Guide de glMiSOM

Introduction

glMiSOM est un prototype de la gestion d'images qui permet à l'utilisateur de chercher des images similaires en représentant toutes les images sur une surface plane où les images plus proches sont plus similaires et vice versa. En même temps, glMiSOM est aussi utilisé comme l'outil de test pour les caractéristiques de bas niveau de la sémantique de l'image et les combinaisons des caractéristiques.

Pour utiliser bien ce logiciel, l'utilisateur doit savoir des connaissances sur les caractéristiques de bas niveau de la sémantique de l'image, le processus d'extraction des caractéristiques et l'algorithme Cartes auto-organisatrices (SOM - Self Organizing Map).

Nécessités

Ce logiciel exécute seulement sur le système d'exploitation Ubuntu avec la plateforme Qt 4.7.4 ou ultérieure et OpenCV 2.1. Le logiciel utilise encore la librairie QOpenGL de Qt. Donc, dans votre système, vous devez installer les paquets suivants, avec toutes les dépendances associées : xord-dev, freeglut3, freeglut3-dev.

Fonctionnalisés

1. Soutenir les types d'images suivants : jpg, jpeg, png, bmp, tiff, ppm, pgm
2. Gérer, charger, enregistrer les bases d'images en utilisant les fichiers metadata
3. Charger, enregistrer les cartes de SOM
4. Charger, enregistrer les codes de SOM
5. Soutenir les extracteurs des caractéristiques sous forme des modules d'extension. Actuellement, le logiciel a installé trois modules d'extension statiques : Color Histogram (l'histogramme de couleur), Hu Moments (les moments de Hu), Matrix Cooccurrence 2 (la texture) et un module d'extension dynamique : Matrix Cooccurrence (la texture).
6. Indexer et visualiser les images en utilisant l'algorithme Cartes auto-organisatrices (SOM - Self Organizing Map)
7. Ré-arranger la grille de Kohonen en utilisant l'algorithme Ré-arrangement (lire le rapport de stage pour savoir plus)
8. Représenter les images sous forme de la grille, soutenir de voir en plein écran et l'animation
9. Exploration simple : zoom, explorer en utilisant la souris ou quatre boutons flèche du clavier.
10. Voir les informations de la base d'images ainsi que chaque image et voir l'image de la taille originale.
11. Importer les nouvelles images à la base d'images
12. Capturer l'écran
13. Changer la couleur du fond des images

14. Chercher l'image basé sur le contenu
15. Classifier en utilisant la couleur

Fichiers metadata

Pour chaque base d'images, glMiSOM stocke ses informations sur des fichiers metadata, en généralement y compris :

1. *database.meta* : stocker les chemins de toutes les images dans la base d'images
2. *features.meta* : stocker toutes les caractéristiques que glMiSOM a extrait
3. *map.meta* : stocker des paramètres de l'algorithme Carte auto-organisatrices, la grille de Kohonen, le résultat de visualisation et celui de ré-arrangement de la dernière fois d'exécution.
4. Les fichiers meta des images <*nom de l'image.meta*> : stocker les vecteurs de caractéristiques extraits de cette image.

Dossiers nécessaires

Dans le dossiers de glMiSOM, on a aussi deux dossiers nécessaires pour exécuter du logiciel :

1. *images* : ce dossier stocke une image *transparent.png*. Cette image sert à dessiner la quantité d'images dans une casse de la grille de SOM.
2. *plugins* : stocker quatre modules d'exécution de quatre extracteurs des caractéristiques.

Exemple d'un processus de l'utilisation de glMiSOM

Voir le vidéo *glMiSOM's demo.m4v*.

Notes : Pour utiliser les bases d'images associées, vous ouvririez le fichier database.meta de chaque base d'images et changez les chemins d'images pour s'adapter avec le endroit où vous mettez les images.

