TP : Descente de gradient et réseaux de neurones

Rémi Flamary

Lors du TP vous aurez besoin des bibliothèque Python `numpy` `pylab` et `torch`. Il vous est conseillé de les importer dès le début avec le code suivant :

```
import numpy as np
import pylab as pl
import torch as th
```

1 Données Digits

Vous utiliserez dans ce TP les données digits dans le fichier 'digitz.npz'.

1. Télécharger le fichier 'digitz.npz' et le charger en mémoire.
2. Transformer les vecteurs de classe pour qu'il contiennent les classes (0,1,2) au lieu de classes (1,7,8).
3. Stocker le nombre d'exemples d'apprentissage `n`, le nombre de variables `d` et le nombre de classes `p=3`.
4. Créer les matrices `Y` et `Yt` de type One-Hot-Encoding (pour chaque exemple la matrice contient un 1 dans la colonne de la classe correspondante).
5. Convertir les array numpy en tenseurs torch (`th.tensor`).

A la fin du TP vous pourrez refaire toutes les étapes pour le jeux de données Pima ou les données jouet 2D. Notez que pour des données de type images il est souvent mieux d'utiliser des réseaux de neurones convolutions ce qui se fait relativement facilement avec Pytorch mais dans ce TP on se concentrera sur la modélisation et l'apprentissage de réseaux classiques.

2 Calcul du coût et du gradient

On va travailler dans cette section sur une réseau de neurone à une couche. La transformation des données $\mathbf{X} \in \mathbb{R}^{n \times d}$ de calcule sous la forme matricielle suivante :

$$\hat{\mathbf{Y}} = \text{softmax}(\mathbf{X}\mathbf{W} + \mathbf{B}) \tag{1}$$

où les paramètres de la couche de neurone sont $\mathbf{W} \in \mathbb{R}^{d \times p}$ et $\mathbf{B} \in \mathbb{R}^{1 \times p}$. L'opérateur de softmax est appliqué aux lignes de la matrice et permet de s'assurer que les sorties sont positives et somment à 1 ce qui correspond à une prédiction de la probabilité de chaque classes.

Le coût de classification que l'on veut optimiser les le suivant :

$$J(\mathbf{Y}, \hat{\mathbf{Y}}) = -\frac{1}{n} \sum_{i,j} Y_{i,j} \log(\hat{Y}_{i,j}) \tag{2}$$

L'objectif de l'apprentissage du réseau à une couche est donc de minimiser le cout ci dessus par rapport aux paramètres $\mathbf{W}, \mathbf{B}$.

Toutes les consignes suivantes doivent être codés à la suite dans une seule cellule de code (spyder ou notebook).

1. Initialiser la matrice $\mathbf{W}$ de taille $d \times p$ avec des nombres aléatoire gaussiens de petite amplitudes (`np.random.randn` multipliés ar 0.01) puis le convertir en tenseur torch (`th.tensor`) en prenant bien soins de mettre le paramètre `requires_grad=True` pour que torch puisse calculer son gradient.
2. Initialiser la matrice de biais $\mathbf{B}$ de taille $1 \times p$ (`th.zeros`) en prenant bien soins de mettre le paramètre `requires_grad=True` pour que torch puisse calculer son gradient.
3. Calculer la prediction $\hat{\mathbf{Y}}$ pour les données d'apprentissage (`th.mm,th.softmax`). Notez que la multiplication entre deux matrice se code en torch avec `th.mm` et qu'il est nécessaire de donner à la fonction `th.softmax` l'axe sur lequel le softmax est appliqué.
4. Vérifier que les lignes de $\hat{\mathbf{Y}}$ somment à 1. La prédiction est elle une bonne approximation des vrais étiquettes $\mathbf{Y}$?
5. Calculer le coût J entre la prédiction $\hat{\mathbf{Y}}$ et $\mathbf{Y}$ (`th.mean,th.sum,th.log`) et le stocker dans une variable `loss`. Ajouter une toute petite valeur (10^{-16}) à l'intérieur du log pour éviter les problèmes numériques.
6. Coder une fonction `def pred(X,W,B)` qui calcule la prédiction et uen fonction `def cout (Y,Yhat)` qui calcule le coût pour une prédiction donnée.
7. Calculer les gradients du coût par rapport à $\mathbf{W}, \mathbf{B}$ en exécutant `loss.backward()`. Les gradients de `W` et `B` sont accessible ensuite dans `W.grad` et `B.grad`.
8. Exécuter plusieurs fois le code pour différentes initialisations aléatoires de $\mathbf{W}$.

