

**Date de remise: 17 octobre (23h00), 2022**

### Instructions

- *Ce devoir est intensive – commencez bien à l’avance !*
- *Pour toutes les questions, montrez votre travail.*
- *Vous trouverez le devoir pratique sous forme de Jupyter Notebook sur Google Colab. Vous devez compléter des parties du fichier `solution.py` (du Github repo). Ce fichier devra être chargé sur le Gradescope du cours.*
- *Vous devez soumettre votre rapport (PDF) sur Gradescope. Votre rapport doit contenir les réponses aux questions 3.4, 3.5, 3.6, 3.7 et 3.8. Vous n’avez pas besoin de soumettre le code pour ces questions, seulement le rapport suffira.*
- *Les auxiliaires d’enseignement pour ce devoir sont **Nanda Harishankar Krishna and Sarthak Mittal**.*

**Lien (Notebook):** [Cliquez ici](#).

### **Question 1 (30). (Implémenter Perceptron multicouche avec NumPy)**

Considérez un Perceptron multicouche avec deux couches cachées avec des unités cachées  $h^1 = 128$  et  $h^2 = 64$ . Le nombre de fonctionnalités des données d’entrée est  $h^0 = 784$ . La sortie du réseau neural est paramétrée par un softmax de  $h^3 = 10$  classes. Créez un Perceptron multicouche avec ces spécifications (n’oubliez pas d’inclure les biais!). Implémenter la propagation avant et arrière du Perceptron multicouche dans NumPy sans utiliser les cadres d’apprentissage profond qui fournissent une différenciation automatique. Aussi écrire du code pour calculer la perte de *Entropie croisée*, qui est utilisé comme critère d’optimisation. Enfin, écrivez une fonction qui met à jour les paramètres du réseau. Notez que vous n’avez pas besoin de former le modèle pour cette question – Il suffit d’écrire du code pour les fonctions mentionnées.

Les pièces suivantes seront automatiquement calibrées:

- (1.1) Ecrivez la fonction `initialize_weights`. Ceci initialise les poids du Perceptron multicouche avec les valeurs échantillonnées à partir d’une distribution uniforme  $\mathcal{U}\left[-\frac{1}{\sqrt{h_0}}, \frac{1}{\sqrt{h_0}}\right]$ . Notez que nous avons déjà attribué des 0 aux biais, vous n’avez pas besoin de les modifier.
- (1.2) Ecrire toutes les fonctions d’activation données, par exemple `relu`, `tanh`, `sigmoid` and `softmax`, et la fonction sélecteur. `activation` qui sélectionne la fonction d’activation spécifiée par la chaîne d’entrée (`activation_str`). Notez que si l’argument `grad` est `Vrai`, nous nous attendons à ce que la fonction retourne le gradient de la fonction d’activation par rapport à son entrée.
- (1.3) Écrire la `forward` fonction de résussite, qui prend une entrée dans le Perceptron multicouche et retourne un dictionnaire `cache` contenant des pré-activations (Zs) et activations (As) pour chaque couche. Notez que vous utiliserez les poids et les biais initialisés précédemment pour ces calculs.

- (1.4) Écrire la `loss` fonction (Entropie croisée pour la classification multi-classes).
- (1.5) Écrire la `backward` fonction de passage. nous supposons que l'entrée à ce. fonction est le vrai label associé à l'exemple d'entrée considéré dans la `forward` passe. Vous pouvez calculer le gradient de la loss par rapport à la couche de sortie en supposant que la perte est multi-classe d'entropie croisée. La fonction doit retourner un dictionnaire `grads` contenant le gradient par rapport aux activations à chaque couche et les paramètres (poids et biais).
- (1.6) Écrire la fonction `update` qui prend le gradient pour mettre à jour les paramètres.