Classes du glMiSOM

J'ai organisé le glMiSOM en cinq modules :

1. **Core** : Ce sont les classes basiques qui servent à stocker la base d'image et la grille de SOM. Ce module comprend les classes suivantes : *baseimage*, *image*, *feature*, *gridsom*, *rowsom*, *cellsom*.
2. **Exception** : Ce module contient les définitions des Exceptions du glMiSOM. Ces exceptions hérite la classe *exception* de la librairie C++ utilisées par les commandes try catch.
3. **Scene** : comprendre deux classes importantes *glsomscene* et *glimage* qui dessine la grille d'images en utilisant la librairie QOpenGL. La classe *animator* est responsable de l'animation des images.
4. **SOM** : implémenter l'algorithme SOM et l'algorithme Ré-arrangement, y compris les classes *mapsom*, *entrysom*, *featureschosenlist*, *stableinfo*, *winnerinfo*.

5. **UI** : contenir les classes d'interface du logiciel : *capturescreendialog*, *chosendescriptors*, *codedialog*, *extrairefeaturesdialog*, *indexationdialog*, *pluginsdialog*, *viewcodesdialog*, *viewimagedialog*, *viewmetadatadialog*, *weighdelegate*.

De plus, j'ai une classe *util* qui comprend des fonctions utiles et statiques.

Pour soutenir les modules d'extension, glMiSOM définir deux classes *extractorinterface* et *extractorwidgetinterface*. Tous les modules d'extension doivent hériter ces deux classes.

Codage d'un nouveau module d'extension de l'extracteur des caractéristiques

Pour coder bien des modules d'extension, il faut savoir des connaissances basiques sur le codage d'un module d'extension dans la plateforme Qt (lire le guide de Qt pour plus information).

Tout d'abord, dans le fichier .pro du projet, vous ajoutez le chemin de la source code de glMiSOM dans *INCLUDEPATH*, celui de cinq fichiers *extractorinterface.h*, *extractorwidgetinterface.h*, *image.h*, *feature.h*, *util.h* dans *HEADERS* et celui de trois fichiers *image.cpp*, *feature.cpp* et *util.cpp* dans *SOURCES*.

Exemple :

```
INCLUDEPATH  += ../../glMiSOM/
HEADERS      += ../../glMiSOM/Extractor/extractorinterface.h \
              ../../glMiSOM/Extractor/extractorwidgetinterface.h \
              ../../glMiSOM/Core/image.h \
              ../../glMiSOM/Core/feature.h \
              ../../glMiSOM/util.h
SOURCES       = ../../glMiSOM/Core/image.cpp \
              ../../glMiSOM/Core/feature.cpp \
              ../../glMiSOM/util.cpp
```

Ensuite, vous créer deux classes qui hériteront deux classes *ExtractorInterface* et *ExtractorWidgetInterface*.

ExtractorWidgetInterface définit l'interface sur l'écran où l'utilisateur peut choisir des paramètres de l'extracteur. C'est une classe héritant la classes QGroupBox. Sa définition est suivante :

```
#include <QGroupBox>

class ExtractorWidgetInterface : public QGroupBox
{
   Q_OBJECT
public:
   ExtractorWidgetInterface(QWidget *parent = 0) : QGroupBox(parent) {}
   virtual ~ExtractorWidgetInterface() {}

   virtual void resetDefault() = 0;
   virtual QStringList getParams() = 0;
```

```
    virtual bool isHaveError() = 0;

    virtual bool isGlobalChecked() = 0;
    virtual bool isLocalChecked() = 0;

private slots:
    virtual void computeQuantity() = 0;
};
```

Dans la fonction de construction, vous créez des widgets nécessaires pour l'interface de l'extracteur. En même temps, vous implémentez six fonctions suivantes :

1. *resetDefault* : restaurer les valeurs dans les widgets en utilisant les valeurs par défaut
2. *getParams* : retourner une liste des paramètres. Généralement, cette liste possède d'un ou deux éléments : un élément pour le choix global et un élément pour le choix local. Chaque élément dans la liste est une chaîne de textes où chaque paramètre sont distingués par un caractère spécial (comme la virgule dans l'exemple). Le développer a tout le droit de décider la façon de définir les paramètres.
3. *isHaveError* : vérifier l'erreur des paramètres appuyés par l'utilisateur
4. *isGlobalChecked* : vérifier le choix sur l'extracteur global
5. *isLocalChecked* : vérifier le choix sur l'extracteur local
6. *computeQuantity* : calculer le nombre des éléments du vecteur de caractéristiques, sert seulement à représenter sur l'interface elle-même.

