Deep Learning TP1

Reconnaissance de chiffres avec les données MNIST

Dernière modification: 19 novembre 2024

L'objectif de ce TP est de programmer un réseau de neurones qui reconnaît des chiffres manuscrits. Nous utiliserons le jeu de données classique MNIST [1], qui sera installé automatiquement, et la bibliothèque PyTorch [2]. Dans ce TP, nous n'utiliserons que des fonctionnalités basiques de PyTorch et nous implémenterons notre réseau de neurone à partir de (presque) rien. Nous pourrions faire la même chose avec seulement NumPy, ou même en réimplémentant les opérations matricielles à la main, mais ici nous utiliserons PyTorch et ses tenseurs pour tout.

Puisque nous utiliserons PyTorch, Il faut installer (via `pip` par exemple) les modules `torch` et `torchvision` pour Python. Une installation classique comprend aussi le module `torchaudio` même si nous n'en aurons pas besoin ici.

Le fichier `tp_deep_mnist.py` fourni contient un programme minimal pour démarrer, qui lit les données d'entraînement, et les télécharge la première fois.

1 Création d'un réseau de neurones

On veut que le réseau lise une image de 28×28 pixels en niveaux de gris compris entre 0 et 255, sous la forme d'une vecteur ligne de valeurs comprises entre 0 et 1. On créera donc une couche d'entrée avec $28 \times 28 = 784$ neurones. On créera également deux couches cachées contenant respectivement 30 et 20 neurones, et une couche de sortie de 10 neurones représentant les chiffres de 0 à 9. La réponse du réseau sera donnée par le numéro du neurone de sortie ayant la plus grande activation. Dans les trois couches qui ne sont pas l'entrée, on utilisera la fonction logistique $\mathcal{L}(x) = \frac{1}{1+e^{-x}}$ de dérivée $\mathcal{L}'(x) = \mathcal{L}(x)(1 - \mathcal{L}(x))$.

Q1. Créer une classe `Net`, avec comme attributs (i) les différentes tailles de couches, (ii) les matrices w^ℓ, pour ℓ de 1 à 3, représentant chacune les paramètres des liens entre les neurones [3] de la couche $\ell - 1$ et la couche ℓ (avec la couche d'entrée numérotée 0), (iii) les vecteurs lignes b^ℓ représentant les biais des différentes couches. On initialisera [4] les biais à 0 et les matrices selon une loi normale de moyenne 0 et de déviation standard 1.

Q2. Changer la représentation du tenseur d'entrée pour obtenir [5] un tenseur de taille 60000×784, au lieu de $60000 \times 28 \times 28$, contenant des valeurs réelles entre 0 et 1 (au

1. `http://yann.lecun.com/exdb/mnist/`
2. `https://pytorch.org/`
3. On reprendra la convention de PyTorch selon laquelle chaque ligne correspond à un neurone de la couche $\ell - 1$, et chaque colonne à un neurone de la couche ℓ.
4. On pourra utiliser les fonctions `torch.zeros` et `torch.normal`
5. On pourra utiliser la méthode `view` des tenseurs.

lieu de valeurs entières entre 0 et 255). On pourra le faire une fois pour toute sur tout le *dataset*.

Q3. Pour les données d'entraînement seulement, transformer les étiquettes, qui sont des nombres entre 0 et 9, en des vecteurs lignes à 10 colonnes, avec un 1 pour la composante dont l'indice correspond à l'étiquette, et des 0 pour les autres [6] ;

Q4. Créer deux fonctions `logistic` et `logistic_d` implémentant la fonction logistique et sa dérivée. On les définit comme sur des scalaires mais en les appliquant à un vecteur, celui-ci sera affecté composante par composante [7].

2 Propagation en avant et test de précision

Q5. Ajouter une méthode `forward` à la classe `Net` qui calcule et renvoie les vecteurs de préactivation z^ℓ et d'activation a^ℓ pour chacune des couches ℓ, avec ℓ de 1 à 3. L'entrée du réseau étant le vecteur ligne a^0, on a la formule suivante [8] :

$$z^\ell = a^{\ell-1} w^\ell + b^\ell \text{ et } a^\ell = \mathcal{L}(z^\ell)$$

Q6. On veut maintenant calculer la précision du réseau non entraîné ; on s'attend donc à environ 10%. Ajouter une fonction globale (pas une méthode) `accuracy` qui prend en entrée un tenseur $n \times 784$ représentant n images, ainsi qu'une tenseur $n \times 10$ représentant les réponses attendues, et calcule pour chaque image la réponse du réseau [9], compte le nombre de succès, et renvoie le ratio nombre de succès sur nombre total.

