits;

return space;

}

void Editor::closeEvent(QCloseEvent \*event)

{

if (maybeSave()) {

event->accept();

}

else {

event->ignore();

}

}

void Editor::documentWasModified()

{

setWindowModified(document()->isModified());

}

//更新行号区域宽度

void Editor::updateLineNumberAreaWidth(int /\*newBlockCount\*/)

{

setViewportMargins(lineNumberAreaWidth(), 0, 0, 0);

}

//将当前行高亮

void Editor::highlightCurrentLine()

{

QList<QTextEdit::ExtraSelection> extraSelections;

if (!isReadOnly()) {

QTextEdit::ExtraSelection selection;

QColor lineColor = QColor(Qt::yellow).lighter(160);

selection.format.setBackground(lineColor);

selection.format.setProperty(QTextFormat::FullWidthSelection, true);

//selection.cursor = textCursor();

//selection.cursor.clearSelection();

extraSelections.append(selection);

}

setExtraSelections(extraSelections);

}

//更新行号区域

void Editor::updateLineNumberArea(const QRect &rect, int dy)

{

if (dy)

lineNumberArea->scroll(0, dy);

else

lineNumberArea->update(0, rect.y(), lineNumberArea->width(), rect.height());

if (rect.contains(viewport()->rect()))

updateLineNumberAreaWidth(0);

}

bool Editor::maybeSave()

{

if (!document()->isModified())

return true;

const QMessageBox::StandardButton ret

= QMessageBox::warning(this, tr("MDI"),

tr("'%1' has been modified.\n"

"Do you want to save your changes?")

.arg(userFriendlyCurrentFile()),

QMessageBox::Save | QMessageBox::Discard

| QMessageBox::Cancel);

switch (ret) {

case QMessageBox::Save:

return save();

case QMessageBox::Cancel:

return false;

default:

break;

}

return true;

}

void Editor::setCurrentFile(const QString &fileName)

{

curFile = QFileInfo(fileName).canonicalFilePath();

isUntitled = false;

document()->setModified(false);

setWindowModified(false);

setWindowTitle(userFriendlyCurrentFile() + "[\*]");

}

QString Editor::strippedName(const QString &fullFileName)

{

return QFileInfo(fullFileName).fileName();

}

//调整大小事件

void Editor::resizeEvent(QResizeEvent \*e)

{

QPlainTextEdit::resizeEvent(e);

QRect cr = contentsRect();

lineNumberArea->setGeometry(QRect(cr.left(), cr.top(), lineNumberAreaWidth(), cr.height()));

}

#pragma once

#include <QWidget>

#include "ui\_featureexp.h"

#include <QProcess>

class FeatureExp : public QWidget

{

Q\_OBJECT

public:

FeatureExp(QString coredir, QProcess \*addprocess, QWidget \*parent = Q\_NULLPTR);

~FeatureExp();

private:

Ui::FeatureExp ui;

QString CoreDir;//保存核心目录

QProcess \*\_AdditionalProcess;//指向主界面类中已经重定向的那个线程,浅拷贝,构造赋值

public slots:

void Browser1();

void Browser2();

void Browser3();

void Verification();

void DoFeatureExp();

};

#include "featureexp.h"

#include <QFileDialog>

#include <QMessageBox>

#include <QTimer>

FeatureExp::FeatureExp(QString coredir, QProcess \*addprocess, QWidget \*parent)

: QWidget(parent)

{

ui.setupUi(this);

ui.radioButton\_lmdb->setChecked(true);

CoreDir = coredir;

\_AdditionalProcess = addprocess;

}

FeatureExp::~FeatureExp()

{

}

//选择训练好的网络模型

void FeatureExp::Browser1()

{

ui.lineEdit\_Model->setText(

QFileDialog::getOpenFileName(this,

QStringLiteral("选择训练好的网络模型"),

" ",

QStringLiteral("caffe网络模型(\*.caffemodel);;所有类型(\*.\*)")));

}

//选择网络部署描述文件

void FeatureExp::Browser2()

{

ui.lineEdit\_Prototxt->setText(

QFileDialog::getOpenFileName(this,

QStringLiteral("选择网络部署描述文件"),

" ",

QStringLiteral("部署描述文件(\*.prototxt);;所有类型(\*.\*)")));

}

//选择导出位置

void FeatureExp::Browser3()

{

ui.lineEdit\_ExpPath->setText(

QFileDialog::getExistingDirectory(this,

QStringLiteral("选择输出目录"), QDir::currentPath()));

}

//验证?验证什么鬼???直接上算了反正已经没时间了

void FeatureExp::Verification()

{

}

//执行特征导出

void FeatureExp::DoFeatureExp()

{

QString tempQS = ui.lineEdit\_ExpPath->text();

tempQS.append("/");

tempQS.append(ui.lineEdit\_DbName->text());

std::string \_temp = "lmdb";

//直接调用业务逻辑

//校验参数

if (ui.lineEdit\_Model->text() != ""&&

ui.lineEdit\_Prototxt->text() != ""&&

ui.lineEdit\_LayerID->text() != ""&&

ui.lineEdit\_MiniBatches->text() != ""&&

tempQS!="")

{

//构造命令

QString cmd = "";

cmd.append(CoreDir);

cmd.append("/extract\_features.exe"); cmd.append(" ");

cmd.append(ui.lineEdit\_Model->text()); cmd.append(" ");

cmd.append(ui.lineEdit\_Prototxt->text()); cmd.append(" ");

cmd.append(ui.lineEdit\_LayerID->text()); cmd.append(" ");

cmd.append(tempQS); cmd.append(" ");

cmd.append(ui.lineEdit\_MiniBatches->text()); cmd.append(" ");

cmd.append((ui.radioButton\_lmdb->isChecked() ? "lmdb" : "leveldb")); cmd.append(" ");

cmd.append((ui.comboBox\_cupgpu->currentIndex()==0 ? "CPU" : "GPU"));

//启动额外线程

\_AdditionalProcess->start(cmd);

}

else

{

QMessageBox::information(this, QStringLiteral("特征导出"), QStringLiteral("参数有误!"));

return;

}

}

#pragma once

#include <QWidget>

#include "ui\_homepage.h"

class HomePage : public QWidget

{

Q\_OBJECT

public:

HomePage(QWidget \*parent = Q\_NULLPTR);

~HomePage();

private:

