Sujet de Travaux Pratiques (3h)
SI343

Mise en œuvre d'algorithmes à base de graph-cuts pour le traitement de l'image

Objectifs de la séance :

Le but de cette séance est de programmer une segmentation binaire dans un cadre markovien avec deux types d'optimisation : une optimisation par recuit simulé, et une optimisation par graph-cut. Ensuite un algorithme de restauration d'une image bruitée sera programmé.

La solution retenue pour la mise en œuvre est l'écriture des algorithmes en C/C++ sous forme de *MEX-functions* pouvant être appelées par un script Matlab.

Le compte-rendu est à rendre pour lundi 22 avril au plus tard, à remettre dans le casier de Florence Tupin (pièce C08). Il peut être rédigé en binôme.

N.B : Le programme `my_tp.m` *vous donne l'ossature de l'ensemble de la séance de TP. Vous ne pouvez pas le lancer directement, mais vous pouvez vous en inspirer pour réaliser le TP.*

1 Classification binaire d'une image bruitée

Vous disposez d'une image sans bruit `IoriginaleBW.png` (image binaire des deux classes) et de sa version observée bruitée `Ibruitee.png`. L'objectif est de réaliser une classification en deux classes de cette image bruitée.

1.1 Analyse des distributions des 2 classes de l'image

- Quelles sont les distributions des deux classes de l'image ($P(Y_s|X_s = 0)$ (classe noire) et $P(Y_s|X_s = 0)$ (classe blanche)) ? Donnez les moyennes et variances des deux classes. *On pourra utiliser l'image sans bruit pour analyser les distributions des niveaux de gris. La commande* `v=I(S==1)` *; met dans le vecteur v toutes les valeurs de I pour lesquelles S vaut 1.*
- Réalisez un seuillage de cette image. Donnez la valeur du seuil optimal au sens du maximum de vraisemblance ponctuel (c'est à dire en maximisant en chaque point $P(Y_s|X_s)$) (par calcul et graphiquement sur la figure des histogrammes normalisés). Pour simplifier, on supposera dans toute la suite que les variances des deux classes sont égales.
- Donnez la valeur du seuil optimal au sens du maximum a posteriori ponctuel (classe qui maximise $P(Y_s|X_s)P(X_s)$) en utilisant graphiquement les histogrammes non normalisés qui tiennent compte de la proportion des classes. Quel seuil vous satisfait le plus ?

1.2 Modèle d'Ising pour la régularisation

Pour améliorer les résultats il est nécessaire d'introduire une régularisation (modèle a priori global).

- Ecrire l'énergie d'attache aux données pour les distributions précédentes.
- Ecrire le potentiel des cliques d'ordre deux pour le modèle d'Ising en fonction de $\delta_{x_p=x_q}$ (qui vaut 0 quand $x_p = x_q$ et 1 sinon) où x_p et x_q sont les classes des pixels p et q et du paramètre de régularisation β.

– Ecrire l'énergie globale et l'énergie conditionnelle locale.
– Quelle est la solution quand β vaut 0 ? ou $+\infty$?
– Comment varie la solution quand β augmente ?

1.3 Optimisation par recuit simulé

On va réaliser l'optimisation de l'énergie globale précédemment définie. *A partir du squelette de programme Matlab fourni* `my_tp.m`*, vous allez programmer le recuit simulé.*

– Complétez la fonction `echantillonneur_Gibbs_a_completer.m` pour les distributions que vous avez trouvées. Faites tourner le recuit simulé. On prendra une valeur de β de l'ordre de 1000 pour un potentiel d'attache aux données non normalisé (i.e $(a-b)^2$).
– Vérifiez les résultats donnés précédemment pour $\beta = 0$ et β très grand.

1.4 Optimisation par recherche de coupure minimale

On va construire un graphe pour chercher la solution minimisant globalement l'énergie par calcul de coupure minimale. A partir du squelette de programme C++ fourni, étudiez et modifiez la fonction `min_exacte_binaire_graph_cut_a_completer.m` (voir Annexes sur les MEX-functions et sur la bibliothèque graph-cut de Kolmogorov et Boykov).

Pour compiler les fonctions, il faut utiliser la commande `mex`*. Dans les salles sous Windows, si le compilateur n'est pas installé, tapez la commande* `mex -setup` *et choisissez Microsoft Visual C++.*

– Dessinez le graphe qui est construit pour la recherche de la coupe de capacité minimale avec seulement deux pixels voisins.
– Executez l'algorithme de coupure minimale et visualisez le résultat. Comparez les résultats obtenus pour différentes valeurs de β avec ceux obtenus pour le recuit simulé.

2 Débruitage d'une image à niveaux de gris

Dans cette seconde partie du TP, nous nous intéressons à l'utilisation des méthodes markoviennes pour *débruiter* des images avec différents potentiels de régularisation.

