IFT 3395 Fondements de l'apprentissage machine
Prof. Guillaume Rabusseau

Devoir 3 - Partie pratique

- Ce devoir doit être déposé sur Gradescope et doit être fait en équipe de 2 ou 3. Vous pouvez discuter avec des étudiants d'autres groupes mais les réponses soumises par le groupe doivent être originales. À noter que nous utiliserons l'outil de détection de plagiat de Gradescope. Tous les cas suspectés de plagiat seront enregistrés et transmis à l'Université pour vérification.
- La partie pratique doit être codée en Python (avec les librairies numpy, matplotlib et PyTorch), et envoyée sur Gradescope sous la forme d'un fichier python. Pour permettre l'évaluation automatique, vous devez travailler directement sur le modèle donné dans le répertoire de ce devoir. Ne modifiez pas le nom du fichier ou aucune des fonctions signatures, sinon l'évaluation automatique ne fonctionnera pas. Vous pouvez bien sûr ajouter de nouvelles fonctions et importations python
- Tous les graphiques, tableaux, dérivations ou autres explications doivent être soumis à Gradescope sous forme de rapport pratique et **séparément de la partie théorique du devoir**. Vous êtes bien sûr encouragés à vous inspirer de ce qui a été fait en TP.

1 Entrainement de réseaux de neurones avec la différentiation automatique [35 points]

Cette partie consiste en la mise en place d'une classe `Trainer` pour entraîner des réseau de neurones pour la classification multi-classes. Elle sera basée sur PyTorch et inclura un ensemble de fonctionnalités. Vous devrez implémenter la plupart des fonctions requises dans la classe `Trainer` fournie dans le modèle de solution. Dans la section suivante, vous mettrez `Trainer` au travail.

Vous explorerez divers perceptrons multicouches (MLP) et réseaux de neurones convolutifs (CNN) mais la dernière couche sera toujours dotée d'une non-linéarité **softmax**.

Vous entraînerez votre réseau de neurones avec une descente de gradient stochastique en mini-batch, en utilisant l'entropie croisée comme fonction de coût et l'optimiseur Adam.

La classe `Trainer` est initialisée avec les paramètres suivants :

- Une clé de l'ensemble {'mlp', 'cnn'} désignant le type de réseau.
- `net_config` : une structure de données définissant l'architecture du réseau de neurones. Il doit s'agir d'un NamedTuple `NetworkConfiguration`, tel que défini dans le fichier de solution. Les différents champs du tuple sont des tuples d'entiers, précisant les hyperparamètres pour les différentes couches cachées du réseau, à savoir : `n_channels`, `kernel_sizes`, `strides`, `padding` et `dense_hiddens`.
  - Si `network_type = 'cnn'`, un CNN doit être créé. Les 4 premiers champs spécifient la topologie des couches convolutives, tandis que `dense_hiddens` spécifie le nombre

de neurones pour les couches cachées entièrement connectées pour la dernière partie du réseau. Par exemple, en appelant `NetworkConfiguration(n_channels=(16,32), kernel_sizes=(4,5), strides=(1,2), paddings=(1,0), dense_hiddens=(256, 256))` on créera un CNN avec deux couches convolutives : la première couche aura 16 canaux, un 4×4 kernel, une "stride" de 1×1 et un remplissage de 1×1 ; la deuxième couche aura 32 canaux, un noyau de 5×5, une "stride" de 2×2 et un rembourrage de 0×0. Le dernier attribut signifie qu'il y a deux couches cachées entièrement connectées avec 256 neurones chacune dans la partie finale du réseau.

  - Si `network_type = 'mlp'`, seul l'attribut `dense_hiddens` est utilisé, et un réseau avec une couche d'entrée, deux couches cachées avec 256 neurones et une couche de sortie doit être construit.