Ui::HomePage ui;

public slots:

void LinkToAuthor();

void LinkToCaffe();

void HelpViewer();

void About();

};

#include "homepage.h"

#include <QDesktopServices>

#include <QUrl>

#include "about.h"

HomePage::HomePage(QWidget \*parent)

: QWidget(parent)

{

ui.setupUi(this);

}

HomePage::~HomePage()

{

}

void HomePage::LinkToAuthor()

{

QDesktopServices::openUrl(QUrl("http://my.csdn.net/Angle\_Cal"));

}

void HomePage::LinkToCaffe()

{

QDesktopServices::openUrl(QUrl("http://caffe.berkeleyvision.org/"));

}

void HomePage::HelpViewer()

{

QDesktopServices::openUrl(QUrl("https://github.com/BVLC/caffe/tree/windows"));

}

void HomePage::About()

{

class About \*\_aboutPage = new class About();

\_aboutPage->show();

}

#pragma once

#ifndef MYHIGHLIGHTER\_H

#define MYHIGHLIGHTER\_H

#include <QSyntaxHighlighter>

#include <QTextCharFormat>

#include <QTextDocument>

class MyHighLighter : public QSyntaxHighlighter

{

Q\_OBJECT

public:

MyHighLighter(QTextDocument \*parent = 0);

protected:

void highlightBlock(const QString &text) Q\_DECL\_OVERRIDE;

private:

struct HighlightingRule

{

QRegExp pattern;

QTextCharFormat format;

};

QVector<HighlightingRule> highlightingRules;

QRegExp commentStartExpression;

QRegExp commentEndExpression;

QTextCharFormat keywordFormat;

QTextCharFormat classFormat;

QTextCharFormat singleLineKey;

QTextCharFormat singleLineValue;

QTextCharFormat singleLineCommentFormat;

QTextCharFormat multiLineCommentFormat;

QTextCharFormat quotationFormat;

QTextCharFormat functionFormat;

};

#endif // MYHIGHLIGHTER\_H

#include "myhighlighter.h"

MyHighLighter::MyHighLighter(QTextDocument \*parent)

: QSyntaxHighlighter(parent)

{

HighlightingRule rule;

keywordFormat.setForeground(Qt::darkBlue);

keywordFormat.setFontWeight(QFont::Bold);

QStringList keywordPatterns;

//定义常量关键字

keywordPatterns << "\\bTEST\\b" << "\\bLMDB\\b" << "\\bMAX\\b"

<< "\\bCPU\\b" << "\\bGPU\\b";

foreach(const QString &pattern, keywordPatterns) {

rule.pattern = QRegExp(pattern);

rule.format = keywordFormat;

highlightingRules.append(rule);

}

classFormat.setFontWeight(QFont::Bold);

classFormat.setForeground(Qt::darkMagenta);

rule.pattern = QRegExp("\\b[A-Za-z0-9\_\\s]+(?=\\:)");//匹配后面带有":"的单词

rule.format = classFormat;

highlightingRules.append(rule);

singleLineCommentFormat.setForeground(Qt::red);

rule.pattern = QRegExp("#[^\n]\*");//匹配#注释符及其后面的语句

rule.format = singleLineCommentFormat;

highlightingRules.append(rule);

multiLineCommentFormat.setForeground(Qt::red);

quotationFormat.setForeground(Qt::darkGreen);

rule.pattern = QRegExp("\".\*\"");//匹配双引号引起来的字符串

rule.format = quotationFormat;

highlightingRules.append(rule);

functionFormat.setFontItalic(true);

functionFormat.setForeground(Qt::blue);

rule.pattern = QRegExp("\\b[A-Za-z0-9\_\\s]+(?=\\{)");//匹配后面带有大括号的标识符串

rule.format = functionFormat;

highlightingRules.append(rule);

commentStartExpression = QRegExp("/\\\*");//斜杠和\*号组成的多行注释体的起止

commentEndExpression = QRegExp("\\\*/");

}

void MyHighLighter::highlightBlock(const QString &text)

{

foreach(const HighlightingRule &rule, highlightingRules) {

QRegExp expression(rule.pattern);

int index = expression.indexIn(text);

while (index >= 0) {

int length = expression.matchedLength();

setFormat(index, length, rule.format);

index = expression.indexIn(text, index + length);

}

}

setCurrentBlockState(0);

int startIndex = 0;

if (previousBlockState() != 1)

startIndex = commentStartExpression.indexIn(text);

while (startIndex >= 0) {

int endIndex = commentEndExpression.indexIn(text, startIndex);

int commentLength;

if (endIndex == -1) {

setCurrentBlockState(1);

commentLength = text.length() - startIndex;

}

else {

commentLength = endIndex - startIndex

+ commentEndExpression.matchedLength();

}

setFormat(startIndex, commentLength, multiLineCommentFormat);

startIndex = commentStartExpression.indexIn(text, startIndex + commentLength);

}

}

#pragma once

#include <QMainWindow>

#include <QAction>

#include <QToolBox>

#include "ui\_neteditor.h"

#include "editor.h"

#include "DiagramScene.h"

class NetEditor : public QMainWindow

{

Q\_OBJECT

public:

///外部操作

NetEditor(QWidget \*parent = Q\_NULLPTR);

NetEditor(QString FileName,QWidget \*parent = Q\_NULLPTR);

~NetEditor();

private:

///内部变量

Ui::NetEditor ui;

Editor \*MyTextEditor;//文本编辑器

DiagramScene \*scene;//绘图区域

QButtonGroup \*buttonGroup;

QButtonGroup \*backgroundButtonGroup;

QGraphicsView \*view;

QButtonGroup \*pointerTypeGroup;

QComboBox \*sceneScaleCombo;

QToolBar \*pointerToolbar;

QAction\*deleteAction;//删除当前图元

QAction\*toFrontAction;

QAction\*sendBackAction;

QToolBox \*toolBox;//工具箱

private:

///内部操作

void createToolBox();

void createToolbars();

QWidget \*createCellWidget(const QString &text, DiagramItem::DiagramType type);//创建图元笔刷按钮

QWidget \*createBackgroundCellWidget(const QString &text, const QString &image);//创建背景切换按钮

private slots:

///槽函数

void itemInserted(DiagramItem \*item);//插入图元

void itemSelected(QGraphicsItem \*item);

void bringToFront();

void sendToBack();

void deleteItem();

void pointerGroupClicked(int id);//鼠标指针

void sceneScaleChanged(const QString &scale);//缩放

void backgroundButtonGroupClicked(QAbstractButton \*button);//点击换背景

void buttonGroupClicked(int id);//选中图元,点击画图

};

#include "neteditor.h"

#include <QToolButton>

#include <QLabel>

#include <QGraphicsView>

#include <QComboBox>

#include <Arrow.h>

//构造函数

NetEditor::NetEditor(QWidget \*parent)

: QMainWindow(parent)

{

ui.setupUi(this);

//启动格式文本编辑器,未来应该传入文件名用来进行初始化和作为文档的标题

MyTextEditor = new Editor();

MyTextEditor->newFile();

ui.gridLayout\_Text->addWidget(MyTextEditor);

//启动蓝图编辑器

createToolBox();//左侧工具箱的创建

scene = new DiagramScene(this);

scene->setSceneRect(QRectF(0, 0, 5000, 5000));

//连接信号槽

connect(scene, SIGNAL(itemInserted(DiagramItem\*)), this, SLOT(itemInserted(DiagramItem\*)));

//connect(scene, SIGNAL(textInserted(QGraphicsTextItem\*)),this, SLOT(textInserted(QGraphicsTextItem\*)));

//connect(scene, SIGNAL(itemSelected(QGraphicsItem\*)),this, SLOT(itemSelected(QGraphicsItem\*)));

createToolbars();//创建额外的工具条

view = new QGraphicsView(scene);

ui.horizontalLayout\_Blueprint->addWidget(toolBox);

ui.horizontalLayout\_Blueprint->addWidget(view);

}

//析构函数

NetEditor::~NetEditor()

{

}

//创建工具箱

void NetEditor::createToolBox()

{

buttonGroup = new QButtonGroup(this);

buttonGroup->setExclusive(false);

connect(buttonGroup, SIGNAL(buttonClicked(int)),this, SLOT(buttonGroupClicked(int)));

QGridLayout \*layout = new QGridLayout;

layout->addWidget(createCellWidget(QStringLiteral("数据源"), DiagramItem::DataSource), 0, 0);

layout->addWidget(createCellWidget(QStringLiteral("输入层"), DiagramItem::InputLayer), 0, 1);

layout->addWidget(createCellWidget(QStringLiteral("卷积层"), DiagramItem::ConvLayer), 1, 0);

layout->addWidget(createCellWidget(QStringLiteral("池化层"), DiagramItem::PoolingLayer), 1, 1);

layout->setRowStretch(3, 10);

layout->setColumnStretch(2, 10);

QWidget \*itemWidget = new QWidget;

itemWidget->setLayout(layout);

backgroundButtonGroup = new QButtonGroup(this);

connect(backgroundButtonGroup, SIGNAL(buttonClicked(QAbstractButton\*)),

this, SLOT(backgroundButtonGroupClicked(QAbstractButton\*)));

QGridLayout \*backgroundLayout = new QGridLayout;

backgroundLayout->addWidget(createBackgroundCellWidget(tr("Blue Grid"),

"./Resources/background1.png"), 0, 0);

backgroundLayout->addWidget(createBackgroundCellWidget(tr("White Grid"),

"./Resources/background2.png"), 0, 1);

backgroundLayout->addWidget(createBackgroundCellWidget(tr("Gray Grid"),

"./Resources/background3.png"), 1, 0);

backgroundLayout->addWidget(createBackgroundCellWidget(tr("No Grid"),

"./Resources/background4.png"), 1, 1);

backgroundLayout->setRowStretch(2, 10);

backgroundLayout->setColumnStretch(2, 10);

QWidget \*backgroundWidget = new QWidget;

backgroundWidget->setLayout(backgroundLayout);

//! [22]

toolBox = new QToolBox;

toolBox->setSizePolicy(QSizePolicy(QSizePolicy::Maximum, QSizePolicy::Ignored));

toolBox->setMinimumWidth(itemWidget->sizeHint().width());

toolBox->addItem(itemWidget, QStringLiteral("图元"));

toolBox->addItem(backgroundWidget, QStringLiteral("背景"));

}

//创建额外的工具条

void NetEditor::createToolbars()

{

QToolButton \*pointerButton = new QToolButton;

pointerButton->setCheckable(true);

pointerButton->setChecked(true);

pointerButton->setIcon(QIcon("./Resources/pointer.png"));

QToolButton \*linePointerButton = new QToolButton;

linePointerButton->setCheckable(true);

linePointerButton->setIcon(QIcon("./Resources/linepointer.png"));

pointerTypeGroup = new QButtonGroup(this);

pointerTypeGroup->addButton(pointerButton, int(DiagramScene::MoveItem));

pointerTypeGroup->addButton(linePointerButton, int(DiagramScene::InsertLine));

connect(pointerTypeGroup, SIGNAL(buttonClicked(int)),

this, SLOT(pointerGroupClicked(int)));

sceneScaleCombo = new QComboBox;

QStringList scales;

scales << tr("50%") << tr("75%") << tr("100%") << tr("125%") << tr("150%");

sceneScaleCombo->addItems(scales);

sceneScaleCombo->setCurrentIndex(2);

connect(sceneScaleCombo, SIGNAL(currentIndexChanged(QString)),

this, SLOT(sceneScaleChanged(QString)));

pointerToolbar = addToolBar(tr("Pointer type"));

pointerToolbar->addWidget(pointerButton);

pointerToolbar->addWidget(linePointerButton);

pointerToolbar->addWidget(sceneScaleCombo);

}

//创建图元画刷按钮

QWidget \* NetEditor::createCellWidget(const QString & text, DiagramItem::DiagramType type)

{

DiagramItem item(type);

QIcon icon(item.image());

QToolButton \*button = new QToolButton;

button->setIcon(icon);

button->setIconSize(QSize(50, 50));

button->setCheckable(true);

buttonGroup->addButton(button, int(type));

QGridLayout \*layout = new QGridLayout;

layout->addWidget(button, 0, 0, Qt::AlignHCenter);

layout->addWidget(new QLabel(text), 1, 0, Qt::AlignCenter);

QWidget \*widget = new QWidget;

widget->setLayout(layout);

return widget;

}

//创建背景切换按钮

QWidget \* NetEditor::createBackgroundCellWidget(const QString & text, const QString & image)

