Due Date: 22nd February, 24:00

Instructions

- *Ce devoir est difficile - commencez le bien en avance.*
- *Pour toute les questions, montrez votre travail!*
- *Soumettez votre rapport (PDF) et votre code électroniquement via la page Gradescope du cours.*
- *Vous pouvez utiliser le notebook Jupyter* `main.ipynb` *pour lancer vos expériences (facultatif).*
- *Pour les questions ouvertes (c.-à-d. les expériences où il b'y a aucun test associé), vous n'avez pas à envoyer de code - le rapport suffit.*
- *Les TAs pour ce devoir sont* ***Yuchen Lu*** *(IFT6135B) and* ***Ghait Boukachab*** *(IFT6135A)*

Sommaire:

Dans ce devoir, vous allez développer plusieurs modèles de classification d'images sur les données CIFAR-10.

CIFAR-10 est constituée de 60000 32×32 images couleur réparties en 10 classes, et chaque classe compte 6 000 images. Nous avons donc 50000 images d'entraînement et 10000 images de test. Vous disposez d'une classe de données sur PyTorch (torch.utils.data.Dataset) intitulée CIFAR10 à partir de la librarie torchvision. les répartitions train, valid et test sont fournies. Tout au long de ce devoir, la forme des données CIFAR10 est la suivante (Batch, Channels, Height, Width).

Dans la première partie, vous êtes censés construire et entraîner un MLP. Dans la deuxième partie, vous allez élaborer et entraîner un ResNet18. Dans la troisième partie, vous aurez à construire et à entraîner une architecture récente appelée MLPMixer. Comme vous pouvez le constater, l'évolution de ces architectures reflète également la récente vague de recherche en vision par ordinateur. Cela a commencé avec les MLP, puis les CNN ont été les plus populaires, et maintenant nous découvrons que les MLP, lorsqu'elles sont bien conçues, ne sont pas si mal !

Instructions de programmation Vous devrez utiliser PyTorch pour répondre à toutes les questions. De plus, ce devoir nécessite l'exécution des modèles sur des GPUs (sinon cela prendra un temps considérable) ; si vous n'avez pas accès à vos propres ressources (par exemple, votre propre machine, un cluster), veuillez vous servir de Google Colab. Le principal fichier pour lancer vos expériences est `main.py` ou `main.ipynb`. Pendant le développement, vous pouvez utiliser les tests unitaires définis dans `test.py` pour vous aider. Rappelez-vous que ce ne sont que des tests de base, et que d'autres tests seront exécutés sur le Gradescope.

Soumission Vous devez soumettre les fichiers `mlp.py`, `resnet18.py`, `mlpmixer.py` et `utils.py` à gradescope pour l'auto-évaluation, ainsi qu'un rapport PDF pour les questions à caractère expérimental et ouvert.

Problem 1

Implémentation d'un MLP (5pts) Dans ce problème, vous allez implémenter un MLP.

Un MLP est constitué de plusieurs couches linéaires suivies d'une fonction de non-linéarité. Vous trouverez la template de code dans `mlp.py`.

1. Vous devez compléter la `class Linear` dans le fichier `mlp.py`. La classe doit avoir deux attributs : `weight` et `bias`. Complétez la fonction forward correspondante
2. Vous pouvez maintenant passer à la `class MLP`. Dans `_build_layers`, vous devez mettre en place les couches cachées ainsi que la couche de sortie (classificateur). Dans `_initialize_linear_layer`, vous devez remplir les biais avec des zéros et utiliser l'initialisation normale de Xavier pour les poids. Essayez de vous référer à la documentation de pytorch pour cette partie.
3. Complétez le `activation_fn` afin qu'il puisse traiter les entrées en fonction de différents `self.activation`. Compléter la fonction forward selon la docstring.

Problem 2

Implémentation de ResNet18 (15pts) ResNet propose de faire usage des connexions résiduelles afin que le réseau puisse être entrainé avec beaucoup plus de couches. Il est démontré que l'augmentation de la profondeur permet aux réseaux de neurones d'obtenir de meilleures performances dans plusieurs applications de vision. Dans ce problème, vous allez implémenter l'architecture de ResNet18 dans `resnet18.py`. Vous pouvez trouver les détails de l'architecture dans la Figure 1.

1. Nous fournissons le template de code pour le bloc ResNet dans `class BasicBlock`. Vous pouvez voir qu'il y a des points d'interrogation dans la fonction d'initialisation pour les arguments manquants. Veuillez compléter les arguments de façon à ce que la forme de chaque composant ait un sens. Ensuite, complétez la fonction forward selon le schéma de la Figure 1.
2. Complétez l'architecture ResNet18 dans `class ResNet18`. Remplissez les arguments manquants et complétez la fonction forward.

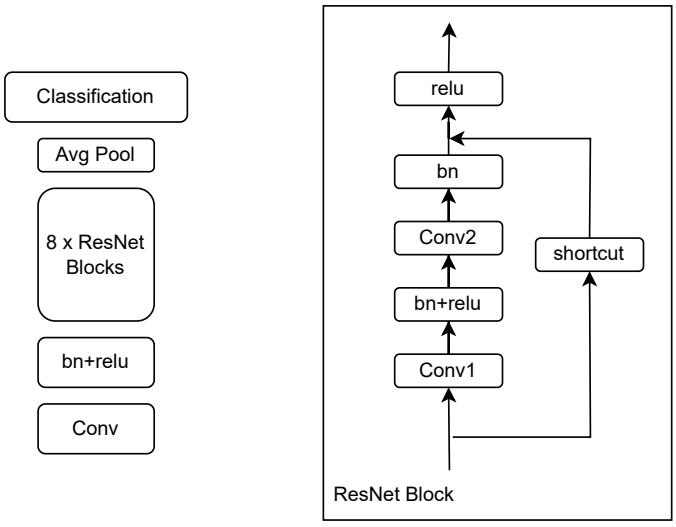

Figure 1: L'architecture ResNet18. **Gauche** : Le diagramme d'architecture d'un ResNet18. Il commence par une couche de convolution 3x3. Le corps principal est constitué de huit blocs ResNet. À la fin, on trouve le pooling moyen global et la tête de classification. **Droite** : Diagramme d'un bloc ResNet. Dans une branche, les entrées passent par deux couches de convolution avec des normalisations par batch et des fonctions de non-linéarité. Dans l'autre branche, les entrées passent par un module de raccourci, qui peut être une fonction d'identité si la taille correspond. Enfin, les sorties des deux branches sont additionnées. La branche de raccourci est également appelée *connexions résiduelles*.

Problem 3

Implémentation de MLPMixer (20pts) Dans cette partie, vous implémenterez l'architecture de MLPMixer. Veuillez fournir votre réponse dans le fichier `mlpmixer.py`. MLPMixer est un article récent dont les auteurs constatent que, si elles sont bien faites, les architectures MLP seules peuvent également être très compétitives. L'architecture est montrée dans la Figure 2.

1. La première composante que vous allez construire est `class PatchEmbed`. Contrairement aux architectures précédentes, MLPMixer divise l'image entière en petits patchs, et effectue une transformation linéaire commune sur ces patchs. Par exemple, si la taille de l'image est de 256×256, et la taille du patch est de 8×8. Nous avons alors $32 \times 32 = 1024$ patchs. En supposant que notre image possède des chaînes RGB et que nous intégrons chaque patch à un vecteur de 512 dimensions, nous avons besoin d'une couche linéaire de forme (192, 512). La procédure ci-dessus est assez complexe, mais en pratique, nous pouvons réaliser l'opération ci-dessus avec une simple couche de convolution. Vous allez l'implémenter. Veuillez compléter l'argument manquant de `self.proj`.

