Projet : inpainting

Vous avez le droit pour ce projet d'utiliser le paquet `scikit-learn` de python, regroupant la plupart des outils usuels de Machine Learning. Vous rendrez un rapport en binôme qui expliquera votre approche, votre protocole expérimental et les analyses de vos expériences, ainsi qu'une archive avec votre code. Vous pouvez rendre le tout sous la forme d'un jupyter-notebook.

1 Préambule : régression linéaire, régression ridge et LASSO

Comme vu dans les séances passées, étant donné un ensemble d'apprentissage $E = \{(\mathbf{x}^i, y^i) \in \mathbb{R}^d \times \mathbb{R}\}$, la régression linéaire s'attache à trouver une fonction linéaire $f_{\mathbf{w}}(\mathbf{x}) = w_0 + \sum_{i=1}^{d} w_i x_i$ de vecteur de poids $\mathbf{w} \in \mathbb{R}^{d+1}$ qui minimise l'erreur aux moindres carrés :

$$\mathrm{MSE}(f_{\mathbf{w}}, E) = \frac{1}{n} \sum_{i=0}^{n} (y^i - w_0 - \sum_{j=1}^{d} w_j x_j^i)^2$$

Cette méthode peut être adaptée à la classification binaire, en considérant les labels dans $Y = \{-1, +1\}$ et en utilisant la fonction de prédiction $\mathrm{sign}(f_{\mathbf{w}}(\mathbf{x}))$: c'est la méthode dite du *plug-in*.
Deux autres variantes sont issues de cette méthode qui considèrent une version pénalisée du coût :

- la régression *ridge* (ou régularisation de Tikhonov, ou régularisation L_2) qui considère une pénalisation par la norme 2 de $\mathbf{w}$:

  $$L_2(f_{\mathbf{w}}, E) = \mathrm{MSE}(f_{\mathbf{w}}, E) + \alpha \|\mathbf{w}\|_2^2$$

  avec $\|\mathbf{w}\|_2 = \sqrt{\sum_{i=0}^{d} w_i^2}$
- l'algorithme du LASSO qui considère une pénalisation par la norme 1 de $\mathbf{w}$:

  $$L_1(f_{\mathbf{w}}, E) = \mathrm{MSE}(f_{\mathbf{w}}, E) + \alpha \|\mathbf{w}\|_1$$

  avec $\|\mathbf{w}\|_1 = \sum_{i=0}^{d} |w_i|$.

Q 1.1 On considère dans la suite le jeu de données USPS (classification de chiffres manuscrits, disponible sur le site de l'UE). Mettez en place un protocole expérimental pour comparer les trois algorithmes dans le contexte de la classification plug-in : la régression linéaire, la régression ridge et l'algorithme du LASSO. En particulier, étudiez l'effet de la régularisation sur le vecteur de poids obtenu en termes de norme et du nombre de composantes non nulles. Que pouvez-vous en déduire sur l'utilité du LASSO ?

2 LASSO et Inpainting

L'*inpainting* en image s'attache à la reconstruction d'images détériorées ou au remplissage de parties manquantes (éliminer une personne ou un objet d'une image par exemple). Cette partie est consacrée à l'implémentation d'une technique d'inpainting présentée dans [1] utilisant le Lasso et sa capacité à trouver une solution parcimonieuse en termes de poids. Ces travaux sont inspirés des recherches en *Apprentissage de dictionnaire et représentation parcimonieuse* pour les signaux [2]. L'idée principale est de considérer qu'un signal complexe peut être décomposé comme une somme pondérée de signaux élémentaires, appelés *atomes*. L'analyse en composante principale (ACP) est un exemple de ce type de décomposition, mais du fait des contraintes d'orthogonalité, les atomes (ou la base canonique) ainsi