{

QToolButton \*button = new QToolButton;

button->setText(text);

button->setIcon(QIcon(image));

button->setIconSize(QSize(50, 50));

button->setCheckable(true);

backgroundButtonGroup->addButton(button);

QGridLayout \*layout = new QGridLayout;

layout->addWidget(button, 0, 0, Qt::AlignHCenter);

layout->addWidget(new QLabel(text), 1, 0, Qt::AlignCenter);

QWidget \*widget = new QWidget;

widget->setLayout(layout);

return widget;

}

/\*==============================================================================\*/

/\* 槽函数 \*/

/\*==============================================================================\*/

//槽函数:图元插入

void NetEditor::itemInserted(DiagramItem \*item)

{

pointerTypeGroup->button(int(DiagramScene::MoveItem))->setChecked(true);

scene->setMode(DiagramScene::Mode(pointerTypeGroup->checkedId()));

buttonGroup->button(int(item->diagramType()))->setChecked(false);

}

//蛤?

void NetEditor::itemSelected(QGraphicsItem \* item)

{

//DiagramTextItem \*textItem =

// qgraphicsitem\_cast<DiagramTextItem \*>(item);

//QFont font = textItem->font();

//fontCombo->setCurrentFont(font);

//fontSizeCombo->setEditText(QString().setNum(font.pointSize()));

//boldAction->setChecked(font.weight() == QFont::Bold);

//italicAction->setChecked(font.italic());

//underlineAction->setChecked(font.underline());

}

//槽函数:置于顶层

void NetEditor::bringToFront()

{

if (scene->selectedItems().isEmpty())

return;

QGraphicsItem \*selectedItem = scene->selectedItems().first();

QList<QGraphicsItem \*> overlapItems = selectedItem->collidingItems();

qreal zValue = 0;

foreach(QGraphicsItem \*item, overlapItems) {

if (item->zValue() >= zValue && item->type() == DiagramItem::Type)

zValue = item->zValue() + 0.1;

}

selectedItem->setZValue(zValue);

}

//槽函数:置于底层

void NetEditor::sendToBack()

{

if (scene->selectedItems().isEmpty())

return;

QGraphicsItem \*selectedItem = scene->selectedItems().first();

QList<QGraphicsItem \*> overlapItems = selectedItem->collidingItems();

qreal zValue = 0;

foreach(QGraphicsItem \*item, overlapItems) {

if (item->zValue() <= zValue && item->type() == DiagramItem::Type)

zValue = item->zValue() - 0.1;

}

selectedItem->setZValue(zValue);

}

//槽函数:删除元素

void NetEditor::deleteItem()

{

foreach(QGraphicsItem \*item, scene->selectedItems()) {

if (item->type() == Arrow::Type) {

scene->removeItem(item);

Arrow \*arrow = qgraphicsitem\_cast<Arrow \*>(item);

arrow->startItem()->removeArrow(arrow);

arrow->endItem()->removeArrow(arrow);

delete item;

}

}

foreach(QGraphicsItem \*item, scene->selectedItems()) {

if (item->type() == DiagramItem::Type)

qgraphicsitem\_cast<DiagramItem \*>(item)->removeArrows();

scene->removeItem(item);

delete item;

}

}

//槽函数:鼠标指针变更

void NetEditor::pointerGroupClicked(int id)

{

scene->setMode(DiagramScene::Mode(pointerTypeGroup->checkedId()));

}

//槽函数:缩放

void NetEditor::sceneScaleChanged(const QString & scale)

{

double newScale = scale.left(scale.indexOf(tr("%"))).toDouble() / 100.0;

QMatrix oldMatrix = view->matrix();

view->resetMatrix();

view->translate(oldMatrix.dx(), oldMatrix.dy());

view->scale(newScale, newScale);

}

//槽函数:点击切换背景

void NetEditor::backgroundButtonGroupClicked(QAbstractButton \* button)

{

QList<QAbstractButton \*> buttons = backgroundButtonGroup->buttons();

foreach(QAbstractButton \*myButton, buttons) {

if (myButton != button)

button->setChecked(false);

}

QString text = button->text();

if (text == tr("Blue Grid"))

scene->setBackgroundBrush(QPixmap("./Resources/background1.png"));

else if (text == tr("White Grid"))

scene->setBackgroundBrush(QPixmap("./Resources/background2.png"));

else if (text == tr("Gray Grid"))

scene->setBackgroundBrush(QPixmap("./Resources/background3.png"));

else

scene->setBackgroundBrush(QPixmap("./Resources/background4.png"));

scene->update();

view->update();

}

//槽函数:选中图元,点击画图

void NetEditor::buttonGroupClicked(int id)

{

QList<QAbstractButton \*> buttons = buttonGroup->buttons();

foreach(QAbstractButton \*button, buttons) {

if (buttonGroup->button(id) != button)

button->setChecked(false);

}

/\* if (id == InsertTextButton) {

scene->setMode(DiagramScene::InsertText);

}

else {\*/

scene->setItemType(DiagramItem::DiagramType(id));

scene->setMode(DiagramScene::InsertItem);

//}

}

#pragma once

#include <QWidget>

#include "ui\_preferences.h"

#include "QStandardItemModel.h"

class Preferences : public QWidget

{

Q\_OBJECT

public:

Preferences(QWidget \*parent = Q\_NULLPTR);

~Preferences();

private:

Ui::Preferences ui;

///业务操作

private:

void initialize();//初始化

void iniFileSwitch(QString path);//配置文件切换

///从文件读取,有问题返回NULL

QString readInstallPath();//读取install目录

QString readBinPath();//读取bin目录

QString readLibPath();//读取Lib目录

QString readScript(QString Func);//读取对应func功能的脚本

///写入文件

void writeInstallPath(QString path);//写入install目录

void writeBinPath(QString path);//写入bin目录

void writeLibPath(QString path);//写入Lib目录

void writeScript(QString Func,QString script);//写入对应func功能的脚本

public:

void awakenPreferencesManager();//唤醒首选项管理器

///从此类外部访问配置

QString defaultInstallPath;//默认install目录

QString defaultBinPath;//默认bin目录

QString defaultLibPath;//默认Lib目录

QVector <QString> FuncList;//存储脚本的key,命名规则为对应脚本的功能

QMap <QString, QString> Scripts;//存储所有的脚本

QString defaultScript\_Train;//默认脚本

public:

///保存默认参数

QString getInstallPath();//获取install目录

QString getBinPath();//获取bin目录

QString getLibPath();//获取Lib目录

QString getScript(QString Func);//获取对应func功能的脚本

private:

///维持本类正常工作所需的内部参数

QString IniFilePath;//保存当前所使用的配置文件路径

QStandardItemModel \*ScriptsModel;//命令表格模型

private slots:

void Browser1();

void Browser2();

void Browser3();

void checkComponent();

void setDefaultPath();

void creatEnvironmentVariable();

void setDefaultScripts();

void save();

};

#include "preferences.h"

#include "QSettings.h"

#include "QFile.h"

#include "QMessageBox.h"

#include "QFileDialog"

//构造函数

Preferences::Preferences(QWidget \*parent)

: QWidget(parent)

{

ui.setupUi(this);

//额外的ui初始化

ScriptsModel = new QStandardItemModel();

ScriptsModel->setColumnCount(2);

ScriptsModel->setHeaderData(0, Qt::Horizontal, QString::fromLocal8Bit("功能"));

ScriptsModel->setHeaderData(1, Qt::Horizontal, QString::fromLocal8Bit("命令"));

ui.tableView->setModel(ScriptsModel);

ui.tableView->horizontalHeader()->setDefaultAlignment(Qt::AlignLeft);//表头信息显示居左

/\*==========默认参数==========\*/

IniFilePath = "./EngineStart.ini";//默认配置文件地址

FuncList.append("train");//脚本的功能,同时也是索引脚本内容的键值

FuncList.append("test");

FuncList.append("convert\_image");

Scripts.insert("train", " train --solver=");//对应键值的脚本的默认内容

Scripts.insert("test", " test\_Script");

Scripts.insert("convert\_image", " convert\_image\_Script");

defaultInstallPath = "E:\\VisualStudio\\FusionStudio\\FusionStudio\\Components\\caffe-CPU-py27";//默认install目录

defaultBinPath="E:\\VisualStudio\\FusionStudio\\FusionStudio\\Components\\caffe-CPU-py27\\bin";//默认bin目录

defaultLibPath="E:\\VisualStudio\\FusionStudio\\FusionStudio\\Components\\caffe-CPU-py27\\lib";//默认Lib目录

//defaultScript\_Train=" train --solver=";//默认脚本

//初始化方法

initialize();

}

//析构函数

Preferences::~Preferences()

{

}

/\*

方法:初始化

功能:查看配置文件是否存在,如果不存在,则建立配置文件,并写入默认数据

\*/

void Preferences::initialize()

{

//QFile file(IniFilePath);

QFile \*file = new QFile(IniFilePath);

if (file->exists()) {

//QMessageBox::information(this, "测试", QStringLiteral("文件存在"));

}

else {//配置文件不存在

//Qt中使用QSettings类读写ini文件

//QSettings构造函数的第一个参数是ini文件的路径,第二个参数表示针对ini文件,第三个参数可以缺省

QSettings \*configIniWrite = new QSettings(IniFilePath, QSettings::IniFormat);

//向ini文件中写入内容,setValue函数的两个参数是键值对

//向ini文件中写入默认配置参数

configIniWrite->setValue("path/install", defaultInstallPath);

configIniWrite->setValue("path/bin", defaultBinPath);

configIniWrite->setValue("path/lib", defaultLibPath);

//将命令脚本的键值(就是对应脚本的的功能)写入ini文件的一个数组里面

configIniWrite->beginWriteArray("FuncList");

for (int i = 0; i < FuncList.size(); i++)

{

configIniWrite->setArrayIndex(i);

configIniWrite->setValue("Func", FuncList[i]);

}

configIniWrite->endArray();

//将键值对应的值写入ini文件

configIniWrite->beginGroup("Scripts");

for (int i = 0; i < FuncList.size(); i++)

{

QMap<QString, QString>::iterator it = Scripts.find(FuncList[i]);

configIniWrite->setValue(FuncList[i], it.value());

}

configIniWrite->endGroup();

//写入完成后删除指针

delete configIniWrite;

};

delete file;

}

//切换配置文件

void Preferences::iniFileSwitch(QString path)

{

IniFilePath = path;

initialize();

}

//唤醒首选项管理器

void Preferences::awakenPreferencesManager()

{

//把目前的状态显示到界面上

//显示路径设置

ui.lineEdit\_BinPath->setText(readBinPath());

ui.lineEdit\_InstallPath->setText(readInstallPath());

ui.lineEdit\_LibPath->setText(readLibPath());

//显示命令设置(表格)

///首先查询出所有的key,放在一个临时容器(向量)里面,

QVector <QString> \_tempFuncList;

QSettings \*mod\_read = new QSettings(IniFilePath, QSettings::IniFormat, 0);

int num = mod\_read->beginReadArray("FuncList");

for (int i = 0; i<num; i++)

{

mod\_read->setArrayIndex(i);

\_tempFuncList.append( mod\_read->value("Func", "该命令为空").toString());

}

mod\_read->endArray();

///然后进入循环,用key从文件中查询出内容,再把key和内容写到同一行中,然后循环变量后移

mod\_read->beginGroup("Scripts");

for (int i = 0; i < num; i++)

{

ScriptsModel->setItem(i, 0, new QStandardItem(\_tempFuncList[i]));//功能字段

ScriptsModel->item(i, 0)->setForeground(QBrush(QColor(255, 0, 0)));//设置字符颜色

ScriptsModel->item(i, 0)->setTextAlignment(Qt::AlignCenter);//设置字符位置

ScriptsModel->setItem(i, 1,

new QStandardItem(mod\_read->value(\_tempFuncList[i]).toString()));//命令字段

}

mod\_read->endGroup();

delete mod\_read;

}

//从文件中读取install路径

QString Preferences::readInstallPath()

{

QSettings \*configIniRead = new QSettings(IniFilePath, QSettings::IniFormat);

//将读取到的ini文件保存在QString中，先取值，然后通过toString()函数转换成QString类型

QString Result = configIniRead->value("path/install").toString();

//读入入完成后删除指针

delete configIniRead;

return Result;

}

//从文件中读取bin的路径

QString Preferences::readBinPath()

{

QSettings \*configIniRead = new QSettings(IniFilePath, QSettings::IniFormat);

QString Result = configIniRead->value("path/bin").toString();

delete configIniRead;

return Result;

