IA716 - Perception pour les systèmes autonomes

Projet 2
Semi-Supervised Learning
Fixmatch

AILLAUD Arnaud

25 avril 2023

Table des matières

1 Principe de l'algorithme

Le fonctionnement de Fixmatch est relativement simple. Il repose sur deux principes :

- Pseudo-labeling : un pseudo label est assigné à des données non labelisées après prédiction par le modèle, si le classifieur est suffisamment confiant sur la classe à laquelle elles appartiennent.
- Consistency regularization : La perturbation d'une image ne doit pas modifier son label : une image fortement perturbée doit donc être classifiée de la même manière qu'une image faiblement perturbée

La fonction de perte utilisée pour l'entraînement se décompose en deux parties : $\mathcal{L} = \ell_s + \lambda_u \ell_u$.
Avec B, taille de batch, H l'entropie croisée, p_b le one-hot encoding du label associé à l'observation x_b, $p_m(y|x)$ la classe prédite à partir de x, α et $\mathcal{A}$ respectivement l'application d'une faible et d'une forte augmentation sur une image, et μ, τ et λ_u tels que définis plus bas :

$$\ell_s = \frac{1}{B}\sum_{b=1}^{B} H(p_b, p_m(y|\alpha(x_b))) \quad \text{fonction de perte supervisée}$$

$$\ell_u = \frac{1}{\mu B}\sum_{b=1}^{\mu B} \mathbb{1}(\max(p_m(y|\alpha(u_b))) \geq \tau) H(p_m(y|\alpha(u_b)), p_m(y|\mathcal{A}(u_b))) \quad \text{fonction de perte non supervisée}$$

Ainsi, à première vue, les hyperparamètres qui semblent les plus importants pour expliquer les bonnes performances du modèle sont les suivants :

- μ : ratio entre le nombre d'images non labelisées et le nombre d'images labelisées utilisés à chaque epoch. La valeur optimale de ce paramètre proposée par les auteurs de l'article est de **7** (7 images non labelisées pour une image labelisée dans chaque batch).
- τ : seuil de confiance. Il s'agit de la probabilité au-delà de laquelle un pseudo label est assigné à une image faiblement perturbée. La valeur optimale pour ce paramètre proposée dans l'article est de **0.95**, i.e. une image est utilisée dans le calcul de la fonction de perte non supervisée si le modèle prédit une classe pour cette image avec une probabilité supérieure à 95%.
- T : température utilisée dans le softmax de prédiction du pseudo label, qui modifie la valeur de probabilité calculée pour chaque classe (sharpening). Une grande valeur de température tend à égaliser les probabilités générées (probabilité uniforme d'appartenir à chaque classe), une faible valeur tend à creuser les écarts (et faire permettre à la classe prédite de plus facilement dépasser τ). Les auteurs étudient les interactions de T avec τ dans la section 5 de l'article[1], et concluent qu'elle n'a finalement que peu d'impact par rapport au seuil de confiance. Son utilisation rajoute un hyperparamètre sans gain de performance réel, sa valeur optimale est donc de **1**.
- λ_u : paramètre de régularisation, permettant de pondérer le coût non supervisé par rapport a coût supervisé. Sa valeur optimale donnée par les auteurs est de **1**.

2 Résultats

L'article détaille de manière exhaustive l'impact des hyperparamètres précédents, et bien d'autres. Pour mon implémentation, j'ai donc choisi de réimplémenter l'algorithme avec les paramètres optimaux, et de comparer les résultats :

- en utilisant l'implémentation de RandAugment de Pytorch d'une part
  `RandAugment(num_ops=2, magnitude=10, num_magnitude_bins=11, interpolation=InterpolationMode.BILINEAR)`
- en utilisant un optimizer Adam basique avec un scheduler ReduceLROnPlateau d'autre part

Au bout de 1024 epochs, l'implémentation optimale obtient un score de précision de **95.06%** sur le jeu de test, ce qui est cohérent avec avec les scores obtenus dans l'article. La courbe de perte sur le jeu de test recommence à augmenter légèrement à partir de 800 epochs, caractéristique d'un léger sur-apprentissage. La précision sur le jeu de test stagne à partir de cette valeur.