2. Implémentez le `class MixerBlock` et le `class MLPMixer` en vous appuyant sur le schéma d'architecture.

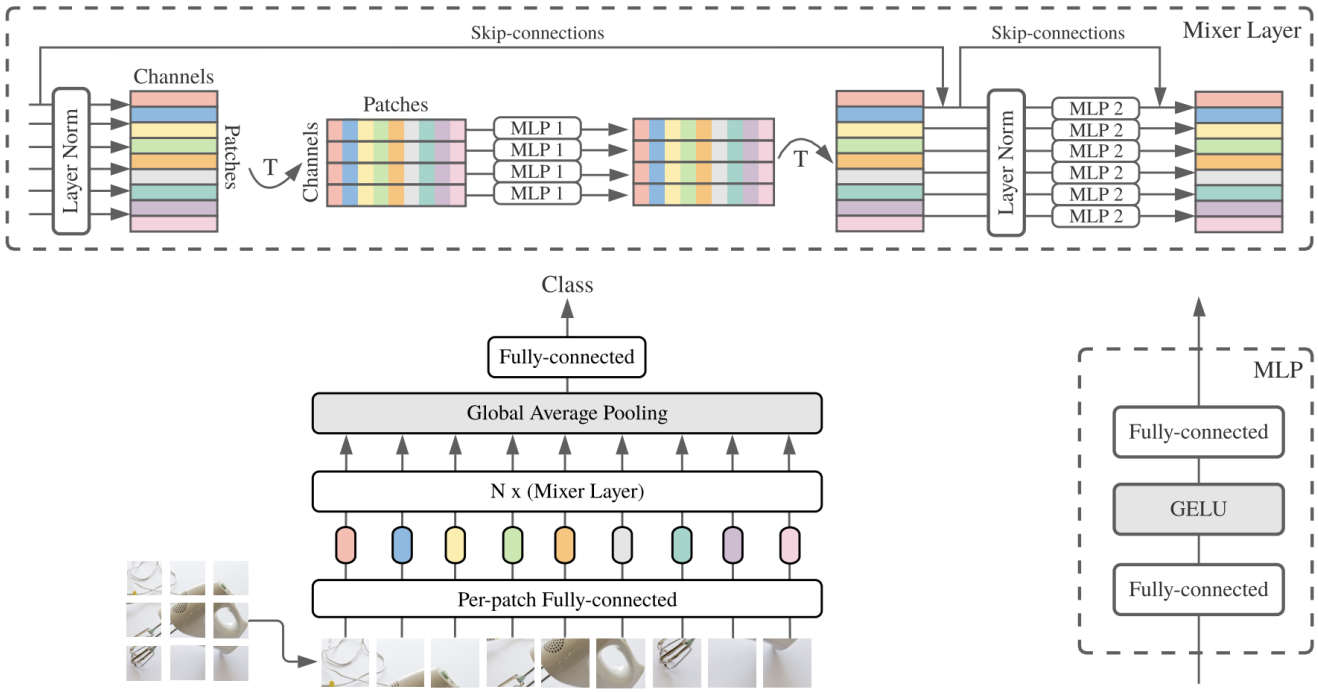

Figure 2: Reproduite de l'article de MLPMixer. MLP-Mixer se compose des embeddings linéaires par patch, des couches Mixer et d'une tête de classification. Les couches de mixage contiennent un MLP de mixage de tokens et un MLP de mixage de chaînes, chacun étant composé de deux couches entièrement connectées et d'une non-linéarité GELU. couches entièrement connectées et d'une non-linéarité GELU. Les autres composants comprennent : des connexions de saut, le dropout, et la norme de couche sur les chaînes.

Problem 4

L'entraînement des modèles (60pts) Dans ce problème, vous allez entraîner l'architecture implémentée ci-dessus et effectuer une recherche d'hyperparamètres. Veuillez consulter `main.py` ou `main.ipynb`. Pour chaque architecture, nous fournissons les fichiers de configuration json du modèle dans `model_configs`. Pour chacune des expériences, entraîner pour au moins 10 époques. Veuillez soumettre un rapport PDF pour répondre aux questions suivantes:

1. (5pts) Complétez la fonction `cross_entropy_loss` dans `utils.py` **sans** recourir aux fonctions préexistantes `torch.nn.CrossEntropyLoss` ou `torch.nn.functional.cross_entropy`.

2. (10pts) Pour l'architecture MLP, étudiez l'effet de la non-linéarité tout en conservant les autres hyperparamètres par défaut. Vous devez fournir quatre figures correspondant à la fonction de coût d'apprentissage, à la fonction de coût de validation, à la précision d'apprentissage et à la précision de validation, l'axe des x indiquant le nombre d'époques. Pour chaque figure, utilisez la légende pour indiquer la non-linéarité utilisée. Déterminez quelle non-linéarité est la meilleure et donnez votre explication. Nous fournissons facultativement la fonction utilitaire de visualisation dans `utils.py`.

Réponse: Aux figures 3, 4, 5, 6 on donne les graphiques correspondant à la fonction de coût d'apprentissage, à la fonction de coût de validation, à la précision d'apprentissage et à la précision de validation pour les trois fonctions d'activation (relu, tanh, sigmoid). La meilleure fonction d'activation selon ces figures est *relu*. *relu* est une fonction non linéaire permettant représenter une grande gamme de problème lorsque combiné dans un réseau de neurones, tout en ayant un potentiel fort gradient ce qui permet une rapide optimisation du réseau par rapport aux autres fonctions testés.

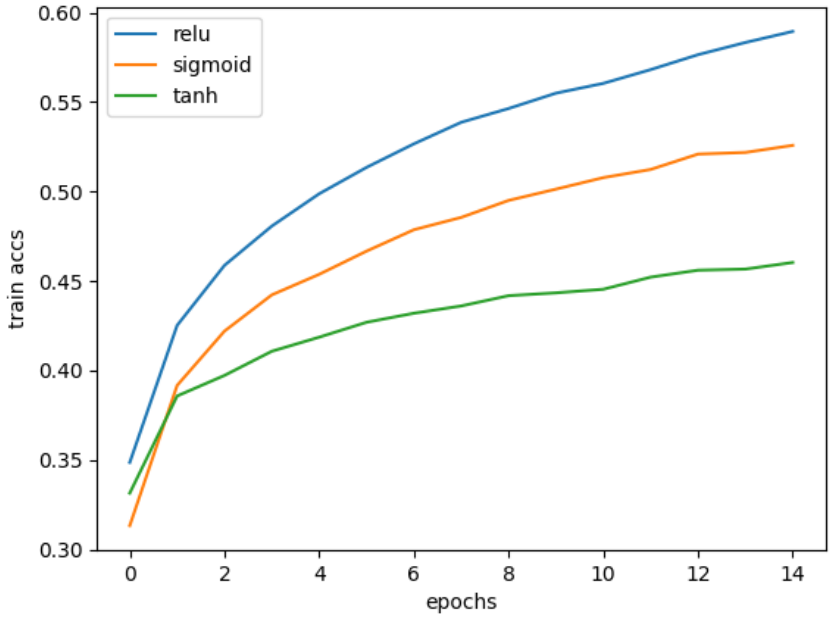

Figure 3: Train accuracy du MLP en fonction du nombre d'époque pour les fonction d'activation relu, tanh, sigmoid.