- `n_classes`: le nombre de classes pour ce problème de classification (et donc aussi le nombre de neurones dans la couche de sortie)
- `lr`: le taux d'apprentissage utilisé pour entraîner le réseau de neurones avec Adam.
- `batch_size`: la taille des batchs pour Adam.
- `activation_name`: une chaîne de caractères décrivant la fonction d'activation à utiliser. Elle peut être `"relu"`, `"sigmoid"`, ou `"tanh"`. On vous rappelle les 3 fonctions d'activation:
  - $\text{RELU}(x) = \max(0, x)$
  - $\sigma(x) = \frac{1}{1+e^{-x}}$
  - $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
- `normalization`: un booléen mis à "vrai" si les données doivent être normalisées.

La fonction `__init__` vous est donnée. En plus du chargement du jeu de données et l'initialisation des variables de classe, cette fonction initialise un dictionnaire de logs d'entraînement, qui contient des informations sur les coûts et les précisions sur les ensembles d'entraînement et de validation pendant l'entraînement.

1. [3 points] Les librairies de différenciation automatique tels que PyTorch sont utiles pour la création des réseaux de neurones, mais les outils qu'elles fournissent sont plus généraux que cela. Elles peuvent calculer le gradient de (presque) n'importe quelle fonction composable. Pour montrer cela, complétez la fonction `gradient_norm`, qui prend une fonction (nous supposons qu'elle est composée de primitives PyTorch) et une liste de tenseurs en entrées, et renvoie la norme euclidienne du gradient de la sortie du fonction par rapport à la liste des entrées. **Notez que la fonction prendra la liste de tenseurs décompressée en entrée, comme dans `function(*tensor_list)`.**

2. [2 points] Pour toucher un peu plus aux capacités de différenciation automatique, complétez maintenant la fonction `jacobian_norm` qui, prend une fonction en argument, ainsi qu'un (unique) tenseur, et devrait sortir la norme de Frobenius de la matrice jacobienne de cette fonction évaluée au tenseur d'entrée fourni comme deuxième argument.

3. [1 points] Complétez la méthode `Trainer.create_activation_function`, qui doit accepter une chaîne de l'ensemble {`"relu"`, `"tanh"`, `"sigmoid"`} comme argument et renvoie un objet `torch.nn.Module` implémentant cette fonction d'activation spécifique.

4. [5 points] Cette question concerne l'écriture d'une fonction constructeur pour un MLP. Vous devrez compléter la fonction `Trainer.create_mlp`, qui a comme arguments la dimension d'entrée pour la première couche, un objet `NetworkConfiguration` définissant la structure du réseau, le nombre de classes, une fonction d'activation sous la forme d'un objet `torch.nn.Module`. La fonction d'activation fournie doit être appliquée après chaque couche, à l'exception de la dernière couche, qui doit avoir une activation softmax. La fonction doit retourner un objet `torch.nn.Module` avec l'architecture souhaitée ; veuillez noter que le réseau doit s'attendre à un tenseur d'image non aplati en entrée et l'aplatir dans le cadre de son traitement interne. **Nous nous attendons à ce que les couches entièrement connectées soient des instances de `torch.nn.Linear`. Nous vous suggérons également d'utiliser un conteneur `torch.nn.Sequential`. Par exemple, un `hidden_sizes` de longueur 3 devrait impliquer un MLP de 4 couches `Linear` au total ; trois avec la fonction d'activation prescrite et une en sortie avec une activation softmax.**

5. [5 points] Cette question concerne l'écriture d'une fonction constructeur pour un CNN. Vous devrez compléter la fonction `Trainer.create_cnn`, qui a comme arguments le nombre de canaux de l'image d'entrée, un objet `NetworkConfiguration`, le nombre de classes et une fonction d'activation sous forme de `torch.nn.Module`. La fonction d'activation doit être appliquée après chaque couche convolutive et entièrement connectée, à l'exception de la dernière, qui doit avoir une activation softmax. Après la fonction d'activation de chaque couche convolutive, une couche `torch.nn.MaxPool2d(kernel_size=2)` doit être appliquée ; la dernière couche convolutive ne devrait pas avoir de pool maximum et être plutôt suivie d'un `torch.nn.AdaptiveMaxPool2d((4, 4))` et d'un `torch.nn.Flatten()`, juste avant la couche complètement connectée, pour s'assurer que le réseau est agnostique (jusqu'à un certain degré) à la taille d'entrée. La fonction doit retourner un objet `torch.nn.Module` avec l'architecture souhaitée. **Encore une fois, nous nous attendons à ce que les couches convolutives et entièrement connectées soient des instances de `torch.nn.Conv2d` et `torch.nn.Linear`. Nous vous suggérons également d'utiliser un conteneur `torch.nn.Sequential`.**

