INF442-5 Détection par “boosting”

Ruoqi HE, Chia-Man HUNG

Introduction

Dans ce projet, nous avons réussi à implémenter l'algorithme proposé par le sujet et à construire un classifieur de détection des visages de haute performance. Dans la suite, nous expliquerons comment est réalisée l'implémentation. Pour cela, nous détaillerons l'architecture générale des classes, la parallélisation des programmes et leur complexité. Nous tenterons de répondre aux questions posées par l'énoncé.

Vocabulaires anglais

Classifier → Classifieur faible
ResultClassifier → Classifieur final
feature → caratéristique

L'architecture générale

La réalisation est divisée en plusieurs classes.
La classe *Pic* permet de stocker les informations nécessaires d'une image ou d'une image intégrale.
La classe *Classifier* permet de stocker un classifieur faible et évaluer la caractéristique d'une image intégrale donnée.
La classe *ResultClassifier* permet de stocker un classifeur final, le sauvegarder dans un fichier, lire un classifieur final d'un fichier et évaluer une image.
La classe *FaceDetectingMPI* contient tous les algorithmes principaux de la détection de visage, à savoir les méthodes pour lire des images d'un répertoire donnée, calculer l'image intégrale à partir d'une image, générer un tableau de classifieurs faibles, calculer des caractéristiques, entraîner les classifeurs faibles et générer un classifeur final par “boosting”.
La classe *MainTraining* implémente les méthodes de *FaceDetectingMPI* et son exécution nous donne les résultats.
La classe *DetectionEvaluation* nous donne les taux d'erreur en fonction de theta, ce qui nous permet de tracer une courbe.
Chaque méthode qui utilise MPI admet également une version séquentielle.

Q1.1 Calcul de l'image intégrale

```
Pic* integral(Pic* pic)
```

L'algorithme parcout l'image une seule fois en utilisant la relation de récurrence suivant :
I(x,y)= I(x-1,y)+I(x,y-1)-I(x-1,y-1)+I(x-1,y-1))
La complexité est donc O(s) où s est le nombre de pixels de l'image.

Q1.2 Calcul des caractéristiques

D'abord nous avons besoin de générer un tableau contenant tous les classifieurs faibles.

```
vector<Classifier>* generateClassifierTable(int width, int height)
```

Les images sont de taille 112 x 92 pixels. Il y a quatre types de classifeur faible.
La taille mininum de côté est 4 pixels et on prend un incrément de position de 4 pixels. Le problème est donc équivalent à une image de 28 x 23 pixels avec un incrément d'un pixel.
type left-right (gauche-droite) :

$$(1 + 3 + \cdots + 27)(1 + 2 + \cdots + 23) = 54096$$

type up-down (haut-bas) :

$$(1 + 2 + \cdots + 28)(2 + 4 + \cdots + 22) = 53592$$

type LMR :

$$(2+5+\cdots+26)(1+2+\cdots+23)=34776$$

type cross :

$$(1+3+\cdots+27)(2+4+\cdots+22)=25872$$

En total, il y a 168336 différents classifieurs faibles, ce qui est cohérent avec le résultat du programme.
Cette méthode est réalisée de façon séquentielle, qui a pour complexité $O(s^2)$.

Ensuite, nous avons créé des méthodes pour calculer toutes les caractéristiques à partir d'une image intégrale.
double* **calculateFeatures**(Pic* pInt, vector<Classifier>* table)
double* **calculateFeaturesMPI**(Pic* pInt, vector<Classifier>* table)
Pour la version MPI, chaque processeur se charge d'une partie de classifeurs faibles parmi les 168336 et rassemble les résultats dans un tableau.
La complexité séquentielle est O(c) où c est le nombre de classifieurs faibles.
La complexité en parallèle est en principe en O(c/p) où p est le nombre de processeurs mais le rassemblage des résultats est coûteux.
Pour cette raison, ces deux méthodes ne sont en fait pas utilisées dans la suite. Nous avons implémenté la méthode suivante à la place.
double* **calculateFeaturesPartial**(Pic* pInt, vector<Classifier>* table)
Elle renvoit seulement une partie du résultat selon le rang de processeur, qui a pour complexité O(c/p).