**Answer 1.** Voir `assignment1_solution.py` pour les réponses à toutes les questions, (1.1) – (1.6).

### Question 2 (20). (Implémentation des couches CNN avec NumPy)

Considérez deux couches, une couche de convolution de taille de filtre ( $3 \times 3$ ) et stride 1 comprenant 64 unités, et une couche de mise en commun maximale de la taille du filtre ( $2 \times 2$ ). Les entrées de la couche de convolution sont chacune de taille ( $1 \times 32 \times 32$ ) (canal unique) tandis que l'entrée de la couche de regroupement maximum est la taille de la sortie de la couche de convolution. Vous allez coder les passes avant et arrière pour ces deux couches en utilisant NumPy. Vous pouvez vous référer à [ces diapos](#) pour plus de détails.

- (2.1) Écrire les `initialize_weights` et `forward` fonctions pour la couche de convolution. L'entrée de `forward` est une matrice de taille ( $n \times 1 \times 32 \times 32$ ) (par exemple  $n$  images à canal unique de taille  $(32 \times 32)$ ) et la sortie est le résultat de l'opération de convolution.
- (2.2) Écrire la `backward` fonction passe pour la couche de convolution. Nous supposons que l'entrée à cette fonction est la dérivée de la perte par rapport à la sortie de la couche. la sortie de la fonction est le gradient de la perte par rapport aux paramètres de la couche.
- (2.3) Écrire la `forward` fonction passe pour la couche de Max-Pooling.
- (2.4) Écrire la `backward` fonction passe pour la couche de Max-Pooling. Nous supposons que l'entrée à cette fonction est la dérivée de la Loss par rapport à la production de la couche. La sortie de la fonction est le gradient de la perte par rapport à l'entrée à la couche.

**Answer 2.** Voir `assignment1_solution.py` pour les réponses à toutes les questions, (2.1) – (2.4).

### Question 3 (50). (Implémentation d'un CNN et comparaison avec les MLP utilisant PyTorch)

Dans cette question, vous utiliserez PyTorch pour implémenter un ResNet-18 jusqu'à ses blocs de composants et l'entraîner à classer les images à partir de l'ensemble de données CIFAR-10. L'architecture a été légèrement modifiée pour mieux s'adapter à l'ensemble de données CIFAR-10. Vous visualiserez les filtres dans le CNN et les cartes de caractéristiques pour une image, et comparer le CNN à un MLP en termes de performance et de temps de formation.

- (3.1) Complétez les `__init__`, `activation`, `residual` and `forward` Méthodes dans la `ResidualBlock` classe. Vous pouvez vous référer à la figure ci-dessous pour la mettre en œuvre.

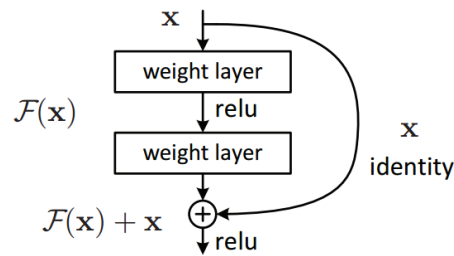


FIGURE 1 – Residual Block (source: “Deep Residual Learning for Image Recognition”, He et al., CVPR 2016)

(3.2) Complétez les `__init__`, `_create_layer` et `forward` méthodes dans la `ResNet18` classe. l’entrée à la `forward` méthode est un lot d’entrées au réseau. La figure ci-dessous montre l’architecture de ResNet-18.