Exemple :

```
//
//
// colorhistogramwidget.cpp
//
// Implementation de la classe colorhistogramwidget.h
//
// LE Viet Man
// 03/08/2011
//
//

#include "colorhistogramwidget.h"

ColorHistogramWidget::ColorHistogramWidget(QWidget *parent)
{
    Q_UNUSED(parent)

    setTitle(QString("Color Histogram"));
    setFlat(false);
    setCheckable(true);
    setChecked(false);

    createUI(); // créer les widgets pour l'interface de paramètres de l'extracteur

    computeQuantity();
}
```

```
void ColorHistogramWidget::resetDefault()
{
    // restaurer les valeurs dans les widgets
    setChecked(false);
    chkGlobal->setChecked(false);
    chkLocal->setChecked(true);
    spiBinNumber->setValue(5);

    // ré-calculer le nombre des éléments du vecteur de caractéristiques
    computeQuantity();
}

QStringList ColorHistogramWidget::getParams()
{
    QStringList list;
    if (chkGlobal->isChecked())
    {
        list << getParam(!chkGlobal->isChecked(), spiBinNumber->value());
    }
    if (chkLocal->isChecked())
    {
        list << getParam(chkLocal->isChecked(), spiBinNumber->value());
    }
    return list;
}

bool ColorHistogramWidget::isHaveError()
{
    if ((isChecked()) && (!chkGlobal->isChecked()) && (!chkLocal->isChecked()))
    {
        QMessageBox::information(this, tr("Error"), tr("You should choose local or global in Color
Histogram."));

        return true;
    }
    return false;
}

bool ColorHistogramWidget::isGlobalChecked()
{
    return chkGlobal->isChecked();
}

bool ColorHistogramWidget::isLocalChecked()
{
    return chkLocal->isChecked();
}

void ColorHistogramWidget::computeQuantity()
{
    int global = 3 * spiBinNumber->value();
    int local = 5 * global;

    QString quantity = tr("Quantity of features : ");
    if (chkGlobal->isChecked())
    {
        quantity += tr("Global - %1     ").arg(global);
    }
    if (chkLocal->isChecked())
```

```
    {
        quantity += tr("Local - %1").arg(local);
    }

    lblQuantity->setText(quantity);
}

void ColorHistogramWidget::createUI()
{
    QVBoxLayout *verticalLayout = new QVBoxLayout(this);

    // quantity of features
    lblQuantity = new QLabel(this);
    QFont font;
    font.setItalic(true);
    lblQuantity->setText(QString("Quantity of features :"));
    lblQuantity->setFont(font);

    verticalLayout->addWidget(lblQuantity);

    // line 1 : Global or Local
    QHBoxLayout *horizontalLayout_1 = new QHBoxLayout();

    chkGlobal = new QCheckBox(this);
    chkGlobal->setText(QString("Global"));
    connect(chkGlobal, SIGNAL(stateChanged(int)), this, SLOT(computeQuantity()));

    horizontalLayout_1->addWidget(chkGlobal);

    chkLocal = new QCheckBox(this);
    chkLocal->setText(QString("Local"));
    chkLocal->setChecked(true);
    connect(chkLocal, SIGNAL(stateChanged(int)), this, SLOT(computeQuantity()));

    horizontalLayout_1->addWidget(chkLocal);

    QSpacerItem *horizontalSpacer_1 = new QSpacerItem(40, 20, QSizePolicy::Expanding,
QSizePolicy::Minimum);

    horizontalLayout_1->addItem(horizontalSpacer_1);

    verticalLayout->addLayout(horizontalLayout_1);

    // line 2 : Bin number
    QHBoxLayout *horizontalLayout_2 = new QHBoxLayout();
    lblBinNumber = new QLabel(this);
    lblBinNumber->setText(QString("Bin number :"));

    horizontalLayout_2->addWidget(lblBinNumber);

    spiBinNumber = new QSpinBox(this);
    spiBinNumber->setMinimum(1);
    spiBinNumber->setMaximum(256);
    spiBinNumber->setValue(5);
    connect(spiBinNumber, SIGNAL(valueChanged(int)), this, SLOT(computeQuantity()));

    horizontalLayout_2->addWidget(spiBinNumber);
```

```
    QSpacerItem *horizontalSpacer_2 = new QSpacerItem(40, 20, QSizePolicy::Expanding,
QSizePolicy::Minimum);

    horizontalLayout_2->addItem(horizontalSpacer_2);

    verticalLayout->addLayout(horizontalLayout_2);
}

QString ColorHistogramWidget::getParam(bool local, int numBin)
{
    QString param = QString("%1,%2").arg(local?1:0).arg(numBin);
    return param;
}
```