}

//从文件中读取lib的路径

QString Preferences::readLibPath()

{

QSettings \*configIniRead = new QSettings(IniFilePath, QSettings::IniFormat);

QString Result = configIniRead->value("path/lib").toString();

delete configIniRead;

return Result;

}

//从文件中读取对应Func的脚本

QString Preferences::readScript(QString Func)

{

QSettings \*configIniRead = new QSettings(IniFilePath, QSettings::IniFormat);

configIniRead->beginGroup("Scripts");

QString Result = configIniRead->value(Func,"").toString();

configIniRead->endGroup();

delete configIniRead;

return Result;

}

//将install路径写入文件

void Preferences::writeInstallPath(QString path)

{

QSettings \*configIniWrite = new QSettings(IniFilePath, QSettings::IniFormat);

configIniWrite->setValue("path/install", path);

delete configIniWrite;

}

//将bin路径写入文件

void Preferences::writeBinPath(QString path)

{

QSettings \*configIniWrite = new QSettings(IniFilePath, QSettings::IniFormat);

configIniWrite->setValue("path/bin", path);

delete configIniWrite;

}

//将lib路径写入文件

void Preferences::writeLibPath(QString path)

{

QSettings \*configIniWrite = new QSettings(IniFilePath, QSettings::IniFormat);

configIniWrite->setValue("path/lib", path);

delete configIniWrite;

}

//将对应Func的脚本写入文件

void Preferences::writeScript(QString Func, QString script)

{

QSettings \*configIniWrite = new QSettings(IniFilePath, QSettings::IniFormat);

configIniWrite->beginGroup("Scripts");

configIniWrite->setValue(Func, script);

configIniWrite->endGroup();

delete configIniWrite;

}

//获取install目录

QString Preferences::getInstallPath()

{

return readInstallPath();

}

//获取bin目录

QString Preferences::getBinPath()

{

return readBinPath();

}

//获取lib目录

QString Preferences::getLibPath()

{

return readLibPath();

}

//获取指定脚本

QString Preferences::getScript(QString Func)

{

return readScript(Func);

}

//浏览1

void Preferences::Browser1()

{

QString tempInstallPath = QFileDialog::getExistingDirectory(this,tr("选择install目录"), QDir::currentPath());

if (tempInstallPath!="")

{

ui.lineEdit\_InstallPath->setText(tempInstallPath);

}

}

//浏览2

void Preferences::Browser2()

{

QString tempBinPath = QFileDialog::getExistingDirectory(this,tr("选择bin目录"), QDir::currentPath());

if (tempBinPath!="")

{

ui.lineEdit\_BinPath->setText(tempBinPath);

}

}

//浏览3

void Preferences::Browser3()

{

QString tempLibPath = QFileDialog::getExistingDirectory(this, tr("选择lib目录"), QDir::currentPath());

if (tempLibPath != "") {

ui.lineEdit\_LibPath->setText(tempLibPath);

}

}

//检查组件状态

void Preferences::checkComponent()

{

//构造检查表

QString CheckList[]{ "caffe.exe","classification.exe","compute\_image\_mean.exe",

"convert\_cifar\_data.exe","convert\_imageset.exe","convert\_mnist\_data.exe",

"convert\_mnist\_siamese\_data.exe","device\_query.exe","extract\_features.exe",

"finetune\_net.exe","net\_speed\_benchmark.exe","test\_net.exe",

"train\_net.exe","upgrade\_net\_proto\_binary.exe","upgrade\_net\_proto\_text.exe",

"upgrade\_solver\_proto\_text.exe" };

ui.plainTextEdit\_ComponentState->clear();

QString tempBinPath = readBinPath();

if (tempBinPath!="")

{

ui.plainTextEdit\_ComponentState->appendPlainText(QStringLiteral("正在校验核心组件:")+ tempBinPath);

tempBinPath = tempBinPath.append("\\");

for (int i = 0; i < 16; i++)

{

QString \_currentFile = tempBinPath;

\_currentFile.append(CheckList[i]);

QFile file(\_currentFile);

if (file.exists()) {

ui.plainTextEdit\_ComponentState->appendPlainText(CheckList[i] + "...OK!");

}

else

{

ui.plainTextEdit\_ComponentState->appendPlainText(CheckList[i] + "...ERROR!");

}

}

}

else

{

ui.plainTextEdit\_ComponentState->appendPlainText(QStringLiteral("bin目录为空!"));

}

}

//全部路径设置成默认参数

void Preferences::setDefaultPath()

{

ui.lineEdit\_BinPath->setText(defaultBinPath);

ui.lineEdit\_InstallPath->setText(defaultInstallPath);

ui.lineEdit\_LibPath->setText(defaultLibPath);

}

void Preferences::creatEnvironmentVariable()

{

//导入环境变量

}

//全部脚本设为默认

void Preferences::setDefaultScripts()

{

for (int i = 0; i < FuncList.size(); i++)

{

QMap<QString, QString>::iterator it = Scripts.find(FuncList[i]);

ScriptsModel->setItem(i, 0, new QStandardItem(FuncList[i]));//功能字段

ScriptsModel->item(i, 0)->setForeground(QBrush(QColor(255, 0, 0)));//设置字符颜色

ScriptsModel->item(i, 0)->setTextAlignment(Qt::AlignCenter);//设置字符位置

ScriptsModel->setItem(i, 1, new QStandardItem(it.value()));//命令字段

}

}

//保存

void Preferences::save()

{

//保存路径设置

writeBinPath(ui.lineEdit\_BinPath->text());

writeInstallPath(ui.lineEdit\_InstallPath->text());

writeLibPath(ui.lineEdit\_LibPath->text());

//保存脚本设置

///读取整个表格,非空的进行覆盖

ScriptsModel->rowCount();//一共多少行

QString \_tempFunc = ScriptsModel->data(ScriptsModel->index(3, 1)).toString();//第3行第1列的内容

QString \_tempScript;

for (int i = 0; i < ScriptsModel->rowCount(); i++)

{

\_tempFunc = ScriptsModel->data(ScriptsModel->index(i, 0)).toString();//第i行第1列的内容

\_tempScript= ScriptsModel->data(ScriptsModel->index(i, 1)).toString();//第i行第2列的内容

if (\_tempFunc!=""&&\_tempScript!="")

{

writeScript(\_tempFunc, \_tempScript);