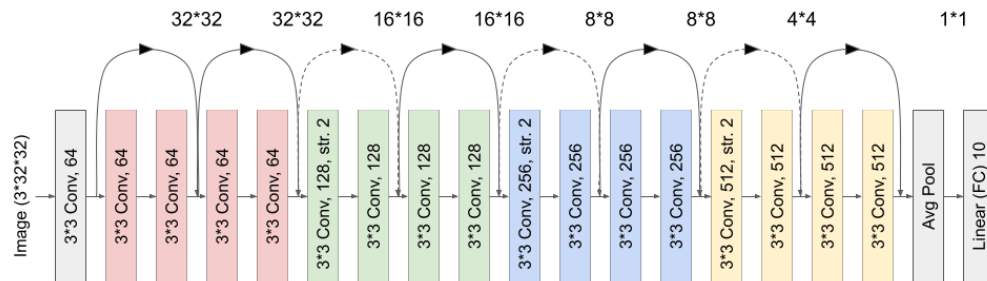


FIGURE 2 – ResNet-18 Architecture

### Détails d’implémentation:

- Les lignes pointillées représentent les connexions résiduelles impliquant un échantillonnage descendant. Dans de tels cas, la sortie de **residual** n’est pas l’identité, mais le résultat d’une convolution avec la taille du noyau (1 × 1), stride 2, bias=False, padding=0.
- Les nombres mentionnés en haut sont la hauteur et la largeur des cartes de caractéristiques après avoir traversé chaque bloc.
- Utilisez padding=1 pour toutes les convolutions sauf la convolution de sous-échantillonnage dont le résiduelle n’a pas besoin de padding (c’est-à-dire padding=0). De même, bias=False pour toutes les couches de convolution.
- Vous devez inclure un `BatchNorm2d` après chacune convolution, avant l’activation.
- La Stride pour les couches de convolution est 1 sauf indication contraire (voir “str. 2” dans la figure, cela signifie que la foulée est de 2 pour ces couches de convolution spécifiques).
- Vous n’avez pas besoin d’appliquer softmax à la couche de sortie car `nn.CrossEntropyLoss` s’en occupe implicitement.

- (3.3) Écrire la fonction `train_loop` et former le modèle sur l'ensemble de données CIFAR-10. Aussi complétez `valid_loop`, qui sert à valider la performance du modèle. Effectuer la validation après chaque époque de formation.
- (3.4) Tracer des courbes de perte d'entraînement, de précision d'entraînement, de perte de validation et l'exactitude de la validation à travers les époques et l'inclure dans votre rapport (ReLU activation, `xavier_normal` initialization).
- (3.5) Visualisez quelques filtres de la première et de la dernière couche de convolution dans le modèle formé. Inclure les images dans votre rapport et commentez les fonctionnalités apprises par les premier et dernier calques de convolution. En quoi diffèrent-elles ?
- (3.6) Visualiser les cartes de caractéristiques à partir des première et dernière couches de convolution (pour quelques filtres) dans le modèle formé pour l'image fournie dans la variable `vis_image`. Inclure les images dans votre rapport . Commentaires sur les caractéristiques de l'entrée détectée par des unités spécifiques qui pourraient aider à classer l'image.
- (3.7) Considérez les fonctions d'activation: (i) ReLU et (ii) tanh, et les schémas d'initialisation des poids: (i) une Glorot uniforme (`init.xavier_uniform_`), (ii) Glorot normale (`init.xavier_normal_`) and (ii) He normal (`init.kaiming_normal_`). Effectuez une recherche par grille sur ces hyperparamètres. Signaler l'ensemble des hyperparamètres qui donnent la plus grande précision de validation, et fournir également les valeurs de l'exactitude et de la perte de validation.
- (3.8) Construire un PLP avec des paramètres approximativement égaux comme le ResNet-18 et l'entraîner sur l'ensemble de données CIFAR-10. Inclure un tableau dans votre rapport comparant le MLP et le CNN – énumérer le nombre de paramètres, l'exactitude de la validation, la perte de validation et le temps de formation (nombre d'époques 25). Quel modèle fonctionne le mieux et pourquoi le pensez-vous ?

**Answer 3.** Voir `assignment1_solution.py` pour les réponses aux questions (3.1) – (3.3).

(3.4) Voici les courbes de perte et de précision :