6. [2 points] Pour répondre à cette question, vous devez compléter la méthode `Trainer.normalize`. Cette fonction doit prendre les ensembles d'apprentissage, de validation et de test comme arguments et renvoyer leur version normalisée. La normalisation devrait se produire en soustrayant des matrices X la moyenne par feature des échantillons d'apprentissage et en les divisant par l'écart type par feature des échantillons d'apprentissage.

7. [2 points] Implémentez la fonction `NN.one_hot`. Cette fonction doit transformer un tensor `y` de taille $batch\_size$ contenant des valeurs entières entre 0 to $n\_classes - 1$ en une matrice de taille $batch\_size \times n\_classes$ contenant les one-hot encodings des vraies étiquettes `y`.

   Par exemple, si le nombre de classes est 5, alors `one_hot(torch.tensor([1, 0, 3, 4]))` devrait retourner la matrice suivante:

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

8. [4 points] Complétez la fonction `Trainer.compute_loss_and_accuracy` qui prend en entrée une matrice d'échantillons et une matrice d'étiquettes one-hot. La fonction doit renvoyer la

perte d'entropie croisée pour le réseau actuel, ainsi que la précision sur ce batch. Attention, la valeur de perte renvoyée doit être un `torch.Tensor` gardant une trace des calculs qui y ont conduit, afin que la différenciation automatique puisse calculer les gradients avec l'algorithme de rétropropagation.

Pour éviter les erreurs numériques, vous devez pré-traiter les probabilités en sortie du réseau en tronquant les très petites valeurs (proche de 0) à une valeur fixe `Trainer.epsilon` et les très grandes valeurs (càd proche de 1) à $1-$ `Trainer.epsilon`.

9. [2 points] Complétez la fonction `Trainer.compute_gradient_norm`, qui devrait prendre un `torch.nn.Module` en entrée et sortir la norme euclidienne du vecteur de gradient de ses paramètres aplati. Le gradient peut être trouvé dans des buffers spécifiques, qui sont remplis lors de la rétropropagation. Ils sont utiles pour deboguer les réseaux de neurones.

10. [6 points] Pour cette question, vous devez utiliser les fonctions précédentes que vous avez implémentées.

    Complétez la fonction `Trainer.training_step`. Cette fonction implémente une seule étape d'apprentissage, en prenant un batch des données et d'étiquettes comme entrées. Il sera ensuite utilisé à l'intérieur de la fonction `train_loop` pour avoir une procédure d'entraînement complète. Votre fonction doit calculer le gradient de la fonction de perte de votre réseau actuel et le mettre à jour à l'aide de l'optimiseur. De plus, la méthode doit retourner la norme du gradient de la fonction de perte par rapport au paramètre du réseau évalué à ce batch.

11. [3 points] Complétez la fonction `Trainer.evaluate`. Cette fonction doit retourner le coût moyen et la précision moyenne sur un ensemble. Vous pouvez utiliser les fonctions qui vous sont déjà données.

2 Expérimentation sur le jeu de données FashionMNIST [(bonus) 20 points]

Dans cette partie du devoir, vous allez faire quelques expérimentations, avec vos `Trainer`, sur l'ensemble de données FashionMNIST.

Le jeu de données sera automatiquement téléchargé en utilisant la méthode `Trainer.load_dataset` que nous avons déjà écrite, en utilisant le module `torchvision`. Il sera téléchargé dans le répertoire spécifié via l'argument `datapath` (`./data` par défaut).