3. (10pts) Pour l'architecture ResNet18, étudiez l'effet du taux d'apprentissage avec l'optimiseur Adam. Choisissez des taux d'apprentissage de 0,1, 0,01, 0,001, 0,0001, 0,00001. Fournissez les graphiques et expliquez vos résultats.

   Réponse:

   Aux figures 7, 8, 9 et 10, on donne les graphiques correspondant à la fonction de coût d'apprentissage, à la fonction de coût de validation, à la précision d'apprentissage et à la précision de validation pour des learning rate de 0.1, 0.01, 0.001, 0.0001, 0.00001 avec adam. Pour des *lr* trop bas ou trop élevé (0.00001 ou 0.1), le modèle performe significativement moins bien. Cela est dû à une convergence trop lente, pour un *lr* trop petit le modèle à reste relativement trop proche de son état initial pour un même nombre d'époque ou reste coincé dans un minimum local proche de son état initial. Pour un *lr* trop grand le modèle est dans l'incapacité de converger, la décente de gradient doit être trop rapide amenant l'optimiseur à osciller entre deux valeurs sans converger. Le meilleur *lr* se trouve à une valeur de 0.001.

4. (15pts) Pour MLPMixer, examinez l'effet de la taille du patch. Aucune valeur recommandée

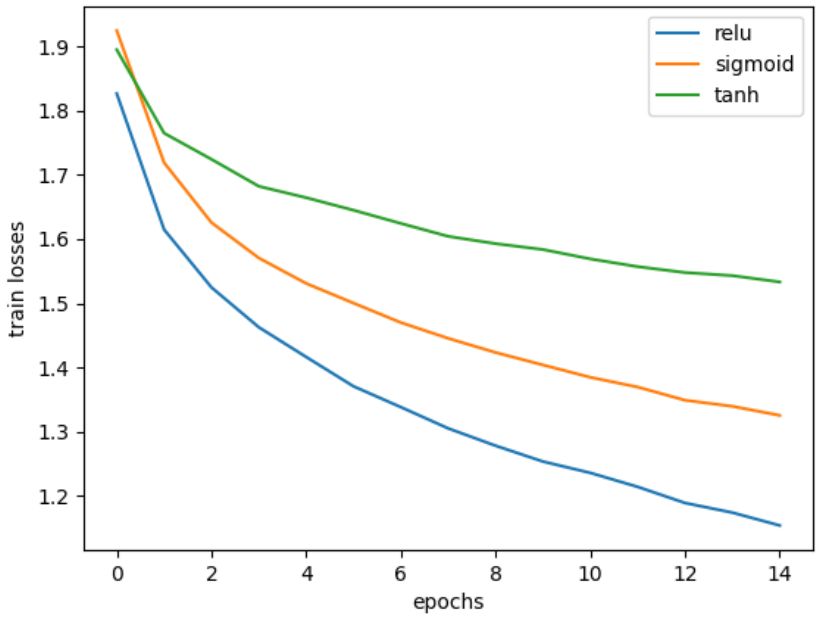

Figure 4: Train losses du MLP en fonction du nombre d'époque pour les fonction d'activation relu, tanh, sigmoid.

n'est donnée, et vous devez effectuer au moins 3 expériences. N'oubliez pas qu'il n'y a que quelques valeurs valables pour la taille du patch pour une taille d'image donnée. Veuillez fournir des graphiques et expliquer vos résultats. Expliquez également par écrit l'effet sur le nombre de paramètres du modèle ainsi que sur le temps d'exécution.

Réponse: Voir les figures 11, 12, 13, 14 où on donne les graphiques correspondant à la fonction de coût d'apprentissage, à la fonction de coût de validation, à la précision d'apprentissage et à la précision de validation pour des patch de taille 2x2, 4x4, 8x8, 16x16, 32x32. Les patch de taille 16x16 et 32x32 couvrent des quart ou l'entièreté de l'image (32x32) et semble performer significativement moins bien que pour des patch de taille 2x2, 4x4, 8x8. Cela s'explique probablement à ce que les MLP-Mixer font des corrélations entre les différentes partie d'une image où un trop faible nombre de patch diminue le nombre de corrélation possible augmentant l'erreur.

La taille du input à l'entré du mlp-token augmente quasi exponentiellement avec des patch de taille de plus en plus petit. La taille du input à l'entré du mlp-channels reste la même. Le temps d'exécution diminue quasi-exponentiellement avec une taille de patch de plus en plus grande (voir 15).

5. (10pts) Trouvez votre meilleur modèle ResNet18 en jouant avec les hyperparamètres. Mentionnez les hyperparamètres dans votre rapport. Visualisez les noyaux de la première couche, qui a un poids de forme (out_channel, in_channel, kernel_size, kernel_size). Vous pouvez modifier le fichier `main.py` ou ajouter une cellule supplémentaire dans `main.ipynb` pour la visualisation. Puisque nous avons 64 chaînes de sortie et 3 chaînes d'entrée (RGB), on peut voir cela comme soixante-quatre 3×3 petites images, où chaque image représente le noyau cor-

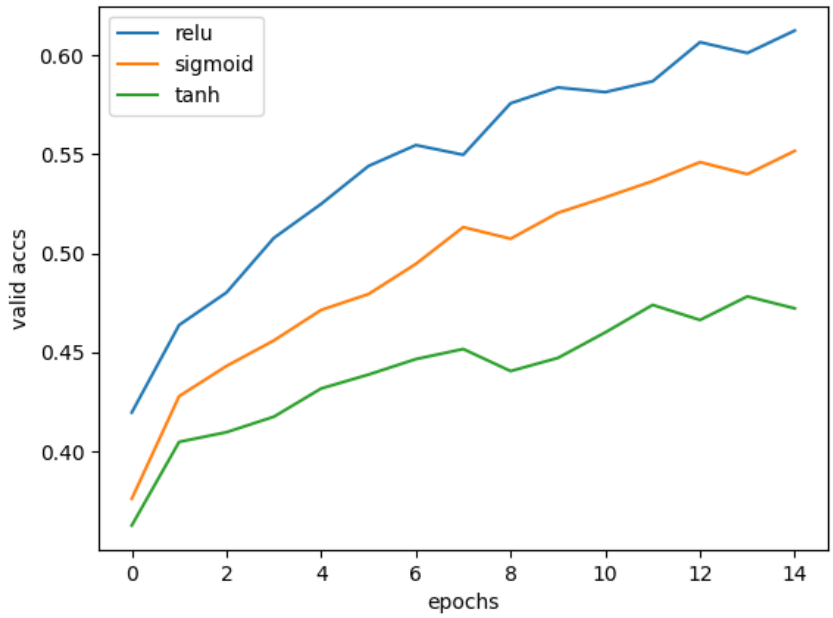

Figure 5: Valid accuracy du MLP en fonction du nombre d'époque pour les fonction d'activation relu, tanh, sigmoid.

respondant à cette chaîne de sortie. Notez qu'il s'agit d'une question ouverte. Vous pouvez effectuer différents prétraitements pour la visualisation, par exemple, normaliser les valeurs de poids, faire la moyenne entre les chaînes pour obtenir des images en échelle de gris, etc. Vous pouvez voir plus de détails et d'exemples dans ce blogpost. Veuillez décrire votre démarche de visualisation dans votre rapport.

Réponse: On varie les hyper-paramètres de batch size, learnin rate et weight decay ainsi que l'algorithme d'optimisation pour 15 époques. Chaque hyper-paramètre et fait varier dans un intervalle avec les autres hyper-paramètres pris par défaut.

Le batch size est fait varier entre 128, 256, 512 et 1024. Le meilleur batch size est de 128 avec une exactitude de validation de 0.931.