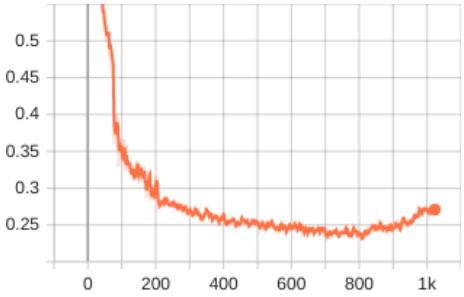

FIGURE 1 – Loss sur le jeu de test

2.1 Utilisation de l'implémentation de RandAugment de Pytorch

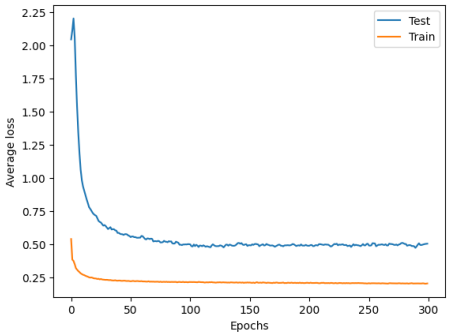

FIGURE 2 – Evolution de la loss sur le jeu de test et d'entraînement pour l'implémentation avec RandAugment de Pytorch

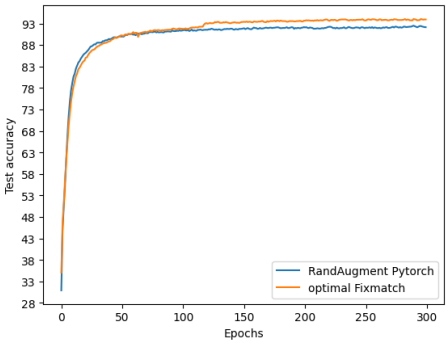

FIGURE 3 – Comparaison de la précision sur le jeu de test entre implémentation optimale et RandAugment de Pytorch

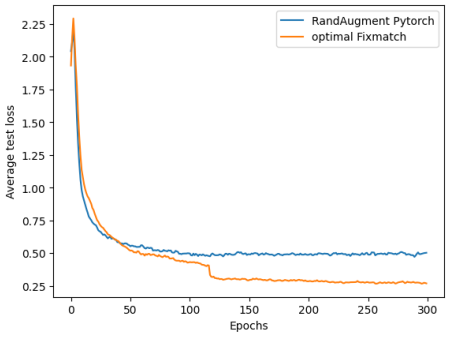

FIGURE 4 – Comparaison de la loss sur le jeu de train entre implémentation optimale et RandAugment de Pytorch

On remarque que la loss stagne et commence même à légèrement augmenter à partir d'environ 150 epochs avec l'utilisation de RandAugment de Pytorch, ce qui n'est pas le cas avec l'implémentation optimale.
De plus, bien la précision sur le jeu de test soit meilleure sur les 50 premières epochs avec RA de Pytorch, à partir d'environ 100 epochs l'implémentation optimale proposée par les auteurs de l'article prend le dessus. On remarque un palier vers 120 epochs tant sur la courbe de loss que de précision.
Cependant, dans l'ensemble, les résultats obtenus avec l'implémentation de Pytorch, i.e. sans l'utilisation de Cutout, semblent très bons et relativement proches de l'implémentation optimale. Au bout de 300 epochs, la précision sur le jeu de test est de **92.3%** au lieu de **93.8%** avec l'implémentation optimale.