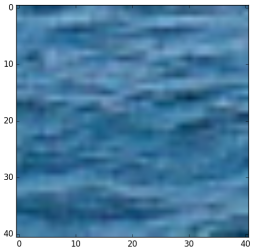
inférés ne sont pas redondants - chacun représente le plus d'information possible et la reconstitution du signal est unique.
Dans le cas de l'inpainting, l'objectif est au contraire d'obtenir un dictionnaire fortement redondant. La difficulté est donc double : réussir à construire un dictionnaire d'atomes - les signaux élémentaires - et trouver un algorithme de décomposition/recomposition d'un signal à partir des atomes du dictionnaire, i.e. trouver des poids parcimonieux sur chaque atome tels que la combinaison linéaire des atomes permette d'approcher le signal d'origine. Dans la suite, nous étudions essentiellement une solution pour la deuxième problématique - la reconstruction - à l'aide de l'algorithme du LASSO.

Formalisation
Pour une image en 2 dimensions, nous noterons $p_{i,j} \in [0,1]^3$ le pixel aux coordonnées (i,j) de l'image exprimée sur 3 canaux : ces 3 canaux sont usuellement le pourcentage de rouge, de vert et de bleu (rgb) de la couleur, ou sous format teinte, saturation, luminosité (hsv), plus proche de notre perception visuelle. Nous appellerons dans la suite un *patch* un petit carré de l'image de longueur de côté h (cette longueur sera une constante fixée au préalable). Nous noterons $\Psi^{p_{i,j}}$ le patch dont le centre est le pixel $p_{i,j}$. Ce patch correspond à une matrice 3d (un tenseur) de taille $h \times h \times 3$ que l'on peut voir sans perte de généralité comme un vecteur colonne de taille $3h^2$ à partir d'une image de taille $(width, height)$, il est possible de construire l'ensemble des patchs $\mathbf{\Psi}$ constituant cette image : $\mathbf{\Psi} = \{\Psi^{p_{i,j}} \in [0,1]^{3h^2} \ i \in \{\frac{h}{2} \dots width - \frac{h}{2}\}, j \in \{\frac{h}{2} \dots height - \frac{h}{2}\}\}$.
De manière générale en inpainting, une hypothèse fondamentale est qu'une image a une cohérence spatiale et de texture : un patch Ψ^p appartenant à une région cohérente en termes de texture doit pouvoir être reconstruit à partir d'une pondération des patchs environnants : $\Psi^p = \sum_{\Psi^{p_k} \in \mathbf{\Psi} \setminus \Psi^p} w_k \Psi^{p_k}$.

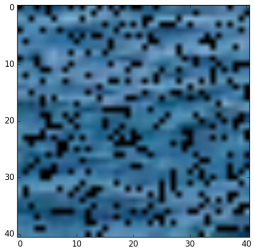
Dans la suite, nous allons considérer qu'une grande partie de l'image n'a pas de pixels manquants. Les patchs issus de cette partie de l'image constitueront notre dictionnaire d'atomes $\mathbf{\Psi}$ sur lesquels porteront nos combinaisons linéaires afin de reconstituer les parties manquantes.
Soit Ψ^p un patch avec des pixels manquants, soit I^p l'ensemble des pixels manquants dans le patch/vecteur Ψ^p et $\bar{I^p}$ son complémentaire (les pixels exprimés). Nous noterons ainsi $\Psi^p[I^p]$ le vecteur restreint aux pixels manquants et $\Psi^p[\bar{I^p}]$ le vecteur restreint aux pixels exprimés du patch. La reconstruction consiste à faire l'hypothèse que si une combinaison linéaire est performante pour approximer $\Psi^p[\bar{I^p}]$, elle sera capable de généraliser aux valeurs manquantes $\Psi^p[I^p]$. Le problème d'optimisation consiste dans ce cas à trouver pour le patch Ψ^p le vecteur $\hat{\mathbf{w}}^p$ tel que :

$$\hat{\mathbf{w}}^p = \text{argmin}_{\mathbf{w}} \|\Psi^p[\bar{I^p}] - \sum_{\Psi^k \in \mathbf{\Psi}} w_k \Psi^k[\bar{I^p}]\|^2 + \lambda \|\mathbf{w}\|_1$$

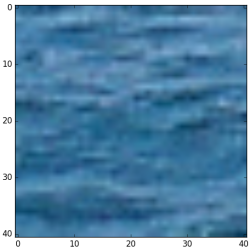
Une fois que la combinaison linéaire qui approche le mieux les pixels exprimés est trouvée, les pixels manquants peuvent être prédits en les complétant par ceux de la combinaison linéaire trouvée.