Le learning rate est fait varier entre 0.1, 0.01, 0.001, 0.0001, 0.00001. Le meilleur learning rate est de 0.001 avec une exactitude de validation de 0.929.

Le weight decay est fait varier entre 0.05, 0.005, 0.0005, 0.00005, 0.000005. Le meilleur weight decay est de 0.005 avec une exactitude de validation de 0.938. Aucun effet significatif n'est observé.

Les algorithmes d'optimisation testé sont adam, adamw, sgd et momentum. Le meilleur algorithme d'optimisation est adamw avec une exactitude de validation de 0.933.

Les meilleurs paramètres choisit sont combiné (bs=128, lr=0.001, wd=0.005, op=adamw) pour un modèle donnant une exactitude de validation de 0.946 et d'entraînement de 0.925 après 15 époques.

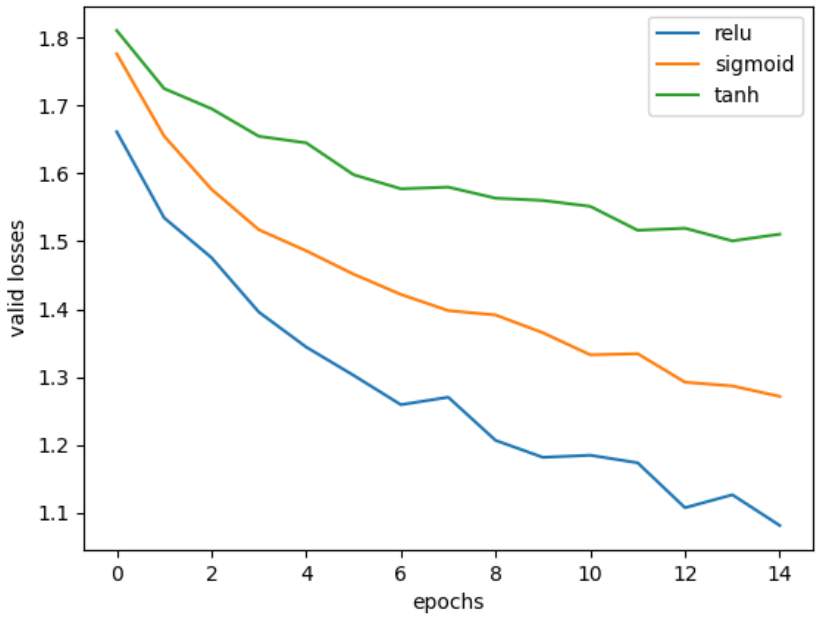

Figure 6: Valid losses du MLP en fonction du nombre d'époque pour les fonction d'activation relu, tanh, sigmoid.

Les poids des noyaux visualisé aux figures 16 et 17 sont premièrement normalisé. À la figure 17 la moyenne est faites sur les canaux. Le cmap utilisé est "bwr" de matplotlib.

À la figure 16 on présente les noyaux de la première couche de convolution du premier canal. Des filtres intéressants sont par exemple des filtres opposé bbr (b=bleu, r=rouge) et rbb horizontaux par exemple (1,1) et (8,7). Il y a des filtres de différente orientation en forme de L, par exemple (2,2). Diagonaux, par exemple (3,1) ou (1,2). Ou "flicker", par exemple (2,3) ou (4,4). Et différent autres, par exemple (8,3) et (8,4).

À la figure 17 on présente les noyaux de la première couche de convolution de la moyenne des canaux. La moyenne des canaux est peut-être moins claire, mais présente des invariants à travers les couleurs. On observe les différents type de filtre trouvé à la couche 0. Des filtres opposés ((1,4), (1,6)), horizontaux (4,2), diagonaux (3,7), forme L (7,8) et autres.

6. (10pts) Définissez la taille du patch à 4, et trouvez les hyper-paramètres pour votre meilleur modèle de MLPMixer. Fournissez les hyper-paramètres complets dans votre rapport. Visualisez les poids (uniquement la première couche) du MLP de mixage de tokens dans le premier bloc comme décrit dans la Figure 5 du document MLPMixer. Commentez et comparez vos résultats avec les visualisations de la convolution. Expliquez ce qui, selon vous, justifie le succès du MLPMixer, notamment par rapport aux MLP normaux.

   Réponse: On varie les hyper-paramètres de batch size, learnin rate, weight decay et embedded dimension ainsi que l'algorithme d'optimisation pour 15 époques. Chaque hyper-paramètre et fait varier dans un intervalle avec les autres hyper-paramètres pris par défaut.

   Le batch size est fait varier entre 128, 256, 512 et 1024 pour des valeurs d'exactitude de

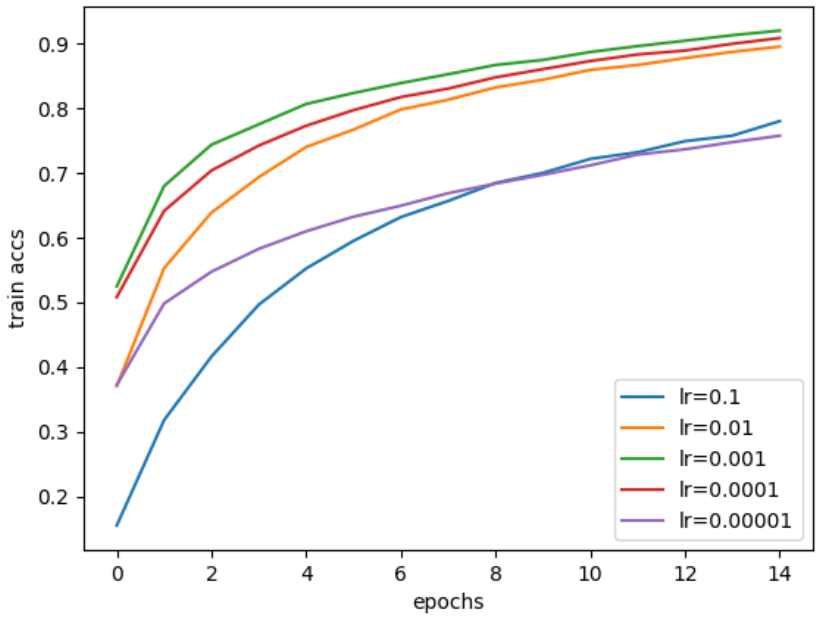

Figure 7: Train accuracy du resnet18 en fonction du nombre d'époque pour des learning rate de 0.1, 0.01, 0.001, 0.0001, 0.00001.

validation de 0.856 0.854 0.811 0.764 respectivement. Le meilleur batch size est de 128 avec une exactitude de validation de 0.856.

Le learning rate est fait varier entre 0.1, 0.01, 0.001, 0.0001, 0.00001 pour des valeurs d'exactitude de validation de 0.649 0.717 0.861 0.721 0.523 respectivement. Le meilleur learning rate est de 0.001 avec une exactitude de validation de 0.861.

Le weight decay est fait varier entre 0.05, 0.005, 0.0005, 0.00005, 0.000005 pour des valeurs d'exactitude de validation de 0.857 0.856 0.856 0.856 0.856 respectivement. Le meilleur weight decay est de 0.05 avec une exactitude de validation de 0.857. Aucun effet significatif n'est observé.

Le embedded dimension est fait varier entre 32,64,128,256,512,1024 pour des valeurs d'exactitude de validation de 0.695 0.754 0.824 0.861 0.872 0.83 respectivement. Le meilleur embedded dimension est de 512 avec une exactitude de validation de 0.872.

