TME 4 - Perceptron

Vous devez avoir fini le TME précédent avant d'aborder celui-là. Vous aurez besoin des fonctions de coût et de leur gradient que vous avez implémentez lors de la séance précédente.

Perceptron et classe `Linéaire`

Implémentez la fonction `perceptron_loss(w,x,y)` qui rend le coût perceptron ($max(0, -y < \mathbf{x}.\mathbf{w} >)$) et son gradient `perceptron_grad(w,x,y)`.
Afin d'être plus générique et plus flexible, plutôt que d'implémenter une version script de la descente de gradient, nous allons coder une classe `Lineaire` qui va regrouper les différentes variantes possibles. Un squelette de classe est fourni dans le fichier source : il contient principalement le constructeur qui initialise les paramètres du modèle (le coût utilisé `loss`, son gradient `loss_g`, le nombre d'itération de la descente du gradient `max_iter`, le pas `eps`, le vecteur de poids `w`). Complétez le squelette en implémentant :

- la méthode `predict(self,datax)` qui permet d'inférer le label des données `datax` (i.e. qui calcule $sign(< \mathbf{w}.\mathbf{x} >)$)
- la méthode `score(self,datax,datay)` qui permet de calculer le pourcentage de bonne classification sur le jeu de données passé en paramètre.
- la méthode `fit(self,datax,datay)` permet de faire la descente de gradient pendant `max_iter` itérations avec un pas `eps` en utilisant le coût `loss` et le gradient `loss_g`. Ainsi pour faire un perceptron, il suffira de passer lors de la construction de l'objet le coût et le gradient du coût perceptron. Votre méthode devra également conserver en mémoire l'historique des coûts en fonction des itérations.

Comme dans le précédent TME, vos méthodes doivent prendre en entrée des matrices de données (et non pas une seule donnée). Testez votre classe sur l'exemple de la semaine dernière.

Données USPS

Les données USPS sont des données de chiffres manuscrits représentés par une matrice de pixel en niveau de gris de taille 16×16. Le code pour les charger vous est fourni ainsi qu'une fonction de visualisation (`load_usps` et `show_usps`).
Chargez les données et visualisez quelques exemples. Isolez 2 classes (par exemple les 6 vs 9) et entraînez un perceptron dessus. Visualisez la matrice de poids obtenue à l'aide de la fonction `show_usps`. Que remarquez vous ? Comment interpréter le résultat ?
Observez la matrice de poids obtenue lorsque vous entraînez le perceptron avec une classe (6 par exemple) contre toutes les autres classes.
En utilisant les données de test, tracer les courbes d'erreurs en apprentissage et en test en fonction du nombre d'itérations (vous pouvez modifier la fonction `fit` de la classe `Linéraire` afin qu'elle prenne également des données test en entrée). Observez-vous du sur-apprentissage ?

Mini-batch et descente stochastique

Modifiez votre méthode `fit` afin de pouvoir faire une descente stochastique (les exemples sont ordonnés aléatoirement et un seul exemple est pris en compte lors du calcul du gradient et de la mise-à-jour des poids) et mini-batch (l'ensemble des exemples est divisé en petits batchs aléatoire de m exemples, le gradient est moyenné sur chaque batch avant de faire la mise-à-jour). Afin de comparer équitablement

les différentes variantes, le paramètre `max_iter` dénotera le nombre d'époques et non le nombre de mise-à-jour du gradient (une époque étant le fait d'avoir vu une fois tous les exemples). Vous pouvez remarquer que la descente stochastique correspond à une descente mini-batch de taille 1, et une descente batch à une descente mini-batch de taille le nombre d'exemples.
Comparez la vitesse de convergence en particulier en fonction du bruit dans le jeu de données.

Projections et pénalisation

Afin d'augmenter l'expressivité du modèle linéaire, nous allons utiliser des projections. Codez la fonction `proj_poly(datax)` qui renvoie la projection polynomiale de degré 2 des données :
$(1, x_1, x_2, \ \ldots, x_d, x_1^2, x_1x_2, \ldots x_d^2)$.
Codez également la fonction `proj_biais(datax)` qui permet d'ajouter une colonne de 1 en première colonne des données afin d'introduire un biais.
Modifiez votre constructeur pour qu'il prenne potentiellement une projection en paramètre, votre fonction `fit` afin qu'elle projette si besoin les données avant de faire l'apprentissage, ainsi que votre fonction `predict`.
Testez votre projection polynomiale sur les données artificielles de la fonction `gen_arti` de type 1 et 2. Tracez les frontières de décision.
Codez la fonction `proj_gauss(datax,base,sigma)` qui réalise une projection gaussienne de `datax` sur les points $(\mathbf{b}_1, \ldots, \mathbf{b}_b)$ de `bases` avec un paramètre `sigma` : pour une donnée `x` de `datax`, sa représentation est $(e^{-\|\mathbf{x}-\mathbf{b}_1\|^2/2\sigma}, e^{-\|\mathbf{x}-\mathbf{b}_2\|^2/2\sigma}, \ldots, e^{-\|\mathbf{x}-\mathbf{b}_b\|^2/2\sigma})$ et expérimentez sur les 3 jeu de données artificiels. Vaut-il mieux beaucoup de points ou peu de points pour la base de projection ? Un grand ou petit σ ? Quels sont les points qui ont le plus de poids ? Représentez les sur les figures et tracez les frontières.
Vous pouvez également étudié l'effet d'une marge et d'une pénalisation sur les poids de la projection gaussienne : introduisez une nouvelle fonction de coût `hinge_loss(w,x,y,alpha,lambda)` qui retourne $max(0, \alpha - y < \mathbf{w}.\mathbf{x} >) + \lambda\|w\|^2$ et son gradient `hinge_loss_grad`. Observez l'effet de `alpha` et `lambda` en particulier sur le problème de l'échiquier.