**IFT 6390 Fundamentals of Machine Learning**
**Ioannis Mitliagkas**

# Devoir 3 - Partie pratique

- Ce devoir doit être déposé sur Gradescope et doit être fait en équipe de 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 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 sur 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**. Pour le rapport il est recommandé d'utiliser un Jupyter notebook, en écrivant les formules mathématiques avec MathJax et en exportant vers pdf. Vous pouvez aussi écrire votre rapport en LaTeX; LyX; Word. Dans tout les cas, exportez votre rapport vers un fichier pdf que vous enverrez. Vous êtes bien sûr encouragés à vous inspirer de ce qui a été fait en TP.
- Vous devez soumettre vos solutions sur Gradescope en utilisant le devoir intitulé `Devoir 3 / Homework 3 - Pratique/Practical - 6390A/B` pour le code et `Devoir 3 / Homework 3 - Pratique/Practical - Rapport/Report - 6390A/B` pour le rapport.

Vous devez travailler sur le modèle `solution.py` du répertoire et compléter les fonctions basiques en utilisant numpy, PyTorch et python.

## 1 Entrainement de réseaux de neurones avec la différentiation automatique [25 points]

Cette partie consiste en la mise en place d'une classe `Trainer` pour entraîner des réseau de neurones pour la régression univariée. 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). Vous entraînerez votre réseau de neurones avec une descente de gradient stochastique en mini-batch, en utilisant l'erreur quadratique moyenne 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` et `dense_hiddens`.
  - Si `network_type = 'cnn'`, un CNN doit être créé. Les 3 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), dense_hiddens=(256, 256))` on créera un CNN avec deux couches convolutives : la première couche aura 16 canaux, un $4 \times 4$ kernel et une "stride" de $1 \times 1$ ; la deuxième couche aura 32 canaux, un noyau de $5 \times 5$ et une "stride" de $2 \times 2$. 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.
- `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:
  - $\text{RELU}(x) = \max(0, x)$
  - $\sigma(x) = \frac{1}{1+e^{-x}}$
  - $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

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 métriques sur les ensembles d'entraînement et de test pendant l'entraînement.

1. [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.

2. [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 et 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 ne devrait pas avoir une activation. 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. Le modèle doit avoir un neurone de sortie, qui est la prédiction de la régression.

**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 sans activation.**

3. [6 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` 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 ne devrait pas avoir une activation. 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. Le modèle doit avoir un neurone de sortie, qui est la prédiction de la régression. **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`.**

4. [4 points] Complétez la fonction `Trainer.compute_loss_and_mae` qui prend en entrée une matrice d'échantillons et une matrice d'étiquettes. La fonction doit renvoyer la perte d'erreur quadratique moyenne pour le réseau actuel, ainsi que l'erreur absolue moyenne sur ce batch. Attention, les valeurs de perte et d'erreur absolue moyenne renvoyées doivent être des `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.

5. [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.

6. [3 points] Complétez la fonction `Trainer.evaluate`. Cette fonction doit retourner le coût moyen et l'erreur absolue moyenne sur un ensemble. Vous pouvez utiliser les fonctions qui vous sont déjà données.

## 2 Expérimentation sur le jeu de données FashionMNIST Tournée [15 points]

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

Le jeu de données peut être chargée en utilisant la méthode `Trainer.load_dataset` fournie.

L'ensemble d'entraînement de FashionMNIST Tournée se compose de 60000 images de dimension $28 \times 28$ en noir et blanc de vêtements (par exemple, T-shirts, pantalons). Ces images ont subi une rotation aléatoire, et la tâche consiste à prédire l'angle de rotation. Nous allons utiliser un ensemble de test de 2000. 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.test` initialisées dans le constructeur du `Trainer` contiendront le jeu de données. Chacune de ces deux variables est un tuple. Par exemple, `self.test[0]` est un tenseur PyTorch à 4 dimensions de taille $2000 \times 1 \times 28 \times 28$, et `self.test[1]` est un Tenseur PyTorch de taille $2000 \times 1$ contenant les étiquettes des images du jeu de test 2000. 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. [10 points] Entraînez un MLP avec 2 couches cachées, de taille 128 et 128 respectivement sur le jeu de données FashionMNIST Tournée, 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 la figure et la réponse à la question suivante :

   - Générez une figure avec l'erreur absolue moyenne du test en fonction d'époques croissantes pour des valeurs de taux d'apprentissage dans l'ensemble $\{0.01, 1\times10^{-4}, 1\times10^{-8}\}$. Quels sont les effets des taux d'apprentissage très petits ou très grands ?

2. [5 points] Entraînez un CNN sur le jeu de données FashionMNIST Tournée pour 50 époques avec une taille de batch 128. Utilisez la fonction d'activation ReLU. Utilisez trois couches convolutionnelles cachées avec un nombre de filtres de $(16, 32, 45)$, des noyaux de taille 3, et un "stride" de 1. Utilisez une dernière couche cachée entièrement connectée avec 128 neurones. 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 la figure et la réponse à la question suivante :

   - Générer une figure avec l'erreur absolue moyenne de test en fonction de l'augmentation des époques pour le CNN ci-dessus.

## 3 Représentations Équivariantes [10 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 trans-

formation 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.

1. A quels types de transformations les couches convolutives sont-elles équivariantes ?
2. 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 et la tâche également ?