Les algorithmes d'optimisation testé sont adam, adamw, sgd et momentum pour des valeurs d'exactitude de validation de 0.861 0.856 0.314 0.563 respectivement. Le meilleur algorithme d'optimisation est adam avec une exactitude de validation de 0.856.

Les meilleurs paramètres choisit sont combiné (bs=128, lr=0.001, wd=0.05, op=adam, ed=512) pour un modèle donnant une exactitude de validation de 0.868 et d'entraînement de 0.842 après 15 époques.

Les poids de la première couche du mlp-tokens du MLP-Mixer sont présenté à la figure 18 (mais après 225 époque, ça donne des plus beaux filtres). Un total de 512 filtres sont présenté, soit avec une dimension caché de 256 donnant un total de 512/2=256 filtres. Chaque filtres

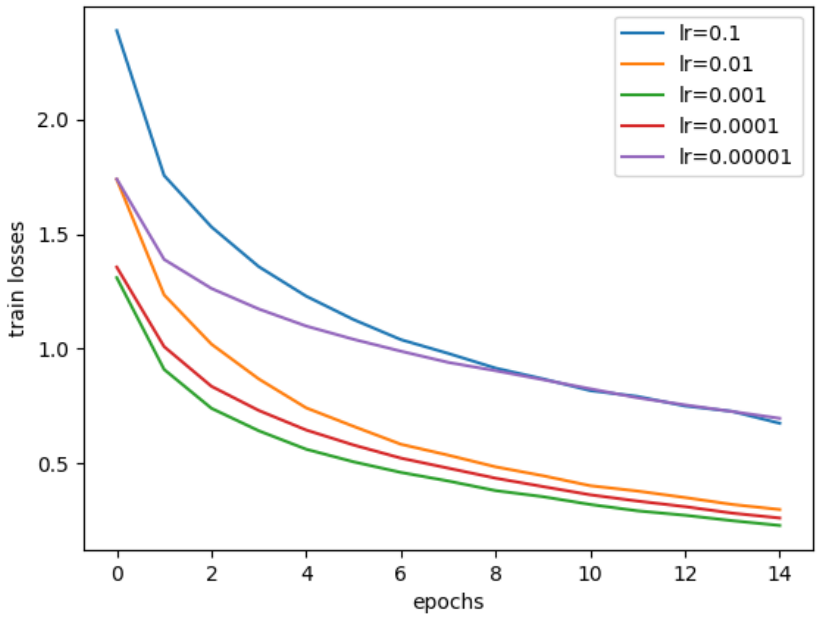

Figure 8: Train losses du resnet18 en fonction du nombre d'époque pour des learning rate de 0.1, 0.01, 0.001, 0.0001, 0.00001.

font FxF=(HxW/P)**2 = (32x32/4)**2 = 8x8. Les filtres sont plus abstrait que pour les MLP. La plupart des formes peuvent ressembler à des nouilles entortillés.

Les images de CIFAR10 ne sont pas très clair car la résolution est très basse, une image haute résolution de chat à été utilisé et redimensionné 128x128 (voir figure 19) pour y appliquer des convolutions à partir des poids de la première couche du mlp-tokens (voir 20).

À la figure 20, on présente l'effet des convolutions des poids (ce n'est pas exactement correct de faire ça) sur l'image du chat (sur le premier channel de l'image original). On peut voir différent motifs ressortir selon le filtre appliqué, par exemple l'image à la position (1,1) ou (5,1). La plupart de filtres semble laisser ressortir différent contour de l'image, l'arrière plan ou l'avant ou différente composante (le chat, l'oreiller ou le sofa).

Selon l'article MLP-Mixer, les premières couches de MLP ont tendance à apprendre des détecteurs qui agissent sur les pixels dans des régions locales d'image. Les token-mixing des MLP-Mixer permettent une communication globale entre différents emplacements spatiaux. Certaines des filtres appris fonctionnent sur l'image entière, tandis que d'autres fonctionnent sur des régions plus spécifique.

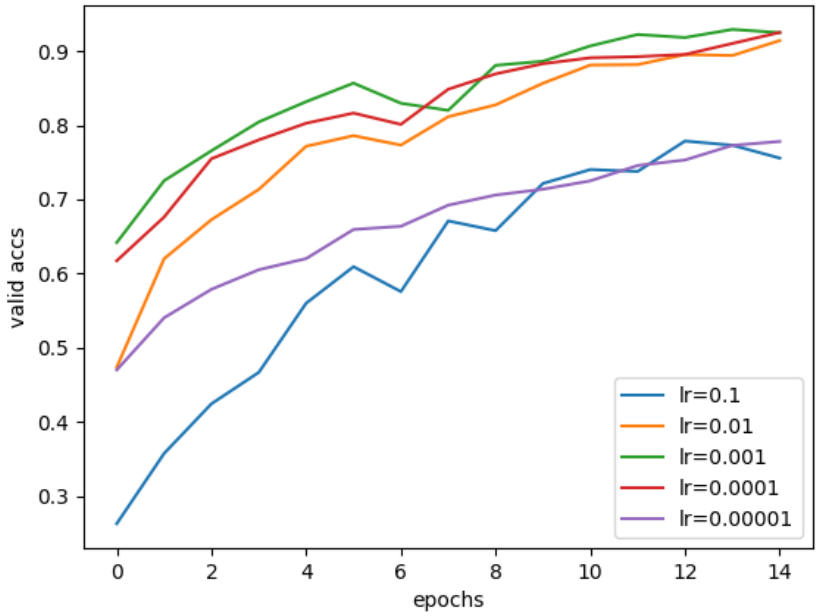

Figure 9: Valid accuracy du resnet18 en fonction du nombre d'époque pour des learning rate de 0.1, 0.01, 0.001, 0.0001, 0.00001.

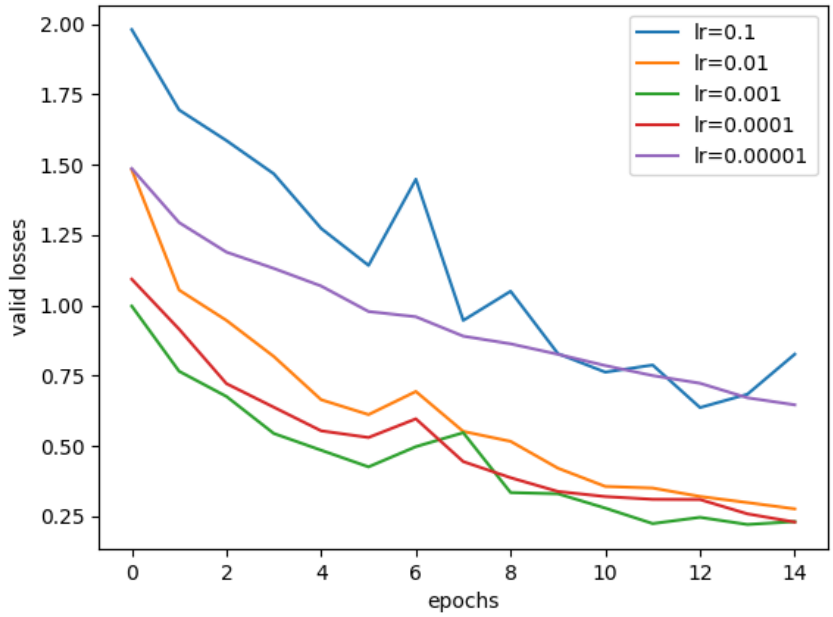

Figure 10: Valid losses du resnet18 en fonction du nombre d'époque pour des learning rate de 0.1, 0.01, 0.001, 0.0001, 0.00001.