La taille d'un vecteur ligne de 10 colonnes (p. ex.), obtenue avec la méthode `size()` des tenseurs, doit être `torch.size([1,10])` et non pas `torch.size([10])`. Il faut bien vérifier que `a0`, en particulier, mais aussi les vecteurs de biais et autres ont bien la bonne taille ; sinon les calculs peuvent être faux, sans que PyTorch ne signale d'erreur : dans le 2e cas, par exemple, la transposition ne fait rien, ce qui sera embêtant pour l'entraînement...

3 Entraînement

On va maintenant entraîner le réseau sur le jeu d'entraînement par une descente de gradient stochastique simple. On utilise la fonction de perte *cross-entropy* `Loss`, de sorte qu'en conjonction avec la fonction d'activation logistique sur la couche de sortie, l'erreur modifiée sur la couche de sortie vaut [10] :

$$\Delta^3 = a^3 - y$$

Dans cette expression, les 3 représentent le numéro de la couche de sortie, pas une puissance, et y est le vecteur ligne à 10 colonnes représentant la réponse attendue (voir Q3). Pour

6. On pourra utiliser la fonction `torch.nn.functional.one_hot`.
7. Utiliser la fonction `torch.exp` pour l'exponentielle.
8. On pourra utiliser la fonction `torch.mm` ou la fonction `torch.matmul`.
9. On pourra utiliser la fonction `torch.argmax`.
10. On n'aura pas besoin d'implémenter la fonction de perte explicitement, car elle n'intervient ici qu'au travers de la définition de Δ^3.

les couches 1 et 2, on utilise un calcul en arrière selon la formule suivante, dans laquelle $\top$ représente la transposition [11] :

$$\Delta^{\ell-1} = \Delta^{\ell}(w^{\ell})^{\top} \circ \mathcal{L}'(z^{\ell-1})$$

Dans cette expression, $\circ$ est la multiplication composante par composante de deux vecteurs (le *produit de Hadamard*).

Une fois l'erreur modifiée calculée, on trouve les gradients par :

$$\frac{\partial \mathsf{Loss}}{\partial w^{\ell}} = (a^{\ell-1})^{\top}\Delta^{\ell} \text{ et } \frac{\partial \mathsf{Loss}}{\partial b^{\ell}} = \Delta^{\ell}$$

Enfin, pour un pas d'apprentissage α donné, qu'on pourra prendre ici égal à 0.05, on modifie les paramètres par :

$$w^{\ell} \leftarrow w^{\ell} - \alpha\frac{\partial \mathsf{Loss}}{\partial w^{\ell}} \text{ et } b^{\ell} \leftarrow b^{\ell} - \alpha\frac{\partial \mathsf{Loss}}{\partial b^{\ell}}$$

Q7. Ajouter une méthode `backward` à la classe `Net`, qui pour des valeurs des z^{ℓ} et a^{ℓ} données, ainsi que la valeur de y, renvoie les gradients associés à chaque matrice et vecteur contenant des paramètres du réseau.

Q8. Ajouter une classe `SGD` qui représente l'algorithme d'optimisation par descente de gradient stochastique. On lui ajoutera α comme attribut, ainsi qu'une méthode `step` qui modifie les paramètres du réseau selon la procédure décrite ci-dessus [12].

Q9. Écrire une fonction (pas une méthode) `epoch`, qui prend en entrée un tenseur représentant des images, un tenseur représentant les réponses attendues, et un optimiseur instanciant la classe `SGD`. Pour toutes les images prises dans une ordre aléatoire [13], elle calcule à chaque image les a^{ℓ} et les z^{ℓ} correspondant, puis les gradients, et enfin modifie les paramètres du réseau.

Q10. Écrire une fonction (pas une méthode) `train` qui initialise l'optimiseur SGD, effectue un nombre d'*epochs* donné en paramètre, et calcule à la fin de chacune la précision sur le jeux de test (et éventuellement d'entraînement). Faire 30 *epochs*. On s'attend à un taux d'erreur inférieur à 10% dès la première *epoch*, avec une lente amélioration sur les *epochs* suivantes jusqu'à 4 ou 5%.

Éventuellement, garder les données produites pour tracer une courbe d'évolution du taux d'erreur en fonction des *epochs* et comparer avec les variations introduites dans les questions suivantes.

4 Descente de gradient par minibatches

En pratique, on effectue plutôt des descentes de gradient par *minibatches*, c'est-à-dire qu'au lieu de traiter chaque image séparément pour l'entraînement, on les traite par paquet de taille m fixée. Ici on prendra $m = 12$. À la fin on calcule le gradient *moyen* sur le paquet de m images. Cela a deux avantages : premièrement, on peut traiter les paquets d'images

11. On pourra utiliser la fonction `torch.t` ou la fonction `torch.transpose`, ou les méthode de tenseur correspondantes.