Nous cherchons à débruiter les images `Ibruitee.png` et `Ibruitee2.png` qui correspondent à la même scène perturbée par deux bruits de nature différente.

Nous allons compléter la fonction `min_approchee_a_expansion_graph_cut_a_completer` qui implémente l'algorithme des α-expansions de Boykov selon la technique de Kolmogorov.

– Quelles sont les expressions respectives des potentiels d'attache aux données dans le cas d'un bruit suivant une distribution gaussienne (équation 1) et d'une distribution de Rayleigh (équation 2) ?

$$p(y_p|x_p) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left[-\frac{(y_p - x_p)^2}{2\sigma^2}\right], \tag{1}$$

$$p(y_p|x_p) = 2\frac{y_p}{x_p^2} \exp\left[-\frac{y_p^2}{x_p^2}\right]. \tag{2}$$

– Etudiez le programme fourni et expliquez à quoi correspondent la source et le puits dans ce cas (dans cette construction, le pixel prend le label de l'arc qui n'est pas coupé).

- Nous allons comparer trois modèles *a priori* : le modèle de Potts $\delta(x_p - x_q)$, la variation totale (discrète) $|x_p - x_q|$, et le modèle gaussien $(x_p - x_q)^2$.
  S'agit-il de métriques ou de semi-métriques ? Complétez les fonctions `attache_aux_donnees` et `regularisation` pour chacun des types de bruit et pour ces trois régularisations.
- Quel modèle de régularisation vous semble le plus adapté ? Commentez les résultats que vous obtenez dans chacun des trois cas.

Annexes

A. Utilisation d'une MEX-function sous Matlab

Matlab permet l'écriture d'une nouvelle fonction soit avec le langage interprété (.m), soit avec un langage compilé — Fortran ou C/C++ — (MEX function). L'utilisation d'un langage compilé est motivée par deux raisons principales :

- l'obtention d'un code *plus rapide*, notamment lorsque l'écriture en langage interprété Matlab conduirait à de trop nombreuses boucles explicites, de trop nombreux tests,
- l'utilisation d'une bibliothèque existante écrite dans un langage compilé.

Dans ce TP, les deux raisons sont valables : nous allons utiliser une bibliothèque existante écrite en C++ pour la manipulation des graphes. Comme il faudra manipuler de nombreuses fois ces graphes, nous avons intérêt à écrire le plus possible de code en C++.

Un premier exemple

L'écriture d'une MEX function (c'est à dire d'une fonction en C/C++ utilisable depuis Matlab) nécessite, en plus du code propre à la fonction, de l'interfacer avec Matlab. Voyons sur un exemple simple en quoi cela consiste.

La fonction C extrêmement simple ci-dessous multiplie par deux un vecteur (ou une matrice) de `n` éléments :

```
1 void timestwo(double* in, double* out, unsigned long n)
2 {
3   unsigned long i;
4
5   for(i=0; i<n; i++)
6     out[i] = 2.0*in[i];
7 }
```

Pour pouvoir appliquer cette fonction C à des vecteurs/matrices depuis Matlab, il faut enrober la fonction dans des portions de code d'interfaçage. Le listing 1 présente le code complet pour cet exemple.

Les éléments constituant une MEX-function

L'écriture d'une MEX-function consiste en 6 étapes que nous allons décrire à partir de l'exemple du listing 1.

Fichier à inclure : Il faut inclure le fichier `mex.h` qui contient les prototypes des différentes fonctions d'interfaçage utilisées. (→ l. 1)

Fonction principale : La fonction principale, qui sera appelée par Matlab pour exécuter la MEX-function, s'appelle `mexFunction`. Les paramètres d'entrée et de sortie de la MEX-function sont passés sous forme de deux tableaux de matrices Matlab, respectivement `prhs` et `plhs` ('rhs' pour "right hand side", 'lhs' pour "left hand side"). (→ l. 12-13)

Contrôle des arguments : La première chose à faire dans la fonction principale `mexFunction` est de vérifier le nombre et le type des arguments d'entrée reçus. (→ l. 18-30)

Allocation des matrices de sortie : La fonction `mxCreateDoubleMatrix` permet l'allocation d'une nouvelle matrice pour un paramètre de sortie. (→ l. 32-33)

Récupération des adresses des données : Un pointeur de type `double *` peut être récupéré pour chacun des paramètres d'entrée et de sortie à l'aide de la fonction `mxGetPr`. (→ l. 35-37)

Appel de la fonction de traitement : La fonction de traitement à proprement parler peut ensuite être appelée. (→ l. 39-40)

Compilation de la fonction : la commande Matlab `mex twotimes.cpp` permet de compiler la mex-function à condition qu'un compilateur C++ soit installé sur la machine.