2.2 Utilisation d'un optimizer Adam avec scheduler ReduceLRonPlateau

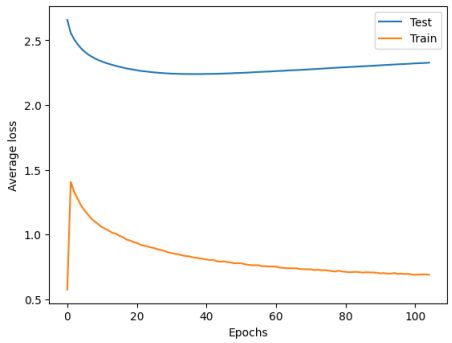

FIGURE 5 – Learning curves pour l'implémentation avec Adam

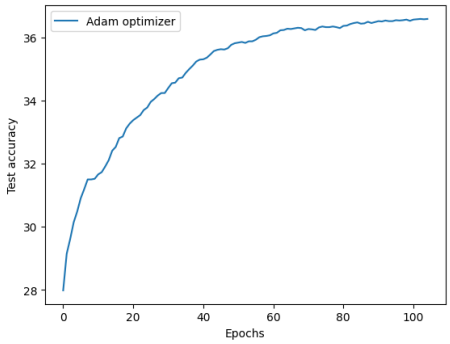

FIGURE 6 – Evolution de la précision sur le jeu de test pour l'implémentation avec Adam

Avec cette implémentation, l'optimizer SGD a été remplacé par un optimizer Adam avec un learning rate plus faible (0.001 au lieu de 0.03) et la même normalisation L2, le scheduler cosinus a été remplacé par un scheduler ReduceLRonPlateau configuré pour réduire le learning rate lorsque la loss stagne, et le modèle Exponential Moving Average n'a pas été utilisé pour l'évaluation.
On remarque que la courbe de loss sur le jeu de test commence à augmenter dès $\sim$ 40 epochs, ce qui fait déjà penser à du surentraînement. En outre, la précision sur le jeu de test stagne vers **36%** au bout de 100 epochs, alors qu'avec l'implémentation optimale un score de plus de 90% est déjà atteint à ce nombre d'epochs.
Il apparaît donc que **l'ingénierie d'implémentation a un impact beaucoup plus important que les méthodes de perturbation d'images ou la modification des hyperparamètres spécifiques à Fixmatch** sur le score final obtenu.

3 Détails d'implémentation

Au-delà de la simplicité apparente de l'algorithme, ce sont les choix d'implémentation qui permettent à FixMatch d'atteindre de si bons scores, en particulier les hyperparamètres d'optimizer, scheduler et méthodes de

mise à jour des paramètres du modèle. Contrairement aux paramètres μ, τ ou T, le choix de ces hyperparamètres n'est pas clairement détaillé dans l'article.

3.1 Scheduler

Pour réduire le learning rate au cours de l'entraînement, un scheduler de type cosinus avec warmup est utilisé. Cependant, son implémentation diffère de celle proposée dans l'article qui a initialement proposé ce scheduler[2]. La formule de mise à jour du learning rate proposée dans cet article est la suivante :

$$\eta_t = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min})(1 + cos(\frac{T_{cur}}{T_i}\pi))$$

avec η_t : learning rate à l'epoch t, T_{cur} : nombre d'epochs depuis le dernier restart, T_i : nombre d'epochs entre 2 restarts.
Tel qu'implémenté dans FixMatch, le scheduler augmente le learning rate de manière linéaire jusqu'au premier restart (multiplication par le ratio entre le numéro d'epoch actuel et le numéro du 1er restart). A partir du 1er restart, la formule suivante est appliquée :

$$\eta_t = \eta_{max} cos(\frac{7t}{16K}\pi)$$

avec t : epoch actuelle, K : nombre de steps maximal (1024×1024 dans l'implémentation optimale).

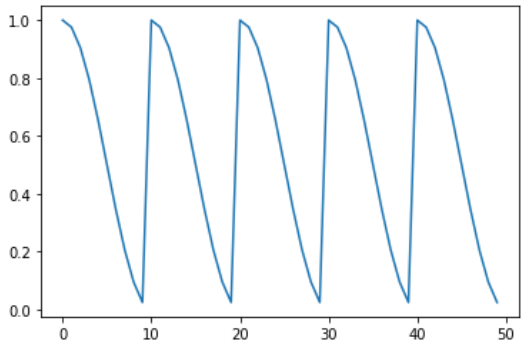

FIGURE 7 – Cosine scheduler standard avec restart toutes les 10 itérations ($\eta_{max} = 1$ et $\eta_{min} = 0$)