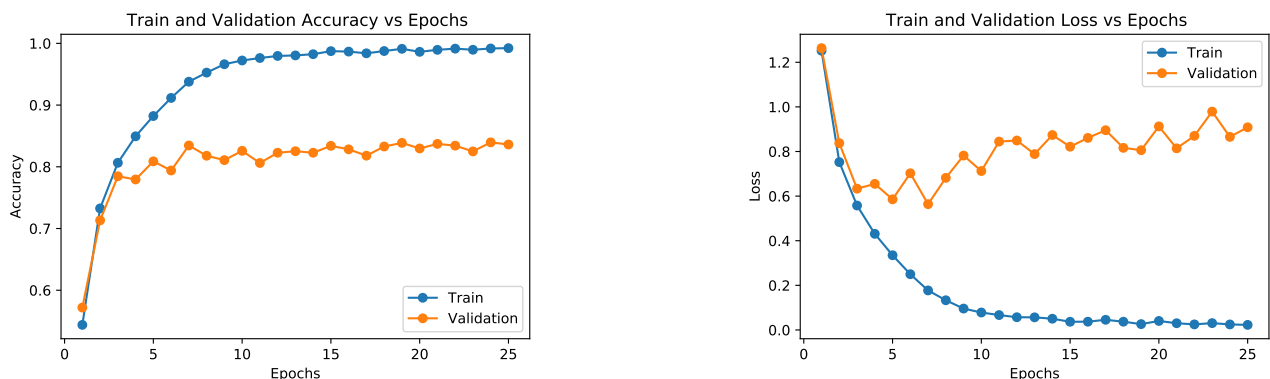


FIGURE 3 – Courbes de perte et de précision (entraînement et validation) pour le ResNet-18 avec initialisation `xavier_normal` et activation ReLU.

*Remarque : Nous avons calculé la moyenne de la perte sur le nombre de lots (la taille des lots est la même pour les ensembles d'entraînement/validation/test, donc cela convient), mais*

*il est également possible de calculer la moyenne sur le nombre d'exemples. Nous n'avons pas déduit de points si vous n'avez pas pris la moyenne, mais il est toujours préférable de le faire pour une comparaison correcte, car le nombre d'exemples varie pour les ensembles d'entraînement/validation/test.*

- (3.5) Vous pouvez choisir quelques filtres retournés par les méthodes `get_first_conv_layer_filters` et `get_last_conv_layer_filters` et les tracer avec `matplotlib` ou un autre module de visualisation. Les premières couches apprennent des caractéristiques comme les bords et les contours (bas niveau), et les dernières couches apprennent des caractéristiques plus complexes (haut niveau). Ceci peut être facilement observé pour la première couche en regardant les cartes de caractéristiques. Pour la dernière couche, l'interprétation visuelle est plus difficile, mais nous pouvons comprendre que les informations de l'image ( $32 \times 32$ ) ont été comprimées à une taille beaucoup plus petite, c'est-à-dire ( $4 \times 4$ ), ce qui signifie que chaque pixel de la carte des caractéristiques représente un concept plus abstrait jusqu'à la dernière couche (couche entièrement connectée), où chaque unité représente une classe entière d'objets. Nous voyons dans les figures que les valeurs prennent une plus grande gamme pour les filtres de la première couche, par rapport à la dernière couche.

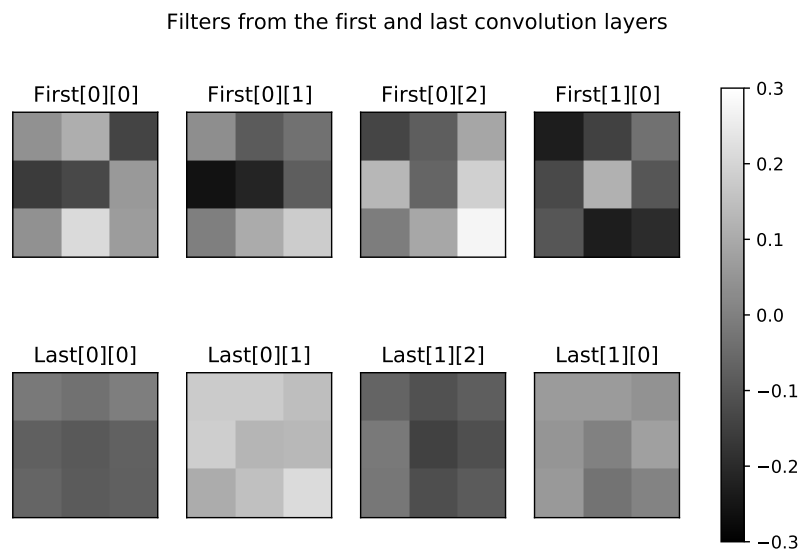


FIGURE 4 – Filtres de la première et de la dernière couche de convolution du ResNet-18.