12. Cela implique d'avoir un attribut qui donne un lien vers le réseau de neurone à optimiser.

13. On pourra utiliser la fonction `torch.randperm`.

efficacement en les encodant dans une matrice, ce qui permet de mieux profiter de l'efficacité des algorithmes de multiplication de matrices par blocs. Deuxièmement, on peut espérer que le gradient moyen est plus représentatif du gradient global que les gradients individuels. En contrepartie, on fait moins de mises-à-jour des paramètres par *epoch*.

Au lieu d'avoir un vecteur ligne a^0, on construit une *matrice* a^0 avec m lignes qui contiennent chacune une image [14]. On peut montrer que les formules vues précédemment restent valables [15] et fournissent des z^ℓ, a^ℓ et Δ^ℓ qui sont également des matrices à m lignes avec chaque ligne qui contient la valeur associée à l'image de la ligne correspondante dans a^0.

Pour le gradient, on veut calculer une moyenne sur les lignes. On obtient alors les formules suivantes, avec $\mathbf{1}^\top$ un vecteur ligne de m colonnes ne contenant que des 1 :

$$\left(\frac{\partial \mathsf{Loss}}{\partial w^\ell}\right)_{\text{moyen}} = \frac{1}{m}(a^{\ell-1})^\top \Delta^\ell \text{ et } \left(\frac{\partial \mathsf{Loss}}{\partial b^\ell}\right)_{\text{moyen}} = \frac{1}{m}\mathbf{1}^\top \Delta^\ell$$

La mise-à-jour des paramètres se fait avec ce gradient moyen.

Q11. Modifier les méthodes `backward` et `epoch` pour procéder par `minibatch`. À chaque *epoch*, chaque *minibatch* contient m images aléatoires. Le plus simple est de calculer une permutation sur les indices des images, puis de prendre les images m par m dans l'ordre sur cette permutation.

Q12. On peut aussi calculer dans `accuracy` les activations sur des `minibatches`, potentiellement plus gros que pour l'entraînement, voire sur tout le jeu de données à la fois. On peut aussi optimiser le calcul du nombre de succès [16].

Q13. Comparer les performances au niveau temps d'exécution, et convergence.

5 Améliorations

Q14. Xavier/Glorot normal initialization. Pour avoir une dérivée pas trop proche de 0 avec la fonction logistique, ce qui impliquerait une erreur modifiée et un gradient proches de 0, il ne faut pas avoir des préactivations (en avant), ou des erreurs modifiées (en arrière) trop grandes en valeurs absolues, et donc il faut que chaque matrice w^ℓ contienne des valeurs suffisamment petites par rapport aux nombres de neurones en entrée et en sortie (et donc par rapport à la moyenne des deux nombres) de la couche qu'elle représente. Une façon simple de faire cela est, pour chaque couche ℓ, d'initialiser les paramètres de w^ℓ selon une loi normale de moyenne 0 mais de déviation standard $\sqrt{\frac{2}{n_{\ell-1}+n_\ell}}$, où n_i est le nombre de neurones dans la couche i. Implémenter cette optimisation et comparer.

Q15. SGD with momentum. On peut aussi améliorer la descente de gradient ; une façon classique de faire est d'ajouter une certaine inertie au gradient pour qu'il ne varie pas trop vite. En notant avec l'indice t la valeur courante et $t-1$ la précédente,

14. On pourra utiliser la fonction `torch.stack` pour créer le *minibatch*.

15. Dans `forward`, il faudrait quand même étendre b^ℓ pour en faire une matrice ; pour cela on pourrait ajouter $\mathbf{1}b^\ell$ à la place, où $\mathbf{1}$ est un vecteur colonne de m lignes, mais en fait PyTorch est capable d'adapter tout seul la dimension de b^ℓ en le dupliquant en lignes (*broadcasting*), donc il n'y a rien à faire.

16. Par exemple, en faisant travailler `argmax` sur les colonnes (avec l'argument `axis = 1`), en utilisant le fait que l'opérateur `==` appliqué à deux matrices de même taille produit une matrice de booléens, et enfin en utilisant la fonction `torch.sum`.

avec g le gradient calculé par `backward`, u un terme représentant l'inertie (initialement nul), w représentant les paramètres (y compris les biais), et γ une constante qu'on prendra ici égale à 0.4, on a :

$$u_t = \gamma u_{t-1} + \frac{\alpha}{m} g_t \text{ et } w_t = w_{t-1} - u_t$$

Implémenter ce calcul dans la classe `SGD` et comparer.