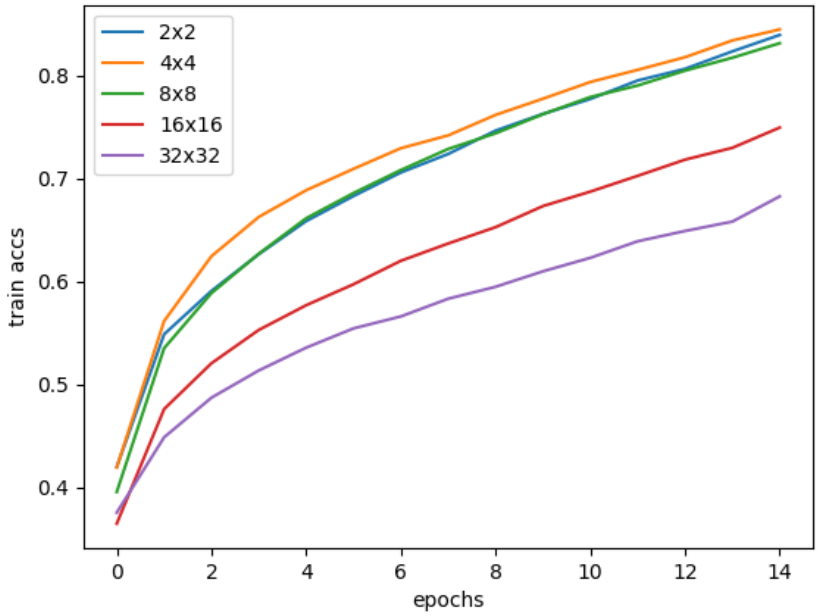

Figure 11: Train accuracy du mlpmixer en fonction du nombre d'époque pour des patch de taille 2x2, 4x4, 8x8, 16x16, 32x32.

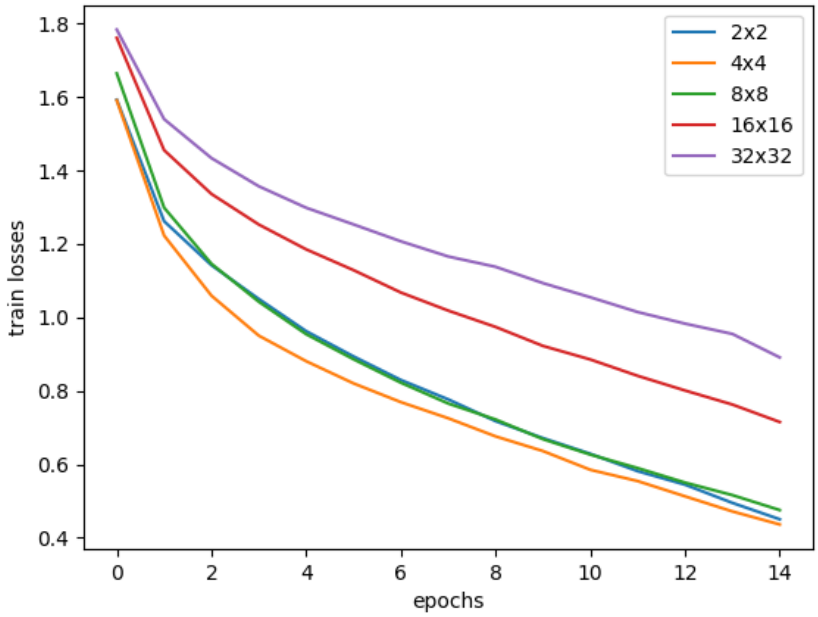

Figure 12: Train losses du mlpmixer en fonction du nombre d'époque pour des patch de taille 2x2, 4x4, 8x8, 16x16, 32x32.

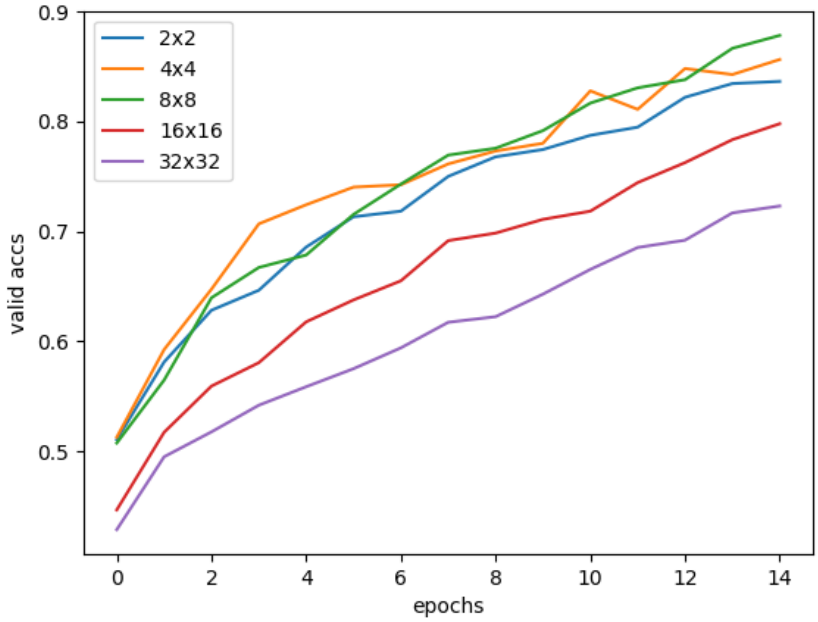

Figure 13: Valid accuracy du mlpmixer en fonction du nombre d'époque pour des patch de taille 2x2, 4x4, 8x8, 16x16, 32x32.

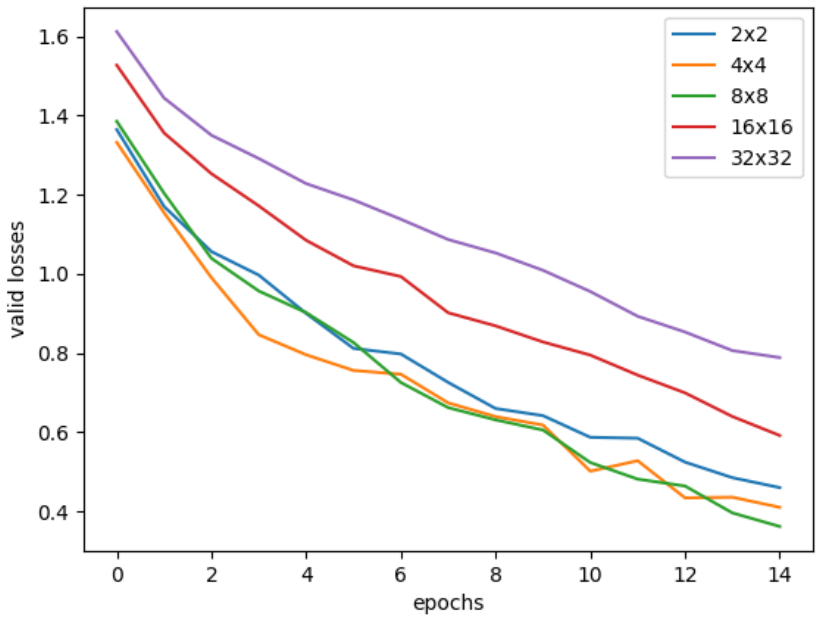

Figure 14: Valid losses du mlpmixer en fonction du nombre d'époque pour des patch de taille 2x2, 4x4, 8x8, 16x16, 32x32.

