ENSG Géomatique

ÉCOLE NATIONALE
DES SCIENCES
GÉOGRAPHIQUES

Ecole Nationale des
Sciences Géographiques

Cycle des ingénieurs diplômés de l'ENSG 3ème année

Appariement de points d'intérêt par Deep Learning
Guide du développeur

Guillemette Fonteix

Commanditaires : Ewelina Rupnik & Marc Pierrot-Deseilligny

Février 2019

☒ Non confidentiel ☐ Confidentiel IGN ☐ Confidentiel Industrie ☐ Jusqu'au ...

Ecole Nationale des Sciences Géographiques
6-8 Avenue Blaise Pascal - Cité Descartes - 77420 Champs-sur-Marne
Téléphone 01 64 15 31 00 Télécopie 01 64 15 31 07

Table des matières

Chapitre 1
Introduction

Dans le cadre des projets informatiques des étudiants de PPMD de l'ENSG, Ewelina Rupnik et Marc Pierrot-Deseilligny (LaSTIG, IGN) ont proposé un sujet se rattachant à l'appariement de points d'intérêt dans les images. Le but de ce projet est de trouver une méthode alternative à l'algorithme SIFT afin de retrouver les points homologues par Deep Learning. Ce rapport développeur a pour objectif de décrire dans le détail le code et les données du programme et permettre aux commanditaires de réutiliser ce code. Il est complété par le rapport sur la phase de développement et par le guide utilisateur. Il est conseillé d'avoir lu le rapport sur la phase de développement avant celui-ci.

Le code est disponible sur GitHub au lien suivant https://github.com/GuillemetteF/CNN_points_homologues.

Chapitre 2
Généralités

2.1 Librairies utilisées

Le langage de programmation utilisé pour le code est **Python** version 3.6.
Des librairies Python sont nécessaires au bon fonctionnement du programme :

• imageio, pour l'ouverture des images .TIFF.

• Pytorch, utilisé pour récupérer des données d'évaluation, de test et de validation et pour développer l'architecture du CNN. https://pytorch.org/get-started/locally/

• sklearn, permettant la création de la matrice de confusion et des courbes ROC.

• matplotlib, permettant la création de graphiques.

2.2 Rappel des différentes étapes

L'algorithme d'apprentissage peut se décomposer en quatre grandes parties comme indiqué sur le schéma. Premièrement, il a fallu préparer les données en créant des classes pour les points homologues et non-homologues. Puis, j'ai subdivisé les données en échantillons d'apprentissage, de validation et de test.
Ensuite, il s'agissait de créer l'architecture du réseau neuronal (choix du nombre des couches de convolutions, Pooling, etc) et de choisir les différentes caractéristiques de l'algorithme d'apprentissage (fonction d'activation, fonction de perte et algorithme d'optimisation). L'étape suivante est l'entraînement des données et la validation du modèle (permettant à chaque époque d'ajuster les poids).
La dernière étape consistait à utiliser des données de test pour évaluer la performance du réseau (courbe ROC, matrice de confusion et score de précision) pour faire le choix du meilleur modèle.

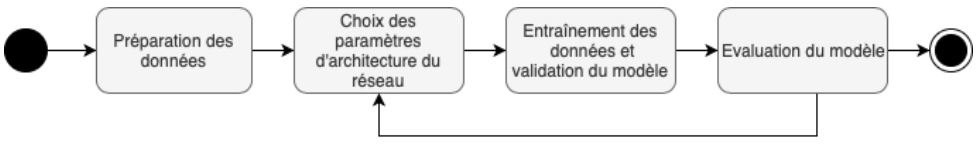

Figure 2.1 – Base de l'algorithme

Chapitre 3
Détails du code

3.1 Découpage des images

Pour chaque type de point caractéristique et pour chaque type de descripteur, il a fallu découper les images .tif en image 10x20 pixels (pour les descripteurs ACGT et ACR0) et en image 10x18 pixels (pour les descripteurs ACGR). On a créé deux classes : homologue et non-homologue.

• **decoupe_homologue(taille_ligne, taille_colonne)** est une fonction permettant de découper les images .tif pour former des couples de descripteurs de points homologues de taille (taille_ligne, taille_colonne) pixels.

• **decoupe_non_homologue(taille_ligne, taille_colonne)** est une fonction permettant de découper les images .tif pour former des couples de descripteurs de points non-homologues de taille (taille_ligne, taille_colonne) pixels.

Il suffit de modifier la taille des images à créer pour réaliser le découpage des images ACGR. On remplace donc *decoupe_homologue(10,20)* par *decoupe_homologue(10,18)* dans le main et il faut faire de même pour **decoupe_non_homologue** .