ExtractorInterface définit les fonctions nécessaires pour extraire des caractéristiques, calculer la distance entre deux vecteurs de caractéristiques et faire l'adaptation dans l'algorithme SOM. Sa définition est suivante :

```
//
//
// extractorinterface.h
//
// Un interface pour tous les extracteurs des descripteurs
//
// LE Viet Man
// 02/09/2011
//
//

#ifndef EXTRACTORINTERFACE_H
#define EXTRACTORINTERFACE_H

#include <QtPlugin>
#include "Core/image.h"
#include "Core/feature.h"
#include "extractorwidgetinterface.h"

class ExtractorInterface
{
public:
    virtual ~ExtractorInterface() {}

    virtual QString about() = 0;
    virtual QString getPluginName() = 0;

    virtual QString getName() = 0;
    virtual QString getFullName() = 0;
    virtual ExtractorWidgetInterface *getUi() = 0;

    virtual ExtractorWidgetInterface *createUi(QWidget *parent = 0) = 0;

    virtual Feature *extract(Image *image, QString params) = 0;
    virtual bool isMyDescriptor(QString codedDescriptor) = 0;
    virtual bool isLocal(QString params) = 0;

    virtual QString getCodedDescriptor(QString params) = 0;
    virtual QString getFullDescriptor(QString codedDes) = 0;
```

```
    virtual double calDistanceEuc(Feature *fea1, Feature *fea2) = 0;
    virtual int computeSizeTexture(QString codedDes) = 0;

    virtual void adapt(Feature *fea, Feature *imgFea, double alpha) = 0;
};

Q_DECLARE_INTERFACE(ExtractorInterface,
                    "com.manleviet.glMiSOM.ExtractorInterface/1.1")

#endif // EXTRACTORINTERFACE_H
```

Alors, un module d'extension doit implémenter les fonctions suivantes :

1. *about* : retourner la chaîne de textes qui introduit au module d'extension. Cette fonction est utilisée par la fenêtre de gestion des modules d'extension.
2. *getPluginName* : retourner le nom du fichier de module d'extension
3. *getName* : retourner le nom abrégé de l'extracteur
4. *getFullName* : retourner le nom complet de l'extracteur
5. *getUi* : retourner un pointeur qui pointe à une instance de la classe *ExtractorWidgetInterface*
6. *createUi* : créer l'interface de l'extracteur et retourner un pointeur
7. *extract* : une fonction importante, sert à extraire des caractéristiques. Elle reçoit une image que l'on veut extraire et une chaîne des paramètres (c'est la chaîne retournée par la fonction *getParams* de la classe qui hérite la classe *ExtractorWidgetInterface* ci-dessus)
8. *isMyDescriptor* : vérifier si une chaîne de description est créée par cet extracteur.
9. *isLocal* : vérifier si une chaîne des paramètres est local.
10. *getCodedDescriptor* : créer une chaîne de description codée à partir des paramètres, sert à stocker
11. *getFullDescriptor* : créer une chaîne de description à partir d'une chaîne de description codée, sert à présenter sur l'interface de glMiSOM.
12. *calDistanceEuc* : calculer la distance euclidienne entre deux vecteurs de caractéristiques
13. *computeSizeTexture* : calculer le nombre d'éléments d'un vecteur de caractéristiques
14. *adapt* : faire l'adaptation, utilisée par l'algorithme SOM