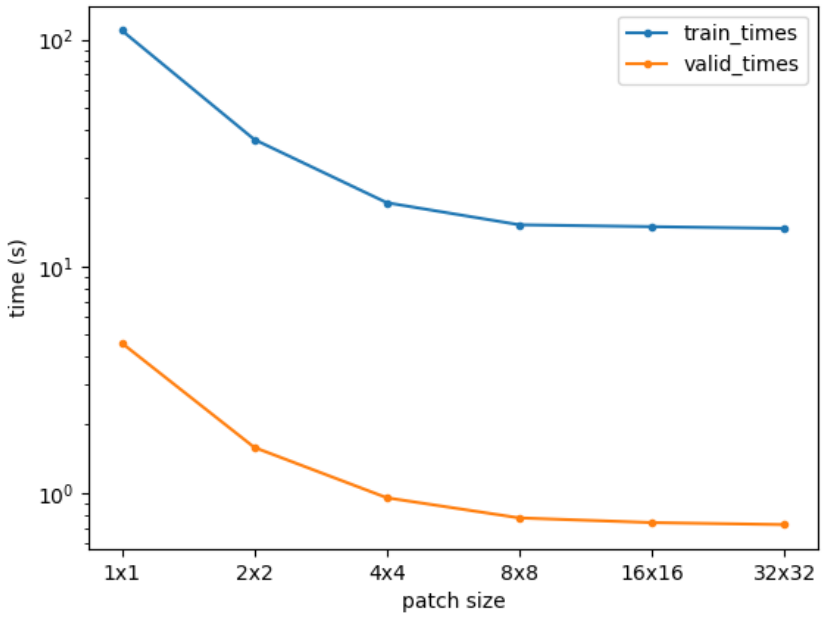

Figure 15: Temps d'exécution du mlpmixer en fonction de la taille du patch pour des patch de taille 1x1, 2x2, 4x4, 8x8, 16x16, 32x32. Note: échelle logarithmique en temps.

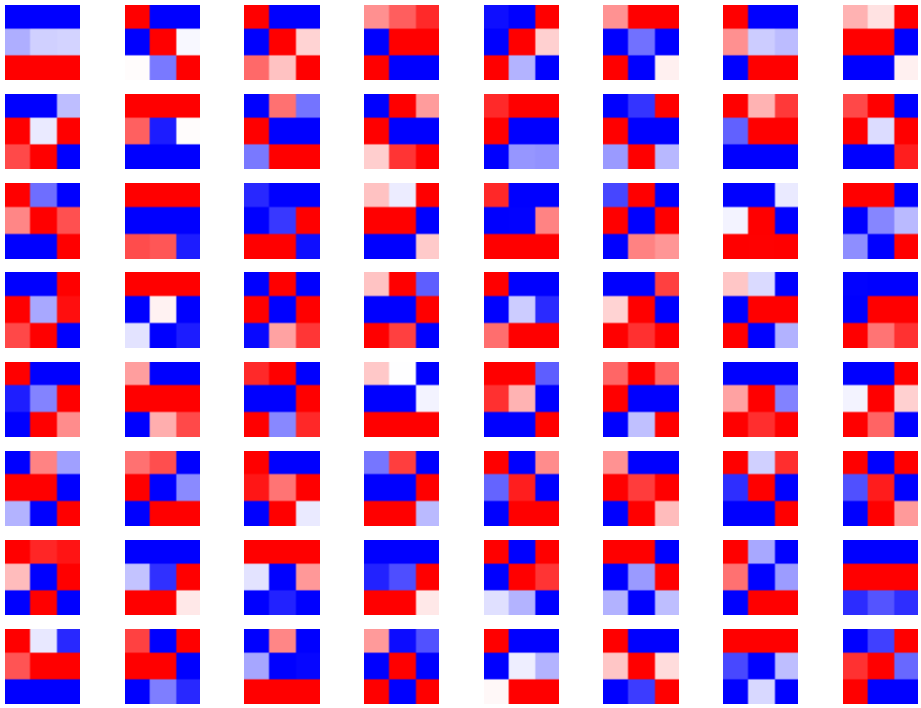

Figure 16: Les noyaux de la première couche de convolution du premier canal.

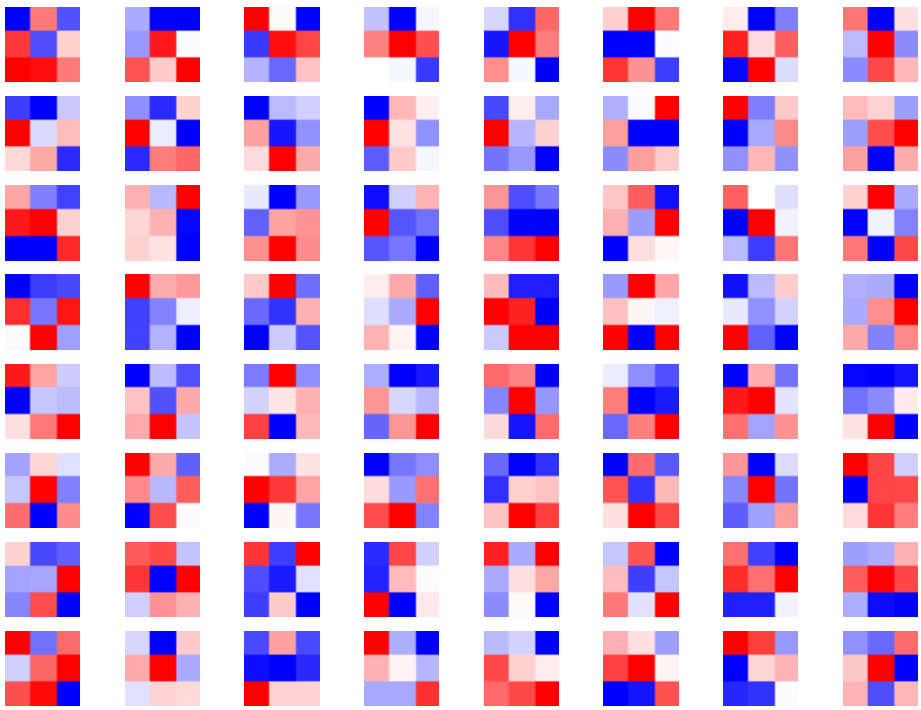

Figure 17: Les noyaux de la première couche de convolution de la moyenne des canaux.