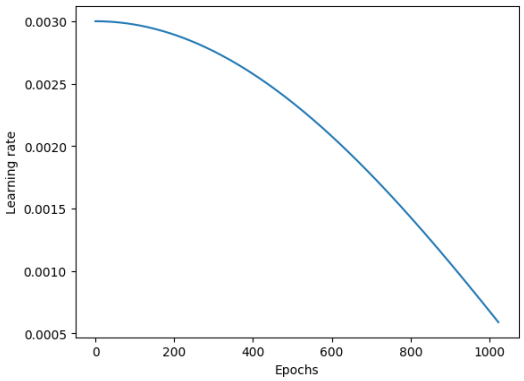

FIGURE 8 – Cosine scheduler utilisé dans Fixmatch (sans restart)

Avec un learning rate initial de 0.003, le leaning rate obtenu après 1024×1024 steps dans Fixmatch est de $\sim$0.00068. Il y a bien une ressemblance entre les 2 implémentations, mais les choix des auteurs concernant cette implémentation particulière, spécifiquement le choix du rapport $\frac{7}{16}$, ne sont pas justifiés et difficilement intuitables.

3.2 Optimizer

L'optimizer utilisé dans l'implémentation optimale est un Gradient Stochastique (SGD) avec Momentum et weight decay (normalisation L2) relativement classique. Les hyperparamètres choisis sont de **0.9** pour le momentum et **0.0005**. Ces valeurs sont en fait celles proposées à la fin de la section 3 *Implementation details* de l'article introduisant les Wide ResNet[3], et semblent donc un choix raisonnable d'hyperparamètres.
L'application de la normalisation L2 est néanmoins faite de manière intelligente : **elle n'est pas appliquée sur les paramètres appris par les couches de Batch Norm, ni les biais**, tel que conseillé dans cet article de 2017[4].

3.3 Exponential Moving Average

L'article mentionne de manière succinte l'utilisation d'une Moyenne mobile exponentielle (EMA en anglais) sur les paramètres : *Finally, we report final performance using an exponential moving average of model parameters.*
Ce paramètre a cependant une importance considérable sur le score obtenu, comme le montre en partie le test réalisé avec optimizer Adam (qui n'utilisait pas de moyenne mobile).
L'implémentation de cette moyenne mobile se fait de la manière suivante :

- Duplication du modèle complet avant entraînement avec la fonction `deepcopy`. Ce nouveau modèle est appelé *EMA*, et ses poids sont donc décorrélés du modèle entraîné (ils ne sont pas mis à jour par l'optimizer)
- A chaque step, mise à jour des paramètres du modèle EMA à partir du modèle entraîné. La formule de mise à jour est la même que celle du momentum pour le SGD, à savoir :

$$w_{ema} = \beta w_{ema} + (1 - \beta) w_{model}$$

avec w_{ema} : poids du modèle EMA et w_{model} poids du modèle entraîné. Dans l'implémentation optimale, $\beta = 0.999$ (ce qui correspond à une moyenne sur les 1000 dernières valeurs environ, ce qui est cohérent avec un nombre de steps par epoch de 1024. La mise à jour des poids est donc moyennée sur une epoch complète)

- Utilisation du modèle EMA à la fin de chaque epoch pour calculer les performances sur le jeu de test

3.4 Boucle d'entraînement

Contrairement à l'entraînement classique d'un modèle, une epoch ne correspond pas à un passage sur l'ensemble des données. En effet, parler de l'ensemble des données n'a pas de sens dans le contexte de Fixmatch, puisque les 3 jeux utilisés (données labelisées, données faiblement perturbées et données fortement perturbées) n'ont pas la même taille. Le choix d'implémentation des auteurs est de faire **1024** steps d'entraînement par epoch, nombre indépendant de la taille des batchs. Avec une taille de batch de 64 et 250 données labelisées, il faut donc 4 batchs pour faire le tour des données labelisées. En moyenne, les données labelisées sont donc utilisées 256 fois par epoch !
En outre, pour réduire les temps de calcul, les données extraites des 3 dataloaders sont concaténées et passées en parallèe au modèle pendant l'entraînement. L'implémentation officielle utilise une fonction `interleave` à cette étape. Son intérêt m'a échappé (reshape dans une dimension de sortie égale à celle d'entrée), et l'entraînement s'est très bien déroulé sans...