Exemple :

```
//
//
// colorhistogramextractor.cpp
//
// Implementation de la classe colorhistogramextractor.h
//
// LE Viet Man
// 12/06/2011
//
//

#include "colorhistogramextractor.h"
```

```
ColorHistogramExtractor::ColorHistogramExtractor()
    : ExtractorInterface()
{
    name = QString("COLORHISTOGRAM");
    fullname = QString("Color Histogram");
    ui = 0;
}

QString ColorHistogramExtractor::about()
{
    return QString("<p><b>Plugin ColorHistogramExtractor version 1.1</b></p>"
                   "<p>Static plugin"
                   "<br>Build date: %1"
                   "<br>This plugins uses Qt 4.7, OpenCV 2.1"
                   "<br>Author: LE Viet Man"
                   "<br>Email: <a href=\"mailto:manleviet@gmail.com>\">manleviet@gmail.com</a>"
                   "<br>09/2011"
                   "</p>").arg(__TIMESTAMP__);
}

QString ColorHistogramExtractor::getPluginName()
{
    return QString("libcolorhistogramextractor.a");
}

ExtractorWidgetInterface *ColorHistogramExtractor::createUi(QWidget *parent)
{
    ColorHistogramWidget *widget = new ColorHistogramWidget(parent);
    ui = widget;
    return widget;
}

Feature *ColorHistogramExtractor::extract(Image *image, QString params)
{
    // parser des parametres
    bool local;
    int numBins;
    parseParams(params, local, numBins);

    IplImage *img = cvLoadImage(image->getPath().toStdString().c_str());

    if (!img)
    {
        throw std::bad_exception();
        return NULL;
    }

    QList<CvRect> parts;
    // a partir de la valeur local pour prendre des partis de l'image
    Util::splitImageToParts(local, img, parts);

    int numFeaturesOfAPart = computeSizeTextureOfAPart(numBins);
    int numFeatures = computeSizeTexture(local, numBins);
    double *histograms = (double*)malloc(sizeof(double) * numFeatures);
    memset(histograms, 0, sizeof(double) * numFeatures);
    CvRect part;
    for (int i = 0; i < parts.size(); i++)
    {
        part = parts[i];
```

```
            cvSetImageROI(img, part);

            calculFeature(img, i, histograms, numBins, numFeaturesOfAPart);
        }
        cvReleaseImage(&img);

        // creer un Feature
        Feature *feature = new Feature(getCodedDescriptor(local, numBins), numFeatures, histograms);

        free(histograms);
        return feature;
    }

    bool ColorHistogramExtractor::isMyDescriptor(QString codedDes)
    {
        QStringList fields = codedDes.split(",");
        if (fields[0] == name)
            return true;
        return false;
    }

    bool ColorHistogramExtractor::isLocal(QString params)
    {
        // parser des parametres
        bool local;
        int numBins;
        parseParams(params, local, numBins);

        return local;
    }

    QString ColorHistogramExtractor::getCodedDescriptor(QString params)
    {
        // parser des parametres
        bool local;
        int numBins;
        parseParams(params, local, numBins);

        return getCodedDescriptor(local, numBins);
    }

    QString ColorHistogramExtractor::getFullDescriptor(QString codedDes)
    {
        QStringList fields = codedDes.split(",");

        QString des = fullname;

        if (fields[1].toInt() == 1)
        {
            des += QString(" - Local - ");
        }
        else
        {
            des += QString(" - Global - ");
        }

        des += QString("Bin number : %1").arg(fields[2].toInt());
```

```
    return des;
}

double ColorHistogramExtractor::calDistanceEuc(Feature *fea1, Feature *fea2)
{
    double diff, difference;