*Remarque : Il est important de fournir la colormap/colorbar, car nous ne pouvons pas interpréter les valeurs sans cela.*

- (3.6) Vous pouvez obtenir les cartes de caractéristiques en retraçant les étapes de la méthode `forward`, en utilisant les “forward hooks” ou le module `feature_extraction` de `torchvision`.

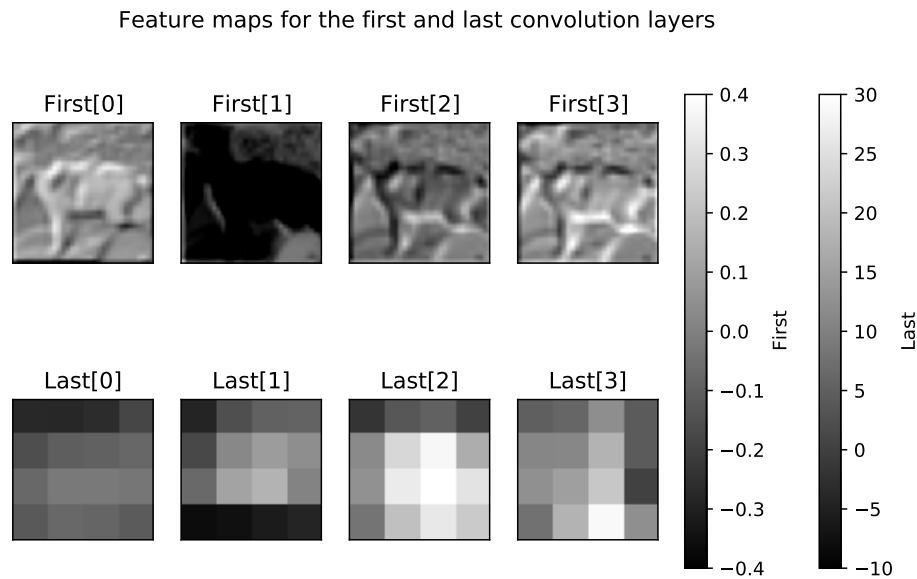


FIGURE 5 – Cartes de caractéristiques pour les première et dernière couches de convolution du ResNet-18 pour `vis_image`.

*Remarque : Il est important de fournir la colormap/colorbar, mais nous n’avons pas déduit de points si cela n’est pas fourni (pour cette question).*

- (3.7) Avant de discuter des résultats, nous noterons qu’il est important de réinitialiser la “seed aléatoire” (*random seed*) à la même valeur, avant d’initialiser et d’entraîner chaque configuration, pour des raisons de reproductibilité. Cependant, nous n’avons pas considéré cet aspect lors de la notation car notre intention principale était de montrer que le choix de la fonction d’activation a un impact plus important sur les performances que l’initialisation *pour ces configurations*. Nous observons que les réseaux utilisant ReLU surpassent largement les réseaux utilisant tanh. Les réseaux ReLU ont une précision de validation  $\sim 84\%$  alors que les réseaux tanh obtiennent  $\sim 74\%$ .
- (3.8) Le ResNet-18 a 11,173,962 paramètres. Vous pouvez construire *n’importe quelle* architecture MLP ayant un nombre similaire de paramètres. Vous constaterez que le MLP est beaucoup moins performant que le ResNet-18, c’est-à-dire qu’il présente une perte de validation plus élevée et une précision de validation beaucoup plus faible. Cependant, l’apprentissage du MLP prend beaucoup moins de temps que celui du CNN, car les calculs sont plus simples. Le CNN est plus performant en raison de son équivariance de *translations* (*shifts*), et il est conçu pour travailler avec des données spatiales telles que des images.