3.2 Transformations appliquées aux images

```
transform = transforms.Compose([
    PIL.ImageOps.grayscale,
    transforms.ToTensor(),
    transforms.Normalize([mean], [std]),
    ])
```

Figure 3.1 – Transformations appliquées aux images

PIL.ImageOps.grayscale
Pytorch utilise de base des images couleurs sur trois canaux, il faut donc lui indiquer que l'image utilisée est en noir et blanc et donc codée sur un seul canal.

transforms.ToTensor()
On convertit les images en tenseurs pour les utiliser avec Pytorch.

transforms.Normalize([mean],[std])
Nous normalisons ici les images en fonction de la moyenne et de l'écart type sur tout l'échantillon.

La normalisation aide le CNN a avoir de meilleures performances, elle lui permet d'obtenir des données dans une plage restreinte et réduit le biais, ce qui permet d'apprendre plus vite et mieux.

Pour chaque type de descripteur nous avons alors calculé la moyenne et l'écart type. Toutes les valeurs sont récapitulées dans le tableau suivant. Il sera préférable d'utiliser ce tableau plutôt que de les recalculer à chaque fois pour gagner en temps de calcul.

	Mean	Std
eTPR_GrayMax/ eTVIR_ACGT	0.52598995	0.019920329
TPR_GrayMax/ eTVIR_ACR0	0.2673323	0.021500709
TPR_GrayMax/ eTVIR_ACGR	0.42710152	0.027675373
eTPR_GrayMin/ eTVIR_ACGT	0.5278909	0.0183015
TPR_GrayMin/ eTVIR_ACR0	0.2826428	0.023257405
/eTPR_BifurqMin/ eTVIR_ACR0/	0.26071602	0.020207308
/eTPR_BifurqMin/ eTVIR_ACGT/	0.5269597	0.016660387
/eTPR_BifurqMax/ eTVIR_ACR0/	0.25485733	0.019357122
/eTPR_BifurqMax/ eTVIR_ACGT/	0.52704865	0.016844988
eTPR_LaplMin/ eTVIR_ACR0	0.41466388	0.024345702
eTPR_LaplMin/ eTVIR_ACGT	0.5272241	0.018035311
eTPR_LaplMax/ eTVIR_ACGT	0.5333765	0.019080767
eTPR_LaplMax/ eTVIR_ACR0	0.40960723	0.024136739

FIGURE 3.2 – Moyenne et écart type sur nos données

Il sera toutefois possible de calculer la moyenne et l'écart type sur de nouvelles données grâce à la fonction **mean_std()**.

D'autres types de transformations peuvent être appliquées aux images. Voir la documentation : https://pytorch.org/docs/stable/torchvision/transforms.html.

3.3 Partitionnement en données d'entraînement, de validation et de test

Les données d'entraînement correspondent aux données d'apprentissage (training set) qui vont servir à apprendre au réseau. On utilise 60% des données globales.

Les données de validation servent à évaluer les progrès de notre algorithme pendant l'apprentissage. Elles vont permettre d'affiner les paramètres du modèle. Elles sont présentes à 20%.

Enfin, les données de test correspondent à des données mises de côté dès le départ. Elles ne sont utilisées qu'une fois le modèle complètement formé. Ces données serviront à évaluer les modèles en concurrence. On prendra ici 20% des données.

3.4 Choix des paramètres d'architecture du réseau : sur quels paramètres jouer en tant que développeur ?

Voici l'architecture du CNN implémentée en Python dans le script **CNN_points_homologues** :

```
133 class Net(nn.Module):
134     def __init__(self):
135         super(Net, self).__init__()
136         #Our batch shape for input x is (1, 10, 20)
137         #Convolution; Input channels = 1 (grayscale), output channels = 27
138         self.conv1 = nn.Conv2d(1, 27, kernel_size=3, stride =1, padding =1)
139         self.conv2 = nn.Conv2d(27, 27, kernel_size=3, stride =1, padding =1)
140 
141         #Fully connected
142         self.fc1 = nn.Linear(27 * 5 * 10, 400)
143         self.fc2 = nn.Linear(400, 84)
144         self.fc3 = nn.Linear(84, 10)
145         self.fc4 = nn.Linear(10, 2)
146 
147         self.drop_out = nn.Dropout()
148 
149     def forward(self, x):
150         #Size changes from (1, 10, 20) to (27, 10, 20)
151         x = F.relu(self.conv1(x))
152         #Size changes from  (27, 10, 20) to (27, 5, 10)
153         x = F.max_pool2d(x, (2, 2))
154         #Size changes from (27, 5, 10) to (27, 5, 10)
155         x = F.relu(self.conv2(x))
156         #Size changes from (27, 5, 10) to (1, 27*5*10)
157 
158         x = x.view(-1, self.num_flat_features(x))     #self.num_flat_features(x) = 27*5*10
159 
160         #Computes the activation of the first fully connected layer
161         #Size changes from (1, 27*5*10) to (1, 400)
162         x = F.relu(self.fc1(x))
163         #Computes the second fully connected layer
164         #Size changes from (1, 400) to (1, 84)
165         x = F.relu(self.fc2(x))
166         #Computes the second fully connected layer
167         #Size changes from (1, 84) to (1, 10)
168         x = F.relu(self.fc3(x))
169         #allow to avoid overfitting
170         x = self.drop_out(x)
```