Remarque :
La caractéristique est définie comme la somme des pixels d'une partie des blocs (définis par le type de classifieur faible) moins celle de l'autre partie et ensuite divisée par le nombre de pixels pour l'homogénéité. De cette manière, elle est toujours comprise entre -255 et 255 et ne dépend en général pas de la taille de classifieur faible.

Q2.1 Entraînement des classifieurs faibles

void train (vector<Classifier>* table)
void trainMPI (vector<Classifier>* table)
Pour la version MPI, chaque processeur traite une partie des classifieurs faibles.
La lecture des images n'est pas parallélisée car son influence est bien inférieure au processus de training.
train a pour complexité O(n*s+r*n*c) où r est le nombre de tours.
n*s est la complexité pour la lecture des images
trainMPI a pour complexité O(n*s+r*n*c/p).

Remarque :
w1*X+w2 est une droite et donc seul le taux w1/w2 importe.
Le choix de la valeur epsilon importe peu et s'interprète globalement sur la vitesse de diminuation de w1 et celle d'augmentation de w2.
If faut un nombre suffisant de tours pour que w1/w2 converge. Nous avons testé et finalement choisi 60 tours.
L'allure générale du comportement de w1 et w2 :
Au début, w1 diminue et w2 augmente. Au bout d'un nombre de tours, w1 et w2 diminuent tous les deux. Après un nombre suffisant de tours, par exemple 60, le taux w1/w2 converge, sachant que w1 et w2 continuent à diminuer tous les deux. Si on laisse le calcul continuer, w1 et w2 deviennent très petits et leur taux diverge.

Q2.2 Boosting

ResultClassifier* **boost** (vector<Classifier>* table, **int** round)
ResultClassifier* **boostMPI** (vector<Classifier>* table, **int** round)
Pour la version MPI, chaque processeur calcule le meilleur classifieur faible local parmi une partie de classifieurs faibles et envoit le résultat au processeur de rang 0, qui calcule ensuite le meilleur classifieur global.

boost a pour complexité O(n*s+r*n*c) où r est le nombre de tours (le paramètre N dans l'énoncé). n*s est la complexité pour la lecture des images et r*n*c pour choisir le meilleur classifieur faible parmi tous les classifieurs faibles.
boostMPI a pour complexité O(n*s+r*n*c/p).

Remarque :
Le calcul prend beaucoup de temps donc nous avons décidé de le découper en plusieurs exécutions en sauvegardant les résultats intermédiaires dans des fichiers. Les méthodes concernées sont les suivantes.
void ResultClassifier::saveToFile(string path)
void ResultClassifier::readFromFile(string path)
En plus des coefficients du classifieur final, on stocke aussi les w1, w2 de tous les classifieurs faibles et le tableau des poids pour pouvoir continuer à partir d'un calcul précédent.

Q3.1 Test du classifieur final en variant theta
void test (ResultClassifier* rc, **int** nPoints)
test a pour complexité O(nPoints*n).
int main (**int** argc, **char*** argv[])
main a pour complexité O(argc*nPoints*n).

Ces courbes représentent les performances des classifieurs finaux avec des nombres de composants différents. La courbe rouge est celle du meilleur classifieur final (300 composants). La courbe noire est de 5 composants.

Ce sont les composants principaux du classifieur final. Ce qui est plus clair a un coefficient alpha plus important.

Q3.2 Test des images quelconques

Commandes
Génération d'un détecteur de visage
Compilation :
```
mpic++  MainTraining.cpp Pic.cpp Classifier.cpp ResultClassifier.cpp *.h -o MainTraining
```
Exécution :
```
salloc -n [p] mpirun MainTraining [r] [saveTo] [readFrom(optional)]
```
p : nombre de processeurs
r : nombre de tours de boosting
saveTo : nom du fichier pour sauvegarder un classifieur final
readFrom : nom du fichier à partir duquel on continue le calcul

Test de détecteur
Compilation :
```
mpic++  DetectionEvaluation.cpp Pic.cpp Classifier.cpp ResultClassifier.cpp *.h -o DetectionEvaluation
```
Exécution :

salloc -n 1 mpirun DetectionEvaluation [nPoints] [nomFichier1] [nomFichier2] ...

Conclusion

Notre classifieur final est composé de 300 classifieurs faibles, calculé avec 100 processeurs en une quinzaine de minutes. Avec un theta choisi, les deux taux d'erreur sont environ 6%.