}

}

}

#pragma once

#include <QObject>

class Project : public QObject

{

Q\_OBJECT

public:

///构造和析构

Project(QObject \*parent = 0);

~Project();

private:

///工程的私有属性

QString WorkingDir; //该工程的工作目录

public:

///属性的get和set方法

void setWorkingDir(QString);//工作目录的set方法

QString getWorkingDir(); //工作目录的get方法

signals :

void ValueChanged(); //自定义值更改信号

};

#include "Project.h"

Project::Project(QObject \*parent)

: QObject(parent)

{

WorkingDir = "";

}

Project::~Project()

{

}

void Project::setWorkingDir(QString NewValue)

{

if (NewValue != WorkingDir)

{

WorkingDir = NewValue;

emit ValueChanged();//值更改,发出信号

}

}

QString Project::getWorkingDir()

{

return WorkingDir;

}

#pragma once

#include <QWidget>

#include "ui\_trainingsetter.h"

#include <QProcess>

#include <QString>

class TrainingSetter : public QWidget

{

Q\_OBJECT

public:

TrainingSetter(QString coredir, QProcess \*addprocess,QWidget \*parent = Q\_NULLPTR);

~TrainingSetter();

private:

///内部属性

Ui::TrainingSetter ui;

QString CoreDir;//保存核心目录

QProcess \*\_AdditionalProcess;//指向主界面类中已经重定向的那个线程,浅拷贝,构造赋值

private:

///内部操作

public slots :

///槽函数

void BrowseSolver();//浏览训练配置文件

void BrowseSnapPath();

void BrowseNet();

void ReadSolver();

void SaveSolver();

void ExecuteTraining();

void policyChange();//衰减策略变更

};

#include "trainingsetter.h"

//#include "caffe/proto/caffe.pb.h"//无法引用这个文件,需要借助中间层

#include <QFileDialog>

#include <QFile>

#include <QTextStream>

#include <QMessageBox>

TrainingSetter::TrainingSetter(QString coredir, QProcess \*addprocess, QWidget \*parent)

: QWidget(parent)

{

ui.setupUi(this);

CoreDir = coredir;

\_AdditionalProcess = addprocess;

}

TrainingSetter::~TrainingSetter()

{

}

//选择求解器文件

void TrainingSetter::BrowseSolver()

{

/\*

参数1：父窗口

参数2：对话框的标题

参数3：默认的打开的位置，如”我的文档“等

参数4：文件的过滤器，注意文件类型之间用 ；； 分开

\*/

ui.lineEdit\_solverFilePath->setText(

QFileDialog::getOpenFileName(this,

QStringLiteral("求解器"),

" ",

QStringLiteral("求解器定义文件(\*.prototxt);;所有类型(\*.\*)")));

}

//选择网络快照的输出位置

void TrainingSetter::BrowseSnapPath()

{

ui.lineEdit\_snapshot\_prefix->setText(

QFileDialog::getExistingDirectory(this,

QStringLiteral("选择快照的输出目录"), QDir::currentPath()));

}

//选择网络结构定义文件

void TrainingSetter::BrowseNet()

{

ui.lineEdit\_net->setText(

QFileDialog::getOpenFileName(this,

QStringLiteral("网络结构定义文件"),

" ",

QStringLiteral("网络结构定义文件(\*.prototxt);;所有类型(\*.\*)")));

}

//将界面的相关参数保存在求解器文件中

void TrainingSetter::SaveSolver()

{

QFile file(ui.lineEdit\_solverFilePath->text());

if (file.open(QFile::WriteOnly | QFile::Text)) {

QTextStream out(&file);

QString \_target, target;

if (ui.lineEdit\_net->text()!="")

{

out << "net: " << "\"" << ui.lineEdit\_net->text() << "\""<<"\n";

}

if (ui.lineEdit\_test\_iter->text()!="")

{

out << "test\_iter: " << ui.lineEdit\_test\_iter->text() << "\n";

}

if (ui.lineEdit\_test\_interval->text() != "") {

out << "test\_interval: " << ui.lineEdit\_test\_interval->text() << "\n";

}

if (ui.lineEdit\_base\_lr->text() != "") {

out << "base\_lr: " << ui.lineEdit\_base\_lr->text() << "\n";

}

if (ui.lineEdit\_momentum->text() != "") {

out << "momentum: " << ui.lineEdit\_momentum->text() << "\n";

}

if (ui.lineEdit\_weight\_decay->text() != "") {

out << "weight\_decay: " << ui.lineEdit\_weight\_decay->text() << "\n";

}

QString \_temp;//lr\_policy

switch (ui.comboBox\_lr\_policy->currentIndex())

{

case 0:

\_temp = "fixed";

break;

case 1:

\_temp = "step";

break;

case 2:

\_temp = "exp";

break;

case 3:

\_temp = "inv";

break;

case 4:

\_temp = "multistep";

break;

case 5:

\_temp = "poly";

break;

case 6:

\_temp = "sigmoid";

break;

default:

break;

}

out << "lr\_policy: " << "\""<<\_temp << "\""<<"\n";

if (ui.lineEdit\_gamma->text() != "") {

out << "gamma: " << ui.lineEdit\_gamma->text() << "\n";

}

if (ui.lineEdit\_power->text() != "") {

out << "power: " << ui.lineEdit\_power->text() << "\n";

}

if (ui.lineEdit\_display->text() != "") {

out << "display: " << ui.lineEdit\_display->text() << "\n";

}

if (ui.lineEdit\_max\_iter->text() != "") {

out << "max\_iter: " << ui.lineEdit\_max\_iter->text() << "\n";

}

if (ui.lineEdit\_snapshot->text() != "") {

out << "snapshot: " << ui.lineEdit\_snapshot->text() << "\n";

}

if (ui.lineEdit\_snapshot\_prefix->text() != "") {

out << "snapshot\_prefix: " << "\"" << ui.lineEdit\_snapshot\_prefix->text() << "\"" << "\n";

}

if (ui.comboBox\_solver\_mode->currentIndex()==0)

{

out << "solver\_mode: " << "CPU" << "\n";

}

else

{

out << "solver\_mode: " << "GPU" << "\n";

}

}//end of if (file.open)

else

{

//文件打开失败

QMessageBox::information(this, QStringLiteral("训练"), QStringLiteral("文件打开失败!"));