L'ensemble d'entraînement de FashionMNIST se compose de 60000 images de dimension 28×28 en noir et blanc, étiquetées de 10 classes de vêtements (par exemple, T-shirts, pantalons). Nous allons utiliser un ensemble de test de 8000 images et un ensemble de validation de 2000 images. Chaque image est représentée par un tenseur $1 \times 28 \times 28$, où la première dimension représente le canal unique composant l'image. Les variables `self.train, self.valid, self.test` initialisées dans le constructeur du `Trainer` contiendront le jeu de données. Chacune de ces trois variables est un tuple. Par exemple, `self.valid[0]` est un tenseur PyTorch à 4 dimensions de taille $2000 \times 1 \times 28 \times 28$, et `self.valid[1]` est un Tenseur PyTorch de taille 2000 contenant les étiquettes des images du jeu de validation 2000, de 0 à 9. Vous pouvez vérifier si les tenseurs ont les bonnes tailles.

Les réponses aux questions de cette section, ainsi que les chiffres requis, doivent être

écrits dans un rapport que vous soumettrez à Gradescope comme partie pratique du devoir (séparément de la partie théorique).

1. [(bonus) 5 points] Cette question vous permettra d'explorer certaines propriétés des CNN. La convolution dans les réseaux de neurones induit de puissants biais inductifs via des **interactions éparses**, **partage du poids** et **représentations équivariantes**.

   Nous allons maintenant étudier cette troisième propriété. Pur une fonction $f : X \to Y$, l'équivariance à une transformation g signifie que $f(g(x)) = g(f(x))$, c'est-à-dire que l'application de cette transformation à l'entrée de la fonction équivaut à l'appliquer à la sortie. La convolution est équivariante à certains types de transformations seulement.

   Vous pouvez trouver une méthode `test_equivariance` incomplète dans l'objet `Trainer`. Il implémente déjà deux fonctions qui déplacent et font pivoter une image respectivement, ainsi qu'un très petit réseau de neurones à convolution uniquement. Utilisez cette fonction pour vous aider à générer les figures suivantes, à inclure dans votre rapport:

   - Une figure montrant l'image originale, qui est la première image `self.train[0][0]` de l'ensemble d'apprentissage ;
   - Une figure montrant les features de sortie du petit `CNN` lorsqu'on lui donne cette première image en entrée ;
   - Une figure montrant l'image de la différence absolue pixel par pixel entre (1) les features de sortie décalées étant donné l'entrée non décalée, et (2) la sortie non décalée du CNN étant donné l'entrée décalée. Les décalages doivent être effectués en utilisant la transformation `shift` fournie;
   - Une figure similaire, mais cette fois avec rotation. En d'autres termes, montrez la différence absolue pixel par pixel entre la sortie pivotée du CNN en fonction de l'image d'origine et la sortie non traitée du CNN en fonction de l'image pivotée. Utilisez la transformation `rotation` fournie, sans autre traitement.

   Ecrivez quelles sont vos observations sur les propriétés d'équivariance des couches convolutives.

   (a) A quels types de transformations les couches convolutives sont-elles équivariantes ?

   (b) Pourquoi l'équivariance à certaines transformations serait-elle plus utile que l'équivariance à d'autres transformations pour l'apprentissage ? Cela dépend-il uniquement de l'algorithme d'apprentissage ou de la distribution des données également ?

2. [(bonus) 3 points] Il vous est maintenant demandé d'entraîner un MLP avec 2 couches cachées, de taille 256 et 256 respectivement sur le jeu de données FashionMNIST, pour 50 époques, et une taille de batch 128. Utilisez la fonction d'activation ReLU. Pour des raisons de reproductibilité, **veuillez ne pas modifier la graine qui est déjà définie dans le fichier de solution**.

   Lorsqu'il n'est pas spécifié, laissez la valeur par défaut pour les arguments du `Trainer`. Incluez dans votre rapport les chiffres suivants et répondez aux questions suivantes :

   - Définissez `normalization=False` et générez un chiffre avec la précision de validation w.r.t. des époques croissantes pour les valeurs des taux d'apprentissage dans l'ensemble $\{0.01, 1 \times 10^{-4}, 1 \times 10^{-8}\}$;

- Maintenant, définissez `normalization=True` et générez le chiffre pour les mêmes taux d'apprentissage.
  (a) Quel est l'effet de la normalisation ?
  (b) Cela aide-t-il à la convergence ? Pourquoi, ou pourquoi pas?
  (c) Quels sont les effets des taux d'apprentissage très petits ou très grands ?

