

TME 4: Perceptron

1 Implémentation

Télécharger les fichiers sources du TME (`tme_etu.py` et `arftools.py`) et compléter dans `tme_etu.py` les fonctions :

- `hinge(datax, datay, w)`, coût du perceptron : $hinge(\mathbf{x}, y, \mathbf{w}) = \max(0, -y < \mathbf{x} \cdot \mathbf{w} >)$
- `hinge_grad(datax, datay, w)` son gradient.
- `mse(datax, datay, w)`, coût des moindres carrés
- `mse_g(datax, datay, w)`, gradient du coût aux moindres carrés.

Ces fonctions renverront la moyenne de l'erreur et du gradient sur l'ensemble des données `datax`.

Attention !

Vos fonctions doivent pouvoir prendre en entrée des matrices : $X \in \mathbb{R}^{n,d}$, $W \in \mathbb{R}^{1,d}$, $Y \in \mathbb{R}^{n,1}$. Il y a quelques pièges pour la manipulation des matrices avec `numpy` :

- l'opérateur `*` permet de multiplier termes à termes deux matrices de même dimension, mais parfois il fait le produit matriciel lorsque les matrices n'ont pas des tailles compatibles !
- l'opérateur `ndarray.dot()` permet de faire la multiplication matricielle.
- faites le moins possible de boucles (aucune dans le cas des deux fonctions précédentes) ! Python est très lent dans ce cas ...
- parfois vous passerez une matrice en entrée, parfois un vecteur (lorsque vous ne sélectionnez qu'une ligne des exemples), or les opérateurs n'auront pas le même comportement selon les cas : un vecteur en `numpy` est une matrice telle que `shape` renvoie une seule valeur. Transformer de préférence vos entrées de la manière suivante (`datax` matrice d'exemples, `datay` de labels) :
`datax, datay=datax.reshape(len(datay), -1), datay.reshape(-1, 1)`
ou `if len(datax.shape)==1: datax = datax.reshape(1, -1)`.
- Dans le fichier source fourni, l'opération est déjà effectuée à l'aide du décorateur `decorator_vec`. Ce décorateur permet de prétraiter les paramètres passés en argument afin de les transformer en matrice.
- Penser à utiliser `np.sign` et `np.maximum`.

Vérifier que votre code fonctionne en traçant la surface de l'erreur sur un exemple 2D à l'aide de la fonction `plot_error` fournie.

Compléter le code de la classe `Linéaire` pour implémenter une descente de gradient sur le coût passé en argument (la classe représentera ainsi un perceptron si la fonction de coût `hinge` est passée, une régression linéaire si la fonction de coût des moindres carrés est passé).

- Tester sur un exemple simple (type deux gaussiennes, utiliser la fonction `gen_arti()` fournie).
- Tracer la trajectoire de l'apprentissage dans l'espace des poids et les frontières obtenues dans l'espace de représentation des exemples.
- Modifier vos fonctions afin de permettre la prise en compte d'un biais.
- (bonus) Modifier vos fonctions afin de permettre une descente de gradient stochastique et/ou mini-batch.

2 Données USPS

Reprendre les données USPS. La fonction ci-dessous permet d'afficher une donnée :

```
def show_usps(data):  
    plt.imshow(data.reshape((16,16)), interpolation="nearest", cmap="gray")  
    plt.colorbar()
```

- Sur quelques exemples de problèmes 2 classes (6 vs 9, 1 vs 8 par exemple), entraîner votre perceptron et visualiser la matrice de poids obtenue sans le biais.
- Observer 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. Observez-vous du sur-apprentissage ?

3 Données 2D et projection

- Tester votre perceptron sur les autres données artificielles fournies (toujours avec la fonction `gen_arti()`). Que remarquez vous? Est-ce normal ?
- Coder une fonction de projection polynomiale des données comme vu en TD. Faites les expériences et tracer les frontières.
- Coder une fonction de projection gaussienne et expérimenter. Vaut-il mieux beaucoup de points ou peu de points pour la base de projection ?