FIGURE 3.3 – Réseau neuronal implémenté

On remarque bien dans le constructeur tous les paramètres choisis expliqués dans le rapport sur la phase de développement. On a :

- 2 couches de convolution (kernel_size = 3, padding =1, stride =1)
- 1 couche de Pooling.
- 4 couches de fully-connected.

L'un des aspects embarrassants de la définition manuelle des réseaux de neurones est la nécessité de spécifier les tailles des entrées et des sorties à chaque partie du processus. Le réseau neuronal va donc dépendre de la taille des images en entrée. Or, selon les descripteurs, la taille des images est différente. Elle est la même pour les descripteurs ACR0 et ACGT (taille 10x20 pixels). Le descripteur ACGR est de taille 10x18 pixels. Dans la figure 3.3 notre réseau est adapté aux descripteurs de taille 10X20 pixels. Il faudra changer la taille des entrées et des sorties dans notre CNN en fonction du descripteur utilisé. Les commentaires dans le code devraient donner une idée de ce qui se passe avec les changements de taille à chaque étape et permettre de modifier les tailles en fonction de données utilisées. Dans la pratique il suffira de remplacer la ligne 142 par :

```
self.fc1 = nn.Linear(27 * 5 * 9, 400)
```

FIGURE 3.4 – Changement pour des images de taille 10x18 pixels

En général, la taille de sortie après une convolution de n'importe quelle dimension de notre jeu

d'entrées peut être définie comme suit :

$$output\_size = \frac{in\_size - kernel\_size + 2 \times (padding)}{stride} + 1$$

Il existe un nombre incalculable de combinaisons possibles pour créer un réseau de neurones. Dans notre cas, sachant que les images sont de petites tailles, il paraît évident que le nombre de couches de convolution et de Pooling seront restreintes. Cependant, toutes les architectures n'ont pas été testées. Je vous propose la meilleure que j'ai pu trouver, mais d'autres combinaisons pourront être envisagées. Il sera possible de modifier cette architecture en faisant varier : le nombre de couches de convolution, de pooling et de fully connected ainsi que leur agencement. Les paramètres propres à la convolution et au Pooling peuvent aussi être modifiés : kernel size, stride et padding.

3.5 Entraînement des données, validation et test

Avant de lancer l'entraînement, on pourra aussi modifier le nombre d'époques, le batch size et le learning rate comme indiqué dans le rapport sur la phase de développement.

Les données de validation permettront d'ajuster les poids pendant la phase d'entraînement. Les données de test permettront d'évaluer le modèle à la fin avec l'affichage de la matrice de confusion et des courbes ROC.

3.6 Réutilisation du modèle

Suite à l'apprentissage, un modèle pourra être enregistré et permettra de réutiliser le CNN qui a précédemment appris sur les données. On pourra alors donner de nouvelles images sans label (on se demande si celles-ci sont homologues ou non-homologues sans avoir de réponse).

Vous pourrez choisir l'endroit où le modèle sera enregistré dans la partie 'parameters and paths' (path_model). Un modèle a précédemment été enregistré pour vous sur les points caractéristiques GrayMax pour le descripteur ACGT. Vous pourrez donc réutiliser directement ce modèle (*model.ckpt*) sans réaliser l'apprentissage si vous le souhaitez. Vous pourrez alors commenter le code de l'apprentissage pour ne pas le lancer à nouveau.

Suite à l'enregistrement du modèle, la fonction *reutilisation(path_im)* pourra être lancée avec comme argument le chemin vers l'image à classifier. Cette fonction indique simplement si votre image correspond à un couple de descripteurs de points homologues ou non-homologues (avec une certaine précsion qui correspond à la précision de l'apprentissage).

La fonction *image_loader(path_im)* permet d'appliquer des transformations à l'image à tester.

Table des figures