}

}

//从求解器文件中读取参数写入界面的相应控件中

void TrainingSetter::ReadSolver()

{

QFile file(ui.lineEdit\_solverFilePath->text());

if (file.open(QFile::ReadOnly | QFile::Text)) {

QTextStream QTS(&file);

while (QTS.atEnd() == false)

{

QString \_temp = QTS.readLine();

if (!\_temp.startsWith("#"))//注释不处理

{

int sat = \_temp.indexOf(" ");

QString target = \_temp.mid(sat + 1, \_temp.length() - (1 + sat));

if (\_temp.startsWith("net"))

{

//处理引号

QString \_target = target.mid(1, target.length() - 2);

ui.lineEdit\_net->setText(\_target);

}

if (\_temp.startsWith("test\_iter"))

{

ui.lineEdit\_test\_iter->setText(target);

}

if (\_temp.startsWith("test\_interval"))

{

ui.lineEdit\_test\_interval->setText(target);

}

if (\_temp.startsWith("base\_lr"))

{

ui.lineEdit\_base\_lr->setText(target);

}

if (\_temp.startsWith("momentum"))

{

ui.lineEdit\_momentum->setText(target);

}

if (\_temp.startsWith("weight\_decay"))

{

ui.lineEdit\_weight\_decay->setText(target);

}

if (\_temp.startsWith("lr\_policy:"))//lr\_policy

{

QString \_target = target.mid(1, target.length() - 2);//处理掉引号

if (\_target == "fixed")

{

ui.comboBox\_lr\_policy->setCurrentIndex(0);

}

if (\_target == "step")

{

ui.comboBox\_lr\_policy->setCurrentIndex(1);

}

if (\_target == "exp")

{

ui.comboBox\_lr\_policy->setCurrentIndex(2);

}

if (\_target == "inv")

{

ui.comboBox\_lr\_policy->setCurrentIndex(3);

}

if (\_target == "multistep")

{

ui.comboBox\_lr\_policy->setCurrentIndex(4);

}

if (\_target == "poly")

{

ui.comboBox\_lr\_policy->setCurrentIndex(5);

}

if (\_target == "sigmoid")

{

ui.comboBox\_lr\_policy->setCurrentIndex(6);

}

}

if (\_temp.startsWith("gamma"))

{

ui.lineEdit\_gamma->setText(target);

}

if (\_temp.startsWith("power"))

{

ui.lineEdit\_power->setText(target);

}

if (\_temp.startsWith("display"))

{

ui.lineEdit\_display->setText(target);

}

if (\_temp.startsWith("max\_iter"))

{

ui.lineEdit\_max\_iter->setText(target);

}

if (\_temp.startsWith("snapshot:"))

{

ui.lineEdit\_snapshot->setText(target);

}

if (\_temp.startsWith("snapshot\_prefix"))

{

//处理引号

QString \_target = target.mid(1, target.length() - 2);

ui.lineEdit\_snapshot\_prefix->setText(\_target);

}

if (\_temp.startsWith("solver\_mode"))//CPU&GPU

{

if (target == "GPU")

{

ui.comboBox\_solver\_mode->setCurrentIndex(1);

}

else {

ui.comboBox\_solver\_mode->setCurrentIndex(0);

}

}

}

}//end of while (QTS.atEnd() == false)

}//end of if (file.open)

else

{

//文件打开失败

QMessageBox::information(this, QStringLiteral("训练"), QStringLiteral("文件打开失败!"));

}

}

//执行训练

void TrainingSetter::ExecuteTraining()

{

//校验参数

if (ui.lineEdit\_solverFilePath->text() != "") {

//构造命令

QString cmd = "";

cmd.append(CoreDir);

cmd.append("/caffe.exe"); cmd.append(" ");

cmd.append("train"); cmd.append(" ");

cmd.append("--solver"); cmd.append("=");

cmd.append(ui.lineEdit\_solverFilePath->text());

//启动额外线程

\_AdditionalProcess->start(cmd);

}

else

{

QMessageBox::information(this, QStringLiteral("训练"), QStringLiteral("参数有误!"));

return;

}

}

//学习率策略变更

void TrainingSetter::policyChange()

{

switch (ui.comboBox\_lr\_policy->currentIndex())

{

case 0://fix

ui.lineEdit\_gamma->setEnabled(false);

ui.lineEdit\_power->setEnabled(false);

ui.lineEdit\_stepsize->setEnabled(false);

ui.lineEdit\_stepvalue->setEnabled(false);

break;

case 1://step(base\_lr \* gamma ^ (floor(iter / stepsize)))

ui.lineEdit\_gamma->setEnabled(true);

ui.lineEdit\_power->setEnabled(false);

ui.lineEdit\_stepsize->setEnabled(true);

ui.lineEdit\_stepvalue->setEnabled(false);

break;

case 2://exp(base\_lr \* gamma ^ iter)

ui.lineEdit\_gamma->setEnabled(true);

ui.lineEdit\_power->setEnabled(false);

ui.lineEdit\_stepsize->setEnabled(false);

ui.lineEdit\_stepvalue->setEnabled(false);

break;

case 3://inv(base\_lr \* (1 + gamma \* iter) ^ (- power))

ui.lineEdit\_gamma->setEnabled(true);

ui.lineEdit\_power->setEnabled(true);

ui.lineEdit\_stepsize->setEnabled(false);

ui.lineEdit\_stepvalue->setEnabled(false);

break;

case 4://multistep(根据stepvalue更新的step方法)

ui.lineEdit\_gamma->setEnabled(true);

ui.lineEdit\_power->setEnabled(false);

ui.lineEdit\_stepsize->setEnabled(true);

ui.lineEdit\_stepvalue->setEnabled(true);

break;

case 5://poly(base\_lr (1 - iter/max\_iter) ^ (power))

ui.lineEdit\_gamma->setEnabled(false);

ui.lineEdit\_power->setEnabled(true);

ui.lineEdit\_stepsize->setEnabled(false);

ui.lineEdit\_stepvalue->setEnabled(false);

break;

case 6://sigmoid(base\_lr ( 1/(1 + exp(-gamma \* (iter - stepsize)))))

ui.lineEdit\_gamma->setEnabled(true);

ui.lineEdit\_power->setEnabled(false);

ui.lineEdit\_stepsize->setEnabled(true);

ui.lineEdit\_stepvalue->setEnabled(false);

break;

default:

break;

}

}