Patch origine



Patch bruité



Patch débruité

Expérimentation

Q 2.1 Développez dans un premier temps les outils dont vous aurez besoin pour la manipulation des images et des patchs. Vous avez intérêt à transformer les valeurs de chaque canal couleur de l'image en des valeurs dans $[-1, 1]$ et d'utiliser plutôt le codage *hsv* que *rgb* (cf. fonctions `rgb_to_hsv, hsv_to_rgb` de `matplotlib.colors`) pour des rendus plus réalistes à nos yeux. Pour coder les valeurs des pixels manquants, vous pouvez utiliser une constante par exemple fixée à -100 (valeur non utilisée pour le codage des pixels exprimés). Vous aurez besoin en particulier de :

- une fonction `read_im(fn)` qui permet de lire une image et de la renvoyer sous forme d'un `numpy.array` (utilisez `plt.imread(fn)` pour lire un fichier image) ;
- une fonction pour l'affichage de l'image (et des pixels manquants) - en utilisant `plt.imshow` ;
- une fonction `get_patch(i,j,h,im)` qui permet de retourner le patch centré en (i, j) et de longueur h d'une image `im` ;
- des fonctions de conversions entre patchs et vecteurs ;
- des fonctions pour bruiter l'image originale : par exemple, `noise(img,prc)` qui permet de supprimer au hasard un pourcentage de pixel dans l'image, `delete_rect(img,i,j,height,width)` qui permet de supprimer tout un rectangle de l'image ;
- une fonction qui permet de renvoyer les patchs de l'image qui contiennent des pixels manquants ;
- une fonction qui permet de renvoyer le dictionnaire, i.e. les patchs qui ne contiennent aucun pixel manquant.

Adaptez les fonctions à vos besoins, la liste n'est qu'indicative. Pour limiter la taille du dictionnaire, plutôt que de construire les patchs à chaque pixel, vous pouvez construire les patchs tous les `step` pixels. En réglant `step` à la longueur du patch, vous obtiendrez ainsi un quadrillage de l'image.

Q 2.2 Faites une fonction qui prend un patch et un dictionnaire en entrée, et qui rend le vecteur de poids sur le dictionnaire qui approxime au mieux le patch (restreint aux pixels exprimés) en utilisant l'algorithme du LASSO. Testez en particulier sur un dictionnaire issu de l'image complète sans bruit, puis ensuite sur un dictionnaire issu d'une image dont vous avez bruité un certain nombre de pixels, sur des patchs bruités et non bruités. Expérimentez en fonction des valeurs de λ, la pénalisation du LASSO.

Q 2.3 Supposons maintenant que c'est toute une partie de l'image qui est manquante. En considérant des patchs centrés sur les pixels manquants, on commence par remplir en partant des bords puis en remplissant au fur et à mesure vers le centre de l'image. A votre avis, l'ordre de remplissage a-t-il une importance ? Implémentez une fonction qui permet de compléter l'image. Proposez des heuristiques pour remplir de manière intelligente (voir [3]).

Références

[1] Bin Shen and Wei Hu and Zhang, Yimin and Zhang, Yu-Jin, *Image Inpainting via Sparse Representation* Proceedings of the 2009 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP '09)

[2] Julien Mairal *Sparse coding and Dictionnary Learning for Image Analysis* INRIA Visual Recognition and Machine Learning Summer School, 2010

[3] A. Criminisi, P. Perez, K. Toyama *Region Filling and Object Removal by Exemplar-Based Image Inpainting* IEEE Transaction on Image Processing (Vol 13-9), 2004