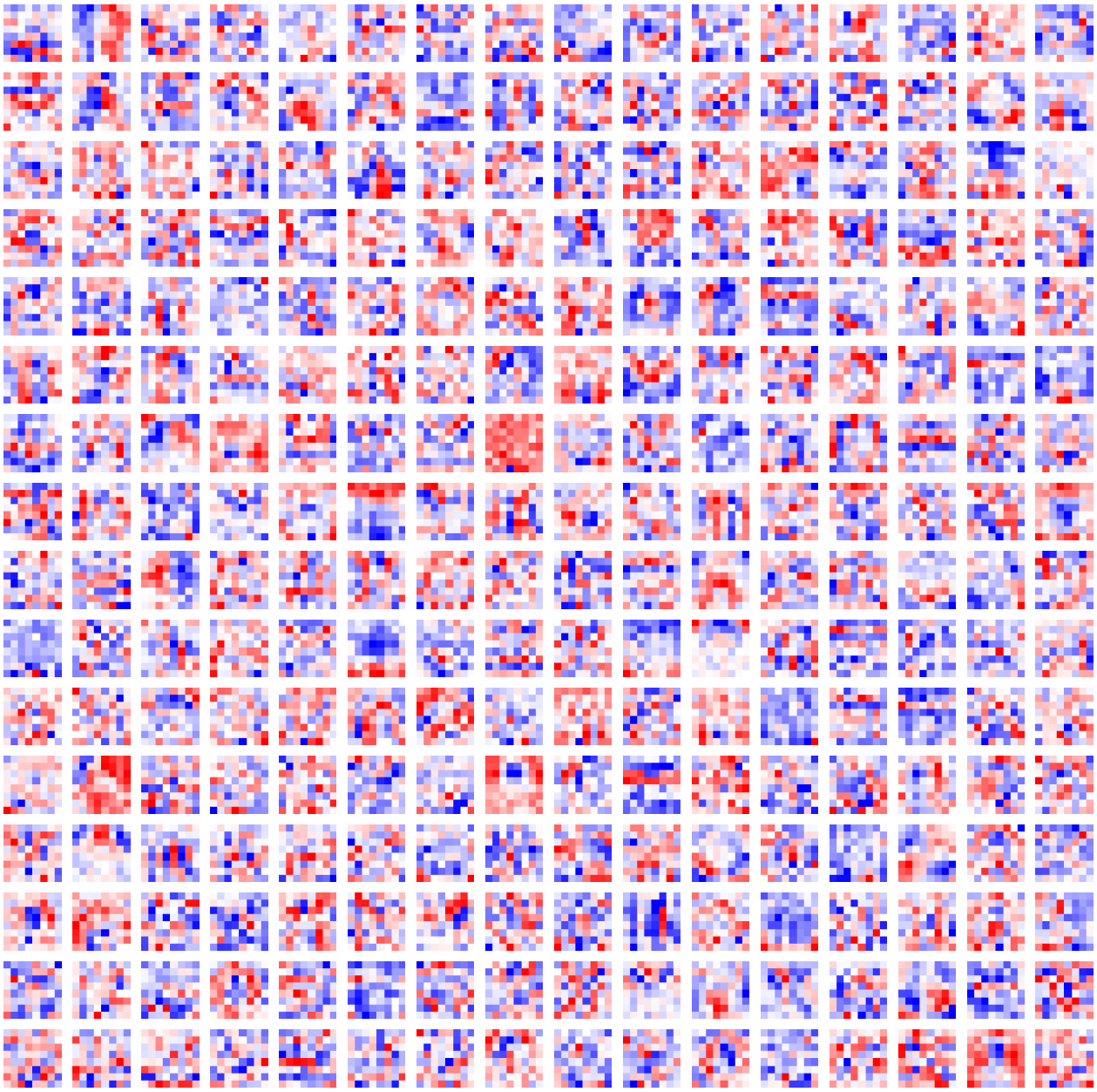

Figure 18: Filtres de la première couche du mlp-tokens du MLP-Mixer. 256 filtres de 8x8.

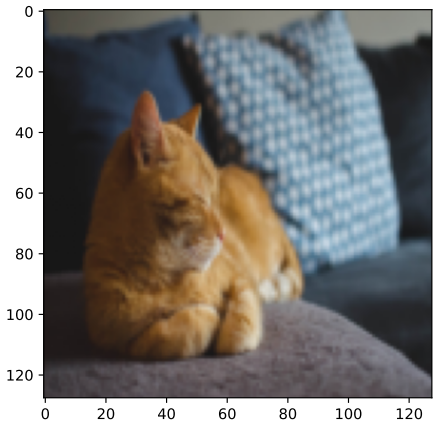

Figure 19: Image du chat (pris de l'internet). Image de dimension 128x128.

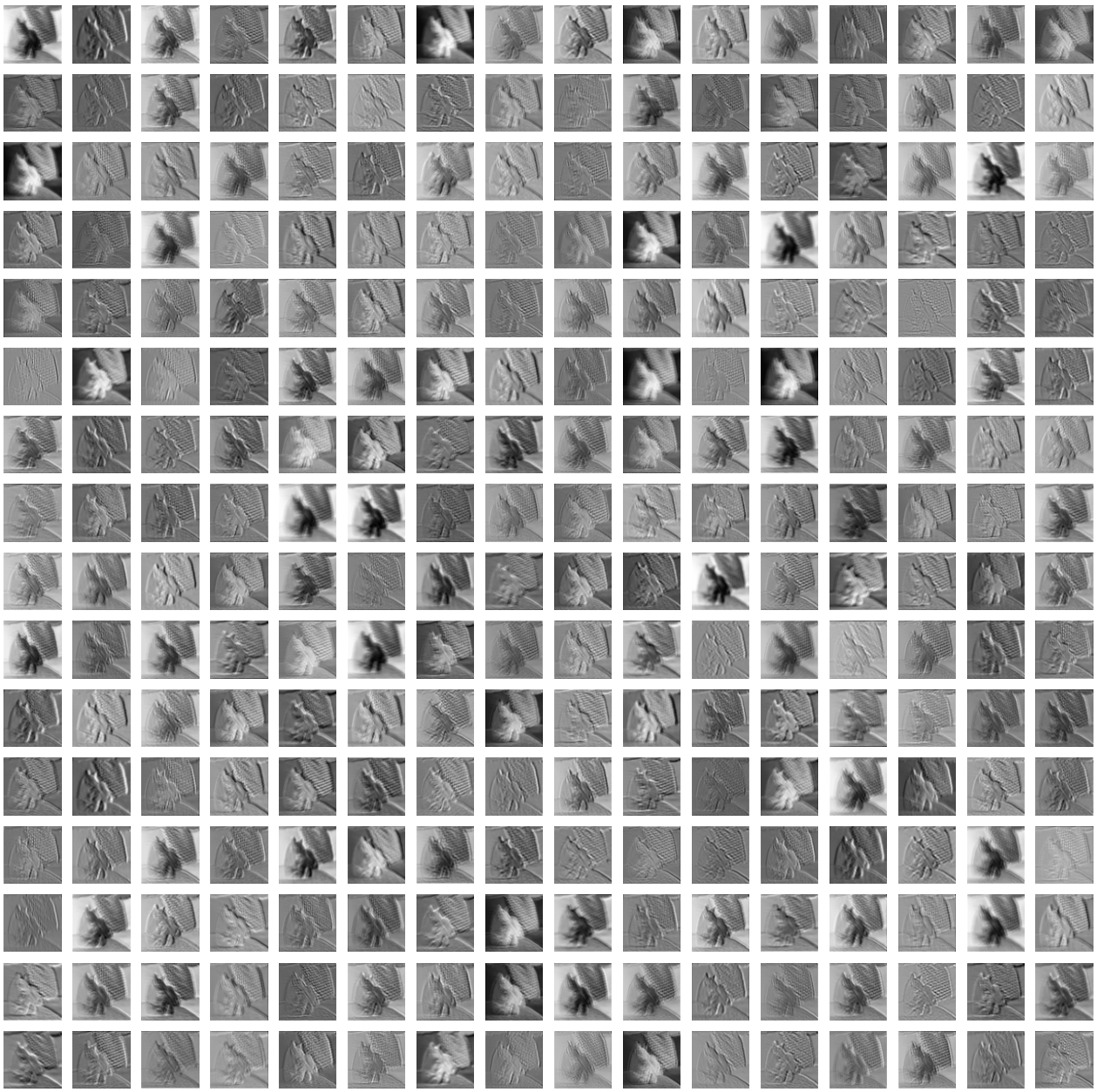

Figure 20: Visualisations de la convolution des poids de la premère couche du mlp-tokens sur l'image du chat.