À des fins de vérification, la meilleure combinaison de taux d'apprentissage et de normalisation devrait permettre d'atteindre des précisions supérieures à 80 %.

3. [(bonus) 5 points] Dans les trois questions suivantes, il vous sera demandé d'expérimenter en profondeur les MLP et les CNN et de comprendre leur comportement. Lorsqu'on vous demande de créer des réseaux avec approximativement le même nombre de paramètres, vous devez vous assurer que le nombre de paramètres diffère d'au plus 10000.

   (a) Combien de paramètres (scalaires) votre réseau de neurones doit-il apprendre?
   (b) Entraînez un réseau de neurones avec approximativement le même nombre de paramètres que celui que vous avez formé, mais des couches cachées avec des neurones à 64 seulement. Indice : il y aura plusieurs couches !
   (c) Tracez les précisions d'entraînement et de validation dans le même tracé que le MLP précédent formé avec les hyperparamètres par défaut. Mettez ce chiffre dans votre rapport ;
   (d) Tracez maintenant l'évolution de la norme de gradient (qui devrait déjà être enregistrée) pendant l'entrainement pour les deux architectures et mettez ce chiffre dans votre rapport ;
   (e) Le comportement de la norme du gradient explique-t-il la performance ?

4. [(bonus) 2 points] Entraînez maintenant un CNN avec approximativement le même nombre de paramètres, avec les contraintes suivantes. Utilisez trois couches convolutives cachées avec un nombre de filtres de $(16, 32, 45)$, une "stride" de 1 et un rembourrage de 0 à chaque couche. Utilisez deux couches cachées finales entièrement connectées avec des neurones à 256 chacune. Trouvez la taille de noyau appropriée pour conserver approximativement le même nombre de paramètres.

   (a) Mettez dans le rapport un chiffre montrant les performances de ce CNN et du MLP original (non profond) ;
   (b) Lequel fonctionne le mieux ? Que pouvez-vous dire des capacités de généralisation des deux architectures ?

5. [(bonus) 5 points] Expérimentons maintenant avec la signification de "profondeur" et de "largeur" dans les réseaux de neurones convolutifs. Comme vu précédemment, les interactions éparses ("sparse") sont l'une des caractéristiques de la convolution appliquée aux réseaux de neurones : cela signifie que, grâce à la localité des noyaux, seule une partie des features de l'échantillon interagissent. Par contre, dans les MLPs, la multiplication matricielle permet une interaction dense entre les entrées et les paramètres. Nous allons maintenant casser cette propriété de deux manières opposées.

   (a) Imaginez un **CNN profond** avec à peu près le même nombre de paramètres que l'original, 4 couches avec un nombre de filtres de $(8, 16, 32, 64)$, des "strides" de 1 et

0 de rembourrage. Cette fois, vous utiliserez un `kernel_size` de 1 dans toutes les couches convolutives, et toujours un MLP final à deux couches cachées de 256 neurones. Quelle est la relation entre le type de couches convolutives utilisées par ce réseau et les couches entièrement connectées ?

(b) Imaginez un **CNN pas profond** avec une seule couche convolutive, avec un `kernel_size` égal à la taille de l'image d'entrée (28×28) et un nombre de canaux dans cette couche tels que le nombre de paramètres est approximativement le même que celui des réseaux précédents. Considérez les autres hyperparamètres comme ceux du CNN précédent ("stride" de 1, zéro rembourrage, deux couches cachées finales de 256 neurones). Quelle est la relation entre le type de couches convolutives utilisées par ce réseau et les couches entièrement connectées ?

(c) Maintenant, entraînez le **CNN profond** et le **CNN pas profond** tels que définis ci-dessus pour 50 époques avec une taille de batch de 128 et mettez dans votre rapport un chiffre comparant les précisions de validation et d'entraînement de ces trois types de CNN (y compris le CNN original de la question précédente). Comment se comparent leurs performances générales et leurs capacités de généralisation ? Et quelle est votre hypothèse sur leurs différents comportements ?