AMAL - TP 8 - Régularisation et protocole expérimental

Nicolas Baskiotis - Benjamin Piwowarski - Laure Soulier

2020-2021

1 Introduction (brève, cf cours)

Dans ce TP, nous allons nous intéresser aux mécanismes qui permettent de généraliser *mieux*.

— Régularisation L1 et L2
— Dropout
— BatchNorm
— LayerNorm

Pour le Dropout et la BatchNorm, dans les deux cas, le réseau de neurones peut être utilisé soit en inférence, soit en apprentissage : son comportement va différer selon le mode. En `PyTorch`, pour indiquer le mode, vous devez faire appel aux méthodes `eval()` et `train()` du module.

2 Préparation des données et du modèle

Vous utiliserez les données MNIST.

Important

Dans ce TP, vous n'utiliserez que 5% des données d'entraînement (sinon les techniques de régularisation n'auront aucun effet).

```
from datamaestro import prepare_dataset
ds = prepare_dataset("com.lecun.mnist")

# Ne pas oublier se sous-échantillonner !
train_img, train_labels = ds.train.images.data(), ds.train.labels.data()
test_img, test_labels =  ds.test.images.data(), ds.test.labels.data()
```

Vous utiliserez un réseau composé de 3 couches linéaires avec 100 sorties, suivis d'une couche linéaire pour la classification (10 classes – les chiffres de 0 à 9). Vous utiliserez un coût cross-entropique, des batchs de taille 300, et 1000 itérations (*epochs*).

Question 1

Afin d'observer les effets de la régularisation, il faudra enregistrer avec `torch.utils.tensorboard` :

- Les coûts (train, validation, test).
- Les poids de chaque couche linéaire (histogramme).
- Le gradient à l'entrée de chaque couche linéaire ; Pour enregister le gradient par rapport à une variable, utilisez la fonction `store_grad`.
- L'entropie sur la sortie (histogramme) – calculez rapidement l'entropie d'un modèle aléatoire en guise de comparaison.
  Attention : Éviter d'enregistrer des histogrammes à chaque itération ; faites en sorte d'en enregistrer une vingtaine au plus lors de l'apprentissage afin d'économiser de l'espace de stockage et du temps de calcul.

3 Régularisation des modèles

Régularisation L1 et L2

La régularisation L1 ou L2 correspond à poser un a priori sur les paramètres. En utilisant le critère de maximum a posteriori – à un a priori de Laplace (pour L1) ou Gaussien (pour L2). En effet, on cherche à maximiser

$$p(\theta|\text{données}) = \frac{p(\text{données}|\theta)p(\theta)}{p(\text{données})}$$

ce qui revient à minimiser

$$\log p(\theta|\text{données}) = \text{constante} - \log p(\text{données}|\theta) - \log p(\theta)$$

Pour un a priori $p(\theta)$ de Laplace, on a :

$$\log p(\theta) = \text{constante} + \underbrace{\frac{1}{b}}_{\lambda_1} ||\theta||_1$$

Pour un a priori $p(\theta)$ Gaussien, on a :

$$\log p(\theta) = \text{constante} + \underbrace{\frac{1}{2\sigma^2}}_{\lambda_2} ||\theta||_2^2$$

Il y a donc un hyperparamètre à régler pour une régularisation L1/L2. Dans la pratique, plus celui-ci est fort, moins les paramètres vont dévier par rapport à 0. Observez les effets

Dropout

L'idée du Dropout est proche du moyennage de modèle : en entraînant k modèles de manière indépendante, on réduit la variance du modèle. Entraîner k modèles présente un surcoût non négligeable, et l'intérêt du Dropout est de réduire la complexité mémoire/temps de calcul. L'idée est de mettre chaque sortie d'une couche de Dropout à zéro avec une probabilité p (un hyperparamètre).

Important

Lors de l'inférence, cette probabilité p doit être ignorée (et valeurs ajustées car toutes les entrées sont présentes en multipliant par $(1-p)^{-1}$). Pour cela, il faut spécifier si le modèle est appris (avec `model.train()`) et quand le modèle est utilisé en inférence (avec `model.eval()`).

BatchNorm

On sait que les données centrées réduites permettent un apprentissage plus rapide et stable d'un modèle ; bien qu'on puisse faire en sorte que les données en entrées soient centrées réduites, cela est plus délicat pour les couches internes d'un réseau de neurones. La technique de BatchNorm consiste à ajouter une couche qui a pour but de centrer/réduire les données en utilisant une moyenne/variance glissante (en inférence) et les statistiques du batch (en apprentissage).

Utilisez la classe `BatchNorm1d`.

Important

Là encore, il y a un mode apprentissage et inférence distinct : dans ce dernier, la moyenne/variance n'est plus mise à jour.

LayerNorm

La technique de LayerNorm est une alternative à BatchNorm, utilisée surtout lorsqu'on a pas ou peu de données pour estimer la variance et la moyenne (comme par exemple en apprentissage par renforcement). Elle consiste à normaliser les données par individu (et non par caractéristique comme dans BatchNorm).

Utilisez la classe `LayerNorm`.

Question 2

Changez votre modèle afin d'utiliser une des différentes régularisations proposées plus haut. Certaines peuvent être combinées (par exemple L1/L2 avec Dropout, BatchNorm ou LayerNorm).

4 Augmentation de données (optionnel)

Une autre technique pour éviter le sur-apprentissage est de créer artificiellement plus de données. Vous pourrez utiliser les fonctions disponibles dans la librairie torchvision.

Question 3

Pour cela, on peut procéder à des transformations qui *ne changent pas la classe attendue* :

- Ajout d'un bruit Gaussien
- Dans le cas des images, des transformations géométriques non destructives (rotations, zoom, etc.)

5 Recherche d'hyperparamètre

Vous essaierez de faire une recherche des meilleurs hyperparamètres en utilisant les 95% du jeu de train restant comme validation. Vous pouvez essayer de combiner différentes formes de régularisation. De nombreuses librairies existent (ex. nevergrad qui est encore en développement), mais nous vous conseillons *optuna* qui permet de faire d'avoir facilement des dépendences entre les hyperparamètres.

Question 4

Implémentez une recherche d'hyperparamètres en utilisant la librairie optuna pour l'optimisation d'hyperparamètres.