Sur la sélection des images labelisées

Le choix de labels est quelque peu artificiel dans notre contexte (impossible de le faire dans une situation réelle). En supposant avoir une certaine connaissance du jeu de données, il semble possible de faire un choix raisonnable de labels. Les auteurs de l'article se sont inspirés d'un autre article de Google écrit par l'un des contributeurs de Fixmatch[5] pour tester s'il était possible d'apprendre à partir d'une seule image. Cet article liste 5 metrics permettant de caractériser un exemple "prototypique" d'un jeu de données. Ils ont appliqué ces metrics à différents jeux de données, et affichent les résultats aux pages 23 à 26 pour CIFAR10 (images les plus prototypiques / les plus outliers par classe). Les auteurs de Fixmatch arrivent à obtenir un score de **84%** en utilisant une image "meilleur prototype" pour chaque classe.
Une idée de sélection d'images pour CIFAR10 (probablement assez naïve) serait donc de prendre 12 prototypes parfaits et 12 outliers par classe (ou utiliser un hyperparamètre pour le ratio prototype/outlier) pour que le modèle apprenne une bonne représentation de chaque classe, mais puisse également classifier les outliers. Je n'ai pas trouvé les indices correspondants aux outliers/prototypes décrits dans l'article, ni eu le temps de réimplémenter les 5 metrics décrites pour tester cette approche .... Evidemment, cette approche est spécifique à ce jeu de données, et n'est pas transposable à une situation réelle

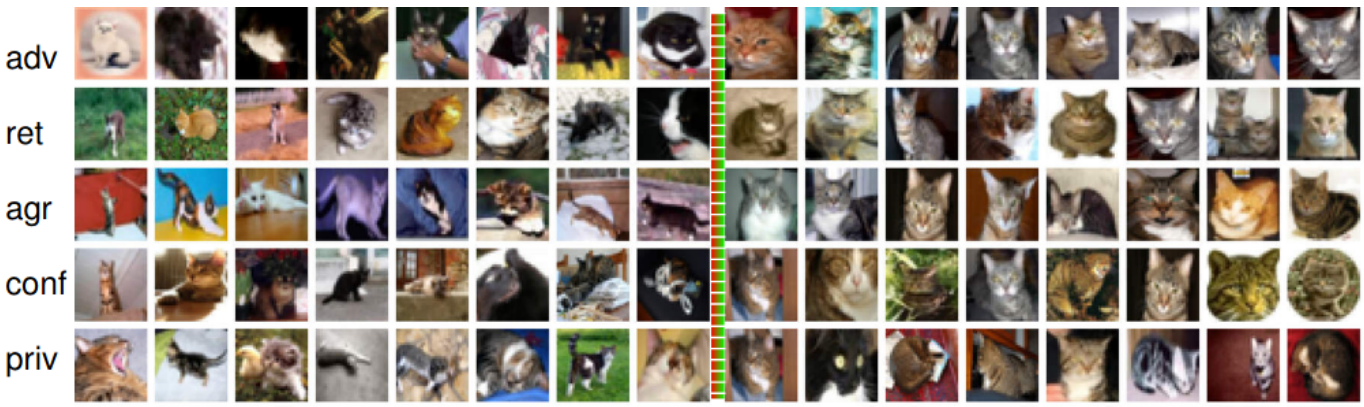

FIGURE 9 – Outliers à gauche, prototypes à droite

Références

[1] Kihyuk Sohn et al. *FixMatch : Simplifying Semi-Supervised Learning with Consistency and Confidence*. 2020. eprint : arXiv:2001.07685.

[2] Ilya Loshchilov et Frank Hutter. *SGDR : Stochastic Gradient Descent with Warm Restarts*. 2016. eprint : arXiv:1608.03983.

[3] Sergey Zagoruyko et Nikos Komodakis. *Wide Residual Networks*. 2016. eprint : arXiv:1605.07146.

[4] Twan van Laarhoven. *L2 Regularization versus Batch and Weight Normalization*. 2017. eprint : arXiv:1706.05350.

[5] Nicholas Carlini, Úlfar Erlingsson et Nicolas Papernot. *Distribution Density, Tails, and Outliers in Machine Learning : Metrics and Applications*. 2019. eprint : arXiv:1910.13427.