3 Descente de gradient

Dans cette section nous allons tout d'abord coder une descente de gradient manuelle puis ensuite utiliser la classe `th.optim.SGD` de torch qui servira également à apprendre des modèles plus complexes. Les étapes d'initialisation de $\mathbf{W}$ et $\mathbf{B}$ doivent toujours être exécutées avant la boucle de descente.

1. Initialiser le pas `step=1`, le nombre d'itérations `niter=200` et la liste des valeur de cout `lst_loss=[]`.
2. Dans une boucle for, exécuter `niter` fois els étapes suivantes :
   (a) Calculer le coût `cost` pour les parameters `W,B` (cf section précédente, utiliser les fonctions).
   (b) Ajouter le coût à la liste des coûts (`lst_cost.append(float(loss))`).
   (c) Calculer les gradients (`loss.backward()`)
   (d) Metter à jour des parameters `W,B` dans un environement `with th.nograd():` :
   — Pas de gradient sur `W,B` avec l'opérateur `-=`.
   — Mise à zero des gradient avec `W.grad.data.zero_()` .
3. En sortie de la boucle visualiser l'évolution du coût `lst_loss` le long dess itérations (`pl.plot`). Est ce que l'algorithme a convergé ?
4. Changer le nombre max d'itération `niter` et la valeur du pas `step` et observer les effets sur la décroissance du coût.

5. Calculer le taux de bonne reconnaissance pour votre modèle. Pour celà vous devrez calculer la prédiction $\hat{\mathbf{Y}}$ puis la convertir en tableau numpy (`Yhat.detach().numpy()`) puis ensuite utiliser `np.argmax,np.mean`.
6. Copier coller votre boucle de descente de gradient et la modifier pour utiliser la classe `th.optim.SGD` qui code une descente de gradient (stochastique) :
   (a) Initialiser le solver avant la boucle en lui passant les variables à optimiser (`[W,B]`) ainsi que le parametre `lr` qui veut dire "learning rate" c'est à dire le pas (`step`).
   (b) Dans la boucle après le calcul du gradient effectuer un pas d'optimisation avec la méthode `.step()` de l'optimiseur (il fera un pas de gradient pour toutes les variables qu'on lui a donné à optimiser).
   (c) Après le pas de gradient pensez à remettre les gradients à 0 avec la méthodes `.zero_grad()` à la fin de la boucle.

   L'implémentation avec l'optimiseur `th.optim.SGD` a l'avantage d'être plus générique et permet d'optimiser un grand nombre de paramètres automatiquement. Celà permettra dans la suite d'optimiser une réseau de neurone multi-couche.

Notez qu'il est relativement facile de changer la fonction de coût, par exemple vous pouvez assez simplement changer le coût J par un coût quadratique. Quelle est la performance de classifieur dans ce cas la ? N'hésitez pas à expérimenter, c'est comme celà qu'on devient un data scientist.

4 Réseau de Neurones multi-couche

Maintenant que vous savez optimiser un critère par rapport à des paramètres, vous pouvez optimiser des modèles plus complexes par exemple un réseau de neurone à deux couches (une couche cachée).

Implémentation manuelle Dans un premier temps nous allons implémenter une réseau de neurone à deux couche et son optimisation de manière "manuelle" c'est à dire en initialisant et en optimisant tous les paramètres dans le code.

1. Initialiser la variable `nb_hidden=100` qui définit combien de neurones sont dans la couche cachée.
2. Initialiser les parametres `W1,B1` de tailles respectivement d$\times$ nb_hidden et 1$\times$ nb_hidden pour la première couche. Initialisez les parametres `W2,B2` de tailles respectivement nb_hidden$\times$ p et 1$\times$ p pour la seconde couche.
3. Implémenter la fonction `def pred2(X,W1,B1, W2, B2)` qui implémente l'opérateur linéaire de la première couche (`th.mm`) suivi par une activation de type ReLU (`th.relu`$= \max(0, x)$), puis par l'opérateur linéaire (`th.mm`) de la seconde couche avec finalement une softmax (`th.softmax`).
4. Initialiser un optimiseur de type `SGD` en lui donnant à optimiser tous les paramètres du modèle et lancer la procédure d'optimisation (comme vous avez appris dans la section précédente).
5. Quelle est la performance ? Que se passe-t-il si on change le nombre de neurones cachées ?