    difference = 0;

    int size = fea1->getSize();
    for (int i = 0; i < size; i++)
    {
        diff = fea1->getValue(i) - fea2->getValue(i);
        difference += diff * diff;
    }
    difference /= size;

    return difference;
}

int ColorHistogramExtractor::computeSizeTexture(QString codedDes)
{
    if (!isMyDescriptor(codedDes)) return -1;

    QString name;
    int numBins;
    bool local;
    parseParams(codedDes, name, local, numBins);

    return computeSizeTexture(local, numBins);
}

void ColorHistogramExtractor::adapt(Feature *fea, Feature *imgFea, double alpha)
{
    double value;
    for (int i = 0; i < fea->getSize(); i++)
    {
        value = fea->getValue(i);
        value += alpha * (imgFea->getValue(i) - value);
        fea->setValue(i, value);
    }
}

// PRIVATE
QString ColorHistogramExtractor::getCodedDescriptor(bool local, int numBins)
{
    QString coded = QString("%1,%2,%3").arg(name).arg(local?1:0).arg(numBins);

    return coded;
}

int ColorHistogramExtractor::computeSizeTextureOfAPart(int numBins)
{
    return numBins * 3; // trois couleurs RGB
}

int ColorHistogramExtractor::computeSizeTexture(bool local, int numBins)
{
    int numFeaturesOfAPart = computeSizeTextureOfAPart(numBins);
```

```
    if (local) return numFeaturesOfAPart * 5;
    return numFeaturesOfAPart;
}

//
// Calculer l'histogramme
// @param :
//      IplImage *img : une image
//      int numBins : niveau de gris
//
void ColorHistogramExtractor::calculFeature(IplImage *img,
                                            int indexPart,
                                            double *histograms,
                                            int numBins,
                                            int numFeatureOfAPart)
{
    if ((histograms == NULL) || (img == NULL))
    {
        throw std::bad_exception();
    }

    IplImage *part = cvCreateImage(cvGetSize(img), img->depth, img->nChannels);
    cvCopy(img, part);

    int index = indexPart * numFeatureOfAPart;
    int divise = 256 / numBins;
    int b, g, r;
    CvScalar s;
    double totalPixels = 0;
    for (int i = 0; i < part->height; i++)
    {
        for (int j = 0; j < part->width; j++)
        {
            s = cvGet2D(part, i, j);
            totalPixels++;
            // reduire le niveau de gris
            b = (int)s.val[0] / divise;
            g = ((int)s.val[1] / divise) + numBins;
            r = ((int)s.val[2] / divise) + (2 * numBins);

            if (b == numBins) b--;
            if (g == (2 * numBins)) g--;
            if (r == (3 * numBins)) r--;

            b += index;
            g += index;
            r += index;

            histograms[b] += 1;
            histograms[g] += 1;
            histograms[r] += 1;
        }
    }
    cvReleaseImage(&part);

    // normaliser
    for (int i = 0; i < numFeatureOfAPart; i++)
    {
        histograms[i + index] /= totalPixels;
```

```
    }
}

void ColorHistogramExtractor::parseParams(QString params, bool &local, int &numBins)
{
    QStringList fields = params.split(",");

    local = (fields[0].toInt() == 1)?true:false;
    numBins = fields[1].toInt();
}

void ColorHistogramExtractor::parseParams(QString codedDes, QString &name, bool &local, int
&numBins)
{
    QStringList fields = codedDes.split(",");

    name = fields[0];
    local = (fields[1].toInt() == 1)?true:false;
    numBins = fields[2].toInt();
}

Q_EXPORT_PLUGIN2(colorhistogramextractor, ColorHistogramExtractor)
```

Vous pouvez voir la source code de quatre extracteurs dans le dossier Plugin pour comprendre bien.