Appel de la fonction sous Matlab : la commande Matlab `I2=timestwo(double(I))` permet d'appliquer le traitement à l'image I.

Listing 1: Un exemple de MEX function

```
#include "mex.h"

void timestwo(double* in, double* out, unsigned long n)
{
  unsigned long i;

  for(i=0; i<n; i++)
    out[i] = 2.0*in[i];
}


void mexFunction(int nlhs, mxArray *plhs[], int nrhs,
                 const mxArray *prhs[])
{
  double *x, *y;
  unsigned long mrows, ncols;

  /* Check for proper number of arguments. */
  if (nrhs != 1) {
    mexErrMsgTxt("One input required.");
  } else if (nlhs != 1) {
    mexErrMsgTxt("One output required.");
  }

  /* The input must be a noncomplex double matrix.*/
  mrows = mxGetM(prhs[0]);
  ncols = mxGetN(prhs[0]);
  if (!mxIsDouble(prhs[0]) || mxIsComplex(prhs[0])) {
    mexErrMsgTxt("Input must be a noncomplex double matrix.");
  }

  /* Create matrix for the return argument. */
  plhs[0] = mxCreateDoubleMatrix(mrows,ncols,mxREAL);

  /* Assign pointers to each input and output. */
  x = mxGetPr(prhs[0]);
  y = mxGetPr(plhs[0]);

  /* Call the timestwo subroutine. */
  timestwo(x,y,mrows*ncols);
}
```

B. Utilisation de la bibliothèque "graph-cuts" de Kolmogorov et Boykov

Vladimir Kolmogorov et Yuri Boykov ont proposé un algorithme efficace pour le calcul de coupes minimales dans le cas des graphes construits pour résoudre des problèmes de type "traitement de l'image". Ils distribuent une version de l'algorithme sous forme d'une petite bibliothèque écrite en C++ (licence GPL). Cette bibliothèque consiste en une classe `Graph` permettant de construire un graphe, puis de calculer une coupe minimale et enfin de déterminer si un nœud donné du graphe se trouve dans la partition S contenant la source, ou dans la partition P contenant le puits.

Principe d'utilisation de la classe `Graph`

L'interface de la classe `Graph` est présentée dans le listing 2. Les opérations de base dont nous aurons besoin sont :

Ajout d'un nœud : obtenu par la méthode `add_node()` qui nous renvoie l'adresse du nœud créé (à conserver quelque part pour identifier ce nœud).

Initialisation / Modification des arcs terminaux : obtenus par les méthodes `set_tweights` et `add_tweights` qui permettent respectivement de fixer ou d'ajouter une capacité donnée aux arcs $\mathscr{S} \rightarrow i$ et $i \rightarrow \mathscr{P}$ reliant le nœud i aux nœuds terminaux.

Ajout d'un arc entre deux nœuds : un arc entre deux nœuds i et j non terminaux est créé par la méthode `add_edge` qui prend en paramètre l'adresse des nœuds i et j et la capacité des arcs $i \rightarrow j$ et $j \rightarrow i$.

Calcul du flot maximum : en appelant la méthode `maxflow`.

Identification de la partition : la partition (S, P) engendrée par la coupe minimale est identifiée par la méthode `what_segment`. Appelée avec un nœud i donné en paramètre, la méthode `what_segment` renvoie le type de nœud terminal `SOURCE` ou `SINK` auquel le nœud i est rattaché (i.e. à quel ensemble de la partition il appartient).

Listing 2: Méthodes publiques de la classe Graph

```
class Graph
{
  public:
  typedef enum {
      SOURCE  = 0,
      SINK  = 1
  } termtype;

  typedef double captype;
  typedef double flowtype;
  typedef void * node_id;

  // Constructor
  Graph(void (*err_function)(char *) = NULL);
  Graph(long nb_nodes, long nb_arcs,
      void (*err_function)(char *) = NULL);

  // Destructor
  ~Graph();

  // Adds a node to the graph
  node_id add_node();

  // Adds an edge to the graph
  void add_edge(node_id from, node_id to, captype cap,
      captype rev_cap);

  // Sets the weights of the edges 'SOURCE->i' and 'i->SINK'
  void set_tweights(node_id i, captype cap_source,
      captype cap_sink);

  // Adds new edges 'SOURCE->i' and 'i->SINK'
  void add_tweights(node_id i, captype cap_source,
      captype cap_sink);

  // This function returns to which segment the node i belongs
  termtype what_segment(node_id i);

  // Computes the maxflow. Can be called only once.
  flowtype maxflow();

  // Reset the graph but keep its structure
  void reset_and_keep_graph_structure();

  // Reset the graph
  void reset();
}
```