Réseau de Neuronen définit comme une classe Pytorch fournit une interface de type programmation objet pour définir un réseau de neurone.

L'idée principale est d'implémenter le réseau à travers une nouvelle classe définie de la manière suivante :

```
class MLP(th.nn.Module):

  def __init__(self):
      super(MLP, self).__init__()
      # creation des couches et stockage dans self

  def forward(self,x):
      # calcul de la sortie pred a partir des couche
      return pred
```

La première ligne de la fonction `__init__` doit être absolument conservée pour initialiser correctement votre modèle. La fonction `__init__` permet d'initialiser les différentes couches du modèle et de les stocker dans l'object (par exemple dans `self.c1`). Elle est exécutée lors de la création de l'objet. La fonction `forward` permet de définir exactement les opérations effectuées par le model, c'est l'équivalent des fonctions `f` et `f2` que vous avez codé précédemment sauf que les opérations n'utilisent pas des variables mais les différentes couches qui ont été définies dans la fonction `__init__`.

1. Copier la définition ci dessus dans votre code. Modifiez la fonction `forward` pour qu'elle retourne `x`. Vous avez codé un réseau de neurone qui retourne exactement son entrée (ça ne sert pas à grand chose). Initialisez un réseau avec `mlp=MLP()` et vérifiez qu'n appelant le réseau comme une fonction (`mlp(x)`) la sortie est bien identique à l'entrée.
2. Modifier la fonction `__init__` en créant à l'intérieur les deux couches linéaires et les stocker dans la classes `self.c1,self.c2` (`th.nn.Linear`). Attention à bien donner à la classe `th.nn.Linear` les bonnes dimensions d'entrée et de sortie des couches (celles de `W1` et `W2`). Il faudra également convertir les deux couche au format double en exécutant `.double()` lors de leur initialisation.
3. Modifier la fonction `forward` pour qu'elle calcule la prédiction de réseau de neurone en utilisant les deux couches stockées dans `self` et les activations (`th.relu,th.softmax`).
4. Initialiser un modèle de votre classe et vérifiez que les lignes de la sortie somment à 1.
5. Optimiser les paramètres du réseau de neurones avec un boucle et un optimiseur de Type `SGD`. Lors de son initialisation, vous pouvez lui donner directement tous les paramètres de votre modèle avec la fonction `mlp.parameters()`.
6. Vérifier que vous retrouvez des performances similaires à l'implémentation manuelle.

5 Gradient stochastique

Vous avez utilisé l'optimiseur `SGD` dans les sections précédentes mais vous avez à chaque fois calculé le gradient sur toutes les données d'apprentissage ce qui veux dire que vous avez en réalité implémenté une descente de gradient classique. Dans cette section vous allez implémenter un vrai gradient stochastique en calculant le coût sur une partie tirées aléatoirement des données.

1. Copier le code d'initialisation et d'optimisation de réseau de neurone à deux couches de la section précédente. Initialisez la taille du minibatch `mb=20`.

2. Dans la boucle d'optimisation au lieu de calculer le coût sur toute la base de donnée, selectionner `mb` exemples aléatoirement avec `th.randint(n,(mb,))`.
3. Observer le coût le long des itérations. Il est beaucoup plus bruité (car calculé sur une petite partie des exemples). Il y a-t-il bien une tendance à la décroissance ?
4. Choisir un nombre d'itérations en gradient stochastique qui permet de "converger".
5. Quelle est la performance finale ? Est-elle meilleure que la performance obtenue avec un gradient complet ?
6. L'optimisation ets elle plus rapide ? plus lente ?

6 Soyez curieux (Bonus)

Vous avez maintenant les outils pour designer et optimiser des réseaux de neurone avec un nombre quelconque de couche (il suffit de modifier la classe MLP). Voici quelques pistes de choses à regarder pour bien comprendre comment marchent les réseaux de neurones :

— Quel est l'effet du nombre de neurones cachés ? Quel est l'effet du nombre de couches.
— Changer la fonction d'activation (autre que ReLU). Le réseaux de neurone profond marche aussi bien ?
— Quel est l'effet de la taille du minibatch ?
— On sait que les images de type MNIST sont mieux classifiées par des réseaux de neurone convolutionels. Suivre un tutoriel en ligne pour voir comment créer et apprendre un réseau de neurone convolutionel en Pytorch.